Nelson H.F. Beebe

The Mathematical-Function Computation Handbook

Programming Using the MathCW Portable Software Library

Springer

The Mathematical-Function Computation Handbook

Nelson H.F. Beebe

The Mathematical-Function Computation Handbook

Programming Using the MathCW Portable Software Library

Springer

Nelson H.F. Beebe
Department of Mathematics
University of Utah
Salt Lake City, UT
USA

ISBN 978-3-319-64109-6 ISBN 978-3-319-64110-2 (eBook)
DOI 10.1007/978-3-319-64110-2

Library of Congress Control Number: 2017947446

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Dedication

This book and its software are dedicated to three Williams: Cody, Kahan, and Waite. They taught us that floating-point arithmetic is interesting, intricate, and worth doing right, and showed us how to do it better.

This book and its software are also dedicated to the DEC PDP-10 computer, on which outstanding interactive computing and the Internet were built, and on which this author spent a dozen productive years.

Preface

> WRITE YOUR OWN STORY.
> DON'T LET OTHERS WRITE IT FOR YOU.
>
> — CHINESE FORTUNE-COOKIE ADVICE.

> A *preface* IS GENERALLY SHORTER THAN AN *introduction*, WHICH CONTAINS MATTER KINDRED IN SUBJECT, AND ADDITIONAL OR LEADING UP TO WHAT FOLLOWS; WHILE A *preface* IS USUALLY CONFINED TO PARTICULARS RELATING TO THE ORIGIN, HISTORY, SCOPE, OR AIM OF THE WORK TO WHICH IT IS PREFIXED.
>
> — *New Century Dictionary* (1914).

This book documents a large library that supplies the mathematical functions required by several programming languages, including at least these:

- the 1983 ANSI and 1995 and 2012 ISO Standards [Ada83, Ada95, Ada12] for the Ada programming language;
- the 1990, 1999, and 2011 ISO Standards for C [C90, C99, C11b];
- the 1998, 2003, and 2011 ISO Standards for C++ [C++98, BSI03b, C++03a, C++11a];
- the 2002 and 2006 ECMA and 2006 ISO Standards for C#® (pronounced *C-sharp*) [ECM06a, HWG04, CLI05, C#06a, CLI06];
- the 1978 ANSI and 1991, 1997, and 2010 ISO Standards for Fortran [ANSI78, FTN91, FTN97, FTN10];
- the widely used, but not yet standardized, Java® programming language [AG96, AG98, CLK99, AGH00, GJSB00, GJSB05, GJS$^+$13, GJS$^+$14]; and
- the 1990 ISO Extended Pascal Standard [PAS90, JW91].

Numerous scripting languages, including awk, ECMAScript®, hoc, JavaScript®, Julia, Lua®, Perl®, PHP, Python®, Rexx, Ruby, and Tcl, offer a subset of mathematical functions that are usually drawn from the venerable Fortran and C repertoires. Many other current and historical programming languages, among them Algol 60, Algol 68, COBOL, D, Go, Lisp, Modula, Oberon, OCaml, PL/1, Rust, and Scheme, as well as the Adobe® PostScript® page-description language, also provide subsets of the Fortran functions.

Although the needs of those languages are a valuable guide for the design of this library, this author's view has been a wider one, and several additional elementary and special functions are provided, including ones for the detection of integer overflow, a sadly neglected area in most architectures and almost all programming languages.

Most current computers, as well as the virtual machines for C#, Java, and PostScript, are based on the *IEEE 754 Standard for Binary Floating-Point Arithmetic* published in 1985, but developed several years earlier, and implemented in hardware in the Intel 8087 coprocessor in 1980. Software intended to be run only on current systems could therefore limit itself to the IEEE 754 architecture. However, three considerations led to the broader view adopted in this book:

- Decimal floating-point arithmetic is part of the 2008 revision of the IEEE 754 Standard [IEEE08, ISO11], and there are proposals to include it in future versions of the ISO C and C++ Standards. Two main reasons to provide decimal arithmetic are its familiarity to humans, and its widespread use in databases.

  IBM is a strong proponent of decimal arithmetic, and has produced a firmware implementation on its mainframe systems, and added hardware support in the PowerPC version 6 and later chips. This arithmetic is based on more than two decades of experience with software decimal arithmetic, notably in the Rexx programming language [Cow85, Cow90] defined by ANSI Standard X3.274-1996 [REXX96], and the NetRexx language [Cow97].

Symbolic-algebra languages, such as Maple®, Mathematica®, Maxima, MuPAD®, PARI/GP, and REDUCE, provide arbitrary-precision arithmetic. The Maple language uses decimal arithmetic, but the others use binary arithmetic.

- There are excellent, and readily available, virtual machines for several historical architectures, including most early microprocessors, the DEC PDP-10, PDP-11, and VAX architectures, the IBM System/360, and even the modern Intel IA-64.

  A recent book on virtual machines [SN05] notes that they provide a good way to prototype new machine designs, and have important advantages for security when users can be isolated in private virtual machines.

  The great success of the Java programming language has in large part been due to its definition in terms of a virtual machine, called the *Java Virtual Machine (JVM)*® [LY97, LY99, SSB01, LYBB13, Set13, LYBB14], and a standard library that provides a uniform environment for Java programs that is independent of the underlying hardware and operating system.

  Microsoft's related C# language is also defined in terms of a virtual machine that forms the core of the Microsoft .NET Framework, and the associated free-software reimplementations of .NET in the DotGNU Project[1] and the Mono Project.[2]

  Once compilers make programming languages available for the virtual machine, applications, and indeed, even entire operating systems, can be lifted from hardware to virtual machines. This broadens markets for software vendors, removes their dependence on hardware suppliers, and allows them to reach future hardware platforms more quickly.

  The commercial Parallels®, VirtualBox®, Virtual Iron®, VMware®, and Xen® systems have demonstrated that virtual machines adapted to particular hardware can support multiple operating systems on a single CPU with little overhead. Hewlett–Packard, IBM, and Sun Microsystems have all introduced products that allow processors to be logically split and shared among multiple operating systems, or multiple instances of the same operating system. GNU/LINUX *containers* and FREEBSD *jails* provide similar capabilities.

- The Cody and Waite *Software Manual for the Elementary Functions* [CW80] that inspired some of the work described in this book was careful to address the algorithmic issues raised by floating-point designs with number bases other than two. Their book also addresses computations of the elementary functions in *fixed-point* arithmetic, a topic that we largely ignore in this book.

Most books and courses on programming languages spend little time on floating-point arithmetic, so student programmers may consequently conclude that floating-point programming must be trivial, uninteresting, and/or unimportant. This author's view is that floating-point programming does not receive the attention that it deserves, and that there are interesting problems in the design of mathematical software that are not encountered in other types of programming.

However, because a substantial portion of the human population seems to have a fear of mathematics, or at least finds it uncomfortable and unfamiliar, this book takes care to minimize the reader's exposure to mathematics. There are many research papers on the elementary and special functions that provide the mathematical detail that is a necessary foundation for computer implementations, but in most cases, we do not need to deal with it directly. As long as you have some familiarity with programming in at least one common language, and you can recall some of your high-school algebra, and are willing to accept the results of a few short excursions into calculus, you should be able to understand this book, and learn much from it.

On the other hand, professional numerical analysts should find this book of interest as well, because it builds on research by the mathematical software community over the last four decades. In some areas, notably those of converting floating-point numbers to whole numbers, and finding remainders, we present cleaner, simpler, and more portable solutions than are found in existing libraries.

[1] See `http://www.gnu.org/projects/dotgnu/`.

[2] See `http://www.mono-project.com/`.

An important recent research topic has been the development for some of the elementary functions of mathematically proven algorithms that guarantee results that are always correctly rounded. In most cases, these advanced algorithms are quite complex, and beyond the scope of this book. However, for some of the simpler elementary functions, we show how to produce always, or almost always, correctly rounded results.

You might wonder why anyone cares about the correctness of the last few digits of a finite approximation to, say, the square root of 27. Certainly, few humans do. However, it is important to remember that computer hardware, and software libraries, are the foundations of all other software. Numerical computing in particular can be remarkably sensitive to the characteristics and quality of the underlying arithmetic system. By improving the reliability and accuracy of software libraries, we strengthen the foundation. By failing to do so, we leave ourselves open to computational disasters.

Acknowledgements

This author has had the interest and opportunity to have programmed in scores of programming languages on dozens of operating systems, and most of the major CPU architectures, since his first encounter with the IBM 7044 and IBM System/360 model 50 as an undergraduate student.

At the University of Utah Department of Mathematics, we have a large computing facility that serves thousands of students, as well as departmental staff and faculty, and invited guests. We have made it a point to provide, and preserve, access to a wide range of physical and virtual hardware, and operating systems, to encourage and facilitate software portability testing. A powerful tool, `build-all`, developed by this author and presented in another book [RB05a], makes it easy to automate the building and validating of software packages in parallel on multiple systems.

In recent years, guest access to computing facilities in other locations has been invaluable for development and testing on even more platforms. Those organizations include:

- the University of Utah Center for High-Performance Computing (formerly, the Utah Supercomputing Institute);
- the Henry Eyring Center for Theoretical Chemistry in the Department of Chemistry at the University of Utah;
- the University of Utah Department of Electrical Engineering;
- the University of Utah Department of Physics and Astronomy;
- the Emulab Network Emulation Testbed of the Flux Research Group in the School of Computing at the University of Utah; .
- the Hewlett–Packard Test Drive Laboratory;
- the IBM Linux Community Development System; and
- the University of Minnesota Supercomputing Institute.

This author is grateful to the management and staff of all of those excellent facilities, as well as to Intel Corporation for the grant to his department of an IA-64 server.

The mailing list of the IEEE 754 committee, and numerous exchanges with list members, continue to stimulate my interest in floating-point arithmetic. In particular, IBM Fellow Mike Cowlishaw provided useful comments on some of the material on decimal arithmetic in this book, and helped to clarify the history and rationale of decimal arithmetic in computers, and its implementation in modern compilers.

Virtual machines used during the development of this book and its software include:

- the fine commercial VMware system on AMD64, EM64T, and IA-32;
- Ken Harrenstien's outstanding KLH10 implementation of the influential and venerable PDP-10 architecture on which this author worked happily and productively for a dozen years, and the far-ahead-of-its-time DEC TOPS-20 operating system kindly made available to hobbyists by its original vendor;
- Bob Supnik's amazing `SIMH` simulator of about thirty historical architectures, including the Interdata 8/32 (target of the first port of UNIX from the DEC PDP-11), and the DEC VAX;

- Roger Bowler's `Hercules` emulator for the IBM System/370, ESA/390, and z/Architecture; and
- The QEMU (Quick EMUlator) hypervisor and KVM (Kernel-based Virtual Machine) that run on most desktop operating systems, including at least Apple MAC OS X and MACOS, DRAGONFLYBSD, FREEBSD, GHOSTBSD, GNU/LINUX, HAIKU, HARDENEDBSD, MIDNIGHTBSD, Microsoft WINDOWS, NETBSD, OPENBSD, PAC BSD, PCBSD, OPENSOLARIS, REACTOS, and TRUEOS.

This book, and all of the software that it describes, is a single-person production by this author. Years of following the scientific research literature, and published books, in broad areas of computer science, theoretical chemistry, computational physics, and numerical mathematics, and a lifetime spent in programming in scientific computing and other areas, gave the experience needed to tackle a project with the breadth of scope of the software library described here.

The Unix family

The Open Group currently owns the all-caps registered trademark UNIX® for a descendant of the operating system developed at AT&T Bell Laboratories starting about 1969, and permits vendors whose implementations pass an extensive test suite to license use of that trademark. Because of legal wrangling, ownership of the specific name UNIX was long disputed in US courts. That led computer manufacturers to rebrand their customized versions under other trademarked names, and similarly, in the free-software world, each O/S distribution seems to acquire its own unique name. In this book, we use the capitalized name UNIX to refer to the entire family.

The GNU® system, where *GNU* stands for the infinitely recursive phrase *GNU is Not Unix*, is, for the purposes of this book, also a UNIX-like system. If this matters to you, just mentally add the suffix *-like* to every mention of UNIX in this book.

Filesystem implementations, operating-system kernel details, software packaging practices, and system-management procedures and tools, differ, sometimes dramatically so, across different members of the UNIX family. However, for the ordinary *user* of those systems, they are all familiar and similar, because most of their commonly used commands are nearly identical in behavior and name. The notions of files as byte streams, devices, kernel data, and networks treated as files, command shells for interactive use and scripting, simple I/O redirection mechanisms, and pipes for easy connection of many smaller programs into larger, and more powerful, ones, are critical features of the UNIX environment. A common window system, X11, provides mechanism but not policy, and separates program, window display, and window management, allowing each to run on separate computers if desired, but joined by secure encrypted communications channels.

Many UNIX distributions have long been available as open-source software, allowing programmers all over the world to contribute documentation, enhancements, and sometimes, radical new ideas. Internet mailing lists allow them to keep in regular contact and develop long-time electronic friendships, even though they may never have a chance to meet in person. The UNIX world is truly a global community.

Trademarks, copyrights, and property ownership

Because it deals with real computers, operating systems, and programming languages, both historic and current, this book necessarily contains many references to names that are copyrighted, registered, or trademarked, and owned by various firms, foundations, organizations, and people. Except in this preface, we do not clutter the book text with the traditional superscript symbols that mark such ownership, but we acknowledge it here, and we index every reference to model, product, and vendor names. Among the commercial software systems most frequently mentioned in this book are Maple, Mathematica, MATLAB®, and MuPAD. In general, we should expect almost any commercial entity, or commercial product, to have a name that is registered or trademarked. The computing industry that has so changed human history is the work of many companies and many people from many countries, and we should view it proudly as part of our shared modern heritage.

The International Organization for Standardization (ISO) kindly gave permission to cite in this book (usually brief) snippets of portions of the ISO Standards for the C language. Because of the language precision in Standards, it is essential for correct software development to be guided by the original exact wording, rather than working from existing practice, hearsay, imperfect human memory, or paraphrased text.

To show code, or not

Authors of books about software have to make a choice of whether to show actual code in a practical programming language, or only to provide imprecise descriptions in flowcharts or pseudocode.

Although the latter approach seems to be more common, it has the serious drawback that the code cannot be tested by a compiler or a computer, and long experience says that untested code is certain to contain bugs, omissions, and pitfalls. Writing software is hard, writing floating-point software is harder yet, and writing such software to work correctly on a broad range of systems is even more difficult. Software often outlives hardware, so portability is important.

There are many instances in this book where subtle issues arise that *must* be properly handled in software, and the only way to do so is to use a real programming language where the code can undergo extensive testing. The floating-point programmer must have broad experience with many architectures, because obscure architectural assumptions and platform dependencies can all too easily riddle code, making it much less useful than it could be, if more care were paid to its design.

This author has therefore chosen to show actual code in many sections of this book, and to index it thoroughly, but not to show all of the code, or even all of the commentary that is part of it. Indeed, a simple printed listing of the code in a readable type size is several times longer than this book.

When the code is routine, as it is for much of the validation testing and interfacing to other programming languages, there is little point in exhibiting it in detail. All of the code is freely available for inspection, use, and modification by the reader anyway, because it is easily accessible online. However, for many shorter routines, and also for complicated algorithms, it is instructive to display the source code, and describe it in prose.

As in most human activities, programmers learn best by hands-on coding, but they can often learn as much, or more, by reading well-written code produced by others. It is this author's view that students of programming can learn a lot about the subject by working through a book like this, with its coverage of a good portion of the run-time library requirements of one of the most widely used programming languages. This is *real* code intended for *real* work, for portable and reliable operation, and for influencing the future development of programming languages to support more numeric data types, higher precision, more functionality, and increased dependability and security.

To cite references, or not

Although research articles are generally expected to contain copious citations of earlier work, textbooks below the level of graduate research may be devoid of references. In this book, we take an intermediate approach: citations, when given, are a guide to further study that the reader may find helpful and interesting. When the citations are to research articles or reports, the bibliography entries contain Web addresses for documents that could be located in online archives at the time of writing this book.

Until the advent of computers, tracking research publications was difficult, tedious, and time consuming. Access to new journal issues often required physical visits to libraries. Photocopiers made it easier to abandon the long-time practice of sending postcards to authors with requests for article reprints. The world-wide Internet has changed that, and many journals, and some books, now appear only electronically. Internet search engines often make it possible to find material of interest, but the search results still require a lot of labor to record and re-use, and also to validate, because in practice, search results are frequently incomplete and unreliable. Most journal publishers have Web sites that provide contents information for journal volumes, but there is no common presentation format. Also, a search of one publisher's site is unlikely to find related works in journals produced by its competitors.

The BIBTEX bibliography markup system developed by Oren Patashnik at Stanford University as part of the decade-long TEX Project has provided an important solution to the problem of making publication information *reusable*. This author has expended considerable effort in writing software to convert library and publisher data into BIBTEX form, and developed many tools for checking, curating, ordering, searching, sorting, and validating BIBTEX data.

Two freely available collections, the *BibNet Project* archives and the *TEX User Group* archives, both hosted at the author's university, and mirrored to other sites, provide a view into the research literature of selected areas of chemistry, computer science, computer standards, cryptography, mathematics, physics, probability and statistics, publication metrics, typesetting, and their associated histories.

Those archives provide a substantial pool of data from which specialized collections, such as the bibliography for this book, can be relatively easily derived, *without* retyping publication data. BIBTEX supports citation, cross referencing, extraction, sorting, and formatting of publication data in hundreds of styles. Additional software written by this author extends BIBTEX by automatically producing the separate author/editor index that appears in the back matter of this book, and enhancing bibliography entries with lists of page numbers where each is cited. Thus, a reader who remembers *just one author* of a cited work can quickly find both the reference, and the locations in the book where it is cited.

The MathCW Web site

This book is accompanied by a Web site maintained by this author at

```
http://www.math.utah.edu/pub/mathcw/
```

That site contains

- source code for the book's software;
- a BIBTEX database, `mathcw.bib`, from a subset of which all references that appear in the bibliography in this book's back matter are automatically derived and formatted;
- related materials developed after the book has been frozen for publication;
- compiled libraries for numerous systems; and
- pre-built C compilers with support for decimal arithmetic on several operating systems.

It is expected to be mirrored to many other sites around the world, to ensure wide availability, and protection against loss.

The mathcw software is released in versioned bundles, and its history is maintained in a revision control system to preserve a record of its development and future evolution, as is common with most large modern software projects.

Contents

List of figures

List of tables

Quick start

THIS QUICK START FEATURE IS NOT COMPREHENSIVE: THERE MAY BE ADDITIONAL OSHA STANDARDS AND GUIDANCE MATERIALS THAT ALSO APPLY.

— WEB SITE DISCLAIMER, U.S. DEPARTMENT OF LABOR, OCCUPATIONAL SAFETY & HEALTH ADMINISTRATION.

The mathcw library implements the elementary mathematical functions mandated by C89 and C99, normally supplied by the `-lm` library on most UNIX systems. **Table 3.4** on page 58 summarizes the mathcw library contents.

Much more information is provided in the remaining chapters of this book, but if you are only interested in library installation, then this short section provides enough information for that task.

To build, validate, and install this library on almost any modern UNIX or POSIX-compatible system, this widely used GNU recipe does the job:

```
% ./configure && make all check install
```

On Hewlett–Packard HP-UX on IA-64 (Intel Architecture 64-bit), change the target `all` to `all-hp` to get library support for two additional precisions available on that platform; see **Section 4.26** on page 100 for details. The corresponding `check-hp` target tests all five supported precisions.

To change the default installation tree from the GNU-standard default of `/usr/local`, define `prefix` to an equivalent value at install time:

```
% make prefix=/depot install
```

That would create the library file `/depot/lib/libmcw.a`.

The mathcw library can be used like this with either C or C++ compilers:

```
% cc [ flags ] [ -I$prefix/include ] file(s) [ -L$prefix ] -lmcw
```

The `-I` option is needed if the code includes this package's `mathcw.h` header file. That may not be necessary, if `<math.h>` is included. Depending on the local conventions, the `-L` option may or may not be required to define a load-library path. `$prefix` must be replaced by the local default, such as `/usr/local`.

Caution is needed on systems for which the C header file `<math.h>` redefines library function names, or where compilers produce inline code for elementary functions, such as on the Intel IA-32 (formerly, x86) architecture, which has single hardware instructions for several of them. For the GNU compiler family, use the option `-fno-builtin` to permit library routines to replace inline code.

Caution is also needed when the host default floating-point behavior is not IEEE-754 conformant. GNU/LINUX and OSF/1 operating systems with Alpha processors have this defect: underflows flush abruptly to zero, and overflows and operations with NaN immediately terminate the process. To get mostly conformant behavior, with GNU compilers, use the `-mieee` flag, and with native compilers, use the `-ieee` flag. To get full conformance, use `-mieee-with-inexact` or `-mieee-conformant`, plus `-mfp-rounding-mode=d`, with GNU compilers, and `-ieee_with_inexact` and `-mfp-rounding-mode=d` with native ones.

On IBM AIX 4.2, `long double` is compiled as `double`, unless the native compiler name is changed from `cc` to `cc128`, or the `-qlongdouble` option is specified. Run-time library support of 128-bit `long double` requires linking with `cc128`, or explicit specification of the `-lc128` library. The easiest way to build the mathcw library is then like this:

```
% make CC=cc128 all check
```

The GNU compiler `gcc` version 2.95.3 does not support a 128-bit `long double` at all, and no newer version builds successfully on AIX 4.2.

It is possible to compile a subset of the package, either manually to select replacements for deficient implementations in the native `-lm` library, or by precision, with the targets `float`, `double`, and `longdouble`. For example, pre-C99 implementations of the C language may lack some or all of the `float` and `long double` elementary functions, providing only the `double` versions.

To clean up after a build, do this:

```
% make clean
```

To return to the original state of a freshly unpacked distribution, use the command

```
% make distclean
```

More details about the interpretation of the test results are given later in **Chapter 22** on page 763.

The `mathcw` library distribution includes interfaces that make the library accessible from several other major programming languages. The interfaces are described in appendices of this book, beginning on page 911.

For up-to-date information about building the library, run the command

```
% make help
```

to get further information about platform-specific targets, and about building shared libraries.

1 Introduction

> MANY PITFALLS AWAIT A SYSTEMS PROGRAMMER WHO ATTEMPTS TO IMPLEMENT BASIC FUNCTION ROUTINES USING INFORMATION GLEANED FROM A CALCULUS TEXT.
>
> — W. J. CODY, JR. AND W. WAITE (1980).

The mathcw package contains source code files that provide replacements for C's elementary functions. It exists to provide improved code for use by hoc (and other software) on platforms where the native library versions are deficient, to provide access to the full C99 math library at a time when few systems have it, and to provide an insulating layer to shield other software from system dependencies. When it is important that software produce (nearly) identical results on all platforms, avoiding the use of different underlying math libraries in favor of a single consistent library can be of significant help.

Two landmark books [HCL$^+$68, CW80] are good sources of decent algorithms for those functions. Those books predate IEEE 754 arithmetic [IEEE85a, IEEE08, ISO11] and that of its radix-independent extension, IEEE 854 arithmetic [ANS87], so modifications are required to make their algorithms correctly handle Infinity, NaN, and signed zero (see **Section 4.7** on page 69), and to extend them to handle the higher precisions of IEEE 754 arithmetic.

In the mathcw package, emphasis is on portability, precisely because the code must be able to be used on a wide variety of platforms. In particular, the code does *not* require that IEEE 754 arithmetic be available, but it does assume that it *might* be. Thus, it tests for Infinity and NaN as needed using `isinf()`, `isnan()`, and their companions for other precisions. It also uses `nextafter()` and its companions. Versions of those functions are provided for systems that lack them.

This book is intended to be a successor to that of Cody and Waite [CW80]. Their book supplies recipes for computations of all of the elementary functions required by the Fortran programming language, which was the most widely used, and most portable, language for numerical computation up through the mid 1980s. However, by the 1990s, the common desktop operating systems and most of their utilities, compilers, and libraries, were written in the C language. As a result, programming languages can interoperate more easily than they could a generation ago, and a function library written in C can be relatively easily used by other languages as well, possibly via a thin interface layer to provide alternate function names and/or calling sequences.

The functions in the C99 mathematical library are a superset of those required by most other programming languages, so C is clearly the best language choice for a universal mathematical library at the time of writing this. Although the C language certainly has plenty of pitfalls for the unwary, it also provides sufficient access to the hardware that escapes to other languages, and notably, platform-specific assembly code, are rare. With care, C programs can be written with a high degree of portability between operating systems and CPU architectures, and can be compiled with both C and C++ compilers. Those practices expose C code to a much wider range of compilation environments than is possible with any other programming language. Combining that broad exposure with rigorous run-time testing can give the developer much more confidence in the correctness and reliability of the code than is possible with languages that are restricted to a single compiler, a single operating system, or a single hardware platform.

C and C++ are each specified by at least three international standards [C90, C99, C11b, C++98, C++03a, C++11a, C++14] that precisely define the behavior of those languages and their standard libraries, and provide rigorous grammars for the languages. As a result, programs in those languages are much more likely to preserve their meaning in different environments. Few other programming languages are as carefully specified as C and C++ are, and even fewer have been subjected to the wide implementation, scrutiny, use, and testing that C and C++ have experienced. The roots of C go back to the late 1960s, and the patina of more than four decades of use has made it one of the best understood and most reliable languages in the programmer's toolbox.

This book deals almost exclusively with the C language for the implementation of the mathcw library. However, interfaces for several other languages are presented in appendices:

- Ada in **Appendix A** on page 911;

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_1

- C# in **Appendix B** on page 917;
- C++ in **Appendix C** on page 923;
- Fortran in **Appendix F** on page 941;
- Java in **Appendix J** on page 979; and
- Pascal in **Appendix P** on page 989.

In each case, it is expected that the mathcw library implementation will remain in C, without the need for rewriting, porting, testing, and maintenance, in multiple programming languages.

Although newer programming languages extol the virtues of *classes*, *interfaces*, *modules*, *namespaces*, and *packages*, for the mathcw library, none of those fancy features is needed, and their use would only clutter the code and reduce its portability and audience. The C preprocessor, the C typedef statement, the C standard header files, and a small set of function primitives, together provide minimal, but sufficient, tools for the major abstractions required in the library to provide independence from *byte order*, *floating-point and integer data formats*, *floating-point architectures*, *hardware architectures*, *operating systems*, and *numerical precision and range*. In addition, the software design of the mathcw library makes it easy to replace the algorithmic internals of any function without altering the user interface.

No use whatever has been made of graphical user interface (GUI) development environments during the preparation of either the mathcw library or this book. Instead, a superb text editor, emacs, a powerful typesetting system, LaTeX, and access to several dozen C and C++ compilers, and also compilers for Ada, C#, Fortran, Java, and Pascal, on more than twenty different operating systems, have been quite sufficient.

Besides the language-interface appendices, there are additional ones that provide background reading and other information:

- decimal floating-point arithmetic in **Appendix D** on page 927;
- errata in the Cody and Waite book in **Appendix E** on page 939;
- historical floating-point architectures in **Appendix H** on page 947;
- integer arithmetic in **Appendix I** on page 969.

1.1 Programming conventions

In order to guarantee portability, even to older systems, and algorithmic clarity, there are several *required* programming practices for the library itself (*not* for code that calls it):

- Functions shall be declared and prototyped in C89 style. The separate ansi2knr program included with the distribution can be used to convert the source code to old-style Kernighan-and-Ritchie C for systems with old compilers.
- The C header file <math.h> *cannot* be included in the implementations, because it might redefine functions, or have conflicting prototypes. Instead, include mathcw.h.
- The compile-time macro P defines the desired polynomial precision in bits. It is used to select different algorithms or different polynomial approximations. Its default value is the IEEE 754 significand precision, defined by the macro T, that corresponds to the precision of the function.
- Each function must be provided in a separate file, so that individual functions can be replaced as needed.
- To the greatest extent possible, each function should have its own separate implementation, depending only on a small set of low-level primitives used by most of the library. Even when functions are related, independent implementations are helpful in limiting the effects of inaccuracies or bugs.
- There shall be only a single point of return from any function, as the last executable statement in the body.
- The goto statement is forbidden.

- Filenames must not exceed the $6 + 3$ limit of older operating systems, with at most a single dot (`filnam.ext`). The only permitted exceptions to that rule are the file `Makefile`, the UNIX manual-page files, which must be named according to the functions that they document (some of those function names are long), the files for decimal floating-point arithmetic, which may require three suffix letters, and the language-interface files, whose names are constrained by other rules.

- Filenames must begin with a letter, followed by letters, digits, and at most one dot.

- Filenames must not collide when remapped to a single lettercase.

- Apart from function names required by the ISO C Standards, external names must be unique in the first six characters without regard to lettercase. For the many long names in the C math library, provision must be made to remap them to unique shorter names on older systems that limit external name lengths and lettercase.

- Numeric constants must be defined to high accuracy, and declared as type `static const fp_t`, instead of the traditional approach of using `#define` preprocessor directives. Named constants are visible in debuggers, whereas preprocessor symbols are hardly ever displayable.

- Floating-point constants with long fractional parts must be provided both in decimal and in C99-style hexadecimal. An exception to that rule is temporarily permitted for the rational polynomial coefficients.

- For each elementary function, there shall be only one algorithm file that is shared by all precisions. Exceptions are made only for low-level primitives that manipulate bits, instead of doing numerical computations, because their code cannot easily be made precision independent.

- Source code files shall be named after the function that they implement: `sqrtf.c`, `sqrt.c`, `sqrtl.c`, and so on. Each such file defines the required precision, and then includes a common header file that provides the implementation of the algorithm. The name of that header file is formed from the name of the double-precision function suffixed with `x.h`, like this: `sqrtx.h`.

  The separation of code and data is an important programming practice that is highlighted in Niklaus Wirth's outstanding and influential book *Algorithms + Data Structures = Programs* [Wir76]. The parser of an early version of MATLAB was based on work described in that book.

- Each algorithm header file shall directly include at most one header file, differing in name only by the absence of the suffix `x` in the base filename. For example, `sqrtx.h` includes only `sqrt.h`, but that file may include other header files.

- Each header file shall protect its body from being included more than once, so that multiple includes of the same file are guaranteed to be safe.

- No algorithm file shall declare function prototypes for any but its own private functions.

- Function prototypes shall not be duplicated, except as needed to support short external names.

- All functions shall be free of I/O, although output statements for analysis and debugging are permitted if they are bracketed in preprocessor conditionals that normally hide them from the compiler.

- It is permissible to assume availability of C89's standard header files `<errno.h>`, `<float.h>` and `<limits.h>`. To support assertions during development, `<assert.h>` may be used. The header file `<stddef.h>` may be used to obtain a definition of `NULL` in those few routines that need to check for null pointers.

- The library code requires a C89-compatible preprocessor. In particular, the preprocessor must recognize `#elif` and `#error` directives, it must inhibit function-like macro expansion of parenthesized function names, and it must support token pasting with the `##` operator. The `lcc` compiler distribution[1] contains a highly portable standalone C89-conformant preprocessor written by Dennis Ritchie, the original designer of the C programming language. It can be used if the local C preprocessor is inadequate.

[1] Available at `ftp://ftp.math.utah.edu/pub/lcc/`.

- The sole user interface to the C99 functions in the mathcw library shall be the single header file mathcw.h. About a dozen other header files are required for access to all of the functions in the library. On UNIX systems, only those header files, and the compiled object library file, libmcw.a, need to be installed. Support for building a shared object library, called libmcw.so on most UNIX systems, is provided via the Makefile target shrlib.
- Dynamic memory management, such as with the C library functions malloc() and free(), is forbidden.
- In those few C99 routines that have pointer arguments for additional returned values, the implementation *shall* check for a non-NULL value before attempting to dereference the pointer.
- Pointer arithmetic is deprecated, except for simple traversal of character strings. Use array notation instead, and check that array indexes are within the array bounds.

There are recipes for decimal floating-point arithmetic in the Cody/Waite book, and all of them have been implemented in the mathcw library. When such arithmetic is supported by the compiler, the library provides a repertoire of mathematical and I/O functions that extends C99, making decimal arithmetic as accessible as binary arithmetic.

1.2 Naming conventions

The C programming language originally had only double-precision versions of elementary function routines, and the names sqrt(), sin(), cos(),... carry no distinguishing prefix like they do in Fortran.

The 1990 ISO C Standard [C90, SAI$^+$90] contains a section on future library directions, noting that suffix letters f and l are reserved for float and long double versions, and the 1999 ISO C Standard [BSI03a, C99] finally includes them. C99 also adds about three dozen new functions to the C mathematical library.

The experience of numerical computation shows that higher precision than even the 80-bit and 128-bit long double formats is sometimes needed, and it is possible that future changes to C might introduce support for them with new floating-point data types. Hewlett–Packard has already done something like that, as described later in **Section 4.26** on page 100, and the mathcw library supports those extensions.

One of the important design goals of this library is to provide a path to such extended precision. Thus, all magic numerical constants are documented so that even more precise specifications can be given in the future, and algorithms contain computational branches selected by compile-time preprocessor symbol tests on the bit precision, T, or the decimal precision, D. Where possible, C99-style hexadecimal constants are provided as well, to ensure that constants are correctly specified to the last bit, with additional trailing bits to determine the final rounded form. Alas, at the time of writing this, few C compilers have reached that language level.

The symbolic-algebra language programs used to compute polynomial approximations to certain functions on limited intervals are part of the library source code, so that even higher-precision results can be obtained in the future. In the meantime, the C code supports computation with data types of up to 75 decimal digits, which is enough to handle a future 256-bit octuple-precision floating-point type (perhaps called REAL*32 in Fortran 77, REAL (SELECTED_REAL_KIND(75)) in Fortran 90, or long long double in C and C++).

Java regrettably supports only float and double floating-point data types at present, and has several other floating-point deficiencies as well [KD98]. A Java class can be written to provide extended floating-point arithmetic, but the lack of operator overloading in the language is a distinct inconvenience for users of such a class.

C# has much the same floating-point deficiencies as Java, but does supply operator overloading.

Libraries with operator overloading are already available for C++ and Fortran 90 in David Bailey's arprec and mpfun90 packages[2] and for Fortran 90 in Lawrie Schonfelder's vpa package.[3] They all provide convenient access to extended precision without the need to wrap every operation in a function call, as was traditionally necessary with multiple-precision arithmetic packages in older programming languages.

Because the computational algorithms are similar, no matter what the precision, routines for each of the three standard precisions are merely two-line wrappers for common code. Thus, sqrtf.c, sqrt.c, and sqrtl.c simply define a single preprocessor symbol to identify the data type, and then include the common generic-precision code in sqrtx.h.

Rudimentary testing of higher-precision approximations at lower precision is possible merely by selecting them with a suitable P value at compile time. For example,

[2] Available at http://crd.lbl.gov/~dhbailey/mpdist.

[3] Available at http://pcwww.liv.ac.uk/~jls/vpa20.htm.

```
make CFLAGS=-DP=200 all check
```

would choose 200-bit approximations (60 decimal digits) for each of the three base precisions.

Complete validation is, of course, not possible until at least one C compiler offering better than 113-bit precision is available, and that will require minor code modifications to support extended data types.

All of the floating-point data types in the mathcw library are parametrized as a generic type `fp_t`, and all floating-point constants are wrapped in a macro, `FP()`, that supplies any required type-suffix letter. Once compiled, the library routines have the normal `float`, `double`, and `long double` types, and for use with C++, have C-style linkage from the `extern "C"` prefix on their prototypes in `mathcw.h`.

1.3 Library contributions and coverage

Although the initial goal of the mathcw library project was to supply implementations of the new functions introduced by C99, it soon became clear that considerably more was needed. Most existing mathematical libraries for the C language are targeted at a single floating-point architecture, and designed for fast execution. As such, they make many internal assumptions about machine architecture, floating-point precision, and data storage formats, tying their algorithms and code to specific platforms. Also, their algorithms are not designed to support any other floating-point precisions or bases. Their code is then impractical to extend with support for other bases or precisions, or to port to other platforms, including historical architectures of interest to this author.

The mathcw library is an entirely new software design that eliminates most of the assumptions made in past libraries, and greatly extends the library repertoire, not only for floating-point arithmetic, but also for integer arithmetic, and input and output. The reader may wish to peak ahead at **Table 3.4** on page 58 through **Table 3.6** on page 60 to get a flavor of some of the offerings. There are numerous other support functions described in this book that are not included in those tables.

The most significant advances made by the mathcw library are these:

- The library has been carefully designed and implemented for consistency by a single software architect, and all of the code is freely available under an open-source license to facilitate its rapid dissemination and use.
- Architecture and programming pitfalls are discussed in depth in this book, and carefully dealt with in the library code.
- There is constant interplay between mathematics and computer programming in this book, and it repeatedly emphasizes the necessity of access to symbolic-algebra systems for mathematical and numerical analysis, and for their offerings of arbitrary-precision arithmetic with wide exponent ranges.
- The computational algorithms are documented at length in this book, and their accuracy is analyzed in detail, and illustrated graphically in error plots. In many cases, new numerical methods have been found that improve library accuracy, extensibility, generality, and portability.
- For many functions, multiple algorithms provide alternate computational routes, and offer independent means of testing accuracy and correctness.
- Separation of algorithms from floating-point precision makes it easy to extend the library code now, and in the future. In particular, *all* functions that are available for the standard C data types `float`, `double`, and `long double` already have companions for a future `long long double` type. On platforms with IEEE 754 floating-point arithmetic, that new octuple-precision format occupies 256 bits (32 bytes), offering a precision equivalent to about 70 decimal digits, and an exponent range of about six decimal digits, as recorded in **Table 4.2** on page 65, and **Table D.1** and **Table D.2** on page 929.
- For almost every function defined for binary floating-point arithmetic, there is a companion for decimal floating-point arithmetic, providing a comfortable programming environment for decimal arithmetic in C and C++. Although most programming practices for binary arithmetic also hold in decimal arithmetic, there are some significant differences that are discussed in **Appendix D** on page 927, and particularly, in **Section D.3** on page 931. The programmer who wishes to enter the new decimal world is strongly advised to learn those differences.

- Decimal arithmetic extends C and C++ to handle the needs of accounting, financial, and tax computations, which have legal requirements that mandate decimal operation, and particular rounding rules. Binary arithmetic is legally unacceptable in those fields.
- The function coverage in the mathcw library goes far beyond that required by ISO programming language standards. In particular, it supplies a major portion of the contents of the famous NBS *Handbook of Mathematical Functions* [AS64, OLBC10], with consistent programming style and naming conventions. It also supplies many additional support functions for floating-point and integer arithmetic.
- A long-time bane of human–computer interaction has been that humans are most comfortable in decimal arithmetic, while computers have been designed to work in binary arithmetic. As a result, the *base-conversion problem* continues to plague humans, and causes surprises for users, and for novice programmers. Switching to decimal arithmetic for most numerical work is the solution.
- Scripting languages play an increasing role in modern software development, yet with few exceptions, they build on binary floating-point arithmetic and the functions supplied by the C library. This author has already demonstrated with three scripting languages that their arithmetic can be converted from binary to decimal with only a few hours of work. He believes that it would be beneficial to do so for all scripting languages, and to use decimal arithmetic of the highest available precision. Indeed, in those languages, binary floating-point arithmetic should probably be banished to the scrap heap of history and mistaken practices.
- Compilers for almost any programming language can likely be extended to fully support decimal floating-point arithmetic with just a few days of work. The years of work that it took to develop the mathcw library can be leveraged by all of them.
- The potential of rapid and wide access to decimal arithmetic should encourage numerical analysts to develop new algorithms that address the needs of both binary and decimal arithmetic.
- **Chapter 15** on page 441 on complex arithmetic, and **Chapter 17** on page 475 on complex functions, discuss the serious deficiencies of support for complex arithmetic on all computers. Much more numerical analysis research is needed, and software implementations of complex arithmetic need immediate attention and improvement.
- Exact argument reduction (see **Chapter 9** on page 243) for trigonometric functions eliminates a widespread problem in existing libraries, and repairs a substantial deficiency in the accuracy of complex functions.

1.4 Summary

The writing of this chapter summary was deliberately delayed until this book, and all of the underlying software, was completed. The design rules of earlier sections have proven their worth many times during the creation of the book's prose and its software. The library is fully operational on all of the systems in this author's test laboratory, and elsewhere, providing support for up to six binary, and four decimal, floating-point formats. Apart from suitable selections of a compiler, library targets, and platform, the library builds without problems, and without further configuration, on all supported systems, and *never* requires site-specific source-code changes. The software has been subjected to many different compilers, and at least four independent source-code analyzers, to ruthlessly root out, and remove, coding errors and portability problems.

The influence of Donald Knuth's practice of *literate programming* is seen many times in this book, although we have not used any special software tool for that purpose. The most important lesson of that programming style is that explaining your software to another, even a teddy bear, as Brian Kernighan and Rob Pike write in their lovely book, *The Practice of Programming* [KP99], is enormously helpful in discovering bugs, design flaws, limitations, and oversights.

This book, and its software, has provided a lot of work, and great pleasure, to its author, who hopes that you, the reader, can enjoy it as much as he, the writer, experienced in creating it. There are many lessons of mathematics, numerical algorithms, computer hardware, and computer software, and their intermingled history, that are recorded in this book for readers to learn without the time and struggles of real life. It is now your turn to go out and make algorithms and software even better.

2 Iterative solutions and other tools

WHAT NEEDS THIS ITERATION, WOMAN?

— SHAKESPEARE'S *Othello* (1604).

Throughout much of this book, we need only a moderate amount of mathematics, mostly at the level that you learned in high school, with an occasional excursion into first-year college calculus. If you find calculus unfamiliar, or have simply forgotten most of what you learned about it, rest assured that the intent of this book is to use just enough mathematics to understand how to solve the problems that we address. Theorems and proofs and clever tricks in mathematics have their place, but we do not need them in this book.

Most of the functions that we deal with are venerable ones that have received extensive study, and their properties that lead to effective computational algorithms are tabulated in mathematical handbooks, or can be recovered relatively easily with expert systems known as symbolic-algebra packages. We shall see that the more difficult problem is to turn well-known mathematical recipes into computationally stable, fast, and reliable algorithms that can be implemented in computer programs adapted to the limitations of the *finite precision* and *limited range* of floating-point arithmetic.

In this chapter, we review series expansions of functions, and show how they can lead to iterative solutions of equations that often cannot be solved explicitly. Moreover, with a good starting guess for the solution, we shall see that convergence to an accurate answer can be astonishingly fast.

2.1 Polynomials and Taylor series

The computation of some of the elementary functions can sometimes be reduced to finding a root, y, of an equation $f(y) = 0$. For example, suppose that we want to compute the exponential function, $y = \exp(x)$, for a given x, and that we have an accurate implementation of the logarithm. Take the logarithm of each side to get $\ln(y) = x$. Then the y that we seek is the solution of $f(y) = \ln(y) - x = 0$.

If $f(y)$ is a polynomial of degree less than five, then explicit formulas are known for its roots, and you can probably recall from grade-school arithmetic what the roots of linear and quadratic equations are. If you wonder why *five* is significant, see *The Equation That Couldn't Be Solved* [Liv05] for the interesting story of how mathematicians searched for hundreds of years for general formulas for the roots of polynomials, only to have a brilliant young Norwegian mathematician, Niels Henrik Abel,[1] prove in 1826 that no formulas using only addition, subtraction, multiplication, division, and extraction of roots exist for arbitrary polynomials of degree five or higher. An English translation of Abel's paper is available in a collection of mathematical classics [Cal95, pages 539–541].

In most cases of interest for the computation of the elementary functions, $f(y)$ is *not* a polynomial, and the only practical way to find a root is to make a sequence of intelligent guesses, in the hope of converging to that root. Systematic procedures for finding the root can be derived by expanding the function in a Taylor[2] series (1715), truncating the series to a manageable size, and then solving the now-polynomial equation for a root. We examine that approach further in the next few sections.

The Taylor series of a function $f(y)$ about a point h is given by

$$\begin{aligned} f(y) &= \sum_{n=0}^{\infty} \frac{f^{(n)}(h)}{n!}(y-h)^n \\ &= f(h) + f'(h)(y-h) + f''(h)(y-h)^2/2! + f'''(h)(y-h)^3/3! + \cdots, \end{aligned}$$

[1] Niels Henrik Abel (1802–1829) also contributed to the theory of functions, and his name gave the adjective *abelian* used in some areas of mathematics. There is a crater on the Moon named after him.

[2] Brook Taylor (1685–1731) was an English mathematician who is credited with the *Taylor series* and *Taylor's theorem*. He also worked on the calculus of finite differences and optical refraction.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_2

where $f^{(n)}(h)$ is the n-th derivative (see **Section 4.1** on page 61) evaluated at $y = h$. The factorial notation, $n!$, means $n \times (n-1) \times (n-2) \cdots \times 2 \times 1$, with the convention that $0! = (-1)! = 1$. The capital Greek letter *sigma*, Σ, is the summation operator, and is often adorned with lower and upper index limits, written below and above the operator when there is sufficient space, and otherwise, written as subscripts and superscripts. Primes indicate the order of the first few derivatives, and ∞ is the symbol for infinity.

As long as $y - h$ is small and less than one in magnitude, and provided that the derivatives do not grow excessively, the decreasing values of powers of small numbers in the numerators, and the factorially increasing denominators, ensure that the Taylor series converges rapidly. When y is close to h, summing the first few terms of the series is often a computationally effective route to finding the function value. Otherwise, we can sometimes replace the series by a few leading terms, plus a correction term that represents the remainder of the sum, but is easier to compute.

Calculus textbooks discuss the conditions on $f(y)$ for which the Taylor series exists, but all that we need in this book is to know that almost all of the functions that we show how to compute *do* have a well-behaved series. General methods for determining whether infinite series converge (sum to a finite value) or diverge (sum grows without limit) are discussed at length in *Mathematical Methods for Physicists* [AW05, Chapter 5] and [AWH13, Chapter 1].

2.2 First-order Taylor series approximation

As a first approximation, truncate the Taylor series to the first two terms, and solve for the root:

$$\begin{aligned} f(y) &= 0 \\ &\approx f(h) + f'(h)(y-h), \\ y &\approx h - f(h)/f'(h). \end{aligned}$$

Here, the operator $\approx$ means *approximately equal*. That solution suggests an iterative procedure starting with an initial guess, y_0:

$$\boldsymbol{y_{n+1} = y_n - f(y_n)/f'(y_n).}$$

That formula is so important that we highlight it in bold type. It is known as the *Newton–Raphson method*, from work by Isaac Newton[3] in 1665 and Joseph Raphson in 1690, or sometimes just the *Newton method*. It predates the Taylor series, because it was originally derived by graphical arguments and binomial expansions. Special cases of the iteration for finding square roots and cube roots were known to the Babylonians (ca. 1900 BCE), and independently, in India (ca. 1000 BCE) and China (ca. 100 BCE).

Ypma [Ypm95] gives an interesting historical review of the Newton–Raphson method, and suggests that Newton–Raphson–Simpson is a more accurate name, considering the later contributions of Thomas Simpson (1710–1761).

With a sufficiently good starting guess, Newton–Raphson convergence is *quadratic*, doubling the number of correct digits each iteration. Importantly, the iterations are *self correcting*: computational errors made in one iteration do not carry over to the next iteration, because that iteration is just a fresh start with an improved guess.

The drawbacks of the Newton–Raphson iteration are that it does not necessarily converge if the initial guess is too far from the root, and that it requires evaluation of both the function and its first derivative, which might be computationally expensive, or even impossible.

If the first derivative is replaced by an approximate numerical derivative, $f'(y_n) \approx (f(y_n) - f(y_{n-1}))/(y_n - y_{n-1})$, the iteration is called the *secant method*, and its convergence is less than quadratic: the error decreases by powers of the *golden ratio*, $\phi = (1 + \sqrt{5})/2 \approx 1.618$. Here, ϕ is the Greek letter *phi* that is commonly used for that ratio. The story of that remarkable number, which turns up in many different areas of mathematics, is told in *A Mathematical History of Golden Number* [HF98] and *The Golden Ratio* [Liv02].

[3] The English scientist Isaac Newton (1643–1727) of Cambridge University is one of the most famous in history, with important contributions to astronomy, mathematics, optics, philosophy, and physics. He and the German polymath Gottfried Leibniz (1646–1716) are jointly credited with the development of calculus; see *The Calculus Wars* [Bar06] for the interesting history of that work. Newton's 1687 masterpiece book *Philosophiae Naturalis Principia Mathematica* described gravitation and the laws of motion of classical mechanics. He invented the reflecting telescope, which gave a three-fold increase in magnification. He was President of the Royal Society for more than two decades, and in his last 30 years, Newton was Warden and Master of the Royal Mint, charged with improving coinage and eliminating counterfeiting [Lev09]. Leibniz invented the binary number system, and about 1671, designed the first mechanical calculator that could add, subtract, multiply, and divide.

Despite the drawbacks, Newton–Raphson iteration is widely used in numerical computations, and we refer to it often in this book. As long as we have an accurate starting value, the first-order iteration is an effective computational technique for many of the functions that we treat. Nevertheless, it can sometimes be useful to consider more complicated algorithms that we derive in the next two sections.

2.3 Second-order Taylor series approximation

The next approximation for finding a solution of $f(y) = 0$ is to include one more term of the Taylor series, and then regroup to form a *quadratic equation*:

$$\begin{aligned} f(y) &= 0 \\ &\approx f(h) + f'(h)(y-h) + f''(h)(y-h)^2/2 \\ &\approx Ay^2 + By + C. \end{aligned}$$

The coefficients are given by these formulas:

$$\begin{aligned} A &= f''(h)/2, \\ B &= f'(h) - f''(h)h, \\ C &= f(h) - f'(h)h + (f''(h)/2)h^2. \end{aligned}$$

The solution of the quadratic equation is the desired value, y. Developing a computer program for the robust determination of the roots of such equations from schoolbook formulas is more difficult than would appear, however, so we delay treatment of that problem until **Section 16.1** on page 465.

When the second derivative is zero, we have $A = 0$, and $y = -C/B = h - f(h)/f'(h)$, which is just the Newton–Raphson formula, as it must be, because it corresponds to dropping the third and higher terms of the Taylor series.

2.4 Another second-order Taylor series approximation

Another way to handle the three-term truncated Taylor series is to partly solve for y:

$$\begin{aligned} 0 &\approx f(h) + f'(h)(y-h) + f''(h)(y-h)^2/2! \\ &\approx f(h) + (y-h)[f'(h) + f''(h)(y-h)/2], \\ y &\approx h - f(h)/[f'(h) + f''(h)(y-h)/2]. \end{aligned}$$

Although that form does not immediately seem helpful, replacing y on the right-hand side by the value predicted by the Newton–Raphson iteration, $h - f(h)/f'(h)$, produces this result:

$$\begin{aligned} y &\approx h - f(h)/[f'(h) + f''(h)((h - f(h)/f'(h)) - h)/2] \\ &\approx h - f(h)/[f'(h) - f''(h)(f(h)/f'(h))/2] \\ &\approx h - 2f(h)f'(h)/[2(f'(h))^2 - f(h)f''(h)]. \end{aligned}$$

That suggests an iterative scheme that we highlight like this:

$$\boldsymbol{y_{n+1} = y_n - 2f(y_n)f'(y_n)/[2(f'(y_n))^2 - f(y_n)f''(y_n)]}$$

That is called *Halley's method*, after Edmund Halley (1656–1742), whose name is also attached to a famous comet that orbits our Sun with a period of approximately 76 years, and was last close to the Earth in 1986. When the second derivative is zero, the method reduces to the Newton–Raphson iteration. We show an application of Halley's method in **Section 8.3** on page 227.

2.5 Convergence of second-order methods

With suitable starting guesses, the quadratic iteration and Halley's method are *cubicly convergent*, tripling the number of correct digits each iteration. If we start with two correct digits, then Newton–Raphson iteration gives 4, 8, 16, 32, … digits, whereas the other two methods give 6, 18, 54, 108, … digits. In typical double-precision computation on modern computers, we have about 16 digits, so the difference is just one iteration.

Halley's method is cheaper to compute, and is therefore usually to be preferred over the quadratic iteration.

The cubicly convergent methods require both a second derivative, and extra work in each iteration. Also, that additional computation increases rounding error, and has two opportunities for significance loss from subtraction. When the target precision and the accuracy of the initial guess are known in advance, careful analysis, and timing measurements of computer programs on multiple hardware platforms, are needed to determine which of those three alternatives is computationally most efficient.

The best way to improve the speed of all of those iterative methods is to start with a better guess for y_0, such as with a polynomial approximation, which is the subject of **Chapter 3**.

2.6 Taylor series for elementary functions

For arbitrary-precision computation of elementary functions of the form $f(x)$, the best approach is often just to sum the Taylor series in a region near $x = h$ where the series converges rapidly, and usually, we have $h = 0$.

Although mathematical handbooks tabulate the Taylor series for all of the important elementary and special functions, we can more easily use a one-line Maple program to display the first few terms in the Taylor series of each of the elementary functions handled in our library and treated by Cody and Waite:

```
% maple
    |\^/|     Maple 8 (SUN SPARC SOLARIS)
._|\|   |/|_. Copyright (c) 2002 by Waterloo Maple Inc.
 \  MAPLE  /  All rights reserved. Maple is a registered trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> for f in [ sqrt, exp, log, sin, cos, tan, cot, arcsin, arccos,
            arctan, sinh, cosh, tanh ] do
      printf("Taylor series of %s:\n", f);
      traperror(taylor(f(x), x = 0, 9));
  end do;
Taylor series of sqrt:
              "does not have a taylor expansion, try series()"


Taylor series of exp:
             2        3         4          5          6           7
1 + x + 1/2 x  + 1/6 x  + 1/24 x  + 1/120 x  + 1/720 x  + 1/5040 x  +

             8      9
    1/40320 x  + O(x )


Taylor series of log:
              "does not have a taylor expansion, try series()"


Taylor series of sin:
                          3          5           7      9
                 x - 1/6 x  + 1/120 x  - 1/5040 x  + O(x )


Taylor series of cos:
                     2         4          6            8      9
            1 - 1/2 x  + 1/24 x  - 1/720 x  + 1/40320 x  + O(x )
```

```
Taylor series of tan:
                            3         5   17   7      9
                   x + 1/3 x  + 2/15 x  + --- x  + O(x )
                                          315


Taylor series of cot:
              "does not have a taylor expansion, try series()"


Taylor series of arcsin:
                           3         5          7      9
                  x + 1/6 x  + 3/40 x  + 5/112 x  + O(x )


Taylor series of arccos:
                Pi             3         5          7      9
               ---- - x - 1/6 x  - 3/40 x  - 5/112 x  + O(x )
                2


Taylor series of arctan:
                             3        5        7      9
                    x - 1/3 x  + 1/5 x  - 1/7 x  + O(x )


Taylor series of sinh:
                          3          5           7      9
                 x + 1/6 x  + 1/120 x  + 1/5040 x  + O(x )


Taylor series of cosh:
                     2         4          6            8      9
            1 + 1/2 x  + 1/24 x  + 1/720 x  + 1/40320 x  + O(x )


Taylor series of tanh:
                            3         5   17   7      9
                   x - 1/3 x  + 2/15 x  - --- x  + O(x )
                                          315
```

In those expansions, the *big-oh* notation, $\mathcal{O}(x^n)$, is read *order of* x^n. It means that the magnitude of the first omitted term is bounded by $|cx^n|$, where c is a constant, x is arbitrary, and the vertical bars mean absolute value.

We have already suggested that the square-root function can be computed from the Newton–Raphson iteration, so the fact that Maple is unable to report its Taylor series does not matter. Because Cody and Waite effectively compute $\cot(x)$ from $1/\tan(x)$, Maple's inability to find a Taylor series for $\cot(x)$ does not matter either. Maple can still provide a series for it, like this:

```
> Order := 9:
> series(cot(x), x = 0);
              -1                 3          5           7      9
             x   - 1/3 x - 1/45 x  - 2/945 x  - 1/4725 x  + O(x )
```

Thus, we can move the reciprocal term to the left, and get a Taylor series like this:

```
> taylor(cot(x) - 1/x, x = 0, 9);
                                3          5           7      9
                - 1/3 x - 1/45 x  - 2/945 x  - 1/4725 x  + O(x )
```

The logarithm goes to negative infinity as $x \to +0$, so it is not surprising that there is no Taylor series there. Instead, we just ask for the series near $x = 1$:

```
> taylor(log(x), x = 1, 9);
                   2              3              4              5
x - 1 - 1/2 (x - 1)  + 1/3 (x - 1)  - 1/4 (x - 1)  + 1/5 (x - 1)  -

               6              7              8            9
    1/6 (x - 1)  + 1/7 (x - 1)  - 1/8 (x - 1)  + O((x - 1) )
```

Notice that `exp()` and `log()` are the only ones with both even and odd powers in the Taylor series. All of the others have only odd terms, or only even terms. Cody and Waite exploit that by rearranging the series to get the first one or two terms exactly, and then using a low-order polynomial approximation to represent the remaining infinite series with accuracy sufficient for the machine precision.

2.7 Continued fractions

> THE THEORY OF CONTINUED FRACTIONS DOES NOT RECEIVE THE ATTENTION THAT IT DESERVES.
>
> — ANDREW M. ROCKETT AND PETER SZÜSZ (1992)

Mathematical handbooks often tabulate function expansions known as *continued fractions*. They take the recursive form $F(x) = \text{constant} + a/(b+c)$, where c itself is a fraction $d/(e+f)$, and so on. At least one of the terms in each numerator or denominator contains x. Such a fraction can be written out like this:

$$F(x) = b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4 + \cfrac{a_5}{b_5 + \cfrac{a_6}{b_6 + \cfrac{a_7}{b_7 + \cfrac{a_8}{b_8 + \cdots}}}}}}}}$$

Because that display takes a lot of space, a more compact notation is common:

$$F(x) = b_0 + \frac{a_1}{b_1 +}\,\frac{a_2}{b_2 +}\,\frac{a_3}{b_3 +}\,\frac{a_4}{b_4 +}\,\frac{a_5}{b_5 +}\,\frac{a_6}{b_6 +}\,\frac{a_7}{b_7 +}\,\frac{a_8}{b_8 +}\cdots.$$

It is not immediately evident how to evaluate a continued fraction, because each denominator involves another denominator whose value we do not yet know. There are two common ways to do the evaluation:

- In *backward evaluation*, we start at term n and arbitrarily set $a_{n+1} = 0$, cutting off the infinite fraction. The resulting finite fraction, denoted $F_n(x)$, is called the n-th *convergent* of the infinite continued fraction.

  In practice, we are interested in a particular range of x, and numerical experiments over that range, and with increasing values of n, let us determine how many terms are needed to achieve the desired accuracy. Aside from the evaluation of each of the terms a_k and b_k ($k = 1, 2, \ldots, n$), the computation requires n adds and n divides.

- From the structure of successive convergents, Euler derived in 1737 a recurrence relation that allows *forward evaluation*. It takes the form

$$\begin{aligned}
A_{-1} &= 1, & B_{-1} &= 0,\\
A_0 &= 0, & B_0 &= 1,\\
&\text{for } k = 1, 2, \ldots, n, \text{ compute} &&\\
A_k &= b_k A_{k-1} + a_k A_{k-2}, & B_k &= b_k B_{k-1} + a_k B_{k-2},\\
&F_n(x) = b_0 + A_n/B_n. &&
\end{aligned}$$

Apart from the terms a_k and b_k, the computation requires $2n+1$ adds and $4n$ multiplies, but just *one* divide.

It sometimes happens that either $a_k = 1$ or $b_k = 1$, in which case, the multiplies can be reduced from $4n$ to $2n$.

Arrays of dimension n are *not* required, because only three successive entries of each of A and B are needed at a time. Instead, they can each be stored in three scalars whose values are shuffled down on each iteration.

On most modern machines, division is comparatively expensive, often costing the time for 3 to 15 multiplications. Thus, even though the operation count is smaller for backward evaluation, it might be more expensive than forward evaluation because of the $n-1$ extra divides. However, the code for backward evaluation is simpler to program than that for forward evaluation, and has fewer memory references. On modern systems, memory traffic is costly, with a load or store taking up to 500 times longer than a floating-point copy inside the CPU. If the value is available in the intermediate memory cache, its access time may be much lower, of the order of 3 to 10 times slower than an in-CPU copy. A good optimizing compiler, and a register-rich architecture, may make it possible for many of the scalars to be held in CPU registers,[4] eliminating most of the expensive memory accesses.

With backward evaluation, the number of terms, n, is fixed in advance by numerical experiment, even though for some x values, fewer terms would suffice. With series evaluation, as long as successive terms decrease in magnitude, it is possible to exit the summation loop as soon as the term just added does not change the sum. With forward evaluation of a continued fraction, early loop exit is possible as well, as soon as $\mathrm{fl}(F_n(x)) = \mathrm{fl}(F_{n-1}(x))$ to machine precision. Here, $\mathrm{fl}(\mathit{expr})$ means the floating-point computation of *expr*: it is not a library function, but rather, an indicator of a transition from mathematics to finite-precision arithmetic. Unfortunately, rounding errors in the computed convergents may prevent that equality from ever being satisfied, so a loop limit is still needed, and early exit cannot be guaranteed.

In *exact* arithmetic, both forward and backward evaluation produce identical values of $F_n(x)$. However, error analysis for the floating-point computation of continued fractions [Bla64, JT74] shows that backward evaluation is usually numerically more stable than forward evaluation. Also, even though the convergents may be of modest magnitude, the terms A_k and B_k may become so large, or so small, that they can no longer be represented in the limited exponent range of a floating-point number. It is then necessary to introduce intermediate rescaling of A_k and B_k, complicating the computation significantly. Alternate forms of forward evaluation exist that eliminate the need for scaling, but they increase the work in each iteration [GST07, pages 181–185]. We present them in **Section 2.8** on page 17.

In summary, it is usually not possible to predict which of the two evaluation methods is faster, except by making numerical experiments with both, and the results of the experiments depend on the CPU, on the compiler, and on the compiler's optimization options. Error analysis favors backward evaluation, unless the speed penalty is too severe.

Because of those difficulties, continued fractions are often not even mentioned in books on numerical analysis, and they are not commonly used in computer algorithms for the evaluation of elementary functions. Nevertheless, they can lead to independent implementations of elementary functions that are of use in software testing, because agreement of function values computed by completely different algorithms makes software blunders unlikely. In addition, continued fractions have three advantages over conventional series representations:

- The domain of convergence (that is, the allowable argument range) of a continued fraction of an elementary function may be much larger than that for its Taylor series.

- Fewer terms may be needed to achieve a given level of accuracy with continued fractions than with Taylor series.

- Little code is needed to compute a continued fraction, especially by backward evaluation, making it easier to get it right on the first try.

The special case where the numerators of the continued fraction are all equal to one, $a_k = 1$, is called a *simple continued fraction*, and is sometimes given a separate notation as a bracketed list of coefficients:

$$\begin{aligned} F(x) &= b_0 + \frac{1}{b_1+}\,\frac{1}{b_2+}\,\frac{1}{b_3+}\,\frac{1}{b_4+}\,\frac{1}{b_5+}\,\frac{1}{b_6+}\,\frac{1}{b_7+}\,\frac{1}{b_8+}\cdots, \\ &= [b_0; b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, \dots]. \end{aligned}$$

[4] A register is a storage area inside the CPU that provides much faster data access than external memory does. There are usually separate register sets for integer and floating-point arithmetic, and sometimes additional sets for status flags. Most CPU designs since the mid-1980s have 16 to 128 floating-point registers, sometimes with particular registers defined to be read-only, and hardwired to hold the commonly needed constants 0.0 and 1.0.

Some books, and the Maple symbolic-algebra system, use a comma instead of a semicolon after the first list entry.

All rational numbers can be uniquely written as *finite* simple continued fractions with integer coefficients, provided that, if the last coefficient is 1, it is discarded and absorbed into the next-to-last coefficient:

$$[b_0; b_1, b_2, b_3, \dots, b_n, 1] \to [b_0; b_1, b_2, b_3, \dots, b_n + 1].$$

Every *irrational*[5] number has a unique *infinite* simple continued fraction.

Four properties of simple continued fractions are of interest for computer arithmetic:

$$\begin{aligned}
1/F(x) &= \begin{cases} [0; b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, \dots], & \textit{if } b_0 \neq 0, \\ [b_1; b_2, b_3, b_4, b_5, b_6, b_7, b_8, \dots], & \textit{if } b_0 = 0, \end{cases} \\
c + F(x) &= [c + b_0; b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, \dots], \qquad \textit{for constant } c, \\
cF(x) &= [cb_0; b_1/c, cb_2, b_3/c, cb_4, b_5/c, cb_6, b_7/c, cb_8, \dots], \\
-F(x) &= [-b_0; -b_1, -b_2, -b_3, -b_4, -b_5, -b_6, -b_7, -b_8, \dots], \\
&= [-b_0 - 1; 1, b_1 - 1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, \dots].
\end{aligned}$$

That is, forming the reciprocal of a simple continued fraction requires just a left or right shift in the coefficient list, adding a constant affects only the first coefficient, and scaling a simple continued fraction by a constant alternately multiplies and divides the coefficients by that constant. That scaling is particularly easy, and exact, if c is a power of the floating-point base. Negation is a special case of scaling, and can be done either by negating all of the coefficients, or else by altering the first two as indicated, and inserting a 1 between them.

We can illustrate those properties for particular x values in Maple, whose conversion function takes an additional argument that gives the number of coefficients to be reported. Some of these examples use the Greek letter *pi*, ϖ, π, the famous ratio of the circumference of a circle to its diameter [Bar92, Bec93, Bar96, Bla97, BBB00, AH01, EL04b, Nah06], and others involve the *golden ratio* [HF98, Liv02]:

```
% maple
> convert(Pi, confrac, 15);
    [3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1]

> convert(1 / Pi, confrac, 15);
    [0, 3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2]

> convert(2 * Pi, confrac, 15);
    [6, 3, 1, 1, 7, 2, 146, 3, 6, 1, 1, 2, 7, 5, 5]

> convert(-Pi, confrac, 15);
    [-4, 1, 6, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2]

> convert(Pi - 3, confrac, 15);
    [0, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1]

> convert(1 / (Pi - 3), confrac, 15);
    [7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1]

> phi := (1 + sqrt(5)) / 2:     # the celebrated golden ratio

> convert(phi, confrac, 15);
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

[5] An irrational number is a number that cannot be represented as the ratio of two whole numbers. For example, about 500BC, the Greeks proved by geometrical arguments that $\sqrt{2}$ is irrational. We can show that with simple algebra. Assume that $\sqrt{2}$ *is* rational, so we can write $\sqrt{2} = p/q$, where p and q are whole numbers that have no factors in common. That means that at least one of those numbers must be odd, for if they were both even, they would have a factor 2 in common. Square the equation and rearrange to find $2q^2 = p^2$. That means that p^2 is even, and because only even numbers have even squares, p must be even. Thus, we can replace it by $2r$, where r is another integer. We then have $2q^2 = 4r^2$, or $q^2 = 2r^2$, so q^2, and thus, q, must be even. However, having both p and q even contradicts the original assertion, which must therefore be false. Thus, $\sqrt{2}$ *must be* irrational.

```
> convert(1 / phi, confrac, 15);
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

> convert(2 * phi, confrac, 15);
    [3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]

> convert(-phi, confrac, 15);
    [-2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Notice in each case that the first coefficient in the simple continued fraction of x is just the nearest integer that is not greater than x, a function that we treat later in **Section 6.7** on page 136.

The particularly simple continued fraction for the golden ratio is due to its definition as the solution of $\phi = 1 + 1/\phi$: repeated substitutions of ϕ on the right-hand side show that $b_k = 1$ for all $k \geq 0$. Integral powers of the golden ratio have a surprising closed form: $\phi^n = F_n\phi + F_{n-1}$, where the F_n are the famous *Fibonacci numbers* introduced in 1202 CE [Sig02, Bee04b, PL07, Dev11] in the same book that brought the digit zero to the Western World, and introduced the words *algorithm* and *algebra*, derived from the name of a Ninth Century Persian mathematician and that of his book. Continued fractions of odd powers of ϕ have constant b_k values, whereas those of even powers of ϕ have only two different b_k values after b_0:

```
> for k from 1 to 20 by 2 do
>   printf("%2d %30a  %2d %30a\n", k, convert(phi^k, confrac, 5),
>                                  k+1, convert(phi^(k+1), confrac,5))
> end do:
 1                [1, 1, 1, 1, 1]   2                [2, 1, 1, 1, 1]
 3                [4, 4, 4, 4, 4]   4                [6, 1, 5, 1, 5]
 5           [11, 11, 11, 11, 11]   6             [17, 1, 16, 1, 16]
 7           [29, 29, 29, 29, 29]   8             [46, 1, 45, 1, 45]
 9           [76, 76, 76, 76, 76]  10          [122, 1, 121, 1, 121]
11      [199, 199, 199, 199, 199]  12          [321, 1, 320, 1, 320]
13      [521, 521, 521, 521, 521]  14          [842, 1, 841, 1, 841]
15 [1364, 1364, 1364, 1364, 1364]  16       [2206, 1, 2205, 1, 2205]
17 [3571, 3571, 3571, 3571, 3571]  18       [5777, 1, 5776, 1, 5776]
19 [9349, 9349, 9349, 9349, 9349]  20    [15126, 1, 15125, 1, 15125]
```

The Fibonacci numbers arise in a problem of rabbit populations, and satisfy the recurrence $F_{n+2} = F_{n+1} + F_n$, with starting conditions $F_0 = 0$ and $F_1 = 1$. As n increases, the ratio F_{n+1}/F_n approaches the limit of $\phi \approx 1.618$, showing that unchecked population growth is exponential. The Fibonacci numbers show up in an astonishing variety of places in mathematics, and even in vegetable gardens, where some plants exhibit the Fibonacci number sequence in their growth patterns. An entire journal, the *Fibonacci Quarterly*,[6] is devoted to their study, which still remains active, more than 800 years after they were first discovered.

Continued fractions for irrational numbers have the nice property of providing the *best rational approximation*, in the sense that if t is irrational, its k-th convergent is the rational number a/b, where a and b have no common factors, and no other rational number c/d with $|d| < |b|$ is closer to t. All floating-point numbers are rational numbers, and that property can be exploited to find function arguments x for which $f(x)$ is almost exactly halfway between two adjacent floating-point numbers. Such arguments provide challenging tests for software that attempts to guarantee correct rounding of function values. They also offer information about how much numerical precision is needed for argument reduction, an important topic that we treat in **Chapter 9**.

Successive convergents of irrational numbers are alternately lower and upper bounds. For example, some of the approximations derived from the continued fraction for π may be familiar from schoolbooks:

$$\begin{aligned}
\pi_0 &= 3 \\
&\approx \pi - 0.1416, \\
\pi_1 &= 3 + 1/7 \\
&= 22/7
\end{aligned}$$

[6] See http://www.math.utah.edu/pub/tex/bib/index-table-f.html#fibquart.

$$\begin{aligned}
&\approx \pi + 1.264 \times 10^{-3}, \\
\pi_2 &= 3 + 1/(7 + 1/15) \\
&= 333/106 \\
&\approx \pi - 8.322 \times 10^{-5}, \\
\pi_3 &= 3 + 1/(7 + 1/(15 + 1/1)) \\
&= 355/113 \\
&\approx \pi + 2.668 \times 10^{-7}.
\end{aligned}$$

With an additional argument to record the convergents, the Maple `convert()` function can easily produce many more approximations to π:

```
> convert(Pi, confrac, 15, cvgts):
> cvgts;
            333  355  103993  104348  208341  312689  833719
    [3, 22/7, ---, ---, ------, ------, ------, ------, ------,
            106  113  33102   33215   66317   99532   265381

     1146408  4272943  5419351  80143857  165707065  245850922
     -------, -------, -------, --------, ---------, ---------]
     364913   1360120  1725033  25510582  52746197   78256779
```

We can then check the bounding and convergence properties like this:

```
> for k to 15 do printf("%2d  % .3e\n", k, op(k,cvgts) - Pi) end do:
 1  -1.416e-01
 2   1.264e-03
 3  -8.322e-05
 4   2.668e-07
 5  -5.779e-10
 6   3.316e-10
 7  -1.224e-10
 8   2.914e-11
 9  -8.715e-12
10   1.611e-12
11  -4.041e-13
12   2.214e-14
13  -5.791e-16
14   1.640e-16
15  -7.820e-17
```

Some common irrational or *transcendental*[7] constants have easy-to-program simple continued fractions that allow them to be computed at run time to arbitrary precision without the need to retrieve their coefficients from tables. Here we generate them in Mathematica, which displays simple continued fractions as braced lists:

```
% math
In[1]:= ContinuedFraction[Sqrt[2], 15]
Out[1]= {1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2}

In[2]:= ContinuedFraction[Sqrt[3], 15]
Out[2]= {1, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2}

In[3]:= ContinuedFraction[(1 + Sqrt[5])/2, 15]
Out[3]= {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

[7] A transcendental number is an irrational number that cannot be the root of any finite polynomial with integer coefficients. The transcendental property of e was proved by Hermite in 1873, and of π by von Lindemann in 1882. Most numbers on the real axis are transcendental, but there are many open problems. For example, we do not know whether $e + \pi$, e^{π}, or π^{e} are transcendental, and we cannot yet prove that the digits of π are random.

```
In[4]:= ContinuedFraction[Exp[1], 18]
Out[4]= {2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1, 12}

In[5]:= ContinuedFraction[Tan[1/2], 18]
Out[5]= {0, 1, 1, 4, 1, 8, 1, 12, 1, 16, 1, 20, 1, 24, 1, 28, 1, 32}
```

Binary floating-point numbers can be converted to and from simple continued fractions with practical hardware, and continued fractions for the computation of certain elementary functions have attracted the interest of a few hardware designers [FO01, Chapter 12].

Continued fractions are not just for evaluation of irrational numbers, however. One of the oldest-known continued fractions is that for the trigonometric tangent function (Lambert, about 1760), which we treat later in **Section 11.2** on page 302:

$$\begin{aligned}
\tan(x) &= 0 + \frac{x}{1+}\;\frac{x^2}{-3+}\;\frac{x^2}{5+}\;\frac{x^2}{-7+}\;\frac{x^2}{9+}\;\frac{x^2}{-11+}\;\cdots, && x \neq (2n+1)\pi/2,\\
b_0 &= 0, \qquad b_k = (-1)^{k+1}(2k-1), && \text{for } k > 0,\\
a_1 &= x, \qquad a_k = x^2, && \text{for } k > 1.
\end{aligned}$$

The coefficients are much simpler than those of the Taylor series, and convergence is rapid for small x. For example, with $x = 1/100$, $\tan(x)$ and the truncated continued fraction $x/(1 + x^2/(-3 + x^2/(5 + x^2/(-7))))$ agree to better than 21 decimal digits. Lambert was able to prove that if x is a nonzero rational number, then $\tan(x)$ is irrational; thus, because $\tan(\pi/4) = 1$, $\pi/4$ and π cannot be rational, so they must be irrational.

2.8 Summation of continued fractions

We promised in the preceding section to show alternative ways of computing continued fractions from forward summations that allow early exit on convergence.

The first of those handles the particular case of a continued fraction where all but the first of the b_k are equal to one, and there are subtractions, instead of additions, in the denominator:

$$F(x) = b_0 + \frac{a_0}{1-}\;\frac{a_1}{1-}\;\frac{a_2}{1-}\;\frac{a_3}{1-}\;\frac{a_4}{1-}\;\frac{a_5}{1-}\;\frac{a_6}{1-}\;\frac{a_7}{1-}\cdots.$$

A fraction of that type occurs in the incomplete gamma function that we consider later in this book in **Section 18.4** on page 560. The continued fraction can be evaluated in the forward direction as a sum with easily computed term recurrences:

$$\begin{aligned}
F(x) &= b_0 + \sum_{k=0}^{\infty} t_k, \\
r_0 &= 0, && t_0 = a_0,\\
r_k &= \frac{a_k(1 + r_{k-1})}{1 - a_k(1 + r_{k-1})}, && t_k = r_k t_{k-1}, && k > 0.
\end{aligned}$$

The n-th partial sum of the terms t_k, plus b_0, is mathematically identical to the n-th convergent of the continued fraction, $F_n(x)$.

The second of those summation formulas was discovered by a mathematical physics researcher [BFSG74], and is called *Steed's algorithm*. A textbook description is available elsewhere [GST07, pages 181–183], so we merely exhibit a code template that can be readily adapted in just *six statements* to evaluate any particular continued fraction:

```
static fp_t
eval_cf_steed(fp_t x)
{
    fp_t a1, ak, b0, b1, bk, Ck, Ckm1, Ckm2, Dk, Dkm1, Ek, Ekm1;
    int k;
    b0 =                                /* TO BE SUPPLIED */;
    b1 =                                /* TO BE SUPPLIED */;
    a1 =                                /* TO BE SUPPLIED */;
    Ckm2 =                              /* TO BE SUPPLIED */;
    Dkm1 = ONE / b1;
    Ekm1 = a1 * Dkm1;
    Ckm1 = Ckm2 + Ekm1;

    for (k = 2; k <= MAXITER; ++k)
    {
        ak =                             /* TO BE SUPPLIED */;
        bk =                             /* TO BE SUPPLIED */;
        Dk = ONE / (Dkm1 * ak + bk);
        Ek = (bk * Dk - ONE) * Ekm1;
        Ck = Ckm1 + Ek;

        if (Ck == Ckm1)                  /* converged */
            break;

        Ckm1 = Ck;
        Dkm1 = Dk;
        Ekm1 = Ek;
    }

    return (Ck);
}
```

The value of MAXITER limits the number of iterations, but its value needs to be determined by numerical experiment for each particular choice of coefficients, and for each numerical precision.

Steed's algorithm implicitly includes scaling of intermediate terms to reduce the likelihood of out-of-range numbers. Nevertheless, numerical experiments must verify that.

If the denominator Dkm1 * ak + bk in any iteration can become tiny, or zero, Steed's algorithm may be unsatisfactory. That brings us to the final recipe in this section, the *modified Lentz algorithm* [TB86]. It handles rescaling differently, and may be less sensitive to small denominators, although it requires a fudge factor that may need adjustment for a particular continued fraction. A textbook treatment provides some background [GST07, pages 183–185], so once again, we just present a code template:

```
static fp_t
eval_cf_lentz(fp_t x)
{
    fp_t ak, b0, bk, Ck, Ckm1, Dk, Dkm1, Ek, Ekm1, eps, Hk;
    int k;

    b0 =                                /* TO BE SUPPLIED */;
    eps = FP(1.0e10) / FP_T_MAX; /* may need to fudge the numerator */
    Ckm1 = b0;

    if (b0 == ZERO)
        Ckm1 = eps;

    Ekm1 = Ckm1;
    Dkm1 = ZERO;

    for (k = 1; k <= MAXITER; ++k)
    {
        ak =                            /* TO BE SUPPLIED */;
        bk =                            /* TO BE SUPPLIED */;
        Dk = bk + ak * Dkm1;
        if (Dk == ZERO)
            Dk = eps;

        Ek = bk + ak / Ekm1;

        if (Ek == ZERO)
            Ek = eps;

        Dk = ONE / Dk;
        Hk = Ek * Dk;
        Ck = Ckm1 * Hk;

        if ((ONE + (Hk - ONE)) == ONE)  /* converged */
            break;

        Ckm1 = Ck;
        Dkm1 = Dk;
        Ekm1 = Ek;
    }

    return (Ck);
}
```

Here, `FP_T_MAX` is the largest representable floating-point number, and `eps`, which replaces zero values that later appear as divisors, must be chosen as a tiny value whose reciprocal is still a finite floating-point number. The value `FP(1.0e10)` may therefore need to be adjusted for some historical floating-point systems. As with Steed's algorithm, the iteration limit `MAXITER` must be suitably adjusted for each application.

The three summation algorithms given in this section are not yet widely known, and are absent from the few textbook treatments of numerical evaluation of continued fractions. The only other textbook presentations of the Steed and Lentz algorithms known to this author are recent editions of the *Numerical Recipes* series [PTVF07, pages 206–209].

In some later chapters of this book, we use those summation algorithms for evaluation of continued fractions, but we do not repeat their code, because only the few lines that define the coefficients a_k and b_k need to be altered.

To learn more about the mathematics of continued fractions, see a recent reprint of the original 1948 edition of Wall's classic book, *Analytic Theory of Continued Fractions* [Wal00], two short monographs [RS92, KE97], and six advanced books [JT84, Bre91, LW92, Hen06, Khr08, LW08]. The second last of those books shows interesting relations of continued fractions to the computation of mathematical constants, to the design of accurate calendar systems, and to the construction of Eastern and Western musical scales.

For their application to some of the functions covered in this book, see the *Handbook of Continued Fractions for Special Functions* [CPV$^+$08]. The authors of that book used symbolic-algebra systems to verify all of their formulas, and in doing so, found and corrected many errors in earlier books and research articles. Computers cannot replace, but can often help, mathematicians.

2.9 Asymptotic expansions

Functions can sometimes be formally represented by a series in inverse powers of the form

$$f(x) \asymp c_0 + c_1/x + c_2/x^2 + \cdots + c_n/x^n + \cdots .$$

Here, the relational operator $\asymp$ indicates an *asymptotic expansion*. It has the property that partial sums to arbitrarily high order *diverge*. However, the first few terms decrease in magnitude, and then increase. A partial sum up to the term of smallest magnitude differs from $f(x)$ by an error that is about the size of the first term omitted.

There is a long mathematical history of asymptotic expansions and their use in the evaluation of elementary and special functions; see, for example, *Asymptotics and Special Functions* [Olv74].

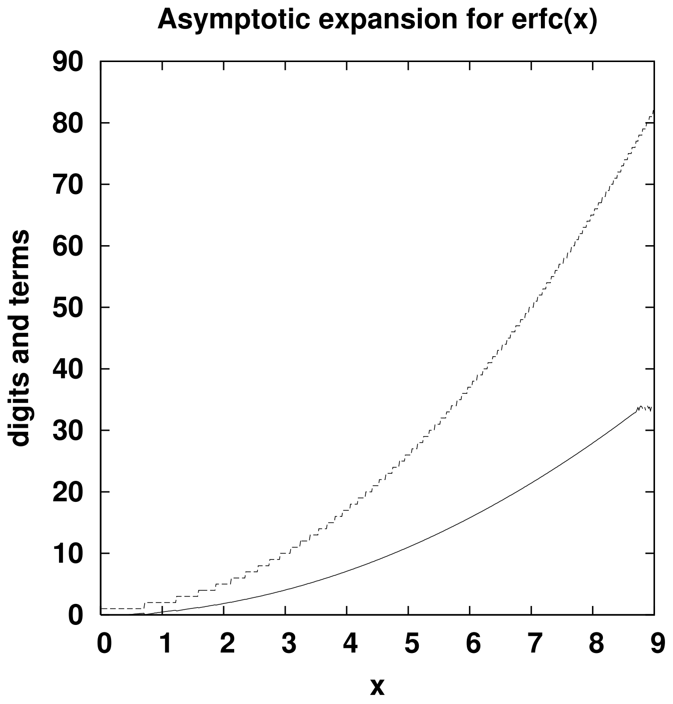


Figure 2.1: Accuracy of the asymptotic expansion for $\mathrm{erfc}(x)$. The solid lower curve shows the number of correct decimal digits produced in 128-bit IEEE 754 arithmetic, and the dashed upper curve shows the number of terms summed.

The problem with asymptotic expansions is that they cannot be used to evaluate a function to arbitrary accuracy just by summing more and more terms. Instead, evaluation must stop as soon as the magnitude of the next term increases. For arguments of large magnitude, high accuracy may be possible, but for arguments of small magnitude, accuracy drops. Figure 2.1 illustrates that phenomenon for the asymptotic expansion of a function that we treat later in Section 19.1 on page 593.

In numerical software that is designed to produce high accuracy for a range of floating-point precisions from a common code base, as we do in the mathcw library, asymptotic expansions are usually inadvisable. Nevertheless, they can be useful in software development and testing, by providing an alternate, and independent, computational route to a function, just as we saw for continued fractions. For numerical work, they can sometimes be replaced by shorter polynomial approximations, a topic that we address in Chapter 3.

Despite the caveat in the preceding paragraph, we use asymptotic expansions later in this book for large-argument computation of a few functions, including gamma, log-gamma, and psi functions (Chapter 18), the cumulative distribution functions for the standard normal distribution (Section 19.4), and some of the Bessel functions (Chapter 21). In each case, in the regions for which the asymptotic series are summed, the arguments are big enough that full machine precision is attainable.

2.10 Series inversion

If we have a representation of a function as the series

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$$

with known coefficients a_k, then for a given x, we can straightforwardly evaluate the right-hand side to find the function value.

However, suppose that we know the value of the left-hand side, $y = f(x)$, and we wish to determine the x value that produces y. That problem is known as *series inversion*. That is, we seek an expansion

$$x = b_0 + b_1 y + b_2 y^2 + b_3 y^3 + \cdots$$

where the coefficients b_k are to be determined. To find them, substitute the expansion of x in the series for y, and then rearrange the result to find coefficients of powers of y. The result is an equation of the form

$$y = [\text{mess}]_0 + [\text{bad mess}]_1 y + [\text{horrid mess}]_2 y^2 + [\text{ghastly mess}]_3 y^3 + \cdots .$$

Because that must hold for arbitrary y, all but one of the right-hand side coefficients must be zero, and the remaining factor, $[\text{bad mess}]_1$, must be one. Each coefficient therefore leads to an equation, and those equations can be solved in the forward direction starting with the zeroth, $[\text{mess}]_0 = 0$.

Series inversion is tedious to work out by hand, but symbolic-algebra systems make it easy. Here is an example in Mathematica:

```
In[1]:= Replace[InverseSeries[Series[Sin[x], {x, 0, 11}], x], x->y, 1]

             3      5      7       9       11
            y    3 y    5 y    35 y    63 y         12
Out[1]= y + -- + ---- + ---- + ----- + ------ + O[y]
            6     40    112    1152     2816
```

For comparison, we can ask for the expansion of the inverse of the sine function, known as the *arcsine*:

```
In[2]:= Series[ArcSin[y], {y, 0, 11}]

             3      5      7       9       11
            y    3 y    5 y    35 y    63 y         12
Out[2]= y + -- + ---- + ---- + ----- + ------ + O[y]
            6     40    112    1152     2816
```

The two outputs are identical, as expected.

Maple can find the inverse series with its general equation solver, but we have to tell the `series()` function the order of the highest term that we want in the output display:

```
> Order := 13:
> solve(series(sin(x), x = 0, 11) = y, x);

             3         5          7    35   9    63   11       13
    y + 1/6 y  + 3/40 y  + 5/112 y  + ---- y  + ---- y   + O(y  )
                                      1152      2816
```

The syntax in the MuPAD algebra system is similar to that of Maple, but MuPAD has a separate function for inverting series expansions:

```
>> subs(revert(series(sin(x), x = 0, 11)), x = y);

         3      5      7       9       11
        y    3 y    5 y    35 y    63 y         13
    y + -- + ---- + ---- + ----- + ------ + O(y  )
         6    40     112    1152    2816
```

Many of the functions in this book have a companion function that is their inverse: square root and square, exponential and logarithm, sine and arcsine, and so on, and you can easily find them on hand calculators and in software libraries. However, the inverse functions are sometimes less well-known, and we consider some examples later in **Chapter 19**.

2.11 Summary

Iterative solutions of nonlinear equations with the first- and second-order procedures described in this chapter have broad applicability in numerical computation. Importantly, the algorithms can be generalized to find numerical solutions of problems involving more than the single variable that we generally consider in this book.

Representations of functions with Taylor series, asymptotic expansions, and continued fractions are essential tools for developing practical computational algorithms for function evaluation. Although traditional textbooks in numerical analysis often rely on mathematical encyclopedias, handbooks, and tables as sources of such expansions, we have seen in this chapter that symbolic-algebra languages offer more accessible, and more powerful, ways for finding, and then manipulating, such material. Those languages also permit numerical experimentation with arithmetic of arbitrary precision, so they can be invaluable for prototyping algorithms before implementing them in more conventional programming languages. We use symbolic-algebra systems often in this book, and we need more than one of them, because none yet supplies all of the capabilities that we require.

The famed computer scientist Richard Hamming is widely cited for the quote *The purpose of computing is insight, not numbers*. Mathematics, symbolic-algebra systems, and graphical displays of data and functions, are helpful tools for developing that insight.

3 Polynomial approximations

> ALL EXACT SCIENCE IS DOMINATED
> BY THE IDEA OF APPROXIMATION.
>
> — BERTRAND RUSSELL.

We begin this chapter by introducing a convenient notation that is common in some branches of mathematics for indicating a numeric interval: a comma-separated pair of values, enclosed in square brackets or parentheses, with the convention that a bracket means that the endpoint is *included*, and a parenthesis indicates that the endpoint is *excluded*. Thus, $[1, 2)$ is the set of values x for which $1 \leq x < 2$, and $[2, 3]$ is the adjacent set for which $2 \leq x \leq 3$. Endpoint distinction is often important in numerical computation, and we use that interval notation extensively in the rest of this book.

The computation of many of the elementary functions involves relating the value of $f(x)$ on the full domain of the function, often $(-\infty, +\infty)$ or $[0, \infty)$, to its value in a small interval near zero where it can be computed by a rapidly convergent Taylor series, or a polynomial approximation to that series. The original function can then be obtained by a few accurate operations from the approximated value.

In the Cody/Waite algorithms, the sole exception to that procedure is the square root. As we show later in **Section 8.1** on page 215, it can be computed by starting with an initial linear approximation that provides about seven correct bits, and then using quadratically convergent Newton–Raphson iterations that double the number of correct bits at each step. Even better, pairs of those iterations can be merged to eliminate one multiply, so the double-step iteration *quadruples* the number of correct bits with just two adds, two divides, and one multiply.

Later in this chapter, we show how to find polynomial approximations to functions and series, but for now, we assume that we have the polynomial, and simply want to evaluate the function accurately. The next three sections first treat two special cases, and then the general case.

3.1 Computation of odd series

In this section, we consider the case of series having only odd-order terms, corresponding to functions that are *asymmetric* about the origin, $f(-x) = -f(x)$:

$$f(x) = c_1 x + c_3 x^3 + c_5 x^5 + \cdots + c_{2n+1} x^{2n+1} + \cdots .$$

Move the first term to the left and divide by x^3 to get

$$(f(x) - c_1 x)/x^3 = c_3 + c_5 x^2 + \cdots + c_{2n+1} x^{2n-2} + \cdots .$$

Now let $g = x^2$, and write that as

$$\begin{aligned}
\mathcal{R}(g) &= (f(\sqrt{g}) - c_1\sqrt{g})/(\sqrt{g}^3) \\
&= c_3 + c_5 g + c_7 g^2 + \cdots + c_{2n+1} g^{n-1} + \cdots \\
&\approx \mathcal{P}(g)/\mathcal{Q}(g), \qquad \textit{computed rational approximation}, \\
&\approx \mathcal{P}(x^2)/\mathcal{Q}(x^2).
\end{aligned}$$

Once we have an approximation to $\mathcal{R}(g)$ as $\mathcal{P}(g)/\mathcal{Q}(g)$, to be computed by methods described later in **Section 3.4** on page 28 and **Section 3.5** on page 32, we can obtain $f(x)$ by a simple rearrangement:

$$\begin{aligned}
f(x) &\approx c_1 x + (\mathcal{P}(x^2)/\mathcal{Q}(x^2))x^3 \\
&\approx c_1 x + x(g\mathcal{P}(g)/\mathcal{Q}(g)).
\end{aligned}$$

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_3

Do *not* factor out x in that last result and compute the function from $x(c_1 + g\mathcal{P}(g)/\mathcal{Q}(g))$: depending on the value of c_1, that form can lose up to three leading bits on systems with a hexadecimal base.

Most of the Cody/Waite algorithms contain those two steps, and eliminate the multiplication by c_1, because it is exactly one for all of the odd-order-term elementary functions considered in their book.

There are several convenient features of that computational scheme:

- Because x is small in the region of interest, g is even smaller. Thus, the right-hand side series converges rapidly, and is close to linear if the two leading terms $c_3 + c_5 g$ dominate the remaining terms. Because $\mathcal{R}(g)$ is nearly linear, low-order polynomials $\mathcal{P}(g)$ and $\mathcal{Q}(g)$ can produce highly accurate fits.

- When the term involving the rational polynomial is much smaller than the leading terms, and if those terms can be computed exactly, or at least with only two or three rounding errors, then provided that the polynomial is accurate, the computed result is usually close to the exact answer.

- Because g is the square of a real number, it is positive. If the coefficients in $\mathcal{P}(g)$ have the same sign, there is no possibility of subtraction loss in its computation, and similarly for $\mathcal{Q}(g)$. Given a choice of several different polynomials, we therefore prefer those with coefficients of *constant sign*.

  When the coefficients are of mixed sign, our program that generates them makes numerical tests for subtraction loss in the interval of approximation, and warns about such problems. Fortunately, such losses are rare, and have been found only in the approximations for the error functions (**Section 19.1** on page 593), the psi function (**Section 18.2** on page 536), and the reciprocal of the gamma function (**Section 18.1** on page 521).

- The nested Horner[1] form of polynomial computation,

$$\begin{aligned} \mathcal{P}_n(g) &= \sum_{k=0}^{n} p_k g^k, && \textit{summation form}, \\ &= p_0 + p_1 g + p_2 g^2 + \cdots + p_n g^n \\ &= (\cdots(p_n g + p_{n-1})g \cdots + p_1)g + p_0, && \textit{Horner form}, \end{aligned}$$

  computes the higher powers first, and they are the smallest terms because powers of g fall off rapidly. That means that the terms are summed in increasing size, which is usually numerically better than summing from largest to smallest term like a straightforward brute-force truncated Taylor series would do.

- Because only the ratio $\mathcal{P}(g)/\mathcal{Q}(g)$ is needed, we can rescale both polynomials by a constant factor to make the high-order coefficient in either of them exactly equal to one, allowing elimination of one multiply.

  For example, with a ratio of two cubic polynomials, we normally have C code like this

```
Pg = ((p[3] * g + p[2]) * g + p[1]) * g + p[0] ;
Qg = ((q[3] * g + q[2]) * g + q[1]) * g + q[0] ;
```

  so that six multiplies and six adds are needed, plus three more multiplies, one add, and one divide to obtain $f(x)$.

  If we scale the coefficients so that $q_3 = 1$ exactly, then we can rewrite the computation of $\mathcal{Q}(g)$ as:

```
Qg = ((       g + q[2]) * g + q[1]) * g + q[0] ;
```

  Alternatively, we could scale to make at least one of p_0 and q_0 have no leading zero bits in a hexadecimal base, which is critically important for the wobbling precision of such systems. Otherwise, with a unit leading term, we lose almost a decimal digit of accuracy.

- In at least one elementary function, the constant term is zero in either $\mathcal{P}(g)$ or $\mathcal{Q}(g)$, so it can be dropped from the computation:

```
Pg = ((p[3]*g + p[2])*g + p[1])*g;  /* + p[0] omitted */
```

- Most of the expressions involve combinations of multiply and add instructions, and some systems can achieve much better speed and accuracy for them. We discuss that further in **Section 4.17** on page 85.

[1] That rule is named after the English mathematician William George Horner (1786–1837). In 1834, Horner also invented the *zoetrope*, an early device for displaying motion pictures.

3.2 Computation of even series

We now turn to the case of series having only even-order terms, for functions that are *symmetric* about the origin, $f(-x) = f(x)$:

$$f(x) = c_0 + c_2x^2 + c_4x^4 + \cdots + c_{2n}x^{2n} + \cdots.$$

As before, move the first term to the left-hand side, and then divide by x^2:

$$(f(x) - c_0)/x^2 = c_2 + c_4x^2 + \cdots + c_{2n}x^{2n-2} + \cdots.$$

Now let $g = x^2$, and write that as

$$\begin{aligned}\mathcal{R}(g) &= (f(\sqrt{g}) - c_0)/g \\ &= c_2 + c_4g + c_6g^2 + \cdots + c_{2n}g^{n-1} + \cdots \\ &\approx \mathcal{P}(g)/\mathcal{Q}(g), && \textit{computed rational approximation}, \\ &\approx \mathcal{P}(x^2)/\mathcal{Q}(x^2).\end{aligned}$$

Once we have an approximation to $\mathcal{R}(g)$ as $\mathcal{P}(g)/\mathcal{Q}(g)$, we can obtain $f(x)$ by a simple rearrangement:

$$\begin{aligned}f(x) &\approx c_0 + x^2\mathcal{R}(x^2) \\ &\approx c_0 + g\mathcal{P}(g)/\mathcal{Q}(g).\end{aligned}$$

That differs only slightly from the results for odd-order series.

Because all of the even-order-term elementary functions have $c_0 = 1$, to reduce the effect of wobbling precision on base-16 systems, it is better to halve the function to be approximated:

$$\begin{aligned}\mathcal{R}(g) &= (f(\sqrt{g}) - c_0)/(2g) \\ &\approx \mathcal{P}(g)/\mathcal{Q}(g), && \textit{computed rational approximation}.\end{aligned}$$

Then compute

$$f(x) \approx 2(\tfrac{1}{2} + g\mathcal{P}(g)/\mathcal{Q}(g)).$$

The value $\frac{1}{2}$ has no leading zero bits in a hexadecimal base, so the parenthesized expression is computed to full precision. In addition, in the common case when the base is 2, the extra factor of two is handled exactly, so the cost of making the code portable is a doubling that requires only one add or multiply.

3.3 Computation of general series

Of the functions treated by Cody and Waite and supported by the mathcw library, only the exponential and log functions have Taylor series with both odd and even terms. The exponential function is represented like this:

$$\exp(x) = 1 + x + x^2/2! + x^3/3! + \cdots + x^n/n! + \cdots.$$

As before, move the first term on the right to the other side of the equation, and then divide by x:

$$\begin{aligned}(\exp(x) - 1)/x &= 1 + x/2! + x^2/3! + \cdots + x^n/(n+1)! + \cdots \\ &\approx 1 + x\mathcal{R}(x).\end{aligned}$$

If we proceed as before, we let $g = x^2$, and find

$$\mathcal{R}(g) = ((\exp(\sqrt{g}) - 1)/\sqrt{g} - 1)/\sqrt{g}.$$

Unfortunately, that function is far from linear in the interval $[0, (\ln(2)/2)^2] \approx [0, 0.1201]$ on which the reduced exponential is computed, which we can readily see with another short Maple program shown in **Figure 3.1** on the following page.

```
% maple
...
> R := proc(g) return ((exp(sqrt(g)) - 1)/sqrt(g) - 1)/sqrt(g) end proc:
> plot(R(g), g = 0 .. ln(2) / 2);
  +                                                          AAAA
0.6                                                      AAAAA
  +                                                 AAAAA
  +                                             AAAAA
  +                                         AAAA
0.58                                    AAAAA
  +                                  AAAA
  +                              AAAA
  +                          AAAA
  +                       AAAA
0.56                   AAAA
  +                 AAAA
  +              AAAA
  +            AAA
0.54         AA
  +       AAA
  +      AA
  +    AA
  +  AAA
0.52AA
  +AA
  +A
  *
  *-+-+-+--+-+-+-+-+-+-+-+--+-+-+-+-+-+-+--+-+-+-+-+-+-+--+-+-+-+-+-+-+--+-+-
  0        0.05       0.1       0.15        0.2      0.25        0.3
```

Figure 3.1: Exponential function approximation, the wrong way. The text-terminal version of Maple produces plots like this. Plots from a window-system version of Maple are more realistic, but less easily displayed everywhere.

Fortunately, Cody and Waite found a much better almost-linear function to approximate, shown in Figure 3.2 on the next page. Thus, we find instead a rational polynomial approximation to the function

$$\begin{aligned}\mathcal{R}(g) &= (\exp(\sqrt{g}) - 1)/\big(\sqrt{g}(\exp(\sqrt{g}) + 1)\big)\\ &= (\exp(x) - 1)/\big(x(\exp(x) + 1)\big)\\ &\approx \mathcal{P}(g)/\mathcal{Q}(g), \qquad \textit{computed rational approximation},\end{aligned}$$

from which we have

$$x\mathcal{R}(g) = (\exp(x) - 1)/(\exp(x) + 1).$$

Rearrange that to find the exponential function:

$$\exp(x) \approx (1 + x\mathcal{R}(g))/(1 - x\mathcal{R}(g)).$$

Add and subtract the right denominator from the numerator:

$$\begin{aligned}\exp(x) &\approx (1 - x\mathcal{R}(g) - 1 + x\mathcal{R}(g) + 1 + x\mathcal{R}(g))/(1 - x\mathcal{R}(g))\\ &\approx 1 + 2x\mathcal{R}(g)/(1 - x\mathcal{R}(g)).\end{aligned}$$

Finally, substitute $\mathcal{R}(g) = \mathcal{P}(g)/\mathcal{Q}(g)$ and rearrange:

$$\begin{aligned}\exp(x) &\approx 1 + 2x\mathcal{P}(g)/(\mathcal{Q}(g) - x\mathcal{P}(g))\\ &\approx 2(\tfrac{1}{2} + x\mathcal{P}(g)/(\mathcal{Q}(g) - x\mathcal{P}(g))).\end{aligned}$$

```
% maple
...
> R := proc(g) return ((exp(sqrt(g)) - 1) /
                       (sqrt(g) * (exp(sqrt(g)) + 1))) end proc:
> plot(R(g), g = 0 .. ln(2) / 2);
0.5A
  +AAA
  +   AAA
  +      AAA
0.498       AA
  +           AAAA
  +              AAAA
  +                 AAA
0.496                  AAA
  +                       AAA
  +                          AA
0.494                          AAAA
  +                               AAAA
  +                                  AAAA
  +                                     AAAA
0.492                                      AAA
  +                                           AAA
  +                                              AAA
  +                                                 AAA
0.49                                                   AAA
  +                                                      AAAA
  +                                                          AAA
0.488                                                           AAA
  +                                                               AAAA
  +                                                                  AAAA
  +                                                                     AAAA
0.486-+-+--+-+-+-+-+-+-+-+--+-+-+-+-+-+-+--+-+-+-+-+-+-+--+-+-+-+-+-+-+--+-+*
  0        0.05       0.1       0.15        0.2      0.25        0.3
```

Figure 3.2: Exponential function approximation, the right way.

As before, the last form is computationally preferable in the presence of wobbling precision, and despite the subtraction, it is straightforward to show by direct evaluation of $\mathcal{P}(g)$ and $\mathcal{Q}(g)$ for $\exp(x)$ that there is no loss of leading bits when x is in the interval $[0, (\ln(2)/2)^2]$ where we use the rational polynomial approximation.

It is instructive to examine the Taylor-series expansion of the right-hand side of $\mathcal{R}(g)$:

$$\begin{aligned}\mathcal{F}(x) &= (\exp(x) - 1)/\big(x(\exp(x) + 1)\big) \\ &= (1/2) - (1/24)x^2 + (1/240)x^4 - (17/40320)x^6 + \\ &\quad (31/725760)x^8 - \mathcal{O}(x^{10}).\end{aligned}$$

Terms of odd order are absent, so when x is small, the function is almost linear in the variable $g = x^2$, as **Figure 3.2** shows. In addition, even though the formation of x^2 introduces a rounding error, the effect of that error on the sum is tiny, and it should often be possible to compute $\mathcal{F}(x)$ correct to machine precision. That way, the only sources of rounding error in the reconstructed elementary function come from the half-dozen operations that add a small correction to the exact constant 1, and the rounding error in that last addition is the dominant one. That suggests that the reconstruction should rarely be in error by more than one unit in the last digit.

The range of the graph in **Figure 3.2** is small, so we could, if desired, gain about two decimal digits of precision by using fits to $\mathcal{G}(x) = \mathcal{F}(x) - \frac{1}{2}$, and about four decimal digits from fits to $\mathcal{H}(x) = (\mathcal{F}(x) - \frac{1}{2})/x^2 + 1/24$, provided that we handle the reconstruction of $\mathcal{F}(x)$ carefully.

If the correction term can be computed in higher precision, then with high probability, the final result is correctly rounded. For example, in IEEE 754 binary arithmetic, moving from single precision to double precision provides 29 additional bits, so incorrect rounding has a probability of approximately $2^{-29} \approx 1.9 \times 10^{-9}$. Because there are about $2^{31} \approx 2.1 \times 10^{9}$ possible single-precision values, we should find only about *four* single-precision results that are not exact to machine precision.

If the arguments leading to those incorrectly rounded results were known in advance, such as by exhaustive search, we could even check for them explicitly, and return correctly rounded constant values, thereby guaranteeing correct rounding of the single-precision exponential for *all* arguments. Unfortunately, exhaustive search is infeasible for higher precisions, but some researchers are working on finding hard cases for rounding of some of the elementary functions, using a combination of clever mathematical analysis and computationally intensive numerical searches [LM01, SLZ02, LM03a, LM03b, SLZ03, dDDL04, dDG04, SZ04, Lef05, SZ05, SLZ05, LSZ06, Zim06, HLSZ07, KLL^{+}10, JLMP11].

Auxiliary functions of the form of $\mathcal{F}(x)$ prove useful in several other elementary functions treated in this book, including error functions, hyperbolic sine, hyperbolic tangent, inverse sine, inverse tangent, logarithm, power functions, sine, and tangent.

3.4 Limitations of Cody/Waite polynomials

In IEEE 754 arithmetic, which we describe in more detail in **Section 4.3** on page 63, the significand precisions are 24, 53, 64, and 113 bits. Cody and Waite usually tabulate polynomial coefficients for various approximations that are accurate to 25 to 60 bits. We therefore have to extend their work to handle the higher precision needed for the IEEE 754 80-bit and 128-bit formats. The `maple` subdirectory of the mathcw library package contains programs to do that.

In general, much higher precision is needed to find the polynomial approximations: we found that 100-decimal-digit computation was needed for approximations good to about 30 to 50 digits, and computation with up to 250 decimal digits was sometimes needed to extend the coverage to 75-digit accuracy. When the precision is too small, Maple's polynomial-fitting routines simply return failure.

Cody and Waite do not describe the origin of their polynomial approximations, or how they were computed. At the time of their book, the choices would have likely been Richard P. Brent's then-recent Fortran multiple-precision library [Bre78a, Bre78b], or the venerable symbolic-algebra languages Macsyma or REDUCE, both of which were developed in the 1960s, the first at MIT, and the second at Utah. A descendant of Macsyma, called Maxima, is available under an open-source license.[2] In early 2009, REDUCE was released as free software [Hea09],[3] and somewhat earlier, its author published a 40-year historical retrospective [Hea05]. Neither Maple nor Mathematica existed in 1980, although together, they have dominated the commercial symbolic-algebra market since the late 1980s.

In Maple, you can use something like this

```
% maple
> with(numapprox):      # load the library of approximation routines
> Digits := 100;        # set the working precision in decimal digits
> the_approx := minimax(F('x'), 'x' = A .. B, [np,nq]):
```

to find a rational polynomial approximation $\mathcal{P}(x)/\mathcal{Q}(x)$ of order `np` and `nq` for x in the interval $[A, B]$ for the function $F(x)$.

The quotes around `x` prevent confusion with any existing variable of that name. The quotes can be omitted if `x` has not yet been assigned a value. If it has, we can remove that value by the assignment `x := 'x':`, a simple, and often useful, technique in Maple sessions. We therefore omit the quotes in Maple code in the rest of this book.

For example, to find an approximation to the ordinary sine function in the interval $[0, \pi/4]$, you can enter this short program:

```
> with(numapprox):  # load the library of approximation routines
>
> Digits := 20:     # set the working precision in decimal digits
>
```

[2] See `http://maxima.sf.net/`.

[3] See `http://reduce-algebra.com/`.

```
> minimax(sin(x), x = 0 .. Pi/4, [3,3]); # find a rational cubic fit
                    -8
(0.5607715527246 10   + (0.97501461057610330856

     + (0.03531206456923826426 - 0.11812024876078269427 x) x) x)/(

    0.97501530982739456391 + (0.03529785259862835146 6

     + (0.04449030904663320313 4 + 0.00549419087972849369 13 x) x) x)
```

Of course, that display form is not convenient for computation in other programming languages, nor does the returned expression offer any hint of its accuracy. The `*.map` programs in the `maple` subdirectory of the `mathcw` distribution therefore go to a lot of extra trouble to compute error estimates, and to format and output the polynomial data so that they are suitable for use in Fortran, C, C++, Java, and `hoc`. Here is a small example of their output, using the modified function $\mathcal{F}(g) = (\sin(\sqrt{g}) - \sqrt{g})/(g\sqrt{g})$:

```
> pq_approx(2, 2):

Attempting minimax approximation of degree [2, 2]
OKAY

WARNING: Forcing F(0) = P(0) / Q(0) by resetting p[0] from -8.3333333326e-02 to -8.3333333333e-02

Fits computed with 20-digit precision
Number of correct bits at uniform spacing on [0, (PI/2)**2]:
   0.00   33.31   32.64   32.41   32.38   32.48   32.69   32.99
  33.41   33.96   34.70   35.77   37.70   39.55   43.29   36.95
  35.48   34.58   33.94   33.46   33.10   32.82   32.60   32.45
  32.35   32.30   32.31   32.36   32.49   32.68   32.97   33.38
  33.99   34.97   37.00   37.84   39.14   34.89   33.32   32.25

Average number of correct bits on [0, (PI/2)**2] = 34.25
Average number of correct decimal digits on [0, (PI/2)**2] = 10.31
Maple minimax() error = 1.48e-11
Fits computed with 20-digit precision

Maximum relative error of 1.96e-10 at x = 2.467401101
where
        F_exact()  = -0.14727245909836825822
        F_approx() = -0.14727245912726088799

================================================================================
Fortran data

*
*     F(x) = (sin(sqrt(x)) - sqrt(x))/(x*sqrt(x))
*     Maximum relative error in F(x) = 1.96e-10 = 2.0**-32.25
*     at x = 2.4674e+00 for x on [0, (PI/2)**2] with minimax degree [2, 2]
*     Average number of correct bits = 34.25
*     Average number of correct decimal digits = 10.31
*     Maple minimax() error = 1.48e-11
*     Fits computed with 20-digit precision
*
      INTEGER np
      PARAMETER (np = 3)
      REAL*16 p(0:np-1)
************************************************************************
* NB: Remember to change all e exponents to q for use in Fortran!
************************************************************************
      DATA p /
     X   -8.3333333333333e-02,
     X    2.4272274723755e-03,
     X   -2.6103416823681e-05 /
```

```
      INTEGER nq
      PARAMETER (nq = 3)
      REAL*16 q(0:nq-1)
************************************************************************
* NB: Remember to change all e exponents to q for use in Fortran!
************************************************************************
      DATA q /
     X    5.0000000000000e-01,
     X    1.0436634248973e-02,
     X    8.3217137588379e-05 /

      pgg = (((p(2)) * g + p(1)) * g + p(0)) * g
      qg  = ((q(2)) * g + q(1)) * g + q(0)
      result = f + f * (pgg / qg)

================================================================================
C/C++/Java data

/***
 ***     F(x) = (sin(sqrt(x)) - sqrt(x))/(x*sqrt(x))
 ***     Maximum relative error in F(x) = 1.96e-10 = pow(2.0, -32.25)
 ***     at x = 2.4674e+00 for x on [0, (PI/2)**2] with minimax degree [2, 2]
 ***     Average number of correct bits = 34.25
 ***     Average number of correct decimal digits = 10.31
 ***     Maple minimax() error = 1.48e-11
 ***     Fits computed with 20-digit precision
 ***/

#define POLY_P(p,x)     POLY_2(p,x)
#define POLY_Q(q,x)     POLY_2(q,x)

static const
fp_t p[] = {
    FP( -8.3333333333333e-02),
    FP(  2.4272274723755e-03),
    FP( -2.6103416823681e-05) };

static const
fp_t q[] = {
    FP(  5.0000000000000e-01),
    FP(  1.0436634248973e-02),
    FP(  8.3217137588379e-05) };

    pg_g = (((p[2]) * g + p[1]) * g + p[0]) * g;
    qg   = ((q[2]) * g + q[1]) * g + q[0];
    result = f + f * (pg_g / qg);

================================================================================
hoc data

func P32(x) \
{
    p0 =  -8.3333333333333e-02
    p1 =   2.4272274723755e-03
    p2 =  -2.6103416823681e-05

    return (((p2) * x + p1) * x +  p0)
}

func Q32(x) \
{
    q0 =   5.0000000000000e-01
    q1 =   1.0436634248973e-02
    q2 =   8.3217137588379e-05

    return (((q2) * x + q1) * x +  q0)
}

func R32(x) \
{
    ###
    ###     F(x) = (sin(sqrt(x)) - sqrt(x))/(x*sqrt(x))
    ###     Maximum relative error in F(x) = 1.96e-10 = 2**(-32.25)
    ###     at x = 2.4674e+00 for x on [0, (PI/2)**2] with minimax degree [2, 2]
    ###     Average number of correct bits = 34.25
    ###     Average number of correct decimal digits = 10.31
    ###     Maple minimax() error = 1.48e-11
    ###     Fits computed with 20-digit precision
    ###

    return (P32(x) / Q32(x))
}

func F32(x) \
{
    z = x
    pz = P32(z)
    qz = Q32(z)
    return (pz / qz)
}
```

The polynomial arrays are named `p` and `q`, and there are short code fragments that show how to evaluate the polynomials and reconstruct the fitted function. The hoc functions have a numeric suffix that records the number of correct bits at the maximum error of the function fit in the argument interval.

Although Maple uses a square-bracketed list for the polynomial degrees of rational approximations, in this book we avoid confusion with interval brackets by using the notation $\langle m/n \rangle$ for the ratio of a polynomial of degree m with another of degree n.

Because we want to support a wide range of past, present, and future computer architectures, we need several different rational polynomials for each elementary function. The Maple programs consider all possible total degrees in a specified range, such as 2 to 20, and then compute all possible partitions of the total degree, `ntotal = np + nq`, for rational approximations of degree ⟨`np`/`nq`⟩. The programs estimate the error by sampling the function uniformly over the interval in which it is required, as well as in small intervals near the endpoints, tracking the maximum relative error compared to the exact function. The value of that error is then expressed as a power of two, and the floor (see **Section 6.7** on page 136) of the negative of that power is the number of bits of precision.

John Rice wrote in his well-known book *The Approximation of Functions* [Ric64, page 147] about the problem of finding optimal polynomial approximations:

> There appears to be no systematic method of determining, in an *a priori* manner, which of these entries provides the best approximation in a particular sense.

Thus, we simply have to try several candidate fits, and pick the best ones.

A small awk program then examines the Maple output, and produces a compact report of approximations of decreasing accuracy, noting the IEEE 754 precision for which they are suitable, and flagging those that are balanced (same degree in numerator and denominator), of uniform sign, or so inaccurate as to be useless. Here is an example of its use (long lines have been wrapped to fit the page):

```
% awk -f summary.awk atan.out
atan.out:06346:* Maximum relative error in F(x) = 2.34e-25 = 2.0**-81.82 \
                at x = 0.0000 for x on [0,(2 - sqrt(3))**2] with minimax \
                degree [6,6]  = 12     80-bit BALANCED!
atan.out:06469:* Maximum relative error in F(x) = 3.66e-25 = 2.0**-81.18 \
                at x = 0.0000 for x on [0,(2 - sqrt(3))**2] with minimax \
                degree [7,5]  = 12     80-bit
atan.out:06224:* Maximum relative error in F(x) = 3.88e-25 = 2.0**-81.09 \
                at x = 0.0000 for x on [0,(2 - sqrt(3))**2] with minimax \
                degree [5,7]  = 12     80-bit
...
atan.out:01457:* Maximum relative error in F(x) = 3.58e-11 = 2.0**-34.70 \
                at x = 0.0000 for x on [0,(2 - sqrt(3))**2] with minimax \
                degree [3,3]  = 6      32-bit BALANCED!
atan.out:01556:* Maximum relative error in F(x) = 8.05e-11 = 2.0**-33.53 \
                at x = 0.0000 for x on [0,(2 - sqrt(3))**2] with minimax \
                degree [4,2]  = 6      32-bit
...
atan.out:00049:* Maximum relative error in F(x) = 9.85e-02 = 2.0**-3.34  \
                at x = 0.0000 for x on [0,(2 - sqrt(3))**2] with minimax \
                degree [1,1]   = 2     USELESS BALANCED!
atan.out:00136:* Maximum relative error in F(x) = 6.18e-01 = 2.0**-0.69  \
                at x = 0.0000 for x on [0,(2 - sqrt(3))**2] with minimax \
                degree [2,0]   = 2     USELESS
```

For a given total degree, which determines the speed of the rational approximation, the one with balanced degrees, ⟨(ntotal/2)/(ntotal/2)⟩, usually offers the best accuracy, often as many as six bits better than a single polynomial of degree ntotal. However, the latter form avoids a costly division, so there is a tradeoff of accuracy against speed.

It is straightforward to copy the polynomial coefficients into a data header file (here, atan.h), wrapped by preprocessor conditionals that select code according to a compile-time-specified precision, P:

```
#if P <= 24
...
#elif P <= 32
...
#elif P <= 50
...
#elif P <= 240
...
#else
#error "atan() family not yet implemented for binary precision P > 240"
#endif
```

The polynomial expression can be copied into a shared algorithm header file (here, atanx.h), although the code that we actually use is slightly different:

```
else
{
    fp_t g, pg_g, qg;

    g = r * r;
    pg_g = POLY_P(p, g) * g;
    qg   = POLY_Q(q, g);
    result = FMA(r, pg_g / qg, r);
}
```

The POLY_P() and POLY_Q() macros hide the polynomial degrees, and are normally defined to expand inline to Horner forms. The FMA(x,y,z) macro encapsulates the computation x * y + z for reasons that are described later in Section 4.17 on page 85.

Not only does that give us flexibility in handling a wide range of hardware precisions, it also allows us to choose any one of the polynomial approximations for use with any hardware precision. That proved useful during development, because many of the polynomials had to be tediously retyped from tables in the Cody/Waite book. Having multiple approximations available while the code was being checked made it easier to find typos in the manually entered coefficients, separating that issue from the debugging of the algorithm implementations.

3.5 Polynomial fits with Maple

The Maple library numapprox contains several routines for computing rational polynomial approximations to a user-specified function, but we describe only two of them here. We discuss a third method later in Section 3.9 on page 43.

The call `pade(f, x = a, [m, n])` finds a Padé approximation of degree $\langle m/n \rangle$ to the function $f(x)$ about the point $x = a$, but gives no ability to specify the interval of interest.

In **Section 3.4** on page 28, we have already seen a simpler version of the minimax fitting function. The general form `minimax(f, x = a..b, [m, n], w, 'maxerror')` computes the best minimax rational approximation of degree $\langle m/n \rangle$ in the interval $[a, b]$ with respect to the positive weight function $w(x)$, and assigns the minimax error norm to the variable `maxerror`. The last two arguments are optional, but we supply them in the Maple programs that accompany that package, setting $w(x) = 1$, and we record the error estimate in comments that prefix the output polynomial coefficients. If $f(x)$ is nonzero on $[a, b]$, then we could use $w(x) = 1/f(x)$ to minimize the *relative*, rather than *absolute*, error. However, we have not done so. Only the last argument is modified by the function, so for reasons discussed earlier in **Section 3.4** on page 28, we use protecting quotes to ensure that its name, rather than its current value, is passed to the function.

The lack of interval specification in the `pade()` routine, and its frequent tendency to return failure, make it unsatisfactory. Almost all of the new polynomial approximations in the mathcw library have therefore been computed by `minimax()`.

Because we strive for high accuracy, the polynomials that satisfy $f(x) \approx \mathcal{P}(x)/\mathcal{Q}(x)$ must ensure that $f(x)$ and $\mathcal{P}(x)/\mathcal{Q}(x)$ be indistinguishable to machine precision, *even when the function is small*. Thus, the measure of the error must be the *relative error*, rather than the absolute error. However, in many cases, we have $f(x) = 0$ at one or the other endpoint of the interval of approximation, leading to a relative error of the form $0/0$, which is computationally undesirable.

The solution is to perturb the endpoint slightly, and modify the function definition to check for internal zero divisors. For example, the Maple program for one of the auxiliary functions required by the power elementary function has this code:

```
A      := 1.0e-20:
A_name := "0":

B      := 1/1024:
B_name := "1/1024":

F      := proc(x)
              if (evalb(x = 0)) then
                  return 0
              else
                  return (log((2 + sqrt(x))/(2 - sqrt(x))) - sqrt(x))/sqrt(x)
              end if
          end proc:
F_name := "(log((2 + sqrt(x))/(2 - sqrt(x))) - sqrt(x))/sqrt(x)":
```

The variables suffixed `_name` are what appear in the output reports, but the computational endpoint A is moved slightly away from the origin, and the function $F(x)$ handles a zero argument specially.

There is one further consideration that is useful for those functions that include the origin in the interval of approximation. In terms of the rational polynomials, the function is evaluated as $F(0) = \mathcal{P}(0)/\mathcal{Q}(0) = p_0/q_0$. The coefficients produced by the minimax approximation do not satisfy that relation numerically, so if zero is contained in the interval, the code sets `p[0] = F(0) * q[0]`. In particular, for functions that satisfy $F(0) = 0$, that forces $p_0 = 0$, and guarantees that the rational polynomial $\mathcal{P}(0)/\mathcal{Q}(0)$ evaluates to zero exactly. In addition, it allows p_0 to be dropped entirely from the polynomial evaluation, saving an unnecessary addition.

Plots of the auxiliary functions to which the rational polynomials are fit show them to be close to linear in the interval of approximation. Although that is necessary if the polynomials are to be of low degree, and thus, fast to evaluate, it is gratifying that high accuracy is possible, as demonstrated by the results in **Table 3.1** on the following page.

3.6 Polynomial fits with Mathematica

Well after work on this book and the mathcw library began, the release in 2008 of version 6 of the Mathematica symbolic-algebra system added support for function approximations. That facility provides an alternative to the Maple `minimax()` function, and allows revisiting fit attempts where Maple was unsuccessful.

Table 3.1: Accuracy of rational polynomials for computation of an auxiliary function for $\exp(x)$ for x in the interval $[0, (\ln(2)/2)^2] \approx [0, 0.12]$. Adding two to the total polynomial degree buys about ten more decimal digits of accuracy.

total polynomial degree	2	4	5	6	7	9	12	14	16	18	20	22	24
$-\log_{10}$(maximum relative error)	9	18	22	26	31	40	55	65	75	86	96	107	118
average accurate bits	33	59	73	88	103	133	182	216	251	286	321	357	394
worst-case accurate bits	29	56	69	85	99	131	180	213	248	282	318	354	391

Although the Mathematica functions are straightforward to use, more work is required to turn their output into text that can be copied unmodified into C source-code files. Because the facility is new, we present a detailed tutorial for it here. We start with the simple example given earlier of fitting the sine function on a modest interval.

The first task is to load the function approximation library into a Mathematica session:

```
% math
In[1]:= << FunctionApproximations‘
```

The double angle brackets are a convenient shorthand for reading text file input; the statement could also have been written in function form like this:

```
In[1]:= Get[ "FunctionApproximations‘" ]
```

The peculiar final back quote is a Mathematica *context* mark, similar to a slash in a Unix pathname. It is essential here, because it indicates that an entire subdirectory tree is to be read from somewhere inside the Mathematica installation. Failure to include the back quote produces an error:

```
In[1]:= Get[ "FunctionApproximations" ]
Get::noopen: Cannot open FunctionApproximations.
Out[1]= Failed
```

Interactive Mathematica sessions normally mark input and output as assignments to internal arrays called `In[]` and `Out[]`. That is convenient, because it allows reuse of earlier input and output expressions. For example, `In[3]` is the current *value* of the third input expression; to get its original form, use `InString[3]`.

In an interactive session, Mathematica normally shows the result of each input expression, but that display can be suppressed by suffixing the input with a semicolon.

Like the Lisp programming-language family, Mathematica is a *functional* language: it uses function calls, instead of statements that each have a unique syntax. Its built-in function names are rarely abbreviated, and by convention, are written with each word capitalized. Unlike many other modern languages, the underscore character is not allowed in variable and function names, because it is given another meaning that we introduce later.

The function that produces minimax fits has some options that we need, and we can display them like this:

```
In[2]:= Options[ MiniMaxApproximation ]

Out[2]= { Bias             -> 0,
>         Brake            -> {5, 5},
>         Derivatives      -> Automatic,
>         MaxIterations    -> 20,
>         WorkingPrecision -> MachinePrecision,
>         PrintFlag        -> False,
>         PlotFlag         -> False }
```

If we do not change the software floating-point working precision, Mathematica uses hardware floating-point arithmetic in the IEEE 754 64-bit binary format, corresponding to roughly 16 decimal digits:

```
In[3]:= $MachinePrecision

Out[3]= 15.9546
```

We now make our first attempt at a $\langle 3/3 \rangle$ fit of the sine function using default options:

```
In[4]:= rs = MiniMaxApproximation[ Sin[ x ], {x, {0, Pi/4}, 3, 3} ]

                                 1
Power::infy: Infinite expression -- encountered.               [twice]
                                 0.

Infinity::indet: Indeterminate expression 0. ComplexInfinity
encountered.                                                   [twice]

LinearSolve::inf: Input matrix contains an infinite entry.

                                             Pi
Out[4]= MiniMaxApproximation[Sin[x], {x, {0, --}, 3, 3}]
                                             4
```

The internal failure results in the right-hand side being returned unevaluated.

The complaints of infinities are surprising, because the function $\sin(x)$ is smooth and well-behaved with values in $[0, \sqrt{\frac{1}{2}}]$ on the fit interval. However, the function's documentation notes that the expression to be approximated must not have a zero on the fit interval. Instead, we use endpoint perturbation, the same trick that solves similar problems with Maple's fitting functions:

```
In[5]:= rs = MiniMaxApproximation[ Sin[x], {x, {2^(-53), Pi/4}, 3, 3}]

MiniMaxApproximation::conv: Warning: convergence was not complete.

                    -16
Out[5]= {{1.11022 10   , 0.00031017, 0.0534226, 0.198256, 0.395484,

>  0.591416, 0.73365, 0.785398},

             -24                    2             3
   6.58177 10    + 1. x + 0.0297665 x  - 0.119725 x              -8
> {-------------------------------------------------, -2.91458 10  }}
                                 2             3
    1 + 0.0297639 x + 0.0469804 x  + 0.00475105 x
```

The fit is partially successful, but there is a warning to deal with before we explain the output. We increase the number of internal iterations beyond the default of 20:

```
In[6]:= rs = MiniMaxApproximation[ Sin[x], {x, {2^(-53), Pi/4}, 3, 3},
                                    MaxIterations -> 40 ]
                    -16
Out[6]= {{1.11022 10   , 0.000103875, 0.0531916, 0.19807, 0.395361,

>  0.591355, 0.733634, 0.785398},

             -24                    2             3
   6.52019 10    + 1. x + 0.0297547 x  - 0.119723 x              -8
> {-------------------------------------------------, -2.91986 10  }}
                                 2             3
     1 + 0.029752 x + 0.0469828 x  + 0.00474939 x
```

The fit is now complete, and successful. The returned approximation is a braced list of two elements, themselves both lists. The first list element is a list of x values where the magnitude of the relative error in the fit reached a local maximum. The second list has two elements: the desired rational polynomial approximation, and the maximum relative error on the interval.

We notice that the leading coefficient in the numerator is tiny, but nonzero, so the rational polynomial reduces to that value at $x = 0$, instead of the correct value, $\sin(0) = 0$. It is therefore a good idea to instead produce a fit to a function, $\sin(x)/x$, that is nonzero over the fit interval:

```
In[7]:= rs = MiniMaxApproximation[ Sin[ x ] / x,
                                   {x, {2^(-53), Pi/4}, 3, 3},
                                   MaxIterations -> 40 ]
                     -16
Out[7]= {{1.11022 10    , 0.0408088, 0.15429, 0.315766, 0.491015,

>  0.644846, 0.748815, 0.785398},
                                2            3
   1. - 0.373319 x - 0.117448 x  + 0.0450069 x
  {-------------------------------------------,
                               2            3
   1 - 0.37332 x + 0.0492233 x  - 0.0172495 x

             -9
>  2.10223 10  }}
```

An even better choice is the smoother function that we used earlier, $(\sin(\sqrt{x}) - \sqrt{x})/(x\sqrt{x})$. However, that does not introduce new features, so we omit it here.

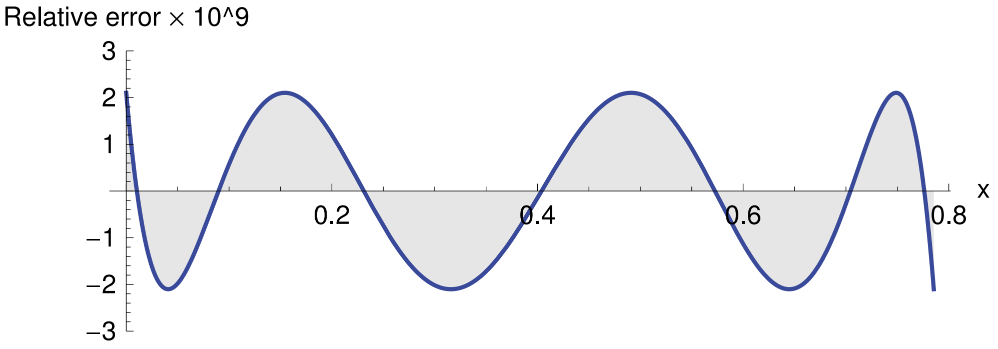
We now need to pick apart the returned lists to recover their separate elements. In Mathematica, if `list` is a list, then we can extract its i-th element with a function call, `Part[list, i]`, or with more compact double-bracket notation, `list[[i]]`. Lists are indexed $1, 2, 3, \ldots$ from the first element, and $-1, -2, -3, \ldots$ from the last element. We have lists of lists, so we can extract the j-th element of the i-th sublist with `Part[Part[list, i], j]`, or `Part[i, j]`, or, more easily, with `list[[i, j]]`. We extract the rational polynomial, and its error, like this:

```
In[8]:= rsratpoly = rs[[2,1]]

                                     2            3
         1. - 0.373319 x - 0.117448 x  + 0.0450069 x
Out[8]= -------------------------------------------
                                    2            3
         1 - 0.37332 x + 0.0492233 x  - 0.0172495 x

In[9]:= rsmaxerr = rs[[2,2]]

                    -9
Out[9]= 2.10223 10
```

The `Plot[]` function makes it easy to get a simple screen plot of an expression, but more work is needed to enhance it with axis labels, shading, and conversion to PostScript suitable for inclusion in a typeset document. The input

```
In[10]:= Export[ "sin-x-by-x-3-3.eps",
      Plot[ (1 - rsratpoly / ( Sin[x] / x )) * 10^9,
            {x, 0, Pi / 4},
            AspectRatio     -> 1 / 3,
            AxesLabel       -> { "x", "Relative error \[Times] 10^9" },
            BaseStyle       -> { FontFamily -> "Helvetica",
                                 FontSize   -> 8 },
            Filling         -> Axis,
            FillingStyle    -> { GrayLevel[0.90] },
            PerformanceGoal -> "Quality",
            PlotRange       -> {-3, 3},
            PlotStyle       -> { Thickness[0.005] }
          ],
      "EPS"
      ]
```

Figure 3.3: Error in minimax $\langle 3/3 \rangle$ rational-polynomial fit of $\sin(x)/x$ for x on $[0, \pi/4]$.



produces the plot shown in **Figure 3.3**. Notice that the extrema have equal height (the *minimax* property), and that two of them fall at the endpoints of the fit interval.

Next, we split the rational polynomial into its two parts:

```
In[11]:= rsnum = Numerator[ rsratpoly ]

                                     2            3
Out[11]= 1. - 0.373319 x - 0.117448 x  + 0.0450069 x

In[12]:= rsden = Denominator[ rsratpoly ]

                                    2            3
Out[12]= 1 - 0.37332 x + 0.0492233 x  - 0.0172495 x
```

We also need to be able to extract polynomial coefficients by specifying the variable and its integer power:

```
In[13]:= Coefficient[ rsnum, x, 3 ]

Out[13]= 0.0450069
```

The next piece of Mathematica function wizardry that we need is a way to make the polynomial expressions easy to use in another programming language, such as C or Fortran:

```
In[14]:= CForm[ rsnum ]

Out[14]//CForm=
   0.999999997897774 - 0.37331936378440406*x -
    0.11744824060104601*Power(x,2) + 0.045006876402198616*Power(x,3)

In[15]:= FortranForm[ rsnum ]

Out[15]//FortranForm=
         0.999999997897774 - 0.37331936378440406*x -
     -   0.11744824060104601*x**2 + 0.045006876402198616*x**3
```

The unfortunately chosen leading minus sign on the last line is not part of the expression, but instead is a Fortran statement-continuation character that appears in column six of a source line in the traditional 80-character punched-card image.

That output carries a surprise: the coefficients were previously displayed with fewer digits, but now they appear with a precision matching the underlying hardware precision.

The power functions in the output are a poor way to compute the polynomials, so we switch to nested Horner form:

```
In[16]:= CForm[ HornerForm[ rsnum ] ]

Out[16]//CForm=
   0.999999997897774 + x*(-0.37331936378440406 +
       (-0.11744824060104601 + 0.045006876402198616*x)*x)

In[17]:= FortranForm[ HornerForm[ rsnum ] ]

Out[17]//FortranForm=
         0.999999997897774 + x*(-0.37331936378440406 +
     -      (-0.11744824060104601 + 0.045006876402198616*x)*x)
```

For one-of-a-kind fits, we have now presented the minimal set of Mathematica function calls that are needed to convert a rational polynomial fit to source code in a conventional programming language.

In the mathcw library, however, we require considerably more, because we need to compute the fits for arbitrary specified precision, record the details of the fit in source-code comments, supply the polynomial coefficients with suitable precision suffixes, and hide them in array initializers.

Mathematica provides only limited control over numeric output: it drops decimal points from numbers without a fractional part, and it discards trailing zeros in fractions. We therefore provide polyfit.mth, a file of Mathematica enhancements. That file starts with our earlier In[1] expression to load the function-approximation library, then defines a wrapper function to improve output appearance by justifying a string into a field of fixed width:

```
StringJustifyRight[ s_, targetwidth_ : 0 ] :=
  Module[ { t },
         t = s;
         While[ StringLength[ t ] < targetwidth, t = " " <> t ];
         t
       ]
```

Notice the unusual syntax: arguments in the function declaration on the left require trailing underscores that mark them as *pattern holders*, but those underscores are absent from the definition on the right. The colon-equals operator is a delayed assignment that prevents evaluation of the right-hand side until the function is actually called.

In the function argument list, the colon indicates a default value for an argument. That argument may therefore be omitted in calls to our function, if the default is acceptable.

The Module[] function is a convenient way to supply a statement block for the right-hand side. Its first argument is a comma-separated braced list of local variables, and its remaining arguments are semicolon-separated statements.

The StringJustifyRight[] function is crude, but clearly correct: it repeatedly prefixes a space to a copy of the source string while the target string is shorter than the desired length. The While[] function loops, evaluating its second argument, the body, only when the first argument evaluates to True.

The paired-angle-bracket operator, <>, is Mathematica shorthand for string concatenation. The statement in which it appears could have been written in functional notation as

```
t = StringJoin[ " ", t ]
```

The final expression, t, in the body is the function result. That fact could be emphasized by writing it instead as Return[t], but that function is rarely used.

We can provide a documentation string for our function like this:

```
StringJustifyRight::usage =
"StringJustifyRight[string, width] returns a copy of string
right-justified with space padding on the left, if needed, to
make it at least width characters long."
```

An input expression of the form ?StringJustifyRight displays that documentation. We supply usage strings for all of the functions and global variables in polyfit.mth, but we omit them in the rest of this section.

Comments in Mathematica are delimited by (* ... *), and those delimiters can be nested, allowing a large block of commented code to itself be turned into a comment if needed.

We format numbers with the help of this function:

```
CFormat[x_, nfrac_, digits_ : Digits] :=
  Block[ { d, e, f, s, t, scale },

    If[ IntegerQ[ digits ] && digits > 0,
        d = digits,
        d = MiniMaxDefaultDigits ];

    scale = If[ (10^(-6) < Abs[ x ]) && (Abs[ x ] < 10^6), 10^25, 1 ];
    s = StringSplit[ ToString[ CForm[ N[ scale * x, d ] ] ],
                     "." | "e" ];

    If[ Length[ s ] == 1, s = AppendTo[s, "0"] ];

    If[ Length[ s ] == 2, s = AppendTo[s, "0"] ];

    e = If[ scale == 1,
            s[[3]],
            ToString[ToExpression[ s[[3]] ] - 25] ];

    (* force two digit exponents for better table alignment *)
    If[ (StringLength[ e ] == 2) && (StringTake[ e, 1 ] == "-"),
         e = "-0" <> StringTake[ e, -1] ];

    If[ StringLength[ e ] == 1, e = "0" <> e];

    (* force explicit plus sign *)
    If[ StringTake[ e, 1 ] != "-", e = "+" <> e ];

    f = StringTake[ s[[2]], Min[ StringLength[ s[[2]] ], nfrac ] ];

    While[ StringLength[f] < nfrac, f = f <> "0" ];

    t = s[[1]] <> "." <> f <> "e" <> e;
    t
        ]
```

This time, we use a `Block[]` instead of a `Module[]`, just to show an alternative way of grouping statements. The differences between the two are subtle, and not relevant here. The default supplied by our global variable `Digits` defines the number of digits wanted in the output values, because that is distinct from the `WorkingPrecision` or `$MachinePrecision` used to compute them.

The `If[]` function supplies a corrected value for the `digits` argument, in case it is not numeric, or is not a positive integer. Like the C-language short-circuiting ternary expression `e ? a : b`, the `If[]` function evaluates only *one* of the second and third arguments, depending on whether the first argument evaluates to `True` or `False`.

The `IntegerQ[]` function tests whether its argument is an integer or not. The fallback value stored in our global variable `MiniMaxDefaultDigits` is 36, which is sufficient for both IEEE 754 128-bit binary and decimal formats.

Because the built-in `CForm[]` function uses exponential form only for numbers outside the range $[10^{-5}, 10^{+5}]$, we choose a scale factor to ensure exponential form. We then split the output of `CForm[]` into a three-element list with the integer, fraction, and exponent. We then do some further cleanup to ensure that the exponent has at least two digits, and, if needed, we add trailing zeros to pad the fraction to the requested size. Finally, we rejoin the pieces and return the result.

A polynomial can be defined by a constant array of coefficients that is produced with this function:

```
CPoly[ poly_, var_, name_ : "p", digits_ : Digits ] :=
  Module[ { d, k, pagewidth },

          If[ IntegerQ[ digits ] && digits > 0,
              d = digits,
              d = MiniMaxDefaultDigits ];

          (* prevent gratuitous linewrapping *)
          pagewidth = Options[$Output, PageWidth];
          SetOptions[$Output, PageWidth -> Infinity];

          Print[ "static const fp_t ", name, "[] = {" ];

          For[ k = 0, k <= Exponent[ poly, var ], k = k + 1,
               Print[ "    FP( ",
                      StringJustifyRight[ CFormat[
                        Coefficient[ poly, var, k ], d ], d + 7],
                      " )",
                      If[ k == Exponent[ poly, var ], "", "," ],
                      "\t/* ", k, " */" ] ];

          Print[ "};\n" ];
          SetOptions[$Output, pagewidth];
        ]
```

The function begins by ensuring that `d` has a positive integer value. It then saves the current page-width limit, and resets it to infinity to prevent unwanted line wrapping; the saved value is restored just before the function returns.

Each `Print[]` call produces a single output line, so we often need to supply multiple arguments to that function. Mathematica recognizes most of the C-style escape sequences in strings, so we use `\n` (newline) and `\t` (tab) for additional spacing control.

The `Exponent[poly, x]` function returns the maximum power of `x` in the polynomial.

The counted loop function has the form `For[`*`init`*`, `*`test`*`, `*`incr`*`, `*`body`*`]`, similar to the C-language loop statement `for (`*`init`*`; `*`test`*`; `*`incr`*`) `*`body`*. Notice that the Mathematica loop has comma separators, because the four fields are just function arguments. If the loop body requires more than one statement, it can be given as a list of semicolon-separated expressions, or if more control and loop-local variables are needed, with a `Block[]` or `Module[]` wrapper.

Another function makes it convenient to produce coefficient tables for both numerator and denominator of the rational polynomial with a single call, with optional further scaling of the coefficients:

```
CRatPoly[ ratpoly_, var_,
          numname_ : "p", denname_ : "q", scale_ : 1 ] :=
  Module[ { },
          CPoly[ scale *   Numerator[ ratpoly ], var, numname ];
          CPoly[ scale * Denominator[ ratpoly ], var, denname ]
        ]
```

We also need to produce a lengthy comment header that records data about the fit:

```
CRatComment[ ratapprox_, var_, funname_,
             numname_ : "p", denname_ : "q" ] :=
  Module[ { },
          Print[ "" ];
          Print[ "/***" ];
          Print[ " ***\tF(", var, ") = ", funname ];
          Print[ " ***\tMaximum relative error in F(", var, ") = ",
                N[ CForm[ ratapprox[[2,2]] ], 4 ] ];
          Print[ " ***\tfor fit with minimax degree [ ",
                 Exponent[   Numerator[ ratapprox[[2,1]] ], var], ", ",
                 Exponent[ Denominator[ ratapprox[[2,1]] ], var], " ]"
               ];
          Print[ " ***\tfor ",
                 var, " on [ ",
                 CForm[ N[ ratapprox[[1, 1]], 5 ] ], ", ",
                 CForm[ N[ ratapprox[[1,-1]], 5 ] ], " ]"
```

```
              ];
          Print[ " ***/\n" ];
          Print[ "#define POLY_", ToUpperCase[ numname ],
                 "(p, ", var, ")\tPOLY_",
                 Exponent[ Numerator[ ratapprox[[2,1]] ], var],
                 "(p, ", var, ")"
              ];
          Print[ "#define POLY_", ToUpperCase[ denname ],
                 "(q, ", var, ")\tPOLY_",
                 Exponent[ Denominator[ ratapprox[[2,1]] ], var],
                 "(q, ", var, ")"
              ];
          Print[ "" ]
        ]
```

The rational-polynomial list structure does not explicitly record the fit-variable range, but we can recover a close approximation to it from the initial and final elements of the first list of locations of maximum errors.

We provide a helper function, modeled after a similar feature of our Maple code, to select a suitable coefficient-dependent scale factor:

```
(*
** Return a scale factor for a rational polynomial.  Scaling is
** determined by normoption:
**    0      larger of lowest-order coefficients is 1/2
**    1      highest-order numerator coefficient is 1
**    2      highest-order denominator coefficient is 1
**    3      lowest-order numerator coefficient is 1
**    4      lowest-order denominator coefficient is 1
**    5      lowest-order numerator coefficient is 1/2
**    6      lowest-order denominator coefficient is 1/2
**
** Out-of-range values of normoption are treated as 0.
*)

RatScale[ ratpoly_, var_, normoption_ : 0 ] :=
  Module[ { },
    Switch[ normoption,
      0, 1 / (2 * Max[Abs[Coefficient[  Numerator[ratpoly], var, 0]],
                      Abs[Coefficient[Denominator[ratpoly], var, 0]]]),
      1, 1 / Coefficient[ Numerator[ ratpoly ],   var,
                         Exponent[ Numerator[ ratpoly ] ] ],
      2, 1 / Coefficient[ Denominator[ ratpoly ], var,
                         Exponent[ Denominator[ ratpoly ] ] ],
      3, 1 / Coefficient[ Numerator[ ratpoly ],   var, 0],
      4, 1 / Coefficient[ Denominator[ ratpoly ], var, 0],
      5, 1 / (2 * Coefficient[ Numerator[ ratpoly ], var, 0]),
      6, 1 / (2 * Coefficient[ Denominator[ ratpoly ], var, 0]),
      _, 1 / (2 * Max[Abs[Coefficient[  Numerator[ratpoly], var, 0]],
                      Abs[Coefficient[Denominator[ratpoly], var, 0]]])
        ]
      ]
```

The `Switch[]` function is similar to a C `switch` statement: its first argument is a case selector, and each subsequent pair of arguments is a case number, and an expression to evaluate for that case. The underscore in the final case is a pattern holder that matches anything: it corresponds to the `default` block in a C `switch` statement. Unlike the C `switch` statement, there is no fall-through from one case to another in the `Switch[]` function.

The reason for the different scaling options is that we often wish to control the form of the coefficients for better accuracy. We usually want one of the low-order coefficients in the numerator or denominator to be an exactly representable floating-point value, such as 1.0 or 0.5.

A final function reports the fit as C-language code:

```
CRatData[ ratapprox_, var_, funname_, numname_ : "p", denname_ : "q",
           normoption_ : 0 ] :=
  Module[ { },
          CRatComment[ ratapprox, var, funname, numname, denname ];
          CRatPoly[ ratapprox[[2,1]], var, numname, denname,
                    RatScale[ ratapprox[[2,1] ], var, normoption ] ]
        ]
```

Here is a fresh session that shows the minimal Mathematica input needed to produce a low-order rational polynomial fit to a sample function:

```
% math
In[1]:= << polyfit.mth
In[2]:= re = MiniMaxApproximation[ (Exp[x] - 1) / x,
                                   {x, {10^(-10), 1/2}, 2, 3},
                                   WorkingPrecision -> 20];
In[3]:= Digits = 9;
In[4]:= CRatData[ re, x, "(exp(x) - 1) / x" ]

 /***
  ***    F(x) = (exp(x) - 1) / x
  ***    Maximum relative error in F(x) = -8.543e-11
  ***    for fit with minimax degree [2, 3]
  ***    for x on [1.e-10, 0.5]
  ***/

 #define POLY_P(p, x)    POLY_2(p, x)
 #define POLY_Q(q, x)    POLY_3(q, x)

 static const fp_t p[] = {
     FP(  5.000000000e-01 ),     /* 0 */
     FP(  5.409484790e-03 ),     /* 1 */
     FP(  8.384744570e-03 )      /* 2 */
 };

 static const fp_t q[] = {
     FP(  5.000000000e-01 ),     /* 0 */
     FP( -2.445905090e-01 ),     /* 1 */
     FP(  4.734653410e-02 ),     /* 2 */
     FP( -3.740454220e-03 )      /* 3 */
 };
```

3.7 Exact polynomial coefficients

Minimax polynomials are computed in high precision, but the coefficients are truncated to machine precision for subsequent polynomial evaluation. The truncations cause the computed polynomial to suffer a small error that we would like to avoid. The Cody/Waite solution to that problem is to use the polynomial as a small correction to a larger exact term.

Researchers have recently considered the problem of finding polynomial coefficients that are constrained to be exact machine numbers and it is possible that symbolic-algebra systems may someday provide software for minimax approximations under those constraints [BH07], [MBdD^{+}10, Section 11.4.3]. We could, of course, truncate computed coefficients to exactly representable values, and then recompute the fitting-error estimates before selecting particular fits for use in our code.

The file `fitrndx.map` extends our support for polynomial fits with a new function `pq_approx_rounded()` that uses the global Maple variables `BASE` and `PRECISION` to constrain the coefficients. Finding suitable constrained polynomials is harder, for we now have the base and precision, as well as degrees, to vary. However, given an existing set of unconstrained polynomial fits, it is usually possible to guess the required degrees for a constrained fit, because they should rarely be more than one or two higher.

Another idea that may be worth investigating is to restrict the coefficient in the term $c_1 x$ to have just one nonzero decimal digit, or two or three nonzero bits, so that from a tenth to an eighth or a fourth of the products in the term are exact. However, we leave that idea for future refinements of the library.

3.8 Cody/Waite rational polynomials

We noted earlier in **Section 3.4** on page 28 that the Cody/Waite book does not document the polynomials that they use for rational approximations. Because the `mathcw` package extends their work to higher precision, the polynomials had to be reverse engineered from the algorithm descriptions, and then validated numerically. **Table 3.2** on the next page summarizes the results of that tedious exercise.

3.9 Chebyshev polynomial economization

Chebyshev[4] polynomials of the first kind of degree n, written as $T_n(u)$, have the useful property for function approximation that their values lie in the range $[-1, +1]$ for u in $[-1, +1]$, and they can be proved to give the smallest maximum error in a fit to a smooth function over that interval by a single polynomial of fixed degree. The first few polynomials, their recurrence relation, and values for special arguments, are shown in **Table 3.3** on page 45, and graphed in **Figure 3.4** on page 46.

Although the recurrence relation is easy to apply, there is also a general summation formula for the Chebyshev polynomials:

$$T_n(u) = \frac{n}{2} \sum_{k=0}^{\lfloor n/2 \rfloor} \frac{(-1)^k}{n-k} \binom{n-k}{k} (2u)^{n-2k}$$

The notation $\binom{n}{k}$ represents a *binomial coefficient*, and can be pronounced *binomial n over k*. It is the coefficient of the x^k term in the polynomial expansion of $(1+x)^n$, and has the value $n!/(k!\,(n-k)!)$. It is also the number of ways of choosing k objects from a set of n objects, and in that context, can be pronounced *n choose k*. The upper limit of the sum is the largest integer that does not exceed $n/2$, using a notation that we describe later in **Section 6.7** on page 136.

Perhaps surprisingly, given the simple forms of the Chebyshev polynomials, there are closed-form representations in terms of trigonometric and hyperbolic functions whose properties we describe in **Chapter 11** and **Chapter 12**:

$$T_n(u) = \begin{cases} (-1)^n \cosh(n \operatorname{acosh}(-u)), & u \text{ in } (-\infty, -1), \\ \cos(n \operatorname{acos}(u)), & u \text{ in } [-1, +1], \\ \cosh(n \operatorname{acosh}(u)), & u \text{ in } (+1, +\infty). \end{cases}$$

Manipulation of the trigonometric form shows that all of the zeros of $T_n(u)$ lie *inside* the interval $[-1, +1]$ at these locations:

$$r_k = \cos((k - \tfrac{1}{2})\pi/n), \qquad T_n(r_k) = 0, \qquad k = 1, 2, \ldots, n.$$

[4] Pafnuty Lvovich Chebyshev [Пафнутий Львович Чебышёв] (1821–1894) was a Russian mathematician noted for his work on the theory of prime numbers, on probability and statistics, and on function approximation. A crater on the Moon, and Asteroid 2010 Chebyshev (1969 TL4), are named in his honor.

His name is variously transliterated from the Cyrillic alphabet into Latin scripts as *Chebyshev*, *Chebychev*, *Cebycev*, *Czebyszew*, *Tchebycheff*, *Tchebychev*, *Tschebyscheff*, and about *two hundred* other variants: see `http://mathreader.livejournal.com/9239.html` for a list. The first two are the most common forms in English-language publications, although *Chebyshóf*, with the last syllable stressed, is closer to the Russian pronunciation. However, many modern Russian mathematics texts spell his name as Чебышев, which matches the commonest English form, *Chébyshev*, with the first syllable stressed. The German transliterations begin with *T*, which perhaps explains the use of that letter for the Chebyshev polynomials of the first kind.

Table 3.2: Low-level polynomial approximations. The second column indicates whether the algorithm is from the Cody/Waite book (CW), or from the mathcw library (MCW). If an interval is not specified, then the function is computed directly from the functions listed in the last column.

Function	Source	Interval	Approximating function $\mathcal{R}(x)$
`acos(x)`	CW	$[0, 1/4]$	$(\mathrm{asin}(\sqrt{x}) - \sqrt{x})/(x\sqrt{x})$
`acosh(x)`	CW		$\log(x), \mathrm{log1p}(x), \sqrt{x}$
`asin(x)`	CW	$[0, 1/4]$	$(\mathrm{asin}(\sqrt{x}) - \sqrt{x})/(x\sqrt{x})$
`asinh(x)`	CW		$\log(x), \mathrm{log1p}(x), \sqrt{x}$
`atanh(x)`	CW		$\mathrm{log1p}(x)$
`cbrt(x)`	MCW	$[1/2, 1)$	Newton–Raphson iteration
`cos(x)`	CW	$[0, \pi/2]$	$(\sin(\sqrt{x}) - \sqrt{x})/(x\sqrt{x})$
`cosh(x)`	CW		$\exp(x)$
`cot(x)`	CW	$[0, \pi/2]$	$\tan(\sqrt{x})/\sqrt{x}$
`erf(x)`	MCW	$[0, 27/32)$	$(\mathrm{erf}(\sqrt{x}) - \sqrt{x})/\sqrt{x}$
`erf(x)`	MCW	$[27/32, 5/4)$	$\mathrm{erf}(x) - 3456/4096$
`erf(x)`	MCW	$[5/4, 1/0.35)$	$\log(\mathrm{erfc}(1/\sqrt{x})/\sqrt{x}) + 1/x + 9/16$
`erf(x)`	MCW	$[0.35, 200)$	$\log(\mathrm{erfc}(1/\sqrt{x})/\sqrt{x}) + 1/x + 9/16$
`erfc(x)`	MCW	$[0, 27/32)$	$(\mathrm{erf}(\sqrt{x}) - \sqrt{x})/\sqrt{x}$
`erfc(x)`	MCW	$[27/32, 5/4)$	$\mathrm{erf}(x) - 3456/4096$
`erfc(x)`	MCW	$[5/4, 1/0.35)$	$\log(\mathrm{erfc}(1/\sqrt{x})/\sqrt{x}) + 1/x + 9/16$
`erfc(x)`	MCW	$[0.35, 200)$	$\log(\mathrm{erfc}(1/\sqrt{x})/\sqrt{x}) + 1/x + 9/16$
`exp(x)`	CW	$[-\log(2)/2, +\log(2)/2]$	$(1/\sqrt{x})(e^{\sqrt{x}} - 1)/(e^{\sqrt{x}} + 1)$
`exp2(x)`	MCW	$[0, 1/4]$	$(1/\sqrt{x})(2^{\sqrt{x}} - 1)/(2^{\sqrt{x}} + 1)$
`exp10(x)`	MCW	$[0, (\log_{10}(2)/2)^2]$	$(1/\sqrt{x})(10^{\sqrt{x}} - 1)/(10^{\sqrt{x}} + 1)$
`hypot(x,y)`	MCW		$\sqrt{x}$
`lgamma(x)`	MCW		$\Gamma(x), \log(x)$
`log(x)`	CW	$[0, (2(\sqrt{\frac{1}{2}} - 1)/(\sqrt{\frac{1}{2}} + 1))^2]$	$(\log((2 + \sqrt{x})/(2 - \sqrt{x})) - \sqrt{x})/(x\sqrt{x})$
`log1p(x)`	MCW		$\log(x)$
`log2(x)`	MCW		$\log(x)$
`log10(x)`	CW		$\log(x)$
`pow(x,y)`	CW	$[0, 1/1024]$	$(\log((2 + \sqrt{x})/(2 - \sqrt{x})) - \sqrt{x})/\sqrt{x}$
`pow(x,y)`	CW	$[-1/16, 0]$	$2^x - 1$
`psi(x)`	MCW	$[1, 2]$	$\psi(x)$
`psiln(x)`	MCW		$\psi(x) - \log(x)$
`rsqrt(x)`	MCW	$[1/2, 1)$	Newton–Raphson iteration
`sin(x)`	CW	$[0, \pi/2]$	$(\sin(\sqrt{x}) - \sqrt{x})/(x\sqrt{x})$
`sinh(x)`	CW	$[0, 1/4]$	$(\sinh(\sqrt{x}) - \sqrt{x})/(x\sqrt{x})$
`sqrt(x)`	CW	$[1/2, 1)$	Newton–Raphson iteration
`tan(x)`	CW	$[0, \pi/2]$	$\tan(\sqrt{x})/\sqrt{x}$
`tanh(x)`	CW	$[0, (\log(3)/2)^2]$	$(\tanh(\sqrt{x}) - \sqrt{x})/(x\sqrt{x})$
`tgamma(x)`	MCW	$[1, 2]$	$\Gamma(x)$

In addition, inside that same interval, the polynomial $T_n(u)$ has $n + 1$ extrema (minima and maxima) of *equal* magnitude:

$$e_k = \cos(k\pi/n), \qquad T_n(e_k) = \pm 1, \qquad k = 0, 1, 2, \ldots, n.$$

Outside the interval $[-1, +1]$, for $n > 0$, the polynomials $T_n(u)$ rise rapidly toward $\pm\infty$ as $2^{n-1}u^n$.

The Chebyshev polynomials are *orthogonal* in the sense of both a continuous integral with a specific weight func-

Table 3.3: Chebyshev polynomials, recurrence relation, and special values.

$\boldsymbol{T_0(u) = 1},$	$T_n(-1) = (-1)^n,$
$\boldsymbol{T_1(u) = u},$	$T_n(+1) = +1,$
$T_2(u) = 2u^2 - 1,$	$T_n(-u) = (-1)^n T_n(u),$
$T_3(u) = 4u^3 - 3u,$	$T_{2n}(0) = (-1)^n,$
$T_4(u) = 8u^4 - 8u^2 + 1,$	$T_{2n+1}(0) = 0,$
$T_5(u) = 16u^5 - 20u^3 + 5u,$	$T_{3n}(-\frac{1}{2}) = +1,$
$T_6(u) = 32u^6 - 48u^4 + 18u^2 - 1,$	$T_{3n}(+\frac{1}{2}) = (-1)^n,$
$T_7(u) = 64u^7 - 112u^5 + 56u^3 - 7u,$	$T_{3n+1}(-\frac{1}{2}) = -\frac{1}{2},$
$T_8(u) = 128u^8 - 256u^6 + 160u^4 - 32u^2 + 1,$	$T_{3n+1}(+\frac{1}{2}) = (-1)^n/2,$
$T_9(u) = 256u^9 - 576u^7 + 432u^5 - 120u^3 + 9u,$	$T_{3n+2}(-\frac{1}{2}) = -\frac{1}{2},$
$T_{10}(u) = 512u^{10} - 1280u^8 + 1120u^6 - 400u^4 + 50u^2 - 1,$	$T_{3n+2}(+\frac{1}{2}) = (-1)^{n+1}/2,$
$\boldsymbol{T_{n+1}(u) = 2uT_n(u) - T_{n-1}(u)}, \qquad n > 0.$	

tion, and a discrete sum:

$$\int_{-1}^{+1} \frac{T_m(u)T_n(u)}{\sqrt{1-u^2}}\,du = \begin{cases} 0, & m \neq n, \\ \pi/2, & m = n, m > 0, \\ \pi, & m = n = 0, \end{cases}$$

$$\sum_{k=1}^{N} T_m(r_k)T_n(r_k) = \begin{cases} 0, & m \neq n, \\ m/2, & m = n, m > 0, \\ m, & m = n = 0. \end{cases}$$

In the sum, the r_k values are the roots of $T_N(u)$.

If $f(x)$, the function to be represented by a Chebyshev expansion, is defined on an arbitrary finite interval $[a, b]$, then a variable transformation is needed to ensure that u moves from -1 to $+1$ as x moves from a to b:

$$u = \frac{2x - (b + a)}{b - a}.$$

In practice, it is desirable to choose a and b carefully, so that rounding error is minimized in the conversion of x to u, even if that means widening the interval, and increasing the length of the Chebyshev expansion. If $b - a$ can be made a power of the base, then the division can be replaced by an exact multiplication.

For a semi-infinite interval $[a, +\infty)$, with $a > 0$, a reciprocal transformation given by

$$u = 2(a/x)^p - 1, \qquad p > 0,$$

does the job, and the power p can be chosen to give the shortest expansion. However, in that case, the power and the division by x both introduce unwanted rounding error in u, so it is advisable to compute the variable transformation in higher precision, if it is cheaply available.

With one of the given mappings from x to u, the $(n-1)$-th order Chebyshev approximation for a function takes the form

$$f(x) \approx \frac{2}{n}\sum_{k=0}^{n-1} c_k T_k(u) - \tfrac{1}{2}c_0, \qquad \text{for } x \text{ on } [a, b] \text{ and } u \text{ on } [-1, +1],$$

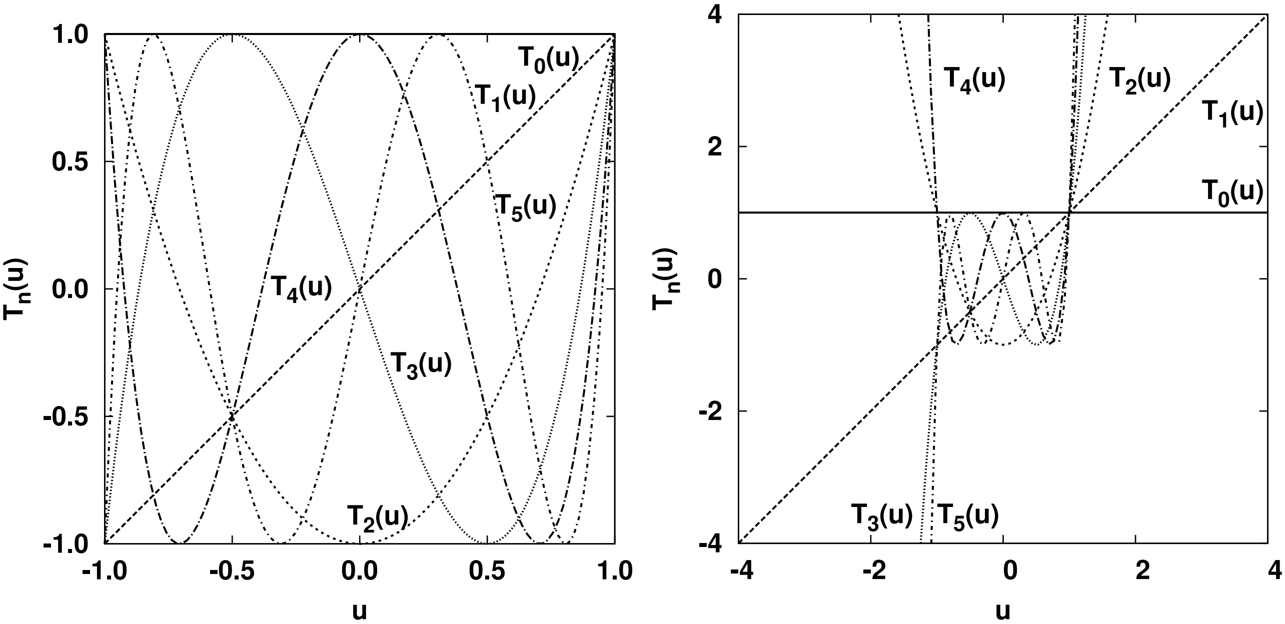


Figure 3.4: Two views of Chebyshev polynomials. The left plot shows the first six polynomials on the interval of interpolation, $[-1, +1]$. The right plot shows the rapid increase of all but $T_0(u)$ outside that interval.

where the coefficients are defined by

$$w_j = \frac{(j+1/2)\pi}{n}, \qquad u_j = \cos(w_j), \qquad x_j = x(u_j), \qquad \text{for } j = 0, 1, 2, \ldots, n-1,$$

$$c_k = \frac{2}{n}\sum_{j=0}^{n-1} f(x_j)T_k(u_j) = \frac{2}{n}\sum_{j=0}^{n-1} f(x_j)\cos(kw_j), \qquad \text{for } k = 0, 1, 2, \ldots, n-1.$$

To estimate the work required, notice that computation of all n coefficients needs just n values of each of w_j, u_j, x_j, and $f(x_j)$, but n^2 values of $\cos(kw_j)$. All n sums for the coefficients can then be found with n^2 multiplications and $n(n-1)$ additions. The total cost is roughly that of $\mathcal{O}(n^2)$ cosines. In exact arithmetic, the sum for $f(x)$ is *exact* at the n zeros of $T_n(u)$.

The irregularity at c_0 is a nuisance, and many authors use *sum-prime notation* to indicate that the first coefficient must be halved:

$$f(x) = {\sum_{k=0}^{n}}' c_k T_k(u) = \tfrac{1}{2}c_0 T_0(u) + c_1 T_1(u) + \cdots + c_n T_n(u), \qquad \text{and map } x \to u.$$

Because $T_0(u) = 1$, we can replace the computed c_0 by half its value, and henceforth work with $\sum c_k T_k(u)$. Our software, and Maple, do that, but be careful to check how the leading coefficient is defined and handled by other software.

Fits to Chebyshev polynomials are popular representations of functions in numerical software, because the expansion coefficients generally decrease with increasing order, and because the error introduced by truncating the expansion is the sum of the omitted terms. The decreasing terms in the omitted sum mean that the error is usually well approximated by the first omitted coefficient. In particular, that means that a *single* Chebyshev expansion can be used to fit a function to an accuracy that is easily tuned to the working precision of the computer, making it of considerable interest for software that is required to be portable across a broad range of computer architectures.

The criterion for a good fit is usually low relative error, so when Chebyshev coefficients are computed by our recipe, some experimentation with the value n may be needed. The worst-case relative error is given by

$\sum_{k=n}^{\infty} |c_k| / \sum_{k=0}^{\infty} |c_k|$, and it can be estimated reasonably well by truncating the sums after, say, $n+5$ terms. Thus, software needs to compute more c_k values than are expected to be needed, and then discard the higher-order coefficients that contribute negligibly to the desired relative error.

The Maple symbolic-algebra system makes it easy to compute Chebyshev fits of smooth continuous functions to any desired precision on any specified interval. Consider a polynomial approximation for the trigonometric tangent on the interval $[0, \pi/4]$. Here is how to find its Chebyshev representation to an accuracy suitable for the C-language data type `double` on many computers:

```
% maple
> with(numapprox):
> interface(prettyprint = 0, quiet = true, screenwidth = infinity):
> Digits := 20:
> a := 0:
> b := Pi/4:
> f := proc (x) return tan(x) end proc:
> c := chebsort(chebyshev(f(x), x = a .. b, 1.0e-16));
c := 0.45565217041349203532     * T(0,  (8/Pi) * x - 1)
   + 0.48946864364506139170     * T(1,  (8/Pi) * x - 1)
   + 0.42848348909108677383e-1  * T(2,  (8/Pi) * x - 1)
   + 0.10244343354792446844e-1  * T(3,  (8/Pi) * x - 1)
   + 0.14532687538039121258e-2  * T(4,  (8/Pi) * x - 1)
   + 0.27878025345169168580e-3  * T(5,  (8/Pi) * x - 1)
   + 0.44830182292260610628e-4  * T(6,  (8/Pi) * x - 1)
   + 0.79925727029127180643e-5  * T(7,  (8/Pi) * x - 1)
   + 0.13408477307061580717e-5  * T(8,  (8/Pi) * x - 1)
   + 0.23312607483191536196e-6  * T(9,  (8/Pi) * x - 1)
   + 0.39687542597331420243e-7  * T(10, (8/Pi) * x - 1)
   + 0.68406713195274508859e-8  * T(11, (8/Pi) * x - 1)
   + 0.11705047919536348158e-8  * T(12, (8/Pi) * x - 1)
   + 0.20114697552525542735e-9  * T(13, (8/Pi) * x - 1)
   + 0.34479027352473801402e-10 * T(14, (8/Pi) * x - 1)
   + 0.59189325851606462089e-11 * T(15, (8/Pi) * x - 1)
   + 0.10151982743517278224e-11 * T(16, (8/Pi) * x - 1)
   + 0.17421382436758336134e-12 * T(17, (8/Pi) * x - 1)
   + 0.29886998983518518519e-13 * T(18, (8/Pi) * x - 1)
   + 0.51281385648690992526e-14 * T(19, (8/Pi) * x - 1)
   + 0.87981510697734638763e-15 * T(20, (8/Pi) * x - 1)
   + 0.15095587991023089012e-15 * T(21, (8/Pi) * x - 1)
   + 0.25899583833164263991e-16 * T(22, (8/Pi) * x - 1)
```

The final argument passed to the `chebyshev()` function is the desired accuracy of the representation. The `chebsort()` function ensures that the polynomials are in ascending order. Truncation of the expansion to 10 terms produces a fit that is accurate to better than 4.8×10^{-8}, a value obtained by summing coefficients, starting with that of $T_{10}(8x/\pi - 1)$.

We can now use the Maple function `unapply()` to convert the expression into a function $q(x)$, and with the help of the operand-extraction function, `op()`, we create another function, $q_{10}(x)$, to compute just the first 10 terms:

```
> q := unapply(c, x):
> s := 0:
> for k from 1 to 10 do s := s + op(k,c) end do:
> q10 := unapply(s, x):
```

We can write the same code more compactly like this:

```
> q   := unapply(c, x):
> q10 := unapply(sum(op(k,c), k = 1 .. 10), x):
```

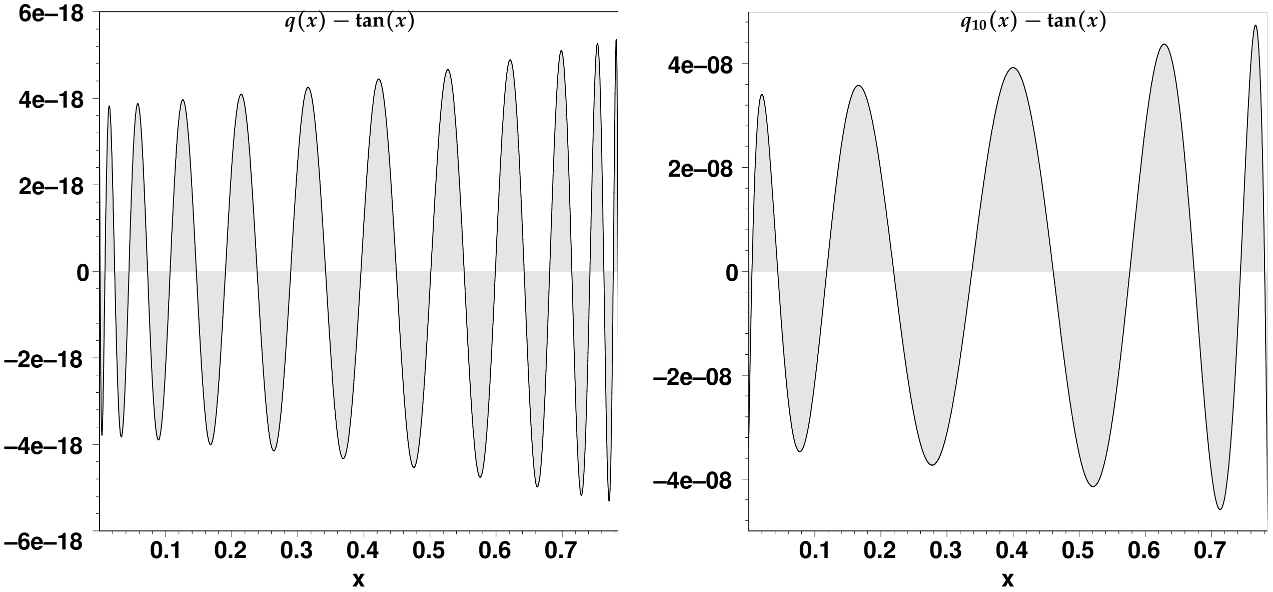


Figure 3.5: Absolute errors in Chebyshev approximations $q(x)$ and $q_{10}(x)$ to $\tan(x)$ on $[0, \pi/4]$. The number of extrema in each plot corresponds to the degree of the approximating Chebyshev polynomial fit.

At this point, we have two functions that compute sums involving $T(k,u)$, but we lack a definition of those Chebyshev polynomials. We load the needed definition from the orthopoly package, and then we increase the precision and compute the maximum error in each approximating function, and its location in the interval, with the `infnorm()` function:[5]

```
> with(orthopoly, T):

> Digits := Digits + 10:

> emax := infnorm(q(x) - f(x), x = a .. b, 'xmax'):
> printf("Maximum error = %.3g at x = %g\n", emax, xmax):
Maximum error = 5.36e-18 at x = 0.781785

> emax := infnorm(q10(x) - f(x), x = a .. b, 'xmax'):
> printf("Maximum error = %.3g at x = %g\n", emax, xmax):
Maximum error = 4.79e-08 at x = 0.785398
```

The largest error in $q_{10}(x)$ is close to the value that we predicted earlier by summing the truncated coefficients.

Figure 3.5 shows the differences between the approximations and the exact function. The oscillatory behavior of the errors, and their roughly even distribution over the interval of approximation, is typical of both Chebyshev and minimax polynomial fits.

3.10 Evaluating Chebyshev polynomials

The recurrence relation for the Chebyshev polynomials makes it easy to evaluate Chebyshev sums, without having to know the Chebyshev polynomial expansions in powers of u [Bro73]. In addition, the Chebyshev form is numerically more stable than the equivalent expansion in powers of u. Here is a C function to evaluate a Chebyshev expansion:

[5] The *infinity norm* of a set of numeric values is the value of largest magnitude.

```
double
echeb (double u, int n, const double c[/* n */])
{   /* evaluate a Chebyshev polynomial expansion, and return
       sum(k=0:n-1) c[k]*T(k,u), where u should be in [-1,1] */
    double b0, b1, b2, result, w;
    int k;

    if (n > 0)
    {
        b0 = 0.0;
        b1 = 0.0;
        w = u + u;

        for (k = n - 1; k >= 0; --k)
        {
            b2 = b1;
            b1 = b0;
            b0 = w * b1 - b2 + c[k];
        }

        result = 0.5 * (c[0] + b0 - b2);
    }
    else /* n <= 0 */
        result = nan("");

    return (result);
}
```

The total floating-point work is $n + 1$ multiplies and $2n + 3$ adds, and the terms are summed from smallest to largest, which is the order generally preferred for numerical stability. By comparison, evaluation of the nested Horner form of a polynomial of degree n takes only n multiplies and n adds, but it is then harder to estimate the error if the Horner form is truncated.

Closer examination of the code in `echeb()` shows that the first two iterations can be optimized away to avoid useless arithmetic with the initial zero values. A second version reduces the work to n multiplies and $2n$ adds:

```
double
echeb (double u, int n, const double c[/* n */])
{   /* evaluate a Chebyshev polynomial expansion, and return
       sum(k=0:n-1) c[k]*T(k,u), where u should be in [-1,1] */
    double b0, b1, b2, result, w;
    int k;

    if (n > 2)
    {
        w = u + u;
        b0 = w * c[n - 1] + c[n - 2];
        b1 = c[n - 1];

        for (k = n - 3; k >= 0; --k)
        {
            b2 = b1;
            b1 = b0;
            b0 = w * b1 - b2 + c[k];
        }

        result = 0.5 * (c[0] + b0 - b2);
    }
    else if (n == 2)
```

```
        result = c[1] * u + c[0];
    else if (n == 1)
        result = c[0];
    else  /* n <= 0 */
        result = nan("");

    return (result);
}
```

The mathcw library version of that function is similar, but it incorporates additional checks on the arguments.

If the *sum-prime notation* is in effect for special handling of the leading coefficient, the assignment of the final result in echeb() must be rewritten:

```
        result = 0.5 * (b0 - b2);
```

No such precaution is necessary if the coefficients are taken directly from the output of Maple's chebyshev() function.

Here is a code fragment that illustrates how the Chebyshev approximation is used:

```
#include <math.h>

static const double C[] =
{   /* tan(x) = sum(k=0:n) C[k] T(n, (8/Pi) * x - 1) */
    0.45565217041349203532,     /* * T(0, (8/Pi) * x - 1) */
    0.48946864364506139170,     /* * T(1, (8/Pi) * x - 1) */
    /* ... 19 more constants not shown here ... */
    0.15095587991023089012e-15, /* * T(21, (8/Pi) * x - 1) */
    0.25899583833164263991e-16  /* * T(22, (8/Pi) * x - 1) */
};

static int NC = (int)(sizeof(C) / sizeof(C[0]));
double a, e, r, u, x;
int n;

/* code omitted */

x = 0.125;                                      /* x on [0, PI/4] */
u = (8.0 / M_PI) * x - 1.0;                     /* u on [-1, +1] */

for (n = 1; n <= NC; n += 2)
{
    a = echeb(u, n, C);                         /* approximate value */
    e = tan(x);                                 /* ‘exact’ value */
    r = (e == 0.0) ? a : ((a - e) / e);         /* relative error */
    (void)printf("n = %2d  x = % 5.3f  relerr = % .3g\n", n, x, r);
}
```

Its output, reformatted in two columns, shows the relative error for truncated Chebyshev approximations:

```
n =  1  x = 0.125  relerr =  2.63         n = 13  x = 0.125  relerr = -6.04e-10
n =  3  x = 0.125  relerr = -0.0533       n = 15  x = 0.125  relerr =  3.91e-11
n =  5  x = 0.125  relerr = -0.0013       n = 17  x = 0.125  relerr =  3.61e-13
n =  7  x = 0.125  relerr =  4.53e-05     n = 19  x = 0.125  relerr = -3.51e-14
n =  9  x = 0.125  relerr =  9.03e-07     n = 21  x = 0.125  relerr =  0
n = 11  x = 0.125  relerr = -4.23e-08     n = 23  x = 0.125  relerr =  4.42e-16
```

3.11 Error compensation in Chebyshev fits

Although Chebyshev coefficients can be computed with high precision, they are stored in working precision, and each stored coefficient is therefore slightly in error. The coefficient magnitudes for a nearly linear function fall off rapidly, so the error in the computed function approximation is dominated by the error in the first term of the sum, $c_0 T_0(u)$. Because $T_0(u) = 1$, the first term reduces to c_0. If we represent that coefficient as a sum of exact high and approximate low parts, we can easily include the error of the leading coefficient in the sum:

$$c_0 \approx c_0^{\text{hi}} + c_0^{\text{lo}}, \qquad\qquad f(x) \approx c_0^{\text{hi}} + (c_0^{\text{lo}} + \sum_{k=1}^{n} c_k T_k(u)).$$

A variant of our `echeb()` evaluator has an extra argument that supplies the rounding error in c_0. The new function, `echeb2(u,n,c,err)`, then requires only a one-line modification to our original code:

```
result = 0.5 * (c[0] + (err + (b0 - b2)));    /* obey parentheses! */
```

The error term is readily found by the caller like this:

```
err = (C0_HI - c[0]) + C0_LO;                /* obey parentheses! */
```

If $c_0^{\text{hi}} = \text{fl}(c_0)$, then we need separate splits for each machine precision. It is more convenient to make just two splits, for binary and decimal arithmetic, so that c_0^{hi} is *exact* for all systems. A reasonable choice for c_0^{hi} is to limit it to just twenty bits, or seven decimal digits. Our symbolic-algebra support code reports suitable values for each split.

3.12 Improving Chebyshev fits

As we saw with minimax fits, it is often better to replace the function to be approximated with another that is almost linear over the interval of approximation. Continuing our example in the same Maple session, we have a new endpoint, a new function, and a new fit:

```
> a := 0:
> b := (Pi/4)**2:
> g := proc (x) if evalb(x = 0)
>               then return 0
>               else return (f(sqrt(x)) - sqrt(x)) / sqrt(x)
>               end if
>       end proc:
> d := chebsort(chebyshev(g(x), x = a .. b, 1.0e-16));
d := 0.12691084927257050148     * T(0,  32/Pi^2 * x - 1)
   + 0.13592332341164079340     * T(1,  32/Pi^2 * x - 1)
   + 0.96589245192773693641e-2  * T(2,  32/Pi^2 * x - 1)
   + 0.69285924900047896406e-3  * T(3,  32/Pi^2 * x - 1)
   + 0.49740846576329239907e-4  * T(4,  32/Pi^2 * x - 1)
   + 0.35712028205685643000e-5  * T(5,  32/Pi^2 * x - 1)
   + 0.25640062114631155556e-6  * T(6,  32/Pi^2 * x - 1)
   + 0.18408734909194089299e-7  * T(7,  32/Pi^2 * x - 1)
   + 0.13216876910270244544e-8  * T(8,  32/Pi^2 * x - 1)
   + 0.94892906728768367893e-10 * T(9,  32/Pi^2 * x - 1)
   + 0.68130041723858627104e-11 * T(10, 32/Pi^2 * x - 1)
   + 0.48915169169379567226e-12 * T(11, 32/Pi^2 * x - 1)
   + 0.35119511366507908804e-13 * T(12, 32/Pi^2 * x - 1)
   + 0.25214674712430615756e-14 * T(13, 32/Pi^2 * x - 1)
   + 0.18103321835591076899e-15 * T(14, 32/Pi^2 * x - 1)
```

To avoid a run-time error of division by zero, the proper return value for a zero argument in the function $g(x)$ must be determined by taking the limit as $x \to 0$. Maple can usually compute it like this:

```
> limit((f(sqrt(x)) - sqrt(x)) / sqrt(x), x = 0);
0
```

Otherwise, one may be able to derive it analytically from the leading terms of a series expansion, or failing that, to estimate it by computing the function in high precision for tiny argument values. Such a numerical check is advisable anyway to ensure that the assigned limit is consistent with a computed limit.

The new Chebyshev expansion for $g(x)$ requires *eight* fewer terms to reach the same accuracy as the original fit to $f(x) = \tan(x)$. However, a minimax approximation with a single polynomial of the same degree does better:

```
> e := minimax(g(x), x = a .. b, [14, 0], 1, 'maxerror'):
> printf("maxerror = %.3g\n", maxerror):
maxerror = 1.31e-17
```

A minimax rational polynomial of the same total degree is accurate to more than twice as many digits:

```
> Digits := 50:
> h := minimax(g(x), x = a .. b, [7, 7], 1, 'maxerror'):
> printf("maxerror = %.3g\n", maxerror):
maxerror = 1.56e-45
```

We can therefore reach the same accuracy of the Chebyshev fit with a minimax rational polynomial of smaller total degree:

```
> i := minimax(g(x), x = a .. b, [3, 3], 1, 'maxerror'):
> printf("maxerror = %.3g\n", maxerror):
maxerror = 2.12e-17
```

The $\langle 3/3 \rangle$ approximation needs only six multiplies, six adds, and one divide, compared to 15 multiplies and 30 adds for the degree-14 Chebyshev approximation with the optimized `echeb()` code, and as a result, might run two to four times faster.

The general superiority of minimax rational polynomials perhaps led Cody and Waite to avoid mention of Chebyshev fits in their book. We introduced them here because Maple generates them much faster than it does minimax fits, they are easy to evaluate, they are common in older software, and they can easily support a range of precisions with a single table of coefficients. When speed is more important than accuracy, software can sometimes truncate the Chebyshev expansion in intermediate computations, and then recover full precision for final results.

Occasionally, Maple cannot find a high-precision minimax polynomial for a difficult function because it runs out of memory or time resources. One such example is the reciprocal of the gamma function (see **Section 18.1** on page 521), where a degree-35 minimax polynomial accurate to about 50 decimal digits can be obtained in a few hours, whereas a degree-40 solution is unreachable. Maple produces a Chebyshev fit to that function, good to 75 decimal digits, in just a few seconds.

3.13 Chebyshev fits in rational form

Although it is awkward to do so, Maple allows a Chebyshev expansion to be converted to a rational polynomial, and that can then be further reduced into two coefficient tables, suitable for inclusion in a C, C++, or Java program. We describe the process in several steps, with interleaved prose and code.

We start a new Maple session and limit the output width so that it fits on these pages. We then make a Chebyshev fit, setting an accuracy goal suitable for the precision of the IEEE 754 64-bit binary type (a topic that we describe in more detail in **Section 4.2** on page 62), but we terminate the statement with a colon instead of a semicolon to suppress output display:

```
% maple
> interface(screenwidth = 70):
> with(numapprox):
> Digits := 17:
> a := chebsort(chebyshev(sin(x), x = 0 .. Pi/4, 2**(-53))):
> nops(a);
12
```

The `nops()` function reports the number of operands to be 12, so the fit produces an expansion with terms up to $T_{11}(8x/\pi - 1)$.

We next convert the Chebyshev expansion to a rational polynomial in the variable `b`, with terms that still involve unspecified functions `T(n,u)`. Loading the orthopoly package provides a definition of `T()`, which the `normal()` function expands to a scale factor times a rational polynomial with coefficients involving π. The `evalf()` wrapper reduces those symbolic coefficient expressions to raw numbers, so we assign the result to the variable `c`, and display the result:

```
> b := convert(a, ratpoly):
> with(orthopoly, T):
> c := evalf(normal(b));
c := - 0.31830988618379068 (-1047.0280656590893 x

                           2                          -13
     + 235.37642837528475 x  + 0.22111951452232001 10

                            6                        3
     + 0.80792428436803832 x  + 140.41239411078434 x

                           4                       5    /
     - 31.741270836138798 x  - 3.5014761266900116 x )  /  (
                                                      /

                                                                    2
    333.27938441117513 - 74.922644126233599 x + 10.851910874408098 x

                           3                         4
     - 2.3835469747678011 x  + 0.14587755374222137 x

                             5
     - 0.030070357754595340 x )
```

We now reassign T to remove its function definition, and restore it to an unset function-like variable. Then we use the type() function to verify the types of the two saved expressions:

```
> T := 'T':
> type(b, ratpoly);
  false
> type(c, ratpoly);
  true
```

Even though b is a rational expression, the presence of the unknown T in its expansion prevents its interpretation as a rational polynomial.

The variable c is a rational polynomial, but it has an unexpected outer scale factor that we need to eliminate. We can use the operand-extraction function, op(), to pick the expression apart into the factor, the numerator, and the denominator:

```
> op(1,c);
                         -0.31830988618379068

> op(2,c);
                                            2
-1047.0280656590893 x + 235.37642837528475 x

                             -13                        6
     + 0.22111951452232001 10    + 0.80792428436803832 x

                           3                        4
     + 140.41239411078434 x  - 31.741270836138798 x

                           5
     - 3.5014761266900116 x

> op(3,c);
                                                                   2
1/(333.27938441117513 - 74.922644126233599 x + 10.851910874408098 x
                           3                        4
     - 2.3835469747678011 x  + 0.14587755374222137 x

                             5
     - 0.030070357754595340 x )
```

The coefficients are displayed here in two-dimensional form, which is not convenient for incorporation in other programs. Maple has a procedure for linear printing of expressions, so we try it to output the denominator:

```
> lprint(1 / op(3,c));
333.27938441117513-74.922644126233599*x+10.851910874408098*x^2-
2.3835469747678011*x^3+.14587755374222137*x^4-.30070357754595340e-1*x^5
```

That does not help much, because we certainly do not want to evaluate it with explicit powers, and the caret, which is the power operator in Maple, means something quite different in the C family. Also, some programming languages do not permit floating-point constants to start with a decimal point instead of a leading digit, and Maple lacks an option to ensure that output style.

However, Maple has functions for converting expressions to input for a few programming languages and typesetting systems, so we can use the one for C to try to get more usable output:

```
> CodeGeneration[C](1 / op(3,c));
cg0 = 0.33327938441117513e3 - 0.74922644126233599e2 * x +
0.10851910874408098e2 * x * x - 0.23835469747678011e1 * pow(x,
0.3e1) + 0.14587755374222137e0 * pow(x, 0.4e1) -
0.30070357754595340e-1 * pow(x, 0.5e1);
```

Although that assignment is syntactically correct, it is ugly, and horribly inefficient because of the calls to the power functions.

We can remove the inefficiency by first converting the polynomial to Horner form, and we can change the target variable name to something more sensible:

```
> CodeGeneration[C](convert(1 / op(3,c), horner), resultname = "den");
den = 0.33327938441117513e3 + (-0.74922644126233599e2 +
(0.10851910874408098e2 + (-0.23835469747678011e1 +
(0.14587755374222137e0 - 0.30070357754595340e-1 * x) * x) * x)
* x) * x;
```

That assignment is now efficient, but it is still grim, and is certainly not what we want to have in portable production code, especially when that code needs to support computation in many different precisions.

We therefore introduce a private `ctable()` procedure to output a neatly formatted coefficient table with accuracy suitable for the IEEE 754 64-bit binary format:

```
> ctable := proc(name, poly)
>         local k:
>         printf("\nstatic const fp_t %s[] =\n{\n", name):
>         for k from 0 to nops(poly) - 1 do
>             printf("    FP(% 23.16e)%s\t/* %s[%2d] */\n",
>                    coeff(poly,x,k),
>                    ‘if‘(k = nops(poly) - 1, "", ","),
>                    name, k)
>         end do:
>         printf("};\n\n")
>     end proc:
```

The Maple conditional operator, `‘if‘(cond,true_expr,false_expr)`, inside the coefficient loop requires surrounding back quotes to prevent its being confused with the reserved name of a Maple conditional statement. We use it to supply a comma after all but the last coefficient. Standard C allows an optional comma after the last constant in an array or structure initializer, but some compilers are known to complain if a final comma is present.

We then call our coefficient-printing function twice to produce the coefficient tables for the numerator and the denominator:

```
> ctable("p", op(1,c) * op(2,c));

static const fp_t p[] =
{
    FP(-7.0384527500614733e-15),        /* p[ 0] */
    FP( 3.3327938441117923e+02),        /* p[ 1] */
    FP(-7.4922644126484048e+01),        /* p[ 2] */
    FP(-4.4694653188197324e+01),        /* p[ 3] */
    FP( 1.0103560307180215e+01),        /* p[ 4] */
    FP( 1.1145544673619578e+00),        /* p[ 5] */
    FP(-2.5717028700231081e-01)         /* p[ 6] */
};

> ctable("q", 1 / op(3,c));

static const fp_t q[] =
{
    FP( 3.3327938441117513e+02),        /* q[ 0] */
    FP(-7.4922644126233599e+01),        /* q[ 1] */
    FP( 1.0851910874408098e+01),        /* q[ 2] */
    FP(-2.3835469747678011e+00),        /* q[ 3] */
    FP( 1.4587755374222137e-01),        /* q[ 4] */
    FP(-3.0070357754595340e-02)         /* q[ 5] */
};
```

Notice that the numerator polynomial has a low-order coefficient, `p[0]`, that is a tiny number, instead of being an exact zero. More work is needed to ensure that requirement.

Turning the output of any symbolic-algebra system into code that is acceptable for another programming language is always a challenging problem. The Maple code shown here is a much-simplified version of what we really have to do in the support files in the `maple` subdirectory to handle binary and decimal bases, and arbitrary precision set at output time to reflect the measured error in the fit.

What have we accomplished with that laborious effort? We now have a way to convert Chebyshev fits to rational form that can be handled exactly like the results of minimax polynomial fits, so we can change initializers in header data files, and leave algorithm files untouched. The work of n multiplies and $2n$ adds for a Chebyshev fit of order n is reduced by n adds at the cost of a single divide. However, we have lost the convenient ability to truncate a single Chebyshev expansion at compile time to a length suitable for a particular precision. Instead, we have to produce a separate set of numerator and denominator coefficient tables for each run-time precision. We also do not have the extra compactness of a real minimax fit. It would seem, therefore, to be of limited utility to proceed that way, and we have refrained from doing so.

What is needed is an improved implementation of the Maple `minimax()` function that is fast, failure free, and robust, and does not require precision far in excess of that needed for the final coefficients. Although it is possible to examine the code for the `minimax()` function with the statements

```
> interface (verboseproc = 3):
> print(minimax):
```

that is proprietary licensed software that is best left unseen and unmodified in favor of developing completely new, independent, and freely distributable code.

Muller describes the ideas behind the minimax fit in the second edition of his book [Mul06], and gives a simple implementation in Maple. However, experiments with that code quickly show that it is not robust, and thus, not suitable for our needs.

New code, if properly documented, could likely facilitate extensions to apply constraints to coefficients for preservation of important mathematical properties, and to ensure that the output coefficients are exactly representable. It would also be desirable to exploit the parallelism available in modern systems with multiple processors, each with several CPU cores, to speed the computation, which is particularly onerous for the high-precision fits needed in the `mathcw` library.

We also need to develop similar fitting routines for other symbolic-algebra systems, none of which currently supplies anything comparable to the Maple and Mathematica fitting functions.

3.14 Chebyshev fits with Mathematica

In Section 3.6 on page 33, we described in detail how support for rational approximations in the Mathematica symbolic-algebra system can be enhanced to make it convenient to convert fits to data suitable for direct inclusion in a C program.

The FunctionApproximations package does not include Chebyshev fits, so we developed a file of support code, chebfit.mth, to make such fits, and their output as C code, easy. The code is lengthy, so we omit it here, but a short example shows how easy it is to use:

```
% math
In[1]:= << chebfit.mth

In[2]:= F[x_] = (Exp[x] - 1) / x;

In[3]:= ca = ChebyshevApproximation[ F, x, 0, 1/2, 10^(-17), 20 ];

In[4]:= CChebyshevFit[ ca, "x", "(exp(x) - 1) / x", "c", 16 ]

/***
 *** Chebyshev fit of
 ***         F(x) = (exp(x) - 1) / x
 *** by
 ***    Fapprox(x) = sum(c[k] * T(k, u(x)), k = 0 .. n)
 ***
 *** for u(x) on [-1, +1] and x on [0.e+00, 5.e-01]
 *** where u(x) = (x + x - (xmax + xmin)) / (xmax - xmin)
 ***/

#define XMIN           FP(0.e+00)

#define XMAX           FP(5.e-01)

#define XtoU(x)        ((x + x - (XMAX + XMIN)) / (XMAX - XMIN))

#define UtoX(u)        (XMIN + (FP(0.5) * u + FP(0.5)) * (XMAX - XMIN))

static const fp_t c[] = {
  FP(  1.142406442850953e+00 ),   /* * T( 0, u ) : relerr = 1.19e-01 */
  FP(  1.485216799669541e-01 ),   /* * T( 1, u ) : relerr = 5.02e-03 */
  FP(  6.309800086663982e-03 ),   /* * T( 2, u ) : relerr = 1.57e-04 */
  FP(  1.994854176176553e-04 ),   /* * T( 3, u ) : relerr = 3.95e-06 */
  FP(  5.025874182378426e-06 ),   /* * T( 4, u ) : relerr = 8.26e-08 */
  FP(  1.052859572070425e-07 ),   /* * T( 5, u ) : relerr = 1.47e-09 */
  FP(  1.887917064282244e-09 ),   /* * T( 6, u ) : relerr = 2.31e-11 */
  FP(  2.959401185872932e-11 ),   /* * T( 7, u ) : relerr = 3.21e-13 */
  FP(  4.120904413190384e-13 ),   /* * T( 8, u ) : relerr = 4.02e-15 */
  FP(  5.162027111524086e-15 )    /* * T( 9, u ) : relerr = 4.57e-17 */
};
```

ChebyshevApproximation[f, x, xmin, xmax, relerr, digits] does the fit to $f(x)$. Here, relerr is the desired relative error in the approximation, and the optional last argument digits is the minimal number of decimal digits in the output coefficients. If omitted, it takes the value of a global variable, Digits. The Chebyshev coefficients are calculated with somewhat more than twice the requested precision.

The function returns a list that includes functions for computing the approximation and its error at the k-th term, and for converting between x and u, followed by lists of coefficients, and the zeros of the n-th order Chebyshev polynomial in units of both x and u. In exact arithmetic, the fit with n coefficients is exact at those zeros.

The Chebyshev fit never requires function values at the endpoints, so the computation avoids the numerical artifact of $f(0) = 0/0$ that requires perturbations in some of our minimax fits.

`CChebyshevFit[chebappr, var, funname, cname, digits]` produces the C data shown to record necessary information about the fit. Each coefficient is followed by a comment that gives the error in the fit if the sum is truncated after that term. That feature makes it easy to select the correct number of terms needed for a given target precision.

3.15 Chebyshev fits for function representation

In several places later in this book, we fall back to Chebyshev fits when minimax rational polynomials cannot be computed in a timely manner, or at all.

Although its results have not been used in this book, the *Chebfun Project*[6] team has developed the technology for numerical representation of smooth continuous functions with bounded first derivatives in the open-source package chebfun for MATLAB. The current implementation is based on IEEE 754 binary arithmetic, without access to the higher precision that we have seen is necessary if we are to derive fits that are 'good to the last bit' for use in MathCW library code.

A clever feature of the Chebfun system is that it allows compact, familiar, and uniform notation for operations on functions that are (possibly only) defined numerically. In particular, they can be convolved, differentiated, integrated, plotted, and transformed, and all of their minima, maxima, and roots in a given interval can be located quickly. They may also appear in differential and integral equations to be solved. Conveniently, much of the chebfun notation and software can be transparently adapted for periodic functions, where the expansions are instead in trigonometric functions. The chebfun package has recently been extended for functions of two and three variables, and for working with polar and spherical coordinates. The state of the art of the project is summarized in a paper published shortly before this book went to press [AT17].

3.16 Extending the library

The elementary functions covered in the Cody/Waite book are adequate for Fortran 77, Fortran 90, and Fortran 95. When that book was written, Fortran was by far the most important programming language for numerical computation. The book's algorithms are described only in prose and flowchart form, independent of any particular programming language, so its authors were clearly targeting other languages too. However, the test package for their functions is written in Fortran, and we discuss it later in **Chapter 22** on page 763.

The coverage is almost sufficient for C89, which adds `ceil()`, `fabs()`, `floor()`, `fmod()`, `frexp()`, `ldexp()`, and `modf()`. There are no `float` and `long double` companions in C89. Those additional functions are not elementary functions in the traditional mathematical sense, but are nevertheless useful in numerical computation. Some require access to the underlying floating-point representation, but apart from `fmod()`, none requires iteration or polynomial approximation.

As we noted earlier in **Section 1.2** on page 4, C99 adds a few dozen new functions to those of C89.

The C# `Math` class provides a subset of the C89 elementary functions, but with capitalized overloaded names, plus `BigMul()` (for exact integer products), `DivRem()` (for integer arithmetic), `IEEERemainder()`, and `Round()`.

The Java package java.lang.math supplies all of the standard elementary functions defined by Fortran, plus `abs()`, `ceil()`, `floor()`, `IEEEremainder()`, `max()`, `min()`, `random()`, `rint()`, and `round()`. Except for `IEEEremainder()`, none of those additional functions requires iteration or polynomial approximation either.

The 1998 ISO C++ Standard [C++98] requires the same elementary functions as C89, plus their `float` and `long double` companions, and the 2003 ISO C++ Standard [C++03a] mandates the same `<math.h>` library as C++98 and C89. Curiously, it does not require the C99 mathematical library extensions. As this book was in its final stages, a proposal [C++10] appeared to include selected special functions in the C++ library: Hermite polynomials, Laguerre polynomials, Legendre polynomials, beta function (not ours), complete and incomplete elliptic integrals of three kinds, cylindrical and spherical Bessel functions, exponential integral, and Riemann zeta function. The proposal function names are long, and the repertoire is seriously deficient.

The mathcw library supplies the full C89 repertoire defined in the system header file `<math.h>`, plus all of the C99 math library additions, and about 400 other function families, as recorded in **Table 3.4** on the next page through **Table 3.6** on page 60.

[6] See `http://www.chebfun.org/` and `http://www.math.utah.edu/pub/bibnet/authors/t/trefethen-lloyd-n.bib`.

Table 3.4: Standard contents of the mathcw library. All of those routines have `float` and `long double` companions, with suffix letters `f` and `l`. In addition, on HP-UX on IA-64, there are also `extended` and `quad` companions with suffix letters `w` and `q`. The future `long long double` type has the suffix `ll`. When decimal floating-point arithmetic is available, the suffix letters are `df`, `d`, `dl`, and `dll`.

C89 conformance requires only the `double` versions, and consequently, many C implementations lack the corresponding `float` and `long double` routines.

The C99 macros are type generic and have corresponding functions of the same names. The macros use the `sizeof` operator on their arguments to determine which function to call.

There is no support in the mathcw library for the rarely used wide-character functions; they are expected to be supplied by the native C library. Most of the C89 and C99 string functions are omitted. because they usually have optimized native implementations.

C89	`acos() asin() atan2() atan() ceil() cosh() cos() exp() fabs() floor() fmod() fprintf() frexp() fscanf() ldexp() log10() log() modf() pow() printf() scanf() sinh() sin() sprintf() sqrt() sscanf() strtol() tanh() tan() vfprintf() vprintf() vsprintf()`
C99 new functions	`acosh() asinh() atanh() cbrt() copysign() erfc() erf() exp2() expm1() fdim() feclearexcept() fegetenv() fegetexceptflag() fegetexcept() fegetprec() fegetround() feholdexcept() feraiseexcept() fesetenv() fesetexceptflag() fesetprec() fesetround() fetestexcept() feupdateenv() fmax() fma() fmin() hypot() ilogb() lgamma() llrint() llround() log1p() log2() logb() lrint() lround() nan() nearbyint() nextafter() nexttoward() remainder() remquo() rint() round() scalbln() scalbn() snprintf() strtold() strtoll() tgamma() trunc() vfscanf() vscanf() vsnprintf() vsscanf()`
C99 macros	`fpclassify() isfinite() isinf() isnan() isnormal() signbit()`
C99 macros (and functions)	`isgreater() isgreaterequal() isless() islessequal() islessgreater() isunordered()`
C99 symbols	`FP_INFINITE FP_NAN FP_NORMAL FP_SUBNORMAL FP_ZERO`

3.17 Summary and further reading

We have seen in this chapter that minimax rational polynomial fits provide accurate, compact, and fast representations of functions over small argument ranges. Such fits provide the critical kernels of many of the elementary and special functions supplied by mathematical libraries, and once the technique is understood, and the needed software is at hand, they should be more-widely employed in user code. However, the state of the art of software for producing such fits is still immature, and there is much room for improvement. The required precision is such that most conventional programming languages are hopelessly inadequate for generating such fits, so the arbitrary-precision arithmetic, and large library-function repertoires, of symbolic-algebra systems are essential.

When it is impractical, or impossible, to compute a minimax fit to a function, we fall back to Chebyshev expansions. They are much easier, and faster, to compute, and their errors can be almost as small as those of minimax fits. However, it usually takes more terms to do so, resulting in lower performance. Our code always provides special handling of the IEEE 754 exceptional values of Infinity, NaN, and negative zeros (and sometimes also for subnormals), and it usually employs fast-converging Taylor series for small arguments to ensure correct limiting behavior. When those cases do not apply, polynomial fits, or series expansions, or continued fractions, or iterative techniques, are the primary tools for computation of the elementary and special functions treated in this book.

Although we have not remarked on it before, there is a hidden chicken-and-egg problem here. To compute a polynomial fit that represents $f(x)$, we first need $f(x)$ itself. A large arbitrary-precision function library, or access to higher precision for brute-force evaluation of Taylor series or continued-fraction representations, is essential.

Mathematical handbooks [AS64, JEL68, BF71, Zwi92, Zwi03, Gau04, GST07, GRJZ07, Bry08, CPV$^+$08, JD08, Wei09, OLBC10, Zwi12] are helpful resources that distill centuries of mathematics research to compact summaries of function properties that provide the necessary clues for constructing fast, robust, and stable numerical algorithms.

Many books on numerical analysis include discussions of the use of Chebyshev polynomials for polynomial economization and fitting of functions, and there are a few books devoted entirely to them [FP68, Riv74, Riv90, MH03]. Although we discuss only the Chebyshev polynomials of the first kind, there are companions $U_n(u)$, $V_n(u)$, and

Table 3.5: Extended contents of the mathcw library, part 1. These functions are not part of the Standard C library, but represent the *value-added* features of the mathcw library. Functions that end in `_r` avoid the need for internal state that is preserved across calls, or made available as global variables. However, the usual type suffixes *precede* the `_r` suffix.

Cody/Waite primitives	`adx(), intxp(), setxp()`
mathcw extensions	`acosdeg() acospi() acosp() agm() annuity() asindeg() asinpi() asinp() atan2deg() atan2pi() atan2p() atandeg() atanpi() atanp() bernum() betam1() beta() betnm1() betnum() bi0() bi1() binom() bin() bis0() bis1() bisn() bk0() bk1() bkn() bks0() bks1() bksn() cadd() cconj() ccopy() cdiv() chisq() clp2() cmul() cneg() compound() cosdeg() cospi() cosp() cotandeg() cotanpi() cotanp() cotan() csub() cvtia() cvtib() cvtid() cvtig() cvtih() cvtio() cvtob() cvtod() cvtog() cvtoh() cvtoi() cvton() cvtoo() cxabs() cxacosh() cxacos() cxadd() cxarg() cxasinh() cxasin() cxatanh() cxatan() cxcbrt() cxconj() cxcopy() cxcosh() cxcos() cxdiv() cxexpm1() cxexp() cximag() cxipow() cxlog1p() cxlog() cxmul() cxneg() cxpow() cxproj() cxreal() cxset() cxsinh() cxsin() cxsqrt() cxsub() cxtanh() cxtan() dfabs() dfact() dfadd() dfdiv() dfmul() dfneg() dfsqrt() dfsub() echeb() echeb2() eljaam() eljacd() eljacn() eljacs() eljadc() eljadn() eljads() eljag() eljam() eljanc() eljand() eljans() eljasc() eljasd() eljasn() eljcd() eljcn() eljcs() eljdc() eljdn() eljds() eljh4() eljh() eljnc() eljnd() eljns() eljsc() eljsd() eljsn() eljt1() eljt2() eljt3() eljt4() eljta() eljtd1() eljtd2() eljtd3() eljtd4() eljtda() eljt() eljz() elkm1() elk() elldi() ellec() ellei() elle() ellfi() ellkc() ellkn() ellk() ellpi() ellrc() ellrd() ellre() ellrf() ellrg() ellrh() ellrj() ellrk() ellrl() ellrm() elntc() elntd() elntn() elnts() elq1p() elqc1p() elqc() elq() elwdp() elwe() elwg() elwip() elwk() elwo() elwp() elws() elwz() ercw_r() ercw() ereduce() erfcs() eriduce() eulnum() exp10m1() exp10() exp16m1() exp16() exp2m1() exp8m1() exp8() fact() fadj() fcoef() fexpon() fibnum() flp2() fmul() frexph() frexpo() fsplit() gamib() gamic() gami()`

$W_n(u)$ of the second, third, and fourth kinds, and *shifted* Chebyshev polynomials, $T_n^*(u)$ through $W_n^*(u)$, defined over the unit interval $[0,1]$. The *Handbook of Mathematical Functions* [AS64, Chapter 22] also treats Chebyshev polynomials $C_n(t)$ and $S_n(t)$ over the interval $[-2,2]$, but those functions are rarely encountered.

We noted that the Chebyshev polynomials are *orthogonal polynomials*, with the property that certain integrals and sums over products $T_m(u)T_n(u)$, possibly with a product weight function, are zero unless $m = n$. There are several other commonly used families of polynomials that have that property as well, and they are attached to the names of famous mathematicians of the 18th and 19th Centuries, including Gegenbauer, Hermite, Jacobi, Laguerre, Legendre, and others.

The orthogonality property is significant for several reasons. One of the most important is that it means that integrals and sums of products of two functions, each represented as n-term expansions in orthogonal polynomials, possibly with an accompanying weight function, reduce from an $\mathcal{O}(n^2)$ problem to a much simpler one of $\mathcal{O}(n)$.

There are several books devoted to orthogonal polynomials [Bec73, Chi78, vA87, Mac98, BGVHN99, LL01, Gau04, El 06, MvA06, BKMM07, Khr08]. Scores more, including the proceedings of many mathematical conferences on the subject, can be found in library catalogs. The *MathSciNet* and *zbMATH* databases each record more than 3500 research articles on orthogonal polynomials. Symbolic-algebra systems usually have substantial loadable packages that provide support for orthogonal polynomials.

Although few mathematics texts connect the two, there is a significant relation between continued fractions and orthogonal polynomials. Most of the great mathematicians of the last four hundred years have worked in either, or both, of those areas. See *Orthogonal Polynomials and Continued Fractions* [Khr08] for details.

There are many biographies of famous mathematicians listed in library catalogs, but we cite only a few of them [Bel37, Der03, Dev11, Dun99, Dun55, DGD04, Fra99, Gau08, GBGL08, Haw05, HHPM07, MP98, Ten06, YM98]. Numerous books provide accounts of the history of mathematics [Bre91, Bur07, Caj91, CKCFR03, CBB$^+$99, Eve83, GBGL08, GG98, Hel06, Ifr00, KIR$^+$07, Kat09, Mao07, McL91, Rud07, Sig02, Smi58, Sti02, Suz02]. Some books concentrate on the history of important mathematical numbers [Bec93, BBB00, Bla97, Cle03, EL04b, Fin03, Gam88, Hav03, HF98, Kap99, Liv02, Mao91, Mao94, Nah06, Rei06, Sei00]. There are also books about the great problems in mathe-

Table 3.6: Extended contents of the mathcw library, part 2. The Bessel functions `j0()`, `j1()`, `jn()`, `y0()`, `y1()`, and `yn()` are required by POSIX Standards. Although they are provided by many UNIX C libraries, their implementations are often substandard. Our implementations remedy that deficiency.

mathcw extensions	ichisq() ierfc() ierf() ilog2() infty() iphic() iphi() ipow() iscinf() iscnan() iscxinf() iscxnan() isqnan() issnan() issubnormal() is_abs_safe() is_add_safe() is_div_safe() is_mul_safe() is_neg_safe() is_rem_safe() is_sub_safe() j0() j1() jn() ldexph() ldexpo() lgamma_r() log101p() log161p() log16() log21p() log81p() log8() logbfact() lrcw_r() lrcw() mchep() nlz() normalize() nrcw_r() nrcw() ntos() ntz() pabs() pacosh() pacos() padd() pasinh() pasin() patan2() patanh() patan() pcbrt() pcmp() pcon() pcopysign() pcopy() pcosh() pcos() pcotan() pdiv() pdot() peps() peval() pexp10() pexp16() pexp2() pexp8() pexpm1() pexp() pfdim() pfmax() pfmin() pfrexph() pfrexp() pgamma() phic() phigh() phi() phypot() pierfc() pierf() pilogb() pinfty() pin() pipow() pisinf() pisnan() pisqnan() pissnan() pldexph() pldexp() plog101p() plog1p() plogb() plog() plow() pmul2() pmul() pneg() pop() pout() pprosum() pqnan() pscalbln() pscalbn() pset() psignbit() psiln() psinh() psin() psi() psnan() psplit() psqrt() psub() psum2() psum() ptanh() ptan() qert() qnan() quantize() rsqrt() samequantum() sbi0() sbi1() sbin() sbis0() sbis1() sbisn() sbj0() sbj1() sbjn() sbk0() sbk1() sbkn() sbks0() sbks1() sbksn() sby0() sby1() sbyn() second() sincospi() sincosp() sincos() sindeg() sinhcosh() sinpi() sinp() snan() strlcat() strlcpy() tandeg() tanpi() tanp() ulpk() ulpmh() urcw1_r() urcw1() urcw2_r() urcw2() urcw3_r() urcw3() urcw4_r() urcw4() urcw_r() urcw() vagm() vbis() vbi() vbj() vbks() vbk() vby() vercw_r() vercw() vlrcw_r() vlrcw() vnrcw_r() vnrcw() vsbis() vsbi() vsbj() vsbks() vsbk() vsby() vsum() vurcw1_r() vurcw1() vurcw2_r() vurcw2() vurcw3_r() vurcw3() vurcw4_r() vurcw4() vurcw_r() vurcw() y0() y1() yn() zetam1() zeta() zetnm1() zetnum()

matics, many of which remained unsolved [Sin97, Wil02, Der03, Sab03, Szp03, dS03, CJW06, Roc06, Szp07, Dev08a, GBGL08].

4 Implementation issues

In theory, theory and practice are the same.
In practice, they are not.

— Attributed to baseball player Yogi Berra.

Cody and Waite specified their algorithms for the computation of the elementary functions in a language-independent way, although they wrote the accompanying ELEFUNT test package in Fortran. In principle, it should be straightforward to implement their recipes in any programming language on any operating system and any arithmetic system. However, for any particular environment, there are likely to be issues that need to be considered. In this chapter we discuss features of the C language and of the IEEE 754 arithmetic system that affect the implementation of the mathcw library, but we first examine a fundamental mathematical limitation on the accuracy of function evaluation.

4.1 Error magnification

In assessing the accuracy of a computed function, an important question is: *What is the sensitivity of the function to errors in its argument?* From calculus, the definition of the first derivative is

$$f'(x) = \lim_{\delta x \to 0} \frac{f(x + \delta x) - f(x)}{\delta x}.$$

Here, δ is the Greek letter delta that is commonly used in mathematics for small things, and lim is the *limit operator*. Thus, the relative change in the function value is given by

$$\frac{f(x + \delta x) - f(x)}{f(x)} \approx x \frac{f'(x)}{f(x)} \frac{\delta x}{x}.$$

That means that the relative error in the argument, $\delta x / x$, is *magnified* by the factor $x f'(x)/f(x)$ to produce the relative error in the function.

Table 4.1 on the next page summarizes the magnification factors for common elementary and special functions. The factors in the exponential, gamma, and complementary error functions grow with x: accuracy for large arguments requires higher precision. By contrast, the square root, reciprocal square root, cube root, and the ordinary error function have factors that never exceed one: it should be possible to evaluate them accurately over the entire argument range.

Some of the inverse trigonometric and hyperbolic functions have factors with denominators that go to zero as $x \to \pm 1$, and also as x approaches a zero of the function; they are thus quite sensitive to errors in arguments near those values.

The cosine is sensitive to arguments near nonzero multiples of $\pi/2$, and the sine to arguments near nonzero multiples of π. However, in both cases, the function values are near zero anyway, so the sensitivity may not be noticed in practice.

The functions with poles ($\Gamma(x)$ (capital Greek letter *gamma*), $\psi(x)$ (Greek letter *psi*), psiln(x), $\log(x)$, ierf(x), ierfc(x), $\tan(x)$, $\cot(x)$, atanh(x), and their companions) are extremely sensitive to small changes in arguments near those poles. Testing such functions in the pole regions requires higher precision, so the test programs may simply avoid those regions.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_4

Table 4.1: Magnification factors for some of the elementary and special functions in the mathcw library. The factor for $\psi(x)$ requires the first polygamma function, $\mathrm{pgamma}(1, x)$.

Function	Magnification factor	Function	Magnification factor
$\sqrt{x}$	$1/2$	$\cos(x)$	$-x\tan(x)$
$\mathrm{rsqrt}(x)$	$-1/2$	$\sin(x)$	$x/\tan(x)$
$\mathrm{cbrt}(x)$	$1/3$	$\tan(x)$	$x(1+\tan^2(x))/\tan(x)$
$\mathrm{pow}(x, y)$	y (w.r.t. x) and $y\log(x)$ (w.r.t. y)	$\cot(x)$	$-x(1+\cot^2(x))/\cot(x)$
$\exp(x)$	x	$\mathrm{acos}(x)$	$-x/(\sqrt{1-x^2}\,\mathrm{acos}(x))$
$\mathrm{exp2}(x)$	$x\log(2)$	$\mathrm{asin}(x)$	$x/(\sqrt{1-x^2}\,\mathrm{asin}(x))$
$\mathrm{exp2m1}(x)$	$x\,\mathrm{exp2}(x)\log(2)/\mathrm{exp2m1}(x)$	$\mathrm{atan}(x)$	$x/(\sqrt{1+x^2}\,\mathrm{atan}(x))$
$\mathrm{exp10}(x)$	$x\log(10)$	$\cosh(x)$	$x\tanh(x)$
$\mathrm{exp10m1}(x)$	$x\,\mathrm{exp10}(x)\log(10)/\mathrm{exp10m1}(x)$	$\sinh(x)$	$x/\tanh(x)$
$\mathrm{expm1}(x)$	$x\exp(x)/\mathrm{expm1}(x)$	$\tanh(x)$	$x(1-\tanh^2(x))/\tanh(x)$
$\mathrm{erf}(x)$	$2x\exp(-x^2)/(\sqrt{\pi}\,\mathrm{erf}(x))$	$\mathrm{acosh}(x)$	$x/(\sqrt{x^2-1}\,\mathrm{acosh}(x))$
$\mathrm{erfc}(x)$	$-2x\exp(-x^2)/(\sqrt{\pi}\,\mathrm{erfc}(x))$	$\mathrm{asinh}(x)$	$x/(\sqrt{1+x^2}\,\mathrm{asinh}(x))$
$\mathrm{ierf}(x)$	$\sqrt{\pi/4}\,x\exp([\mathrm{ierf}(x)]^2)/\mathrm{ierf}(x)$	$\mathrm{atanh}(x)$	$x/(\sqrt{1-x^2}\,\mathrm{atanh}(x))$
$\mathrm{ierfc}(x)$	$-\sqrt{\pi/4}\,x\exp([\mathrm{ierfc}(x)]^2)/\mathrm{ierfc}(x)$	$I_n(x)$	$n + xI_{n+1}(x)/I_n(x)$
$\Gamma(x)$	$x\psi(x)$	$J_n(x)$	$n - xJ_{n+1}(x)/J_n(x)$
$\log(\Gamma(x))$	$x\psi(x)/\log(\Gamma(x))$	$K_n(x)$	$n - xK_{n+1}(x)/K_n(x)$
$\psi(x)$	$x\,\mathrm{pgamma}(1, x)/\psi(x)$	$Y_n(x)$	$n - xY_{n+1}(x)/Y_n(x)$
$\mathrm{psiln}(x)$	$(x\,\mathrm{pgamma}(1, x) - 1)/\mathrm{psiln}(x)$	$i_n(x)$	$-\frac{1}{2} + \frac{1}{2}x(i_{n-1}(x) + i_{n+1}(x))/i_n(x)$
$\log(x)$	$1/\log(x)$	$j_n(x)$	$-\frac{1}{2} + \frac{1}{2}x(j_{n-1}(x) - j_{n+1}(x))/j_n(x)$
$\mathrm{log1p}(x)$	$x/((1+x)\,\mathrm{log1p}(x))$	$k_n(x)$	$-\frac{1}{2} - \frac{1}{2}x(k_{n-1}(x) + k_{n+1}(x))/k_n(x)$
$\mathrm{log21p}(x)$	$x/((1+x)\,\mathrm{log1p}(x))$	$y_n(x)$	$-\frac{1}{2} + \frac{1}{2}x(y_{n-1}(x) - y_{n+1}(x))/y_n(x)$
$\mathrm{log101p}(x)$	$x/((1+x)\,\mathrm{log1p}(x))$		

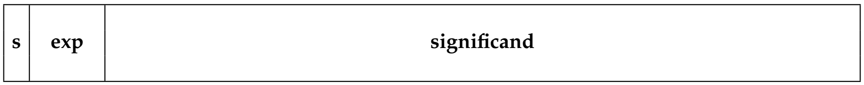
4.2 Machine representation and machine epsilon

Floating-point arithmetic represents numerical values as *rational numbers*, with the denominators constrained to be powers of the *base*. The base is conventionally denoted by β, the Greek letter *beta*. *Radix* is a name used by some authors for the base, and it appears in the macro `FLT_RADIX` that is defined in the C header file, `<float.h>`. The number of digits in the numerator, and the range of powers of the base, are fixed by storage constraints and by the floating-point design. The fixed number of digits means that between any two floating-point values, there are infinitely many real numbers. Only those real numbers that exactly match a floating-point number can be represented, and they are called *machine numbers*.

Numbers are said to be *normalized* when the numerator has the maximal number of digits, and the first of them is nonzero. For a given exponent of the base, the spacing between consecutive numbers is constant, but the spacing changes when the exponent changes.

To see that, consider a simple two-digit decimal floating-point system with a one-digit signed exponent, and integer numerators. Apart from zero, that permits representation of normalized numbers of the form $10, 11, 12, \ldots, 97, 98, 99$, each multiplied by a power $10^{-9}, 10^{-8}, \ldots, 10^{0}, \ldots, 10^{8}, 10^{9}$. The number spacing is 1×10^{-9} at the smallest exponent, then 1×10^{-8} at the next higher exponent, and increases to 1×10^{9} at the highest exponent.

For a general base β with t digits in that base, we adopt the convention that the fractional point follows the first digit of the numerator, so numbers have the form $d.ddd\ldots ddd_{\beta} \times \beta^{n}$, where the subscript records the base. There are then $t-1$ fractional digits, and the last of them has the value $d \times \beta^{-(t-1)}$. Each digit d is an integer in the range $[0, \beta-1]$. When $n = 0$, the smallest nonzero normalized number is $1.000\ldots000_{\beta}$, and the next number after it is $1.000\ldots001_{\beta}$. The difference between those two values, $\beta^{-(t-1)} = \beta^{1-t}$, is called the *machine epsilon*. It is denoted by the Greek letter *epsilon*, ϵ, which is commonly used in mathematics to mean something rather small.



bit	0	1	9	31	single
	0	1	12	47	single extended (*unused*)
	0	1	12	63	double
	0	1	16	79	double extended
	0	1	16	127	quadruple
	0	1	22	255	octuple

Figure 4.1: Extended IEEE-754-like binary floating-point data layout.
s is the sign bit (0 for $+$, 1 for $-$).
exp is the unsigned biased exponent field, with the smallest value reserved for zero and subnormal numbers, and the largest value reserved for Infinity and NaN. Infinity has a zero significand, whereas quiet and signaling NaN have nonzero significands, with an implementation-dependent bit in the significand to distinguish between them.
The remaining bits hold the **significand**. Except in the 80-bit format, it implicitly contains a high-order *hidden* (not stored) bit that is 1 for normal numbers and 0 for subnormal numbers. The binary point follows the first significand bit, after supplying any hidden bit.

The value ϵ is the smallest representable number that can be added to 1.0, such that the sum differs from 1.0, and is therefore sometimes called the *positive epsilon* or the *big epsilon*. The number closest to 1.0, but below it, has the form $D.DDD \ldots DDD_\beta \times \beta^{-1}$, where $D = \beta - 1$ is the largest digit. The difference between 1.0 and that number is β^{-t}, and is called the *negative epsilon* or the *little epsilon*. It is a factor of β smaller than the machine epsilon, and thus has the value ϵ/β.

4.3 IEEE 754 arithmetic

The floating-point arithmetic used in almost all computers designed since the early 1980s follows the *IEEE 754 Standard for Binary Floating-Point Arithmetic* [IEEE85a, IEEE85b, IEEE87]. The chief architect of that arithmetic is Professor William Kahan of the University of California, Berkeley, and an interview with him [Sev98a, Sev98b] records a short summary of its development. **Appendix H** on page 947 surveys some of the important historical machine families and floating-point systems whose arithmetical deficiencies are largely corrected by the IEEE 754 design.

Starting in 1977, Kahan led a large team of hardware architects and experts in numerical computing, and building on their long experience with earlier floating-point systems, they worked out the details, and published them for early review and comment [Coo80, Cod81, Coo81a, Coo81b, Hou81, Ste81a, Ste81b]. In 1980, team members from Intel produced the first hardware, the 8087 coprocessor chip, to implement the new design [PM84]. However, the Standard was not finalized until 1985, and as might be expected, the early implementation in the 8087 does not fully conform to the final Standard. Alas, by 1985, the 8087 user base was sufficiently large that Intel chose stability over conformance, and members of the IA-32 architecture CPU family manufactured more than 25 years later still largely adhere to the 8087 design.

We address more of the details of IEEE 754 arithmetic in later sections, but here, we summarize only the main features:

- Floating-point numbers are represented in 32-bit, 64-bit, 80-bit, and 128-bit storage formats[1] with a one-bit sign, a power of two suitably biased to avoid the need for an additional sign, and a *significand* that for finite nonzero normal numbers is considered to lie in the range $[1, 2)$. The data layouts are illustrated in **Figure 4.1**. The corresponding layouts for decimal floating-point arithmetic are shown in **Figure D.1** and **Figure D.2** on page 930.

  Only the 32-bit and 64-bit formats are considered essential, and most current platforms include just one of the longer formats, if they do so at all.

[1] Although the IEEE 754 specification includes a *single extended* format, with at least 32 significand bits and at least 11 exponent bits, in practice, it has never been widely implemented in desktop (and larger) computers. We make no further mention of it in this book.

- Because the sign is stored separately, both positive and negative zeros can be represented. Although they compare equal, their signs may usefully reflect how they were computed.

- For normal numbers, significands are normalized so that the leading (high-order) bit is one. Except in the 80-bit format, that bit is not stored, and is therefore called a *hidden bit*.

- When the smallest (most negative) representable exponent is reached, the normalization requirement may be relaxed, allowing significands to have leading zero bits so that small values experience *gradual underflow* with progressive loss of significance until they finally become zero.

- Four dynamic rounding modes are available under program control. The default rounding mode minimizes the average rounding error.

- Special representations of Infinity and NaN (Not A Number) allow nonstop computing when computed results could not otherwise be represented as finite numbers. Two kinds of NaN, quiet and signaling, are provided. The intent is that the arithmetic generates quiet NaNs, but programmers can initialize storage to signaling NaNs whose subsequent use in numeric instructions can be trapped.

- Exceptional conditions are normally handled by setting *sticky flags* in a status field that can be read and written by the user programs, rather than by trapping to an exception handler that might choose to terminate the job.

  Once set, flags remain set until explicitly cleared. The program can clear the flags, perform an exception-free computation, and later check to see whether any important flag is set. If so, the program can then take remedial action, perhaps retrying the computation with a different algorithm.

The view of the original designers is that all of those features are *essential*. Alas, subsequent architects have sometimes ignored one or more of the items in our list: Infinity might be replaced by the largest representable finite number, underflow might be abrupt, exception flags might be eliminated, and only one rounding mode might be available [KD98]. On some systems, the ignored features can be restored by special compiler options or run-time library calls, but doing so may hurt performance.

An important lesson from many of the functions that we treat in this book is that higher intermediate precision can often provide simple solutions to accuracy problems that are difficult to resolve in normal working precision. The `mathcw` library is therefore designed to handle a future 256-bit format based on IEEE 754 arithmetic that would naturally be selected by `long long double` in the C language family, or `REAL*32` in old-style Fortran declarations. Informally, we call it *octuple precision*. Lack of run-time library support is always a major impediment to adding any new data type to a programming language. The `mathcw` library removes that obstruction for the C family, and other languages for which library interfaces are available (see the appendices in this book).

It is important for numerical programmers to be familiar with the range and precision of floating-point arithmetic, so we summarize the characteristics of IEEE 754 binary arithmetic in **Table 4.2** on the next page. Parameters for decimal arithmetic are given in **Table D.1** and **Table D.2** on page 929.

Although we do not describe them further in this book, it is worth noting that floating-point representations have even been defined for 16-bit and 8-bit quantities. The 16-bit format is sometimes called *half float* or *minifloat*. They find use in some computer-graphics and signal-processing applications, but their precision is too limited to be of much interest elsewhere.

4.4 Evaluation order in C

Most programming languages adopt that convention that evaluation order of expressions can be controlled by parentheses: thus, `(a + b) - c` means that the addition must precede the subtraction. That is a sensible choice, because computer arithmetic is *not* associative. The order in which arithmetic operations are carried out to evaluate an expression can affect the result dramatically: one order might produce a correct answer, another a mildly incorrect one, and yet another, a totally incorrect result.

Unfortunately, prior to the 1990 ISO C Standard, the C language did not require parentheses to be obeyed. A C compiler could then legally evaluate our sample expression as `(a - c) + b` or as `(b - c) + a`, and many did, especially when optimization was selected. That reordering is *completely unacceptable* for numerical software.

Table 4.2: Extended IEEE-754-like binary floating-point characteristics and limits.

	single	double	extended	quadruple	octuple
Format length	32	64	80	128	256
Stored significand bits	23	52	64	112	236
Precision (t)	24	53	64	113	237
Biased-exponent bits	8	11	15	15	19
Minimum exponent	−126	−1022	−16 382	−16 382	−262 142
Maximum exponent	127	1023	16 383	16 383	262 143
Exponent bias	127	1023	16 383	16 383	262 143
Machine epsilon	2^{-23}	2^{-52}	2^{-63}	2^{-112}	2^{-236}
(2^{-t+1})	≈ 1.19e−07	≈ 2.22e−16	≈ 1.08e−19	≈ 1.93e−34	≈ 9.06e−72
Largest normal	$(1-2^{-24})2^{128}$	$(1-2^{-53})2^{1024}$	$(1-2^{-64})2^{16\,384}$	$(1-2^{-113})2^{16\,384}$	$(1-2^{-237})2^{262\,144}$
	≈ 3.40e+38	≈ 1.80e+308	≈ 1.19e+4932	≈ 1.19e+4932	≈ 1.611e+78 913
Smallest normal	2^{-126}	2^{-1022}	$2^{-16\,382}$	$2^{-16\,382}$	$2^{-262\,142}$
	≈ 1.18e−38	≈ 2.23e−308	≈ 3.36e−4932	≈ 3.36e−4932	≈ 2.482e−78913
Smallest subnormal	2^{-149}	2^{-1074}	$2^{-16\,445}$	$2^{-16\,494}$	$2^{-262\,378}$
	≈ 1.40e−45	≈ 4.94e−324	≈ 3.64e−4951	≈ 6.48e−4966	≈ 2.25e−78 984

Today, almost all C compilers on current systems support at least the 1990 ISO C Standard, and, thus, obey parentheses. However, if you port software to an older system, perhaps one running on a simulator, you need to investigate whether your C compiler respects parentheses or not.

The code in the mathcw library has been carefully written to evaluate order-dependent expressions in multiple steps, storing intermediate results. Thus, our sample expression would be coded as `t = a + b; t -= c;`. However, on some systems, that contains a pitfall that we address in the next section.

4.5 The volatile type qualifier

Besides the critical evaluation-order dependence of numerical operations that we discussed in the previous section, there is a related issue that arises on some widely used platforms. Traditional programming languages assume that the storage precision is the same as the computational precision, because that was historically nearly always the case.

However, General Electric and Honeywell mainframes of the 1960s, the Motorola 68000 of the 1980s, the Intel IA-32 and IA-64, and the AMD AMD64 and Intel EM64T in IA-32 mode, all have floating-point CPU registers that are wider than storage words. The wider register provides additional bits for both the significand and the exponent, and most commonly, the extra precision and range are beneficial. Although the IBM POWER architecture supports the IEEE 754 32-bit format in storage, its floating-point instructions operate on 64-bit values, so it too effectively provides extended precision and range.

Nevertheless, in parts of the elementary-function algorithms, a series of operations are carried out that must be done in storage precision, and higher intermediate precision can produce incorrect answers.

Prior to the 1990 ISO C Standard, the only portable way for numerical programmers to deal with that issue was to force memory storage of intermediate results by a call to an external routine. Here is a simple example:

```
double
store(px)
double *px;
{
    return (*px);
}
...
a = b + c;          /* compute a = (((b + c) + d) + e) */
(void)store(&a);
a += d;
```

```
(void)store(&a);
a += e;
```

The `store()` function needs to be compiled separately, to prevent optimizing compilers from inlining it, and then optimizing it away.

Because each call to `store()` passes the *address* of the argument, rather than its *value*, the compiler is forced to store the variable in memory if it currently resides in a CPU register. Then, because the address of the variable was passed, the called routine could have modified the variable, and thus, the compiler must generate code to reload the variable from memory into a CPU register the next time that it is needed.

It is clearly tedious and error prone to have to program like that, and fortunately, a new feature introduced in the 1990 ISO C Standard provides a way to force variables into memory. The `volatile` type qualifier was added to the language primarily to deal with systems on which certain memory locations are special: reading or writing them may cause communication with external processors or devices. Many architectures of the 1960s and 1970s, including the DEC PDP-11 on which much of the early UNIX development was done, have that feature.

What `volatile` really does is to inform the compiler that the variable cannot be cached inside the CPU, but must instead be accessed in memory, where it might be changed unpredictably and without notification. That is exactly what we need for precision-controlled floating-point computation:

```
volatile double a;
double b, c, d, e;
...
a = b + c;          /* compute a = (((b + c) + d) + e) */
a += d;
a += e;
```

We still cannot write the sum in one statement, because the right-hand side could then be evaluated (incorrectly) in higher precision, but at least we can dispense with the messy `store()` wrappers.

The `mathcw` library code makes heavy use of both the `store()` function and the `volatile` qualifier to control intermediate precision. If the code is compiled on one of the architectures with extended registers with an old compiler that does not recognize the `volatile` keyword, you are at least alerted to the problem when the compiler rejects the code. You can temporarily define the keyword to an empty string with a `-Dvolatile=` compiler option to allow compilation to complete successfully, but you must then examine the package test reports carefully to see whether higher intermediate precision is (paradoxically) causing accuracy loss.

4.6 Rounding in floating-point arithmetic

Rounding of the last stored digit in any computed result clearly depends on accurate knowledge of at least a few additional digits, and to help our understanding, it is useful to show some details of how rounding works. We use that material in later chapters where we consider the problem of computation of correctly rounded elementary functions.

In binary arithmetic, with IEEE 754 default rounding, if the next bit beyond the last stored is zero, the stored result is already correctly rounded. Otherwise, the next bit is one, and there are then two cases to consider. If all bits beyond that one-bit are zero, then the default rounding rule in IEEE 754 says that the result should be rounded to the nearest even value, so that the last stored bit is zero. Otherwise, at least two of the additional bits are nonzero, and they represent a value greater than half of the last stored bit, so the result is rounded up by adding one to the last stored bit. Those cases are most easily understood by close examination of the bit patterns shown in **Table 4.3** on the next page.

Notice that rounding can affect more than the last two stored bits in only two of the twelve cases shown in **Table 4.3**. There is then a carry into bits before the last two, and the changed bits are indicated by c. In the worst case, all stored bits are affected, and the overflow of the first bit requires a right shift and an increase by one of the power-of-two exponent. The rightmost bit that is shifted off is discarded without further rounding of the stored bits, because the result is already properly rounded.

From the bit patterns in **Table 4.3**, we can see that only a few extra bits need to be known precisely to decide how to round the result, *except* for the more difficult case of *round to nearest with ties to even*. If we know just the first two extra bits, and in addition, whether all remaining bits are zero, then we have all that is required to handle even that

Table 4.3: Rounding in binary arithmetic for the default IEEE 754 rounding mode. The letters b and c represent arbitrary binary digits. The vertical bar in the first column separates stored bits from additional ones needed to determine rounding.

Stored and extra bits	Stored result	Action
$1.bbb\ldots b00\|0bbb\ldots$	$1.bbb\ldots b00$	discard extra bits
$1.bbb\ldots b01\|0bbb\ldots$	$1.bbb\ldots b01$	discard extra bits
$1.bbb\ldots b10\|0bbb\ldots$	$1.bbb\ldots b10$	discard extra bits
$1.bbb\ldots b11\|0bbb\ldots$	$1.bbb\ldots b11$	discard extra bits
$1.bbb\ldots b00\|1000\ldots$	$1.bbb\ldots b00$	round to nearest even
$1.bbb\ldots b01\|1000\ldots$	$1.bbb\ldots b10$	round to nearest even
$1.bbb\ldots b10\|1000\ldots$	$1.bbb\ldots b10$	round to nearest even
$1.bbb\ldots b11\|1000\ldots$	$cc.ccc\ldots c00$	round to nearest even and carry
$1.bbb\ldots b00\|1bbb\ldots$	$1.bbb\ldots b01$	add 1 to round up
$1.bbb\ldots b01\|1bbb\ldots$	$1.bbb\ldots b10$	add 1 to round up
$1.bbb\ldots b10\|1bbb\ldots$	$1.bbb\ldots b11$	add 1 to round up
$1.bbb\ldots b11\|1bbb\ldots$	$cc.ccc\ldots c00$	add 1 to round up (maybe carry)

difficult case. That information can be encoded in just three bits, which some architecture manuals call the *guard* bit, the *rounding* bit, and the *sticky* bit. The first, the guard bit, tells us whether we can truncate or must round. The second, the rounding bit, is needed in case normalization is required: an example is given shortly. The third, the sticky bit, is set to one if any other trailing bits are nonzero.

IEEE 754 binary arithmetic supports four rounding directions: *to* $-\infty$, *to zero* (truncation), *to* $+\infty$, and the default, *to nearest* (or *to nearest even number* in the event of a tie). All of those can easily be accommodated with the three extra bits, with roughly similar work, as summarized in Table 4.4 on the following page. Round to zero is the easiest to implement, because all nonstored bits are simply truncated, and many historical floating-point architectures take that approach.

Here is an example of the use of the three extra bits, adapted from Overton's short book [Ove01, page 35]. Consider a simple floating-point system with a sign bit, seven exponent bits, and nine significand bits (the first hidden), as might be used in a 16-bit digital signal processor. Let us first carry out the subtraction $1-(2^{-10}+2^{-16})$ in *exact* arithmetic, using a vertical bar to bound the stored bits:

$$\begin{aligned}
& 1.0000\ \ 0000\,|\,0000\ \ 0000_2\times 2^0 && \textit{pad for alignment},\\
-\ & 0.0000\ \ 0000\,|\,0100\ \ 0001_2\times 2^0 && \textit{align binary point},\\
=\ & 0.1111\ \ 1111\,|\,1011\ \ 1111_2\times 2^0 && \textit{subtract},\\
=\ & 1.1111\ \ 1111\,|\,0111\ \ 1110_2\times 2^{-1} && \textit{left shift to normalize},\\
=\ & 1.1111\ \ 1111_2\times 2^{-1} && \textit{truncate to stored result}.
\end{aligned}$$

Now repeat the subtraction, but this time, keep only two extra bits:

$$\begin{aligned}
& 1.0000\ \ 0000\,|\,00_2\times 2^0 && \textit{pad for alignment},\\
-\ & 0.0000\ \ 0000\,|\,01_2\times 2^0 && \textit{align binary point},\\
=\ & 0.1111\ \ 1111\,|\,11_2\times 2^0 && \textit{subtract},\\
=\ & 1.1111\ \ 1111\,|\,10_2\times 2^{-1} && \textit{left shift to normalize},\\
=\ & 10.0000\ \ 0000\,|\,10_2\times 2^{-1} && \textit{round to nearest even},\\
=\ & 1.0000\ \ 0000\,|\,01_2\times 2^0 && \textit{right shift to renormalize},\\
=\ & 1.0000\ \ 0000_2\times 2^0 && \textit{final stored result}.
\end{aligned}$$

The final result differs by one ulp (**u**nit in the **l**ast **p**lace) from the exact result, so two extra bits are *not* sufficient to produce correct rounding. Indeed, the same off-by-one-ulp result would have been obtained if we had kept as many as *seven* extra bits (i.e., all bits up to, but excluding, the last nonzero bit).

A third attempt uses a final sticky bit that only records whether any bits shifted off were nonzero, which is easily

Table 4.4: IEEE 754 rounding-mode actions for the eight possible settings of the three extra bits (**g**uard, **r**ound, and **s**ticky). The phrase *if odd* refers to the last storable bit.

sign	**G**	**R**	**S**	**to $-\infty$**	**to zero**	**to $+\infty$**	**nearest even**
$+$	1	1	1	truncate	truncate	add 1	add 1
$+$	1	1	0	truncate	truncate	add 1	add 1
$+$	1	0	1	truncate	truncate	add 1	add 1
$+$	1	0	0	truncate	truncate	add 1	add 1 if odd
$+$	0	1	1	truncate	truncate	add 1	truncate
$+$	0	1	0	truncate	truncate	add 1	truncate
$+$	0	0	1	truncate	truncate	add 1	truncate
$+$	0	0	0	truncate	truncate	truncate	truncate
$-$	0	0	0	truncate	truncate	truncate	truncate
$-$	0	0	1	subtract 1	truncate	truncate	truncate
$-$	0	1	0	subtract 1	truncate	truncate	truncate
$-$	0	1	1	subtract 1	truncate	truncate	truncate
$-$	1	0	0	subtract 1	truncate	truncate	subtract 1 if odd
$-$	1	0	1	subtract 1	truncate	truncate	subtract 1
$-$	1	1	0	subtract 1	truncate	truncate	subtract 1
$-$	1	1	1	subtract 1	truncate	truncate	subtract 1

done in hardware by OR'ing each discarded bit with the sticky bit:

$$\begin{array}{rlll} & 1.0000\ \ 0000\,|\,000_2 \times 2^0 & & \textit{pad for alignment}, \\ - & 0.0000\ \ 0000\,|\,011_2 \times 2^0 & & \textit{align binary point}, \\ = & 0.1111\ \ 1111\,|\,101_2 \times 2^0 & & \textit{subtract}, \\ = & 1.1111\ \ 1111\,|\,010_2 \times 2^{-1} & & \textit{left shift to normalize}, \\ = & 1.1111\ \ 1111_2 \times 2^{-1} & & \textit{truncate to stored result}. \end{array}$$

This time, the final result is exact. A more detailed analysis ([Par00, pages 303–304] or [Kor02, Section 4.6]) shows that the guard and rounding bits plus a sticky bit are always sufficient to support the IEEE 754 rounding rules.

Before IEEE 754 arithmetic, most historical floating-point architectures had at most two guard bits. The first IBM System/360 models had no guard bits, but later models added four such bits (one hexadecimal digit). Although that was an improvement, the lack of a sticky bit still prevents correct rounding.

The first Cray supercomputers had no guard bits, so $1-(1-\epsilon)$ produced 2ϵ instead of the correct ϵ. Some Cray models rounded operands after binary point alignment, computing $1-(1-\epsilon)$ as $1-1=0$. That could cause a conditional statement like `if (x != y) z = 1.0/(x - y)` to fail to protect against division by zero.

The IEEE 754 Standard leaves rounding behavior at the overflow and underflow limits (see **Section 4.9** on page 71 and **Section 4.11** on page 77) up to the implementation. The issue is whether rounding is done *before* the limit check, or *after*:

- At the underflow limit, if the smallest subnormal value (see **Section 4.12** on page 78) is multiplied by 3/4, the result is zero if the limit check is done first, but in default rounding, is the smallest subnormal number if rounding is first. A small test program run on several platforms shows that 68000, AMD64, IA-32, IA-64, PA-RISC, PowerPC, and SPARC CPUs round first, whereas Alpha and MIPS processors do the limit check first. It is even conceivable that the order might change between different models of the same CPU family.
- A similar test at the overflow limit is harder, because it requires setting a nondefault rounding mode, a topic that we address in **Chapter 5**. In addition, on Alpha processors, extra compilation flags are needed to allow dynamic rounding, as mentioned in the front matter on page xxxv.

  With downward rounding, multiplying the largest floating-point number by $1+\epsilon$ produces Infinity on processors that check for overflow before rounding: Alpha and MIPS processors do so. When rounding precedes the limit check, as it does on the other processors tested, the result is the largest floating-point number.

- In the only implementation of decimal floating-point arithmetic available at the time of writing this, rounding precedes the underflow check on the AMD64 CPU. That is likely to hold for other architectures where decimal arithmetic is provided in software. Investigation of rounding at the overflow limit is not yet feasible because of the temporary inability to set the decimal rounding mode.

Although few programs are likely to be affected by that implementation flexibility, it is yet another example of the difficulty of reproducing floating-point results across platforms.

4.7 Signed zero

IEEE 754 arithmetic, and also some historical arithmetic systems, use a sign-magnitude representation that permits a zero to be either positive or negative. The *signed zero* is an intentional feature of the IEEE 754 design, and it is expected that the sign of zero reflects how the zero was computed. If it is the underflow limit of a small negative value, it should be negative.

In most computations, the sign of zero is not significant, but that is not always so: see Kahan's famous paper [Kah87] on why the sign is sometimes critical.

It is regrettable that about one C/C++ compiler in ten handles signed zero incorrectly, based on scores of tests with this author's features package.[2]

4.7.1 Detecting the sign of zero

In comparisons, negative and positive zero are equal to each other, so you cannot use a condition like `(x == -0.0)` to test for a negative zero: it is true for a positive one as well.

Instead, to detect the sign of zero, you have to do one of two things. The best way is to use the `copysign(x,y)` function, which copies the sign of `y` to the value of `x`, without further examination of the bits of `y` to see whether it might be finite, infinite, or some kind of NaN. Another way to test the sign of zero is to reciprocate it, producing a signed infinity that can be compared against zero:

```
if (x == 0.0)
    return ((1.0/x) < 0.0) ? "negative" : "positive" ;
```

That works because $1/-0 \to -\infty$ and $1/+0 \to +\infty$, both of which differ from zero. However, on non-IEEE-754 floating-point architectures, the computation is likely to abort with a zero-divide error, but then, few of those architectures have signed zeros either.

From the Taylor series of the elementary functions shown in **Section 2.6** on page 10, many of them satisfy $f(x) \approx x$ for $x \to 0$. Thus, when $x = -0$, the function result should also be a negative zero. That is easy to achieve in C without having to extract the sign bit: simply write

```
if (x == 0.0)
    result = x;
```

instead of

```
if (x == 0.0)
    result = 0.0;
```

The first `if` statement preserves the sign of zero, whereas the second loses it.

IEEE 754 mandates that $\sqrt{-0} = -0$. That too can be handled easily by the technique of the preceding paragraph.

4.7.2 Signed-zero constants

It was the IEEE 754 designers' intent that signed-zero constants should preserve the sign, but sadly, many C compilers fail to do so. Thus, instead of

```
x = -0.0;
```

[2] Available at `http://www.math.utah.edu/pub/features/`.

you may have to write

```
x = 0.0;
x = -x;
```

Fortunately, the need for a negative zero constant is rare, so that subterfuge is not often needed. Nevertheless, it *is* an impediment to defining a preprocessor symbol or a named initialized constant with a negative zero value.

4.7.3 Arc tangent and signed zero

The two-argument form of the arc tangent is affected by signed zeros. The 1990 ISO C Standard specifies it like this:

> *The `atan2()` function computes the principal value of the arc tangent of `y/x`, using the signs of both arguments to determine the quadrant of the return value. A domain error may occur if both arguments are zero.*

The 1999 ISO C Standard description is similar, but omits the adjective *principal*. Notice the wording of the last sentence: whether or not an error occurs when both arguments are zero is not specified by the Standard, and is thus left up to the implementation.

The 1978 ANSI Fortran Standard says that both arguments must not be zero, but does not specify what should happen if they are. The 1995 ISO Fortran Standard is silent about the case where both arguments are zero. The 1982 ISO Pascal Standard [PAS82] defines only the single-argument arc tangent function, thereby avoiding the issue. The Java and C# library documents do not describe what the return value of `Math.atan2(0,0)` should be.

Because $0/0$ is mathematically undefined, there is no obvious answer to what the value of `atan2(0.0, 0.0)` ought to be. The Cody/Waite algorithm raises an error.

One common application of the two-argument arc-tangent function is for coordinate conversion from Cartesian (x, y) to polar (r, θ) (here, θ is the Greek letter *theta*):

```
r = hypot(x,y);    /* (usually) exception-free sqrt(x*x + y*y) */
theta = atan2(y,x);
```

Because the Cartesian point $(x, y) = (0, 0)$ is quite ordinary, there should be nothing exceptional in the polar point $(r, \theta) = (0, \theta)$, other than that the angle θ is arbitrary. Thus, $\theta = 0$ is a reasonable choice, and for that application at least, `atan2()` should not raise an exception for zero arguments.

Tests of the native math libraries on a wide range of UNIX systems produced the results shown in **Table 4.5** on the next page, and file `atan2x.h` in the `mathcw` package was modified to conform to the majority view, rather than recording a domain error in the global `errno` variable and returning a NaN.

The lesson for programmers is that care must be taken to avoid invoking an arc tangent function with two zero arguments, and shows the negative effect of programming-language standards that allow behavior to be defined by the implementation.

4.8 Floating-point zero divide

In historical floating-point architectures, division by zero was often a fatal error, or else it caused a trap that reported the error. The result of the operation could be the numerator, or zero, or the largest representable number, or some arbitrary value.

In IEEE 754 arithmetic, the *divbyzero* exception flag is set on division of a finite nonzero value by a zero of either sign, and the result is +Infinity if numerator and denominator are of like sign, and −Infinity if they are of opposite sign.

If the numerator is Infinity of either sign, no exception flag is set, and the result is +Infinity if numerator and denominator are of like sign, and −Infinity if they are of opposite sign.

If the numerator is a NaN, no exception flag is set, and the result is the numerator.

If the numerator is a zero of either sign, the signs are ignored, the *invalid* exception flag is set, and the result is a NaN.

Table 4.5: The nonstandardness of atan2(±0, ±0) in C and Java libraries. The `errno` handling is inconsistent: on several systems, the arc tangent functions in one or two precisions set it to `EDOM` (argument domain error), and in others, leave it unset! On AIX and SOLARIS, the return values are inconsistent among the three precisions.

System	`errno`	`atan2(0,0)`	`atan2(0,-0)`	`atan2(-0,0)`	`atan2(-0,-0)`
C/C++					
Apple Mac OS X PowerPC	unset	0	π	-0	$-\pi$
FreeBSD 4.4 IA-32	unset	0	π	-0	$-\pi$
FreeBSD 5.0 IA-32	unset	0	π	-0	$-\pi$
GNU/Linux Alpha, AMD64, IA-32, MIPS, and SPARC	unset	0	π	-0	$-\pi$
GNU/Linux IA-64	EDOM	0	π	-0	$-\pi$
HP HP-UX 10.20 PA-RISC	unset	QNaN	QNaN	QNaN	QNaN
HP HP-UX 11 PA-RISC and IA-64	unset	0	π	-0	$-\pi$
IBM AIX 4.1 PowerPC (`atan2()`)	EDOM	0	π	-0	$-\pi$
IBM AIX 4.1 PowerPC (`atan2l()`)	EDOM	QNaN	QNaN	QNaN	QNaN
NetBSD 1.6 IA-32	EDOM	0	0	0	0
NetBSD 1.6.2 VAX (no negative zero)	unset	0	0	0	0
NeXT Mach 3.3 Motorola 68040	unset	0	π	0	$-\pi$
OpenBSD 3.2 IA-32	EDOM	0	0	0	0
DEC/Compaq/HP OSF/1 4.0, 5.1 Alpha	EDOM	QNaN	QNaN	QNaN	QNaN
DEC TOPS-20 PDP-10 (no negative zero)	EDOM	0	0	0	0
SGI IRIX 6.5 MIPS	EDOM	0	π	-0	$-\pi$
Sun Solaris 7, 8, 9, 10 IA-32 and SPARC (`atan2()`)	unset	0	0	0	0
Sun Solaris 7, 8, 9, 10 IA-32 and SPARC (`atan2f()` and `atan2l()`)	unset	0	π	-0	$-\pi$
mathcw (all platforms)	unset	0	π	-0	$-\pi$
Java					
Apple Mac OS X PowerPC	n/a	0	π	-0	$-\pi$
DEC/Compaq/HP OSF/1 4.0, 5.1 Alpha	n/a	QNaN	QNaN	QNaN	QNaN
GNU/Linux AMD64, IA-32, IA-64	n/a	0	π	-0	$-\pi$
SGI IRIX 6.5 MIPS	n/a	0	π	-0	$-\pi$
Sun Solaris 7 SPARC	n/a	0	0	0	0
Sun Solaris 8, 9, 10 IA-32 and SPARC	n/a	0	π	-0	$-\pi$
hardware					
Intel IA-32, EM64T, and IA-64	unset	0	π	-0	$-\pi$

4.9 Floating-point overflow

An overflow occurs when a computed result is too large to represent. IEEE 754 arithmetic handles that gracefully by returning a special value, Infinity, and usually, setting a sticky exception flag to record its occurrence.

The IEEE 754 overflow flag is a complex flag, because its setting depends on the rounding mode. Here are the conditions in which it is set:

- *Round to nearest* produces an Infinity from overflow.
- *Round to zero* produces ±`MAXNORMAL` from overflow.
- *Round to negative infinity* produces +`MAXNORMAL` from positive overflow.
- *Round to positive infinity* produces −`MAXNORMAL` from negative overflow.

However, if the result is infinite because an operand is infinite, the flag is *not* set. Thus, `Infinity * 2` does not set the flag, but `MAXNORMAL * 2` does. If the result is infinite because of division by zero, the *overflow* flag is not set, but the *divbyzero* flag is set if the numerator is nonzero, and otherwise, the *invalid* flag is set.

Many historical floating-point systems simply terminated the job at the first overflow, although a few provided the ability to trap the overflow and continue with a substitute value, such as the largest representable number.

Some systems did not detect the overflow, and the returned result was the result of dropping the overflowed exponent bits, usually resulting in an effectively tiny value. Here is a run of a small test program in C that prints positive powers of two on a PDP-10:

```
@cc ofl.c
@ofl
   0    1
   1    2
   2    4
   3    8
   4    16
   5    32
...
 124    2.1267647933e+37
 125    4.2535295865e+37
 126    8.507059173e+37
 127    1.4693679385e-39    # overflow wraps!
 128    2.9387358771e-39
 129    5.8774717541e-39
...
```

By contrast, the Fortran compiler on that system correctly traps the overflow, capping the results at the maximum representable number, and reporting the total number of exceptions on job completion:

```
@exec ofl.for
FORTRAN: OFL
OFL
LINK:   Loading
[LNKXCT OFL execution]
  0       1.0000000000
  1       2.0000000000
  2       4.0000000000
  3       8.0000000000
...
    126    0.85070591730E+38
%Floating overflow at OFL+14 in OFL (PC 174)
  127    0.17014118219E+39
%Floating overflow at OFL+14 in OFL (PC 174)
  128    0.17014118219E+39
  129    0.17014118219E+39
...
  139    0.17014118219E+39
  140    0.17014118219E+39
CPU time 0.06   Elapsed time 0.05
[15 Floating overflows]
```

By default, only the first two exceptions are reported to the terminal, along with the program locations where they occurred.

4.10 Integer overflow

In the previous section, we discussed floating-point overflow, but it is important to remember that integers can overflow as well, sometimes with disastrous consequences. Appendix I on page 969 summarizes the main features of integer arithmetic, and may provide helpful background. There are several important points to note about the limitations of integer arithmetic:

- The range of representable integers is usually smaller than that of C's `double`, and on common desktop systems, is insufficient to count the number of people on Earth, or the number of nanoseconds in a minute, or the U.S. national debt in dollars, or even the annual budget of a large European organization.
- There is no hardware representation of an integer infinity.
- In the now almost universally used two's-complement representation of binary integers, negation of the most negative number overflows, returning that negative value. That leads to the unexpected result that the absolute value of the most negative number is negative.
- Most CPU architectures cannot interrupt, trap, or otherwise signal the occurrence of integer overflows. Instead, on some, integer-arithmetic instructions set condition code flags that must be explicitly tested to detect exceptions, but generating additional code for such testing is rarely even a user-selectable compiler option.
- Integer arithmetic on MIPS processors does not set any flags, but the authors of *MIPS RISC Architecture* show how overflow can be detected after integer add, subtract, and multiply operations [KH92, page C-8].
- The Java Virtual Machine specification of integer arithmetic is that it returns the low-order 32 bits of the exact two's-complement operation. No status flags are set, nor is any trap handler invoked on overflow.
- Almost all programming languages effectively suppress trapping and reporting of integer overflows, often by defining the behavior to be implementation dependent, and thus, relegated to whatever the designers of the underlying hardware chose to do.
- Unsigned integer arithmetic in C and C++ is defined as modular arithmetic, so by definition, it cannot overflow.

Some of those points are illustrated in this report of nonsensical results from a small test program run on a machine with 32-bit `int` values:

```
% cc intofl.c && ./a.out
2147483647 +          1 = -2147483648
2147483647 * 2147483647 =           1
-(-2147483648)          = -2147483648
```

In a small programming project implemented in about 50 different programming languages [Bee04b], this author found only a single language, Algol 60, that detected an integer overflow.

Even the carefully written TeX typesetting system does not reliably detect integer overflow: it does so on multiplication and division, but not on addition or subtraction:

```
% tex
This is TeX, Version 3.1415 (C version 6.1.2.1)
**\relax
*\count0 = 2147483647
*\multiply \count0 by 2
! Arithmetic overflow.

*\showthe \count0
> 2147483647.

*\divide \count0 by 0
! Arithmetic overflow.

*\showthe \count0
> 2147483647.

*\advance \count0 by 1
*\showthe \count0
> -2147483648.               % overflow wraps to negative value

*\dimen0 = 16383pt
```

```
*\advance \dimen0 by \dimen0
*\showthe \dimen0
> 32766.0pt.

*\advance \dimen0 by 2pt
*\showthe \dimen0
> --32768.0pt.              % notice the double minus

*\dimen0 = 16383pt
*\advance \dimen0 by 1pt
*\showthe \dimen0
> -32767.0pt.

*\dimen0 = 3\dimen0
! Dimension too large.
```

Examination of programming language standards or specifications for C, C++, Fortran, and Pascal shows that they either do not mention integer overflow, or else they define the behavior to be implementation dependent. Java's specification says that built-in integer operators do not indicate overflow in any way. Experiments show that some C implementations detect an integer divide-by-zero and signal a `SIGFPE` exception, whereas others silently produce an arbitrary result, such as zero, and continue execution.

4.10.1 Preventing integer overflow

It is instructive to consider how to program with integer arithmetic if you need to detect integer overflow *before* it happens. What is required is a test function for each operation that can detect an overflowed result without actually generating one. We could then write safe code like this:

```
if (is_add_safe(a,b))
    c = a + b;
else
    error("a + b would overflow");
```

A suitable definition of the `error()` macro can use the preprocessor macros `__FILE__` and `__LINE__` to print a message identifying the source-code location, and then terminate the job.

In practice, for C99, we need versions of the test functions for arguments of types `int`, `long int`, and `long long int`. The `short int` type is not supported by any standard numerical library functions, so we ignore it. By analogy with the floating-point library, we give the test functions names with suffix letters *l* and *ll*: for example, `is_add_safe()` for `int` arguments, `is_add_safel()` for `long int` arguments, and `is_add_safell()` for `long long int` arguments.

As we do for the floating-point library routines, our implementations use a generic integer type, `int_t`, with values in the range `[INT_T_MIN,INT_T_MAX]`. We hide constant type suffixes in a suitably defined wrapper macro `INT()`. Short wrapper files then provide the required type definitions and value ranges before including the code body from a common header file.

Two's-complement representation destroys the symmetry about zero, and makes the test functions more complex. However, all modern desktop and larger machines use two's-complement for binary integer arithmetic, so our code must cater to it, and must be revised if it is ever used on a computer with sign-magnitude, one's-complement, or any other representation of integer values. Important historical architectures that used one's-complement integer arithmetic include the 60-bit CDC 6000 and 7000 series and the 36-bit Univac 1100 series. Only the latter had the C language, and then just on those few machines that ran the UNIX operating system [BHK$^+$84].

We handle the operations of absolute value, add, divide, multiply, negate, remainder, and subtract, and present the test functions now in alphabetical order in the following subsections.

4.10.1.1 Safe integer absolute value

Absolute value is safe, except for the most negative integer:

```
int
IS_ABS_SAFE(int_t a)
{   /* return 1 if |a| will not overflow, 0 otherwise */
    return (a != INT_T_MIN);
}
```

4.10.1.2 Safe integer addition

Addition is safe when the signs differ, but otherwise requires more care:

```
int
IS_ADD_SAFE(int_t a, int_t b)
{   /* return 1 if a + b will not overflow, 0 otherwise */
    int result;

    if ( (a >= INT(0)) != (b >= INT(0)) )
        result = 1;     /* signs differ */
    else if ( (a >= INT(0)) && ((INT_T_MAX - b) >= a) )
        result = 1;     /* both nonnegative */
    else if ( (a < INT(0)) && ((INT_T_MIN - b) <= a) )
        result = 1;     /* both negative */
    else
        result = 0;     /* a + b overflows */

    return (result);
}
```

4.10.1.3 Safe integer division

Although division is harder than addition and multiplication, its overflow test is one of the simplest, because we just have to avoid two possibilities: division by zero, and the case `INT_T_MIN/-1`.

```
int
IS_DIV_SAFE(int_t a, int_t b)
{   /* return 1 if a / b will not overflow, 0 otherwise */
    return ( (b != INT(0)) &&
             !( (b == INT(-1)) && (a == INT_T_MIN) ) );
}
```

4.10.1.4 Safe integer multiplication

Multiplication is by far the hardest to test for overflow, and we have to take special care that the tests themselves do not overflow.

```
int
IS_MUL_SAFE(int_t a, int_t b)
{   /* return 1 if a * b will not overflow, 0 otherwise */
    int result;

    if ( (a == INT(0)) || (b == INT(0)) )
        result = 1;     /* product is zero */
    else if ( (a >= INT(0)) == (b >= INT(0)) )
    {                   /* signs identical */
        if ( (a > INT(0)) && (b <= (INT_T_MAX / a)) )
            result = 1;
        else if ( (a < INT(0)) &&
                  (a != INT_T_MIN) &&
                  (b != INT_T_MIN) &&
                  (b >= (INT_T_MIN / -a)) )
            result = 1;
        else
            result = 0;
    }
    else
    {                   /* signs differ */
        if ( (a > INT(0)) && (b >= (INT_T_MIN / a)) )
            result = 1;
        else if ( (a < INT(0)) && (a >= (INT_T_MIN / b)) )
            result = 1;
        else
            result = 0;
    }

    return (result);
}
```

In the worst case, we have to do a dozen numerical operations to determine whether integer overflow could occur.

4.10.1.5 Safe integer negation

Although negation is logically distinct from absolute value, its implementation is identical:

```
int
IS_NEG_SAFE(int_t a)
{   /* return 1 if -a will not overflow, 0 otherwise */
    return (a != INT_T_MIN);
}
```

4.10.1.6 Safe integer remainder

Remainder is safe whenever division is safe, but allows one more safe case, `INT_T_MIN % -1`, that we have to test for:

```
int
IS_REM_SAFE(int_t a, int_t b)
{   /* return 1 if a % b will not overflow, 0 otherwise */
    return ( IS_DIV_SAFE(a,b) ||
             ( (a == INT_T_MIN) && (b == INT(-1)) ) );
}
```

4.10.1.7 Safe integer subtraction

The last test function, for subtraction, can use the test for addition, except when the second argument cannot be safely negated:

```
int
IS_SUB_SAFE(int_t a, int_t b)
{   /* return 1 if a - b will not overflow, 0 otherwise */
    return (IS_NEG_SAFE(b) ? IS_ADD_SAFE(a,-b) : (a < INT(0)));
}
```

4.10.1.8 Safe integer operations: a retrospective

We do not include a test function for the left-shift operator, because that operator is most commonly used for bit-field access in *unsigned* integer types, where by definition of unsigned integer arithmetic, there is no possibility of overflow.

Because those test functions are nontrivial to program, they are good candidates for library inclusion, and the mathcw library includes them as a bonus to users, with a separate header file, `issafe.h`, to define their function prototypes.

4.11 Floating-point underflow

An underflow occurs when the computed result is too small to represent. Most historical systems replace the underflowed value with zero and continue the computation, sometimes after issuing a warning message. A few do not detect the underflow, so that the exponent simply wraps from a small negative value to a large positive value. Here is a run of a small test program in C that prints negative powers of two on a PDP-10:

```
@cc ufl.c
@ufl
   0    1
  -1    0.5
  -2    0.25
  -3    0.125
  -4    0.0625
  -5    0.03125
...
-127    5.8774717541e-39
-128    2.9387358771e-39
-129    1.4693679385e-39
-130    8.507059173e+37    # underflow wraps!
-131    4.2535295865e+37
-132    2.1267647933e+37
```

Such behavior can certainly make loop-termination tests unreliable.

The Fortran implementation on the PDP-10 correctly traps the underflow, replaces it with zero, and reports a summary of exceptions on job completion:

```
@exec ufl.for
LINK:   Loading
[LNKXCT OFL execution]
   0      1.0000000000
  -1     0.50000000000
  -2     0.25000000000
  -3     0.12500000000
...
-129     0.14693679385E-38
%Floating underflow at OFL+14 in OFL (PC 174)
-130     0.00000000000E+00
-131     0.00000000000E+00
...
-139     0.00000000000E+00
-140     0.00000000000E+00
CPU time 0.05   Elapsed time 0.10
[1  Floating underflow]
```

In the next section, we describe how underflow is handled in IEEE 754 arithmetic.

4.12 Subnormal numbers

IEEE 754 arithmetic is unusual compared to its predecessors in that, once a normalized value reaches the minimum exponent, instead of underflowing abruptly to zero, the normalization requirement is relaxed, and with the exponent fixed, the leading significand bit is permitted to march off to the right, reducing precision until finally no bits are left, and the result is zero. Such numbers are called *subnormal*, and the process in which they are generated is called *gradual underflow*.

The original term used in the IEEE 754 Standard for those numbers was *denormalized*, but that term is now deprecated. However, it is still commonly found in computer-architecture manuals and hardware descriptions.

Gradual underflow preserves an important mathematical property in floating-point numbers: if $x \neq y$, then it is always true that $x - y \neq 0$. However, on systems that practice *abrupt underflow*, two small numbers can differ, yet when subtracted, their difference is zero. Abrupt underflow is almost universal on historical computers, yet some of them lack a floating-point comparison instruction, using subtraction and a test for zero instead. That produces anomalies that we discuss further in **Appendix H.2** on page 952.

Here is a hoc example that displays decreasing powers of two in 80-bit IEEE 754 arithmetic, showing the results in decimal, in hexadecimal storage representation, and in C99 hexadecimal floating-point, which represents numbers as a significand and a power of two:

```
hoc80> x = 2**(-16350)
hoc80> for (k = -16350; k >= -16446; --k)            \
hoc80> {                                             \
hoc80>     printf("%6d %d %12.5g %s %a\n",           \
hoc80>            k, issubnormal(x), x, ftoh(x), x)
hoc80>     x *= 0.5
hoc80> }
-16350 0  1.444e-4922 0021_80000000_00000000 0x8p-16353
-16351 0 7.2201e-4923 0020_80000000_00000000 0x8p-16354
-16352 0   3.61e-4923 001f_80000000_00000000 0x8p-16355
...
-16380 0 1.3448e-4931 0003_80000000_00000000 0x8p-16383
-16381 0 6.7242e-4932 0002_80000000_00000000 0x8p-16384
-16382 0 3.3621e-4932 0001_80000000_00000000 0x8p-16385
-16383 1 1.6811e-4932 0000_40000000_00000000 0x4p-16385
-16384 1 8.4053e-4933 0000_20000000_00000000 0x2p-16385
-16385 1 4.2026e-4933 0000_10000000_00000000 0x1p-16385
-16386 1 2.1013e-4933 0000_08000000_00000000 0x0.8p-16385
-16387 1 1.0507e-4933 0000_04000000_00000000 0x0.4p-16385
-16388 1 5.2533e-4934 0000_02000000_00000000 0x0.2p-16385
-16389 1 2.6266e-4934 0000_01000000_00000000 0x0.1p-16385
-16390 1 1.3133e-4934 0000_00800000_00000000 0x0.08p-16385
...
-16443 1 1.4581e-4950 0000_00000000_00000004 0x0.000000000000004p-16385
-16444 1 7.2904e-4951 0000_00000000_00000002 0x0.000000000000002p-16385
-16445 1 3.6452e-4951 0000_00000000_00000001 0x0.000000000000001p-16385
-16446 0            0 0000_00000000_00000000 0x0p+0
```

In the 80-bit format, the sign and exponent occupy the first four hexadecimal digits, and there is no hidden leading significand bit. The onset of gradual underflow occurs at 2^{-16383}, when the stored biased exponent reaches zero, and the significand is no longer normalized. Although C99 does not require that subnormal numbers printed with the %a format item be displayed with leading zeros, the GNU/LINUX implementation used in that example does so.

Subnormal numbers are relatively expensive to compute on some architectures, so compilers on those systems may produce instructions that flush underflows abruptly to zero without generating subnormals; it may take a special compile-time option to enable them. Alternatively, certain optimization levels may remove support for subnormals. On Silicon Graphics IRIX MIPS, subnormals are not supported at all unless library calls to get_fpc_csr() and set_fpc_csr() are invoked at run time; the command man sigfpe gives details.

In conforming IEEE 754 arithmetic, an *underflow* exception is recorded in a sticky exception flag when the value is too small to be represented as a normal number, and *in addition*, there is loss of accuracy because the result cannot be represented exactly as a subnormal number. Thus, computations of increasingly negative powers of two generate normal, then subnormal, results without ever setting the *underflow* flag. Only when the smallest subnormal is multiplied by $1/2$, producing zero, are the *inexact* and *underflow* flags set. Similarly, when subnormals are supported, `MINNORMAL / 4` does *not* set the underflow flag, because it is an exactly representable subnormal, but `MINNORMAL / 3` *does* set the underflow flag, because it is not exactly representable as a subnormal.

For most of the elementary functions, the algorithms are not aware of computations near the underflow or overflow limits. However, for `atan2(y,x)`, the Cody/Waite recipe explicitly checks for small and large ratios y/x and handles those cases separately, returning either ± 0 or $\pm\pi$ for small ratios, and $\pm\pi/2$ for large ones. Because the Taylor series (see **Section 2.6** on page 10) tells us that $\operatorname{atan} x \approx x$ as $x \to 0$, the `mathcw` code changes the Cody/Waite algorithm for `atan2(y,x)` to replace 0 by y/x. That may or may not be zero, but if it is representable as a subnormal, it is correct to the precision of the subnormal, whereas an answer of 0 is simply wrong.

The most controversial feature of IEEE 754 arithmetic was, and remains, gradual underflow. Some hardware designers argue that it is expensive to implement, costing chip circuitry that could better be used for other purposes. Timing tests of arithmetic with and without subnormals on a wide range of platforms show large differences in the relative cost of subnormals, but on IBM PowerPC, there is effectively no penalty for subnormals. That suggests that chip implementation cost need not be an issue.

Numerical programmers raise concerns about subnormal values that are later scaled to larger values with silently reduced precision, possibly contaminating the remainder of the computation. Although it is true that two sticky exception flags record the generation of subnormals that have lost nonzero bits, most numerical programs never check or report exception flags. In any event, the *inexact* flag is almost certain to have been set by other computations with normal numbers. Nevertheless, detailed numerical analysis argues for generation of subnormals instead of abrupt underflow [Coo81b, Dem81]. We discuss subnormals further in **Section 5.4** on page 107.

Compilers on several platforms offer a choice between subnormals and flush-to-zero underflow, so on those systems, one can make a numerical experiment by running a program both ways. If the program output from the two tests differs substantially, then it is clear that subnormals have a noticeable effect, and also that cross-platform portability is questionable. It may be possible to introduce suitable exact scaling at key points in the code to avoid the subnormal regions, as we do for some of the Taylor-series computations for small arguments. Alternatively, one can convert all, or parts, of the program to use a higher precision with a wider exponent range. Some compilers provide an option to do that without code changes.

4.13 Floating-point inexact operation

The most commonly set exception flag in most floating-point operations is the *inexact* flag, which is set whenever the computed result is inexact because of rounding. That can be done by a logical *OR* of the flag with each of the **G**, **R**, and **S** bits. The flag is also set whenever overflow occurs without an exception trap, and when underflow happens.

However, at least one software implementation of IEEE 754 arithmetic fails to conform: 128-bit arithmetic on Compaq OSF/1 Alpha does not set *inexact* for either overflow or underflow.

The primary use of that flag is in carefully written software libraries where exact floating-point arithmetic is carried out inside a block of code. The *inexact* flag can be cleared before executing the block, and tested afterward to make sure that it is still clear. If it is found to be set, then at least one operation incurred rounding, and remedial action must then be taken to do the calculation some other way. Such code is clearly unusual.

4.14 Floating-point invalid operation

The IEEE 754 and 854 *NaN* (Not a Number) is a special value that is generated when the result of an operation cannot produce a well-defined answer, or at least a clear indication of its limiting value.

The Standards define two types of NaN: quiet and signaling, and require that they be distinguished by a bit in their significands, rather than the perhaps obvious choice of using the sign bit, because the sign of a NaN is never significant.

When there is a need to distinguish between the two kinds in writing, we follow IEEE 754-2008, and use *QNaN* and *SNaN*, although some authors call them *NaNQ* and *NaNS* instead.

Because the floating-point system in the Intel IA-32 was based on an early draft of the IEEE 754 Standard, and put into commercial production five years before that Standard was published, it has only one kind of NaN. Most hardware architecture families designed since 1985 supply both kinds.

The Java and C# virtual machines [LY97, LY99, HWG04, C#03b, CLI05] provide only one kind of NaN, perhaps because their designers felt that it would require unacceptable additional overhead to implement both quiet and signaling NaNs when the underlying hardware might supply only one.

The IEEE 754 Standard has two kinds of NaNs because the designers felt that it would be useful to have quiet ones that make their way silently through arithmetic operations, or are produced as results of such operations, as well as noisy ones that make their presence known by causing a trap when they are encountered.

For example, if storage locations of floating-point variables are initialized to signaling NaNs on program startup, then any attempt to use the value of a variable before it has been assigned a valid value could be caught by the hardware and reported to the user. That practice was common, for example, on CDC 6000 and 7000 series computers in the 1960s and 1970s. They have a special floating-point representation called *Indefinite* that behaves much like a NaN, and influenced the IEEE 754 design (see **Appendix H.1** on page 949).

The *invalid* exception flag, one of the five standard floating-point sticky exception flags, records the occurrence of a NaN in an arithmetic operation, but the circumstances in which that happens are complex. In a conforming implementation, the *invalid* exception flag is set by any of these conditions:

- An operand is a signaling NaN.
- Subtraction of like-signed Infinity, or addition of opposite-signed Infinity.
- Multiplication of zero by Infinity.
- Division of zero by zero, or Infinity by Infinity, irrespective of sign.
- Square root of a negative nonzero number.
- Conversion of a binary floating-point value to an integer or to a decimal format when the result is not representable.
- Ordered comparison (less than, less than or equal, greater than, or greater than or equal) when either, or both, of the operands are NaNs.

In particular, common cases like `QNaN + QNaN`, `QNaN - QNaN`, `QNaN * QNaN`, and `QNaN / QNaN` *do not* set the *invalid* flag, but the corresponding expressions with SNaN in place of QNaN *do* set it.

Comparisons for equality and inequality do not set the *invalid* flag.

Because those rules are complex, it is not surprising that they are sometimes disobeyed. The IA-32 architecture, for example, does not set the *invalid* flag in every one of those cases.

When the definition of an elementary function calls for setting the *invalid* flag, the practice in the `mathcw` library is to call one of the `qnan()` functions. They compute `0.0/0.0`, and that operation generates a QNaN, and as a required side effect, sets the *invalid* flag.

4.15 Remarks on NaN tests

NaNs have the property that they are not equal to anything, even themselves. The intent of the IEEE 754 designers was that the feature could be used to test for a NaN in a way that is fast, universal, programming-language independent, and that avoids any need to inspect the underlying bit representation. For example, in the C-language family, the expression `x != x` is true if, and *only if*, `x` is a NaN. Regrettably, compilers on several systems botch that, either by incorrectly optimizing it away at compile time, or by using integer instructions for comparison of floating-point values, or by failing to test the proper flags after a correct floating-point comparison.

As a result, we need a separate `isnan()` function, instead of simple inline comparisons, for NaN tests. The `mathcw` implementation of that function makes a one-time test at run time to see if NaN comparisons are mishandled. If they are, it selects a more complex algorithm that is platform independent, without having to examine the underlying bit representation. That algorithm computes the answer as `((x != 1.0) && (x != -1.0) && (x == 1.0/x))`, but in a convoluted way that works around the compiler errors. The equality test in the third part is written to invert the result of faulty code generation that incorrectly reports that NaNs are equal.

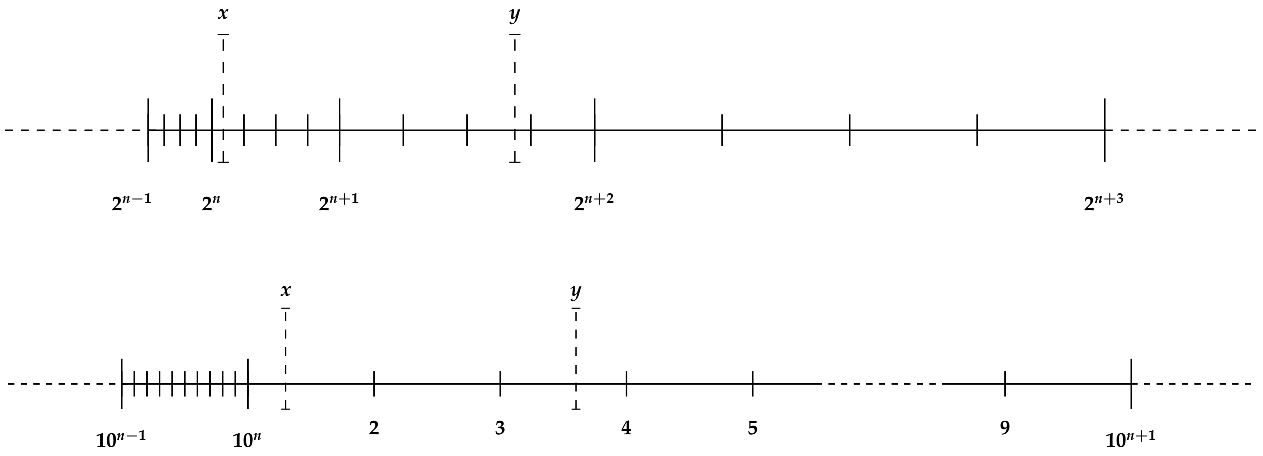


Figure 4.2: Number spacing in three-bit binary ($\beta = 2, t = 3$) and one-digit decimal ($\beta = 10, t = 1$) floating-point systems. Machine numbers exist only at tics on the horizontal axis. The gap between any pair of adjacent tics is one *ulp*. The size of the ulp changes by a factor of β at the longer tics marking the exponent boundaries.
The big machine epsilon, ϵ, is the ulp to the right of the tic at β^0, and the little machine epsilon, ϵ/β, is the ulp to the left of that tic.
An arbitrary (infinite-precision) real number x may lie between two adjacent machine numbers, and can only be approximated by one of those numbers, according to the choice of rounding mode.
With x positioned as shown, Kahan's ulp is the smaller one to the left of the exponent boundary, whereas the Muller–Harrison ulp is the gap containing x.
By contrast, for most real numbers, such as that labeled y, both ulp measures are identical, and equal to the containing gap.

4.16 Ulps — units in the last place

To assess the accuracy of floating-point computations, it is useful to remove the dependence on number range and precision by normalizing errors to multiples of a machine epsilon, such that one unit of the measure corresponds to a change of one unit in the last base-β digit.

The commonly used acronym *ulp* was introduced by William Kahan in unpublished work in 1960 with this original definition [Kah04a]:

> `ulp(x)` *is the gap between the two floating-point numbers nearest x, even if x is one of them.*

Here, x is an arbitrary (infinite-precision) real number, and thus unlikely to be exactly equal to a machine number.

Different authors have chosen varying definitions of the ulp measure, so be cautious when comparing results from the literature, because the definitions may disagree near exponent boundaries, and at the underflow and overflow limits.

The introduction of Infinity in IEEE 754 arithmetic caused Kahan to modernize his definition of the ulp:

> `ulp(x)` *is the gap between the two* finite *floating-point numbers nearest x, even if x is not contained in that interval.*

That revision shows how to handle x values above the overflow limit: the Kahan ulp is then the gap between the largest finite floating-point number and the machine number just below it. Near the end of this section, we show how to compute the Kahan-ulp function.

The diagrams in **Figure 4.2** may be helpful for visualizing the ulp unit.

One simple use of the ulp is in the *relative ulps* function that divides the relative error by the big epsilon, an expression that is readily computed from known quantities. We could write a draft version of such a function for one data type in C like this:

```
#include <float.h>      /* needed only for FLT_EPSILON */
#include <math.h>       /* needed for fabs() prototype */

float
relulpsf(float approx, double exact)
{
    return (float)fabs(((double)approx - exact) / exact) / FLT_EPSILON;
}
```

Here, the argument `approx` is the value computed in data type `float`, and `exact` is an estimate of the exact answer, rounded to the nearest machine number in a higher-precision data type. We are rarely interested in more than the first few digits of the answer from that function, so the higher precision of the second argument only needs to supply a few extra digits beyond working precision.

Although it is simple to write, our `relulpsf()` function has several defects:

- If the second argument is zero, the function produces Infinity or a NaN in IEEE 754 arithmetic, and may terminate the job with a *divide-by-zero* error in older arithmetic designs.
- If the second argument has the value $1.0 \times \beta^n$, we are at a boundary where the size of the machine epsilon increases by a factor of β, so there is an ambiguity from the choice of that epsilon.
- In IEEE 754 t-digit arithmetic, if the second argument is subnormal when coerced to the data type of the first argument, then it has fewer than t significant digits, so we ought to reduce the epsilon accordingly.
- When the arguments are near the value $1.0 \times \beta^n$, the computed result is a good approximation to the error in the last digit. However, with fixed exponent, as the arguments increase to their representable maxima just below $\beta \times \beta^n$, the division by the second argument makes the computed value almost β times smaller than the machine epsilon. The returned value is then an overly optimistic estimate of the error.
- When the approximate value is computed in the highest working precision, there is no data type available to represent a rounded exact value.
- If we try to solve the preceding problem by representing the exact value as a sum of high and low parts in working precision (here, `float`), rather than the next higher precision, then we introduce new difficulties:
  - ❑ The user interface is unpleasantly complex.
  - ❑ We cause new defects when the low part becomes subnormal or underflows to zero, because we lose some or all of the extra digits needed to compute the numerator difference, `approx - exact`, accurately.
  - ❑ We are unable to closely represent the exact result when it is too large or too small for the working-precision format.

  Both extended precision and extended exponent range are required features of the data type of the `exact` argument. Regrettably, some computer designs, both historical and modern, fail to provide the required extended range.

In summary, the computation of the error measured in units in the last place is not the simple task that we expected when we coded the single-statement function body of `relulpsf()`.

Jean-Michel Muller wrote a useful technical report [Mul05] that compares several existing textbook definitions of the ulp measure. We leave the mathematical details to that report, but reproduce here some of its important results.

Muller proposes a function `ulp(x)` that returns the value of one unit in the last digit of x, defined like this [Mul05, page 13]:

> *If x is a real number that lies between two finite consecutive floating-point numbers a and b, without being equal to one of them, then* `ulp(x)` $= |b - a|$, *otherwise* `ulp(x)` *is the distance between the two* finite *floating-point numbers nearest x. Moreover,* `ulp(NaN)` *is NaN.*

Muller credits the behavior of that definition away from the overflow limit to John Harrison, and at or above the overflow limit, to William Kahan, and gives Maple code to implement both functions.

That definition is equivalent to Kahan's when x is a machine number. However, the two specifications differ when x is not a machine number, and x lies just above an exponent boundary. In that case, Kahan's `ulp(x)` is the gap *below* that boundary, whereas the Muller–Harrison `ulp(x)` is the gap *above* that boundary. For example, in a three-digit decimal system with $x = 1.004$, the Kahan ulp is 0.001, because the nearest machine numbers are 0.999 and 1.00, whereas the Muller–Harrison ulp is 0.01. With $x = 0.9995$, both produce an ulp of 0.001, and with $x = 1.005$, both yield an ulp of 0.01.

Careful reading of Muller's definition shows us how to handle various cases:

- The definition involves only the (positive) distance between two machine numbers, so `ulp(-x)` $=$ `ulp(x)`, and `ulp(x)` > 0.
- When x lies away from an exponent boundary (that is, it has the value $s \times \beta^n$ where s is in $(1, \beta)$), `ulp(x)` returns β^{n-t+1}, which is the same as $\beta^n \epsilon$.
- When x lies at an exponent boundary (that is, it has the value $1.0 \times \beta^n$), we choose $b = x$, so `ulp(x)` returns β^{n-t}, equivalent to $\beta^n \epsilon / \beta$.
- If x is larger than the overflow limit, then `ulp(x)` returns $\beta^{\mathtt{EMAX}-t+1}$, where `EMAX` is the exponent of the largest normal number.
- If x is zero, or less than or equal to the smallest normal number, the function returns the smallest subnormal value, equal to $\beta^{\mathtt{EMIN}-t+1}$, where `EMIN` is the exponent of the smallest normal number.

  Because we have to deal with systems that lack support for subnormals, a situation that Muller does not consider, the appropriate return value on such machines is the smallest normal value, $\beta^{\mathtt{EMIN}}$.

As long as the argument is not a NaN, the function value is always a *finite nonzero number* of the form $1.0 \times \beta^m$, for some integer m. Multiplication and division by `ulp(x)` is then *exact* in the absence of underflow or overflow.

We can better assess the error in a computed value with a revised function like this:

```
float
ulpsf(float approx, double exact)
{
    return (float)fabs((double)approx - exact) / ulp(exact);
}
```

The singular and plural of ulp distinguish the names of the functions of one and two arguments.

Muller introduces four rounding functions, `RN(x)`, `RU(x)`, `RD(x)`, and `RZ(x)`, that round to nearest, up (to $+\infty$), down (to $-\infty$), and to zero, respectively, and proves that his definition leads to several important properties of the ulp function for an exact number x and a machine number X that approximates it. As usual in logic, a statement that A implies B does not mean that B also implies A: fish are swimmers, but not all swimmers are fish. To disprove the converse relation, we only need to supply a single counterexample that shows the relation to be false: penguins will do!

Here is Muller's list of properties for the Muller–Harrison `ulp(x)` function, and for Kahan's revised function, temporarily renamed to `KahanUlp(x)`:

❶ If $\beta = 2$ and $|X - x| < \frac{1}{2}\mathtt{ulp(x)}$ then $X = \mathtt{RN(x)}$.

The restriction to binary arithmetic is bothersome, but essential. Muller gives this general counterexample for $\beta > 2$:

$$\begin{aligned} X &= 1 - \beta^{-t}, & \mathtt{RN(x)} &= 1, \\ x &< 1 + \beta^{-t}/2, & \tfrac{1}{2}\mathtt{ulp(x)} &= \tfrac{1}{2}\beta^{1-t} \\ |X - x| &< \tfrac{3}{2}\beta^{-t}, & &= \tfrac{1}{2}\beta\beta^{-t}. \end{aligned}$$

The two $<$ operators can be changed to $\leq$ when $\beta > 3$. The precondition is satisfied, but $X \neq \mathtt{RN(x)}$.

A numeric counterexample in a three-digit decimal system is

$$X = 0.999, \qquad x = 1.0005, \qquad |X - x| = 0.0015, \qquad \tfrac{1}{2}\mathtt{ulp(x)} = 0.005, \qquad \mathtt{RN(x)} = 1.$$

❷ For any integer base β, if $X = \mathtt{RN(x)}$ then $|X - x| \le \frac{1}{2}\mathtt{ulp(x)}$. That is nearly the converse of the first property, but is true for more bases.

❸ For any integer base β, if $|X - x| < \frac{1}{2}\mathtt{KahanUlp(x)}$ then $X = \mathtt{RN(x)}$.

❹ If $\beta = 2$ and $X = \mathtt{RN(x)}$ then $|X - x| \le \frac{1}{2}\mathtt{KahanUlp(x)}$. That is almost the converse of the third property, but holds only for binary arithmetic.

This set of choices provides a suitable counterexample:

$$X = 1, \qquad 1 + \tfrac{1}{2}\beta^{-t} < x < 1 + \beta^{-t}, \qquad |X - x| < \beta^{-t}, \qquad \tfrac{1}{2}\mathtt{KahanUlp(x)} = \tfrac{1}{2}\beta^{-t}.$$

A numeric counterexample looks like this:

$$X = 1, \qquad x = 1.007, \qquad |X - x| = 0.007, \qquad \tfrac{1}{2}\mathtt{KahanUlp(x)} = 0.005.$$

❺ If $|X - x| < \mathtt{ulp(x)}$ then X lies in the interval $[\mathtt{RD(x)}, \mathtt{RU(x)}]$.

The converse does *not* hold. The same counterexample as in the first property can be used here. We have $\mathtt{RD(x)} = 1$ and $\mathtt{RU(x)} = 1 + \beta^{1-t}$, but X is outside the interval $[1, 1 + \beta^{1-t}]$. Our three-digit decimal example has $X = 0.999$, outside the rounded interval $[1.00, 1.01]$.

There is no similar relation between the Kahan ulp and the round-down/round-up interval. The newer Muller–Harrison definition is of particular use in *interval arithmetic*, a topic that receives only brief mention later in this book (see **Section 5.8** on page 115 and **Appendix H.8** on page 966).

❻ Regrettably, when $\beta > 3$, then there is *no reasonable definition* of an $\mathtt{ulp(x)}$ function that makes these relations both true:

- if $|X - x| < \frac{1}{2}\mathtt{ulp(x)}$ then $X = \mathtt{RN(x)}$;
- if $X = \mathtt{RN(x)}$ then $|X - x| \le \frac{1}{2}\mathtt{ulp(x)}$.

The chief significance of those properties for the remainder of this book is this statement:

> *If we can show that all of a large number of measured errors of function values computed in binary arithmetic are below $\frac{1}{2}$ ulp using either the Kahan or the Muller–Harrison definition, then our function is, with high probability, correctly rounded.*

However, we need to use Kahan's definition of the ulp if we are to make the same conclusion for computation in octal, decimal, or hexadecimal arithmetic.

Implementation of the ulp functions requires considerable care, and they are sufficiently complicated that they deserve to be part of all mathematical software libraries. Regrettably, they are absent from standardized programming languages. Here is the generic code from the mathcw library for Kahan's definition:

```
#define EMAX  (FP_T_MAX_EXP - 1)  /* exponent of maximum normal */
#define EMIN  (FP_T_MIN_EXP - 1)  /* exponent of minimum normal */
#define ESUB  (EMIN - T + 1)      /* exponent of minimum subnormal */
#define T     (FP_T_MANT_DIG)     /* precision in base-B digits */

fp_t
ULPK(hp_t x)
{   /* Kahan ulp: return fp_t gap around or below ‘exact’ x */
    hp_t result;

    if (ISNAN((fp_t)x))
        result = x;
    else if (ISINF((fp_t)x))
        result = HP_SCALBN(ONE, EMAX - T + 1);
    else if (QABS(x) <= FP_T_MIN)
        result = HP_SCALBN(ONE, ESUB);
    else                      /* x is finite, nonzero, and normal */
    {
        hp_t s;               /* significand */
        int e;                /* unbiased base-B exponent */

        e = HP_ILOGB(x);
        s = HP_SCALBN(QABS(x), -e);
        e -= T - 1;

        if ( (s - HP(1.0)) < (hp_t)(HALF * FP_T_EPSILON) )
            e--;              /* special case at exponent boundary */

        if (e < ESUB)
            e = ESUB;

        result = HP_SCALBN(ONE, e);
    }

    if ((fp_t)result == ZERO) /* subnormals are flushed to zero */
        result = (hp_t)FP_T_MIN;

    return ((fp_t)result);
}
```

That code works without change for any floating-point base. It runs on modern IEEE 754 systems and many historical architectures, as long as the external functions are available, as they are with the mathcw library.

A separate function family, ULPMH(), provides the Muller–Harrison variant. Its code requires only a change in the function name, and the exponent-boundary test, which becomes

```
        if (s == ONE)
```

The data type fp_t corresponds to working precision, and the data type hp_t to the next higher precision. When no such precision is available, the two types are equivalent, and we are then forced to approximate a real value x by the closest machine number in working precision. In that case, the Kahan and Muller–Harrison ulp functions produce identical output.

The HP_ILOGB() and HP_SCALBN() macros expand to C99 functions to return the unbiased floating-point exponent, and to scale a number by a power of the base.

The private macro QABS() implements a quick inline absolute-value computation that is only safe after it is known that the argument is neither Infinity nor a NaN. Its definition in prec.h is simple:

```
#define QABS(x) ( (-(x) > (x)) ? -(x) : (x) )
```

The ternary expression avoids reference to a precision-dependent zero constant. It has the misfeature that its argument is evaluated three times, and that QABS(-0.0) evaluates to a negative zero, but those infelicities are harmless in our careful uses of the macro.

The final if statement tests for a zero result. That can happen only if subnormals are unsupported, or are flushed to zero. The handling of subnormals on some platforms can be selected at compile time, and on a few architectures, controlled at run time by library calls. That final test is mandatory, since we cannot determine portably at compile time whether subnormals are possible.

4.17 Fused multiply-add

During the 1980s, building on earlier work at CDC, Cray, and IBM, research projects at Stanford University and the University of California, Berkeley produced a promising new computer design model called the RISC (**R**educed **I**nstruction **S**et **C**omputer) architecture. The RISC idea was soon adopted by most of the major manufacturers, each with its own unique processor design:

- In 1986, IBM announced the RT workstation based on the ROMP chip, a descendant of the 801 CPU that was developed in the late 1970s, but never marketed. Also in 1986, Hewlett–Packard announced PA-RISC, and MIPS introduced the R2000.
- MIPS announced the R3000 in 1988.
- In 1989, Intel introduced the i860, Motorola announced the 88100, and Sun Microsystems shipped their first SPARC-based systems.
- IBM introduced its new RS/6000 product line based on the POWER architecture in 1990.
- DEC Alpha systems shipped in 1992.

RISC processors are characterized by large register sets (POWER has thirty-two 64-bit floating-point registers), multiple functional units, relatively simple instructions designed to run no more than a few clock cycles, and memory access only through load and store instructions.

POWER introduced a new instruction, the *floating-point fused multiply-add*, which computes $xy + z$ in about the same time as a single multiply or add. PA-RISC later added support for that instruction, and IA-64 does as well, making it a critical component of divide and square-root instruction sequences [Mar00]. In 1999, IBM added the G5 processor to the venerable System/360 mainframe family, providing for the first time both IEEE 754 arithmetic (including 128-bit arithmetic in hardware), and a fused multiply-add instruction, on their mainframes.

Some of those processors do not really even have separate instructions for multiply or add: they use hardwired constants with the fused multiply-add instruction to compute $1 \times x + y$ for addition, and $xy + 0$ for multiplication. Most systems have variants to handle $xy - z$, $-xy + z$, and $-xy - z$, so as to avoid the need for separate negation instructions.

Some embedded graphics processors provide a multiply-add instruction, although it may be restricted to integer or fixed-point operands. Most mainstream RISC processors have later been extended with video instruction sets, and some of those extensions include a limited form of multiply-add.

Apart from roughly doubling floating-point performance, what is significant about the fused multiply-add instruction is that it computes an *exact* double-length product xy and then adds z to that result, producing a final normal-length result with only a *single* rounding.

Here are three uses of the fused multiply-add instruction that are relevant for this book:

- Compute an exact double-length product with this simple code:

  ```
  hi = x * y;
  lo = fma(x, y, -hi);    /* hi + lo = x * y exactly */
  ```

  The term `lo` is the recovered error in multiplication.

- Recover the approximate error in division and square root like this:

  $$\begin{aligned} q &= \mathrm{fl}(x/y), & s &= \mathrm{fl}(\sqrt{x}), \\ x/y &= q + e, & x &= (s+e)^2, \\ e &= x/y - q, & &= s^2 + 2se + e^2, \\ &= (-qy + x)/y, & e &\approx (-s^2 + x)/(2s), \\ &= \mathrm{fma}(-q, y, x)/y, & &\approx \mathrm{fma}(-s, s, x)/(s+s). \end{aligned}$$

  The error terms e are accurate, but *not* exact, because the mathematical operations can produce results with an unbounded number of digits.

- Reconstruct elementary functions from rational polynomial approximations, by replacing

  ```
  result = c + x * R;     /* R = P(x) / Q(x) */
  ```

  with

```
result = fma(x, R, c);
```

The fused multiply-add operation avoids cancellation errors when the terms of the sum have opposite signs, and makes a correctly rounded result more likely.

Fused multiply-add instructions have been found to have an amazing variety of important applications: see Markstein's book [Mar00] for an extensive discussion of their great utility on IA-64, and Nievergelt's article [Nie03] for how the fused multiply-add instruction can be exploited to produce matrix arithmetic provably correct to the next-to-last bit. Consequently, the 1999 ISO C Standard adds new math-library functions `fmaf(x,y,z)`, `fma(x,y,z)`, and `fmal(x,y,z)` to provide programmer access to it. Compilers on architectures that support it in hardware may generate it by default, or may require a certain optimization level to do so, or alternatively, may need a command-line option to prevent its use. On IA-64, GNU `gcc` seems reticent about using fused multiply-add instructions, whereas Intel `icc` employs them at every opportunity, even with debug-level compilation.

When hardware support is not available, there seems to be no simple way to implement a correct fused multiply-add without resorting to multiple-precision arithmetic. Although you can readily split x and y into upper and lower halves and compute an exact double-length product, it is quite hard to then do the addition of z with only a *single* rounding. Thus, library implementations of the `fma()` family of functions may be slow. We show the mathcw library code for the `fma()` functions in **Section 13.26** on page 388.

Regrettably, tests show that virtually all GNU/Linux versions of the `fma()` routines are *incorrect*: they compute the result with the C expression `x*y + z`, and that is wrong unless the compiler happens to generate a hardware multiply-add instruction. Whether it does or not depends on optimization flags chosen when the library was built. POWER systems only have 64-bit fused multiply-add, although PowerPC later added the 32-bit companion. Thus, even if `fma()` is correct on IBM AIX systems, `fmaf()` in the same executable may be correct on one machine, and wrong on another, and `fmal()` is just plain wrong. The sad lesson is that the good intents of the ISO C Committee have been thwarted by faulty implementations of those important library functions.

You can easily see the impact of the exact product by computing $xy + z$ for $x = y = 1 + \delta$ and $z = -(1 + 2\delta)$. For sufficiently small δ, the (incorrect) result is zero when computed by an ordinary multiply followed by an add, whereas with the fused multiply-add, the answer is the mathematically correct δ^2.

Here is a demonstration with `hoc`, which implements `fma()` correctly on all Unix platforms, using hardware when available, and otherwise, using a multiple-precision software package:

```
hoc> epsilon = macheps(1)
hoc> hexfp(epsilon)
        +0x1.0p-52
hoc> delta = sqrt(epsilon)/2
hoc> hexfp(delta)
        +0x1.0p-27
hoc> x = y = 1 + delta
hoc> z = -(1 + 2*delta)
hoc> x * y + z
        0
hoc> fma(x,y,z)
        5.5511151231257827e-17
hoc> hexfp(fma(x,y,z))
        +0x1.0p-54
hoc> fma(x,y,z) / delta**2
        1
```

The relevance of the fused multiply-add instruction to elementary-function computation, and indeed, most numerical computation, is that the combination of a multiply followed by an add is common. For example, from the body of the square-root function in `sqrtx.h`, the critical steps are

```
y = FP(0.41731) + FP(0.59016) * f;
```

followed by repeated application of these two statements:

```
z = y + f / y;
y = FP(0.25) * z + f / z;
```

At the cost of one more multiply, and slightly altered rounding from taking the reciprocal, they could be written as:

```
y = FMA(FP(0.59016), f, FP(0.41731));
z = FMA(f, FP(1.0) / y, y);
y = FMA(FP(0.25), z, f / z);
```

4.18 Fused multiply-add and polynomials

Rational polynomial approximations are at the core of algorithms for computing most of the elementary functions, and the polynomial evaluations are full of multiply-add opportunities. Consider the $\langle 7/7 \rangle$-degree rational approximation for 116-bit precision in `atanx.h`:

```
pg_g = (((((((p[7] * g + p[6]) * g + p[5]) * g + p[4]) * g +
           p[3]) * g + p[2]) * g + p[1]) * g + p[0]) * g;
qg   = ((((((g[7] * g + q[6]) * g + q[5]) * g + q[4]) * g +
           q[3]) * g +q[2]) * g + q[1]) * g + q[0];
```

With multiply-add wrappers, those statements turn into fourteen nested multiply-adds with only sixteen memory loads and two stores, producing a 2 : 1 ratio of computation to memory access:

```
pg_g = FMA(FMA(FMA(FMA(FMA(FMA(FMA(p[7], g, p[6]), g, p[5]),
        g, p[4]), g, p[3]), g, p[2]), g, p[1]), g, p[0]) * g;
qg   = FMA(FMA(FMA(FMA(FMA(FMA(FMA(g[7], g, q[6]), g, q[5]),
        g, q[4]), g, q[3]), g, q[2]), g, q[1]), g, q[0]);
```

Unlike the cases discussed earlier where additional precision must be prevented, here the extra precision of the fused multiply-add can be helpful.

To avoid having order-dependent inline polynomial evaluation in the program source code, the polynomials could be evaluated with code loops like this:

```
for (pg = p[np], k = np - 1; k >= 0; --k)
    pg = pg * g + p[k];

pg_g = pg * g;

for (qg = q[nq], k = nq - 1; k >= 0; --k)
    qg = qg * g + q[k];
```

Loop code with multiply-add wrappers looks like this:

```
for (pg = p[np], k = np - 1; k >= 0; --k)
    pg = FMA(pg, g, p[k]);

pg_g = pg * g;

for (qg = q[nq], k = nq - 1; k >= 0; --k)
    qg = FMA(qg, g, q[k]);
```

Because the loop bounds are compile-time constants, a good optimizing compiler might unroll them to the equivalent of our explicit inline evaluation, but not all compilers are capable of that optimization.

We could even use an external polynomial-evaluation routine like this:

```
pg_g = POLY(p, np, g) * g;
qg   = POLY(q, nq, g);
```

However, loops and function calls entail additional overhead that is avoided on all platforms by inline evaluation, and that is important because the speed of the elementary functions is sometimes the limiting factor in inner loops of numerically intensive programs.

The practice adopted in the mathcw library is to code the polynomial evaluations as macros that can be defined to expand to inline code, or to calls to multiply-add macros. Those macros in turn can be either expanded inline, or to calls to the C99-style `fma(x,y,z)` functions, which smart compilers can turn into single fused multiply-add instructions on systems that have them. Thus, for the $\langle 7/7 \rangle$-degree rational approximation in `atanx.h`, the actual code reads

```
pg_g = POLY_P(p, g) * g;
qg   = POLY_Q(q, g);
```

The two macros hide the polynomial degrees, which are hidden in their definitions in `atan.h`. That code in turn is macro-expanded to efficient inline code. That is shorter and clearer than the expanded form shown earlier, and for hand coding, is more likely to be correct. To eliminate human error, the data and code for all of the new polynomial evaluations used in the mathcw library are generated by a Maple program, and then copied into the source code verbatim.

Perhaps surprisingly, it is possible to rearrange and refactor polynomials of degree five or higher to reduce operation counts. Knuth [Knu97, pages 490ff] shows that a fourth-order polynomial can be evaluated in three multiplies and five adds, instead of four of each, and a sixth-order polynomial can be handled with four multiplies and seven adds, instead of six of each. For general order-n polynomials, $\lfloor n/2 \rfloor + 2$ (see **Section 6.7** on page 136) multiplies and n adds suffice. However, the rearrangement may change the numerical stability, because the new coefficients may have mixed signs when the original ones all had like signs. In addition, the new coefficients depend on the roots of the polynomial, and it may be difficult to find those roots with sufficient accuracy unless higher-precision computation is available.

We have therefore chosen to retain the simple Horner form in the mathcw library, wrapped inside invocations of the `POLY_n()` macros.

The reliability of Horner's rule computation has been the subject of recent research [HPW90, Pri92, BD03b, BD04, GLL05, SW05, GLL06, BH07, LL07, RBJ16]. Inaccuracies arise mainly near zeros of the polynomial, and our use of minimax polynomial fits in the mathcw library avoids those regions.

4.19 Significance loss

When two similar numbers of the same sign are subtracted, many leading digits can be lost, and the effective precision of the result may be sharply reduced. In the worst case, that lowered precision can contaminate subsequent computations so badly that they become inaccurate, or even worthless.

Just when does significance loss occur? In binary arithmetic, it happens whenever the difference of two numbers loses one or more leading bits in the subtraction, and that can happen only if the ratio of the absolute value of the difference to the larger of the absolute values of the two numbers is less than or equal to $1/2$. That is, we get subtraction loss in $x - y$ if $|x - y| / \max(|x|, |y|) \leq 1/2$. More generally, for arbitrary base β, there is definitely subtraction loss if $|x - y| / \max(|x|, |y|) \leq 1/\beta$. However, when $\beta > 2$, subtraction loss is also possible for larger ratios: the example in decimal arithmetic of $x = 0.1999\ldots$ and $y = 0.1000\ldots$ shows that ratios up to $1/2$ can sometimes result in subtraction loss.

Significance loss from subtractions is unfortunately common, often unnoticed, and generally a much larger source of computational error in numerical programs than is cumulative rounding error. Programmers always need to be alert for the possibility of significance loss, and take steps to reduce it, or prevent it entirely.

Sometimes significance loss can only be avoided by computing in higher precision, and that is an important reason why such precision needs to be easily, and cheaply, available; sadly, that is rarely the case in most programming languages.

In other cases, however, including several that we treat in this book, it is possible to rearrange the computation so that part of the subtraction can be done analytically, and thus, exactly, so that what is finally computed can then be much more accurate.

4.20 Error handling and reporting

Library designers must decide how to handle errors, whether they arise from erroneous user-provided arguments, or because the computation cannot produce a valid result.

Some languages, such as C++, C#, Java, Lisp, and some symbolic-algebra systems, provide support for `try`, `catch`, and `throw` statements. For example, a fragment of a C# program for investigating the limits and behavior of integer arithmetic looks like this:

```
int k, n;

n = 0;
k = 1;

try
{
    while (k > 0)
    {
        Console.WriteLine("2**(" + n + ") = " +  k);
        n++;
        checked { k *= 2; }
    }
}
catch (System.OverflowException)
{
    Console.WriteLine("OverflowException caught: k = " + k);
}
```

The `try` block of the code attempts repeated doubling of k, reporting powers of two at each iteration. As we discuss in **Section 4.10** on page 72, generation of integer results that are too big to represent is normally not caught, but that language has a `checked` statement that enforces detection of numeric errors. When the result can no longer be represented, the compiler-generated code throws an exception that is then caught by the first `catch` block in the call history with a matching exception type. Here, that block immediately follows the `try` block, and the code reports the exception with a message

```
OverflowException caught: k = 1073741824
```

and execution then continues normally. From the output, we can conclude that the largest representable signed integer is less than twice the last value of k; in fact, it is exactly $2k - 1 = 2\,147\,483\,647 = 2^{31} - 1$. That corresponds to a 32-bit `int` type, which C# mandates.

In such languages, there is a default `catch` block in the startup code that calls the user's `main()` program. That block fields any uncaught exception, reports it, and terminates the job. For example, if the `try` statement and its companion `catch` block are commented out and the program is compiled and run, the output looks like this:

```
2**(0) = 1
2**(1) = 2
...
2**(29) = 536870912
2**(30) = 1073741824

Unhandled Exception: System.OverflowException: Number overflow.
in <0x003f7> IntOfl:Main ()
```

The motivation for that style of error handling is that it guarantees that uncaught errors are always reported, even though that means terminating the job prematurely, and risking other possibly severe problems, such as an incomplete database transaction.

The C language, however, lacks the `try/throw/catch` style of error handling. Instead, it provides two similar mechanisms, the *signal* and *long jump* facilities.

The first of those is supported by the standard header file, `<signal.h>`, and the `raise()` and `signal()` functions. The operating system, or the run-time library, maintains a process-specific block of signal handlers for each defined exception type, and when a named exception is signaled by a call to `raise()`, the last-registered handler for that particular signal name is called. Handlers may be user defined, or provided by default, but it is usually not feasible for them to do much more than report the error and terminate the job. Because it is possible on some systems for signals to be lost, they do not provide a reliable communication mechanism without further programming. Although the signal feature appears simple, in practice, it is a complex topic that needs a large book chapter to treat properly [SR05, Chapter 10].

The second is supplied by `<setjmp.h>`, and the functions `longjmp()` and `setjmp()`. The `setjmp()` function saves the current calling environment in a user-provided argument, and a later call to `longjmp()` with that argument returns control to the statement following the `setjmp()`, bypassing all of the intervening functions in the call history. Interactive programming-language interpreters, and text editors, normally use that facility to catch error conditions, report them, and continue execution at top level.

All three of those styles of exception handling require considerable programming care to minimize data corruption. For example, an exception that is thrown between the allocation and freeing of a dynamic memory block results in a *memory leak*: a memory block is left allocated, but can no longer be used. If the memory allocator supports run-time *garbage collection*, then it may be possible to recover the lost block sometime later. Otherwise, as more such exceptions are thrown and caught, the memory requirements of the running process continue to grow, and eventually, it may run out of resources. That problem is commonly seen in long-running programs, such as desktop window managers, document viewers, operating-system service daemons, text editors, and Web browsers, and is evidently difficult to solve in complex software systems.

Older Fortran language standards provide nothing comparable to those error-handling facilities, so functions and subroutines in libraries for that language often include additional arguments that, on return, are set to status values that record success or failure.

The designers of the C run-time library take a simpler approach that avoids both run-time exception handling for error reporting, and additional error-flag arguments. Instead, exceptional conditions are reported via the function return value (such as the largest representable value, when the true result is bigger than that), and possibly in a global error value, `errno`.

The name `errno` may be either a variable of type `int`, or a dereferenced pointer from a function that returns a pointer to an `int`. For that reason, it should *never* be declared by user code, but instead be expected to be supplied by `<errno.h>`. Before C89, some systems supplied a definition of `errno` in that file, whereas others did not, so it was common for code to include a declaration `extern int errno;`, but that fatally conflicts with the declaration on a system that uses a dereferenced pointer. Code that is expected to run on older systems may have to deal with that aberration, and for that reason, it is a good idea in portable code never to include `<errno.h>` directly, but instead provide access to it via a private wrapper header file that can clean up any system-dependent mess.

The header file `<errno.h>` also defines dozens of error-code macros to represent library errors. Each begins with the letter `E` and has a distinct, positive, nonzero, and small, integer value. The standard way for library functions to record an error condition is to set `errno` to one of those values.

Standard C mandates support for only three error codes:

`EDOM` domain error: an argument is outside the range for which the mathematical function is defined;

`EILSEQ` illegal sequence [C99 only]: an invalid or incomplete byte sequence is detected in a character string with multibyte or wide-character encoding;

`ERANGE` range error: the mathematical function value is not representable in the host arithmetic.

All of the other error symbols defined in `<errno.h>` are available to user code, and to code in other libraries that interface to the filesystem, networks, operating system, window system, and so on, but they are *never* used by the Standard C library. Their use may be an impediment to portability, but because they are guaranteed to be macros, their code can be guarded with suitable preprocessor conditionals.

User programs can set `errno` to zero before calling a library function, and then test it for a nonzero value on return from that function. Library code *never* sets `errno` to zero, and never references it unless errors are detected. Thus, `errno` is a sticky flag that records only its most-recently assigned value.

The C Standards require that `errno` be set to `ERANGE` if the exact mathematical result is too big to represent. The function then should return a value of the largest representable magnitude, with the same sign as the correct result. However, if the value is too small to represent, then a zero value is returned, but the implementation may optionally set `errno` to `ERANGE`. Programs therefore cannot rely on `errno` being set when a zero is returned.

The largest representable floating-point value is problematic. C89 recommends use of the macro `HUGE_VAL`, which is defined in `<math.h>`. It is a positive compile-time constant expression whose value is of type `double`. Older systems may not supply such a value, or may give it a different name, such as `HUGE`. However, many vendors added support for `float` and `long double` data types and library functions before they were officially recognized by C99, and because the ranges of those types generally differ from that of `double`, a single definition of the largest representable value is no longer adequate. C99 therefore adds `HUGE_VALF` and `HUGE_VALL` for the two extra limits. It allows the three values to be positive infinities if the arithmetic supports such values, but does not require their use: an implementor may set them to the largest finite values instead. A test program to display those values looks like this:

```
% cat huge.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int
main(void)
{
    (void)printf("HUGE_VAL  = %g\n", HUGE_VAL);

#if defined(HUGE_VALF)
    (void)printf("HUGE_VALF = %g\n", (double)HUGE_VAL);
#endif

#if defined(HUGE_VALL)
    (void)printf("HUGE_VALL = %Lg\n", HUGE_VALL);
#endif

    return (EXIT_SUCCESS);
}

% cc huge.c && ./a.out
HUGE_VAL  = Inf
HUGE_VALF = Inf
HUGE_VALL = Inf
```

When that program is run on the systems in this author's test laboratory, it reports infinities on all but one (MINIX on IA-32), where it reports the largest finite number. Some systems require a C99 compilation environment to expose the `float` and `long double` constants, some do not have them at all, and a few others have them in the normal C environment.

When a nondefault rounding mode, such as *round-to-zero*, is in effect, then an overflowed, but mathematically finite, result should be returned as the largest finite number, rather than Infinity. The use of the `HUGE_VAL` constants is then clearly wrong, but Standard C does not cater to that possibility. That point is relevant for functions, like $\exp(x)$, that have cutoffs outside of which the function overflows or underflows. To conform to the current rounding mode, and properly set the sticky floating-point exception flags, for finite arguments outside the interval defined by the cutoffs, the computed value should be a run-time computation of an explicit overflow or underflow, such as with code like this:

```
else if (x < UFL_CUTOFF)
{
    volatile fp_t tiny;

    tiny = FP_T_MIN;
    STORE(&tiny);
    result = SET_ERANGE(tiny * tiny);
}
else if (OFL_CUTOFF < x)
{

#if defined(HAVE_IEEE_754)
    volatile fp_t big;
    big = FP_T_MAX;
    STORE(&big);
    result = SET_ERANGE(big * big);
#else
    result = SET_ERANGE(FP_T_MAX);
#endif

}
```

To prevent premature underflow to zero, the algorithm should be designed to permit subnormal function results to be generated. Whether subnormals are supported or not may depend on the CPU architecture and model, the operating system, compilation options, and even on run-time library settings (see **Section 4.12** on page 78), so it is not in general possible to decide between subnormals and abrupt underflow when the library is built. The limit `UFL_CUTOFF` should be set to the x value for which $\exp(x)$ is below half the smallest subnormal number, thereby accounting for default rounding. That means that the subnormal region is handled together with the normal region later in the code, and that code block should then check for a result that is smaller than `FP_T_MIN`, and if so, and the library design goal is to report underflows via `errno`, record a domain error.

4.21 Interpreting error codes

The values of the error macros are system dependent, but the call to the C89 function `strerror(`*number*`)` returns a pointer to a short message string that describes the error. That function is declared in `<string.h>`. Here is how it can be used:

```
% cat strerr.c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(void)
{
    (void)printf("%-6s -> %s\n", "EDOM",   strerror(EDOM));
    (void)printf("%-6s -> %s\n", "EILSEQ", strerror(EILSEQ));
    (void)printf("%-6s -> %s\n", "ERANGE", strerror(ERANGE));

    return (EXIT_SUCCESS);
}

% cc strerr.c && ./a.out
EDOM   -> Argument out of domain
EILSEQ -> Illegal byte sequence
ERANGE -> Result too large
```

Although most systems report for `ERANGE` something like *`Result too large`*, on GNU/Linux systems, we find *`Numerical result out of range`*, suggesting that both underflow and overflow limits may be considered when `errno` is set.

Alternatively, an older mechanism from historical C, and still supported by C89 and C99, can be used. A call to `perror("msg")` outputs its argument string to the standard error unit, followed by a colon, a space, a short description of the error currently recorded in `errno`, and a newline. The `perror()` function is declared in `<stdio.h>`. Here is an example:

```
% cat perror.c
#include <errno.h>
#include <limits.h>
#include <math.h>
```

```
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    errno = -1;      perror("errno == -1 means");
    errno = 0;       perror("errno == 0 means");
    errno = EDOM;    perror("errno == EDOM means");
    errno = INT_MAX; perror("errno == INT_MAX means");

    return (EXIT_SUCCESS);
}

% cc perror.c && ./a.out
errno == -1 means: Unknown error
errno == 0 means: Error 0
errno == EDOM means: Argument out of domain
errno == INT_MAX means: Unknown error
```

Explicit access to the error-description strings is provided by some systems via a global error-number index limit, and a companion array of strings, that can be declared like this:

```
extern int sys_nerr;
extern char * sys_errlist[];
```

However, the programmer can never be sure whether a particular system provides those definitions in `<errno.h>`, or in `<stdio.h>`, or in some other header file, or perhaps not at all. Some systems supply a `const` qualifier on the string table declaration, which then conflicts with the conventional declaration. The `sys_nerr` and `sys_errlist` features should therefore be *scrupulously avoided* in favor of `strerror()`. If an older system lacks that function, just supply a private definition that uses that system's version of `sys_nerr` and `sys_errlist`, or provides its own system-specific definitions.

4.22 C99 changes to error reporting

C99 introduces a new feature in `<math.h>` that is not yet widely supported at the time of writing this: the macros `math_errhandling`, `MATH_ERRNO`, and `MATH_ERREXCEPT`. The first may expand to an integer constant, or to a global `int` variable, or to a dereferenced pointer from a function. The value of `math_errhandling` is either of the second or third macros, or their bitwise-OR. When a domain error is detected, the function should then set `errno` to `EDOM` only if `math_errhandling & MATH_ERRNO` is nonzero. If `math_errhandling & MATH_ERREXCEPT` is nonzero, then the *invalid* floating-point exception should be raised. Here is a test program to investigate current practice:

```
% cat mtherr.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int
main(void)
{

#if defined(math_errhandling) && \
    defined(MATH_ERREXCEPT)   && \
    defined(MATH_ERRNO)

    (void)printf("C99 math_errhandling = %d\n", math_errhandling);
    (void)printf("C99 MATH_ERRNO       = %d\n", MATH_ERRNO);
    (void)printf("C99 MATH_ERREXCEPT   = %d\n", MATH_ERREXCEPT);

#else

    (void)printf("No support for C99 math_errhandling facility\n");

#endif

    return (EXIT_SUCCESS);
}

# On most systems:
% cc mtherr.c && ./a.out
No support for C99 math_errhandling facility

# On a few systems:
% cc mtherr.c && ./a.out
C99 math_errhandling = 2
C99 MATH_ERRNO       = 1
C99 MATH_ERREXCEPT   = 2
```

The systems that report such support produce identical output, with the surprising result that errno would apparently not be set for domain errors, deviating from decades-long practice!

Because the mathcw library conceals all access to errno in macros, we can easily support the C99 extension with an alternate definition:

```
#define SET_EDOM(val) ((math_errhandling & MATH_ERRNO) && errno = EDOM, val)
```

We have not implemented that change, because it would prevent the setting of errno on some systems, and because, in this author's view, the designers of those systems have made an incorrect, and undesirable, implementation choice that is likely to cause knowledgeable users to complain. Indeed, that new feature of C99 seems to have questionable value for a language as mature and widely used as C is.

4.23 Error reporting with threads

Global variables are not reliable in the presence of multiple execution threads, a topic that we discuss later in Section 5.3 on page 106. For example, one thread might clear errno immediately before calling a library function, and then test errno on return. Even if that function does not set errno, it is possible that a function called in another thread does set it, with the result that the first thread might wrongly change its execution behavior. Access to a global variable in a multithreaded environment requires a locking mechanism to permit only one thread at a time to access the variable, and there is no provision in Standard C for creating such a lock.

Tests on numerous modern systems at the time of writing this show that errno is thread-safe on most, because it expands to a call to a thread-local function, but some compilers may require special options to ensure that choice.

To improve throughput and reduce chip heating, some time after the year 2000, vendors began to design and manufacture CPUs that are split into multiple subunits, called *cores*. Each core may itself be capable of running multiple threads, so multiple CPUs, multiple cores, and multiple threads are now a reality, even on small personal computers. When threading is enabled, getting reliable behavior in languages that have freely accessible global variables is likely to remain a challenging problem, and C programmers should therefore be cautious about relying on the value of errno.

4.24 Comments on error reporting

The ambiguity of the meaning of HUGE_VAL and its companions is a nuisance for both library designers, and for users. In addition, the C Standards do not entirely address the behavior of IEEE 754 arithmetic, in that return of the constant

HUGE_VAL does *not* set exception flags, whereas computation that actually produced such a value *would* set them. The handling of NaN arguments and NaN results is not clearly specified either.

The uniform approach adopted in the mathcw library is normally to check for Infinity, NaN, and out-of-range and special arguments first, with code like this fragment from the square-root function in sqrtx.h:

```
if (ISNAN(x))
    result = SET_EDOM(x);
else if (x < ZERO)
    result = SET_EDOM(QNAN(""));
else if (ISINF(x))
    result = SET_ERANGE(x);
else if (x == ZERO)
    result = x;         /* preserve sign of zero, so sqrt(-0) = -0 */
```

The SET_Exxx() macros conceal errno, and are defined in a private header file, seterr.h, with C comma expressions like this:

```
#define SET_EDOM(val)           (errno = EDOM,   val)
#define SET_ERANGE(val)         (errno = ERANGE, val)
```

There are several effects of that design choice:

- For an argument NaN, errno is set to EDOM, and the NaN is returned unchanged. That preserves its quiet/signaling status, and its significand payload, both of which may be of use to the caller. It does *not* set sticky floating-point exception flags.

  That is analogous to the handling of a *quiet* NaNs IEEE 754 arithmetic operations, as described in **Section 4.14** on page 79, but differs from the behavior of those operations with a *signaling* NaN.

- A negative argument is outside the domain of the square-root function in real arithmetic, so the code sets errno to EDOM and returns a quiet NaN that is generated at *run time* by our extended function family provided by the QNAN() wrapper macro. The NaN has a unique payload that changes with each call to QNAN(), although it is possible to run out of payload bits, and wrap around to the initial payload. Thus, our implementation always sets sticky floating-point exception flags.

- For a positive infinite argument, the code sets errno to ERANGE, and returns that argument, *without* setting sticky exception flags. That too follows the behavior of IEEE 754 arithmetic operations.

- For that particular function, because IEEE 754 requires that $\sqrt{-0}$ evaluate to -0, the code returns a zero argument unchanged, thereby preserving its sign.

4.25 Testing function implementations

The ELEFUNT package uses two basic techniques for generating test function values: rapidly convergent Taylor series for small arguments, and function addition relations. We can illustrate that for the function $\operatorname{atanh}(x)$, sketched later in **Figure 12.6** on page 350, and defined only when the argument x is restricted to the range $[-1, +1]$. That function has a simple Taylor series

$$\begin{aligned}\operatorname{atanh}(x) &= x + (1/3)x^3 + (1/5)x^5 + \cdots + (1/(2n+1))x^{2n+1} + \cdots \\ &= x(((((\cdots + (1/9))x^2 + (1/7))x^2 + (1/5))x^2 + (1/3))x^2 + 1),\end{aligned}$$

and this addition formula:

$$\operatorname{atanh}(x) \pm \operatorname{atanh}(y) = \operatorname{atanh}\left(\frac{x \pm y}{1 \pm xy}\right).$$

If those equations are used for comparison with results generated by an implementation of atanh(), then it is essential to minimize errors in their computation. Errors arise in argument generation, term summation, and function evaluation.

4.25.1 Taylor-series evaluation

We see from the nested Horner form of the Taylor series for $\operatorname{atanh}(x)$ that reciprocals of odd numbers and multiplication by x^2 are the key components.

For small $|x|$, the Taylor series converges rapidly and the Horner form sums the terms in the desirable order of smallest to largest. Because $x^2 \geq 0$ and all coefficients are positive, there is no possibility of subtraction loss in the summation.

The accuracy of the Taylor series can be improved in two ways: limiting the number of terms required by restricting the range of x, and reducing, or eliminating, the error in each term.

For example, if we truncate the inner series after $(1/9)x^8$, then because the series begins with one, the first term omitted, $(1/11)x^{10}$, must be smaller than the rounding error. That error is half the machine epsilon, $\epsilon = \beta^{-(t-1)}$, for base β and a t-digit significand. We then pick a simple form of the cutoff, $x = \beta^{-n}$, where n is to be determined, so that we have

$$\begin{aligned}
(1/11)x^{10} &= (1/11)\beta^{-10n} \\
&< \beta^{-(t-1)}/2, \\
\beta^{-10n} &< (11/2)\beta^{-(t-1)}, \\
\beta^{-10n+t-1} &< 11/2, \\
(-10n+t-1)\log(\beta) &< \log(11/2).
\end{aligned}$$

We can then easily solve for n:

$$n > (t-1-\log(11/2)/\log(\beta))/10.$$

For extended IEEE 754 binary arithmetic, we can use $n = 3$, 5, 7, 11, and 24, respectively, for the five significand precisions ($t = 24$, 53, 64, 113, and 237). For decimal arithmetic, we find $n = 1$, 2, 4, and 7 for the four significand precisions ($t = 7$, 16, 34, and 70).

Thus, in IEEE 754 32-bit binary arithmetic, we can use the five-term truncated Taylor series for test arguments x in $[-2^{-3}, +2^{-3}] = [-0.125, +0.125]$, and in single-precision decimal arithmetic, for test arguments x in $[-10^{-1}, +10^{-1}] = [-0.1, +0.1]$.

We can remove the rounding errors from the inexact coefficient divisions by factoring out the product $3 \times 5 \times 7 \times 9 = 945$, rewriting the truncated Taylor series with exactly representable coefficients as:

$$\begin{aligned}
\operatorname{atanh}(x) &\approx (1/945)(945 + 315x^2 + 189x^4 + 135x^6 + 105x^8)x \\
&\approx x + (x/945)(315x^2 + 189x^4 + 135x^6 + 105x^8).
\end{aligned}$$

The second form avoids contaminating the first term with three rounding errors, and the remaining terms then provide a small correction to an exact result.

Because the number of bits required grows with the powers of x, each term of the truncated Taylor series is exact if we can choose test arguments x to make the last term, $105x^9$, exact. Unfortunately, that is not achievable: the constant 105 needs seven bits, and x^9 needs nine times as many bits as there are in x. With $t = 24$, that only permits one bit in x, and even with $t = 113$, we can have only 11 bits in x. Because we want to be able to test with a large number of different randomly chosen x values, we cannot accept such a restriction. Instead, we can ask that the first two terms, which are the larger ones in the series, be exact: the coefficient 315 needs nine bits, leaving us seven bits for x when $t = 24$, and 22 or more bits for x when $t \geq 53$. Seven bits gives only $2^7 = 128$ different x values, so for single precision, at least, that is too severe a restriction.

One reasonable solution is to compute the Taylor series in at least double precision, because that is almost always available on desktop and larger computers. Otherwise, we have to give up on the refinement of limiting the precision of test arguments in the Taylor series in single-precision computation.

4.25.2 Argument purification

The considerations of the last section lead to the question: how do we *purify* a random test argument x to reduce the number of bits in its significand?

One obvious way would be to use bit-masking operations to clear low-order bits in the significand, but that has a serious portability problem, because it requires knowledge of the machine-dependent storage representation of floating-point numbers.

Fortunately, there is a much better way that is easier, faster, and *completely portable*: simply add and subtract a suitable large number. For example, to clear the bottom two digits of x in $[0, 1)$ in a six-digit decimal system, we have $x = 0.\text{dddddd}$, so $10 + x = 10.\text{dddd}$ has lost the bottom two digits, and $(10 + x) - 10 = 0.\text{dddd}$ is the desired result. That technique of argument purification is used extensively in ELEFUNT.

Exact preservation of all but the low-order bits requires that the subtraction and addition be done in *round-to-zero* (truncating) arithmetic. However, for argument purification, we can use any rounding mode, because the goal is merely to obtain a result with a reduced number of significant bits.

4.25.3 Addition-rule evaluation

In the addition formula for atanh(x), we have two choices of sign:

$$\operatorname{atanh}(x) = \begin{cases} -\operatorname{atanh}(y) + \operatorname{atanh}\left(\dfrac{x+y}{1+xy}\right), \\ +\operatorname{atanh}(y) + \operatorname{atanh}\left(\dfrac{x-y}{1-xy}\right). \end{cases}$$

When $x \geq 0$, atanh$(x) > 0$, so if we pick $y > 0$ in the first of those, there could be subtraction loss in the sum of the function values. If we use the second, then both functions are positive, but there can be subtraction loss in forming the second argument. We also have the problem of argument purification: for any particular choice of y, we want both $x - y$ and $1 - xy$ to be computed exactly. For a fixed y, the value atanh(y) is a constant that we can precompute, and for improved accuracy, represent as a sum of an exact term and an approximate correction. In addition, we require y to be exactly representable.

Because of the division in the function argument, we cannot guarantee that it is exact, but at least the other possible sources of error in the argument have been eliminated. Addition rules for most of the other elementary functions are simpler than that for atanh(x), and for all of the functions treated by Cody and Waite, they are able to guarantee exact arguments in the ELEFUNT tests.

For the moment, assume the common case of binary arithmetic, and let $y = 2^{-n} = 1/N$, where the integer n satisfies $n > 0$. Then the second form of the addition rule says:

$$\begin{aligned} \operatorname{atanh}(x) &= \operatorname{atanh}(1/N) + \operatorname{atanh}\left(\frac{x - 1/N}{1 - x/N}\right) \\ &= \operatorname{atanh}(1/N) + \operatorname{atanh}\left(\frac{Nx - 1}{N - x}\right). \end{aligned}$$

Because x and y are limited to the range $[-1, +1]$, and because $N \geq 2$, the denominator $N - x$ cannot suffer subtraction loss, because

$$\begin{aligned} |N - y| / \max(|N|, |x|) &= |N - y|/N \\ &= |1 - y/N| \\ &> 1 - 1/N \\ &> 1/2, \end{aligned}$$

where the last result holds only if we also ensure that $N > 2$.

The numerator is free of subtraction loss only if x is negative, or if $(|x| - 1/N)/\max(|x|, 1/N) > 1/2$. We have already seen that the truncated Taylor series is optimal for small x, so we only need the addition rule for larger values. In particular, if we choose $1/N$ at the Taylor-series cutoff, then we only need $|x|$ values in $[1/N, 1]$, and there is then no subtraction loss if $(|x| - 1/N)/|x| > (1/2)$, which we can rewrite as the condition $|x| > 2/N$. Thus, to cover the complete range of x, we should revise our choice of $1/N$ to *half* the Taylor-series cutoff. Any larger value of N is also acceptable, because that just extends the addition-rule range to overlap into the Taylor-series range.

For IEEE 754 binary arithmetic, using half the Taylor-series cutoffs, we have $n = 3$, 6, 7, and 12. For decimal arithmetic, we need $|x| > 10/N$, or $1/N$ values that are a tenth of the Taylor-series cutoffs: $n = 2$, 3, and 5. In particular, for single-precision binary arithmetic, $n = 3$ means $N = 2^n = 8$. For the other three precisions, we have $N = 64$, 128, and 4096. Similarly, for the three precisions of decimal arithmetic, we have $N = 100$, 1000, and 100000. Argument purification is then achieved by adding and subtracting N from x.

We would prefer for the test program to be portable, and be usable for all practical precisions, so that only two precomputed values of $\operatorname{atanh}(y)$ are needed: one for decimal arithmetic, and one for binary arithmetic. That means that we should use the largest N values, with stored values of $\operatorname{atanh}(1/4096)$ and $\operatorname{atanh}(1/100000)$. Unfortunately, that is a problem for single-precision decimal, where $t = 7$: we require six digits for N, leaving only one free digit after argument purification. The requirements of a large N to shorten the Taylor series and avoid subtraction loss in the addition rule conflict with the need to have many possible random arguments. Consequently, the ELEFUNT-style test code in `tatanh.c` has three separate choices of N for each of binary and decimal.

4.25.4 Relative error computation

Each program in the ELEFUNT package computes the relative error of the test function compared to a carefully computed 'exact' value, using purified random arguments in each of several test intervals. The programs produce separate results for each test interval, reporting the number of times the test function was less than, equal to, or greater than the 'exact' value, along with the largest absolute relative error, and the root-mean-square relative error. The base-2 logarithm of the relative errors is then added to the precision to determine the number of bits lost.

A fragment of the report for the exponential function on a Sun Microsystems SOLARIS SPARC system with 128-bit IEEE 754 arithmetic looks something like this:

```
TEST OF EXP(X- 2.8125) VS EXP(X)/EXP( 2.8125)

  2000 RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL
     (-3.4657e+00, -1.1275e+04)

EXP(X-V) WAS LARGER  361 TIMES,
             AGREED 1304 TIMES, AND
        WAS SMALLER  335 TIMES.

THERE ARE 113 BASE 2 SIGNIFICANT DIGITS IN A FLOATING-POINT NUMBER

THE MAXIMUM RELATIVE ERROR OF 5.6809e-34 = 2 **-110.44
   OCCURRED FOR X = -8.268208e+03
THE ESTIMATED LOSS OF BASE 2 SIGNIFICANT DIGITS IS 2.56

THE ROOT MEAN SQUARE RELATIVE ERROR WAS 8.6872e-35 = 2 **-113.15
THE ESTIMATED LOSS OF BASE 2 SIGNIFICANT DIGITS IS 0.00
```

The output is entirely in upper case, because that was the only portable choice when the ELEFUNT Fortran code was written. This author's C and Java translations (described later in **Chapter 22** on page 763) of the ELEFUNT package retain the same output format to facilitate comparing results with those from the Fortran version.

The original ELEFUNT code that computes the relative error looks like this:

```
W = ONE
IF (ZZ .NE. ZERO) W = (Z - ZZ) / ZZ
```

Here, `ZZ` is the 'exact' value, `Z` is the approximate value, and `W` is the relative error. Although that looks simple and straightforward, it conceals two nasty bugs that turned up only after extensive testing.

The first bug was found when the number of tests was increased beyond the default. For some of the elementary functions, it is possible to have `Z` $= 0$ and also `ZZ` $= 0$. When `Z` and `ZZ` are equal, the relative error is exactly zero, and the initialization needs to be readjusted:

```
W = ONE
IF (Z .EQ. ZZ) W = ZERO
IF (ZZ .NE. ZERO) W = (Z - ZZ) / ZZ
```

That can be expressed more succinctly in C as

```
w = (z == zz) ? ZERO : ((zz == ZERO) ? ONE : (z - zz) / zz);
```

The second bug surfaced when new test programs were written to handle functions not covered by the original ELEFUNT code, and in particular, when test regions for erf(x) and erfc(x) were chosen to examine their behavior for subnormal results. Subnormals have one or more leading zero bits in their significands, and thus, have lower precision than normal numbers. Because the relative error leads to a bit-loss determination, the lowered precision needs to be taken into account. Consider, for example, what happens when `zz` is the smallest subnormal number (hexadecimal `0000_0001` in the internal encoding of IEEE 754 32-bit arithmetic, or `+0x1.0p-149` in C99-style hexadecimal encoding), and `z` is the next larger subnormal (hexadecimal `0000_0002`). The ELEFUNT code then reports a relative error of $(2-1)/1 = 1$, and the bit loss is reported to be $24 - \log_2(1) = 24$. However, we really have not 'lost' 24 bits: the two values are actually close, differing only in a single bit.

If both `zz` and `z` are positive, and `zz` is subnormal, then a better estimate of the relative error is `(z - zz) / MAXSUBNORMAL`, where `MAXSUBNORMAL` is the largest subnormal floating-point number. Let's see why that is an improvement. Consider first the case `zz == MAXSUBNORMAL`, and `z` subnormal and close to, but different from, that value. The computed value is then close to `(z - MAXSUBNORMAL) / MAXSUBNORMAL`, which lies in $[-1, +1]$, as expected for a relative error. Next, consider the case `zz == MINSUBNORMAL`: the computation is then roughly `(z - zz) / MAXSUBNORMAL`, which reaches a maximum of almost 1.0 when `z == MAXSUBNORMAL`, and has a minimum of 0 when `z == zz`. That case also lies in $[-1, +1]$. In practice, we can use `MINNORMAL` in place of the nearby `MAXSUBNORMAL`, because the former is more readily available in ELEFUNT as the value `xmin`, and in C/C++ as one of `FLT_MIN`, `DBL_MIN`, or `LDBL_MIN` from `<float.h>`.

Java's `Float.MIN_VALUE` is *not* suitable: it corresponds to `MINSUBNORMAL`. Java is deficient in its failure to provide standard named constants for the minimum normal floating-point number in each precision, but because its arithmetic is guaranteed to be a subset of IEEE 754 arithmetic, portable definitions are easily supplied.

C# has similar deficiencies, but makes them worse with confusing and nonstandard names. In C#, the value `Single.MinValue` is the negative number of largest magnitude, whereas `Single.Epsilon` is the minimum positive subnormal number. There are no `Single` structure members for the machine epsilon, or the smallest normal number, but because the arithmetic is a subset of IEEE 754 arithmetic, portable definitions can be readily provided.

4.26 Extended data types on Hewlett–Packard HP-UX IA-64

As the co-developer (with Intel) of the IA-64 architecture, Hewlett–Packard might be expected to have special support for the many interesting features of that platform, about which we say more in **Chapter 25.4** on page 824.

The HP-UX IA-64 C compiler is unusual in treating the C `long double` data type as a 128-bit IEEE 754 type, rather than the 80-bit type provided by hardware that is normally used for `long double` on other operating systems for IA-32 and IA-64 systems. HP-UX is even more unusual in providing access to two additional floating-point data types in C: they correspond to the 80-bit and 82-bit formats. Those new types are normally not visible unless the `-fpwidetypes` compiler option is used, along with inclusion of at least one of `<float.h>`, `<math.h>`, or `<stdlib.h>`.

The 80-bit type is called `extended`, with constant suffix letter `w` (or `W`) instead of `l` (or `L`), an I/O specifier of `hL` instead of `L`, and math library functions named with a suffix letter `w` instead of `l`. Here is an example of its use to compute the machine epsilon and its square root:

```
% cat extended.c
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    extended x;
    int k;

    k = 0;
    x = 1.0w;

    while ((1.0w + x/2.0w) > 1.0w)
        k--, x /= 2.0w;
    (void)printf("extended machine epsilon = %.20hLe = 2**(%d)\n", x, k);
    (void)printf("square root thereof      = %.20hLe\n", sqrtw(x));

    return (EXIT_SUCCESS);
}

% cc -fpwidetypes extended.c && ./a.out
extended machine epsilon = 1.08420217248550443401e-19 = 2**(-63)
square root thereof      = 3.29272253991359623327e-10
```

By contrast, the 128-bit type is available with normal `long double` programming, but it can also be obtained through a type synonym called `quad`, using a constant suffix letter of `q` (or `Q`), an I/O specifier of `lL`, and math-library functions suffixed with `q`:

```
% cat quad.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int
main(void)
{
    quad x;
    int k;

    k = 0;
    x = 1.0q;

    while ((1.0q + x/2.0q) > 1.0q)
        k--, x /= 2.0q;

    (void)printf("quad machine epsilon = %.35lLe = 2**(%d)\n", x, k);
    (void)printf("square root thereof  = %.35lLe\n", sqrtq(x));

    return (EXIT_SUCCESS);
}

% cc -fpwidetypes quad.c  -lm && ./a.out
quad machine epsilon = 1.92592994438723585305597794258492732e-34 = 2**(-112)
square root thereof  = 1.38777878078144567552953958511352539e-17
```

For portability reasons, there is little need to use the `quad` style of programming, unless it is important to emphasize that a 128-bit type is required in place of an 80-bit one. Even so, care would be needed to limit the exposure of the new suffixes for constants and math library functions, and the new I/O specifiers, because none of them are recognized by compilers on other architectures, not even HP-UX for PA-RISC, or on other operating systems.

The 82-bit data type corresponding to the IA-64 floating-point register format is called `__fpreg`: it differs from the 80-bit format only in having two additional bits in the exponent to guard against premature overflow and underflow. It is not fully supported in C: there is no math library support, no I/O support, and no division. In storage, it occupies 16 bytes (128 bits). In general, it must be converted to other data types for full access.

More information on the IA-64 architecture can be found in Markstein's book [Mar00].

The design of the `mathcw` library makes it easy to support the HP-UX `extended` and `quad` data types, because the algorithm implementations are unaware of the floating-point data type, seeing it only as the generic type `fp_t`. The new types are just `typedef` synonyms for implementation-specific types `__float80` and `__float128` that are always known to the Hewlett–Packard and GNU C compilers on HP-UX for IA-64, even in the absence of `-fpwidetypes` options or the inclusion of system header files. Recent releases of the `gcc` compiler family support those types on a few other platforms.

4.27 Extensions for decimal arithmetic

Proposals have been submitted to the ISO C and C++ Committees to add support for decimal floating-point arithmetic in those languages, and in late 2006, the GNU compiler family offered preliminary support for decimal types and the four basic operations of add, subtract, multiply, and divide. However, the required run-time library was absent, and the debugger was ignorant of decimal types.

Appendix D on page 927 summarizes the main features of the decimal arithmetic system. The Cody/Waite algorithms are designed to support decimal arithmetic as well as binary arithmetic, and the mathcw library provides a complete suite of decimal analogues of the binary floating-point function repertoire.

To insulate the mathcw library from changes to the proposals for decimal arithmetic, the suggested names for the decimal data types are not used directly. Instead, typedef statements define these aliases for the proposed decimal data types:

```
decimal_float        decimal_double
decimal_long_double  decimal_long_long_double
```

On modern machines, they might correspond to 32-bit, 64-bit, 128-bit, and 256-bit storage formats, respectively. Once the type names are standardized, then a simple change to four typedef statements in a single header file will make them conform.

To allow testing on systems lacking support for decimal arithmetic by at least one C or C++ compiler, the decimal type aliases are normally defined to be binary floating-point types, but when certain compiler-defined macros that indicate decimal support are recognized, code in the deccw.h header file defines the aliases to be true decimal types.

Library routines for the four decimal data types have names suffixed with df, d, dl, and dll. Thus, the complete set of prototypes for the square-root functions looks like this:

```
float sqrtf (float x);
double sqrt (double x);
long double sqrtl (long double x);

extended sqrtw (extended x);
quad sqrtq (quad x);

long_long_double sqrtll (long_long_double x);

decimal_float sqrtdf (decimal_float x);
decimal_double sqrtd (decimal_double x);
decimal_long_double sqrtdl (decimal_long_double x);

decimal_long_long_double sqrtdll (decimal_long_long_double x);
```

The long_long_double type supports a future octuple-precision type, but until such support is available in compilers, it is equivalent to the long double type. Its library functions have the suffix ll, and its constants are suffixed LL.

The decimal_long_long_double type is mapped to decimal_long_double until compilers recognize an octuple-precision decimal type.

Elsewhere in this book, when the computation of particular elementary functions is described, we show the prototypes only for the three standard floating-point types, float, double, and long double.

Decimal floating-point constants require distinct type suffixes: DF, DD, DL, and DLL. However, no code in the mathcw package uses those suffixes directly. Instead, the usual wrapper macro for constants, FP(), supplies the suffix. The decimal_double constant suffix is DD rather than D, because Java usurped the D suffix for the binary double data type, even though that suffix is never needed, and rarely used. The considerable similarity between the C, C++, C#, and Java languages mandates avoiding confusion from having the same syntax with quite different meanings in those languages.

C# has a 128-bit scaled decimal type called decimal, intended primarily for monetary calculations. The C# decimal data type offers a precision of about 28 digits, with an approximate range of $[1.0 \times 10^{-28}, 7.8 \times 10^{28}]$, but lacks Infinity, NaN, and subnormals. The exponent range is much too small to be of use for serious numerical floating-point computation. Arithmetic with the C# decimal type underflows abruptly and silently to zero, throws a run-time exception on overflow, has only one rounding direction (*round to nearest*), and lacks sticky exception flags. There is no standard support for elementary functions of decimal type, not even power or square root, which are presumably needed for interest calculations. Decimal constants are suffixed with M, either for *Money*, or for *deciMal* [JPS07, page 91].

Although the C# data type is called *decimal*, it is actually implemented in binary arithmetic, as demonstrated by a run-time computation of its base with a standard algorithm [Mal72] as implemented in this function:

```
static void ShowBase()
{
    decimal a, b;
    int the_base;

    a = 2.0M;

    try
    {
        while ((((a + 1.0M) - a) - 1.0M) == 0.0M)
            a += a;
    }
    catch (System.OverflowException) { }

    b = 2.0M;

    while (((a + b) - a) == 0.0M)
        b += b;

    the_base = (int)((a + b) - a);

    Console.WriteLine("a = {0}  b = {1}  base = {2}",
                      a, b, the_base);
}
```

Execution of the function produces the output

```
a = 39614081257132168796771975168  b = 2  base = 2
```

Future implementations of the `decimal` type in C# are permitted to migrate to the 128-bit IEEE 754 decimal format [JPS07, page 168].

The mathcw package provides the proposed standard system header file, `<decfloat.h>`. That file defines symbolic names for the precisions, the smallest and largest exponents of ten, the machine epsilons, the largest numbers, and the smallest normalized and subnormal numbers, in the decimal floating-point system. Its contents are summarized in **Table D.6** on page 937.

4.28 Further reading

We design the software in this book to work across a broad range of historical, current, and suggested future floating-point architectures. The wide availability of IEEE 754 arithmetic makes it the most important, and although we devote the next chapter to a study of more of its features, it is always wise to consider the views of multiple authors.

Two good sources about IEEE 754 arithmetic are Michael Overton's short monograph *Numerical Computing with IEEE Floating Point Arithmetic* [Ove01], and David Goldberg's widely cited *What Every Computer Scientist Should Know About Floating-Point Arithmetic* [Gol91b]. Its bibliography entry in this book points to corrections and commentary by other authors, and some vendors include it in their online documentation. A revised version appears as an electronic appendix [Gol02] in one volume of a famous series of books on computer architecture, organization, and design written by architects John Hennessy and David Patterson [HP90, HP94, HP96, HP97, PH02, HP03, HP04, PH08, HP12, PH12]. Although those books have appeared in five successive editions, the older ones remain of interest because they include in-depth studies of important computer architectures, and the selection changes with new editions. With a few exceptions, those books concentrate on post-1980 designs. Their coverage does not descend to the circuit level, and they can easily, and usefully, be read by scientists and engineers outside the field of computer science.

Software implementations of IEEE 754 arithmetic are described in [Knu99, Chapter MMIX-ARITH] and [MBdD$^+$10, Chapter 10]. Outlines of hardware implementations of fused multiply-add instructions can be found in [EL04a, Section 8.5.6] and [MBdD$^+$10, Section 8.5.4].

For older computer systems built before 1980, the best single source is *Computer Architecture: Concepts and Evolution* [BB97] by two of the leading architects of the IBM System/360, Gerry Blaauw and Fred Brooks. There is also a book of selected papers from the early years of computing [Ran82].

Several members of the IEEE 754 design committee wrote technical articles to document the design decisions behind that arithmetic system [CKP$^+$79, Coo80, Coo81b, CCG$^+$84, Ste84]. John Palmer and Stephen Morse's book *The 8087 Primer* [PM84] describes Intel's 1980 implementation of an early draft of the design. The IEEE 754 Standard was officially published in 1985 [IEEE85a, IEEE87], augmented for nonbinary bases in the IEEE 854 Standard [ANS87], and revised in 2008 [IEEE08, ISO11]. That latest specification requires both binary and decimal arithmetic, and a fused multiply-add operation.

At a more technical level, the proceedings of the IEEE *Conference on Computer Arithmetic* continue to be a good source of material from the world's leading designers in that area. Although we cite only a few of them in this book, the proceedings volumes, and all of the papers in them, are all documented in an online collection of bibliographic entries for publications about floating-point arithmetic.[3] The most recent prior to the completion of this book is the 23rd conference [MSH$^+$16]. All of the conference papers are available online at a single site.[4]

Besides those conferences, many accounts of current research in computer arithmetic can be found in the journals *IEEE Micro* and the more technical *IEEE Transactions on Computers* and *IEEE Transactions on Circuits and Systems*.

There are two volumes of reprints of important historical research papers on the design of computer arithmetic [Swa90a, Swa90b].

The new *Handbook of Floating-Point Arithmetic* [MBdD$^+$10] and *Modern Computer Arithmetic* [BZ11] appeared just as most of the work on this book was completed, but too late to influence the material in the book.

4.29 Summary

In this chapter, we have discussed several topics that affect how software implements mathematical recipes for the computation of the elementary functions. The most important topic is *error magnification*, because it sets the ultimate mathematical limit on accuracy.

The next most important issue is the behavior of computer arithmetic, both floating-point and integer arithmetic. The programmer must clearly understand the behavior of floating-point arithmetic in the presence of *exceptional conditions*, such as underflow, overflow, and zero divide, and *exceptional operands*, such as IEEE 754 Infinity and NaN, and design the code to handle them properly. Neither premature job termination, nor a nonsensical result, is acceptable.

To write *portable* software, the programmer must be aware of the characteristics of particular implementations of computer arithmetic. They include variations in the degree of conformance to the IEEE 754 Standard, such as the presence or absence of subnormal numbers, one or two kinds of NaN, rounding behavior and mode, and the handling of signed zeros. They also include the wobbling precision that arises for number bases other than 2 or 10.

Long internal registers and fused multiply-add instructions are often beneficial for numerical software, yet we sometimes find in this book that we must take steps to prevent their use.

Testing with multiple compilers on the same platform, and on many different platforms, is imperative. Practical experience shows that NaNs and signed zeros are mishandled by some compilers, so both need special care. Accurate representation of constants may require that they be coded in rational form with exact numerators and denominators, where the denominators are powers of the base, or else they must be reconstructed at run time from the sum of an exact high part and an accurate, but inexact, low part.

The choice of programming language is significant, because some offer only a subset of floating-point types, or lack access to the floating-point environment, or restrict the arithmetic, as C# and Java do. The operating system and library implementations on modern machines make the C language the best choice for writing library software that can be used from many other languages. Nevertheless, the mathcw library code can be used as a reference implementation from which native code in other languages can be derived, often by a straightforward transcription.

[3] See `http://www.math.utah.edu/pub/tex/bib/index-table-f.html#fparith`.

[4] See `http://www.acsel-lab.com/arithmetic/`.

5 The floating-point environment

> HISTORICALLY, MOST NUMERICAL SOFTWARE HAS BEEN WRITTEN WITHOUT REGARD TO EXCEPTIONS . . . , AND MANY PROGRAMMERS HAVE BECOME ACCUSTOMED TO ENVIRONMENTS IN WHICH EXCEPTIONS CAUSE A PROGRAM TO ABORT IMMEDIATELY.
>
> — SUN MICROSYSTEMS, *Numerical Computation Guide* (2000).

Prior to the 1999 ISO C Standard, no standard programming language offered a uniform way to access features of the floating-point environment. Until the advent of IEEE 754 floating-point arithmetic, no such access was even possible, because almost none of the older floating-point systems have programmable rounding directions or precision control, and floating-point exceptions are not dealt with by setting programmer-accessible flags, but instead are trapped by software interrupt handlers that set underflows to zero, and usually terminate the job on overflow or division by zero.

Some older systems do not even provide the luxury of a run-time fixup, but simply proceed with whatever results the hardware generated, such as an overflow wrapped to a small number near the underflow limit, an underflow wrapped to a large number near the overflow limit, and a suppressed zero divide returned as the numerator.

Most higher-level programming languages, including C, are defined so as to be largely independent of the underlying hardware. The sizes of various numerical types are only required to form an ordered hierarchy, and their range and precision are left unspecified, although helpful information may be available via environmental inquiry routines, or predefined constants or macros.

The floating-point environment and the software access to it that is described in this chapter is an exception to this generality, and it is certainly the part of the C99 language that comes closest to the hardware. The widespread adoption of (most of the features of) a single floating-point system by almost all CPU vendors, and even by a few programming languages, such as C#, Java, and PostScript, makes it possible to introduce a software interface to the floating-point environment that is likely to be both useful, and actually used, in practical programs.

Although almost all of the mathcw library can be written without reference to the floating-point environment, apart from use of constants that define range and precision, environment access is essential in one set of C99 functions discussed later on page 139.

5.1 IEEE 754 and programming languages

IEEE 754 arithmetic defines four rounding directions, intended to be selectable during execution. Exceptions normally do not cause software interrupts, but instead are merely recorded in sticky exception flags that, once set, remain set until explicitly cleared by the programmer.

Some floating-point implementations, notably on the Alpha processor, normally freeze the rounding choice at compile time, by generating instructions that encode a fixed rounding direction. On the Alpha, it is possible to select dynamic rounding with additional compiler options, possibly causing a subsequent run-time performance penalty. Embedded processors may not even implement all four rounding modes. The Java language regrettably offers only one of them [KD98], and C# inherits the floating-point deficiencies of Java.

The IEEE 754 Standard did not define a complete software interface when it was published in 1985, five years after the first hardware implementation on the Intel 8087 coprocessor of a draft of the specification. The belief was that such an interface was language dependent, and thus beyond the scope of a hardware standard. It was up to language-standards committees, computer vendors, and compiler and library developers to invent suitable interfaces to the floating-point environment.

Unfortunately, the 1990 ISO C Standard conservatively refuses to acknowledge IEEE 754 arithmetic, despite the fact that tens of millions of processors implementing it in desktop computers were in world-wide use before the 1990 Standard was issued.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_5

The 1991 ISO Fortran Standard [BGA90, FTN91, MR90], known as Fortran 90, and its 1997 successor [ABM$^+$97, ANSI97], known as Fortran 95, similarly fail to acknowledge or support IEEE 754 arithmetic.

The 1998 and 2003 ISO C++ Standards [C++98, C++03a] add minimal support, mostly in the form of features-inquiry constants in the `numeric_limits` class.

The 1999 ISO C Standard [C99] is the first international standard to define a uniform interface to most of the features of IEEE 754 arithmetic.

The 2004 ISO Fortran Standard, known as Fortran 2003, supplies three modules for support of that arithmetic.

5.2 IEEE 754 and the mathcw library

Historically, it has taken at least five years for at least one compiler to conform to an ISO programming-language standard, and about ten years for such conformance to be widely available from many compilers on many platforms. The sad truth is that more than *three decades* will have elapsed from the first implementation of IEEE 754 arithmetic until Fortran programmers will be able to access all of its features portably.

The lack of precise specifications of a programming interface by the various standards bodies left vendors of compilers on their own. In the UNIX environment, at least *five* different and incompatible interfaces have been offered on various systems. That needless variety strongly discourages access to the floating-point environment, and impedes development of portable software.

The approach taken in the mathcw library is to provide the C99 interface to IEEE 754 features by implementing them in terms of whatever support is provided by the underlying operating system. Fortunately, on all UNIX systems, that has been possible without having to resort to processor-specific assembly language, except for access to precision control.

Although there are similarities in the handling of the IEEE 754 environment among some of those operating systems, for clarity and reduction of code clutter, the mathcw library code provides separate implementations of the C99 interface for each of them. A fallback implementation is provided when the host operating system is not supported, or does not have IEEE 754 arithmetic. The implementation behaves sensibly, so that code that uses the C99 routines can function, as long as it is prepared to handle a system that lacks some, or all, of the features of IEEE 754 arithmetic.

5.3 Exceptions and traps

The normal behavior of IEEE 754 floating-point arithmetic when any of the five standard exceptions occur is to record them in the sticky exception flags and silently continue. That preserves run-time performance and control flow. A program that needs to detect such exceptions would typically clear the flags before executing a critical code block, test them on completion of the block, and take remedial action if any are set.

One example of such a situation is using floating-point arithmetic for computations that are believed to be exact, but occasionally, perhaps due to algorithmic error, or unexpected data, might not be. A multiple-precision arithmetic package that uses floating-point, rather than integer, coefficients, would almost certainly need to test exception flags, and particularly, the *inexact* and *underflow* flags. Related examples are given later in **Section 5.7** on page 112 and **Section 5.10** on page 120.

The IEEE 754 and 854 Standards also allow implementations to provide for exceptions to be caught by a *trap handler*, which is a software routine that is invoked when the exception occurs. The routine diagnoses the cause of the exception, and either terminates the job, or returns a specified value to be used in place of the normal result. Normally, traps are disabled, but once the trap handler is registered, traps can be enabled and disabled at will.

The trap description in both standards is almost identical; here is what the more-recent IEEE 854 Standard says:

> ***8. Traps***
>
> *A user should be able to request a trap on any of the five exceptions by specifying a handler for it. He should be able to request that an existing handler be disabled, saved, or restored. He should also be able to determine whether a specific trap handler for a designated exception has been enabled. When an exception whose trap is disabled is signaled, it shall be handled in the manner specified in Section 7 [Exceptions]. When an exception whose trap is enabled is signaled, the execution of the program in which the exception occurred shall be suspended, the trap handler previously specified by the user shall be activated, and a result, if specified in Section 7., shall be delivered to it.*

> ***8.1 Trap Handler***
> *A trap handler should have the capabilities of a subroutine that can return a value to be used in lieu of the exceptional operation's result; this result is undefined unless delivered by the trap handler. Similarly, the flag(s) corresponding to the exceptions being signaled with their associated traps enabled may be undefined unless set or reset by the trap handler. When a system traps, the trap handler invoked should be able to determine the following:*
>
> 1. *Which exception(s) occurred on this operation*
> 2. *The kind of operation that was being performed*
> 3. *The destination's precision*
> 4. *In overflow, underflow, and inexact exceptions, the correctly rounded result, including information that might not fit in the destination's precision*
> 5. *In invalid operation and divide by zero exceptions, the operand values*
>
> ***8.2 Precedence***
> *If enabled, the overflow and underflow traps take precedence over a separate inexact trap.*

A trap handler is clearly complex, and on modern machines with heavily pipelined (overlapped) operation, its requirements may be impossible to satisfy. When multiple instructions are in progress, perhaps even in multiple functional units (adders, multipliers, or dividers), the instruction that caused the exception may not be identifiable, and the exception might not be registered, forcing the trap, until sometime after the instruction that caused it has completed. Floating-point exceptions are then called *imprecise*, and most high-performance computers built since the late 1960s have that feature.

Threads, also known as *lightweight processes*, are separate instances of control flow through the same program in the same address space, and sharing the same global variables. Traps are difficult to handle correctly in multithreaded programs, because the library support for traps is rarely generalized to handle thread-specific traps. If only global traps are available, then registering, enabling, and disabling them requires threads to be able to restrict access to the trap data structures during updates to the data. However, those data structures are usually inside run-time libraries or even the operating-system kernel, and thus, not subject to thread data-access controls.

Even though the IEEE Standards suggest that trap handlers can return a replacement value and continue execution from the point of exception, some floating-point architects take the view expressed in the DEC Alpha architecture handbook [SW95, page I-4-63]:

> In general, it is not feasible to fix up the result value or to continue from the trap.

That is, you may be able to register a trap handler, but it will be invoked at most once, and can do little else but terminate. Much the same situation exists with C signal handlers: you can register a handler to catch the `SIGFPE` floating-point exception, but it may be able to do nothing but print a message and terminate the job.

Interpreted languages, such as hoc, may catch the exception, abort the current statement, print a message, and then return control to top level, ready for the next statement. Here is an example from the PDP-10, where hoc uses the `setjmp()` and `longjmp()` C-library functions to implement catch-and-continue operation:

```
@hoc36
hoc36> 1/0
       /dev/stdin:1:division by zero
hoc36> MAXNORMAL * MAXNORMAL
       /dev/stdin:2:floating point exception
hoc36> MINNORMAL * MINNORMAL
       /dev/stdin:3:floating point exception
```

Trap handling is highly dependent on both the CPU and the operating system, and is unlikely to be available on most systems anyway, so in the mathcw library, we simply pretend that traps do not exist, and that computation on systems with IEEE 754 arithmetic is *nonstop* with sticky exception flags.

5.4 Access to exception flags and rounding control

The file `fenvcw.c` provides the mathcw library with access to the floating-point environment. It preprocesses to a code-free file when the compilation environment is a C99 one with a working `<fenv.h>`, or else includes `fenvx.h` to supply an interface to whatever native support for IEEE 754 features is available.

By analogy with mathcw.h and <math.h>, we supply the user with the header file fenvcw.h as an alternative to <fenv.h>.

The fenvcw.h file provides two opaque data types and a related macro,

```
fenv_t              fexcept_t           FE_DFL_ENV,
```

macros for the four rounding directions,

```
FE_DOWNWARD         FE_TONEAREST        FE_TOWARDZERO       FE_UPWARD,
```

macros for the three precision controls,

```
FE_DBLPREC          FE_FLTPREC          FE_LDBLPREC,
```

and macros for at least six exception-flag masks,

```
FE_ALL_EXCEPT       FE_DIVBYZERO        FE_INEXACT
FE_INVALID          FE_OVERFLOW         FE_UNDERFLOW,
```

as required or recommended by C99.

The exception-flag and rounding-direction masks are specified in Section 7.6 of the 1999 ISO C Standard like this:

> *Each of the macros*
>
> ```
> FE_DIVBYZERO
> FE_INEXACT
> FE_INVALID
> FE_OVERFLOW
> FE_UNDERFLOW
> ```
>
> *is defined if and only if the implementation supports the floating-point exception by means of the functions in 7.6.2. Additional implementation-defined floating-point exceptions, with macro definitions beginning with* FE_ *and an uppercase letter, may also be specified by the implementation. The defined macros expand to integer constant expressions with values such that bitwise ORs of all combinations of the macros result in distinct values.*
>
> *The macro*
>
> ```
> FE_ALL_EXCEPT
> ```
>
> *is simply the bitwise OR of all floating-point exception macros defined by the implementation.*
>
> *Each of the macros*
>
> ```
> FE_DOWNWARD
> FE_TONEAREST
> FE_TOWARDZERO
> FE_UPWARD
> ```
>
> *is defined if and only if the implementation supports getting and setting the represented rounding direction by means of the* fegetround() *and* fesetround() *functions. Additional implementation-defined rounding directions, with macro definitions beginning with* FE_ *and an uppercase letter, may also be specified by the implementation. The defined macros expand to integer constant expressions whose values are distinct nonnegative values.*

There are several points that are important to note about these flags:

- They must be combined with the bitwise-OR operator, *not* by addition. That is an issue for interfacing the C library functions to other programming languages, such as Fortran, that lack bitwise operators.
- None of them is guaranteed to be present, so code that uses them must first test for their existence with preprocessor conditionals. It would have been much cleaner to mandate zero values for unsupported masks, because that would have made them no-ops in bitwise-OR expressions.
- It would have been useful to survey existing practice, and standardize names for additional flags on common processors. For example, the Alpha, AMD64, IA-32, and IA-64 processors all set a flag for subnormal results, but vendors have chosen at least three different macro names for that flag. We check for each of them in fenvcw.h, and when one is found, we define a new exception flag, FE_SUBNORMAL, with the value of the just-found flag, thereby providing a sensible common name for the subnormal flag.

- Apart from the subnormal flag, most CPU architectures provide only the five flags mandated by IEEE 754. However, POWER and PowerPC have a particularly rich set, with nearly *twenty* additional exception flags. GNU/LINUX on those CPUs defines macros for those extra flags, but fails to include them in the definition of `FE_ALL_EXCEPT`. Apple MAC OS X defines symbols only for the basic five on both PowerPC and IA-32. IBM AIX includes in `FE_ALL_EXCEPT` only the five standard exception flags.

- At least one platform, GNU/LINUX on SPARC, violates the specification by having two of the rounding-direction flags overlap the sign bit.

- The proposed revision of the IEEE 854 Standard [ANS87] supplies one additional rounding direction: *round to nearest bigger* or *round-half-up*. That is similar to *round to nearest with ties to even*, except that results that lie exactly halfway between two representable numbers are always rounded to the representable value that is larger in magnitude. The new mode might be called *taxman's rounding*, and is provided because of its widespread use in financial computation. Because few implementations of the C language provide access to decimal floating-point arithmetic, there is no mention of, or support for, the *round-half-up* flag in the ISO C standards.

- A proposal for decimal floating-point arithmetic in C++ [Kla05] extends `<fenv.h>` to contain five additional macros for decimal roundings:

  ```
  FE_DEC_DOWNWARD                              FE_DEC_TONEARESTFROMZERO
  FE_DEC_TONEAREST                             FE_DEC_TOWARDZERO
  FE_DEC_UPWARD
  ```

  The `FE_DEC_TONEARESTFROMZERO` flag corresponds to *round to nearest bigger*. For more on rounding in decimal arithmetic, see **Section D.6** on page 936.

- The IBM specification of decimal floating-point arithmetic [Cow05] defines two additional, but optional, rounding modes: *round half down* (round halfway cases downward), and *round up* (round the magnitude of any nonrepresentable value upward).

- The IBM PowerPC version 6 architecture manual [IBM07, page 13] defines four additional rounding modes: *round to nearest ties away from zero*, *round to nearest ties toward zero*, *round away from zero*, and *round to prepare for shorter precision*.

The `<fenv.h>` and `fenvcw.h` header files also supply prototypes of these 13 C99 functions, each of which is described in more detail shortly:

```
int feclearexcept   (int excepts);
int fegetenv        (fenv_t *envp);
int fegetexceptflag (fexcept_t *flagp, int excepts);
int fegetprec       (void);
int fegetround      (void);
int feholdexcept    (fenv_t *envp);
int feraiseexcept   (int excepts);
int fesetenv        (const fenv_t *envp);
int fesetexceptflag (const fexcept_t *flagp, int excepts);
int fesetprec       (int prec);
int fesetround      (int mode);
int fetestexcept    (int excepts);
int feupdateenv     (const fenv_t *envp);
```

It is important to note that in the original 1999 ISO C Standard, some of them were defined as functions of type `void`, but that flaw was fixed by two subsequent technical corrigenda. The functions all return a negative value on failure (-1 in the mathcw implementation), and a value of zero or greater on success.

5.5 The environment access pragma

The `fenvcw.h` file sets the `FENV_ACCESS` pragma to serve as a warning to the compiler that the floating-point environment may be manipulated by the program, preventing certain optimizations, such as code movement, or retention of variables in registers.

Here is what the Standard says about that pragma, with some paragraph breaks inserted to improve readability:

> *The `FENV_ACCESS` pragma provides a means to inform the implementation when a program might access the floating-point environment to test floating-point status flags or run under non-default floating-point control modes. [The purpose of the `FENV_ACCESS` pragma is to allow certain optimizations that could subvert flag tests and mode changes (e.g., global common subexpression elimination, code motion, and constant folding). In general, if the state of `FENV_ACCESS` is "off", the translator can assume that default modes are in effect and the flags are not tested.]*
>
> *The pragma shall occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement.*
>
> *When outside external declarations, the pragma takes effect from its occurrence until another `FENV_ACCESS` pragma is encountered, or until the end of the translation unit.*
>
> *When inside a compound statement, the pragma takes effect from its occurrence until another `FENV_ACCESS` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement.*
>
> *If this pragma is used in any other context, the behavior is undefined.*
>
> *If part of a program tests floating-point status flags, sets floating-point control modes, or runs under non-default mode settings, but was translated with the state for the `FENV_ACCESS` pragma "off", the behavior is undefined. The default state ("on" or "off") for the pragma is implementation-defined. (When execution passes from a part of the program translated with `FENV_ACCESS` "off" to a part translated with `FENV_ACCESS` "on", the state of the floating-point status flags is unspecified and the floating-point control modes have their default settings.)*

The Standard leaves the default state of `FENV_ACCESS` up to the implementation. However, the only safe setting when the floating-point environment is accessed is "on", so that is what `fenvcw.h` automatically sets. That may inhibit some compiler optimizations, but only a few source files are likely to include that header file, so the run-time penalty is likely to be small. Performance-conscious programmers should in any event bracket their use of the floating-point environment functions like this:

```
#include "fenvcw.h"

#pragma FENV_ACCESS OFF
... ordinary code here ...
#pragma FENV_ACCESS ON
... delicate environment-manipulating code here ...
#pragma FENV_ACCESS OFF
... more ordinary code ...
```

5.6 Implementation of exception-flag and rounding-control access

The implementation code in `fenvx.h` is lengthy, with separate sections for each of several different platforms, so we do not show all of it here. Instead, we exhibit a typical implementation, that for older Sun Microsystems SOLARIS systems. Newer SOLARIS systems have full C99 support from the vendor.

Five of the functions provide the essential core of support. In the following sections, we present their code for SOLARIS, prefixed with short descriptions from the 1999 ISO C Standard and its technical corrigenda.

5.6.1 Clearing exception flags: `feclearexcept()`

> *The `feclearexcept()` function attempts to clear the supported floating-point exceptions represented by its argument.*
>
> *The `feclearexcept()` function returns zero if the `excepts` argument is zero or if all the specified exceptions were successfully cleared. Otherwise, it returns a nonzero value.*

```
int
(feclearexcept)(int excepts)
{   /* return 0 on success, or -1 on failure */
    (void)fpsetsticky(fpgetsticky() & ~((fp_except)(excepts)));

    return (FE_SUCCESS);
}
```

5.6.2 Getting the rounding direction: `fegetround()`

> *The `fegetround()` function gets the current rounding direction.*
>
> *The `fegetround()` function returns the value of the rounding direction macro representing the current rounding direction or a negative value if there is no such rounding direction macro or the current rounding direction is not determinable.*

```
int
(fegetround)(void)
{   /* return current rounding mode, or -1 on failure */
    return ((int)fpgetround());
}
```

5.6.3 Raising exception flags: `feraiseexcept()`

> *The `feraiseexcept()` function attempts to raise the supported floating-point exceptions represented by its argument. [The effect is intended to be similar to that of floating-point exceptions raised by arithmetic operations.]*
>
> *Hence, enabled traps for floating-point exceptions raised by this function are taken. The specification in F.7.6 is in the same spirit.*
>
> *The order in which these floating-point exceptions are raised is unspecified, except as stated in F.7.6. Whether the `feraiseexcept()` function additionally raises the* inexact *floating-point exception whenever it raises the* overflow *or* underflow *floating-point exception is implementation-defined.*
>
> *The `feraiseexcept()` function returns zero if the excepts argument is zero or if all the specified exceptions were successfully raised. Otherwise, it returns a nonzero value.*

```
int
(feraiseexcept)(int excepts)
{   /* return 0 on success, or -1 on failure */
    (void)fpsetsticky(fpgetsticky() | ((fp_except)(excepts)));

    return (FE_SUCCESS); /* always succeeds (fpsetsticky()
                            has no documented return value) */
}
```

We assume a simple facility where exceptions set the sticky flags, but do not trap to an external interrupt handler.

The reason that raising the *overflow* or *underflow* flags can also raise the *inexact* flag is that on some architectures, the only practical way to set the flags is to execute a floating-point instruction that has the exception as a side effect. The IEEE 754 Standard requires that *inexact* be set whenever *overflow* or *underflow* are set.

5.6.4 Setting the rounding direction: `fesetround()`

> *The `fesetround()` function establishes the rounding direction represented by its argument `round`. If the argument is not equal to the value of a rounding direction macro, the rounding direction is not changed.*
>
> *The `fesetround()` function returns zero if and only if the requested rounding direction was established.*

```
int
(fesetround)(int round)
{   /* return 0 on success, or -1 on failure */
    int result;
    switch (round)
    {
    case FE_DOWNWARD:            /*FALLTHROUGH*/
    case FE_TONEAREST:           /*FALLTHROUGH*/
    case FE_TOWARDZERO:          /*FALLTHROUGH*/
    case FE_UPWARD:
        (void)fpsetround((fp_rnd)(round));
        result = FE_SUCCESS;
        break;

    default:
        result = FE_FAILURE;
        break;
    }

    return (result);
}
```

5.6.5 Testing exception flags: `fetestexcept()`

> *The `fetestexcept()` function determines which of a specified subset of the floating-point exception flags are currently set. The `excepts` argument specifies the floating-point status flags to be queried. [This mechanism allows testing several floating-point exceptions with just one function call.]*
>
> *The `fetestexcept()` function returns the value of the bitwise OR of the floating-point exception macros corresponding to the currently set floating-point exceptions included in `excepts`.*

```
int
(fetestexcept)(int excepts)
{   /* return bitwise-OR of the floating-point sticky exception
       flags */
    return ((int)(fpgetsticky() & (fp_except)(excepts)));
}
```

5.6.6 Comments on the core five

The five functions are short, requiring little more than possibly a conditional test, the calling of one or two other functions that provide the native IEEE 754 feature access, and a return.

The names of the five functions are parenthesized in our implementation to prevent unwanted macro expansion, a valuable feature of the C preprocessor that seems not to be widely known or appreciated.

The private macros `FE_FAILURE` and `FE_SUCCESS` have the values -1 and 0, respectively. They are *not* available externally to users of `fenvcw.h`.

5.7 Using exception flags: simple cases

To illustrate how the floating-point environment functions can be used in a computation, consider the following task: *Compute the largest factorial that can be represented exactly in the host arithmetic.* What is needed to solve the problem is to clear the exception flags, compute factorials iteratively until the next one to be used raises the *inexact* flag, and then report the most recent exact result.

Here is a program to do that:

```
#include <fenv.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#pragma FENV_ACCESS ON

int
main(void)
{
    int k;
    double nextfact, nfact;

    (void)feclearexcept(FE_ALL_EXCEPT);
    nfact = 1;

    for (k = 1; k < 40; ++k)
    {
        nextfact = nfact * (k + 1);

        if (fetestexcept(FE_INEXACT))
            break;

        nfact = nextfact;
    }

    (void)printf("Largest exact factorial: %d! = %.0f\n",
                 k, nfact);

    return (EXIT_SUCCESS);
}
```

Here is what it reports on any system with IEEE 754 arithmetic:

```
Largest exact factorial: 22! = 1124000727777607680000
```

That result may seem surprising: $\log_2(22!) \approx 69.92$, so about 70 bits are needed to represent the number, and we have only 53 in the significand of the IEEE 754 64-bit format. However, examination of the hexadecimal representation, $22! =$ `0x3c_eea4_c2b3_e0d8_0000`, shows that only the first 52 bits can be nonzero, so the number can indeed be represented exactly in that format.

Sometimes, there is a fast way to compute a result, but the fast way is not always safe, and the safe way is always slow. One such example is the computation of the Euclidean norm in two dimensions, or equivalently, finding the longest side of a right triangle by Pythagoras' Theorem. The desired result is just $\sqrt{x^2 + y^2}$, but if we use that formula directly, premature overflow or underflow can produce wildly incorrect results. The C library provides a standard function, `hypot()`, to do that computation, but we give our sample function a different name, `e2norm()`, to distinguish it from the standard one, and as usual, we implement it for a generic floating-point type. We discuss the `hypot()` function in more detail later in **Section 8.2** on page 222 and **Section 8.3** on page 227.

Because IEEE 754 arithmetic encourages nonstop computing, we can clear the exception flags, do the fast computation, test the exception flags to see if either underflow or overflow occurred, and if so, redo the computation more carefully. Here is code that does just that:

```
#include "e2norm.h"

#pragma FENV_ACCESS ON

fp_t
E2NORM(fp_t x, fp_t y)
{
    fp_t result;

    if (x == ZERO)
        result = FABS(y);
    else if (y == ZERO)
        result = FABS(x);
    else
    {
        (void)feclearexcept(FE_ALL_EXCEPT);
        result = SQRT(x * x + y * y);

        if ( fetestexcept(FE_OVERFLOW | FE_UNDERFLOW) )
        {
            fp_t ratio, xabs, yabs;

            xabs = FABS(x);
            yabs = FABS(y);

            if (xabs > yabs)
            {
                ratio = yabs / xabs;
                result = xabs * SQRT(ONE + ratio * ratio);
            }
            else
            {
                ratio = xabs / yabs;
                result = yabs * SQRT(ratio * ratio + ONE);
            }
        }
    }

    return (result);
}
```

Notice that we test for zero arguments to guarantee an exact result in those cases, but there are no tests for Infinity or NaN arguments. We do not need them here, but to see why, we must consider what happens for various inputs. The computation is symmetric in x and y, so we only need to consider half the number of possible cases:

x is a NaN and/or y is a NaN: The first computation sets `result` to NaN, but sets only the *invalid* flag, so we are done.

$x = \infty$ and/or y is a NaN: The first computation sets `result` to NaN, but sets only the *invalid* flag, so we are done.

$x = \infty$ and/or $y = \infty$: The first computation sets `result` to Infinity, but sets only the *inexact* flag, so we are done.

$x =$ tiny and/or $y =$ tiny: If underflow occurs in either product, then the *underflow* and *inexact* flags are set. The result may be zero, which is incorrect, but we abandon it anyway, and recompute with a stable formula that requires more work.

$x =$ large and/or $y =$ tiny: If either overflow or underflow occurs in the products, then an exception flag is set that causes the stable computation to be done.

$x =$ large and/or $y =$ large: If overflow occurs in either of the products, then the *overflow* flag is set, causing the stable computation to be done.

$x = \pm 0$ and $y = \pm 0$: No exceptions are set, and the computed result from the first absolute value assignment is zero (correct).

$x =$ moderate and $y =$ moderate: No exceptions other than, probably, *inexact* are set, and the result from the first computation is correct.

Whether our version of `e2norm()` is faster in practice than a version that uses only the stable algorithm is platform dependent. It depends on the relative cost of division versus function calls to clear and test flags, and it also requires that IEEE 754 exception-flag access be available. As long as flag clearing is reasonably fast, and if most of the calls

Table 5.1: Interval-arithmetic operations. Each number X is strictly contained in an interval $[\underline{x}, \overline{x}]$, where $\underline{x} \leq \overline{x}$. The complications from Infinity and NaN, and from division by intervals containing zero, are ignored here. Computation of $\underline{\min}(u, v)$ must be done with *round-to-minus-infinity*, and $\overline{\max}(u, v)$ must be done with *round-to-plus-infinity*.

$$
\begin{aligned}
X + Y &= [\underline{\underline{x} + \underline{y}}, \overline{\overline{x} + \overline{y}}] \\
X - Y &= [\underline{\underline{x} - \overline{y}}, \overline{\overline{x} - \underline{y}}] \\
X \cdot Y &= [\underline{\min}(\underline{x} \cdot \underline{y}, \underline{x} \cdot \overline{y}, \overline{x} \cdot \underline{y}, \overline{x} \cdot \overline{y}), \overline{\max}(\underline{x} \cdot \underline{y}, \underline{x} \cdot \overline{y}, \overline{x} \cdot \underline{y}, \overline{x} \cdot \overline{y})] \\
X / Y &= [\underline{\min}(\underline{x}/\underline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}, \overline{x}/\overline{y}), \overline{\max}(\underline{x}/\underline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}, \overline{x}/\overline{y})]
\end{aligned}
$$

to the function have arguments that cause neither overflow nor underflow, then the slower computation is avoided most of the time, and our version is faster on average than the alternative code.

Our example shows that the limitation of finite range in computer arithmetic makes mathematically simple computations considerably more complex to do accurately on a computer.

5.8 Using rounding control

One of the most important applications for IEEE 754 rounding control is for the implementation of *interval arithmetic*. That is an area of growing importance, and done properly, can provide rigorous upper and lower bounds for computed results. When those bounds are close, one can have confidence in the computed values, though not necessarily in the algorithm that led to them! When they are far apart, then it is likely that even tiny changes in the input data would have produced a large change in the output, suggesting that the computational algorithm may be unstable, or equivalently, that a result obtained by the same algorithm, but using normal scalar arithmetic, may not be trustworthy.

The four primitive operations for interval arithmetic are summarized in Table 5.1. In reality, those four operations are insufficient: interval versions of all of the elementary functions are required too, and they are decidedly nontrivial to compute. However, for simplicity, here we consider only the case of addition in interval arithmetic.

Intervals can be represented in more than one way, but the simplest seems to be as pairs of endpoints. A suitable data structure might then look like this:

```
typedef struct { fp_t lo; fp_t hi; } interval_t;
```

In the absence of operator overloading in the programming language, arithmetic expressions for interval data need to be rewritten with function calls for every binary and unary operator. To make it easier to pass the result of one operation to the next, it is desirable to make the interval functions return an interval object, so that we can use nested function calls. The interval object requires storage, but we cannot safely use internal static storage for it, and we certainly do not want the substantial overhead of dynamic storage, so we provide a pointer to the result object as the first argument, and declare the result object as a local variable in the calling routine.

Although C89 added support for passing structure arguments, there is a long tradition in C programming of passing pointers to structures instead. They require less stack space, fewer instructions in the caller to pass, and fewer instructions to access in the called routine. We therefore use pointer arguments, and because the functions are at the core of numerically intensive calculations, we reduce the overhead of their use by declaring them as `inline` functions.

We also must assume that interval arithmetic is used together with normal arithmetic, so we need to make sure that each interval primitive preserves the original rounding direction.

Here is one way to write a function for interval addition:

```
inline interval_t *
I_ADD(interval_t *sum, interval_t *x, interval_t *y)
{    /* interval sum of x + y */
    int round;
    round = fegetround();
    (void)fesetround(FE_DOWNWARD);
    sum->lo = x->lo + y->lo;
    (void)fesetround(FE_UPWARD);
    sum->hi = x->hi + y->hi;
    (void)fesetround(round);

    return (sum);
}
```

Notice that our code requires four calls to get or set the rounding mode. We can eliminate one of them by noting that rounding a sum upward is the same as the negative of rounding the sum of the negatives downward. Thus, we can rewrite the addition function like this:

```
inline interval_t *
I_ADD(interval_t *sum, interval_t *x, interval_t *y)
{   /* interval sum of x + y */
    int round;

    round = fegetround();
    (void)fesetround(FE_DOWNWARD);
    sum->lo = x->lo + y->lo;
    sum->hi = -(-x->hi + (-y->hi));
    (void)fesetround(round);

    return (sum);
}
```

5.9 Additional exception flag access

Six of the remaining eight functions use the opaque data types `fenv_t` and `fexcept_t` defined in `fenvcw.h` and `<fenv.h>`. The structure and contents of those types are not standardized, and user code should never reference their internals. The mathcw library uses simple definitions of those types:

```
typedef struct
{
    int fe_sticky_flags;
    int unused_padding[63]; /* longest native fenv_t is 100 bytes */
} fenv_t;

typedef struct
{
    int fe_sticky_flags;
    int unused_padding[3]; /* longest native fexcept_t is 8 bytes */
} fexcept_t;
```

For portability and safety, the data structures are padded to at least twice the longest known lengths of the native types, in case code compiled with mathcw header files is linked with native libraries, instead of with the mathcw library. The reverse case of compilation with native header files and linking with the mathcw library is less safe, because a few systems represent `fexcept_t` with a `short int`.

The library also provides the standard macro that points to a default environment object:

```
extern fenv_t      __mcw_fe_dfl_env;
#define FE_DFL_ENV (&__mcw_fe_dfl_env)
```

The default object is initialized like this in `fenvx.h`:

```
const fenv_t __mcw_fe_dfl_env = { 0 };
```

Although we initialize only one cell of the data structure that way, the rules of C guarantee that the remaining cells are also zero.

The six functions can be defined in terms of the core five presented in **Section 5.6** on page 110, and our definitions of the additional functions suffice for all platforms. As before, in the following sections, we prefix them with their descriptions from the 1999 ISO C Standard and its technical corrigenda.

5.9.1 Getting the environment: `fegetenv()`

> *The `fegetenv()` function attempts to store the current floating-point environment in the object pointed to by `envp`.*
>
> *The `fegetenv()` function returns zero if the environment was successfully stored. Otherwise, it returns a nonzero value.*

For the mathcw library, the floating-point environment is just the complete set of sticky exception flags, which we already know how to retrieve with `fetestexcept()`. The code is then straightforward:

```
int
(fegetenv)(fenv_t *envp)
{   /* store floating-point environment in *envp, return 0 on
       success, or -1 on failure */
    int result;

    if (envp != (fenv_t*)NULL)
    {
        envp->fe_sticky_flags = fetestexcept(FE_ALL_EXCEPT);
        result = FE_SUCCESS;
    }
    else
        result = FE_FAILURE;

    return (result);
}
```

5.9.2 Setting the environment: `fesetenv()`

> *The `fesetenv()` function attempts to establish the floating-point environment represented by the object pointed to by `envp()`. The argument `envp` shall point to an object set by a call to `fegetenv()` or `feholdexcept()`, or equal a floating-point environment macro. Note that `fesetenv()` merely installs the state of the floating-point status flags represented through its argument, and does not raise these floating-point exceptions.*
>
> *The `fesetenv()` function returns zero if the environment was successfully established. Otherwise, it returns a nonzero value.*

We assume that traps are never used, so that `feraiseexcept()` can be used to set the sticky exception flags saved by a previous call to `fegetenv()`. If system-dependent traps are set, then we would have to first disable them, set the flags, and then restore them; however, we cannot do so in a portable library because C99 does not provide an interface to floating-point traps.

```
int
(fesetenv)(const fenv_t *envp)
{   /* set floating-point environment from *envp (possibly
       FE_DFL_ENV), return 0 on success, or -1 on failure */
    int result;

    if (envp != (fenv_t*)NULL)
        result = feraiseexcept(envp->fe_sticky_flags);
    else
        result = FE_FAILURE;
    return (result);
}
```

5.9.3 Getting exception flags: `fegetexceptflag()`

> *The `fegetexceptflag()` function attempts to store an implementation-defined representation of the states of the floating-point status flags indicated by the argument `excepts` in the object pointed to by the argument `flagp`.*
>
> *The `fegetexceptflag()` function returns zero if the representation was successfully stored. Otherwise, it returns a nonzero value.*

This function is little more than an alternate interface to `fetestexcept()`, storing the requested flags in a location determined by a pointer argument, instead of returning them as a function value.

```
int
(fegetexceptflag)(fexcept_t *flagp, int excepts)
{   /* set floating-point exceptions flags in *envp, return 0 on
       success, or -1 on failure */
    int result;

    if (flagp != (fexcept_t *)NULL)
    {
        flagp->fe_sticky_flags = fetestexcept(excepts);
        result = FE_SUCCESS;
    }
    else
        result = FE_FAILURE;

    return (result);
}
```

5.9.4 Setting exception flags: `fesetexceptflag()`

> *The `fesetexceptflag()` function attempts to set the floating-point status flags indicated by the argument `excepts` to the states stored in the object pointed to by `flagp`. The value of `*flagp` shall have been set by a previous call to `fegetexceptflag()` whose second argument represented at least those floating-point exceptions represented by the argument `excepts`. This function does not raise floating-point exceptions, but only sets the state of the flags.*
>
> *The `fesetexceptflag()` function returns zero if the `excepts` argument is zero or if all the specified flags were successfully set to the appropriate state. Otherwise, it returns a nonzero value.*

As with `fesetenv()`, we assume that traps are never used, so that we can safely use `feraiseexcept()` to set the sticky exception flags. The function is then a simple wrapper around a call to set the flags.

```
int
(fesetexceptflag)(const fexcept_t *flagp, int excepts)
{   /* set sticky flags selected by excepts to states stored in
       *flagp, return 0 on success, or -1 on failure */
    int result;

    if (flagp != (const fexcept_t *)NULL)
        result = feraiseexcept(flagp->fe_sticky_flags & excepts);
    else
        result = FE_FAILURE;

    return (result);
}
```

5.9.5 Holding exception flags: `feholdexcept()`

> *The* `feholdexcept()` *function saves the current floating-point environment in the object pointed to by* `envp`*, clears the floating-point status flags, and then installs a non-stop (continue on floating-point exceptions) mode, if available, for all floating-point exceptions. [IEC 60559 systems have a default non-stop mode, and typically at least one other mode for trap handling or aborting; if the system provides only the non-stop mode then installing it is trivial. For such systems, the* `feholdexcept()` *function can be used in conjunction with the* `feupdateenv()` *function to write routines that hide spurious floating-point exceptions from their callers.]*

Because we assume traps are absent, the function simply requires saving the current sticky exception flags, and then clearing them.

```
int
(feholdexcept)(fenv_t *envp)
{   /* save current floating-point environment in *envp, clear
       all exception flags, install nonstop mode, and return
       0 on success, or -1 on failure */
    int result;

    if (envp != (fenv_t*)NULL)
    {
        envp->fe_sticky_flags = fetestexcept(FE_ALL_EXCEPT);
        result = feclearexcept(FE_ALL_EXCEPT);
    }
    else
        result = FE_FAILURE;

    return (result);
}
```

5.9.6 Updating the environment: `feupdateenv()`

> *The* `feupdateenv()` *function attempts to save the currently raised floating-point exceptions in its automatic storage, install the floating-point environment represented by the object pointed to by* `envp`*, and then raise the saved floating-point exceptions. The argument* `envp` *shall point to an object set by a call to* `feholdexcept()` *or* `fegetenv()`*, or equal a floating-point environment macro.*
>
> *The* `feupdateenv()` *function returns zero if all the actions were successfully carried out. Otherwise, it returns a nonzero value.*

Because we assume that no traps are set, we can update the floating-point environment just by restoring the saved exception flags with `feraiseexcept()`.

```
int
(feupdateenv)(const fenv_t *envp)
{
   /* save exceptions, install environment from *envp, raise
       saved exceptions, and return 0 on success, or -1 on
       failure */
    int result;

    if (envp != (fenv_t*)NULL)
        result = feraiseexcept(envp->fe_sticky_flags);
    else
        result = FE_FAILURE;

    return (result);
}
```

5.9.7 Comments on the six functions

These six functions are notably more complex than the core five described in **Section 5.6** on page 110.

Because the `fenv_t` type is opaque, and no functions are provided to manipulate its contents, we can only use `fegetenv()` and `fesetenv()` in pairs to save and restore the floating-point environment around a block of code. No provision is made for setting or clearing of user-specified exception flags in a saved environment.

The functions `fegetexceptflag()` and `fesetexceptflag()` allow similar save/restore bracketing of computations, and in addition, allow us to set zero or more exception flags when the environment is restored. However, there is no provision for clearing those flags. The `feraiseexcept()` function could almost do that job, but it can cause exception traps to be taken.

It would appear to be a design flaw in C99 that the task of *setting* the flags is not separated from possible invocation of a trap handler in `feraiseexcept()`. The reason that the separation of setting from trapping could not be made is that some hardware implementations of IEEE 754 arithmetic do not permit that distinction. Had that not been the case, it would have been cleaner to have two independent functions, `fesetexcept()` and `fetrapexcept()`, each taking a single argument with a bitwise-OR of the flags to be set or trapped. Of course, we have `fesetexceptflag()` which could do the job, but it is burdened with an additional opaque argument that is supposed to have been returned by a previous call to `fegetexceptflag()`.

The function pair `feholdexcept()` and `feupdateenv()` can be used to wrap a block of code that starts with all exception flags clear, and all traps disabled, so that nonstop operation is known to be in effect. On return from `feupdateenv()`, all flags and traps are restored to their original settings.

The functions `fegetenv()` and `feupdateenv()` can be used to wrap a code block where the programmer does not care what the initial environment for the block is, but just wants to restore the environment when the block completes.

5.10 Using exception flags: complex case

In this section, we present a small test program that exercises several of the functions for manipulating the floating-point environment. We start by defining a header block that can be used with either native C99 compilation, or with the mathcw library:

```
#if defined(USE_MCW)

#include <mathcw.h>
#include <fenvcw.h>

#else /* native C99 environment */

#include <math.h>
#include <fenv.h>

extern double infty (void);
extern double j0    (double);
extern double j1    (double);
extern double yn    (int, double);

#endif /* defined(USE_MCW) */
```

In the native environment, we supply prototypes for four library functions. The first is a convenient mathcw library extension to C99 to compute Infinity at run-time. The other three are Bessel functions (see **Chapter 21** on page 693) that are not part of C99, but are mandated by POSIX. They are computationally difficult, and are certainly not provided in hardware on any current desktop CPU architecture. We therefore expect their computation to set one or more exception flags. Next, we introduce some macros to shorten and clarify the test code:

```
#define PRINTF          (void)printf
#define CLR()           result = feclearexcept(FE_ALL_EXCEPT)
#define TEST(s)         { CLR(); s; show_flags(#s); }
#define TEST_NO_CLR(s)  { s; show_flags(#s); }
```

The last two macros exploit the C89 preprocessor # operator to convert an argument to a character string, so that we can execute a test statement, and also display it in the output.

Although we expect the library functions to be successful, it is nevertheless a good idea to check their return codes and report any failures, so we provide a function to do that:

```
void
check(int result, const char *s)
{
    if (result != 0)
        PRINTF("%s returned %d [FAILURE]\n", s, result);
}
```

To make the output compact, we define a function to display a list of the floating-point sticky exception flags that are found to be set. Because `printf()` processing is complex, it could alter the flags, so we must be careful to save them before printing their values:

```
void
show_flags(const char *s)
{
    int r[6];

    r[0] = fetestexcept(FE_DIVBYZERO);
    r[1] = fetestexcept(FE_INEXACT);
    r[2] = fetestexcept(FE_INVALID);
    r[3] = fetestexcept(FE_OVERFLOW);

#if defined(FE_SUBNORMAL)
    r[4] = fetestexcept(FE_SUBNORMAL);
#else
    r[4] = 0;
#endif

    r[5] = fetestexcept(FE_UNDERFLOW);

    PRINTF("Flags after %37s :", s);

    if (r[0]) PRINTF(" DIVBYZERO");
    if (r[1]) PRINTF(" INEXACT");
    if (r[2]) PRINTF(" INVALID");
    if (r[3]) PRINTF(" OVERFLOW");
    if (r[4]) PRINTF(" SUBNORMAL");
    if (r[5]) PRINTF(" UNDERFLOW");

    PRINTF("\n");
}
```

The `FE_SUBNORMAL` flag is not one of the IEEE 754 standard flags, but it is available on at least Alpha, AMD64, IA-32, and IA-64 CPUs, although its native names in those environments differ. The name differences are hidden inside `fenvcw.h`. We therefore check that the flag name is defined before using it.

We are now ready for the `main()` program. Its job is to execute a series of single floating-point statements, and report the exception flags that they set. The flags are normally cleared before each test statement. All floating-variables are declared `volatile` to prevent compile-time evaluation, and to keep the code simple, we assume that the compiler obeys that keyword; otherwise, we would have to clutter the test code with calls to a `store()` function. The variables `x` and `y` hold floating-point results, and `z` is always assigned a computed zero value.

```
int
main(void)
{
    fenv_t env;
    int result;
    volatile double x, y, z;

    PRINTF("sizeof(fenv_t) = %u\n", sizeof(fenv_t));

    TEST(result = feclearexcept(FE_ALL_EXCEPT));
    check(result, "feclearexcept()");

    TEST(x = j0(1.0));
    TEST(z = x - x);                    /* 0.0 */
    TEST(x = 1.0 / z);                  /* 1.0 / 0.0 -> Inf */

    TEST(result = feholdexcept(&env));
    check(result, "feholdexcept(&env)");

    TEST(x = j1(1.0));
    TEST(z = x - x);                    /* 0.0 */
    TEST(x = 1.0 / z);                  /* 1.0 / 0.0 -> Inf */
    TEST(x = z / z);                    /* 0.0 / 0.0 -> NaN */
    TEST(x = nan(""));
    TEST(y = x - x);                    /* NaN - NaN -> NaN */
    TEST(x = infty());
    TEST(z = x - x);                    /* Inf - Inf -> NaN */
    TEST(x = yn(2,DBL_MIN));            /* should be Inf */
    TEST(x = DBL_MIN; y = x / 256.0);   /* exact subnormal */
    TEST(x = DBL_MIN; y = x / 100.0);   /* approximate subnormal */

    TEST_NO_CLR(result = feupdateenv(&env));
    check(result, "feupdateenv(&env)");

    TEST(result = feupdateenv(&env));
    check(result, "feupdateenv(&env)");

    return (EXIT_SUCCESS);
}
```

A run of the test program on a SOLARIS SPARC system with a native C99 compiler looks like this:

```
% c99 fecmplx.c -lm ../libmcw.a && ./a.out
sizeof(fenv_t) = 100
Flags after result = feclearexcept(FE_ALL_EXCEPT) :
Flags after                          x = j0(1.0) : INEXACT
Flags after                            z = x - x :
Flags after                          x = 1.0 / z : DIVBYZERO
Flags after           result = feholdexcept(&env) :
Flags after                          x = j1(1.0) : INEXACT
Flags after                            z = x - x :
Flags after                          x = 1.0 / z : DIVBYZERO
Flags after                            x = z / z : INVALID
Flags after                          x = nan("") :
Flags after                      y = x - nan("") :
Flags after                          x = infty() : DIVBYZERO
Flags after                            z = x - x : INVALID
Flags after                    x = yn(2,DBL_MIN) : INEXACT OVERFLOW
Flags after           x = DBL_MIN; y = x / 256.0 :
Flags after           x = DBL_MIN; y = x / 100.0 : INEXACT UNDERFLOW
Flags after            result = feupdateenv(&env) : INEXACT UNDERFLOW
Flags after            result = feupdateenv(&env) :
```

This system has a well-engineered floating-point implementation, and as expected, the first test statement clears the flags, and none of the environmental functions returns a failure code. The result of the first Bessel function call is about 0.765, and we find only the *inexact* flag to be set. The subtraction $x - x$ is exact, and sets no flags. The division 1.0/0.0 sets the *divbyzero* flag, and only that flag is set when the environment is saved by the call to `feholdexcept()`.

The next four statements set the expected flags, but the failure of the `nan()` function to set flags tells us that the implementation simply returns a stored quiet NaN, rather than generating one on-the-fly, as the `mathcw` version does. The C99 Standard does not specify the behavior, so both implementations conform. Subtraction of two quiet NaNs does not set an exception flag, as we reported in **Section 4.14** on page 79. However, subtraction of two infinities *does* set the *invalid* flag, and produces a quiet NaN.

The test system has a binary base, so division of the smallest normalized floating-point number, `DBL_MIN`, by $2^8 = 256.0$ is an exact operation, although it produces a subnormal result, and no flags are set. However, division by 100.0 is an inexact operation, and both the *inexact* and *underflow* flags are set, as we discussed in **Section 4.12** on page 78.

In the second-last test statement, we suppress clearing of the flags, and simply invoke `feupdateenv()`. After that call, we should therefore see the previous two flags set, as well as the saved *divbyzero* flag. However, that flag is lost by the implementation on this system.

The last test statement clears the flags, and once again restores the saved flags, but this implementation has lost the *divbyzero* flag.

The test has been repeated on several different platforms, with native C99 implementations when available, and on all systems, with the `mathcw` library. The results are generally similar. One system, GNU/LINUX on AMD64, warns `feupdateenv is not implemented and will always fail`. The same operating system on IA-64 does not produce a warning, but `feholdexcept()` reports failure.

Most native implementations lose the *divbyzero* flag, although the GNU/LINUX IA-64 system preserves it. The OSF/1 Alpha system sets the *subnormal* flag in the call to `yn()`, but not in the division by 100.0; a debugger session shows that the system uses abrupt underflow to zero, instead of generating a subnormal result. The GNU/LINUX Alpha system has subnormals too, but defines the exception flags as enumeration constants, rather than as preprocessor macros, so our code disables the test for the *subnormal* flag.

The experiments are useful, and tell us that although we can clear, set, and test exception flags fairly consistently across platforms, we need to allow for abrupt underflow without subnormals, and saving and restoring the floating-point environment around a block of code may not preserve flags that were in effect when `feholdexcept()` was called. Because that is one of many functions that are new with C99, it is not yet in wide use, and the discrepancies in behavior across platforms have yet to be settled by vendors and Standards committees.

The replacements

```
feholdexcept(&env) → fegetenv(&env)
 feupdateenv(&env) → fesetenv(&env)
```

convert our code to a second test program. Similarly, the replacements

```
            fenv_t → fexcept_t
               env → flags
feholdexcept(&env) → fegetexceptflag(&flags, FE_ALL_EXCEPT)
 feupdateenv(&env) → fesetexceptflag(&flags, FE_ALL_EXCEPT)
```

produce a third program. The files `fecmplx.c`, `fecmplx2.c`, and `fecmplx3.c` in the `exp` subdirectory of the `mathcw` library distribution contain the three test programs.

5.11 Access to precision control

The remaining two functions provide access to precision control. They are not standardized in C99, but there are recommendations for them in the accompanying Rationale document, which says:

> *The IEC 60559 floating-point standard prescribes rounding precision modes (in addition to the rounding direction modes covered by the functions in this section) as a means for systems whose results are always double or extended to mimic systems that deliver results to narrower formats. An implementation of C can meet this goal in any of the following ways:*

> 1. *By supporting the evaluation method indicated by* `FLT_EVAL_METHOD` *equal to 0.*
> 2. *By providing pragmas or compile options to shorten results by rounding to IEC 60559 single or double precision.*
> 3. *By providing functions to dynamically set and get rounding precision modes which shorten results by rounding to IEC 60559 single or double precision. Recommended are functions* `fesetprec()` *and* `fegetprec()` *and macros* `FE_FLTPREC`, `FE_DBLPREC`*, and* `FE_LDBLPREC`*, analogous to the functions and macros for the rounding direction modes.*
>
> *This specification does not include a portable interface for precision control because the IEC 60559 floating-point standard is ambivalent on whether it intends for precision control to be dynamic (like the rounding direction modes) or static. Indeed, some floating-point architectures provide control modes suitable for a dynamic mechanism, and others rely on instructions to deliver single- and double-format results suitable only for a static mechanism.*

5.11.1 Precision control in hardware

As the quotation in Section 5.11 on the preceding page notes, precision control is not part of the IEEE 754 specification, but the first implementations by Intel chose to provide only 80-bit arithmetic inside the floating-point coprocessor, and eventually, directly inside the CPU. The Motorola 68000 family, and the later IA-64 architecture, widen the exponent by two bits, using an 82-bit format inside the CPU.

Those systems convert 32-bit and 64-bit values as they are loaded from external memory into CPU registers, widening the exponent field, and padding the significand with trailing zeros. All subsequent floating-point arithmetic inside the CPU is then normally done with the 80- or 82-bit format, and only on storage to memory are values converted to storage precision.

In most cases, the extra precision of intermediate computations is beneficial, but it does mean that *underflow*, *overflow*, and *subnormal* exceptions might not be raised until a store instruction is executed, and that double roundings are common, once in each numerical operation, and once in the store instruction.

The extra precision also means that comparison with computations on other machines that do all arithmetic in storage precision may be difficult. For that reason, the IA-32, IA-64, and 68000 architectures allow the programmer to set a precision-control mask to request that, on completion of each floating-point operation, the result be rounded and range reduced to either a 32-bit or a 64-bit format, instead of remaining in the default full-length format.

5.11.2 Precision control and the AMD64 architecture

The AMD64 architecture, and Intel's clone of it, EM64T, is an extension of the IA-32 architecture, widening data paths and integer registers from 32 bits to 64 bits, and adding new floating-point instructions. The latter work with sixteen directly addressable 128-bit registers and can execute four 32-bit floating-point operations, or two 64-bit operations, in a single parallel instruction. Alternatively, those registers can be used for sequential operations on 32-bit and 64-bit data. However, no data type longer than 64 bits is supported in those 128-bit registers, nor is precision control provided.

The availability of several directly addressable registers, in addition to the stack of eight 80-bit registers of the older IA-32 architecture, gives compilers more opportunities, but complicates life for the careful floating-point programmer.

Examination of code generated by several Fortran, C, and C++ compilers on AMD64 systems shows that they use the new registers for 32-bit and 64-bit floating-point data, and the old registers for 80-bit data. That means that precision control is effective only for the longest data type, such as `long double` in C and C++.

Given the lack of standardization of access to precision control, and its absence from several compilers and operating systems on platforms that have it in hardware, it is doubtful whether any major important body of existing code depends on it. Nevertheless, there *are* circumstances where it could be useful, so it makes sense to provide such access in a uniform and easy-to-use way.

5.11.3 Precision control in the mathcw library

A survey of UNIX systems current when the mathcw library was written found that only two offered access to precision control on IA-32: FREEBSD with the functions `fpgetprec()` and `fpsetprec()`, and Sun Microsystems SOLARIS 10 with the C99-style primitives. Unfortunately, the SOLARIS ones have reversed return codes: nonzero on success, and zero on failure. That is in contradiction to the other 11 routines defined in C99 for access to sticky exception flags and rounding direction modes, so our implementation follows those ISO standard routines instead: a negative

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	*x*	*x*	IC	RC		PC		*x*	*x*	PM	UM	OM	ZM	DM	IM

Figure 5.1: IA-32 floating-point control word. Numbering is from the low-order bit on the right, and reserved bits are marked with x. The one- or two-bit fields are IC (infinity control), RC (rounding control), PC (precision control), PM (precision mask), UM (underflow mask), OM (overflow mask), ZM (zero-divide mask), DM (subnormal operand mask), and IM (invalid operation mask).

return means failure, and zero or positive, success. Microsoft WINDOWS C and C++ provide access to precision control through the library function `_controlfp_s()`.

Most commonly, the return value of `fesetprec()` is ignored, and that of `fegetprec()` is used only to record the current precision prior to a temporary precision change, and then later, the recorded value is passed to `fesetprec()` to restore the original precision. The difference in failure codes for the SOLARIS versions and ours is not likely to be a serious problem, and we anticipate that the definition of the SOLARIS versions will change in a future compiler release, just as the type of several of the original C99 functions was changed from `void` to `int` to allow reporting a return code.

For the systems with IA-32 processors, two private macros, `_FPU_GETCW()` and `_FPU_SETCW()`, expand either to a single inline assembly-code instruction that gets or sets the floating-point hardware control word, or else to a short function that wraps that assembly code. Those macros are modeled on ones of the same name provided on GNU/LINUX in the platform-specific system header file `<fpu_control.h>`.

Once the control word is accessible via those two macros, the user-level code is straightforward, requiring only knowledge of the location of the precision-control field in the IA-32 control word, as illustrated in **Figure 5.1**.

```
typedef unsigned short int fpu_control_t;

static fpu_control_t cw;

int
(fegetprec)(void)
{   /* return the current rounding precision as one of the
       values FE_FLTPREC, FE_DBLPREC, or FE_LDBLPREC, or -1
       if precision control is not available */

    _FPU_GETCW(cw);

    return ((int)((cw >> 8) & 0x3));
}


int
(fesetprec)(int prec)
{   /* set the rounding precision to prec (one of FE_FLTPREC,
       FE_DBLPREC, or FE_LDBLPREC), and return that value on
       success, or -1 on failure */
    int result;

    if ((prec == FE_FLTPREC) ||
        (prec == FE_DBLPREC) ||
        (prec == FE_LDBLPREC))
    {
        _FPU_GETCW(cw);
        cw &= ~0x300;           /* clear just the PC bits */
        cw |= (prec << 8);      /* set PC bits */
        _FPU_SETCW(cw);         /* store the control word */
        result = prec;          /* success */
    }
    else
        result = FE_FAILURE;

    return (result);
}
```

The control word cw is defined at file scope, rather than inside functions, because of a compiler restriction on access to it from short inline assembly-code fragments.

The actual code in fenvx.h is more complex than that, but only because we choose to implement support for multiple operating systems in one body of code.

On architectures that lack precision control, those two functions are still provided, but they simply return FE_FAILURE (-1).

5.12 Using precision control

As an example of how precision control can be used on the IA-32 architecture, we sum to machine precision the Taylor series of the exponential function of a small argument with this program:

```
#include <stdio.h>
#include <stdlib.h>
#include <mathcw.h>
#include <fenvcw.h>

static int kmax = 0;

long double
exp_ts (long double x)
{
    int k;
    long double newsum, sum, term;

    term = 1.0L;                        /* k = 0 */
    sum = term;

    for (k = 1; k < 50; ++k)
    {
        term *= x / (long double)k;
        newsum = sum + term;

        if (newsum == sum)
        {
            kmax = k;
            break;
        }

        sum = newsum;
    }

    return (sum);
}

int
main(void)
{
    long double x, y;
    x = 1.0L / 64.0L;

    (void)printf("Computing y = exp(%Lg) with "
                 "IA-32 precision control\n", x);

    (void)fesetprec(FE_LDBLPREC);
    y = exp_ts(x);
    (void)printf("80-bit: kmax = %d  y = %.21Lg = %La\n", kmax, y, y);

    (void)fesetprec(FE_DBLPREC);
    y = exp_ts(x);
    (void)fesetprec(FE_LDBLPREC);
    (void)printf("64-bit: kmax = %d  y = %.21Lg = %La\n", kmax, y, y);

    (void)fesetprec(FE_FLTPREC);
    y = exp_ts(x);
    (void)fesetprec(FE_LDBLPREC);
    (void)printf("32-bit: kmax = %d  y = %.21Lg = %La\n", kmax, y, y);

    return (EXIT_SUCCESS);
}
```

When that program is compiled, linked, and run on a GNU/LINUX IA-32 system, the output looks like this:

```
% cc -I/usr/local/include prcctl.c && ./a.out
Computing y = exp(0.015625) with IA-32 precision control
80-bit: kmax = 9  y = 1.01574770858668574742 = 0x8.204055aaef1c8bdp-3
64-bit: kmax = 7  y = 1.01574770858668594897 = 0x8.204055aaef1dp-3
32-bit: kmax = 4  y = 1.01574766635894775391 = 0x8.20405p-3
```

The same code computes the sum in each case, but a global setting of the hardware floating-point precision affects the number of terms required, and the number of bits in the final sum. On the test system, the native run-time library trims trailing zero bits from values output with the hexadecimal format specifier, `%a`. On SOLARIS IA-32, the `%a` format is handled differently:

```
% cc -I.. prcctl.c feprec.c -L/usr/local/lib -lmcw -lm && ./a.out
prcctl.c:
feprec.c:
Computing y = exp(0.015625) with IA-32 precision control
80-bit: kmax = 9  y = 1.01574770858668574744 = 0x1.04080ab55de3917ap+0
64-bit: kmax = 7  y = 1.01574770858668594906 = 0x1.04080ab55de3a000p+0
32-bit: kmax = 4  y = 1.01574766635894775392 = 0x1.04080a0000000000p+0
```

5.13 Summary

The C99 specification of a software interface to the floating-point environment, particularly for IEEE 754 systems, is a substantial improvement over that available in older definitions or implementations of the C language. The new interface is also considerably richer than that offered by most other programming languages. It is simply unfortunate that it took so long to specify, implement, and deploy after IEEE 754 hardware came into wide use.

Although the exception and trap handling facilities are likely to remain a barrier to software portability, access to the sticky exception flags can be of significant value in the design of numerical software that can exploit fast and simple algorithms in most cases, and only rarely needs to fall back to more complex code when exception flags are found to be set.

Precision control can also be of utility, and we comment in many places in this book about the difficulties that higher intermediate precision poses for some numerical algorithms. However, programmers are advised that it is usually better to write code for a wide range of architectures, and instead use the `volatile` and `STORE()` techniques to restrict computational precision.

Rounding control is essential in some applications, such as interval arithmetic. It is also useful for making numerical experiments to determine whether a program's final output is particularly sensitive to rounding directions. If it is, then the program's reliability is doubtful. Iterative processes that rely on numerical convergence tests may misbehave when a nondefault rounding mode is selected, so rounding control is a useful addition to the software-test toolbox.

6 Converting floating-point values to integers

> I HAVE FOUND A BUG ON `QEMU` SPARC:
> THE FLOATING POINT TO INTEGER CONVERSION INSTRUCTIONS
> (`fstoi`, `fdtoi`, `fstox`, `fdtox`) ARE NOT CORRECTLY EMULATED.
>
> — AURELIEN JARNO
> BUG REPORT TO `Qemu-devel` MAILING LIST (2007).

This chapter treats the important subject of converting arbitrary floating-point numbers to whole numbers, where the results may be in either floating-point or integer data formats. A few important historical machines targeted at the scientific computing market, and described in **Appendix H** on page 947, make such conversions easy by virtue of not having a separate integer storage format. Integers are then just floating-point values with a zero exponent, and conversions may require little more than bit shifting, and possibly, rounding. Some scripting languages provide only numbers and strings, where all numbers are represented as floating-point values.

Conversions between numeric data types are common in computer software, and many programming languages allow implicit conversions in assignments, expressions, and in passing arguments to other routines. Computer languages also generally offer explicit conversions through calls to built-in or user-defined functions. The C language and its descendants allow conversions with type casts, such as `(int)x`. Another style in languages with classes is `x.int()`, meaning to apply the method function `int()` from the class of `x` to `x` itself.

Modern computer hardware provides substantial support for numeric conversions, often doing so with single machine instructions. Although it might appear that the conversion problem is both well understood, and fully supported, by hardware and software, we show in this chapter that this is far from the case. Almost all languages and machines are severely lacking in numeric-conversion facilities, and too many of them fail to provide *safe* conversions.

6.1 Integer conversion in programming languages

The 1999 ISO C Standard defines 33 functions in seven different groups for converting floating-point values to integers. There are several reasons for that diversity and richness:

- Integer-valued results may be needed in any of three floating-point types (`float`, `double`, and `long double`), or converted to either of two integer data types (`long int` and `long long int`).
- Conversions can be according to the current rounding mode (for example, any of the four standard IEEE 754 rounding directions), or can be defined to be independent of the rounding mode.
- Applications that need such conversions may want the nearest integer, or the next higher integer, or the next lower integer, or the closest integer in the direction of zero.
- Although the conversions are simple to describe, they are not simple to program correctly.
- Their addition remedies a serious deficiency in the run-time library defined by the earlier 1990 ISO C Standard, which, apart from type casts, requires only three such functions: `ceil()`, `floor()`, and `modf()`.

The original 1956 definition of Fortran includes only `intf(x)` to produce the largest floating-point whole number whose magnitude does not exceed x, and has the sign of x, `xintf(x)` that behaves the same way, but returns a result of `INTEGER` type, and `modf(x,y)` to compute $x - \mathrm{intf}(x/y) \times y$.

Fortran 66 renames `intf(x)` to `aint(x)`, `xintf(x)` to `int(x)`, and `modf(x,y)` to `amod(x,y)`. It also introduces `ifix(x)` as a synonym for `int(x)`, and `idint(x)` to truncate a `DOUBLE PRECISION` value to an `INTEGER`.

Fortran 77 adds `dint(x)` to truncate a `DOUBLE PRECISION` value to a whole number in floating-point format, and functions `anint(x)`, `dnint(x)`, `nint(x)`, and `idnint(x)` to convert to the nearest integer. Halfway cases round away from zero, so `nint(`± 2.5`)` produces ± 3.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_6

Fortran 90 adds `ceiling(x)`, `floor(x)`, and `modulo(x,y)`, where the latter computes $x - \lfloor x/y \rfloor \times y$. The Fortran 2003 and Fortran 2008 Standards leave the conversion facilities unchanged.

Pascal provides only `trunc(x)` and `round(x)`, equivalent to Fortran's `int(x)` and `nint(x)`.

Common Lisp supplies conversion functions `(ceiling ...)`, `(fceiling ...)`, `(ffloor ...)`, `(floor ...)`, `(fround ...)`, `(ftruncate ...)`, `(mod ...)`, `(rem ...)`, `(round ...)`, and `(truncate ...)`. For example, the call `(round` ±2.5`)` returns ±2, and `(round` ±3.5`)` produces ±4, because that function uses a *round-to-nearest-ties-to-even* rule to reduce rounding bias.

6.2 Programming issues for conversions to integers

Three important considerations guide our software design:

- The range and precision of floating-point data are often wider than that provided by integer data types, so overflows must be dealt with if the conversion must yield an integer type.
- A floating-point number, x, can always be split *exactly* into the sum of a whole number and a fraction, $x = w + f$, where both w and f have the same sign as x, and $|f| < 1$. All of the remaining conversions can be described relatively easily in terms of that single operation.
- The ISO C Standard specifies that the treatment of overflows in such conversions is implementation dependent. We address such overflows in our code for the functions returning integer data types by setting the global `errno` value to `ERANGE` (result out of range for the target data type), and by returning an integer of the appropriate sign and largest representable magnitude for that sign. The constants `INT_MIN`, `INT_MAX`, `LONG_MIN`, `LONG_MAX`, `LLONG_MIN`, and `LLONG_MAX` in the C Standard header file, `<limits.h>`, provide the needed values.

In the following sections, we show prototypes for the 33 functions, but we omit header files. In each case, it should be understood that the prototypes must be provided by either of these preprocessor statements:

```
#include <math.h>

#include <mathcw.h>
```

The second header file is effectively a superset of the first, because the `mathcw` library offers more functions than the C99 math library does.

Similarly, in the implementations of the conversion functions, we do not show header files, but it should be understood that each of them includes a *single* file with a lowercase name related to the function name of type `double`, shortened to six characters in the base name if necessary to match the package design rules described on page 3. For example, the file `modfx.h` looks something like this:

```
#if !defined(MODFX_H)
#define MODFX_H

#include "modf.h"

fp_t
MODF(fp_t x, fp_t *y)
{
    /* ... code omitted ... */
}
#endif /* !defined(MODFX_H) */
```

Its single header file in turn includes a minimal set of other required header files. For that particular function, the header file `modf.h` looks like this:

```
#if !defined(MODF_H)
#define MODF_H

#define HIDE_MATHCW_GENERICS
```

Table 6.1: Out-of-range conversions to integers measured with type casts in C. The 64-bit `long long int` type on several operating systems is handled by software arithmetic. For the 36-bit PDP-10, the input values are increased by 2^4.

Processor	**O/S**	`(short)`($\pm 2^{18}$)	`(int)`($\pm 2^{34}$)	`(long)`($\pm 2^{34}$)	`(long long)`($\pm 2^{66}$)
AMD64	GNU/Linux	+0	−2147483648	±17179869184	−9223372036854775808
Alpha	GNU/Linux	+0	+0	±17179869184	+0
Alpha	OSF/1	+0	+0	±17179869184	+0
IA-32	FreeBSD	−32768	−2147483648	−2147483648	−9223372036854775808
IA-32	GNU/Linux	−32768	−2147483648	−2147483648	−9223372036854775808
IA-32	NetBSD	+0	−2147483648	−2147483648	−9223372036854775808
IA-32	OpenBSD	+0	−2147483648	−2147483648	−9223372036854775808
IA-32	Solaris	+0	−2147483648	−2147483648	−9223372036854775808
IA-64	GNU/Linux	+0	+0	±17179869184	−9223372036854775808
IA-64	HP-UX	+0	+0	+0	−9223372036854775808
Interdata	Unix V6	±262144	±2147483647	±2147483647	n/a
MC68040	Mach	+32767	+2147483647	+2147483647	n/a
		−32768	−2147483648	−2147483648	n/a
MIPS R4400SC	GNU/Linux	+0	+2147483647	+2147483647	∓1
MIPS R10000	IRIX	+0	+2147483647	+2147483647	+9223372036854775807
PA-RISC	HP-UX	+0	+2147483647, −2147483648	+2147483647, −2147483648	+9223372036854775807, −9223372036854775808
PDP-10	TOPS-20	+0	+0	+0	n/a
PowerPC	GNU/Linux	+0	+2147483647, −2147483648	+2147483647, −2147483648	∓1
PowerPC	Mac OS X	+0	+2147483647, −2147483648	+2147483647, −2147483648	∓1
SPARC	GNU/Linux	+0	+2147483647, −2147483648	+2147483647 −2147483648	∓1
SPARC	Solaris	+0	+2147483647, −2147483648	+2147483647, −2147483648	+9223372036854775807, −9223372036854775808
VAX	NetBSD	+0	+0	+0	+0

```
#include "mathcw.h"
#include "prec.h"
#include "store.h"
#include <stddef.h>

#endif /* !defined(MODF_H) */
```

The outer conditionals protect against multiple inclusions of the same header file.

6.3 Hardware out-of-range conversions

Before we begin a description of the implementation of the C99 library support for converting floating-point values to integers, a topic that occupies several following sections, it is worthwhile to investigate how hardware architects have chosen to handle overflows in such conversions. Hardware conventions can sometimes guide software design, but as we shall see in this case, *there are none*. Indeed, experiments show that there is surprising variety in how common hardware platforms handle type conversions when the result is too large for the target data type. Table 6.1 records results from several systems.

Table 6.2: Rounding to integer values with the C99 `rint()` function, which rounds its argument to a floating-point whole number according to the current rounding direction.
Although we show negative-zero results, their signs are lost when they are converted to a two's-complement integer type, such as with the `lrint()` function.

rounding	**number to convert**									
direction	−3.9	−3.1	−1.5	−0.5	−0.0	+0.0	+0.5	+1.5	+3.1	+3.9
downward	−4	−4	−2	−1	−0	0	0	1	3	3
to nearest	−4	−3	−2	−0	−0	0	0	2	3	4
toward zero	−3	−3	−1	−0	−0	0	0	1	3	3
upward	−3	−3	−1	−0	−0	0	1	2	4	4

The chief lesson from the tabulated results is that out-of-range conversions are handled inconsistently across CPUs, across operating systems, and even across data types. The presence of many zeros and ones in the table shows that one cannot rely on the result having a large magnitude, nor can one even expect to preserve the original sign!

Using type casts to convert floating-point values to integer values is clearly risky, and should not be done unless the range of the floating-point values is guaranteed to be small enough to avoid overflow in the conversion.

A much safer approach is to use the library functions described in the following sections, and for those that return integer types, to check the value of `errno` after each such function call, and take remedial action if an out-of-range conversion is detected. However, as we discuss in **Section 4.23** on page 95 and **Section 5.3** on page 106, it is worth recalling that `errno` may be a single global value, so it may not be useful in multithreaded applications: a particular execution thread cannot tell if `errno` was set by the thread itself, or by another thread. Fortunately, many systems now define that value to be thread private.

6.4 Rounding modes and integer conversions

The IEEE 754 rounding modes affect the conversion of floating-point numbers to values of integer types in possibly surprising ways. They can be understood if we recall the purpose of rounding. When we have an exact result that is not representable in the target data type, we must choose one of the two representable values between which the exact result lies, *making that choice according to the current rounding direction*.

Table 6.2 illustrates the actions taken for each of the four directions for several input values. Notice in particular the rounding of the values ±0.5 and ±1.5 in the case of the default *to nearest* mode, which breaks such ties by choosing the even result.

6.5 Extracting integral and fractional parts

The functions

```
float       modff (float x,       float *y);
double      modf  (double x,      double *y);
long double modfl (long double x, long double *y);
```

carry out the split of x into the sum of a whole number and a fraction, $x = w + f$, with w and f having the same sign as x, and $|f| < 1$, as described on page 130. They return the fraction f as the function value, and store the whole number w in the location defined by the second argument. That location has the same type as x, so there is no possibility of overflow or underflow.

The ISO C Standards do not address the issue of whether the second argument can be a `NULL` pointer. Our code checks for that case, and avoids storage of the whole-number value.

Table 6.3: Fixed-point representation of large integers.

β^t	$\mathtt{10000\cdots0000}_\beta$		
$\beta^t - 1$	$\mathtt{dddd\cdots dddd}_\beta$	$(\mathtt{d} = \beta - 1)$	
β^{t-1}	$\mathtt{1000\cdots0000}_\beta$		
$\beta^{t-1} - (1/\beta)$	$\mathtt{ddd\cdots dddd.d}_\beta$		
β^{t-2}	$\mathtt{100\cdots0000.0}_\beta$		
β^{t-3}	$\mathtt{10\cdots0000.00}_\beta$		
$\beta^{t-1} + x$	$\mathtt{1yyy\cdots yyyy.yyy\cdots yyy}_\beta$	$(0 \leq \mathtt{x} < \beta^{t-1})$	*(exact)*
$\beta^{t-1} + x$	$\mathtt{1yyy\cdots yyyy}_\beta$	$(0 \leq \mathtt{x} < \beta^{t-1})$	*(truncated)*
$(\beta^{t-1} + x) - \beta^{t-1}$	$\mathtt{0yyy\cdots yyyy}_\beta$	*(stored)*	

Appendix F of the 1999 ISO C Standard describes the behavior of the language when the environment supports the IEC 60559:1989 Standard, *Binary Floating-Point Arithmetic for Microprocessor Systems*, Second Edition. That Standard is the international version of IEEE 754. For the `modf()` family, Appendix F of the C99 Standard requires that when the first argument is a NaN, then both whole number and fraction are NaNs. When the first argument is $\pm\infty$, the fraction returned as the function value is ± 0, and the whole-number location is set to $\pm\infty$.

The GNU C mathematical library (glibc), Moshier's Cephes library [Mos89], Plauger's Standard C library [Pla92], and the Sun Microsystems fdlibm library all use low-level bit manipulation to split the argument into fraction and exponent, and then use tests against various magic constants that, after bit shifts and arithmetic, eventually lead to the desired result. For related conversion functions, the glibc and fdlibm code also take care to raise the *inexact* exception flag when appropriate.

Unfortunately, the floating-point-to-integer conversion code in those four libraries is messy, and certainly not portable between floating-point precisions, or to older systems with non-IEEE 754 arithmetic. Even though we observed in the introduction to this chapter that all of the conversions can be straightforwardly handled, once the functions in the `modf()` family are available, those libraries have complex code for all of the conversion functions.

The mathcw library is intended to be usable on a wide range of architectures, and we need a much cleaner, and much more portable, approach. Fortunately, that is possible, as we now show. Even better, our code is certainly shorter than that in those other libraries, and thus, might even be faster, especially if the Infinity and NaN tests can be compiled into efficient inline code.

In a fixed-point arithmetic system with n fractional digits, we could isolate the integer part by shifting right by n digits to discard the fractional digits, then shifting left by n digits, supplying zero fractional digits. We can accomplish the same thing in a floating-point system in a different way, by adding and subtracting a suitable large number. The problem is now to find that special number.

Recall that in any unsigned arithmetic system with base β and t digits of precision, the largest exactly representable whole number is $\beta^t - 1$. However, in a floating-point system with a sufficiently wide exponent range, the next larger whole number, β^t, is also exactly representable, because it can be stored with a one-digit significand of one, and an exponent of t, even though when written as an integer value, it really requires $t + 1$ digits. **Table 6.3** summarizes the digit sequences in those two values, and some related numbers.

Thus, in a floating-point arithmetic system, values β^{t-1} and larger have no fractional part, and are therefore whole numbers. The `modf()` family functions for such values then can simply return zero, after storing their first argument in the location of their second argument. Otherwise, for any finite $|x| < \beta^{t-1}$, adding and subtracting β^{t-1} in *round-to-zero* (truncating) arithmetic removes fractional digits, as shown in the bottom part of **Table 6.3**. The adjustment operation must be done in storage precision, as discussed earlier in **Section 4.5** on page 65.

Access to rounding-mode control is slow on some platforms, and nonexistent on others, particularly pre-IEEE 754 architectures. However, we do not need to manipulate rounding modes at all.

Consider what happens if the rounding mode is set to something other than *round-to-zero* (truncate). The intermediate value, $\beta^{t-1} + x$, is then off by one about half the time: one too large for positive arguments, and one too small for negative arguments. The result is a whole number, and in neither case can it differ from the mathematically exact result by more than one. The subsequent subtraction to form $(\beta^{t-1} + x) - \beta^{t-1}$ involves only whole numbers. It is therefore exact, and immune to rounding modes. If we can compensate for the unwanted rounding of $\beta^{t-1} + x$, then we can recover the exact fraction by another subtraction.

The unwanted rounding is easily detected: for positive x, it happens when the sum exceeds x. The sum can then simply be reduced by one, an exact operation, because the sum is a whole number.

Here is what our code looks like:

```
fp_t
MODF(fp_t x, fp_t *y)
{   /* x = w + f, |f| < 1, w and f have sign of x */
    fp_t f;
    volatile fp_t w;

    if (ISINF(x))
    {
        w = SET_ERANGE(x);
        f = COPYSIGN(ZERO, x);
    }
    else if (ISNAN(x))
        w = f = SET_EDOM(QNAN(""));
    else
    {
        fp_t xabs;

        xabs = QABS(x);

        if (xabs == ZERO)
            w = f = ZERO;
        else if (xabs >= B_TO_T_MINUS_ONE)
        {
            w = xabs;
            f = ZERO;
        }
        else
        {
            w = B_TO_T_MINUS_ONE + xabs;
            STORE(&w);
            w -= B_TO_T_MINUS_ONE;
            STORE(&w);

            if (w > xabs)       /* rounding occurred */
            {
                --w;
                STORE(&w);
            }

            f = xabs - w;       /* NB: exact! */
        }

        if (SIGNBIT(x))
        {
            w = -w;
            f = -f;
        }
    }

    if (y != (fp_t *)NULL)
        *y = w;
    return (f);
}
```

We hide the data-type dependencies in the compile-time type `fp_t` and the uppercase macro names. The only architecture dependence is the value of the compile-time constant `B_TO_T_MINUS_ONE` ($= \beta^{t-1}$). That value is, alas, not determinable by a legal compile-time expression using other symbolic constants defined in the header file `<float.h>`. Subsidiary header files included by our private header file, `prec.h`, provide a definition, because it is needed in other library functions as well.

There are several points to note about that code:

- The result of the computation of $(\beta^{t-1} + x)$ *must* be in storage precision, a topic addressed earlier in **Section 4.5** on page 65. For architectures with higher-than-storage-precision registers with old compilers that do not implement the C89 `volatile` keyword, or fail to implement it correctly, the `STORE()` function call ensures storage of its argument. On modern systems, it can be a macro that does nothing.

- Some compilers on systems with IEEE 754 arithmetic are sloppy about the handling of negative zero. In the code for a first argument of Infinity, we used `COPYSIGN()` to ensure the correct sign of a zero result.

- The only IEEE 754 sticky exception flags that can be set are *invalid*, in the event that the first argument is a NaN, and *inexact*, when the first argument is finite and its fraction is nonzero.

  The *invalid* flag is intentionally set by the `QNAN()` function. Although a simple assignment `w = f = x` would supply the required NaN results, it would not set the exception flag.

  The 1999 ISO C Standard does not specify whether exception flags can be set by the `modf()` function family, but *inexact* is commonly set by most floating-point operations. It is therefore unlikely that any realistic code that uses *any* floating-point library function would depend on *inexact* not being set by the library code.

- Special treatment is required for two different cases with zero fractions: the first argument is a positive or negative zero, or its magnitude is at least β^{t-1}.

  In the first case, we must set both whole number and fraction to zero. Otherwise, when the rounding mode is set to *downward*, if we fell through to use the code in the final `else` block that computes $w = (\beta^{t-1} + |x|) - \beta^{t-1}$, we would get $w = -0$ instead of the correct $w = +0$. That subtle point was initially overlooked when our code was developed, but the error was caught in testing.

  In the second case, we can compute the almost-final results by simple assignment to f and w.

- Once the obligatory tests for Infinity, NaN, and a zero-fraction result are handled, the hot spot in the code is the `else` block that computes $(\beta^{t-1} + x) - \beta^{t-1}$. It requires on average just 3.5 additions, 4.5 loads, 2.5 stores, one comparison, and 0.5 branches.

- We need to use `SIGNBIT()`, rather than a simple test, `x < ZERO`, to correctly handle the case where the first argument is a negative zero.

For IEEE 754 single-precision computation, it is feasible to make an exhaustive test that compares our implementation with the native `modff()` library code for the 350 million or so possible arguments that have nonzero integer parts and nonzero fractions: no differences are found. A relative timing test showed that our code ran 0.7 to 3.0 times slower than the native version on a dozen systems with different CPU architectures or operating systems, and many different compilers.

6.6 Truncation functions

The truncation functions

```
float       truncf (float x);
double      trunc  (double x);
long double truncl (long double x);
```

convert their argument to the nearest whole number whose magnitude does not exceed that of the argument, and return that whole number as a floating-point value. Thus, $\mathrm{trunc}(-3.5) \to -3.0$ and $\mathrm{trunc}(3.9) \to 3.0$. Their values always have the same sign as their arguments, so $\mathrm{trunc}(\pm 0.0) \to \pm 0.0$.

The `modf()` function extracts the whole number, and the rest of the code is just a simple wrapper:

```
fp_t
TRUNC(fp_t x)
{
    fp_t w;

    (void)MODF(x, &w);

    return (w);
}
```

The only IEEE 754 floating-point sticky exception flags that can be set are those set by `modf()`: *inexact* if `x` has a nonzero fraction, and *invalid* if the argument is a NaN.

The `errno` global value is never set.

6.7 Ceiling and floor functions

The *ceiling* of a fractional number is defined to be the nearest whole number that is not less than that number. Similarly, the *floor* of a fractional number is the nearest whole number that is not greater than that number. The mathematical notations for those quantities are $\lceil x \rceil$ for the ceiling, and $\lfloor x \rfloor$ for the floor.

The functions defined by 1999 ISO Standard C for those operations are:

```
float       ceilf (float x);
double      ceil  (double x);
long double ceill (long double x);


float       floorf (float x);
double      floor  (double x);
long double floorl (long double x);
```

Notice that the functions return floating-point results, *not* integer values. For example, $\mathrm{ceil}(-3.4) \to -3.0$, $\mathrm{floor}(-3.4) \to -4.0$, and $\mathrm{ceil}(3.0) = \mathrm{floor}(3.0) \to 3.0$.

The two functions satisfy the simple relation $\mathrm{floor}(x) = -\mathrm{ceil}(-x)$, and their values always have the same sign as their arguments, so $\mathrm{ceil}(\pm 0.0) = \mathrm{floor}(\pm 0.0) \to \pm 0.0$. They are related to the truncation function as follows: $\mathrm{trunc}(|x|) = \mathrm{floor}(|x|)$ and $\mathrm{trunc}(-|x|) = \mathrm{ceil}(-|x|)$.

The `modf()` function allows retrieval of the whole number, and a simple adjustment of that value provides the ceiling or floor:

```
fp_t
CEIL(fp_t x)
{
    fp_t w;

    (void)MODF(x, &w);

    return ((w < x) ? ++w : w);
}


fp_t
FLOOR(fp_t x)
{
    fp_t w;
    (void)MODF(x, &w);

    return ((w > x) ? --w : w);
}


fp_t
FLOOR(fp_t x)
{
    return (-CEIL(-x));
}
```

We showed two implementations of the floor function; the second is shorter, but requires one extra function call.

In IEEE 754 arithmetic, comparisons are exact, and neither overflow nor underflow. They may set the *invalid* flag if an operand is a NaN, but that cannot happen in our code unless the argument is a NaN. The addition and subtraction in the adjustments of `w` are exact, and can neither overflow nor underflow. Thus, the only IEEE 754 floating-point sticky exception flags that can be set are the same as those set by `modf()`: *inexact* if the argument has a nonzero fraction, and *invalid* if the argument is a NaN.

The `errno` global value is never set.

6.8 Floating-point rounding functions with fixed rounding

The rounding functions

```
float       roundf (float x);
double      round  (double x);
long double roundl (long double x);
```

convert their argument to the *nearest* whole number and return that whole number as a floating-point value. Halfway cases are rounded away from zero, independent of the current rounding direction. For example, $\mathrm{round}(-3.4) \to -3.0$, $\mathrm{round}(-3.5) \to -4.0$, and $\mathrm{round}(3.5) \to 4.0$, Their values always have the same sign as their arguments, so $\mathrm{round}(\pm 0.0) \to \pm 0.0$.

The `modf()` function allows retrieval of the whole number, and only a little extra work is needed to complete the job:

```
fp_t
ROUND(fp_t x)
{
    fp_t f, w;

    f = MODF(x, &w);

    if (f <= -HALF)
        --w;
    else if (f >= HALF)
        ++w;

    return (w);
}
```

We do not require special handling of Infinity and NaN here, because `modf()` does the necessary work. An argument of Infinity results in a zero fraction, neither conditional succeeds, and the result is Infinity. A NaN argument results in NaN values for both f and w, and even if the conditionals succeed, as they do with some defective compilers, w remains a NaN.

The only IEEE 754 floating-point sticky exception flags that can be set are those set by `modf()`: *inexact* if the argument has a nonzero fraction, and *invalid* if the argument is a NaN.

The `errno` global value is never set.

6.9 Floating-point rounding functions with current rounding

The rounding functions

```
float       rintf (float x);
double      rint  (double x);
long double rintl (long double x);
```

convert their argument to a whole number using the current rounding direction, and return that whole number. For example, $\mathrm{rint}(-3.1) \to -4.0$ if the rounding direction is *downward*, and $\mathrm{rint}(-3.1) \to -3.0$ if the rounding direction is *toward zero*, *upward*, or *to nearest*. More examples are given in **Table 6.2** on page 132. Their values always have the same sign as their arguments, so $\mathrm{rint}(\pm 0.0) \to \pm 0.0$.

The `modf()` function allows retrieval of the whole number, but more work is needed to handle the rounding properly. The brute-force approach would fetch the current rounding mode, and then take different actions for each of the four possible modes. However, that is complex, and rounding-mode access is slow on some architectures.

A better approach is to let the floating-point hardware do the job for us. Recall that β^{t-1} is the start of the floating-point range where there are no fractional parts. If we add the fraction returned by `modf()` to β^{t-1}, the addition takes place according to the current rounding mode, and we can then test the result to see whether the integer part needs adjustment by comparing the sum to β^{t-1}. If the sum is larger, increase the result by one, and if smaller, decrease the result by one.

Unfortunately, that algorithm is not quite right. It fails for the IEEE 754 default *to nearest* rounding mode, which rounds to an even result when the value is exactly halfway between two representable results. For example, as suggested by **Table 6.2** on page 132, $\mathrm{rint}(0.5) \to 0.0$, $\mathrm{rint}(1.5) \to 2.0$, $\mathrm{rint}(2.5) \to 2.0$, and $\mathrm{rint}(3.5) \to 4.0$. For the split $x = w + f$ returned by `modf()`, because β^{t-1} is even for the number bases that we find in practice (2, 4, 8, 10, and 16), we can use $\beta^{t-1} + f$ if w is even, but when w is odd, we must use $\beta^{t-1} + 1 + f$.

Here is our code to compute `rint()`:

```
fp_t
RINT(fp_t x)
{
    fp_t delta, f, w, w_half;
    volatile fp_t sum;

    f = MODF(x, &w);
    (void)MODF(w * HALF, &w_half);

    /* Ensure that delta has same parity as w to preserve
       rounding behavior for to-nearest mode. */
    if (w == (w_half + w_half))            /* w is even */
        delta = B_TO_T_MINUS_ONE;          /* delta is even */
    else                                   /* w is odd */
        delta = B_TO_T_MINUS_ONE + ONE;    /* delta is odd */

    if (x >= ZERO)
    {
        sum = delta + f;
        STORE(&sum);

        if (sum > delta)
            ++w;
        else if (sum < delta)
            --w;
    }
    else
    {
        sum = -delta + f;
        STORE(&sum);
        if (sum > -delta)
            ++w;
        else if (sum < -delta)
            --w;
    }

    return (w);
}
```

Overflow and underflow are impossible in the computation of `sum`, and for large magnitude arguments $|x| >= \beta^{t-1}$, `modf()` returns $w = x$ and $f = \pm 0$, so no further adjustments are made on the result variable.

We handle positive and negative values separately to ensure that the magnitude of `sum` in exact arithmetic is at least as large as δ, forcing rounding to occur when the fraction is nonzero. It is not practical to merge the two branches of the `if` statement, because three of the four IEEE 754 rounding directions are not symmetric about zero, and our code must work correctly in all of them.

No special code is needed to handle the case of negative zero. It is taken care of by the first part of the `if` statement, where `modf()` sets $w = -0$, and `sum` gets the value `delta`. No further changes are made to w, so the returned value has the correct sign.

The `sum` variable must be evaluated in storage precision. The `volatile` qualifier, or the `STORE()` function (or macro), ensures that.

We do not require special handling of Infinity and NaN here, for the same reasons cited in **Section 6.8** on page 137 for `round()`.

The only IEEE 754 floating-point sticky exception flags that can be set are those set by `modf()`: *inexact* if the argument has a nonzero fraction, and *invalid* if the argument is a NaN.

The `errno` global value is never set.

6.10 Floating-point rounding functions without *inexact* exception

The 1999 ISO C Standard defines a set of related functions

```
float       nearbyintf (float x);
double      nearbyint  (double x);
long double nearbyintl (long double x);
```

which differ from those in the `rint()` function family only in that they do not raise the *inexact* exception flag. One way to prevent that from happening is to use only low-level integer operations to construct the result, but as we observed in **Section 6.5** on page 132, that is messy and nonportable.

Use of the `nearbyint()` functions is likely to be uncommon, and we can therefore afford to sacrifice some speed in favor of clarity and simplicity. We choose to implement them by saving the value of the *inexact* flag, doing the computation, and then clearing the flag if it was set in the computation, but was not already set before.

```
fp_t
NEARBYINT(fp_t x)
{
    fp_t result;
    int new_flags, old_flags;

    old_flags = fetestexcept(FE_INEXACT);
    result = RINT(x);
    new_flags = fetestexcept(FE_INEXACT);

    if ( ((old_flags & FE_INEXACT) == 0) && ((new_flags & FE_INEXACT) != 0) )
        (void)feclearexcept(FE_INEXACT);

    return (result);
}
```

Alternatively, we can save the flag, do the computation, and then clear the flag if it was clear on entry:

```
fp_t
NEARBYINT(fp_t x)
{
    fp_t result;
    int old_flags;

    old_flags = fetestexcept(FE_INEXACT);
    result = RINT(x);

    if ((old_flags & FE_INEXACT) == 0)
        (void)feclearexcept(FE_INEXACT);

    return (result);
}
```

No special handling of either Infinity or NaN arguments is required, because `rint()` does that for us.

Because the *inexact* flag is set on most floating-point operations in practical computations, it is likely to be already set on entry, and thus, only rarely needs to be cleared before return. The extra overhead compared to `rint()` is then mostly that of two calls to `fetestexcept()` in the first version, or one call to each of the exception routines in the second version. That version is shorter, and is probably slightly faster because of the simpler `if`-expression.

The only IEEE 754 floating-point sticky exception flag that can be set by the `nearbyint()` function family is *invalid*, and that happens only when the argument is a NaN.

The `errno` global value is never set.

6.11 Integer rounding functions with fixed rounding

The rounding functions

```
long int lroundf (float x);
long int lround  (double x);
long int lroundl (long double x);
```

convert their argument to the nearest whole number and return that whole number as a `long int` value. Halfway cases are rounded away from zero, independent of the current rounding direction. For example, $\mathrm{lround}(-3.4) \to -3$, $\mathrm{lround}(-3.5) \to -4$, and $\mathrm{lround}(3.5) \to 4$. Because two's-complement integer arithmetic has only one zero, the sign of a zero argument is not preserved: $\mathrm{lround}(\pm 0.0) \to +0$.

The companion functions

```
long long int llroundf (float x);
long long int llround  (double x);
long long int llroundl (long double x);
```

behave the same away, except for the difference in the type of the returned value.

Those functions behave like those in the `round()` family, except that the final result requires conversion to an integer type. Because that conversion involves a range reduction, the 1999 ISO C Standard says:

> *If the rounded value is outside the range of the return type, the numeric result is unspecified. A range error may occur if the magnitude of x is too large.*

The most reasonable approach for out-of-range results is to return the closest representable integer, and set `errno`. Here is our code:

```
long int
LROUND(fp_t x)
{
    long int result;
    if (ISNAN(x))
        result = SET_INVALID(EDOM, LONG_MIN);
    else
    {
        fp_t w;

        w = ROUND(x);

        if (w < (fp_t)LONG_MIN)
            result = SET_INVALID(ERANGE, LONG_MIN);
        else if (w > (fp_t)LONG_MAX)
            result = SET_INVALID(ERANGE, LONG_MAX);
        else
            result = (long int)w;
    }
    return (result);
}


long long int
LLROUND(fp_t x)
{
    long long int result;

    if (ISNAN(x))
        result = SET_INVALID(EDOM, LLONG_MIN);
    else
    {
        fp_t w;

        w = ROUND(x);

        if (w < (fp_t)LLONG_MIN)
            result = SET_INVALID(ERANGE, LLONG_MIN);
        else if (w > (fp_t)LLONG_MAX)
            result = SET_INVALID(ERANGE, LLONG_MAX);
        else
            result = (long long int)w;
    }
    return (result);
}
```

The tests for a NaN argument are essential here, because the result of converting a NaN to an integer value by a type cast is platform dependent. We choose to return the most negative integer and set `errno` to `EDOM` (argument is out of the domain of the function), to distinguish the result from a finite value that is too large to represent, and for which `errno` is set to `ERANGE`.

The convenience macro `SET_INVALID()` sets the *invalid* exception flag, as required by Appendix F of the 1999 ISO C Standard. That could be done by calling the exception library routine `feraiseexcept(FE_INVALID)`, but in practice, it is faster to simply generate a quiet NaN by calling `QNAN("")`. The macro also sets `errno` to its first argument, and returns its second argument. The C-language comma expression makes that easy, and the net effect is that the first assignment in `LROUND()` looks like this after macro expansion:

```
result = ((void)QNAN(""), errno = EDOM, LONG_MIN);
```

The Standard is silent about the return values from those functions for a NaN argument. Our choice seems to be the best that one can do, because traditional integer arithmetic on computers lacks the concept of *not-a-number*, and as we noted in **Section 4.10** on page 72, exceptions or traps on integer overflow are rarely possible.

The only IEEE 754 floating-point sticky exception flags that can be set are *inexact* if the argument has a nonzero fraction, and *invalid* if the argument is a NaN, or else the result is too big to represent in a value of the integer return type.

The errno global value may be set to EDOM or ERANGE.

6.12 Integer rounding functions with current rounding

The rounding functions

```
long int lrintf (float x);
long int lrint  (double x);
long int lrintl (long double x);
```

convert their argument to the 'nearest' whole number according to the current rounding direction, and return that whole number. For example, $\text{lrint}(-3.1) \rightarrow -4$ if the rounding direction is *downward*, and $\text{lrint}(-3.1) \rightarrow -3$ if the rounding direction is *toward zero*, *upward*, or *to nearest*. Because two's-complement integer arithmetic has only one zero, the sign of a zero argument is not preserved: $\text{lrint}(\pm 0.0) \rightarrow +0$.

The companion functions

```
long long int llrintf (float x);
long long int llrint  (double x);
long long int llrintl (long double x);
```

are similar, differing only in the type of the returned value.

Those functions behave like those in the rint() family, except that the final result requires conversion to an integer type. Because that conversion involves a range reduction, the 1999 ISO C Standard says:

> *If the rounded value is outside the range of the return type, the numeric result is unspecified. A range error may occur if the magnitude of* x *is too large.*

Appendix F of the 1999 ISO C Standard places a further requirement on those functions:

> *If the rounded value is outside the range of the return type, the numeric result is unspecified and the* invalid *floating-point exception is raised. When they raise no other floating-point exception and the result differs from the argument, they raise the* inexact *floating-point exception.*

The most reasonable approach for out-of-range results is to return the closest representable integer, and set errno and the exception flags. Here is our code:

```
long int
LRINT(fp_t x)
{
    long int result;

    if (ISNAN(x))
        result = SET_INVALID(EDOM, LONG_MIN);
    else
    {
        fp_t w;

        w = RINT(x);

        if (w < (fp_t)LONG_MIN)
            result = SET_INVALID(ERANGE, LONG_MIN);
        else if (w > (fp_t)LONG_MAX)
            result = SET_INVALID(ERANGE, LONG_MAX);
        else
            result = (long int)w;
    }
    return (result);
}


long long int
LLRINT(fp_t x)
{
    long long int result;

    if (ISNAN(x))
        result = SET_INVALID(EDOM, LLONG_MIN);
    else
    {
        fp_t w;

        w = RINT(x);

        if (w < (fp_t)LLONG_MIN)
            result = SET_INVALID(ERANGE, LLONG_MIN);
        else if (w > (fp_t)LLONG_MAX)
            result = SET_INVALID(ERANGE, LLONG_MAX);
        else
            result = (long long int)w;
    }

    return (result);
}
```

As with the `lround()` function family, we need to handle NaN arguments explicitly, to avoid platform dependence in the conversion of NaNs to integers.

The only IEEE 754 floating-point sticky exception flags that can be set are *inexact* if the argument has a nonzero fraction, and *invalid* if the argument is a NaN, or else the result is too big to represent in a value of the integer return type.

The `errno` global value may be set to `EDOM` or `ERANGE`.

6.13 Remainder

The 1999 ISO C Standard defines three families of functions related to the remainder in division:

```
float       fmodf (float x, float y);
double      fmod  (double x, double y);
long double fmodl (long double x, long double y);


float       remainderf (float x, float y);
double      remainder  (double x, double y);
long double remainderl (long double x, long double y);


float       remquof (float x, float y, int *quo);
double      remquo  (double x, double y, int *quo);
long double remquol (long double x, long double y, int *quo);
```

Table 6.4 on the following page shows some typical results of those functions.

All of the remainder functions produce *exact* results. Their use is critical in the argument-reduction steps needed by some of the algorithms for computing elementary functions.

Table 6.4: Remainder function examples. The result of $r = \mathrm{fmod}(x, y)$ has the sign of x with $|r|$ in $[0, |y|)$. By contrast, $r = \mathrm{remainder}(x, y)$ has r in $[-|y/2|, +|y/2|]$.

								n								
−8	−7	−6	−5	−4	−3	−2	−1	0	1	2	3	4	5	6	7	8
								fmod(8, *n*)								
0	1	2	3	0	2	0	0	QNaN	0	0	2	0	3	2	1	0
								fmod(−8, *n*)								
−0	−1	−2	−3	−0	−2	−0	−0	QNaN	−0	−0	−2	−0	−3	−2	−1	−0
								remainder(8, *n*)								
0	1	2	−2	0	−1	0	0	QNaN	0	0	−1	0	−2	2	1	0
								remainder(−8, *n*)								
−0	−1	−2	2	−0	1	−0	−0	QNaN	−0	−0	1	−0	2	−2	−1	−0

When x and y are finite, and y is nonzero, the functions satisfy these symmetry relations:

$$\mathrm{fmod}(x, y) = \mathrm{fmod}(x, -y) = -\,\mathrm{fmod}(x, y) = -\,\mathrm{fmod}(x, -y),$$
$$\mathrm{remainder}(x, y) = \mathrm{remainder}(x, -y) = -\,\mathrm{remainder}(x, y) = -\,\mathrm{remainder}(x, -y).$$

Section 7.12.10 of the 1999 ISO C Standard, and the subsequent Technical Corrigendum 2 (2005), define their behavior in these words:

> *The `fmod()` functions return the value $x - ny$, for some integer n such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y. If y is zero, whether a domain error occurs or the `fmod()` functions return zero is implementation-defined.*
>
> *The `remainder()` functions compute the remainder x REM y required by IEC 60559:*
>
> > *When $y \neq 0$, the remainder $r = x$ REM y is defined regardless of the rounding mode by the mathematical relation $r = x - ny$, where n is the integer nearest the exact value of x/y; whenever $|n - x/y| = 1/2$, then n is even. Thus, the remainder is always exact. If $r = 0$, its sign shall be that of x. This definition is applicable for all implementations.*
>
> *The `remquo()` functions compute the same remainder as the `remainder()` functions. In the object pointed to by `quo` they store a value whose sign is the sign of x/y and whose magnitude is congruent modulo 2^N to the magnitude of the integral quotient of x/y, where N is an implementation-defined integer greater than or equal to 3.*
>
> *The `remquo()` functions return x REM y. If y is zero, the value stored in the object pointed to by `quo` is unspecified and whether a domain error occurs or the functions return zero is implementation defined.*

6.14 Why the remainder functions are hard

From their descriptions in the Standard, `fmod(x,y)` is just `x - trunc(x/y)*y`, and with the IEEE 754 default *to nearest* rounding mode, `remainder(x,y)` is `x - rint(x/y)*y`.

Implementation looks trivial, but it is *not*, because those equations hold only in *exact arithmetic*. The simple formula $x - ny$ contains two nasty pitfalls for computer arithmetic: *overflow* and *subtraction loss*. Neither pitfall is rare in remainder computations.

It is instructive to work through a simple example to see how the computation of the remainder can fail. Figure 6.1 on the next page shows four attempts in low-precision decimal arithmetic, leading to these observations:

- The value of (x/y) is $802.439\cdots$e16, so whether `trunc()` or `rint()` is used, the three-digit result is 802e16. Thus, in that experiment, both `fmod()` and `remainder()` must give the same result.

Choose initial values near overflow and underflow limits:

$$\begin{aligned} x &= 987 \times 10^{9}\,, \\ y &= 123 \times 10^{-9}. \end{aligned}$$

First try: straightforward brute-force computation:

$$\begin{aligned} n &= \mathrm{trunc}(x/y) \\ &= \mathrm{trunc}(8\,024\,390\,243\,902\,439\,024), && \textit{overflow!} \\ &= \infty, \\ r &= x - ny \\ &= 987 \times 10^{9} - \infty y \\ &= \infty, && \textbf{\textit{wrong!}} \end{aligned}$$

Second try: use larger exponent range to eliminate overflow:

$$\begin{aligned} n &= \mathrm{trunc}(x/y) \\ &= \mathrm{trunc}(8\,024\,390\,243\,902\,439\,024) \\ &= 802 \times 10^{16}, && \textit{truncate to 3 significant digits,} \\ r &= x - ny \\ &= 987 \times 10^{9} - 802 \times 10^{16} \times 123 \times 10^{-9} \\ &= 987 \times 10^{9} - 986.46 \times 10^{9}, && \textit{exact,} \\ &= 987 \times 10^{9} - 986 \times 10^{9}, && \textit{truncate to 3 significant digits,} \\ &= 1 \times 10^{9}, && \textbf{\textit{wrong!}} \end{aligned}$$

Third try: use fused multiply-add to reduce subtraction loss:

$$\begin{aligned} r &= \mathrm{fma}(-n, y, x) \\ &= -802 \times 10^{16} \times 123 \times 10^{-9} + 987 \times 10^{9} \\ &= 540 \times 10^{6}, && \textbf{\textit{wrong!}} \end{aligned}$$

Fourth try: use exact arithmetic:

$$\begin{aligned} r &= x - ny \\ &= 987 \times 10^{9} - 8\,024\,390\,243\,902\,439\,024 \times 123 \times 10^{-9} \\ &= 987\,000\,000\,000 - 986\,999\,999\,999.999\,999\,952 \\ &= 48.0 \times 10^{-9}, && \textbf{\textit{correct answer!}} \end{aligned}$$

Figure 6.1: Computing the remainder, using low-precision decimal IEEE-like arithmetic. We have base $\beta = 10$ and precision $t = 3$, and we assume a single-digit exponent. The correct remainder, r, must satisfy $0 \leq |r| < |y|$, yet the first three attempts at computing it are grossly far off. We seem to need much wider precision, and a wider exponent range, to get the correct answer, which experiences a loss of 19 leading digits in the subtraction.

- The first attempt suffers from destructive overflow, producing an infinite remainder.

  That overflow can be eliminated by widening the exponent range. Such a feature is implemented in hardware in the IA-64 and Motorola 68000 architectures, which have two additional bits in the exponent field in floating-point registers. That allows computations like $\lfloor x/y \rfloor y$ and $\sqrt{x^2 + y^2}$ to be carried out inside the CPU without possibility of overflow for *any* values x and y that are representable in external memory storage.

- Alas, even with a wider exponent range, the second attempt still gets a remainder that is far outside the possible range, $0 \leq |r| < |y|$, of the correct answer. That is due to two problems: the product ny has insufficient precision, and the subtraction $x - ny$ loses accuracy.

- The fused multiply-add operation discussed in **Section 4.17** on page 85 can often help to reduce accuracy loss, because it computes $\mathrm{fma}(x, y, z) = xy + z$ with an exact double-length product xy and a single rounding after the addition of z.

  The third attempt introduces a fused multiply-add to try to repair the damage from the subtraction, but is still unsuccessful because n lacks adequate precision. It gets the right answer for the wrong n, and because the answer is outside the allowed range, we know that it must be wrong.

- The fourth attempt retains all digits, doing the computation in exact arithmetic, and as expected, produces the correct answer. Notice in particular that the subtraction of the 21-digit exact product ny from the 12-digit exact x loses 19 digits, producing a tiny remainder that is exactly representable in our three-digit arithmetic system.

  It is disturbing that, even if we did not use exact arithmetic, intermediate working precision *seven* times larger than storage precision is needed to get the right answer.

  What precision do we apparently need in the worst case? The largest exponent range that we need to handle in binary arithmetic is that in the IEEE 754 128-bit format, whose nonzero positive range is about $[10^{-4966}, 10^{+4932}]$, and whose decimal precision is about 35 digits. The largest n comes from dividing the largest x by the smallest y: it needs about $4932 + 4966 = 9898$ decimal digits. The product ny then has about $9898 - 4966 = 4932$ digits before the decimal point, and $35 + 4966 = 5001$ fractional digits. The subtraction $x - ny$ therefore needs about $4932 + 5001 = 9933$ decimal digits to obtain a result that we round to about 35 digits, after loss of 9898 digits in the subtraction.

  That looks computationally bleak, but fortunately, and perhaps surprisingly, it is possible to compute the remainder correctly without access to any precision higher than storage precision. In the next section, we show how to accomplish that task.

6.15 Computing `fmod()`

The discussion in the previous section makes it clear that computation of the functions `fmod()` and `remainder()` is not going to be as easy as we might have hoped from their simple definitions, which require only a divide, a multiply and an add. Indeed, in existing high-quality mathematical-function libraries, those functions are among the more complex to implement, and most of the libraries do so by messing about with bits at a low level.

The Intel IA-32 architecture has hardware instructions, `FPREM` and `FPREM1`, that compute the two kinds of remainders exactly over the entire domain of their arguments, and the Motorola 68000 architecture has similar instructions, FMOD and `FREM`. The DEC VAX `EMOD` instructions compute the lower 32 bits of the integer part, and a floating-point remainder from a product with the reciprocal of the divisor extended by 8 to 15 bits. However, after 1986, that instruction family was no longer provided in hardware, possibly due to its implementation complexity, and its use was deprecated. The IBM System/390 `DIEBR` and `DIDBR` divide-to-integer instructions for 32-bit and 64-bit IEEE 754 arithmetic produce the integer and remainder as floating-point values, with the integer part holding the lower bits of the true integer. There are no corresponding instructions for the IBM hexadecimal formats. No RISC architecture, or IA-64, supplies a comparable instruction. However, a manual for the MIPS architecture [KH92, page E-4] gives this hint about how the remainder might be computed:

> *The `remainder()` function is accomplished by repeated magnitude subtraction of a scaled form of the divisor, until the dividend/remainder is one half of the divisor, or until the magnitude is less than one half of the magnitude of the divisor. The scaling of the divisor ensures that each subtraction step is exact; thus, the remainder function is always exact.*

The definition of the remainder is equivalent to subtracting $|y|$ from $|x|$ as many times as needed to obtain a result smaller than $|y|$, *provided* that each subtraction is exact.

Given two positive numbers whose fractional points are aligned, subtraction of the smaller from the larger is an *exact* operation. Thus, in a base-β floating-point system, if we scale y by a power of the base (an exact operation) to obtain a number in the range $(x/\beta, x]$, then the subtraction is exact. The scaling is by an integer k, so that all we have done is produce a remainder $r' = x - ky$. The key observation that leads to an effective algorithm is that the remainder r' has the same remainder with respect to y as x does, but is smaller than x. We can therefore iterate the computation until $0 \leq r' < y$, and the desired remainder is then $r = r'$. Here is an implementation of that algorithm:

```
fp_t
FMOD(fp_t x, fp_t y)
{
    fp_t result;
    if (ISNAN(x) || ISNAN(y) || ISINF(x) || (y == ZERO))
        result = SET_EDOM(QNAN(""));
    else
    {
        fp_t xabs, yabs;

        xabs = FABS(x);
        yabs = FABS(y);

        if (xabs < yabs)
            result = x;
        else if (xabs == yabs)
            result = COPYSIGN(ZERO, x);
        else                    /* finite nonzero operands */
        {
            fp_t r;
            int nr, ny;

            r = xabs;
            (void)FREXP(yabs, &ny);

            while (r >= yabs)   /* loop arithmetic is EXACT! */
            {
                fp_t yabs_scaled;
                int n;

                (void)FREXP(r, &nr);
                n = nr - ny;
                yabs_scaled = LDEXP(yabs, n);

                if (yabs_scaled > r)
                    yabs_scaled = LDEXP(yabs, n - 1);

                r -= yabs_scaled;
            }

            result = (x < ZERO) ? -r : r;
        }
    }
    return (result);
}
```

The four tests in the outer `if` statement identify problem arguments that require special handling: for all of them, we record a domain error and return a NaN. That seems more useful in IEEE 754 arithmetic than the zero value suggested by the Standard for the result when the second argument is zero. Most implementations tested on about twenty flavors of UNIX returned a NaN in that case; only on OSF/1 Alpha is the result zero. On the PDP-10, the result returned by the native C library is also zero, but its floating-point system does not support NaN or Infinity.

Our code leaves to the `QNAN()` wrapper the decision of what to do about NaNs on systems that lack them. A reasonable choice might be to return the largest floating-point number, in the hope of causing noticeable problems later. Another reason for using a private function whose return value we can control, is that the 1999 ISO C Standard in Section 7.12.11.2 defines the functions `nanf()`, `nan()`, and `nanl()`, and requires that they return zero if the implementation does not support quiet NaNs. The `mathcw` library code for the `qnan()` and `snan()` function families behaves like the `nan()` family on systems that lack IEEE 754 arithmetic, but the functions can easily be replaced with private versions if particular nonzero return values are preferred.

Otherwise, we save the arguments as positive values to avoid having to deal with signs in the inner loop.

If $|x| < |y|$, the result is just x. That also nicely handles the case of x being a signed zero, as required by the Standard.

Table 6.5: Loop counts in `fmod()`. Test values are selected from a pseudorandom logarithmic distribution over the indicated ranges, averaging counts over 100 000 runs (or 10 000 for the higher counts). M and m represent the largest and smallest normal numbers for the indicated data format. The counts for ranges $(1, X)$, $(Y, 1)$ are closely approximated by $\log_2(X/Y)/4$. The 256-bit binary and decimal formats would require in the worst case more than 520 000 and 2 612 000 iterations, respectively.

x **range**	y **range**	**Average loop count**			
		32-bit	64-bit	80-bit	128-bit
$(1, 16)$	$(1/16, 1)$	3	3	3	3
$(1, 1024)$	$(1/1024, 1)$	6	6	6	6
$(1, 1024^2)$	$(1/1024^2, 1)$	11	11	11	11
$(1, 1024^4)$	$(1/1024^4, 1)$	21	21	21	21
$(1, 1024^8)$	$(1/1024^8, 1)$	41	41	41	40
$(1, \sqrt{M})$	$(\sqrt{m}, 1)$	32	257	4109	4098
$(1, \sqrt{M})$	$(m, 1)$	48	384	6173	6109
$(1, M)$	$(\sqrt{m}, 1)$	48	384	6148	6145
$(1, M)$	$(m, 1)$	64	512	8237	8366
$(M/2, M)$	$(m, 2m)$	127	1023	16383	16382

If $|x| = |y|$, then y divides x without a remainder, so we return a zero of the same sign as x, again following the Standard.

The final `else` block is where the hard work happens.

The `FREXP()` function is the easiest way to get the floating-point exponent, and the `void` cast discards the significand, which we do not require. Because y is constant inside the loop, its exponent does not change, so we only need to extract its exponent once, outside the loop.

The variable r takes the place of $|x|$ in our iteration, so it is initialized outside the loop to that value, and the loop is only entered when r is at least as large as $|y|$. Because we already handled the case of arguments equal in magnitude, the loop is always entered at least once.

Inside the loop, we extract the exponent of r, because r changes every iteration, and then form an exactly scaled y by exponent adjustment using `LDEXP()` to do the work of multiplying y by β^n. If the result is too big, we reduce the exponent by one and rescale. Those actions produce a scaled y that can be subtracted exactly from r, reducing its magnitude. Eventually, r is reduced to the required range, and the loop exits.

The final assignment sets the appropriate sign of the remainder, and the function returns the value of `result`.

The most critical part of `fmod()` is the loop. Its iteration count depends on the relative sizes of the arguments x and y, and can be measured by instrumenting the code with a loop counter, and then making numerical experiments that are summarized in Table 6.5. The iteration counts depend on the ratio x/y, and in the two longer formats for wide data ranges, can run to thousands. The large counts are a primary reason why RISC architectures do not support the remainder operation in hardware, because they are designed to have short instruction-cycle counts.

Timing tests on Intel IA-32 systems where the `FPREM` instruction does the work in the `fmod()` native-library routine show that it too runs hundreds of times slower when the ratio x/y is large.

Because all of the arithmetic is exact, no floating-point sticky exception flags can be set in `fmod()` unless the result is a run-time NaN, and `errno` is set to `EDOM` only when a NaN is returned.

6.16 Computing `remainder()`

Although the definitions of `fmod()` and `remainder()` are similar, their subtle differences make it nontrivial to compute one from another, a topic that we address in Section 6.18 on page 152.

Instead, we produce code for `remainder()` by a straightforward modification of the code for `fmod()`. There is a bit more work to be done, because `remainder()` requires that n be even when $x - ny = \pm 1/2$.

Because n can be much too large to represent, we cannot compute it, but we do not need to: we only require knowledge of whether it is odd or even. In the inner loop of fmod(), we scaled *y* by a power of the base. That scale factor is either an even number (assuming historical bases 2, 4, 8, 10, and 16) when the power is greater than one, or else it is one, when the power is zero. We just need to count the number of times the zero-power case occurs, and that is infrequent. Instrumentation of a version of our code shows that the average frequency of a zero power in the scaling is about 1.005 in the almost-worst case of random *x* near the overflow limit, and random *y* just above the smallest normal number. There is clearly no danger of an integer overflow in the frequency counter.

Here is the code that implements remainder(), with the lines that differ from the code for fmod() marked with a comment /* REM */:

```
fp_t
REMAINDER(fp_t x, fp_t y)
{
    fp_t result;

    if (ISNAN(x) || ISNAN(y) || ISINF(x) || (y == ZERO))
        result = SET_EDOM(QNAN(""));
    else
    {
        fp_t xabs, yabs;

        xabs = FABS(x);
        yabs = FABS(y);

        if (xabs < yabs)
            result = x;
        else if (xabs == yabs)
            result = COPYSIGN(ZERO, x);
        else                    /* finite nonzero operands */
        {
            fp_t r;
            int nr, ny, parity;                     /* REM */

            r = xabs;
            (void)FREXP(yabs, &ny);
            parity = 0;                             /* REM */

            while (r >= yabs)   /* loop arithmetic is EXACT! */
            {
                fp_t yabs_scaled;
                int n;

                (void)FREXP(r, &nr);
                n = nr - ny;
                yabs_scaled = LDEXP(yabs, n);

                if (yabs_scaled > r)
                {                                   /* REM */
                    --n;                            /* REM */
                    yabs_scaled = LDEXP(yabs, n);
                }                                   /* REM */

                if (n == 0)                         /* REM */
                    ++parity;                       /* REM */

                r -= yabs_scaled;
            }

            if (r > HALF * yabs)                    /* REM */
            {                                          /* REM */
                r -= yabs;                             /* REM */
                ++parity;                              /* REM */
            }                                          /* REM */

            if ( (parity & 1) && (r == (HALF * yabs)) )/* REM */
                r = -r;                                /* REM */

            result = (x < ZERO) ? -r : r;
        }
    }
    return (result);
}
```

When the loop exits, we have r in $[0, |y|)$, but `remainder()` requires a result in $[-|y/2|, +|y/2|]$. We can readily ensure that by subtracting $|y|$ from r when $r > |y/2|$, and incrementing the frequency counter. The next step is to check for the special case $r = 1/2$, and if the frequency counter is odd, to invert the sign of r. The last step fulfills the symmetry relation on the first argument, just as in `fmod()`: if x is negative, then replace r by $-r$. The case of $x = \pm 0$ has already been handled in the first and second branches of the second-level `if` statement, so we do not need to use `SIGNBIT()` in the test for a negative argument.

Because the code is so similar to that for `fmod()`, the execution time for `remainder()` is only a little bit longer. The extra adjustment of r after the loop exits amounts to about half an iteration more, which is of little significance in view of the counts recorded in **Table 6.5** on page 148.

All of the arithmetic is exact, so no flags can be set in `remainder()` unless the result is a run-time NaN, and `errno` is set to `EDOM` only when a NaN is returned.

6.17 Computing `remquo()`

The `remquo()` functions are a straightforward extension of the `remainder()` functions. Instead of tracking the parity of n in $r = x - ny$, we now have to determine n modulo some power of two larger than two. The reason for that peculiar choice is existing hardware: the Intel IA-32 remainder instructions, `FPREM` and `FPREM1`, record the three least-significant bits of the quotient in status flags, and the Motorola 68000 family `FMOD` and `FREM` instructions produce the seven least-significant bits.

One important application of the remainder is reduction of arguments to trigonometric functions. If at least three low-order bits of the quotient are available, they can be used to select an octant, quadrant, or half-plane in which the angle lies.

Here is the code for `remquo()`, with the changes from the code in `remainder()` marked with a comment `/* REMQUO */`:

```
fp_t
REMQUO(fp_t x, fp_t y, int *quo)
{
    fp_t result;
    unsigned int q;                                 /* REMQUO */

    if (ISNAN(x) || ISNAN(y) || ISINF(x) || (y == ZERO))
    {
        result = SET_EDOM(QNAN(""));
        q = UI(0);                                  /* REMQUO */
    }
    else
    {
        fp_t xabs, yabs;

        xabs = FABS(x);
        yabs = FABS(y);
```

```
if (xabs < yabs)
{
    result = x;
    q = UI(0);                              /* REMQUO */
}
else if (xabs == yabs)
{
    result = COPYSIGN(ZERO, x);
    q = UI(1);                              /* REMQUO */
}
else                    /* finite nonzero operands */
{
    fp_t r;
    int nr, ny, parity;

    r = xabs;
    (void)FREXP(yabs, &ny);
    parity = 0;
    q = UI(0);                              /* REMQUO */

    while (r >= yabs)   /* loop arithmetic is EXACT! */
    {
        fp_t yabs_scaled;
        int p;

        (void)FREXP(r, &nr);
        p = nr - ny;
        yabs_scaled = LDEXP(yabs, p);

        if (yabs_scaled > r)
        {
            --p;
            yabs_scaled = LDEXP(yabs, p);
        }

        if (p == 0)
            ++parity;

        if ( (0 < p) && (p < UINT_BITS) )   /* REMQUO */
            q += (UI(1) << (unsigned int)p);/* REMQUO */

        r -= yabs_scaled;
    }

    if (r > HALF * yabs)
    {
        r -= yabs;
        ++parity;
        ++q;                                /* REMQUO */
    }

    if ( (parity & 1) && (r == (HALF * yabs)) )
        r = -r;

    result = (x < ZERO) ? -r : r;
}
```

```
    }

    q &= INT_MAX;  /* modular reduction to int */   /* REMQUO */

    if (quo != (int *)NULL)                         /* REMQUO */
        *quo = ((COPYSIGN(ONE, x) *                 /* REMQUO */
                 COPYSIGN(ONE, y)) < ZERO) ?        /* REMQUO */
                -(int)(q) : (int)q;                 /* REMQUO */

    return (result);
}
```

The five instances of the `UI()` wrapper macro reduce clutter and hide a portability issue. They return their argument prefixed with an `unsigned int` type cast. In modern C code, their values would simply be written `0U` and `1U`, using a type suffix to indicate an unsigned integer constant. The `U` suffix was introduced into the language by 1990 ISO Standard C, but because the mathcw library is designed to work with compilers on older systems as well, we cannot use that suffix. Although few compilers would complain if the wrapper, and thus, the cast, were omitted, the package author considers it poor programming practice to mix signed and unsigned arithmetic.

In our software implementation, there is no good reason to restrict the number of bits returned to fewer than can fit in the integer pointed to by the final argument of `remquo()`. For simplicity, we accumulate it as an unsigned value, which provides the modulo property automatically during addition, and then just before return, we mask off the sign bit to obtain a positive signed integer, and then apply the correct sign.

When `q` is incremented by `(UI(1) << p)`, which is just a fast way to compute 2^p, it is imperative to check that the shift count is smaller than the number of bits in an unsigned integer. The reason for that check is this statement about the bit-shift operators from Section 6.5.7 of the 1999 ISO C Standard:

> *If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.*

When the shift count is too large, we want the result of the shift to be zero, because that corresponds to arithmetic modulo 2^N, and in that case, no increment of q is needed.

Curiously, we have to supply our own definition of `UINT_BITS`, because the standard header files define no constants from which the number of bits in an unsigned integer can be easily obtained. The header file `remqu.h` defines the macro `UINT_BITS`, and makes a sanity check with preprocessor arithmetic that its value is consistent with `INT_MAX`, a constant that *is* provided in `<limits.h>`.

As in `remainder()`, all of the arithmetic is exact, and no floating-point sticky exception flags can be set in `remquo()` unless the result is a run-time NaN, and `errno` is set to `EDOM` only when a NaN is returned.

Clearly, we could view `remquo()` as the fundamental remainder primitive, and then define `remainder()` like this:

```
fp_t
REMAINDER(fp_t x, fp_t y)
{
    return (REMQUO(x, y, (int *)NULL));
}
```

All that would be lost would be a bit of speed, but even that is not much, because the two extra statements in the time-consuming inner loop require only an integer comparison, shift, and unsigned addition, which is relatively cheap compared to the floating-point arithmetic and exponent extraction and scaling.

In the mathcw library, however, we use our three independent implementations of `fmod()`, `remainder()`, and `remquo()`, conforming to the design requirements given earlier in **Section 1.1** on page 2.

6.18 Computing one remainder from the other

The description of the IEEE 754 Standard [CCG+84, page 92, column 2] contains this remark:

> *REM is defined as it is, instead of matching the "mod" function, found in many programming languages, because the latter can always be computed from the former, but the converse is not always true. This is so because REM's remainder is the smallest*

> *possible remainder in magnitude, and is always exact. ... If y is near the underflow threshold (i.e., $|y| < \beta^{Emin+t}$), it is possible that $r = x$ REM y may be subnormal.*

As that quotation suggests, it is relatively straightforward to compute `fmod()` from `remainder()`, provided that $\beta = 2$. We essentially have to move the result range from $[-|y/2|, +|y/2|]$ to $[0, |y|)$ by computing the IEEE-style remainder for positive arguments, add y to a negative result, and then apply the correct sign, because `fmod()` has the same sign as its first argument, provided that the arguments are finite numbers. Here is code to do just that:

```
fp_t
FMOD(fp_t x, fp_t y)
{   /* compute fmod(x,y) from remainder(x,y) */
    fp_t r;

    r = REMAINDER(FABS(x), FABS(y));

    if (r < ZERO)
        r += FABS(y);

    return (COPYSIGN(r, x));
}
```

We dispense with the initial checks that lead to a NaN return value, because they are done again inside `remainder()`. If a NaN is returned, the only thing that can happen to it is for its sign to be changed, and that is acceptable because the sign of NaN is never significant.

If r is negative, then $r + |y| < |y|$, so the adjustment to r cannot cause overflow when $|y|$ is near the largest floating-point number.

If r is subnormal, which can happen when x and y are both just above the smallest positive normal number, then as long as subnormals are not flushed to zero, the code behaves correctly. If subnormals are not supported, then both `REMAINDER()` and `FMOD()` return a zero result when the exact result would be subnormal.

Tests of that algorithm on several CPU architectures turned up problems with `remainderf()`, but not for `remainder()`, on two platforms. On DEC OSF/1 Alpha, with subnormals enabled at compile time, a test for a NaN produces an incorrect result for subnormal arguments. On SGI IRIX MIPS with subnormals enabled at run time, a test for a zero denominator produces an incorrect result. Switching from native compilers to `gcc` restored correct behavior. Although those failures of `remainderf()` are both vendor errors, they are in nondefault environments that evidently received inadequate testing.

It is substantially harder to go in the reverse direction, because `fmod()` has lost information about whether n is odd or even. Nevertheless, that is what Sun Microsystems' `fdlibm` library does. As previously noted, that library grovels around at the bit level, but we can rewrite its algorithm to be portable.

The critical clue to deriving `remainder()` from `fmod()` is to realize that if we first compute $r = x - 2ny$, or equivalently, $r = x - n(2y) = \mathrm{fmod}(x, 2y)$, then we know that $2n$ is even. That produces a result r whose magnitude is in the range $[0, |2y|)$, whereas we need a result in $[-|y/2|, +|y/2|]$. If $|r|$ is in $(|y/2|, |2y|)$, we subtract $|y|$ to reduce the range to $[-|y/2|, +|y|)$; otherwise, r is already in range. Then, if we did the subtraction, and r is now in $[|y/2|, |y|)$, we again subtract $|y|$, reducing the range to $[-|y/2|, 0)$. The result of those adjustments is that r is now in $[-|y/2|, +|y/2|]$. The last step is to copy the sign of x to r, obtaining a result in $[-|y/2|, +|y/2|]$. In the comparisons, care is needed to handle the end-of-range tests properly.

The work is not quite done, however, because in forming $2y$, that value overflows if $|y|$ is bigger than half the largest normal number. The solution is surprisingly simple: call `FMOD()` only if that overflow is impossible. When that call is avoided, we have $|y| >$ `FP_T_MAX/2`, and there are three possibilities that the remainder of the code properly handles:

$|x| > |y|$: the result is $|x| - |y|$, an *exact* operation because both values have the same exponent;

$|x| = |y|$: the result is 0.0;

$|x| < |y|$: the result is $|x|$.

At the other end of the floating-point range, if y is smaller than twice the smallest normal number, then $y/2$ underflows, either to zero if subnormals are not supported, or losing bits if a subnormal result is produced. Either way, our requirement of *exact* arithmetic is violated.

Those additional complications make the code about two-thirds the size of our original code for the function, and the complexity is such that the control-flow logic needs detailed thought by the programmer in order to have confidence in its correctness.

```
const fp_t HALF_MAXNORMAL  = FP(0.5) * MAXNORMAL;
const fp_t TWICE_MINNORMAL = FP(2.0) * MINNORMAL;

fp_t
REMAINDER(fp_t x, fp_t y)
{   /* compute remainder(x,y) from fmod(x,y) */
    fp_t result;

    if (ISNAN(x) || ISNAN(y) || ISINF(x) || (y == ZERO))
        result = SET_EDOM(QNAN(""));
    else
    {
        fp_t xabs, yabs;

        xabs = FABS(x);
        yabs = FABS(y);

        if (y <= HALF_MAXNORMAL)
            xabs = FMOD(xabs, yabs + yabs);

        if (xabs == yabs)
            result = COPYSIGN(ZERO, x);
        else if (yabs < TWICE_MINNORMAL)  /* near underflow limit */
        {
            if ((xabs + xabs) > yabs)
            {
                xabs -= yabs;

                if ((xabs + xabs) >= yabs)
                    xabs -= yabs;
            }
        }
        else                              /* safe from underflow */
        {
            fp_t yabs_half;

            yabs_half = HALF * yabs;

            if (xabs > yabs_half)
            {
                xabs -= yabs;

                if (xabs >= yabs_half)
                    xabs -= yabs;
            }
        }
        result = (x > ZERO) ? xabs : -xabs;
    }
    return (result);
}
```

Because all of the arithmetic is exact, no floating-point sticky exception flags can be set in either of those alternate implementations of `fmod()` and `remainder()` unless the result is a run-time NaN, and `errno` is set to `EDOM` only when a NaN is returned.

6.19 Computing the remainder in nonbinary bases

Throughout this book, and the design of the `mathcw` library, we avoid making assumptions about the base of floating-point numbers, and we comment on the effect of wobbling precision from hexadecimal normalization, and modify our algorithms to accommodate it.

The floating-point arithmetic in the remainder-function families of `fmod()`, `remainder()`, and `remquo()` has been carefully arranged to be *exact*. We must ensure that ports of our software to additional platforms do not silently invalidate that design requirement.

Unfortunately, our implementation of code in the C language has quietly introduced an assumption that the floating-point base is two, as it is for IEEE 754 arithmetic, and most computers designed since 1965. That assumption crept in through our use of the C library function families `frexp()` and `ldexp()`, which are defined to manipulate numbers represented as a significand multiplied by a *power of two*. On IBM System/360 with hexadecimal normalization, computation of $(8+\epsilon) \times 2^{-3}$ by `ldexp(8.0 + epsilon, -3)` loses three low-order bits in the significand, violating our requirement of exact arithmetic.

What we need instead is for our wrapper macros `FREXP()` and `LDEXP()` to work in terms of the native base. With decimal floating-point arithmetic, they should work with powers of 10. With hexadecimal normalization on IBM mainframes, they need to work with powers of 16. The required functions are provided by the C99 `logb()` and `scalbn()` family.

No further changes are needed in our code for `FMOD()` or `REMAINDER()`, but one more change is needed in `REMQUO()`. In that function, we accumulate in the variable `q` the value of n modulo 2^N, and we do so with the statement

```
if (p < UINT_BITS)
    q += (UI(1) << p);
```

When the floating-point base is not 2, the test is wrong, and the correct increment is β^p, which cannot be optimized to a simple bit shift. The C library contains no standard function for computing powers of integers, but we can get what we need with the help of our base-β version of `LDEXP()`, and we can reduce the result modulo 2^N with `FMOD()`. The accumulation then looks instead like this:

```
q += UI(fmodl(ldexpbl((long double)BASE, p), (long double)1.0 + (long double)UINT_MAX));
```

In order to do an exact unsigned integer computation in floating-point arithmetic, we need to ensure that the floating-point range is sufficient to hold all values that are representable in the integer type. In practice, that requires `double` if we have fewer than about 50 bits in an integer, and `long double` in the case of a 64-bit integer type.

If those constraints cannot be met, then the definition of the `remquo()` family gives us the freedom to reduce the size of the modulo value, 2^N. For example, to guarantee a 16-bit modular quotient, we could instead use this code:

```
q = (q + UI(fmod(ldexpb((double)BASE, p), 65536.0))) & UI(65535);
```

with the final increment and modular reduction statements

```
++q;

q &= INT_MAX;
```

replaced by this statement:

```
q = (q + UI(1)) & UI(65535);
```

In the implementation of `REMQUO()` that we showed earlier, we did not clutter the code with preprocessor conditionals to handle those variants, but our actual library code has to do so.

6.20 Summary

The functions in this chapter are important equipment in the programmer's toolbox. Although the C-language family makes it easy to convert floating-point values to integers with a type cast, that practice is *not* recommended unless it is known that the floating-point value is finite and within the range of representable integers. As **Table 6.1** on page 131 shows, there is no consensus about what hardware should do for out-of-range conversions, and unchecked conversions are likely to lead to surprises, and nasty bugs, when software is moved to a new platform.

C99, and the `mathcw` library, provide a powerful collection of functions for safe conversions with different rounding choices, and those functions deserve to be used more often in place of type casts. Checks for integer overflow are generally advisable as well, and are too often omitted. Indeed, both programming-language architects and hardware designers are guilty of making integer arithmetic unsafe, and violations of integer bounds are one of the commonest causes of software failures. The safe-integer arithmetic functions described in **Section 4.10** on page 72 offer additional ways to make such arithmetic trustworthy.

Decomposition of a floating-point value into integer and fractional parts via the `MODF()` family is exact and fast. That operation lies at the core of other functions for ceiling, floor, rounding, and truncation, all of whose results are floating-point whole numbers.

The functions that compute remainders are always exact, and that property makes them particularly useful in the argument reduction required for some of the elementary functions. However, they are not a complete solution to the problem, for reasons that we describe in more detail in **Chapter 9**.

When the arguments of a remainder function can be negative, consider carefully what the result should be. Programming languages differ in their definitions of remainder operations in such cases, and identically named remainder functions in two languages may mean different things.

Section 6.14 on page 144 shows by a low-precision example why the remainder functions are hard, and why simple shortcuts in their computation lead to catastrophically wrong results. As floating-point precision and range increase, the remainder functions become computationally intensive in their worst cases (see **Table 6.5** on page 148), and can take much longer than ordinary division, and even much longer than most of the elementary functions. They should therefore be viewed as potential hot spots when code is examined for efficiency.

7 Random numbers

ANY ONE WHO CONSIDERS ARITHMETICAL METHODS
OF PRODUCING RANDOM NUMBERS IS, OF COURSE,
IN A STATE OF SIN.

— JOHN VON NEUMANN (1951).

A RANDOM NUMBER GENERATOR CHOSEN
AT RANDOM ISN'T VERY RANDOM.

— DONALD E. KNUTH.

Certain kinds of computations, such as sampling and simulation, need a source of random numbers. However, there are three significant problems when we try to compute such numbers:

- Numbers in computers, whether integer or floating-point, are *rational* numbers. Such numbers can only approximate mathematical real numbers, and therefore, truly random numbers cannot be produced by any computer algorithm.
- Most algorithms for generation of 'random' numbers produce a sequence of almost-always-different values that eventually repeats. The length of that sequence is called the *period*. By contrast, a stream of truly random numbers has occasional repetitions, and is never periodic.
- Software for random-number generation often contains subtle dependencies upon the behavior, precision, and range of integer and floating-point arithmetic.

We can therefore at best hope to create approximations to random numbers. Some authors prefer to call 'random' numbers produced by a computer algorithm *pseudo-random*, and such an algorithm is called a *pseudo-random-number generator*, often shortened to the acronym *PRNG*. For brevity, in the rest of this chapter, we omit the qualifier 'pseudo'.

Most random-number algorithms are characterized by a fixed set of parameters, together with one or more user-settable values called *seeds*. For a given choice of initial seeds, the generator produces the same sequence of values.

7.1 Guidelines for random-number software

Several decades of computer use in many different fields show that software for producing random numbers should have several desirable characteristics:

- The sequence of the random numbers, and *any sequence of bits* extracted from them, must be apparently unpredictable.
- The *range* of possible numbers must be documented.
- The *period* must be large and documented.
- It must be possible to retrieve the current seed(s).
- It must be possible to set the seed(s) at any time, thereby starting a new sequence of random numbers.
- It should be possible to generate multiple simultaneous, independent, and nonoverlapping streams of random numbers. Although that can be achieved by swapping seed(s), it is more convenient to have generator *families*, with the family selected at the function call.
- The generator must be portable, and give identical results on each platform.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_7

- The generator must begin with the same initial seed(s) each time the user program is run, so that computations are reproducible. The user may choose to alter that behavior by changing the seed(s) on each run.

- The generator must not require the user to call an initializing routine before calling the generator; any required initialization must be handled internally.

- The generator must not require randomness in user-provided seeds to produce random output: seeds of $0, 1, 2, 3, \ldots$ should be as satisfactory as random seeds.

- The central generator algorithm must produce *integer* results. Conversion to floating-point values is then a scaling and translation, although as shown in **Section 7.3** on page 160, considerable care is required to implement that apparently simple operation correctly.

  Algorithms that produce floating-point values directly are unlikely to be able to produce the same number sequence on different platforms.

- The generator must be available as source code, rather than hidden in a vendor-provided binary load library, so that user programs can be ported to other computers and still give the same results.

- The generator must be fast, costing no more than a few elementary arithmetic operations for each result. Speed is a requirement because some applications may require as many as 10^{15} random numbers.

- There should be a simple validation check that can be used as a sanity test on the generator's correct operation. That should be done automatically when the generator is used for the first time, and should also be available as a user-callable routine. It could, for example, verify that, after starting with a specified seed, and discarding the first hundred random numbers, the next ten values match the expected ones.

- The generator algorithm should be implementable in multiple programming languages.

- There should be a routine to compute sequences of random numbers in batches of user-specified size considerably faster than they can be computed by individual function calls.

It is surprising how often most of those requirements have been ignored by software developers: indeed, there are programming languages with predefined random-number generators that fail to make the seed(s) accessible, or do not document the period and range. It is a common failing in many random-number research publications to leave out that essential information as well. The Scalable Parallel Random Number Generators (SPRNG) library [MS00a] is a notable exception that addresses many of our points.

7.2 Creating generator seeds

Novices sometimes think that the generator should be set to produce different random-number sequences on each run of the program. However, that is undesirable for three good reasons:

- Debugging may be infeasible if program behavior is not reproducible.

- It makes numerical experiments impossible to reproduce, violating one of the key tenets of the scientific method.

- When multiple software models of a simulation are compared, it is essential that they receive identical streams of random numbers. Otherwise, differences in model predictions could just be due to differing input data, rather than to the models themselves. Research papers in that area often use the capitalized phrase *Common Random Numbers*, or *variance reducing*, to indicate the choice of identical random streams.

If different sequences are required on each run, either supply the seed(s) as part of the input data, or create a unique seed by combining data that distinguish one run from another. For example, modern computers have a `time()` function that reports the value of a calendar clock that counts up, usually once per second, from a date several decades in the past, and a `clock()` function that returns the number of CPU microseconds used since the job began, although the job-timer resolution may be as poor as 50 ticks per second. Combining the clock counters with other job attributes, such as process and thread numbers, produces a seed that is almost certain to be unique and nonreproducible. Although process numbers are often small and sequentially assigned, some newer operating systems increase their range, and randomize them to defend against process-number guessing attacks. Here is a short C function that creates a seed that way, using the *exclusive-OR* operation (the caret operator, ^) to mix bits quickly and without overflow:

```
#include <limits.h>
#include <time.h>
#include <unistd.h>

typedef unsigned int UINT_T;

UINT_T
makeseed(void)
{   /* make unique generator seed */
    UINT_T n, p, seed, t;
    static UINT_T k = 0;
    static const UINT_T c = 0xfeedface;
    static const int half_wordsize = CHAR_BIT * sizeof(UINT_T) / 2;

    n = 1 | (--k & 0x0f);

    p = (UINT_T)getpid();
    p ^= (p << half_wordsize);
    p ^= p << n;

    t = (UINT_T)time((time_t *)NULL);
    t ^= (UINT_T)clock();
    t ^= t << half_wordsize;
    t ^= t << n;

    seed = c ^ k ^ p ^ t;

    return (seed);
}
```

Changing the initial `typedef` statement allows the code to produce unsigned integers of any supported size.

For each bit position in the operands, the exclusive-OR produces 0 when the corresponding input bits are identical, and 1 when they differ. It therefore produces 0 and 1 with equal probability when the input bits are random, and for that reason, is often useful in software for generating random numbers.

The counter `k` decrements on each call. The variable `n` also changes each call, and is restricted to lie in $[1, 15]$, so that it is a legal shift count that always moves bits. The two shifts help to mix bits from the process number and two time values, one in seconds, and the other in microseconds. Using the process number ensures that multiple instances of the job started about the same time each get different seeds, and changing `k` and `n` produces different results on repeated calls within the same process. The final seed is the exclusive-OR of a constant with the counter, the mixed process number, and the scrambled time values.

To test our seed generator, we use a short test program that calls `makeseed()` repeatedly and outputs the returned values, one per line. A command loop runs the test multiple times *sequentially*, sorting the output into unique lines and counting the number of unique seeds produced in each test. The loop output is then formatted into lines of a convenient size:

```
$ cc -O3 -DMAXTEST=1000000 test-makeseed.c
$ for k in ‘seq 1 40‘
> do
>     ./a.out | sort -u | wc -l
> done | fmt -w70
 999345  999654  999590 1000000 1000000 1000000 1000000  999870
1000000  999726 1000000 1000000  999315  999664 1000000 1000000
1000000  999893 1000000 1000000  999936 1000000 1000000  999550
```

```
1000000  999686  999526  999904 1000000  999806  999798 1000000
1000000 1000000 1000000 1000000 1000000  999822 1000000 1000000
```

That demonstrates that the seed generator can often produce unique seeds in one million repeated calls, and at worst, repeated seeds occur with a frequency below 0.001.

However, when a similar test is repeated on a large multiprocessor system with *parallel*, and longer, runs of the test program, the results show repeated-seed frequencies up to about 0.012:

```
$ cc -O3 -DMAXTEST=100000000 test-makeseed.c
$ for f in ‘seq 1 40‘
> do
>     ./a.out | sort -u | wc -l &
> done | fmt -w70
98875567 98844854 98885473 98885800 98907838 98829088 98813968
98903619 98838402 98815331 98879819 98862152 98879141 98844464
98836598 98874230 98883376 98856824 98854752 98838834 98818495
98893962 98831324 98848314 98810846 98864727 98898506 98837900
98854240 98908321 98891786 98879817 98839639 98854624 98847177
98868476 98799848 98880452 98845182 98861177
```

If the job uses threads, then the updates to the retained counter `k`, or else all calls to the `makeseed()` function, need to be protected with locks to ensure access by only one thread at a time.

In most cases, however, the same starting seed should be used for reproducibility. The program output should record the initial seed(s), and the first and last few random numbers, to allow checking results against supposedly identical runs made on other computer systems.

7.3 Random floating-point values

Many applications of random numbers require a random floating-point value between zero and one. If we have a random-number generator that returns an integer result n in the interval $[a, b]$ it is *almost* straightforward to produce a floating-point result x conforming to any of the four possible interval-endpoint conventions:

$$\begin{aligned}
x &= (n-a)/(b-a), && x \textit{ in } [0,1],\\
x &= (n-a+1)/(b-a+1), && x \textit{ in } (0,1],\\
x &= (n-a)/(b-a+k), && x \textit{ in } [0,1), k \geq 1 \textit{ to be determined},\\
x &= (n-a+1)/(b-a+k), && x \textit{ in } (0,1), k > 1 \textit{ to be determined}.
\end{aligned}$$

Here, the divisions are done in floating-point arithmetic, and the scale factors $1/(b-a+1)$ and $1/(b-a+k)$ must be nonzero and representable in floating-point arithmetic. The width of the interval $[a, b]$ is often 2^p or $2^p - 1$, where $p = 15, 16, 31, 32, 63$, or 64. The scale-factor requirement is easily met with current, and most historical, floating-point systems.

The tricky cases are the last two: k must be carefully chosen to ensure that the correctly rounded result is *strictly less than one*. Let $M = b - a$ be the width of the range of the integer generator. We then need to find the smallest k such that $M/(M + k) < 1$ in floating-point arithmetic. By dividing numerator and denominator by M, we can rewrite that requirement as $1/(1 + k/M) < 1$. Then we recall from school algebra that when the magnitude of x is small compared to 1, the expansion $1/(1 + x) \approx 1 - x + x^2 - x^3 + \cdots$ converges rapidly. We therefore have

$$\begin{aligned}
x &= k/M\\
&\geq \beta^{-t}, && \textit{the little, or negative, epsilon},\\
k &\geq \lceil M\beta^{-t} \rceil.
\end{aligned}$$

The last equation gives a good starting guess for k, but is not guaranteed to be the smallest value that satisfies $M/(M + k) < 1$, because of the approximations in its derivation, and because different implementations of floating-point arithmetic may vary slightly in the accuracy of division. We therefore wrap it in a function, `randoffset()`, that determines the correct value of k by direct measurement.

```
unsigned long int
randoffset(unsigned long int m)
{   /* return smallest k such that fl(m)/(fl(m) + fl(k)) < 1 */
    double mm;
    volatile double x;
    unsigned long int d, k;

    mm = (double)m;
    k = (unsigned long int)ceil(mm * DBL_EPSILON / (double)FLT_RADIX);
    d = (unsigned long int)nextafter((double)k, DBL_MAX) - k;
    d = (d < 1) ? 1 : d;

    for (;;)
    {
        x = mm / (mm + (double)k);
        STORE(&x;)

        if (x < 1.0)
            break;

        k += d;
    }

    return (k);
}
```

An accompanying test program in the file `rndoff.c` produces this report of offsets on a system with a 64-bit `long int` data type, where the initial output lines contain n, $m = 2^n$, k (from `randoffset()`), and $\mathrm{fl}(m)/(\mathrm{fl}(m) + \mathrm{fl}(k))$:

```
% cc rndoff.c -lm && ./a.out
 1                   2     1  0.66666666666666663
 2                   4     1  0.80000000000000004
 3                   8     1  0.88888888888888884
 4                  16     1  0.94117647058823528
 5                  32     1  0.96969696969696972
 ...
51      2251799813685248     1  0.99999999999999956
52      4503599627370496     1  0.99999999999999978
53      9007199254740992     2  0.99999999999999978  0x1p+0
54     18014398509481984     3  0.99999999999999978  0x1p+0
55     36028797018963968     5  0.99999999999999978  0x1p+0
56     72057594037927936     9  0.99999999999999978  0x1p+0
57    144115188075855872    17  0.99999999999999978  0x1p+0
58    288230376151711744    33  0.99999999999999978  0x1p+0
59    576460752303423488    65  0.99999999999999978  0x1p+0
60   1152921504606846976   129  0.99999999999999978  0x1p+0
61   2305843009213693952   257  0.99999999999999978  0x1p+0
62   4611686018427387904   513  0.99999999999999978  0x1p+0
63   9223372036854775808  1025  0.99999999999999978  0x1p+0
```

When k is bigger than one, the output also reports the quotient for the next smaller representable divisor in hexadecimal floating-point form. That quotient must be exactly one if we have chosen k correctly. The output shows that the requirement is satisfied.

The optimal choice of k clearly depends on the range of the integer random-number generator, whether that range can be represented exactly as a floating-point value, and on the precision and rounding characteristics of the host computer. That means that code that implements the conversion of random integers to floating-point values on the intervals $[0, 1)$ and $(0, 1)$ is *inherently not portable*, unless k is determined at run time by a routine like our `randoffset()`.

Sanity checks in the validation code should include tests of the endpoint behavior in the production of floating-point random numbers.

Published code that produces floating-point values from integer random numbers often contains magic multipliers, like `4.656612873e-10`, for the conversion from integer to floating-point values. Closer examination shows that those constants are approximations to the value $1/(M+k)$, and our cited example corresponds to $M = 2^{31} - 1$ and $k = 1$. That practice conceals three nasty problems:

- Floating-point constants are subject to base-conversion errors (see **Chapter 27** on page 879), unless they can be written as exact values in the floating-point base, or as rational decimal numbers that can be evaluated exactly.
- The magic constants implicitly depend on the host floating-point precision and on the accuracy and rounding characteristics of its multiply and divide operations.
- If the IEEE 754 rounding mode is changed dynamically, then even though the sequence of random integers is unaffected, the sequence of random floating-point values may be altered. Worse, but rarely, some values could even be slightly outside the promised range of the generator.

Those issues make the code nonportable, and likely to produce surprises on at least one platform where the computed result turns out to be exactly one, instead of strictly less than one. The conversion of integer random numbers to floating-point values is another instance of the danger of replacing (slow) division with (fast) multiplication by the reciprocal.

In this section, we have taken considerable care to show how a stream of random integers can be converted to floating-point values on four variations of the unit interval. That is necessary if essentially identical (apart from rounding error) floating-point values must be produced on every computing platform, which is usually the case.

If portability and identical streams are not important, then there is an easier approach: discard any unwanted endpoint values, and try again. For example, suppose that we have an integer random-number generator that produces values on $[0, M]$. A possible candidate is the C89 library function `rand()` that returns values of data type `int` in the range $[0,$ `RAND_MAX`$]$. The upper limit is typically either $2^{15} - 1 = 32\,767$, or $2^{31} - 1 = 2\,147\,483\,647$. A better choice is the POSIX library routine, `lrand48()`, that returns `long int` values in the range $[0, 2^{31} - 1]$, independent of the underlying architecture [Rob82]. It is a generator of a special kind that we discuss later in **Section 7.7.1** on page 169 with parameters $A = 25\,214\,903\,917$, $C = 11$, and $M = 2^{48}$. We can then write the four functions like this:

```
#include <stdlib.h>     /* for prototypes of rand() and lrand48() */

#if defined(_XPG4)      /* switch to improved POSIX generator */
#define rand() lrand48()
#undef RAND_MAX
#define RAND_MAX 0x7fffffffL
#endif /* defined(_XPG4) */

double
rand1(void)
{   /* return random value on [0,1] */
    volatile double r;

    r = (double)RAND_MAX;
    STORE(&r);

    return ((double)rand() / r);
}

double
rand2(void)
{   /* return random value on [0,1) */
    double result;

    do { result = rand1(); } while (result == 1.0);
    return (result);
}

double
rand3(void)
{   /* return random value on (0,1] */
    double result;

    do { result = rand1(); } while (result == 0.0);

    return (result);
}

double
rand4(void)
{   /* return random value on (0,1) */
    double result;

    do { result = rand1();} while ((result == 0.0) || (result == 1.0));

    return (result);
}
```

A compile-time definition of the macro _XPG4 selects a POSIX compilation environment, and exposes the needed declarations of `lrand48()` and its family members in `<stdlib.h>`. The redefinitions of `rand()` and `RAND_MAX` then replace them with settings appropriate for the improved POSIX generator.

There are subtle portability issues in the four functions:

- We assume that the number of bits in an `int` is not larger than the number of bits in the significand of a `double`. That is almost always safe, but would not be on a system, such as an embedded processor, where `double`, `float` and `int` might all be 32-bit types.
- If we simply divide the result of `rand()` by `RAND_MAX`, a compiler might replace the division by a compile-time constant with multiplication by its reciprocal, which is unlikely to be an exactly representable value. That introduces a rounding error that might prevent the value 1.0 from ever being generated. Our `volatile` and `STORE()` subterfuges in `rand1()` force retention of the division.
- The range of possible values from `rand()` is implementation dependent, and on many systems, inadequate for all but toy applications.
- The limited range of `rand()` propagates to our four derived functions, and their return values cannot contain any more random bits than `rand()` can produce.
- Because of those dependencies, we cannot guarantee the same number stream on all platforms.

We cannot satisfy the identical-streams requirement without providing our own portable version of `rand()`, and we revisit that issue later.

We can fix the problem of not having sufficient random bits by invoking the integer generator multiple times, and combining its results with exact arithmetic to populate the full significand. Here is a new version of the $[0,1]$ generator that implements those extensions:

```
double
rand1_new(void)
{   /* return random value on [0,1] */
    double r, result;
    int k;
    static const double R = 1.0 / 32768.0; /* exact */
    assert(RAND_MAX >= 32767);

    r = R;
    result = 0.0;

    for (k = 0; k < (DBL_MANT_DIG + 14) / 15 ; ++k)
    {
        result += (double)(rand() & 0x7fff) * r;
        r *= R;
    }

    return (result);
}
```

The assertion guarantees that we do not proceed unless we have at least 15 bits from `rand()`. The loop iteration count is $\lceil t/15 \rceil$, which is the number of 15-bit chunks needed. The value `R` is a negative power of two that is exactly representable for bases $\beta = 2, 4, 8$, and 16, and the scale factor `r` is also exact in those bases.

For decimal arithmetic, however, we should do something different, because repeated conversions from binary to decimal are expensive. A reasonable approach is to first compute a random number in binary arithmetic, and then make a single conversion of that value to decimal. That may be acceptable for the single- and double-precision formats, but given that the precision of the decimal type `decimal_long_double` exceeds that of `long double` on many systems, we would have to combine at least two binary values with suitable scaling to make one decimal result. The ideal for decimal arithmetic would be to have a generator that works entirely in decimal, but most good algorithms for generating random numbers require binary operations.

Unfortunately, our attempt to guarantee that all significand bits returned by `rand1_new()` are random has broken our promise that the results lie in $[0, 1]$; instead, they lie in $[0, 1)$. The right endpoint is no longer reachable, because we do not know the largest significand that successive calls to `rand()` can produce, and thus, we cannot scale the final significand correctly so that 1.0 is a possible value. Although it might be feasible for a simple generator to enumerate all possible random numbers and find the sequence that produces the largest significand, that is impossible with the good generators that have long periods. That deficiency is not serious for most applications, because even if we used a generator that returned, with one call, enough bits for the t-bit significand, the probability of producing the maximal value is likely to be much less than 2^{-t}.

There is another issue that is often overlooked when a stream of random integers is scaled to obtain floating-point values on the unit interval: their average. With a sufficiently large number of random floating-point values on $[0, 1]$ or $(0, 1)$, we should expect that their average should tend to $\frac{1}{2}$. Suppose that our integer generator produces all possible random integers, r_i, in $[0, M-1]$ exactly once in each period. We then compute the average of the floating-point values like this:

$$\begin{aligned}
u_i &= r_i/(M-1+k), && \textit{uniform scaling}, \\
\text{average} &= (1/M)\sum_{i=0}^{M-1} u_i, \\
&= (1/M)\sum_{i=0}^{M-1} r_i/(M-1+k), \\
&= \frac{1}{M(M-1+k)}[0+1+2+\cdots+(M-1)], && \textit{sorted integers}, \\
&= \frac{1}{M(M-1+k)}\frac{M(M-1)}{2}, \\
&= \frac{(M-1)}{2(M-1+k)}.
\end{aligned}$$

Unless we choose $k = 0$, the average differs slightly from the expected value $\frac{1}{2}$, and if $k > 0$, is below that value. Scaling therefore introduces a slight bias in the uniform distribution if we allow nonzero k values to avoid the endpoints of the unit interval. That observation suggests that it is better to set $k = 0$, and then discard unwanted endpoint values for unit intervals other than $[0, 1]$.

7.4 Random integers from floating-point generator

Some programming-language environments offer only a floating-point generator for uniform distributions over the unit interval. We look here at how to recover uniformly distributed integers from those floating-point values.

We showed in **Section 7.3** on page 160 that there are unexpected subtleties in the conversion of random integers to floating-point values, so we should be prepared for similar problems in the reverse direction.

To avoid tying our code to specific library random-number routines, we use the name `urand()` as a wrapper for any function that generates uniformly distributed floating-point random numbers on the unit interval, without being specific about the endpoint behavior. Similarly, we use the name `uirand()` as a wrapper for a generator that returns random integers.

Many programmers incorrectly assume that `m + (int)((n - m)*urand())` produces random integers uniformly distributed on $[m, n]$. The results lie on that interval, but unfortunately, not with equal probability.

To see why, consider the small interval $[0, 1]$: the proposed conversion becomes `(int)urand()`, and that almost always evaluates to exactly zero.

The first hack that programmers then make is to rewrite the conversion as `m + (int)((n - m + 1)*urand())`, fudging the endpoint by one. That form gets the right answer for the two cited generator intervals, but erroneously produces results on $[m, n + 1]$ if the generator interval is either $(0, 1]$ or $[0, 1]$, and the endpoint $n + 1$ is produced only rarely (about once in $2^{32} \approx 4 \times 10^9$ times for a 32-bit generator).

We pointed out earlier that it is regrettably common for the endpoint conditions of the floating-point generator to be undocumented, so neither of those attempts produces *portable* and *correct* code.

As we saw in the nonportable versions of `rand2()` through `rand4()`, the proper solution of that problem requires some of the random floating-point values to be discarded. Because of the generator endpoint uncertainty, we produce numbers for an extended interval, and then discard out-of-range results. Here is a function that implements the algorithm:

```
int
urandtoint(int lo, int hi)
{   /* return random integer in [lo, hi] */
    double ext_range;
    int k, result;
    static const double MAXINT = (double)(1L << (DBL_MANT_DIG / 2)) *
                                 (double)(1L << ((DBL_MANT_DIG + 1) / 2));

    if (lo >= hi)                       /* sanity check */
        result = lo;
    else if ( (hi < -MAXINT) || (MAXINT < hi) ) /* sanity check */
        result = lo;
    else
    {
        ext_range = (double)hi - (double)lo + 2.0;

        if (ext_range > MAXINT)         /* sanity check */
            result = lo;
        else
        {
            do
            {
                k = lo - 1 + (int)floor(urand() * ext_range);
            } while ( (k < lo) || (hi < k) );

            result = k;
        }
    }
    return (result);
}
```

In the inner loop, it is critical that the reduction to an integer value from a floating-point value be done with the `floor()` function, which always produces a result that does not exceed its argument. If the `trunc()` function were used, and the interval [`lo`, `hi`] contained zero, then values in the range $(-1, +1)$ would all be reduced to zero, making that result twice as likely as any other. That would violate the contract that the function returns each possible integer value with equal probability. That is an issue in translation to other programming languages, most of which lack the `floor()` function.

7.5 Random integers from an integer generator

If we have a source of uniformly distributed random integers, we can produce random subsets of those numbers in a specified interval [`lo`, `hi`], but once again, care is needed.

If the range of the results, plus one, is 2^k, then we can take the k low-order bits as an integer, add it to `lo` and get the desired result.

If the range is not of the form $2^k - 1$, however, we cannot replace the bit masking by a modulo operation and get equally distributed results. To see why, consider a 5-bit generator used to produce integers in the interval $[0, 4]$, and compute all possible moduli with a short hoc program:

```
hoc> for (k = 0; k < 32; ++k) printf("%d ", k % 5);
hoc> printf("\n");

0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1
```

Examination of the resulting digit frequencies shows that 0 and 1 occur once more than the other digits. That must be the case, because 5 is not a divisor of 32: we simply cannot partition 32 things into 5 equal subsets.

One solution is to compute random integers in the next larger interval of size $2^k + 1$, add that integer to `lo`, and then try again if the result is out of range.

To do that, we first need a way to find the smallest value 2^k that is at least as large as the range to be selected from. We could do that with an expression like `(int)ceil(log2((double)(hi - lo + 1)))`, but that is comparatively costly, and it discards most of the bits of the computed logarithm. There is a much better and faster way, however, shown in *Hacker's Delight* [War03, War13], a treasure trove of tricks with integer arithmetic:

```
unsigned int
clp2(unsigned int n)
{   /* return 2**ceil(log2(n)) (assume wordsize in [16,64]) */
    n--;
    n |= n >> 1;
    n |= n >> 2;
    n |= n >> 4;
    n |= n >> 8;

#if UINT_MAX > 0xffff
    n |= n >> 16;
#endif

#if UINT_MAX > 0xffffffff
    n |= n >> 32;
#endif

    return (n + 1);
}
```

That function, and variants `lclp2()` and `llclp2()` for the longer integer types, are included in the `mathcw` library. There are also companion function families `flp2()`, `nlz()`, `ntz()`, and `pop()` to compute the nearest power of two below the argument, the number of leading and trailing 0 bits, and the population count (the number of 1 bits). On most architectures, their jobs can be done without loops or calls to other functions, in a time that is largely data independent. A few CPU designs have single instructions that can be used to replace the bodies of some of the five function families, making them even faster. Analysis of the optimized assembly code produced by compilers on eleven current and historical CPU architectures shows that `clp2()` takes between one dozen and two dozen instructions, and the most complex of them, `nlz()`, takes just over twice as many.

The task in `clp2()` is to replicate the highest 1 bit into lower-order positions, creating a mask equivalent to $2^k - 1$, and then to add one to the final result to recover 2^k. The first shift-OR operation copies the highest 1 bit into the next lower position, so that the result, after any leading string of 0 bits, begins with two 1 bits. The second shift-OR copies those two 1 bits, producing a string of four 1 bits. Each step doubles the number of 1 bits, until bits are lost in the shift, and that continues until the last shift-OR copies the remaining half word. The code works on unsigned integers, so it does not depend on how negative numbers are represented (see **Appendix I** on page 969). We have to include enough shift-OR operations to handle every word size in common use, but avoid shifts larger than the word size, which are undefined according to ISO Standard C. Our code handles all word sizes from 16 to 64.

The questionable cases are those for the smallest and largest arguments. With a zero argument, the initial subtraction turns `n` into a word of 1 bits, and the shift-OR operations leave the value unchanged. The final addition overflows to a zero return value, which we can regard as the definition of `clp2(0)`. With any argument above `0x8000 0000`, assuming a 32-bit word, the leading bit is 1, the shift-OR operations fill the word with 1 bits, and the final addition again overflows to zero.

Here is a function that uses `clp2()` to implement an algorithm for selecting random integers in a given range:

```
static int
irandtoint(int lo, int hi)
{   /* return random integer in [lo, hi] */
    int k, mask, n, result;

    if (lo >= hi)
        result = lo;              /* sanity check */
    else
    {
        k = 0;
        n = (hi - lo) + 1;        /* possible integer overflow here! */
        mask = (int)clp2(n) - 1;

        do
        {
            k = uirand() & mask;
        } while (k >= n);

        result = lo + k;
    }

    return (result);            /* random integer in [lo, hi] */
}
```

In the worst case, when the requested range is $\lfloor(2^k + 1)/2\rfloor$, about half of the results from `uirand()` are discarded.

As the comment on the assignment to `n` notes, if the arguments are large and their range contains zero, there could an undetected integer overflow.

Here is a simple test of the `irandtoint()` function, producing five runs of 20 values:

```
% cc -DMAXSHOW=20 irnint.c -L.. -lmcw && ./a.out
Test of generating random integers on [10,99]
10 32 49 96 62 83 87 38 51 10 40 59 38 52 86 40 93 68 12 53
34 58 79 24 20 88 40 52 83 61 38 25 65 18 90 78 20 66 10 16
36 10 65 56 97 10 24 73 77 82 76 80 67 17 98 86 19 33 88 43
46 29 51 48 76 31 94 53 86 67 98 51 30 83 74 42 10 81 88 33
65 51 74 93 64 62 56 83 67 86 34 72 36 65 58 99 51 41 48 65
```

As we intended, each sequence contains unordered integers on the interval $[10, 99]$, and each differs from the other sequences.

As suggested in *Practical Cryptography* [FS03, page 184], we can largely eliminate the problem of wasting half our random integers in the worst case with an alternate version of the code. If we ensure that the power of two is above the largest representable multiple of the range, then discards are rare. For example, with a 32-bit generator and a range of 10, the largest multiple of that range is $2^{32} - 6 = 4\,294\,967\,290$. We then only need to discard the five integers above that value, and otherwise, we can take the remainder with respect to 10 to get values uniformly distributed on $[0, 9]$. Here is a version of our code that implements the idea:

```
static int
irandtoint_new(int lo, int hi)
{   /* return random integer in [lo, hi] */
    unsigned int k, n, nq, q;
    int result;

    if (lo >= hi)
        result = lo;      /* sanity check */
    else
    {
        n = (hi - lo) + 1;  /* need random integers in [0, n - 1] */

        if (n == 0)         /* had integer overflow */
            result = lo;    /* sanity check */
        else
        {
            k = 0;          /* keep optimizers happy */
            q = RAND_HI / n;
            nq = n * q;     /* large multiple of n */

            for (;;)
            {
                k = uirand();

                if (k < nq) /* almost always true */
                {
                    k %= n; /* random integer in [0, n - 1] */
                    break;
                }
            }

            result = lo + k;
        }
    }

    return (result);    /* random integer in [lo, hi] */
}
```

The code no longer requires `clp2()`, but needs an extra integer division and an integer remainder. The removal of that function call, and as few as half the calls to `uirand()`, suggest that the new version could be about twice as fast as the original. The problem of undetected integer overflow remains: the output in such a case fails an important test suite (see **Section 7.15.2** on page 200), whereas the output passes the tests when there is no overflow.

7.6 Random integers in ascending order

The next application of random numbers that we consider is the production of random integers in sorted order. Of course, one can always produce a stream of random integers and then use a suitable sort library function in memory, or a sort utility on a file: the good ones run in $\mathcal{O}(n \log_2 n)$ time for n objects.

AT&T Bell Labs researchers [BS80] discovered a clever way to avoid the sort, and get the results in nearly linear time. A program that implements their algorithm looks like this:

```
void
select(int m, int n)
{   /* output m random integers from [0,n) in ascending order */
    int i, mleft, remaining;

    mleft = m;
    remaining = n;

    for (i = 0; i < n; ++i)
    {
        if ((uirand() % remaining) < mleft)
        {
            (void)printf("%2d ", i);
            mleft--;
        }
        remaining--;
    }
    (void)printf("\n");
}
```

Here is a demonstration of `select()` in action, with five experiments:

```
for (k = 1; k <= 5; ++k) select(20, 100);
 0  7 10 14 32 39 45 48 51 58 61 64 66 67 81 85 88 90 95 96
 0  3  4  5  6  8 11 13 24 33 39 62 68 70 73 78 79 85 86 97
 4  6 10 11 12 20 25 28 30 41 42 43 47 49 54 57 72 79 82 83
 0 11 18 21 24 39 42 49 51 57 58 59 62 70 72 78 83 91 95 98
21 22 25 26 48 49 50 53 54 55 57 59 65 71 74 77 80 84 89 97
```

Each experiment produces a different selection, and each output list is in ascending order. Although the sequences are no longer random, because of their increasing order, their individual members are nevertheless chosen randomly.

7.7 How random numbers are generated

In this section, we look at how streams of random numbers can be produced by computer algorithms. We consider several common techniques, showing how random numbers are computed with them, and why some of the commonly used methods are deficient.

7.7.1 Linear congruential generators

The oldest, and most widely used, algorithm for generation of random numbers is called a *linear congruential generator* (*LCG*). It was introduced in a classified paper by I. J. Good in 1948 [Goo69], and independently, and more widely known, by D. H. Lehmer in 1949 [Leh51]. It takes the form

$$x_{n+1} = (Ax_n + C) \bmod M, \qquad\qquad x_0 = \textit{initial seed, and } n = 0, 1, 2, \ldots,$$

with these constraints on the *unsigned integer* parameters:

$$A > 1, \quad C \geq 0, \quad M > 1, \quad x_0 = \text{ initial seed}, \quad 0 \leq x_n < M.$$

The mod operator means *modulus*: for integers $p \geq 0$ and $q > 0$, the expression $r = p \bmod q$ is the nonnegative integer *remainder* when p is divided by q. Thus, $p = nq + r$, where $n = \operatorname{floor}(p/q) \geq 0$, and $0 \leq r < q$.

The right-hand side operations in the LCG must be carried out in *exact* arithmetic, a requirement that we return to in Section 7.7.3 on page 171.

The special case $C = 0$ is called a *multiplicative congruential generator* (*MCG*), in which case, the last constraint must be changed to $0 < x_{n-1} < M$; otherwise, the generator would produce only zero from a zero seed.

The modulo operation means that the range of the LCG is $[0, M-1]$, and that of the MCG is $[1, M-1]$, but the period depends on the choice of parameters. From number theory, it has been proved that the maximum period of M for the LCG, or $M - 1$ for the MCG, can be reached when these three conditions are satisfied:

- C and M are *relatively prime* (i.e., have no prime[1] factors in common);
- $A - 1$ is a multiple of p for every prime number p that divides M;
- $A - 1$ is a multiple of 4 if M is a multiple of 4.

When $M = 2^n$, the generator computation can sometimes be simplified. The maximum period is then $M/4$, and it can be reached when A mod 8 is 3 or 5.

It is not necessary to remember those rules, and choose one's own parameter values, because the LCG and MCG have received intensive study, and many good choices of the parameters have been found and published (see [FM82, FM86a, PM88, LBC93, KW96, DH97b, Wu97, DH00, Tan06, LS07, TC11] and [Knu97, page 106]). However, exceptionally good parameter values are not likely to be found by chance: an exhaustive study for the case $M = 2^{31} - 1$, a modulus that is convenient for use with signed-integer arithmetic, found fewer than one acceptable (A, C) pair for every million pairs examined [FM86a].

Here is an example of a small LCG whose parameters are chosen according to the listed rules:

```
hoc> A = 32; C = 19; M = 31; x = 0
hoc> for (k = 1; k <= 72; ++k) { x = (A*x + C) % M; printf("%2d ", x) }
19  7 26 14  2 21  9 28 16  4 23 11 30 18  6 25 13  1 20  8 27 15  3 22
10 29 17  5 24 12  0
                     19  7 26 14  2 21  9 28 16  4 23 11 30 18  6 25 13
 1 20  8 27 15  3 22 10 29 17  5 24 12  0
                                          19  7 26 14  2 21  9 28 16  4
```

The period is $M = 31$, and each number in $[0, 30]$ appears exactly once before the sequence repeats, as indicated by the indented lines in the output.

7.7.2 Deficiencies of congruential generators

There are several problems with congruential generators:

- Good parameters are hard to find [FM86a].
- The modulus value M determines the maximum period, but arithmetic considerations limit its practical size to values that are much too small for the requirements of many modern applications. Some limited investigations of large M values may provide a solution to that problem [LBC93, DH97a, DH97b, Wu97, DH00, ESU01, TC11].
- Moduli of the form $M = 2^p$ are commonly chosen to simplify computation, but the sequences produced often have obvious correlations.
- The low-order bits in the output numbers are less random than the high-order bits.
- Extra precision may be needed to form $Ax + C$ exactly, and if that precision is not available in hardware, then it must be simulated in software, making the programming more complex, and reducing the speed of the generator.
- In an important paper, *Random Numbers Fall Mainly in the Planes* [Mar68], George Marsaglia gives a mathematical proof that random numbers from *all* LCGs and MCGs have strong correlations between values produced at certain fixed intervals in the generator cycles.

The last of those is the most serious, and it is instructive to examine the problem graphically. A common application of random-number generators is to produce uniformly distributed samples of points in an n-dimensional space. Thus, in two dimensions, successive pairs of random numbers define (x, y) coordinates in the plane, and one would expect them to cover the unit square fairly evenly, as shown in **Figure 7.1**(a). With a different set of parameters, one can instead get the strongly correlated points plotted in **Figure 7.1**(b). The two plots have the *same* number of coordinate pairs, but in the second one, many pairs fall at each plotted location.

[1] Prime numbers are integers that cannot be divided, without a remainder, by any integer other than 1 and themselves. The first few primes are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, ...

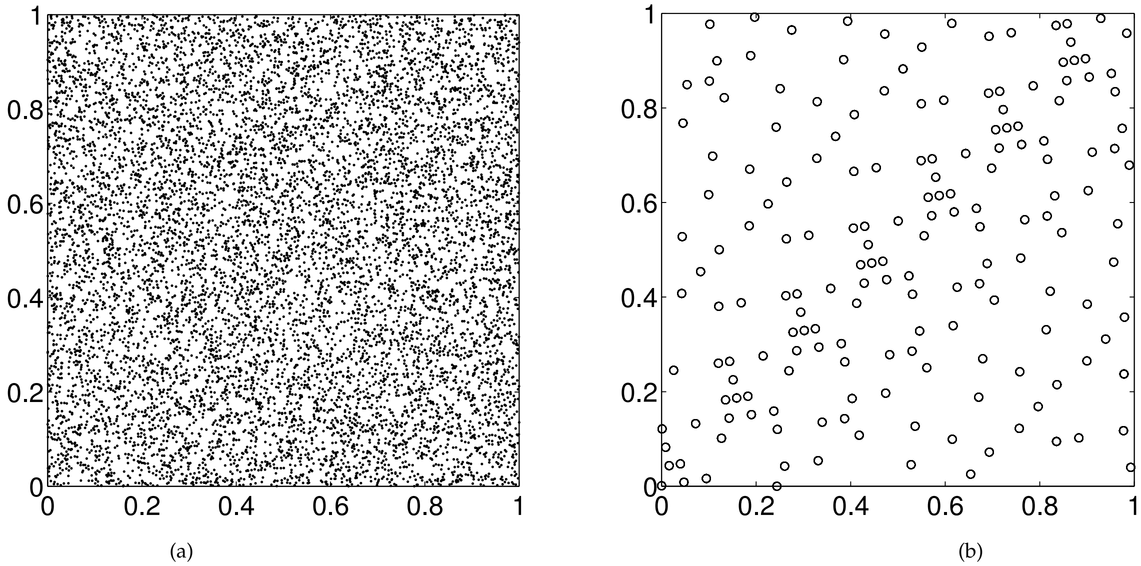


Figure 7.1: The correlation problem in linear congruential generators. Each plot shows the distribution of 10 000 (x, y) points determined by successive random values from a generator. Points in plot (a) are from a good generator, MATLAB's `rand()`. Those in plot (b) are from a bad LCG; its parameters are $A = 2^7 - 1$, $C = 2^5 - 1$, and $M = 2^{15} - 1$. The period of that LCG is only 175. Notice that many of the points fall along diagonal lines, and others form peculiar repeating patterns, like the 'bird' flying along the center diagonal.

Those effects are harder to see in three (or higher) dimensions, because there are many more possible views of the data. **Figure 7.2** on the following page shows several frames from an animation that demonstrates the problem: at certain view angles, the (x, y, z) points are seen to lie on planes, rather than being spread out fairly uniformly through the unit cube. In that case, coordinates are collected every 105 cycles of the generator. If instead they are collected every 104 or every 106 cycles, the correlations are not evident: see **Figure 7.3** on page 173.

The correlations in congruential generators in practice mean that a generator may be in wide use for a long time with apparently satisfactory behavior, until one day, a user happens to stumble on a cycle step size where the correlations suddenly become evident. That happened to the infamous `randu()` generator introduced in the early 1960s in the IBM *Scientific Subroutine Package*, and widely used for several years on other systems as well. It has parameters $A = 2^{16} + 3$, $C = 0$, $M = 2^{31}$, and $x_0 = 1$. *The unavoidable correlations of random-number sequences from congruential generators strongly suggest that such generators should not be used at all.*

7.7.3 Computing congruential generators

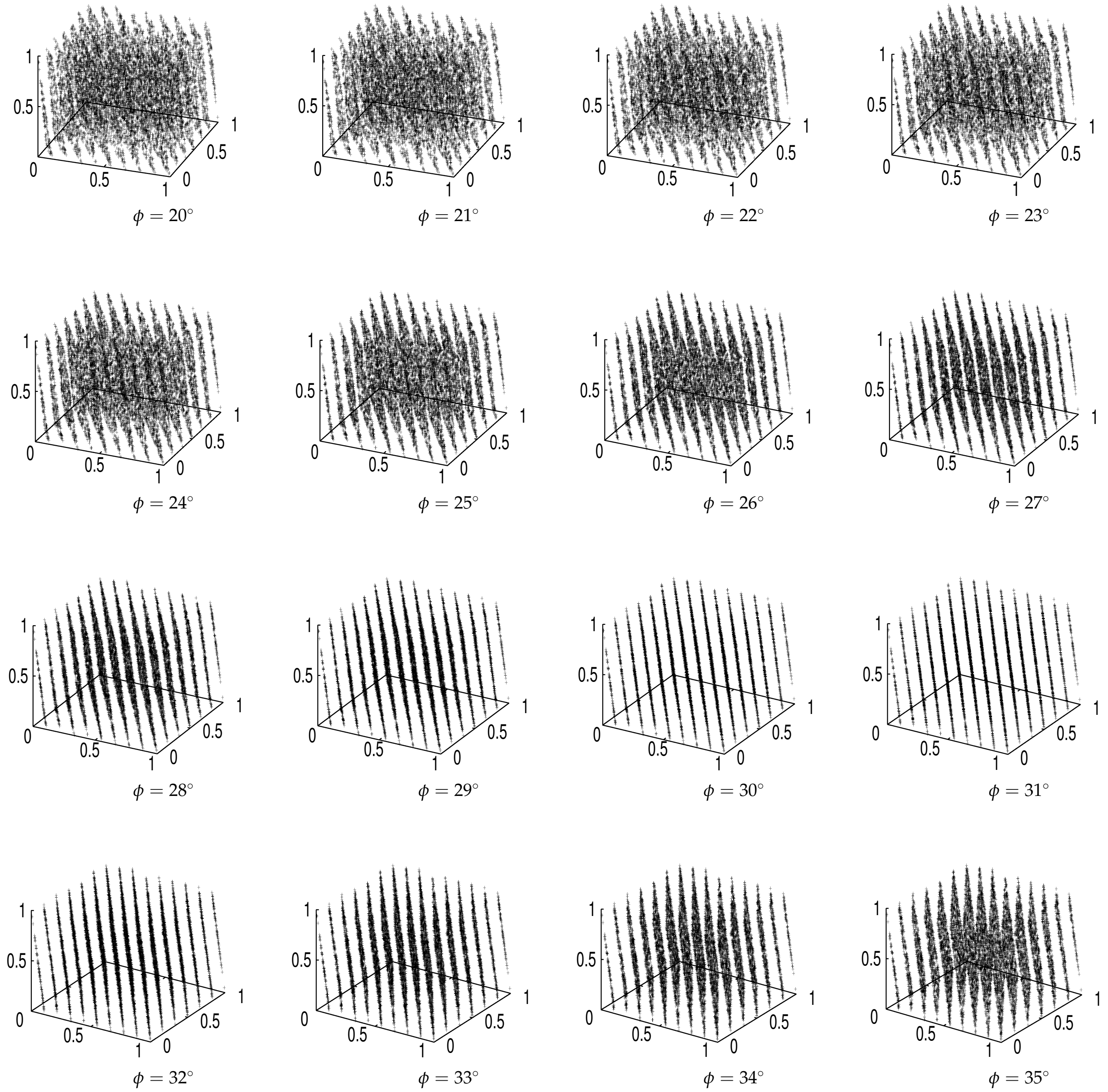
The congruential generator algorithm

$$x_{n+1} = (Ax_n + C) \bmod M, \qquad n = 0, 1, 2, \ldots,$$

looks innocuous, requiring only a multiply, an add, and a remainder operation. The difficulty is that the computation must be *exact*.

In the special case that $M = 2^{\text{wordsize}}$, and if $Ax_n + C$ can be computed in exact double-length unsigned-integer arithmetic, the modulo operation is equivalent to dropping the high-order word of the result. For that reason, many of the early recommendations for parameter sets for congruential generators made that restriction on the modulus M, even though it reduces the maximum period to $M/4$.

The number of bits required for the multiply is the sum of the number of bits in the operands, and the add may require one more than the number of bits in the larger operand. The modulo operation needs as many bits as its larger operand. If we choose, for example, a 16-bit A and 31-bit C and M, the parameters can be stored in 32-bit signed integers on a typical desktop computer. The computation requires 48 bits for the product and sum, but



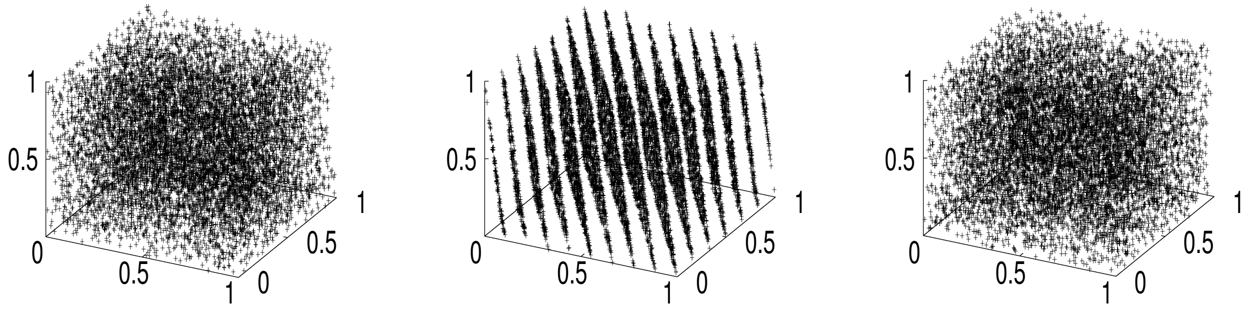
$\phi = 20^\circ$ $\phi = 21^\circ$ $\phi = 22^\circ$ $\phi = 23^\circ$

$\phi = 24^\circ$ $\phi = 25^\circ$ $\phi = 26^\circ$ $\phi = 27^\circ$

$\phi = 28^\circ$ $\phi = 29^\circ$ $\phi = 30^\circ$ $\phi = 31^\circ$

$\phi = 32^\circ$ $\phi = 33^\circ$ $\phi = 34^\circ$ $\phi = 35^\circ$

Figure 7.2: Frames from animated rotation of LCG 3-D point sequence. The Greek letter *phi*, ϕ, is the angle of rotation in degrees. The generator is $x_n = (2\,396\,548\,189 x_{n-1} + 0) \bmod 2^{32}$, with samples taken every 105 steps. Each plot has 10 240 points. See Entacher [Ent98] for that example, and many others.



steps = 104 steps = 105 steps = 106

Figure 7.3: Frames from animated rotation of LCG 3-D point sequence ($\phi = 27^\circ$). The generator is the same as in Figure 7.2, but samples are taken at varying numbers of steps.

the hardware often supports only 32-bit arithmetic. However, recall that IEEE 754 64-bit arithmetic provides 53-bit precision, so the generator can be evaluated exactly in that arithmetic by code like this:

```
double x, A, C, M;
x = fmod(A*x + C, M);
```

Many publications of LCG and MCG parameters limit their values precisely to allow simple computation in double-precision floating-point arithmetic.

If integer arithmetic is used, it may still be possible to use a one-line computation, if extended precision integer data types are supported, such as with Fortran `INTEGER*8`, C/C++ `unsigned long long int`, or Java `long`. However, except for Java, such extended precision is not universally available, and thus, cannot be used in portable software. C99 requires support for such data, but compilers for that language level are not available on some systems at the time of writing this.

Without access to a package for general multiple-precision integer arithmetic, the only choice left is to split the computation into parts that can be done within the limits of normal integer arithmetic. We can then take advantage of two properties of modular arithmetic:

$$(A + B) \bmod M = ((A \bmod M) + (B \bmod M)) \bmod M,$$
$$(A \times B) \bmod M = ((A \bmod M) \times (B \bmod M)) \bmod M.$$

The operands are then split into three parts, each with half as many bits:

$$A = a_1 a_2 + a_3,$$
$$B = b_1 b_2 + b_3.$$

The split reduces $Ax \bmod M$ to $(a_1 x \bmod M)a_2 \bmod M$ without integer overflow if x is small enough. Otherwise, we must split x as well, and get an even more complicated expression.

Here are two useful functions for modular arithmetic that implement splitting of each operand:

```
static const double MOD_SCALE = 8192.0 * 8192.0; /* 2**26 */
static const double MOD_A2_B2 = 8192.0 * 8192.0 * 8192.0 * 8192.0;
                                                 /* 2**52 */
double
mod_add (double A, double B, double M)
{   /* return exact (A + B) mod M */
```

```
    double a1, a2, a3, b1, b2, b3;

    a1 = floor(A / MOD_SCALE);
    a2 = MOD_SCALE;
    a3 = A - a1 * a2;      /* Now A = a1*a2 + a3 */

    b1 = floor(B / MOD_SCALE);
    b2 = MOD_SCALE;
    b3 = B - b1 * b2;      /* Now B = b1*b2 + b3 */

    return (fmod(fmod(fmod(a1 * a2, M) + fmod(b1 * b2, M), M) +
            fmod(a3 + b3, M), M));
}


double
mod_mul (double A, double B, double M)
{   /* return exact (A * B) mod M */
    double a1, a2, a3, b1, b2, b3;

    a1 = floor(A / MOD_SCALE);
    a2 = MOD_SCALE;
    a3 = A - a1 * a2;      /* Now A = a1*a2 + a3 */

    b1 = floor(B / MOD_SCALE);
    b2 = MOD_SCALE;
    b3 = B - b1 * b2;      /* Now B = b1*b2 + b3 */

    /* NB: multiplication by a2 and b2 is an exact scaling */
    return (mod_add(mod_add(mod_add(fmod((a1 * b1) * MOD_A2_B2, M), fmod((a3 * b1) * b2, M), M),
                    fmod((a1 * b3) * a2, M), M), fmod(a3 * b3, M), M));
}
```

The modular-multiplication function uses the modular-addition function to collect intermediate terms, and in both functions, the code is much more complex than the simple mathematical expressions that they compute.

With those two functions, as long as the operands are whole numbers that are exactly representable in fewer than 53 bits as type `double`, the linear congruential generator operation $(Ax + C) \bmod M$ can be done exactly with code like this:

```
seed = mod_add(mod_mul(A, seed, M), C, M);
```

The code for the multiplicative congruential generator is even simpler:

```
seed = mod_mul(A, seed, M);
```

In each case, the old seed is the input x value, and the new seed is returned as the next random number. The variable `seed` must be retained across calls to the generator, normally by declaring it with the `static` modifier.

7.7.4 Faster congruential generators

Apart from the need for extended precision, the modulo operations are the computational bottleneck in linear and multiplicative congruential generators. For certain choices of the modulus M, it is possible to replace the expensive divide by faster operations [Sch79b], and we show here how to do so. The conditions on the parameters A, C, and M that we gave earlier suggest that M should be a prime number near the largest representable integer, where that limit is $2^p - 1$ for a p-bit word. We can find candidates for M like this:

```
% maple
> for m from 1 to 59 by 2 do
>     printf("%2d: ", m):
```

```
>     for p from 1 to 64 do
>         if isprime(2^p - m) then printf("%2d ", p) end if
>     end do:
>     printf("\n")
> end do:
 1:  2  3  5  7 13 17 19 31 61
 3:  3  4  5  6  9 10 12 14 20 22 24 29
 5:  3  4  6  8 10 12 18 20 26 32 36 56
...
25:  5  7  9 13 33 37 57 63
...
31:  7 11 13 17 19 35 37 41 61
...
59:  6  8 20 48 64
```

The choices $M = 2^{31} - 1$ and $M = 2^{32} - 5$ are suitable for signed and unsigned 32-bit arithmetic. For 36-bit arithmetic on several historical machines, use $M = 2^{35} - 31$ or $M = 2^{36} - 5$. On newer 64-bit architectures, pick $M = 2^{63} - 25$ or $M = 2^{64} - 59$.

To simplify things, let us assume that the parameters and integer data type are chosen such that $N = Ax + C$ is exactly representable. We need to find the remainder $x = N \bmod M$, which simply means that $N = mM + x$, where $m = \lfloor N/M \rfloor$. To avoid the divide, we instead proceed like this:

$$\begin{aligned}
K &= 2^p, && \textit{one larger than maximum integer,} \\
&= M + d, && \textit{for } d = 1, 2, 3, \ldots, \\
y &= N \bmod K, && \textit{modulus to be related to } x, \\
&= \operatorname{and}(N, K - 1), && \textit{fast computation of } y, \\
k &= \lfloor N/K \rfloor, && \textit{by } p\textit{-bit right shift or word extraction, } \text{not } \textit{division,} \\
N &= kK + y, && \textit{for } y = 0, 1, 2, \ldots, K - 1, \\
r &= y + kd, && \textit{initial estimate for } x, \\
&= (N - kK) + kd && \\
&= N - k(K - d) && \\
&= N - kM, && \textit{form needed for } x.
\end{aligned}$$

Because $M \approx K$, we expect that $m \approx k$, and also that $m \geq k$. If r lies in $[0, M)$, then we conclude that $k = m$ and $x = r$. Otherwise, we increase k by 1, reducing r by M. If that new value lies in $[0, M)$, then we have found x without division.

The paper [Sch79b, page 173] proves that the condition for the initial r to lie in $[0, 2M)$ when $C = 0$, and thus needing at most one downward step, is

$$A < \frac{K(M - d + 1)}{dM} = K\left(\frac{1}{d} - \frac{1}{M} + \frac{1}{dM}\right).$$

In particular, for the choice $M = 2^{31} - 1$, we have $d = 1$, and the condition reduces to $A < K$, which is satisfied if we ensure that $A \leq M$. For large M and $d \ll M$, the condition is approximately $A < M/d$. That is unlikely to be a problem, because the preferred A values are often much smaller than M.

If the condition is not satisfied, then we could use a loop to reduce r by M in each iteration until we find an acceptable value for x. However, it is then probably better to choose different LCG parameters.

The alternate route to the new random value x lets us replace the normal code

```
m = N / M;                  /* N = A * x + C; M = modulus of LCG */
x = N - m * M;              /* x = N mod M = next random number */
```

that requires a slow integer division with new code that uses bit operations:

```
k = N >> p;                 /* k = floor(N / K); K = 2**p */
```

```
x = (N & (K - 1)) + k * d;  /* x = N mod K + k * d = trial remainder */

if (x >= M)                 /* check that x is in [0, M) */
    x -= M;                 /* otherwise, force x into [0, M) */
```

If p is chosen to be the number of bits in a word, then the right shift might be replaced by a faster extraction of the high part of the double word N. Tests on three dozen systems show that an LCG with the new code is on average 2.0 to 2.9 times faster, with a largest speedup of 9.3 times.

The technique of replacing slow integer divisions by fast bit operations is of interest for optimizing compilers, and some can now do so for particular constant divisors. Chapter 10 of *Hacker's Delight* [War03, War13] provides much detail, and has been extended with a lengthy online supplement,[2] but the treatment is mostly for small divisors. Recent research [CW08, CW11] has generalized the idea to arbitrary integers, and demonstrates that integer division can be replaced by faster code for about 80% of possible compile-time constant divisors.

7.7.5 Other generator algorithms

We described linear and multiplicative congruential generators in detail, because they have been widely used in the past, and they are among the easiest to compute. Despite their simple mathematical formulas, we found that limitations of hardware arithmetic require that more care be taken in their computation. However, the unavoidable correlations in their random-number sequences suggest that we look for other methods.

The first possible improvement that one might think of is to make the next random number depend quadratically on the seed:

$$x_{n+1} = (Ax_n^2 + Bx_n + C) \bmod M.$$

That is called a *quadratic congruential generator* (QCG), and it has been shown that its period cannot exceed M. That is no better than the period of a linear congruential generator, so quadratic generators are of little interest, although they do eliminate the correlations of LCGs and MCGs.

Another approach is to make the next random number depend linearly on several preceding ones:

$$x_{n+1} = (A_1x_n + A_2x_{n-1} + \cdots + A_kx_{n-k+1} + C) \bmod M.$$

That is called a *multiple recursive congruential generator* (MRCG), and if M is prime, with suitable conditions on the other parameters, the period can be $M^k - 1$, which is a great improvement. Good parameters for two- and many-term MRCGs have been published [Tan06, DSL12a], but some of the recommended parameter choices lead to serious correlation problems [LS14].

Yet another variant is to drop A and C entirely, and instead depend on two earlier seeds spaced a fixed distance apart in the sequence:

$$x_{n+1} = (x_{n-p} + x_{n-q}) \bmod M, \qquad \textit{for fixed integers } q > p.$$

That is called a *lagged Fibonacci generator*. At startup, the generator has to be run for several cycles to generate the two needed historical seeds. Alternatively, those seeds could be provided by the user, but that has recently been shown to be perilous [MWKA07]. If M is prime, with suitable conditions on the other parameters, the period can be $M^q - 1$. In particular, the period can be made *enormous*, while keeping a fast kernel that needs only an addition and a modulo operation. However, if p and q are small, it is a poor generator.

Still another variant, invented by a distinguished researcher in the field, George Marsaglia, uses a combination of exclusive-OR and shift operations, with the computation expressible in just three lines of C/C++ code [Mar03b]:

```
x ^= x << a;
x ^= x >> b;
x ^= x >> c;
```

Java code is similar: just replace the >> operator by Java's unsigned right-shift operator >>>. Here, x is the initial seed and the final result, and the shift values a, b, and c are known constants. The generator is extremely fast, requiring just six instruction cycles on IA-64, and there is no expensive modulo operation. Some of Marsaglia's proposed sets of shift values produce poor generator-test results and exhibit strong correlations, but there are many good sets that

[2] See http://www.hackersdelight.org/divcMore.pdf.

allow the method to provide a family of hundreds to thousands of distinct generator streams. The generator range is $[1, 2^{\text{wordsize}} - 1]$, and the period is $2^{\text{wordsize}} - 1$. With a few changes to the kernel computation, the period can be pushed to $2^{192} - 2^{32} \approx 10^{57}$ in 32-bit integer arithmetic.

7.7.6 Combined generators

It is sometimes possible to combine two different generator algorithms to get a new generator whose period is the *product* of the periods of its components. One such generator that has performed well on extensive tests is the *KISS generator*, which combines an LCG with an exclusive-OR-shift generator. Although it passes most test suites, and it has a period of about $2^{123} \approx 10^{37}$, its range has not been published. It appears to be excellent for sampling and simulation, but *not* for cryptography, because it is possible to recover its parameters from as few as 70 output values [Ros11].

Instrumentation and testing of the KISS generator with additional code in `tstkis.c` to track and report the current extreme values each time they change demonstrates that the range is $[0, 2^{32} - 1]$. With the default seed, the minimum value is reached after about 2.17×10^9 calls, but the maximum value does not appear until about 10.28×10^9 calls, equivalent to about 2.4 cycles through all possible 32-bit values. Experiments with a few other initial seeds show both long and short cycle counts: the seed `0xffff ffff` produces extremal values after 0.96 cycles, but a zero seed requires 3.83 cycles. On the fastest machine tested, a 3 GHz AMD64 processor, our scalar implementation of KISS produces one result every 14.7 nsec, and the code from an optimized compilation needs only 37 CPU instructions.

The widely used Wichmann–Hill generator [WH82] combines the results of three small MCGs that can be safely evaluated in 16-bit integer arithmetic. It has since been extended for 32-bit integers with four small MCGs, and passes major test suites [WH06].

The *combined Tausworthe generator* [L'E96] passes important test suites. It uses a complex sequence of shift and exclusive-OR operations, with a range of $[0, 2^{32} - 1]$ and a period of $2^{88} \approx 10^{26}$.

One of the best current generators is known as the *Mersenne Twister* [MN98], provided that code improvements made by its authors after their article was published are incorporated. Its kernel operations include shifts, exclusive-OR, and logical-AND, but optimized to maintain high performance. It passes almost all of the tests discussed in **Section 7.15** on page 196. The Mersenne Twister has a modest range of $[0, 2^{32} - 1]$, but it has an enormous period of $2^{19937} \approx 10^{6001}$, a number that is too large to represent even in 128-bit IEEE 754 binary arithmetic.

George Marsaglia posted a compact generator based on a multiply-with-carry algorithm, called *MWC1038*. Its kernel requires only a single multiply, add, and shift operation in 64-bit integer arithmetic. Its range is unspecified, and its period is gigantic: $3\,056\,868\,392^{33\,216} - 1 \approx 10^{315\,063}$. The generator state is a vector of 1038 random integers, plus one extra integer. MWC1038 passes most test suites, but its need for 64-bit integer arithmetic destroys portability, unless that arithmetic can be supplied in software, as we do with the function `umul64()` in **Section 9.3** on page 261.

Marsaglia and co-workers introduced a related generator family that uses an algorithm called add-with-carry or subtract-with-borrow [Jam90, MNZ90, MZ91]. It requires no higher intermediate precision, and is fast. Unfortunately, tests show that its initial output is far from random unless its seed array is thoroughly mixed, failing one of our original design criteria. Also, the algorithm has been proved to exhibit correlations in high dimensions [TLC93, EH95].

In 1916, the famous mathematician and theoretical physicist Hermann Weyl found that, given an irrational number x, such as $\sqrt{2}$ or π, the fractional parts of the sequence nx, denoted as $\{nx\}$, for $n = 1, 2, 3, \ldots$, are uniform on $(0, 1)$. They are easily computed with `fmod(n * x, 1.0)`, or `modf(n * x, NULL)`, or $nx - \operatorname{floor}(nx)$. Unfortunately, they are also correlated. Decades later, physics researchers showed that the nested Weyl sequence $\{n\{nx\}\}$ seems to eliminate the correlations [HPWW94, TW99]. Combined with a technique that we describe in **Section 7.9** on the following page, they produce a satisfactory generator that passes important test suites. Importantly, its period is *effectively infinite*, so picking widely separated starting values of n, or different starting irrational numbers, makes it easy to generate multiple independent and nonoverlapping streams of uniformly distributed floating-point values. If the starting irrational constant is fixed, the seed can be as simple as the single number n, and it can even be the index in a buffer of computed random numbers that is recycled when the buffer is refilled. Alternatively, the seed can be the most recent random value, with the sequence index n maintained internally. The drawback is that the use of floating-point arithmetic makes it unlikely that the same sequence can be produced on other systems with different floating-point chip models or designs.

Shift-register generators have been investigated by several authors, and pass extensive test suites. Ripley's `srg521()` code [Rip90] is a sample implementation that needs somewhat complex state initialization from a single seed, but has a fast generator in just four simple statements:

```
result = w[p];
w[p] = result ^ w[q];
if (--p < 0) p = MAX_W - 1;
if (--q < 0) q = MAX_W - 1;
```

They retrieve the next random number, update another, and then adjust the two staggered buffer indexes so that they cycle through the buffer. The size of the buffer `w[]` is 521, and that buffer is the entire state of the generator. The generator period is $2^{521} \approx 6.86 \times 10^{156}$, long enough that different starting seeds should give independent nonoverlapping sequences, as long as each maintains its own state. The generator algorithm is independent of word size, so simple type-statement changes can produce code for random integers of any desired size.

7.7.7 Cryptographic generators

The encryption functions from the *Tiny Encryption Algorithm (TEA)*, and its extensions XTEA and XXTEA [WN95, NW97, WN98], can be used as random-number generators. For that purpose, they receive as input two 32-bit unsigned integers (the generator *state*) and output two new ones that can be used as two 32-bit random values, or joined into a single 64-bit random number. Processing consists of iterations (called *rounds* in the cryptographic literature) with a loop body containing shift, exclusive-OR, and addition operations. Cryptographic security is obtained by increasing the number of rounds, and for that purpose, 64 or more are required. For random-number generation, fewer suffice: six rounds are known to change at least half of the bits in the input, and eight rounds produce generators that pass important test suites. The loop body compiles into about twenty instructions on many CPU architectures, and function execution requires as few as two loads and two stores, so operation is fast. Tests on about 40 different CPUs show that eight-round versions of TEA, XTEA, and XXTEA are on average about four times slower than the LCG `lrand48()`, with run-time ratios relative to `lrand48()` from about 0.6 to 16. The periods may be effectively infinite, and the small state makes them suitable for parallel processing: a unique 32-bit starting seed for each process should guarantee distinct and nonoverlapping random-number sequences.

Researchers have investigated the NIST *Advanced Encryption Standard (AES)* as a source of random numbers, and found it to be promising [HW03]. A generator based on the AES, and another that uses the NIST *Secure Hash Algorithm (SHA-1)*, are included in a test suite described later [LS07], and pass extensive tests. With suitable parameter choices, AES generator periods as high as $2^{128} \approx 10^{38}$ are possible. Random-number generation is a requirement of the NIST SHA-3 algorithm [NIS15]. For a comprehensive review of quantum cryptographic generators, see [HCGE17].

7.8 Removing generator bias

If a generator produces a bit stream where bits 0 and 1 are uncorrelated, and occur with *constant*, but *unequal*, probabilities, there is an easy way to repair that bias. If the probability of getting a 0 bit is p, then the probability of getting a 1 bit must be $1 - p$. If we examine successive pairs of bits, then 0 0 occurs with probability p^2, 0 1 and 1 0 each with probability $p(1 - p)$, and 1 1 with probability $(1 - p)^2$. Thus, if we discard the first and last cases, and output the rightmost bit from the pairs 0 1 and 1 0, then the output bits have equal probability.

Because the generator output must be processed two bits at a time, instead of a word at a time, and because we discard roughly six of every eight generated bits, a software solution is slow. However, the processing can be done quickly in hardware, and could be useful in devices that use physical phenomena to produce random numbers, especially because some such devices fail tests of randomness badly.

7.9 Improving a poor random number generator

If you are stuck with a poor random-number generator, there is an easy way to improve it without revising either its algorithm or its parameters: shuffle the order of its results. Bays and Durham [BD76] propose that idea in an article with the same title as this section. They implement it like this for a generator that returns floating-point values uniformly distributed in $[0, 1]$:

- Before starting, save $n+1$ successive random numbers from the generator in an internal buffer, $b_0, b_1, \ldots, b_n$, and set s by another call to the generator.
- To get the next random number, save the next result from the generator in a variable r, select a random index $k = \text{floor}(sn)$, set $s = b_k$, $b_k = r$, and return s.

If the generator returns integer values, rather than floating-point values, then floating-point arithmetic and integer divisions can be avoided in the index computation if we instead choose $n = 2^q$, mask off q adjacent bits in s, and use them for the index. For example, with $q = 6$, we get an index in $[0, 63]$ from `k = (s >> 20) & 0x3f`, avoiding the low-order bits that are known to be less random in an LCG.

A little thought shows that shuffling destroys, or at least lengthens, the period of the generator, because when we reach the start of the next period, we instead swap that value with b_k and return the old b_k. Bays and Durham analyzed their algorithm and found this estimate of its period:

$$P_{\text{shuffle}} \approx \sqrt{\pi n!/(2P_{\text{generator}})}, \qquad n \ll P_{\textit{generator}} \ll n!.$$

Because of the rapid growth of the factorial, the buffer size n does not have to be very large. For a 32-bit generator producing signed integers in $[0, 2^{31} - 1]$, $n = 50$ increases the period by a factor of about $2^{61} \approx 10^{18}$, and $n = 100$ by about $2^{216} \approx 10^{65}$. For a full-period 64-bit signed integer generator, with $n = 75$, the period increases by a factor of $2^{87} \approx 10^{26}$, and with $n = 150$, by $2^{342} \approx 10^{103}$.

It is *imperative* that the saved previous random number s be used in the computation of the index k, because Bays later showed that a slightly different earlier algorithm that uses the new value r to compute k does not lengthen the effective period, and only marginally reduces correlations from an LCG [MM65, LL73, Bay90].

Shuffling is therefore a fast and easy way to remove the problem of too-short periods in commonly used random number generators, and supply periods that are computationally inexhaustible. If the generator produces numbers over the full representable range, shuffling may also provide a significant improvement in test-suite results.

7.10 Why long periods matter

In parallel programs that need independent streams of random numbers, the generator can be a resource bottleneck if it must be shared by all subprocesses or threads. With processes on separate machines, large batches of random numbers need to be sent over the network from a master process. With threads on the same machine, access to the generator must be controlled with suitable locks that delay thread execution.

With linear congruential generators, advancing k steps be done like this, exploiting a property of the mod operation that allows us to remove it from intermediate expressions:

$$\begin{aligned}
x_{n+1} &= (Ax_n + C) \bmod M, \\
x_{n+2} &= (A(Ax_n + C) + C) \bmod M, \\
x_{n+3} &= (A(A(Ax_n + C) + C) + C) \bmod M, \\
\ldots &= \ldots \\
x_{n+k} &= (A^k x + (A^{k-1} + A^{k-2} + \ldots + 1)C) \bmod M \\
&= (A^k x + ((A^k - 1)/(A-1))C) \bmod M \\
&= ((A^k \bmod M)x + ((((A^k - 1)/(A-1))C) \bmod M)) \bmod M \\
&= (A'x_n + C') \bmod M.
\end{aligned}$$

That is, starting a new stream at step k is equivalent to running a different generator with a revised multiplier and additive constant. If k is known in advance, then those values can be worked out once and for all, and stored in tables of starting parameters for increments 0, k, $2k$, $3k$, ... Each process or thread that needs a random number stream is then given a different table index, and each then has a distinct nonoverlapping sequence of random numbers. That technique is sometimes called the *leap-frog method*, and it can be applied to a few other generator algorithms as well. The disadvantage is its lack of applicability to all generators, and that k must be decided in advance, limiting the number of distinct random values available to each process. In many applications, the randomness of the computation makes it impossible to predict the number of random values that might be needed.

A mathematical analysis [Mac92] suggests that one should never use more than $N^{2/3}$ values from a generator, where N is the smaller of the period and the upper limit of the range. With the large processor and thread counts now available to many scientists, LCGs by themselves do not provide sufficiently long independent streams.

However, if the period is large enough, then having each process or thread start with distinct generator seeds makes the probability of sequence overlap vanishingly small, and no special techniques, like the leap-frog method, are needed. Thus, if you must use an LCG, consider combining it with shuffling to expand the period enough to ensure independent streams. Otherwise, use a different generator algorithm that has well-understood properties, a huge period, and has been shown to pass numerous tests of randomness.

7.11 Inversive congruential generators

A new family of nonlinear random-number generators, called *inversive congruential generators (ICGs)*, was introduced in 1986 by Eichenauer and Lehn [EL86], and led to a series of more than 50 papers by the first author and his coworkers. One of the more recent provides a survey of that extensive work [EHHW98], and an earlier tutorial recommends particular parameter values [EH92].

The nonlinearity is achieved by an apparently small change in the generator formulas, replacing the input random number by its *multiplicative inverse*:

$$\begin{aligned} x_{n+1} &= (Ax_n + C) \bmod M, && \textit{standard LCG}, \\ x_{n+1} &= (A\bar{x}_n + C) \bmod M, && \textit{inversive CG}, \\ (\bar{x}_n x_n) \bmod M &= 1, && \bar{x}_n \textit{ is multiplicative inverse of } x_n. \end{aligned}$$

As with linear congruential generators, restrictions on A and C are needed to ensure maximal period length and optimum randomness. For a 32-bit generator family, recommended settings are $A = 163\,011\,135$, $C = 1\,623\,164\,762$ and $M = 2^{31} - 1$ (the largest signed 32-bit integer, and also a prime number). Another choice is $A = 1$, $C = 13$, and M as before. With those parameters, the period length is 2^{31} and the range is $[0, 2^{31} - 1]$, so the generator produces all possible nonnegative integers. For a 64-bit generator family, a good choice is $A = 5\,520\,335\,699\,031\,059\,059$, $C = 2\,752\,743\,153\,957\,480\,735$, and $M = 2^{63} - 25$ (the largest prime for 64-bit signed integers). Alternatively, use $A = 1$ and $C = 1$ with the same M.

All of the code from a software implementation of an LCG can be reused, including that needed for simulating higher-precision arithmetic, except that x_n in the product is replaced by a function call that computes the multiplicative inverse. We show how to find that inverse shortly.

The extensive studies of inversive congruential generators have shown that, as with LCGs, it is undesirable to choose M as a power of two, even though that can simplify the generator computation. With a prime modulus in an ICG, the undesirable planar structures of LCGs disappear, and the advantage of a single seed is preserved, but the period still cannot exceed the modulus M. With a 32-bit LCG, the period can be exhausted in a few minutes on common desktop computers, and on the largest supercomputers, the full period of a 64-bit LCG can be reached in a few hours.

With good parameter choices, inversive congruential generators perform well with the test suites described near the end of this chapter, and unlike LCGs, subsets of bits, such as the top, middle, or bottom few from each word, also pass the tests.

Thus, inversive congruential generators are promising, but the main question is, how much slower are they than LCGs? The answer to that depends on how fast we can compute the multiplicative inverse, for which we require the excursions of the next two subsections to obtain some new library functions that are needed for the job.

7.11.1 Digression: Euclid's algorithm

The multiplicative inverse can be computed from an extension of the famous Euclid algorithm for finding the *greatest common divisor (gcd)* of two integers. In this section, we treat that algorithm in detail, deferring its extension to a shorter following section.

Computation of the greatest common divisor is one of the most important nontrivial operations in integer arithmetic, yet computer instruction sets do not supply it as a single operation. The gcd operation is also useful in multiple-precision integer arithmetic, and for symbolic computations with polynomials.

A brute-force solution of finding `gcd(x, y)` is easy to state, but slow to run: simply try all divisors from one up to the smaller of the two argument magnitudes, and record the largest found that divides the arguments without remainder. The execution time is clearly $\mathcal{O}(\min(|x|, |y|))$.

The solution of the gcd problem by the Greek mathematician, Euclid of Alexandria, dates from about 300 BCE, and perhaps was found before him. It is regarded as the oldest computational algorithm known to humans, in the sense that it requires arithmetic, repetition, and conditional tests. Euclid's algorithm can be written as this recurrence:

$$\begin{aligned} &\gcd(x, 0) \to x, && \textit{for } x > 0, \\ &\gcd(x, y) \to \gcd(y, x \bmod y), && \textit{for } x \geq y \geq 0. \end{aligned}$$

The algorithm is most easily written in a language that supports recursion, and its code in the C family is short:

```
int
gcd(int x, int y)
{
    int q, r;

    assert( (x > 0) && (y > 0) );

    q = x / y;          /* integer quotient */
    r = x - y * q;      /* integer remainder */

    return ((r == 0) ? y : gcd(y, r));
}
```

As written here, the code requires positive nonzero arguments, and the assertion guarantees immediate termination if that condition is not satisfied. We show later how to remove that restriction in order to write a robust gcd function that handles all representable integer arguments. Intermediate values during the recursion never exceed the sizes of the original arguments, so extra internal precision is not needed.

Even though the mathematical recurrence uses the mod operator, we refrain from using the C remainder operator, `%`: its behavior is implementation-dependent when its operands are negative, and we need to handle such values later. To avoid surprises, programmers should always be careful about negative operands in integer division and remainder.

Because the sole recursive call to the function occurs just before the return, it is easy for a good optimizing compiler to convert the *tail recursion* into a loop that requires no additional space on the call stack. We can easily do so as well with this alternate version suitable for programming languages that lack recursion:

```
int
gcd(int x, int y)
{
    assert( (x > 0) && (y > 0) );

    for (;;)
    {
        int q, r;

        q = x / y;              /* integer quotient */
        r = x - y * q;          /* integer remainder */

        if (r == 0)
            break;              /* here is the only loop exit */

        x = y;
        y = r;
    }

    return (y);
}
```

The arguments to `gcd(x, y)` are unordered, but during the first call or iteration, if $x < y$, we find $q = 0$ and $r = x$, so in the next step, we compute `gcd(y, x)`, putting the smaller argument last. All subsequent steps retain that order. Because integer division truncates, the remainder r is always nonnegative, and one of the arguments in the recursive calls always decreases, so we eventually find that r is zero, and the algorithm terminates.

Here is an example of Euclid's algorithm using the first eight digits of the mathematical constants e and π, displaying the successive decompositions $x = y * q + r$ with an instrumented version in hoc:

```
hoc> gcd_trace(27_182_818, 31_415_926)
#
# output:             x =          y *  q +           r
#
step =  1    27_182_818 =  31_415_926 *  0 +  27_182_818
step =  2    31_415_926 =  27_182_818 *  1 +   4_233_108
step =  3    27_182_818 =   4_233_108 *  6 +   1_784_170
step =  4     4_233_108 =   1_784_170 *  2 +     664_768
step =  5     1_784_170 =     664_768 *  2 +     454_634
step =  6       664_768 =     454_634 *  1 +     210_134
step =  7       454_634 =     210_134 *  2 +      34_366
step =  8       210_134 =      34_366 *  6 +       3_938
step =  9        34_366 =       3_938 *  8 +       2_862
step = 10         3_938 =       2_862 *  1 +       1_076
step = 11         2_862 =       1_076 *  2 +         710
step = 12         1_076 =         710 *  1 +         366
step = 13           710 =         366 *  1 +         344
step = 14           366 =         344 *  1 +          22
step = 15           344 =          22 * 15 +          14
step = 16            22 =          14 *  1 +           8
step = 17            14 =           8 *  1 +           6
step = 18             8 =           6 *  1 +           2
step = 19             6 =           2 *  3 +           0
2
```

The final answer for the gcd is 2. The x and y values reported on each line show that the arguments decrease quickly at each step; indeed, it is known that the remainder at least halves every two iterations. Also, notice that the preceding two remainders are the arguments for the current step, and the last nonzero remainder divides all previous ones.

The slowest operation is the division in each recursion or loop iteration, and we show later how we can reduce the number of divisions, and even eliminate them entirely.

What is perhaps most surprising about Euclid's algorithm is that two critical questions about its efficiency compared to the brute-force approach could not be answered with mathematical proofs until the 1970s and 1990s: what are the *average* and *worst-case* recursion or iteration counts?

Presentation and analysis of Euclid's gcd algorithm, and some variants that are often faster, occupies almost fifty complicated pages of volume 2 of *The Art of Computer Programming* [Knu97, pages 333–379]. The answers to the two questions are difficult to derive, and despite the fact that the algorithm involves just three arithmetic operations in the body, and no constants other than zero, the answers contain π, logarithms, and Fibonacci numbers (see **Section 2.7** on page 15 and **Section 18.6** on page 575):

$$\begin{aligned}
n &= \min(|x|, |y|), \\
\text{Euclid gcd minimum count} &= 1 \\
\text{Euclid gcd average count} &= (12 \ln 2/(\pi^2)) \ln n \\
&\approx 1.9405 \log_{10} n, \\
\text{Euclid gcd maximum count} &\approx (4.8 \log_{10} n) + 0.06.
\end{aligned}$$

The best case happens when the arguments are equal, or one is a multiple of the other: we find $r = 0$ on the first iteration, and can return immediately. The count in the average case is roughly twice the number of decimal digits in the *smaller argument*. The worst case occurs when the arguments are successive Fibonacci numbers, yet it needs less than three times the work for the average case. For 32-bit integers, the average and maximum counts are about 18 and 45, and for 64-bit integers, 36 and 91.

7.11.1.1 Euclid's algorithm for any integers

The mathcw library provides functions for computing the greatest common divisor, and the related *least common multiple (lcm)* operation, with these prototypes in the header file `gcdcw.h`

```
extern           int   gcd (          int x,           int y);
extern      long int  lgcd (     long int x,      long int y);
extern long long int llgcd (long long int x, long long int y);
extern           int   lcm (          int x,           int y);
extern      long int  llcm (     long int x,      long int y);
extern long long int lllcm (long long int x, long long int y);
```

The functions must be robust, and able to process correctly all possible integer arguments, so we have to supply additional code to handle negative and zero arguments.

The functions should satisfy these relations for the gcd and lcm operations:

$$\begin{aligned}
\gcd(0,0) &= 0, && \textit{standard mathematical convention},\\
\gcd(0,y) &= |y|, && \textit{standard mathematical convention},\\
\gcd(x,0) &= |x|, && \textit{standard mathematical convention},\\
\gcd(x,y) &= \gcd(y,x), && \textit{argument exchange symmetry},\\
\gcd(x,y) &= \gcd(x,-y), && \textit{argument sign symmetry},\\
&= \gcd(-x,y) &&\\
&= \gcd(-x,-y), &&\\
\gcd(x,y) &\geq 0, && \textit{divisor is nonnegative by convention},\\
\gcd(x,y) &= \gcd(x-ny,y) && \textit{if } x \geq ny\textit{, and } n = 1,2,3,\ldots,\\
\gcd(nx,ny) &= n\gcd(x,y), && \textit{argument scaling relation for } n \geq 0,\\
\mathrm{lcm}(nx,ny) &= n\,\mathrm{lcm}(x,y), && \textit{argument scaling relation for } n \geq 0,\\
|xy| &= \gcd(x,y)\,\mathrm{lcm}(x,y), && \textit{relation with least common multiple}.
\end{aligned}$$

The first three equations define the special handling of zero arguments, and are clearly needed because we cannot permit zero divisors. The last equation shows how to compute the least common multiple. Notice that, unlike the gcd, the lcm is subject to integer overflow; our code silently returns the low-order word of a too-long product $|(x/\gcd(x,y))y|$.

Apart from needing absolute-value wrappers on return values, negative arguments pose no special problem for Euclid's algorithm as we have presented it so far: the computations of q and r are valid for any sign combinations. Here is a recursive version that incorporates the additional checks:

```
int
gcd(int x, int y)
{
    int result;

    if (x == 0)
        result = QABS(y);
    else if (y == 0)
        result = QABS(x);
    else
    {
        int q, r;

        q = x / y;          /* quotient */
        r = x - y * q;      /* remainder */
        result = ((r == 0) ? QABS(y) : gcd(y, r));
    }

    return (result);
}
```

Our `QABS()` macro provides a fast sign conversion.

The only anomaly in our function is that in two's-complement arithmetic (see **Appendix I.2.3** on page 972), the most negative representable integer has no positive counterpart, and reproduces itself when negated. Thus, on essentially all modern computers, `gcd(0, INT_MIN)`, `gcd(INT_MIN, 0)`, and `gcd(INT_MIN, INT_MIN)` all return `INT_MIN`, a negative value. There is no reasonable alternative here, so programmers just need to remember that, like the integer absolute-value function, while the gcd is normally *positive*, it can also be *negative* when an argument is the most negative integer.

7.11.1.2 Division-free gcd algorithm

Several authors have studied Euclid's algorithm in binary arithmetic with a view to reducing, or eliminating, the expensive divisions. See the cited section of *The Art of Computer Programming*, and these papers: [Ste67, Har70, Bre76a, SS94, Sor95, Bre99, Bre00, Har06]. Josef Stein is apparently the first to publish a binary variant of the greatest common denominator computation that completely eliminates divisions, at the cost of more iterations, and more complicated code. His work appears in a physics journal, and the article title is unrelated to Euclid or the gcd, although his abstract mentions the gcd. Brent cites unpublished (and thus, likely unavailable) earlier work from 1962, and Knuth quotes a first-century Chinese manuscript that describes a procedure for reducing fractions that is essentially a binary gcd method. Scientists often ignore history, or are unaware of it, so Stein gets the credit for the first improvement on Euclid's algorithm in more than two thousand years.

The binary gcd algorithms generally require positive nonzero arguments. Negative arguments, especially the most negative representable number, must be transformed with special care to equivalents with positive numbers. Also, some algorithms may require longer integer types internally, and some need the fast `ilog2()`, `ntz()`, and `pop()` functions introduced earlier in this chapter. Their average and worst-case iteration counts (not all of which have been proven yet) have usually been shown to be $\mathcal{O}(\log(\max(|x|, |y|)))$, whereas Euclid's algorithm has $\mathcal{O}(\log(\min(|x|, |y|)))$. However, the binary algorithms replace most, or all, of the divisions by faster bit testing and shifting operations, so they can still outperform Euclid's.

A test program written by this author, and available in the file `tgcd.c` in the `mathcw` library distribution, implements most of the proposals in the cited papers and allows comparison of iteration counts and run times across different architectures, different compilers, and different optimization levels, as well as with different integer types. Extensive tests show considerable variations, and no single algorithm is uniformly the winner. Knuth's Algorithm B [Knu97, page 338] is one of the best, and may run from two to ten times faster than Euclid's. His procedure is a six-step recipe with three `goto` statements. Such statements are forbidden in our library, but fortunately, it is possible to replace them with loops, and our version has been tested against his original algorithm to ensure identical, and correct, output. Here is the complete code for our replacement, with comments linking our code to Knuth's original prescription:

```
int
gcd_knuth_binary_new(int x, int y)
{
    int result;

    if (x == 0)
        result = QABS(y);
    else if (y == 0)
        result = QABS(x);
    else
    {
        int double_count, t;
        double_count = 0;

        if (IS_EVEN(x))
        {
            x = HALF_EVEN(x);

            if (IS_EVEN(y))     /* both even */
            {
                y = HALF_EVEN(y);
                double_count++;
            }
        }
        else if (IS_EVEN(y))    /* x is known to be odd */
            y = HALF_EVEN(y);

        if (x < 0)
            x = -x;

        if (y < 0)
            y = -y;

        assert( (x > 0) && (y > 0) );

        while (IS_EVEN(x) && IS_EVEN(y))/* Step B1: reduce even args */
        {
            double_count++;
            x = HALF_EVEN(x);
            y = HALF_EVEN(y);
        }

        assert(IS_ODD(x) || IS_ODD(y));

        t = IS_ODD(x) ? -y : x;     /* Step B2: initialize */

        while (t != 0)
        {
            while (IS_EVEN(t))      /* Step B3: halve t */
                t = HALF_EVEN(t);

            if (IS_ODD(t))          /* Step B4: is t even or odd? */
            {
                if (t > 0)          /* Step B5: reset max(x, y) */
                    x = t;
                else
                    y = -t;

                assert( (x > 0) && (y > 0) );

                t = x - y;          /* Step B6: subtract */
            }
        }

        result = x;

        if (double_count)
            result <<= double_count;
    }
```

```
        return (result);
    }
```

As before, the assertions contain critical sanity checks that can be silently eliminated by defining the standard C-language macro `NDEBUG` at compile time.

Tests for even and odd can be done with fast bit tests, but we hide the details behind obvious macro names.

Halving of binary integers can be done with a fast bit shift, rather than slow division, but correct halving of negative integers requires representation-dependent code that again is best hidden inside an obviously named macro.

The `double_count` variable records the number of doublings required to scale the final result, and we do that inline with a left-shift operation. Because the result can never exceed the input arguments, the shift count is always in bounds, and the operation is safe.

The most troublesome case for gcd algorithms is a value of `INT_MIN` for either or both arguments, so we should think through what happens in that situation. The first two tests on entry handle the cases where one has that value, and the other is zero. However, for calls `gcd(INT_MIN, n)` or `gcd(n, INT_MIN)` with n nonzero , execution reaches the main block. In two's-complement arithmetic, `INT_MIN` has the *even* value $-(2^{w-1})$, where w is the integer width in bits. The tests for even x and y safely produce `INT_MIN / 2`, and its negation is positive and representable.

7.11.2 Another digression: the extended Euclid's algorithm

An extension of Euclid's algorithm (see [Knu97, pages 342–345], [BB87a], and [Sil06, Chapter 6]) finds an integer solution (a, b, c) to the equation pair

$$\begin{aligned} ax + by &= c, \\ \gcd(x, y) &= c, \end{aligned}$$

in time that is $\mathcal{O}(\log(|x| + |y|))$. The relation $ax + by = c$ is sometimes called *Bézout's identity*. A solution is *not* unique, because we can increment a by ny, and decrement b by nx, for $n = 1, 2, 3, \ldots$, giving $(a + ny)x + (b - nx)y = ax + nyx + by - nxy = ax + by = c$. Indeed, all of the other possible solutions have that form.

The extended Euclid's algorithm is of interest in this chapter because of its relation to the multiplicative inverse:

$$\begin{aligned} (\bar{x}x) \bmod M &= 1, && \textit{definition of multiplicative inverse } \bar{x}, \\ \bar{x}x &= kM + 1, && \textit{equivalent statement for integer } k, \\ \bar{x}x - kM &= 1, && \textit{extended Euclid's algorithm form}. \end{aligned}$$

In general, that inverse exists if, and only if, x and M are *coprime*, that is, they have no integer factors in common other than 1. Thus, if M is a prime number, then the multiplicative inverse exists for all integers x in $(0, M)$, and also for almost any integral multiple of those numbers: only multiples with integral-power factors, M^n $(n = 1, 2, 3, \ldots)$, are disallowed.

When the multiplicative inverse exists, it is not unique: we can increase $\bar{x}$ by any integer multiple of M. For example, with $M = 5$ and $x = 3$, $\bar{x} = \ldots, -8, -3, 2, 7, 12, \ldots$ are all valid. It is conventional to choose the smallest positive value of $\bar{x}$.

If we have a function `egcd(&a, &b, x, y)` that implements the extended algorithm for signed integer arguments, then we can easily write a function for the multiplicative inverse, with a small final fixup to ensure a nonnegative result:

```
    int
    invmodp(int x, int p)
    {   /*
        ** Return a multiplicative inverse of x modulo p.
        ** NB: x and p MUST be coprime, but that is not checked!
        **
        ** Equivalents in symbolic-algebra systems:
        **
        ** Maple:         x^(-1) mod p
        ** Mathematica:   PowerMod[x, -1, p]
        ** Maxima:        power_mod(x, -1, p) and inv_mod(x, p)
        ** MuPAD:         powermod(x, -1, p)
        */

        int a, b;

        p = QABS(p);
        (void)egcd(&a, &b, x, -p);

        return((a < 0) ? (a + p) : a);
    }
```

The multiplicative inverse of zero is infinity, but that is not representable in integer arithmetic. Our `invmodp(0, p)` always returns zero for *any* value of *p*, including zero.

Knuth's Algorithm X for the extended Euclid's algorithm employs three working arrays of three elements each, and uses iteration and vector operations to solve for the three values *a*, *b*, and *c*. Coding is simplified if we define four array operations as macros, so that most subscripts can be hidden from the function body:

```
#define COPY(t, u)             (t[0] = u[0], t[1] = u[1], t[2] = u[2])

#define SET(t, a, b, c)        (t[0] = a, t[1] = b, t[2] = c)

#define SUB(t, u, v)           (t[0] = u[0] - v[0], t[1] = u[1] - v[1], t[2] = u[2] - v[2])

#define U_MINUS_V_Q(t, u, v, q) (t[0] = u[0] - v[0] * q, t[1] = u[1] - v[1] * q, t[2] = u[2] - v[2] * q)
```

Our implementation of the complete algorithm looks like this:

```
int
egcd(int *pa, int *pb, int x, int y)
{   /*
    ** Find a solution of a * x + b * y = c = gcd(x, y).
    ** Return c as the function value, and return a and b
    ** via the pointer arguments, if they are not NULL.
    **
    ** Equivalents in symbolic-algebra systems:
    **
    ** M2:          gcdCoefficients(x, y) # returns list [a, b]
    ** Maple:       gcd(x, y, 'a', 'b')   # returns c, sets a and b
    ** Mathematica: ExtendedGCD[a, b]     # returns list {c, {a, b}}
    ** Maxima:      gcdex(x, y)           # returns list [a, b, c]
    ** MuPAD:       igcdex(x, y)          # returns list [c, a, b]
    ** Pari/GP:     bezout(x, y)          # returns list [a, b, c]
    */

    int q, t[3], u[3], v[3];

    if (x == 0)
    {
        u[0] = 0;
        u[1] = (y < 0) ? -1 : ((y == 0) ? 0 : 1);
        u[2] = QABS(y);
    }
    else if (y == 0)
    {
        u[0] = (x < 0) ? -1 : 1;
        u[1] = 0;
        u[2] = QABS(x);
    }
    else
    {
        SET(u, 1, 0, x);    /* Step X1: Initialize */
        SET(v, 0, 1, y);

        while (v[2] != 0)   /* Step X2: termination check */
        {   /* loop invariant: u[0] * x + u[1] * y = u[2] */
            q = u[2] / v[2];
            U_MINUS_V_Q(t, u, v, q);
            COPY(u, v);
            COPY(v, t);
        }
    }

    if (u[2] < 0)           /* force gcd positive */
    {
        u[2] = -u[2];
        u[1] = (u[2] < 0) ? u[1] : -u[1];
        u[0] = -u[0];
    }

    if (pa != (int *)NULL)
        *pa = u[0];

    if (pb != (int *)NULL)
        *pb = u[1];

    return (u[2]);
}
```

The sign-inversion block that follows the loop includes a check for u[2] remaining negative, which happens only when it is INT_MIN.

Knuth shows that the computation of the middle element of each array can be omitted in the loop, which we can easily do by commenting out the relevant parts of the four macros. On loop exit, we can recover the needed value u[1] from the loop invariant u[0] * x + u[1] * y = u[2]. However, doing so requires a double-length integer type. Apart from that possibility, the code requires no arithmetic of higher precision than that for the argument types.

Knuth also gives a binary version, Algorithm Y [Knu97, page 646], that replaces multiplications and divisions by faster bit-shift operations, at the expense of more loop iterations. For brevity, we omit its code here, but we supply it as a compile-time alternative in the algorithm file egcdx.h.

The mathcw library provides the extended Euclid's algorithm, and the multiplicative inverse operation, in functions with these prototypes supplied in the header file gcdcw.h:

```
extern           int      egcd (          int *pa,           int *pb,
                                          int x,             int y);
extern      long int     legcd (     long int *pa,      long int *pb,
                                     long int x,        long int y);
extern long long int    llegcd (long long int *pa, long long int *pb,
                                long long int x,   long long int y);
extern           int   invmodp (          int x,             int p);
extern      long int  linvmodp (     long int x,        long int p);
extern long long int llinvmodp (long long int x,   long long int p);
```

Library functions for the greatest common divisor, least common multiple, and multiplicative inverse are a first step in developing a toolbox for number theory. However, in practice, the limitations of fixed-width integers soon become evident, and serious work with exact integer arithmetic generally requires multiple-precision integers. Symbolic-algebra systems, and the Lisp language family, provide the machinery needed to support such integers without the special programming needed in most other languages.

7.12 Inversive congruential generators, revisited

The excursions of the preceding subsections finally led us to the critical function, `invmodp(x, p)`, needed to replace x_n in an LCG with `invmodp(`x_n`, M)` to make an ICG.

Timing tests with compiler-optimized code were run on about three dozen CPU variants from all of the major desktop architectures, using our version of `invmodp()` that calls the implementation of the `egcd()` function shown in **Section 7.11.2** on page 187. They show that the ICG is 8 to 60 times slower than the LCG, with a slowdown of 20 perhaps being 'typical'. On some systems, a binary version of `egcd()` could reduce that gap somewhat. However, it is worth reporting that similar tests of several other generator algorithms show that the quickest of them are up to 10 times faster than the LCG. The inversive congruential generator must therefore be regarded as sluggish, compared to alternatives.

7.13 Distributions of random numbers

All of the random-number recipes that we have discussed so far in this chapter produce numbers that (ideally) are nearly evenly sprinkled throughout the range of the generator. That is called a *uniform distribution*.

There are many other distributions than uniform, however, and we look at three of the important nonuniform ones in the following subsections. It is almost always possible to compute random numbers for nonuniform distributions from uniformly distributed values, so the hard work of finding a good generator does not have to be repeated.

To better understand distributions, it is helpful to make plots of them. In the following subsections, we look at three kinds of characteristic plots. Except for really dreadful generators, it is usually not possible to tell anything about generator quality from such plots, but it is easy to verify from them that the random numbers conform at least roughly to their expected distribution.

7.13.1 The uniform distribution

For a uniform distribution of random numbers, if we count the number of values in each of a set of intervals evenly spanning the range of the generator, we should expect to see about the same count in each interval. The totals can be conveniently visualized as a *histogram*.

If we plot the random value along the vertical axis as a function of its position in the sequence measured along the horizontal axis, then we should expect to see a uniform distribution of points in the plot.

We can get a third view of the data by sorting them, and plotting their values on the vertical axis against their sorted position along the horizontal axis. The plot of that *cumulative distribution function* should be a straight line rising from zero to one.

Those three views of a uniform distribution are shown in **Figure 7.4** on the next page.

7.13.2 The exponential distribution

It is easy to compute random numbers in the *exponential distribution*, using a one-line function like this one:

```
double rnexp(void) { return (-log(urand())); }
```

The function `urand()` here should be defined on either of the intervals $(0, 1]$ or $(0, 1)$, so as to avoid the possibility of a zero argument of the logarithm.

Here is another way to compute exponentially distributed random numbers from samples of a uniform distribution [DH97a]:

```
double
rnexp(void)
{   /* return random number exponentially distributed on (0,Inf) */
    int n;
    double beta1, betasum, t;

    n = 1;
```

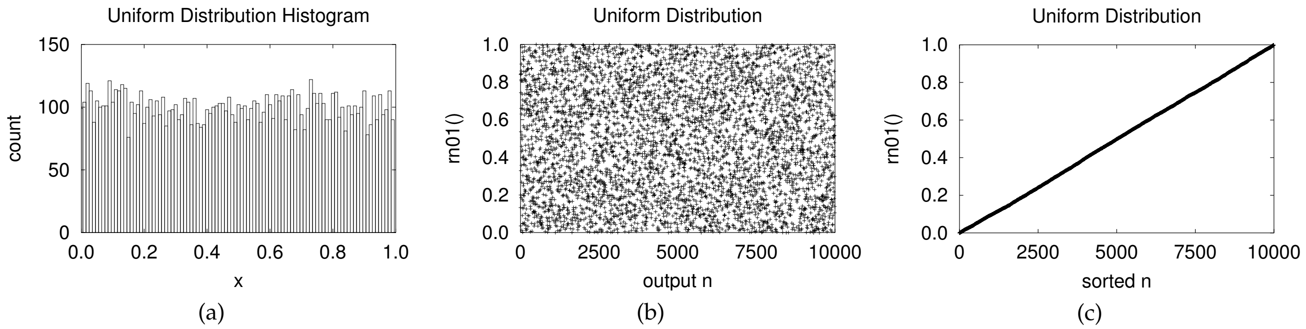


Figure 7.4: Views of uniform distributions of random numbers. Plot (a) shows a histogram of the counts of 10 000 values falling into each of 100 intervals of equal size. Plot (b) shows the 50% of the same values plotted against their sequence number. Plot (c) shows sorted values plotted against their sequence number, showing an approximate cumulative distribution function.

```
    t = -1;

    while ((n % 2) == 1)
    {                         /* tcov: average loop count = 1.58 */
        beta1 = urand();
        betasum = urand();
        n = 2;

        while (betasum < beta1)
        {                     /* tcov: average loop count = 0.72 */
            n++;
            betasum += urand();
        }

        t++;
    }

    return (t + beta1);
}
```

That variant uses a method first introduced in 1949 by John von Neumann [vN51] that avoids calls to elementary functions. Timing tests on various platforms show it to be up to 1.3 times faster than the simple one-line form on some systems, and twice as slow on others. The random values produced by the two methods differ, because they consume different numbers of results from `urand()`.

A test-coverage analysis of the second version of `rnexp()` shows that, despite the doubly nested loops, fewer than four calls to `urand()` are required on average. Even though the function can return values on $(0, \infty)$, in practice, the results are never large: repeated tests with 10^9 calls to `rnexp()` produce a maximum return value of only 22.45, about what is expected because $\exp(22.45\cdots) \approx 5.62 \times 10^9$.

Figure 7.5 on the facing page shows three views of an *exponential distribution*. Here, small random numbers are more likely than large ones, and the histogram in Figure 7.5(a) falls off like a decaying exponential.

7.13.3 The logarithmic distribution

For sampling from a range of floating-point values, the *logarithmic distribution* is important. It can be readily computed from a uniform distribution with a function like this:

```
double
randlog (double a, double b)
```

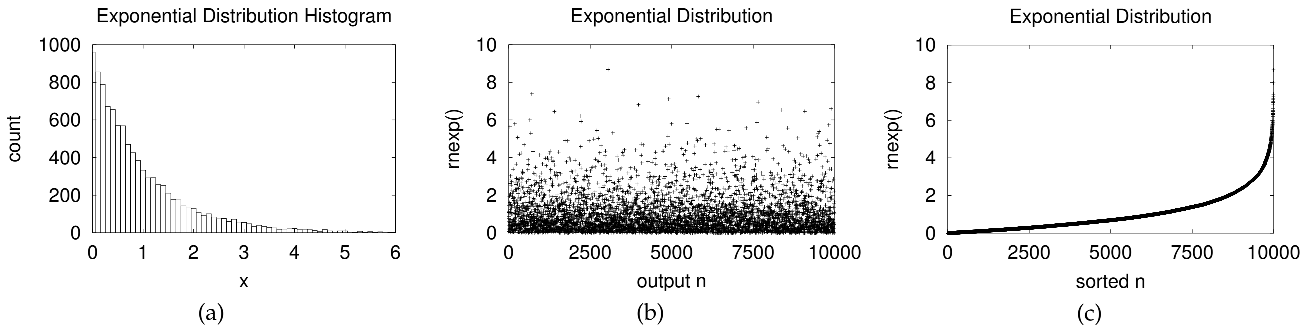


Figure 7.5: Views of exponential distributions of random numbers. See **Figure 7.4** on the facing page for an explanation of the three plots.

```
{
    double t;

    t = log(abs(b)) - log(abs(a))
    return (a * exp(t * urand()));
}
```

The function has the property that it returns values logarithmically distributed on the interval $[a, b]$, which must not contain zero. That function can, of course, be optimized for a fixed interval by precomputing the logarithms, or by saving them across calls. The intermediate value could be written more compactly as `log(abs(b/a))`, but the division could then cause premature overflow or underflow.

Our function as written is still subject to premature overflow if the argument range is too wide, such as with a call `randlog(DBL_MIN, DBL_MAX)` to sample over the entire floating-point range. To avoid overflow, that range must be split into two parts, with calls `randlog(DBL_MIN, 1.0)` and `randlog(1.0, DBL_MAX)`. We could use those values as random arguments for testing a function $f(x)$ defined for all real positive arguments. By contrast, if we use scaled results of `urand()` to sample over those two intervals, almost all of the results are near their upper limits.

There is still another problem lurking in our function, and that is premature underflow or overflow from faulty implementations of the exponential function. We encountered such defects during testing with native libraries on several platforms. The final code in our library contains checks for special arguments and invalid ranges, caches values of the logarithms, and rejects out-of-range results. It looks like this:

```
fp_t
LRCW(fp_t a, fp_t b)
{   /* return random number logarithmically distributed on [a,b] */
    fp_t result;

    if (ISNAN(a))
        result = SET_EDOM(a);
    else if (ISNAN(b))
        result = SET_EDOM(b);
    else if (SIGNBIT(a) != SIGNBIT(b))
        result = SET_EDOM(QNAN(""));
    else if ( (a == ZERO) || (b == ZERO) )
        result = SET_EDOM(QNAN(""));
    else if (ISINF(a))
        result = SET_ERANGE(a);
    else if (ISINF(b))
        result = SET_ERANGE(b);
    else
    {
        static fp_t a_last = FP(0.0);
        static fp_t b_last = FP(0.0);
        static fp_t log_a = FP(0.0);
        static fp_t log_b = FP(0.0);

        if (a != a_last)
        {
            a_last = a;
            log_a = LOG(a < ZERO ? -a : a);
        }

        if (b != b_last)
        {
            b_last = b;
            log_b = LOG(b < ZERO ? -b : b);
        }

        do
        {
            result = a * EXP((log_b - log_a) * URAND());
        }
        while ( (result < a) || (b < result) );
    }

    return (result);
}
```

Figure 7.6 on the next page shows our three standard views for the logarithmic distribution. Notice the substantial similarity to the exponential distribution in Figure 7.5 on the preceding page.

7.13.4 The normal distribution

The last important distribution that we treat is the famous *normal distribution*, also known as the bell-shaped curve. The distribution is shown in Figure 7.7.

The normal distribution has received extensive study, and there are several good ways to compute normally distributed random numbers from samples of a uniform distribution. The polar method is one of the simplest to implement, and its code looks like this:

```
double
rnnorm(void)
{   /* polar method for normally distributed random numbers */
    double s, v1, v2;

    do
    {
        v1 = urand();           /* v1 in [ 0, +1] */
        v2 = urand();           /* v2 in [ 0, +1] */
        v1 += v1 - ONE;         /* v1 in [-1, +1] */
        v2 += v2 - ONE;         /* v2 in [-1, +1] */
        s = v1 * v1 + v2 * v2;  /* s  in [ 0, +2] */
    }
    while (s >= ONE);

    return (v1 * sqrt(-TWO * log(s) / s));
}
```

In the `return` statement, we could replace `v1` by `v2`, so with a bit more code, we could generate two values, saving one of them for a fast return on the next call.

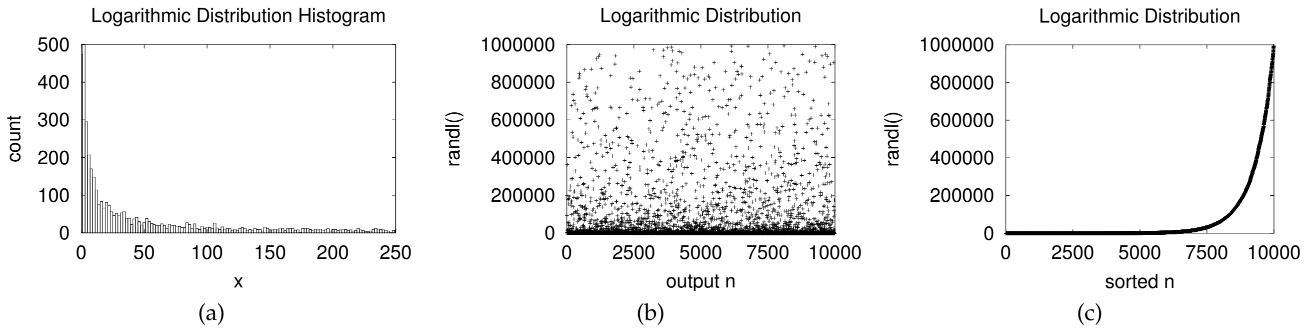


Figure 7.6: Views of logarithmic distributions of random numbers from `randlog(1.0, 1000000.0)`. See **Figure 7.4** on page 190 for an explanation of the three plots.

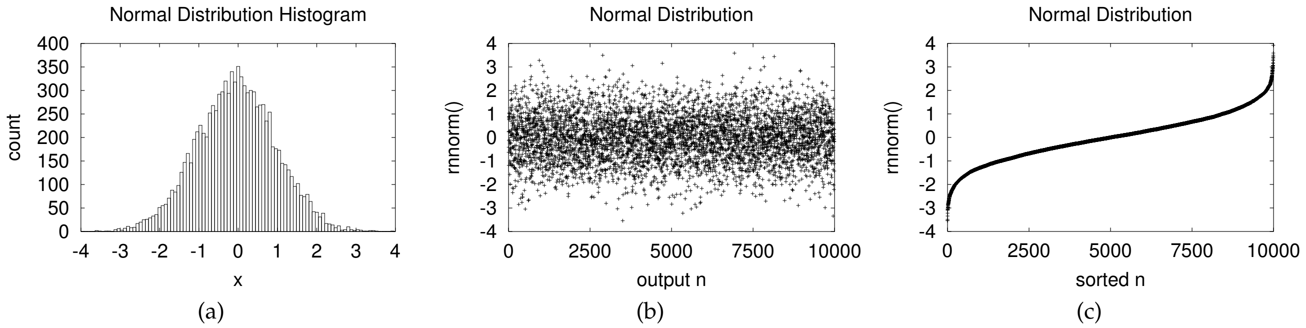


Figure 7.7: Views of normal distributions of random numbers. See **Figure 7.4** on page 190 for an explanation of the three plots.

As with von Neumann's method for the exponential distribution, the polar method for the normal distribution consumes a variable number of random values from `urand()`, and that count depends on the properties of the host arithmetic systems, as well as on the uniform generator. We therefore cannot expect to get the same sequence of values from the code for the polar method when it is run on different CPU architectures.

The *Box–Muller transformation* [BM58] is another popular way to produce normally distributed random numbers x, y from a uniformly distributed pair u, v:

$$x = \cos(2\pi v)\sqrt{-2\log u}, \qquad\qquad y = \sin(2\pi u)\sqrt{-2\log v}.$$

Because of the six function calls, it is more expensive than the polar method, but it consumes a fixed number of values from a uniform generator. However, it is known to amplify correlations between pairs of normally distributed values, so it might be better to compute and use only x or y. Avoid the Box–Muller and polar methods entirely if the uniform generator is an LCG [Nea73, AW88].

Although a generator for the normal distribution can potentially produce *any* positive or negative floating-point number, the rapid fall of the normal curve makes large normally distributed random values highly unlikely. In **Section 19.4** on page 610, we show that fewer than one in a million should lie outside $[-5, +5]$, and only about one in 10^{23} is expected outside $[-10, +10]$.

7.13.5 More on the normal distribution

> The
> normal
> law of error
> stands out in the
> experience of mankind
> as one of the broadest
> generalizations of natural
> philosophy ◇ It serves as the
> guiding instrument in researches
> in the physical and social sciences and
> in medicine agriculture and engineering ◇
> It is an indispensable tool for the analysis and the
> interpretation of the basic data obtained by observation and experiment.
>
> — W. J. Youdon (1956)
> from Stephen M. Stigler, *Statistics on the Table* (1999).

The importance of the normal distribution is its relation to the most famous theorem of probability, the *Central Limit Theorem* (de Moivre 1718, Laplace 1810, and Cauchy 1853), which can be stated informally like this:

> A suitably normalized sum of independent random variables is likely to be normally distributed, as the number of variables grows beyond all bounds. It is not necessary that the variables all have the same distribution function or even that they be wholly independent.
>
> — I. S. Sokolnikoff and R. M. Redheffer
> *Mathematics of Physics and Modern Engineering*,
> second edition (1966).

The quotation can be expressed mathematically like this:

$$P(n\mu + r_1\sqrt{n} \leq \sum_{k=1}^{n} X_k \leq n\mu + r_2\sqrt{n}) \approx \frac{1}{\sigma\sqrt{2\pi}} \int_{r_1}^{r_2} \exp(-t^2/(2\sigma^2))\, dt.$$

Here, the X_k are *independent*, *identically distributed*, and *bounded* random variables, μ (Greek letter *mu*) is their *mean value*, σ (Greek letter *sigma*) is their *standard deviation*, and σ^2 is their *variance*. The formula says that the probability that the sum of the n variables lies between $n\mu + r_1\sqrt{n}$ and $n\mu + r_2\sqrt{n}$ is the area under the normal curve between r_1 and r_2.

The integrand of the normal distribution's probability function is shown in **Figure 7.8**. Although the curve extends to infinity in both directions, it falls off extremely rapidly, with almost all of the area within just a few multiples of σ away from the origin.

It is common in scientific and medical experiments to report standard deviations and variances for experimental measurements, and our brief description shows how they are related to the normal distribution. We consider the computation of the normal distribution function, and discuss the meaning and significance of the standard deviation, in **Section 19.4** on page 610.

The distribution that our `rnnorm()` function produces is scaled to zero mean and unit standard deviation and variance. Although the return values lie on $(-\infty, +\infty)$, only one sample in 10^{80} (the approximate number of particles in the universe) will exceed 19 in magnitude.

If x is normally distributed with zero mean and unit variance, then $y = \mu + \sigma x$ is also normally distributed, but with mean μ and standard deviation σ. A simple wrapper function hides the computation:

```
double
rnnorm_ext (double mu, double sigma)
{
    return (mu + sigma * rnnorm());
}
```

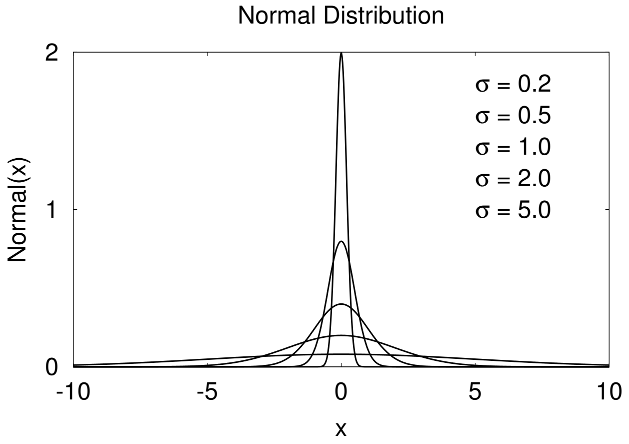


Figure 7.8: Normal curves for various σ values. The curve is most sharply peaked for small σ, and the area under the curve is normalized to one.

7.14 Other distributions

Although we do not describe them in detail, or develop software for them, there are several other important distributions that are commonly encountered in applications of random numbers and analysis of data. They are often divided into two classes, continuous and discrete, where in the latter case, probabilities can be worked out by simple counting operations.

- The discrete *binomial distribution* describes the behavior of sets of experiments of random processes, each independent of the other, and having two possible outcomes. Samples are selected, measured, and replaced in the original set. Examples include dice throws, picking (and replacing) beans from a jar of mixed red and blue beans, and defects in product manufacturing.

- The discrete *hypergeometric distribution* is similar to the binomial distribution, but differs from it, because samples are *removed*, decreasing the original population of data.

- The continuous *F distribution* is used in the statistical analysis of ratios of sample variances.

- The continuous *gamma distribution* represents *sums* of independent identically distributed exponential random variables. It is used in the analysis of waiting times.

  The *chi-square distribution*, for which we consider a related quantity later in **Section 7.15.1** on page 197, is a special case of the gamma distribution.

- The continuous *Laplace distribution* (1812) describes the *differences* between independent identically distributed exponential random variables, in contrast to the sums in the gamma distribution.

  The Laplace distribution function has an exponential whose argument is an absolute value, instead of the square found in the normal distribution, giving the distribution a cusp shape.

  The Laplace distribution is sometimes called the *double exponential distribution*, but that name should be avoided, because it is also used for a completely different distribution.

- The discrete *Poisson distribution* (1837) arises in the analysis of events that occur continuously with constant probability, such as customers arriving at a service desk, and radioactive decay. In the limit of large sample sizes with a finite mean, the binomial distribution becomes a Poisson distribution.

- The continuous *Student's t distribution* is used in the analysis of small sample sizes, a common situation in experimental science and medicine. As the sample size increases, the *t* distribution becomes a normal distribution.

Student is a pseudonym for W. S. Gosset [PGPB90], who published the analysis in 1908. Gosset was trained in chemistry and mathematics at Oxford University, and was later employed as a statistician at the Guinness Brewery in Dublin, Ireland, to help improve the process and product.

For an interesting account of the development of Gosset's t-test, how credit for it was usurped by another, and how it is too often abused and misused, see *The Cult of Statistical Significance* [ZM08]. See also *Why Most Published Research Findings Are False* [Ioa05] and *The Widespread Misinterpretation of p-Values as Error Probabilities* [Hub11] for detailed looks at common misinterpretations of statistical measures, and frequent failures to consider properly the effect of small sample sizes.

- The continuous *Weibull distribution* (1939) [MXJ04] is a generalized distribution that can mimic several others, including the exponential and normal distribution. It has extensive applications in the analysis of actuarial data and product failure rates.

For compact or elementary coverage of those, and other distributions, see [AS64, Chapter 26], [BE92], [AW05, Chapter 19], [AWH13, Chapter 23], [Law06, Chapter 8], [PTVF07, §6.14], [Dev08b, Chapters 3 and 4], and [AF09]. Advanced treatments are available in several volumes [JK77, Dev86, JKB94, JKB97, Che98, L'E98, KBJ00, HLD04, MXJ04, JKK05]. There is also a recent survey article on univariate distributions [LM08], an excellent review of power-law distributions [New05] that have broad applications in many areas of science and technology, and a survey of Gaussian random-number generation [MH16].

7.14.1 Numerical demonstration of the Central Limit Theorem

With the help of a random-number generator for a uniform distribution, we can show the Central Limit Theorem in action with a coin-tossing experiment implemented in hoc. Each coin flip is simulated by asking the generator to return a random integer on $[0, 1]$, and we use 1 for heads, and 0 for tails:

```
hoc> for (k = 1; k <= 10; ++k) print randint(0,1), ""; println ""
     0 1 1 1 0 0 0 0 1 0
```

Here, we got four heads and six tails. Now, instead of printing the results, we count the number of heads by summing the values that we previously printed. We wrap our program with another loop that repeats the experiment 100 times:

```
for (n = 1; n <= 100; ++n) \
{
    sum = 0

    for (k = 1; k <= 10; ++k) sum += randint(0,1)

    print sum, ""
}
println ""
        4 4 7 3 5 5 5 2 5 6 6 6 3 6 6 7 4 5 4 5 5 4 3 6 6 9 5 3 4 5 4 4 4 5 4 5 5 4 6 3 5 5 3 4 4 7 2 6 5 3
        6 5 6 7 6 2 5 3 5 5 5 7 8 7 3 7 8 4 2 7 7 3 3 5 4 7 3 6 2 4 5 1 4 5 5 5 6 6 5 6 5 5 4 8 7 7 5 5 4 5
```

In **Figure 7.9**, we plot the measured frequencies of the numbers of heads found in the 100 experiments. The histogram roughly approximates a normal curve, and from its header, we find that the coin flips produced five heads $Y_5 = 31$ times.

Simply by increasing the experiment count in the outer loop limit, we can gather more and more samples. **Figure 7.10** shows the results of 100 000 experiments, corresponding to a million coin tosses. As the Central Limit Theorem predicts, the results fall nicely along a normal curve.

7.15 Testing random-number generators

Determining whether the output of a random-number generator is sufficiently random is much harder than testing elementary functions, because there is no expected relation between function values: indeed, they should be entirely

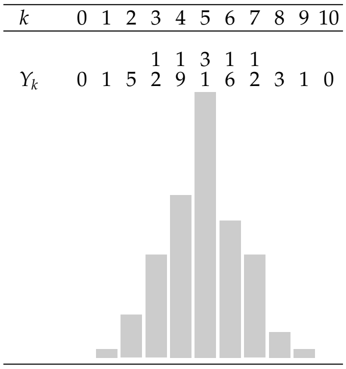


Figure 7.9: Counts of heads for 100 coin-flip experiments. The counts are displayed vertically from top to bottom above each bar: the tallest histogram bar has a count of 31.

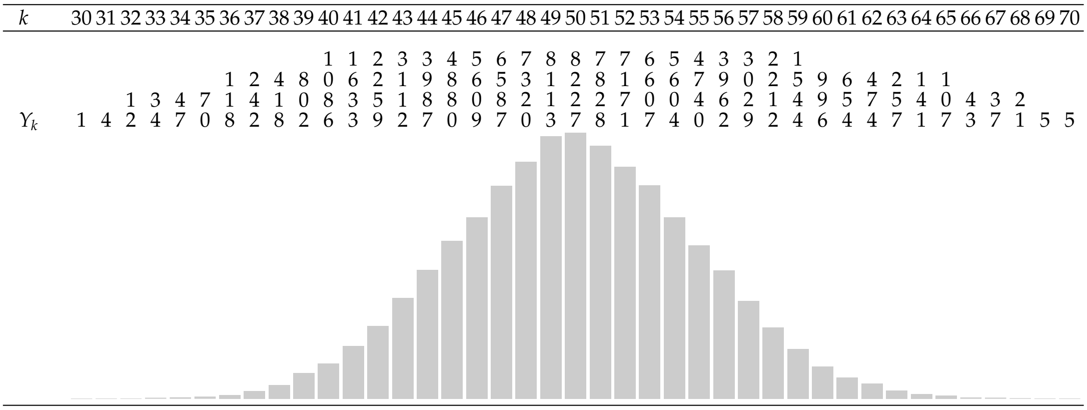


Figure 7.10: Counts of heads for 100 000 coin-flip experiments.

unrelated. Instead, all that can be done is to use the generator to model a particular problem for which an exact solution is known, and then compute statistical measures of how close the model is to the correct answer. Different models have different dependencies on the generator's randomness, and no single test suffices. Instead, we need to have a *suite* of widely differing tests, and only after a generator has passed a large number of independent tests can we begin to have confidence in it.

7.15.1 The chi-square test

One of the most important statistical measures is the famous chi-square test (for the Greek letter *chi*, χ), introduced by the eminent American statistician Karl Pearson in 1900 [Pea00]. The test, which has been ranked among the top twenty scientific discoveries of the Twentieth Century [Hac84], is simple. If the k-th experiment produces a measured count M_k when the expected count is E_k, then for n experiments, we compute:

$$\chi^2 \text{ measure} = \sum_{k=1}^{n} (M_k - E_k)^2 / E_k$$

Standard mathematical handbooks give tables of the chi-square measure, and **Table 7.1** on the following page is a small sample from such tables. We show later in **Section 18.4** on page 560 how those values can be computed. For sufficiently large n, the measure depends only on the number of degrees of freedom in the measurement.

Table 7.1: Selected percentage points of the chi-square distribution for ν (Greek letter *nu*) degrees of freedom. For example, from the shaded last entry in the first row, for one degree of freedom, such as in a coin-flip experiment, there is a 99% probability that the chi-square measure does not exceed 6.635.

ν	$p = 1\%$	$p = 5\%$	$p = 25\%$	$p = 50\%$	$p = 75\%$	$p = 95\%$	$p = 99\%$
1	0.00016	0.00393	0.1015	0.4549	1.323	3.841	6.635
2	0.02010	0.1026	0.5754	1.386	2.773	5.991	9.210
3	0.1148	0.3518	1.213	2.366	4.108	7.815	11.34
4	0.2971	0.7107	1.923	3.357	5.385	9.488	13.28
5	0.5543	1.145	2.675	4.351	6.626	11.07	15.09
6	0.8721	1.635	3.455	5.348	7.841	12.59	16.81
7	1.239	2.167	4.255	6.346	9.037	14.07	18.48
8	1.646	2.733	5.071	7.344	10.22	15.51	20.09
9	2.088	3.325	5.899	8.343	11.39	16.92	21.67
10	2.558	3.940	6.737	9.342	12.55	18.31	23.21
11	3.053	4.575	7.584	10.34	13.70	19.68	24.72
12	3.571	5.226	8.438	11.34	14.85	21.03	26.22
15	5.229	7.261	11.04	14.34	18.25	25.00	30.58
20	8.260	10.85	15.45	19.34	23.83	31.41	37.57
30	14.95	18.49	24.48	29.34	34.80	43.77	50.89
50	29.71	34.76	42.94	49.33	56.33	67.50	76.15
99	69.23	77.05	89.18	98.33	108.1	123.2	134.6
100	70.06	77.93	90.13	99.33	109.1	124.3	135.8
999	898.0	926.6	968.5	998.3	1029	1074	1106
1000	898.9	927.6	969.5	999.3	1030	1075	1107

As a concrete example of the computation and application of the chi-square measure, we can use a random-number generator to produce integers on the interval $[0, 9]$. There are ten possible digits, so the number of degrees of freedom is *nine*, one less than the number of categories. Here are two functions and a test procedure to carry out the experiment, which consists of sampling one-digit integers `nrun` times in function `digit_frequency()` and counting the number of times a particular digit is found, and then in a second function, computing the chi-square measure:

```
double
digit_frequency(int digit, int nrun)
{
    double sum;
    int k;

    sum = 0;

    for (k = 0; k < nrun; ++k)
    {
        if (urandtoint(0,9) == digit)
            sum++;
    }

    return (sum);
}

double
chisq_measure(int digit, int nrun)
{
    double expect, sum, term;
    int k;

    expect = (double)nrun / 10.0;
    sum = 0.0;
    for (k = 1; k <= 10; ++k)
    {
        term = digit_frequency(digit,nrun) - expect;
        sum += term * term / expect;
    }

    return (sum);
}

void
test_chisq(int nrun)
{
    double x;
    int digit;

    (void)printf("digit   chi-square  probability\n");

    for (digit = 0; digit <= 9; ++digit)
    {
        x = chisq_measure(digit,nrun);
        (void)printf("%5d  %11.4f  %11.4f\n", digit, x, chisq(9,x));
    }
}
```

The test procedure `test_chisq()` uses an external function, `chisq()`, whose code is not shown here, as a convenient substitute for a mathematical table, and makes a report like this when `nrun` is 100 000:

```
digit   chi-square  probability
    0      10.5470       0.6920
    1       9.1112       0.5729
    2      22.6200       0.9929
    3      13.4653       0.8573
    4       7.5559       0.4206
    5       5.4183       0.2036
    6      10.6368       0.6986
    7       5.8483       0.2450
    8       5.9518       0.2553
    9      11.2099       0.7384
```

For example, the computed chi-square measure for digit 9 is 11.2099, and the probability that a model-conforming chi-square measure does not exceed that value is 0.7384. Here, all of the computed probabilities are reasonable values, except possibly digit 2, but another run with a different generator seed gave it a probability of 0.5944.

Now watch what happens if we have a faulty generator that only returns digits in $[0,8]$. We simply replace `randint(0,9)` with `randint(0,8)`, and rerun the experiment:

```
digit   chi-square  probability
    0    1209.4584       1.0000
    1    1309.8509       1.0000
    2    1295.1959       1.0000
    3    1239.8429       1.0000
    4    1198.6093       1.0000
    5    1270.2240       1.0000
    6    1183.5953       1.0000
    7    1199.0114       1.0000
    8    1340.3970       1.0000
    9  100000.0000       1.0000
```

The chi-square measures are now large, and all of the probability values are 1.0000, meaning that it is almost a certainty that model-conforming measures should not be as large as they are found to be. In other words, the model of equal distributions of the ten digits is poorly met by our modified generator, and the large chi-square measure in the last output line identifies the faulty digit.

Many tests of random-number generators produce probability values similar to those from the chi-square tests, although sometimes, the statistics are based on other measures that are outside the scope of this book.

7.15.2 Random-number generator test suites

Unfortunately, most of the standard tests that have been historically recommended for checking the randomness of a sequence of numbers, including the simpler ones discussed in Knuth's comprehensive treatise [Knu97, Chapter 3], turn out not to be very discriminating. Although they can often identify really bad generators, they are of little use in distinguishing the mediocre from the superb.

Happily, there are now four good test suites that have proved to be effective:

- The Diehard Battery suite (`http://stat.fsu.edu/pub/diehard/`) was developed at Florida State University by George Marsaglia in 1985, but never formally published, although there are descriptions of some of the tests in the literature [MZ93, PW95]. The Diehard Battery suite is available in C and Fortran versions, and each expects to read a file of about 10MB of random bits. That means that the program can be built once and for all, and then run on test files produced by any generator. Any or all of 16 tests can be selected, and typical run times for the complete set are one to three minutes on modern workstations. The one thing to be careful of is to ensure that each file byte contains random bits: if the generator range is only $[0, 2^{31} - 1]$, an extra random bit needs to be supplied for the high-order bit of each 32-bit word written to the file.

- The recent *tuftest* suite by Marsaglia and Tsang [MT02] consists of three new tests that have been found to be as good as the entire Diehard Battery suite. Its authors report *"... generators that pass the three tests seem to pass all the tests in the* Diehard Battery *of Tests."* The code is in C, and instead of a data file, it requires a 32-bit generator that can be called at run time.

  This author has extended the tuftest suite as follows:

  - A new driver program allows easy testing of many different generators.
  - The driver allows selection of subsets of the tests.
  - The driver can quickly create Diehard Battery test files without running the tuftest suite.
  - Portability problems and faulty architectural assumptions are repaired. The code can now be used with both C and C++ compilers, and on both 32-bit and 64-bit architectures.

  Run times are typically 10 to 90 minutes on the same systems used for Diehard Battery tests.

- The *NIST Statistical Test Suite* [RSN$^+$01] is similar to the Diehard Battery suite.

- The TestU01 suite [LS02, LS07] is accompanied by a 200-page manual. It includes the Diehard Battery and NIST tests, and many others.

Except for the report from the tuftest suite, the test reports for those packages are much too long to show here, and a complete understanding of the reports requires more background than some readers of this book might have. However, the essential idea of the tests is that they report a special number, called a p-value, limited to the interval $[0, 1]$, and similar to the chi-square probability discussed in **Section 7.15.1** on page 197. For a good generator, p is expected to have values away from the endpoints. Normally, one should expect p values in, say, $[0.1, 0.9]$, but it is acceptable to have an occasional p value at or near the endpoints.

In the tuftest suite, the *Birthday Spacings* and the *Euclid* tests measure the randomness of successive elements in the random number sequence. The *Gorilla* test measures the randomness in 8-bit sequences within the numbers, with bit numbering from left (high-order) to right (low-order).

Here is the tuftest report for the Mersenne Twister, run on a 1.4 GHz IA-64 system:

```
Generator: mt           ngen = 0
Birthday spacings test: 4096 birthdays, 2^32 days in year
           Table of Expected vs. Observed counts:
Duplicates 0     1     2     3     4     5     6     7     8     9  >=10
```

```
Expected 91.6 366.3 732.6 976.8 976.8 781.5 521.0 297.7 148.9  66.2  40.7
Observed 87   388   755   987   950   779   511   271   154    80    38
(O-E)^2/E 0.2   1.3   0.7   0.1   0.7   0.0   0.2   2.4   0.2   2.9   0.2
        Birthday Spacings: Sum(O-E)^2/E =   8.882, p =  0.457

bday    time = 23.268 sec       calls to generator =       20480000

Euclid’s algorithm:
 p-value, steps to gcd:   0.747254
 p-value, dist. of gcd’s: 0.339220

gcd     time = 7.790 sec        calls to generator =       20000000

 Gorilla test for 2^26 bits, positions 0 to 31:
 Note: lengthy test---for example, ~20 minutes for 850 MHz PC
 Bits  0 to  7---> 0.676 0.270 0.618 0.610 0.760 0.756 0.244 0.660
 Bits  8 to 15---> 0.499 0.838 0.281 0.968 0.167 0.606 0.928 0.159
 Bits 16 to 23---> 0.077 0.828 0.871 0.831 0.166 0.403 0.746 0.969
 Bits 24 to 31---> 0.120 0.300 0.731 0.053 0.854 0.577 0.175 0.135
 KS test for the above 32 p values:  0.320

gorilla time = 484.274 sec      calls to generator =     2147483648
```

The Mersenne Twister does extremely well on those tests, because all reported p values are midrange, but many generators fail *all* of the tuftest tests.

For comparison, here are the tuftest results for the bad `randu()` congruential generator (see **Section 7.7.2** on page 170):

```
Generator: urandu               ngen = 0
Birthday spacings test: 4096 birthdays, 2^32 days in year
           Table of Expected vs. Observed counts:
Duplicates 0     1     2     3     4     5     6     7     8     9  >=10

Expected 91.6 366.3 732.6 976.8 976.8 781.5 521.0 297.7 148.9  66.2  40.7
Observed  0     0     0     0     1     3    18    53    82   144  4699
(O-E)^2/E 91.6 366.3 732.6 976.8 974.8 775.5 485.6 201.1 30. 91.6 533681.
        Birthday Spacings: Sum(O-E)^2/E = 538407.147, p =  1.000

bday    time = 22.975 sec       calls to generator =       20480000

Euclid’s algorithm:
 p-value, steps to gcd:   1.000000
 p-value, dist. of gcd’s: 1.000000

gcd     time = 7.517 sec        calls to generator =       20000000

 Gorilla test for 2^26 bits, positions 0 to 31:
 Note: lengthy test---for example, ~20 minutes for 850 MHz PC
 Bits  0 to  7---> 0.000 0.000 0.000 0.000 0.000 1.000 1.000 1.000
 Bits  8 to 15---> 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000
 Bits 16 to 23---> 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000
 Bits 24 to 31---> 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000
 KS test for the above 32 p values:  1.000

gorilla time = 249.136 sec      calls to generator =     2147483648
```

Here, all of the p-values are either 0.000 or 1.000, indicating utter failure of the tests.

The timing reports in the two output listings suggest that the Mersenne Twister runs at about half the speed of a simple congruential generator, which is probably acceptable, given its much better test results. Indeed, generator timing tests on about 40 different CPUs show relative times of the Mersenne Twister compared to `lrand48()` of 0.14 to 1.94, with an average of 0.7, making it often faster than an LCG. The KISS generator time is almost identical to the Mersenne Twister time, and KISS also passes the tuftest tests. Both pass the Diehard Battery tests as well.

7.15.3 Testing generators for nonuniform distributions

Although the generation of random numbers for nonuniform distributions is almost always based on an underlying generator for a uniform distribution, there is always the possibility that an error in the algorithm, the software, or the compilation process, will result in numbers that fail to conform to the expected distribution.

The standard test packages assume uniform distributions, so to use them for testing nonuniform generators, we need to recover uniformly distributed random numbers. Whether that is easy or not depends on the distribution. We therefore record here procedures for the three nonuniform distributions that we discussed:

- For the exponential distribution, take the negative logarithm of the generator output, because $-\log(\exp(-\text{random})) = \text{random}$.

- For the logarithmic distribution, values $\log(\text{randlog}(a,b)/a)/\log(b/a)$ are uniformly distributed on the same unit interval that `urand()` is defined on. That expression may need to be computed with three separate logarithm calls to avoid underflow or overflow.

- For the normal distribution, consider successive pairs (x, y) as a two-dimensional vector, and express in polar form (r, θ), like this:

```
r = hypot(x,y);
theta = atan2(y, x) + PI;
```

  The angle θ is then uniformly distributed in $[0, 2\pi)$, and $\theta/(2\pi)$ is in $[0, 1)$.

7.16 Applications of random numbers

In this section, we look at two important uses of random numbers in computation: *numerical integration*, and *cryptography*. The first of them involves finding areas under curves, volumes of solids, and extensions of those concepts to higher dimensions. The second is of increasing importance for privacy, reliability, and security of electronic data and electronic communications, whether personal, commercial, or governmental. A third application, *sampling*, appears many times in the remainder of this book, in plots of errors in computed functions for arguments selected from logarithmic or uniform distributions. A fourth application, *unbiased decision making* by coin toss, is demonstrated in **Section 7.14.1** on page 196. Other uses include color, intensity, and surface roughening in image processing, data coarsening for evaluation of algorithm sensitivity, testing of hardware and software data and event processing, parameter determination in function optimization, adding unpredictability to computer-based games, image selection for advertising displays, low-level noise generation for softening of audio and video signals, and so on.

7.16.1 Monte Carlo quadrature

In the Manhattan Project at Los Alamos during World War II, physicists were faced with calculating integrals that could not be evaluated analytically, or accurately by standard numerical methods. Stanisław Ulam and John von Neumann invented a simple way to handle them. Instead of using a standard quadrature formula [KMN89] of the form

$$\int_a^b f(x)\,dx \approx \sum_{k=1}^{n} w_k f(x_k)$$

for specified weights w_k and nodes x_k, they observed that because the area under a curve is just its average height times the interval width, it might be reasonable to evaluate the function at random points r_k in (a, b) (the endpoints are excluded because integrands sometimes have singularities there), and then take the average of the function values. The result is a quadrature formula like this:

$$\int_a^b f(x)\,dx \approx \frac{(b-a)}{n}\sum_{k=1}^{n} f(r_k).$$

When their work was later declassified and published [MU49], the formula was named *Monte Carlo quadrature*, after the gaming casino town of Monte Carlo in Monaco.

Later analysis showed that the convergence of the method is poor: the error drops roughly as $1/\sqrt{n}$. That means that one has to work a hundred times harder to get an extra decimal digit. Nevertheless, the error behavior does not depend on d, the number of dimensions in the integration, whereas application of standard quadrature rules requires $\mathcal{O}(n^d)$ numerical operations. Thus, for high-dimensional integration, Monte Carlo methods may be the only practical game in town.

Here is a function that implements the Monte Carlo rule:

```
double
mcrule (double a, double b, int n, double (*f) (double))
{   /* n-point Monte Carlo quadrature of f(x) on [a,b] */
    /* error Order(1/sqrt(n)) */
    double sum;
    int k;

    sum = 0.0;

    for (k = 1; k <= n; ++k)
        sum += f(a + urand()*(b - a));

    return ((b - a) * (sum / (double)n));
}
```

If necessary, premature overflow in the sum can be avoided by moving the division by `n` into the loop.

The poor convergence of Monte Carlo quadrature was improved in the 1950s by a variant known as quasi-Monte Carlo quadrature [Nie78, Nie92, BFN94]. Its error falls as $1/n$, so each additional decimal digit in the answer requires only ten times as much work. The method still requires a source of random numbers, but is too complex to treat further here.

7.16.2 Encryption and decryption

A dictionary definition of *cryptography* is the *science or study of secret writing systems, particularly codes and ciphers*. For hundreds of years, studies of the subject were mostly secret and restricted to government and military organizations, but that changed in the mid-1970s with the discovery of *public-key cryptography*.

The essential idea behind cryptography is that if one has a secret key, it can be used to *encrypt* a *plaintext message*. Knowledge of the secret key is needed to *decrypt* the *ciphertext* to recover the original plaintext.

Public-key cryptography with sufficiently long keys is believed to be secure, but that could change in the future if certain mathematical breakthroughs are made; at present, the required mathematics appears to be intractable. However, keys known to humans can be obtained by extortion, threat, or eavesdropping during their entry: no cryptographic system is ever totally secure.

7.16.2.1 Problems with cryptography

There are two fundamental problems in historical cryptography:

- The secret key must be known to both sender and receiver in order to carry out secure communications. If the two are geographically separated, such as a government headquarters and its army in the field, then it is possible that an attacker might have captured the key and used it to decipher communications, or to encipher false messages, unknown to at least one of the parties.

- If the encrypted text contains patterns, it may be possible by *cryptanalysis* to exploit those patterns to recover the plaintext, and perhaps also the secret key.

Public-key cryptography mostly solved the first of those problems, by having keys come in pairs: a public one, and a private one. Messages encrypted with one of the keys can be decrypted by the other, and vice versa. Not only does that allow secure communications, it also allows the creation of *digital signatures* that can be used to verify the authenticity of a document, either encrypted, or in plaintext.

The main problem with public-key cryptography is that it must be possible for the receiver to verify that the sender's public key is not a forgery, and that in itself is challenging. A solution in common use when the sender and receiver cannot exchange public keys in person is to have the keys certified by a trusted third party, such as a large commercial organization. However, that just transfers the problem of trust from an individual to a group, which may or may not be an improvement. Web browser security-preference settings may list many examples of such *certificate authorities*. Such sites are prime targets: some attacks on them have regrettably been successful, and resulted in issuance of significant numbers of fraudulent certificates.

7.16.2.2 A provably secure encryption method

Virtually all methods for encryption ultimately produce patterns that allow eavesdroppers to cryptanalyze and perhaps decipher messages, and possibly even determine the key. However, there is one method known that is unbreakable, because it eliminates the patterns; it is called a *one-time pad*.

The idea of the one-time pad is simple: the sender and receiver are each in possession of the same secret list of random numbers, and employ them to encrypt and decrypt messages, using each random number only *once*.

For example, if the text to be sent contains only the 26 Latin letters, assign them the numbers $0, 1, \ldots, 25$ in order, and then encrypt each in turn by adding the next random number, taking the result modulo 26, and sending the letter corresponding to that number. The receiver generates a table of remapped letters for each random number using the same procedure, and can then use the received encrypted letter to recover the original letter by simple table lookup. Because each letter is encrypted differently from every other letter, there are no correlations in the output to aid the cryptanalyst, and the encrypted message is secure.

If the communications channel can be read by eavesdroppers, then *traffic analysis* can still reveal who is communicating, and how much is being sent, compromising security.

Radio and telephone transmissions can be easily tracked to identify the sender, and network packets used for computer and telephone communication carry both source and destination addresses, or a virtual circuit identifier that can be used to recover those addresses. Global Positioning System (GPS) sensors are increasingly common, and provide geographic location records. In practice, therefore, in the design of secure communications, it must be assumed that traffic can be read, and located, by an adversary.

7.16.2.3 Demonstration of a one-time pad

Encryption and decryption with a one-time pad is tedious to do by hand, but is quite easy for a computer. For ease of presentation here, the encrypted message is further encoded in hexadecimal so that it contains only printable characters. The encryption and decryption functions use a random-number generator as a replacement for the numbers recorded on the one-time pad:

```
static void
encrypt(int key, const char * plaintext, char * ciphertext)
{
    size_t k, np, np_padded;

    np = strlen(plaintext);
    np_padded = (((np + 31) / 32) * 32);

    setseed(key);

    for (k = 0; k < np_padded; ++k)
    {
        unsigned int c, u;
        u = (unsigned int)randint(0,255);

        if (k < np)
            c = (unsigned int)plaintext[k];
        else
        {
            setseed(key + u);
            c = (unsigned int)randint(0,255);
        }

#if defined(USE_XOR_ENCRYPTION)
        c ^= u;
#else
        c = (c + u) & 0xff;
#endif

        (void)snprintf(&ciphertext[2 * k], 3, "%02x", c);
    }

    ciphertext[2 * k] = '\0';
}


static void
decrypt(int key, char * plaintext, const char * ciphertext)
{
    size_t k, nc;

    setseed(key);
    nc = strlen(ciphertext);

    for (k = 0; k < nc; k += 2)
    {
        int c;

        c = 16 * hexval((int)ciphertext[k]) + hexval((int)ciphertext[k+1]);

#if defined(USE_XOR_ENCRYPTION)
        c ^= randint(0,255);
#else
        c = (c + (256 - randint(0,255))) & 0xff;
#endif

        if (c == 0)
            break;

        plaintext[k/2] = isprint(c) ? (char)c : '?';
    }

    plaintext[k/2] = '\0';
}
```

We omit the code for the short helper function, `hexval()`, that converts a hexadecimal character to its numerical value, and for another function, `dump()`, that prints the ciphertext in four-byte hexadecimal blocks for improved readability.

The key is the seed of the generator: it can be communicated by public-key cryptography if direct key exchange in secret is not possible.

Our representation of the key as data type `int` means that on common desktop systems, there are only $2^{32} \approx 4.29 \times 10^9$ possible keys, a number that is small enough that the encryption is subject to a brute-force attack by an adversary who simply tries all possible keys. The task is made easier by a mathematical curiosity known as the *Birthday Paradox* [FS03, §3.6.6]: if there are n possible keys, and many encrypted messages, then the secret key can be guessed after about $\sqrt{n}$ attempts with random keys. Consequently, serious cryptography uses much longer keys, sometimes with 256 or more bits, to make such attacks infeasible. The subdirectory `exp` in the `mathcw` distribution contains two demonstration programs, `crack1.c` and `crack2.c`, that implement brute-force and Birthday-Paradox attacks on an n-bit key.

Our default implementation of the one-time pad uses bit-masking to provide a fast modulo operation. A widely used alternative is the exclusive-OR operation, primarily because it is a fast low-level hardware bit operation, and compile-time definition of the symbol `USE_XOR_ENCRYPTION` selects that approach in our code. If the exclusive-OR operator is not available in the programming language, then library functions might allow use of code similar to either of these assignments:

```
c = xor(c, randint(0, 255));      c = xor(c, and(rand(), 0xff));
```

The encryption function uses the C string terminator character, `NUL`, to implicitly pad the message to a length that is a multiple of 32 characters, but when it does so, it changes the key for every such character to obscure the padding. In practice, the padding length would be chosen much larger, perhaps a few thousand characters, so that most messages would have the same ciphertext length. Uniform message lengths make it harder for an attacker to guess the contents: otherwise, the length of a short message containing either *yes* or *no* would reveal the contents without decryption.

Here is a demonstration of how the encryption works at the sender. First, we encrypt some short messages and show the ciphertext following each statement. The encryption does not reveal message length or where the padding characters are, but it *does* reveal common plaintext prefixes:

```
encrypt(123, "A", ciphertext);
ded04643 a57468bc f6dbd17a a3d5ccf4 f20940be 5fe32b6f 61028717 d5ccf4f2

encrypt(123, "AB", ciphertext);
de3caafb 61028717 d5ccf4f2 0940be5f e32b6f61 028717d5 ccf4f209 40be5fe3

encrypt(123, "ABC", ciphertext);
de3cc9da 536d783a 4d325fe3 2b6f6102 8717d5cc f4f20940 be5fe32b 6f610287

encrypt(123, "ABCD", ciphertext);
de3cc96a a122975f 19be9576 5184b000 68bcf6db d17aa3d5 ccf4f209 40be5fe3
```

The common prefix is easily concealed by changing the key with each message: the bits in the next random number from the one-time pad can be manipulated with logical or arithmetic operations to create the new key. However, for simplicity here, we keep the key constant.[3]

The encryption does not reveal letter repetitions, showing that the one-time pad encrypts the same letter differently on each occurrence:

```
encrypt(123, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA", ciphertext);
de3bc767 b9855e5d 9bdef442 649ece44 d45b654b 1aacfe8c fb08e73b 9bcca6a9
```

Now encrypt a famous message from American revolutionary history:

```
encrypt(123, "One if by land, two if by sea: Paul Revere’s Ride, 16 April 1775", ciphertext);
ec68eb46 e1aa3d7e d3bd1f62 91c1b923 0791932a 42d1ddad 33e7195f bbc52e6c
0c8fee9a c1837616 5d52e0da 4a5ad1fc 4fe96a3f 5e0cfa5f b54fd590 00a874dd
f8e7c67e 917e5ac4 b9a4cd5e 2981cf9a 1c9fe2a4 dba4f612 e43da39a b61702e3
```

The receiver uses the decryption function to recover the original message from the ciphertext. Three different keys are tried, but only the correct one recovers the original plaintext, and the others produce unintelligible decryptions:

[3] Modern practice is to change encryption keys often. For example, some secure wireless network protocols require key changes at least once per hour, and in addition, combine the base key with a serial number that is incremented with every network packet transmitted.

```
decrypt(122, plaintext, ciphertext);
?????c??

decrypt(123, plaintext, ciphertext);
One if by land, two if by sea: Paul Revere’s Ride, 16 April 1775

decrypt(124, plaintext, ciphertext);
?:??Kj????AI
```

The failed decryptions are short because the incorrect key resulted in premature generation of the string-terminator character.

7.16.2.4 Attacks on one-time-pad encryption

Although the one-time pad is completely secure against attack by an eavesdropper on the communications channel, it is still subject to attack by capture of the one-time pad. Also, the pad needs to be larger than the total volume of encrypted traffic, and that may be difficult to predict in advance. If the pad is used up, then secure communication is no longer possible.

If we use a computer algorithm to generate the random numbers, as in `encrypt()` and `decrypt()`, instead of taking them from a truly random stream on a one-time pad, then there are necessarily relations between successive random numbers that may make cryptanalysis possible. For example, for *any* polynomial congruential generator, the generator parameters can be recovered from knowledge of a subsequence of the stream [Ree77, Ree79, Knu85, Boy89, Sch96, Mar03a], even if the sender has tried to make the job harder by discarding a few of the low-order bits, which are known to be less random than high-order bits in LCG streams. There is no similar published analysis for the Mersenne Twister generator, but that does not mean that no such study has been done: secret agencies do not publish their research.

One approach that has been proposed is to further scramble the bits of each random number from an LCG used for the one-time pad. However, it is probably better instead to use a superior generator.

The huge periods of good generators like the Mersenne Twister make it impossible to exhaust the pad, solving one of the big problems of the one-time pad. However, it is likely that any important computer algorithm for generating random numbers has already been studied, or will be in the future, with a view to cracking its use for a one-time pad. The necessary key sharing between sender and receiver also remains an avenue of attack.

7.16.2.5 Choice of keys and encryption methods

Our `encrypt()` and `decrypt()` functions are examples of *symmetric key* cryptography: the same key serves in both directions. Most fast algorithms for encryption have that symmetric-key property. Public-key cryptography does not, because its keys come in public/private pairs. Apart from one-time pads based on truly random sequences, public-key methods provide the currently strongest algorithms for protection against attacks. If the keys are sufficiently long, practical attacks are believed to be infeasible.

Unfortunately, processing long keys requires multiword integer arithmetic, which is much slower than ordinary arithmetic. Although several vendors offer specialized hardware assists called *cryptographic accelerators*, for most purposes, public-key cryptography is impractical for high-volume data communications. Instead, most systems use public-key methods just for the secure initial exchange of dynamically generated symmetric keys.

The symmetric keys are usually generated either from special devices that use physical phenomena, such as electrical noise or radioactive decay, to produce random streams of bits, or from software simulations of such devices. Several flavors of UNIX now support two software devices called `/dev/random` and `/dev/urandom` that return random bytes when read [dRHG^{+}99]. The first is usually cryptographically stronger, and may not return until sufficient randomness has been gathered from other activities in the computer. However, that makes it subject to *denial-of-service attacks*: repeated reads of the device by one process can prevent another process for ever getting enough data from the device. The second device may be able to return random data immediately each time it is read, but to do so, it may have to resort to invoking random-number algorithms, possibly making its output more predictable.

7.16.2.6 Caveats about cryptography

We introduced the one-time pad example because it is an interesting application of random numbers to an important problem in cryptography. However, the reader is cautioned that the field of cryptography is difficult and rapidly changing. Toy encryption methods like our `encrypt()` should *never* be used for the protection of real data.

The best advice that can be given for use of encryption is to stick to well-studied and well-tested methods, such as the *Advanced Encryption Standard (AES)*: it is unlikely that any good cryptographic methods will be discovered by amateurs. See *In Code* [FF01] for an initially promising algorithm that was later shown to be flawed.

In seeking off-the-shelf software solutions for encryption, remember that *security by obscurity*, practiced by disreputable software vendors who claim uncrackable encryption from use of 'secret' algorithms, is evidence of cryptographic incompetence: avoid them and their software.

Modern cryptographic methods are good enough that they are unlikely to be crackable by brute force. Thus, a forgotten or lost encryption key for an encrypted file or filesystem probably means permanent loss of data. Organizations can suffer denial of access to data if a critical employee leaves with the only copy of the key, and they can be blackmailed by the key holder. Once keys are entered into the computer, it may be necessary to retain them in memory for an extended period, such as while remote communications are in progress, or while a filesystem is accessed. The key memory is normally kept inside the operating system where it cannot be accessed by ordinary user processes. However, sometimes software security holes are discovered that make such access possible. It has recently been demonstrated that even after a computer is powered down, memory dumps can recover keys for some time, and that time can be extended by cooling the memory chips [HSH+09]. A stolen portable computer with an encrypted filesystem can thus be subject to key recovery and data theft.

There are many other issues in this area that we could discuss here, but we leave them to other books listed in Section 7.18 on page 214.

7.17 The mathcw random number routines

Marsaglia's KISS generator produces uniformly distributed random integers for the mathcw library. The KISS kernel is small and fast, provides a full 32-bit unsigned integer range with an acceptably long period of about $2^{123} \approx 10^{37}$, and passes most tests of randomness.

Three statements in the KISS code require 64-bit unsigned arithmetic, but when support for that is lacking, we fall back to using software routines already developed for the library.

The code for the generator looks like this:

```
#define DEFAULT_C               ((UINT_LEAST32_T)129281L)
#define DEFAULT_JSR             ((UINT_LEAST32_T)362436069L)
#define DEFAULT_X               ((UINT_LEAST32_T)123456789L)
#define DEFAULT_Y               ((UINT_LEAST32_T)871119182L)

static UINT_LEAST32_T c         = DEFAULT_C;
static UINT_LEAST32_T jsr       = DEFAULT_JSR;
static UINT_LEAST32_T x         = DEFAULT_X;
static UINT_LEAST32_T y         = DEFAULT_Y;

UINT_LEAST32_T
lrancw(void)
{   /* return random unsigned integer on [0,2^{32} - 1] */
#if defined(HAVE_UINT_LEAST64_T)
    const UINT_LEAST64_T a = (UINT_LEAST64_T)333333314L;
    UINT_LEAST64_T t;
#else
    const UINT_LEAST32_T a = (UINT_LEAST32_T)333333314L;
    UINT_LEAST32_T t[2];
#endif

    UINT_LEAST32_T result;
    y = (UINT_LEAST32_T)69069L * y + (UINT_LEAST32_T)12345L;

#if MAX_UINT_LEAST32 != 0xffffffffL
    y &= (UINT_LEAST32_T)0xffffffffL;
#endif

    jsr ^= (jsr << 13);
    jsr ^= (jsr >> 17);
    jsr ^= (jsr << 5);

#if defined(HAVE_UINT_LEAST64_T)
    t = a * x + c;
    c = (t >> 32);
    x = t + c;
#else
    umul64(t, a, x);
    uadd32(t, c);
    c = t[0];
    uadd32(t, c);
    x = t[1];
#endif

    if (x < c)
    {
        x++;
        c++;
    }

    x = ~x + 1;

#if MAX_UINT_LEAST32 != 0xffffffffL
    x &= (UINT_LEAST32_T)0xffffffffL;
#endif

    result = x + y + jsr;

#if MAX_UINT_LEAST32 != 0xffffffffL
    result &= (UINT_LEAST32_T)0xffffffffL;
#endif

    return (result);
}
```

The interface requires an unsigned integer data type of at least 32 bits, but C89 guarantees only a minimum of 16 bits for an `int` and 32 bits for a `long int`. Most desktop computer systems now provide a 32-bit `int`, but the size of a `long int` is 32 bits on some systems, and 64 bits on others. C99 deals with that by introducing several new integer types that provide at least a specified number of bits. The new types that are useful for the KISS generator are `uint_least32_t` and `uint_least64_t`, but because those names are not recognized by older compilers, we use uppercase equivalents defined with `typedef` statements in our header file `inttcw.h` that is always included by `randcw.h`, and that in turn is included by `mathcw.h`. If only the random-number support is required, then the header file `mathcw.h` need not be included.

The preprocessor conditionals clutter the code, but are needed to deal with systems that lack an unsigned integer type of at least 64 bits, and machines with wordsizes other than 32.

The state of the generator is represented by four unsigned 32-bit variables (`c`, `jsr`, `x`, and `y`) that are updated on each call, so the simplicity of the single seed of the LCG and MCG is lost. However, most random-number algorithms that pass the test suites also require a multiword state.

A second function combines two 32-bit random values to make a 64-bit value:

```
UINT_LEAST64_T
llrancw(void)
{   /* return random unsigned integer on [0,2^{64} - 1] */
    UINT_LEAST64_T n;

    n = (UINT_LEAST64_T)lrancw();

    return ((n << 32) | lrancw());
}
```

However, it is only usable when a 64-bit integer type is natively supported by the compiler, as is required for C99.

Because software random-number generators do not normally return the same value repeatedly, the combination of two 32-bit values can produce neither a zero 64-bit value, nor the extreme value $2^{64} - 1 =$ `0xffff ffff ffff ffff`. Tests of the KISS generator show that duplicate return values *are* possible, though they occur only about once per 32-bit cycle. The period of the KISS generator is so large that it is impossible to perform an exhaustive test to determine the true extrema for 64-bit, 128-bit, and 256-bit sequences. In practice, that means that floating-point results on the unit interval for data type `double` and longer always lie in $(0, 1)$, with the smallest `double` value being about $2^{-64} \approx 5.42 \times 10^{-20}$, and the largest about $1 - \epsilon/\beta$.

Although the significand size of the `float` type is similar across current and most historical architectures, and the same is true for the `double` type, the significand of `long double` can have as few as 64 bits, or as many as 113 bits. We could construct the shorter significand from two 32-bit values, but the longer sizes would need four of them. Our `long double` functions always use four 32-bit values to guarantee the same sequences (within rounding error) on all platforms.

Much of the random-number support in the mathcw library is provided by constants and routines with these definitions and prototypes:

```
static const UINT_LEAST32_T LRANDCW_MAX = (UINT_LEAST32_T)0xffffffffL;
static const UINT_LEAST64_T LLRANDCW_MAX = (UINT_LEAST64_T)0xffffffffffffffffLL;

UINT_LEAST32_T lrincw    (UINT_LEAST32_T a, UINT_LEAST32_T b);
UINT_LEAST64_T llrincw   (UINT_LEAST64_T a, UINT_LEAST64_T b);

void           gscw (randcw_state_t state);
void           sscw (const randcw_state_t state);

int            sccw (void);

unsigned int   mscw (void);
UINT_LEAST32_T lmscw (void);
UINT_LEAST64_T llmscw (void);

double ercw    (void);
double lrcw    (double a, double b);
double nrcw    (void);

double urcw    (void);
double urcw1   (void);
double urcw2   (void);
double urcw3   (void);
double urcw4   (void);

void vlrancw   (int n, UINT_LEAST32_T u[]);
void vllrancw  (int n, UINT_LEAST64_T u[]);
void vlrincw   (int n, UINT_LEAST32_T u[], UINT_LEAST32_T a, UINT_LEAST32_T b);
void vllrincw  (int n, UINT_LEAST64_T u[], UINT_LEAST64_T a, UINT_LEAST64_T b);

void vercw     (int n, double u[]);
void vlrcw     (int n, double u[], double a, double b);
void vnrcw     (int n, double u[]);
void vurcw     (int n, double u[]);
void vurcw1    (int n, double u[]);
void vurcw2    (int n, double u[]);
void vurcw3    (int n, double u[]);
void vurcw4    (int n, double u[]);
```

We omit the prototypes of the family members that produce floating-point values in nine other types: their names have the usual type suffixes. The function names all contain the letters `cw` to make conflicts with names of other generators unlikely.

The internal state means that most of those functions are not thread safe: simultaneous updates of the state in two or more threads could change the sequence of random numbers. To solve that problem, function variants whose names end in `_r`, for *reentrant*, store the generator state in a user-provided thread-unique array:

```
void           incw_r (randcw_state_t state, UINT_LEAST32_T seed);

UINT_LEAST32_T lrancw_r  (randcw_state_t state);
UINT_LEAST64_T llrancw_r (randcw_state_t state);

UINT_LEAST32_T lrincw_r  (randcw_state_t state, UINT_LEAST32_T a, UINT_LEAST32_T b);
UINT_LEAST64_T llrincw_r (randcw_state_t state, UINT_LEAST64_T a, UINT_LEAST64_T b);

double ercw_r   (randcw_state_t state);
double lrcw_r   (randcw_state_t state, double a, double b);
double nrcw_r   (randcw_state_t state);

double urcw_r   (randcw_state_t state);
double urcw1_r  (randcw_state_t state);
double urcw2_r  (randcw_state_t state);
double urcw3_r  (randcw_state_t state);
double urcw4_r  (randcw_state_t state);

void vlrancw_r  (randcw_state_t state, int n, UINT_LEAST32_T u[]);
void vllrancw_r (randcw_state_t state, int n, UINT_LEAST64_T u[]);

void vllrincw_r (randcw_state_t state, int n, UINT_LEAST64_T u[], UINT_LEAST64_T a, UINT_LEAST64_T b);
void vllrincw_r (randcw_state_t state, int n, UINT_LEAST64_T u[], UINT_LEAST64_T a, UINT_LEAST64_T b);

void vercw_r    (randcw_state_t state, int n, double u[]);
void vlrcw_r    (randcw_state_t state, int n, double u[], double a, double b);
void vnrcw_r    (randcw_state_t state, int n, double u[]);

void vurcw_r    (randcw_state_t state, int n, double u[]);
void vurcw1_r   (randcw_state_t state, int n, double u[]);
void vurcw2_r   (randcw_state_t state, int n, double u[]);
void vurcw3_r   (randcw_state_t state, int n, double u[]);
void vurcw4_r   (randcw_state_t state, int n, double u[]);
```

Once again, we have omitted prototypes for the family members of other supported floating-point types. Type suffixes *precede* the reentrant suffix, so the generator that returns a `decimal_float` result on $[0, 1]$ is called `urcw1df_r()`.

The generator state is given its own data type in `randcw.h` with this simple definition:

```
typedef UINT_LEAST32_T randcw_state_t[4];
```

That makes `randcw_state_t` an array of four unsigned integers, and an integer array is easy to supply from languages that can call routines written in C. Support for unsigned integers is lacking in most other programming languages, but that does not matter here, because a signed type requires the same storage, and the state variables rarely need to be set or examined in the calling program. Most C or C++ code that uses the reentrant routines can treat the data type as if it were opaque, without ever needing to know the size of the state array, which holds the KISS state variables `c`, `jsr`, `x`, and `y`, in that order.

The functions `lrincw()` and `llrincw()` return random integers in the range $[a, b]$, with code similar to that illustrated in `urandtoint()` in **Section 7.4** on page 165.

The functions `gscw()` and `sscw()` get and set the state of the generator. An initial invocation of `gscw()` before any other family members are called can provide a suitable default state for the reentrant functions that is identical across separate runs of the job. Alternatively, one can ask one of the functions `mscw()`, `lmscw()`, and `llmscw()` to make and return a unique seed with code similar to that shown in function `makeseed()` in **Section 7.1** on page 157. The seed-maker functions are independent of each other, and of the random-number generator, and do not affect the generator internal state. User code could employ one of them to initialize the generator differently on each run like this:

```
randcw_state_t state;

incw_r(state, lmscw());
sscw(state);
```

The second argument of the function `incw_r()` acts as a seed to initialize one of the components of the state, and the others are filled with the generator's default values. The call to `sscw()` copies the external state into the internal state used by the nonreentrant routines.

The function `sccw()` performs a run-time sanity check. It saves the generator state with `gscw()`, sets it to a known state with `sscw()`, then uses `lrancw()` to sample the generator, discarding the first one hundred values, and verifying that the next ten values are as expected. The check is then repeated with `lrancw_r()`. If a 64-bit integer type is supported, `sccw()` makes similar checks using `llrancw()` and `llrancw_r()`. Finally, it calls `sscw()` to restore the original state of the generator at entry, and returns the number of mismatches, which should be zero if all is well.

The reentrant functions solve the thread problem, and they also satisfy one of the important goals listed at the beginning of this chapter, by making it possible to have *families of generators*, each with its own state. For example, a program could have initializing code like this:

```
#define NFAMILY 1000

randcw_state_t family[NFAMILY];

for (k = 0; k < NFAMILY; ++k)
  incw_r(family[k], lmscw());
```

A subsequent reference to `lrancw_r(family[k])` would then return a random integer from the k-th generator family, without affecting the sequences produced for other families. The period of the generator is so large that there is negligible chance of separate families producing identical sequences of even a few elements.

The reentrant functions also solve the problem of how to provide a block of code with its own private generator without having to save and restore state:

```
{
    randcw_state_t state;

    incw_r(state, lmscw());

    for (m = 0; m < MAXTEST; ++m)
      test(lrancw_r(state));
}
```

The functions `ercw()`, `lrcw()`, and `nrcw()` return values from exponential, logarithmic, and normal distributions, respectively, using the von Neumann, `randlog()`, and polar methods. The two arguments to `lrcw()` define the range from which samples are taken, and that range must not include zero. Both endpoints of the range can be reached by the generator.

The functions `urcw1()` through `urcw4()` return values from a uniform distribution on the unit intervals $[0, 1]$, $[0, 1)$, $(0, 1]$, and $(0, 1)$, respectively. The function `urcw()` is recommended for general use; it is equivalent to `urcw4()`.

In view of the scaling issue that we raised in **Section 7.3** on page 160, it is worthwhile here to document exactly how that problem is handled in two sample routines, one for `float`, and one for `long double`. The first of them looks like this:

```
#define _MAXRANL 0xffffffffL

static const float FLT_MAXRANL_INV = (float)1.0 / (float)_MAXRANL;

float
(urcw2f)(void)
{                                       /* result in [0,1) */
    float result;

    result = (float)lrancw() * FLT_MAXRANL_INV;

    if (result >= (float)1.0)           /* rare, but necessary */
        result = nextafterf((float)1.0, (float)0.0);

    return (result);
}
```

The `float` significand contains fewer than 32 bits, so the conversion of the limit `_MAXRANL` loses trailing bits. When the random integer from `lrancw()` is scaled by the stored reciprocal of that value, depending on the rounding behavior and mode, the result could be at, or slightly above, 1.0. The code must therefore check for that unlikely case, and reduce the result to $1 - \epsilon/\beta$. The `nextafterf()` function provides an easy way to get that value, whereas a stored constant would require separate definitions for each floating-point architecture. We could also have used simpler code for the exceptional case, using the fact that $1 - u$ is uniformly distributed if u is:

```
    if (result >= (float)1.0)           /* rare, but necessary */
        result -= (float)1.0;
```

The `long double` code samples four random integers, and reduces an out-of-range result:

```
#define _MAXRANL 0xffffffffL

static const long double LDBL_MAXRANLL_INV = 1.0L /
       ((long double)_MAXRANL * _TWO_TO_96 + (long double)_MAXRANL * _TWO_TO_64 +
        (long double)_MAXRANL * _TWO_TO_32 + (long double)_MAXRANL);

long double
(urcw2l)(void)
{                                       /* result in [0,1) */
    long double result;

    result = (long double)lrancw();
    result = (long double)_TWO_TO_32 * result + (long double)lrancw();
    result = (long double)_TWO_TO_32 * result + (long double)lrancw();
    result = (long double)_TWO_TO_32 * result + (long double)lrancw();
    result *= LDBL_MAXRANLL_INV;

    if (result >= 1.0L)                 /* rare, but necessary */
        result = nextafterl(1.0L, 0.0L);

    return (result);
}
```

In the IEEE 754 default rounding mode, a result above 1.0 is not possible, but that is not the case in other IEEE 754 rounding modes, and in some historical systems. In any event, a value of 1.0 must be reduced to fit the documented range, and the reduction introduces a slight perturbation in the probability of getting that endpoint result.

The vector functions `vurcw1()` through `vurcw4()` take an array size and an array, and return the requested number of random values from a uniform distribution. The function `vurcw()` is equivalent to `vurcw4()`. All of those functions transform the results of the vector integer function `vlrancw()`, which contains a copy of the generator algorithm wrapped inside a loop to remove function call overhead. The four internal state variables are shared by the scalar and vector generators. Timing tests show that the vector functions provide a modest speedup of about 1.1 to 1.3 for binary arithmetic, and 5 to 10 for decimal arithmetic.

7.18 Summary, advice, and further reading

THE GENERATION OF RANDOM NUMBERS IS TOO IMPORTANT TO BE LEFT TO CHANCE.

— ROBERT R. COVEYOU
OAK RIDGE NATIONAL LABORATORY.

The last two decades of the millennium saw great progress in the mathematical theory behind random-number generation algorithms, as well as the development of software packages that provide reliable tests of the quality of random numbers.

As a general rule, one should avoid random-number algorithms published before about 1990, because almost all have been shown to be deficient, and much better methods are now known. Indeed, a 1988 survey of random-number algorithms recommended in more than 50 textbooks [PM88] found that the recipes were *all* inferior to a portable 32-bit MCG [Sch79b] with $A = 7^5 = 16\,907$ and $M = 2^{31} - 1$, and their periods, portability, and quality varied from bad to dreadful.

If you select a particular random-number generator for serious work, revisit the guidelines given in **Section 7.1** on page 157 and see how many of them are followed by your choice. Before you commit to it, validate your selection with one or more of the test suites discussed in **Section 7.15.2** on page 200, and preferably, do so on different computer architectures. Also, make sure that both the period and the range are adequate for your applications, and be careful about the handling of range boundaries.

Because random numbers are used in many computational disciplines, the literature on random-number generation is huge, and it is often difficult to find, or recognize, the important publications. As a useful guide, the journals *ACM Transactions on Mathematical Software*, *ACM Transactions on Modeling and Computer Simulation*, *Applied Statistics*, *Computer Physics Communications*, and *Journal of Computational Physics* are generally reliable sources of reputable research in this area. There are complete online bibliographies for all of those journals in the archives cited at the end of this section, and a Web search should find publisher sites that may offer article search and retrieval services.

For textbook treatments, the definitive treatise on generation and testing of random numbers is a volume of *The Art of Computer Programming* [Knu97, Chapter 3]. It should be augmented with more recent work, such as the description of the TestU01 suite [LS07], which includes test results for many popular generators. Another good source with a strong emphasis on applications in statistics is Gentle's book [Gen03]. For discussions of the problems of generating truly random numbers, which is of extreme importance for cryptography and thus, the security of modern communications, computers, and networks, see *Practical Cryptography* [FS03, Chapter 10], *Cryptography Engineering* [FSK10, Chapter 9], *RFC 4086* [ESC05], and Gutmann's work [Gut04, Chapter 6].

Cryptography is a fascinating and important subject, but its modern research requires deep understanding of mathematical number theory. Nevertheless, a readable, and highly recommended, presentation of the field can be found in *The Code Book* [Sin99], and its impact on society is well treated in *Secrets and Lies* [Sch00]. The latter should be read *before* its predecessor, *Applied Cryptography* [Sch96], which gives the details of cryptographic algorithms, because *Secrets and Lies* was written to dispel the widely held belief that cryptography is the complete answer to data security.

There are extensive bibliographies at

`http://www.math.utah.edu/pub/tex/bib/index-table.html`

that record most of the important publications in the field; look for those with *crypt* and *prng* in their names. One of the journals in that collection is *Cryptologia*, which contains research articles on the long and interesting history of cryptography. There is a bibliography of George Marsaglia's collected works at

`http://www.math.utah.edu/pub/bibnet/authors/m/`.

He is one of the greatest experts on random numbers, and his ideas and advice are worthy of study.

Finally, remember that random-number generation is an area of computing where for most of its history, widespread practices, software libraries, and textbook recommendations, have later been shown to be severely deficient. When it matters, check with experts, and the recent literature of the field.

8 Roots

I cannot delve him to the root.

— Shakespeare's *Cymbeline* (about 1611).

The *root* of any quantity is such a quantity as,
when multiplied by itself a certain number of times,
will exactly produce that quantity.

— *New Century Dictionary* (1914).

The character marking the root is $\sqrt{\ }$ (a modification
of *r* for *radix*, which has been used probably
since the middle of the Sixteenth Century).

— *New Century Dictionary* (1914).

The simplest of the elementary functions are those that compute roots of numbers. They are most easily calculated by a combination of argument range reduction, polynomial approximation to get an accurate starting value, and a rapidly convergent iterative solution. Some computer architectures include an instruction to compute the square root, because the required circuits can often be shared with those of the divide instruction.

The IEEE 754 Standard mandates a *correctly rounded* square-root operation as one of the basic five arithmetic operations on which all numerical computation is built. We show how to achieve that requirement in software, and we approach that goal for other roots. Near the end of this chapter, we sketch how to take advantage of hardware implementations of the square root.

8.1 Square root

The square root may be the most common elementary function, and it can be calculated quickly, and to high accuracy, by Newton–Raphson iteration. Cody and Waite first reduce the argument of `sqrt(x)` to the representation $x = f \times \beta^n$, with f in $[1/\beta, 1)$. The solution of $y = \texttt{sqrt}(x)$ with $f(y) = y^2 - x$ for a given fixed x comes from a few iterations of $y_{k+1} = \frac{1}{2}(y_k + x/y_k)$. The starting estimate y_0 is computed from a $\langle 1/0 \rangle$ polynomial approximation to $\sqrt{x}$ for x in $[1/\beta, 1)$.

Cody and Waite observe that if two successive iterations are combined and simplified, one multiply disappears. The double-step iteration computes $z = y_k + f/y_n$ followed by $y_{k+2} = \frac{1}{4}z + f/z$. Each double step *quadruples* the number of correct bits with just two adds, two divides, and one multiply, and the multiply is exact for bases 2 and 4. Because the precision is fixed at compilation time, the number of steps can be predetermined as well, so the code is free of loops.

If n is odd, the final y is multiplied by the constant $1/\sqrt{\beta}$ and n is replaced by $n + 1$. The final function value is then obtained by adding $n/2$ to the exponent of y.

After implementing and testing the Cody/Waite recipe, it occurred to this author that the multiplication by $1/\sqrt{\beta}$ for odd n can be eliminated by moving the parity test on n before the iterations. If n is odd, replace f by f/β (an exact operation) and increment n. Now n is guaranteed to be even, and the post-iteration adjustment factor $1/\sqrt{\beta}$ is no longer required. However, the range of f is widened from $[1/\beta, 1)$ to $[1/\beta^2, 1)$, so a new polynomial approximation is required. Maple found a $\langle 2/0 \rangle$ polynomial that provides eight correct bits for base 2, a $\langle 2/1 \rangle$ polynomial for two digits in base 10, and a $\langle 2/2 \rangle$ polynomial for eight bits in base 16. The rational polynomials are rescaled to make the high-order coefficient in the denominator exactly one, eliminating a multiply. Because the computed result is only a starting guess, the effects of wobbling precision in hexadecimal normalization are not of concern here. One double-step iteration suffices for IEEE 754 32-bit arithmetic, and two double steps handle the three other IEEE 754 precisions. Two, three, and four steps handle the decimal precisions.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_8

The algorithm change improves accuracy: the ELEFUNT test results for the revised `sqrt()` function showed that the worst-case error dropped by 0.5 bits, which is the reduction expected in rounding arithmetic from elimination of the final multiplication by the adjustment factor, $1/\sqrt{\beta}$.

The Newton–Raphson iteration is self-correcting, so the error in the computed result arises entirely from the five floating-point operations in the last double step. At convergence, we expect to have $y \approx \sqrt{f}$, $z \approx 2\sqrt{f}$, and the final result is computed as $y = \frac{1}{2}\sqrt{f} + \frac{1}{2}\sqrt{f}$. With *round-to-nearest* arithmetic, the worst-case error is therefore expected to be 2.5 bits, and the average error close to 0.0 bits. In the next two sections, we show how to eliminate those errors entirely.

8.1.1 Considerations for rounding of the square root

Given the simplicity of the square-root function algorithm, we can ask how often the computed result can be expected to be correctly rounded according to the rules described in **Section 4.6** on page 66, and when it is not, whether the result can be adjusted to the correct value. That question was first studied by Hull and Abrham [HA85] in the context of decimal arithmetic in a programming language that supports variable-precision arithmetic. They found that as long as the precision is at least three decimal digits, then carrying out the final iteration with two additional decimal digits is *almost* sufficient to guarantee correct rounding. However, a fixup is still required: if the square of the floating-point number $\frac{1}{2}$ ulp below the computed result exceeds x, reduce the computed result by 1 ulp. Similarly, if the square of the value $\frac{1}{2}$ ulp above the computed result is below x, increase the result by 1 ulp. That fixup requires higher precision, so it may be impractical with hardware arithmetic, but we revisit that issue shortly. Numerical experiments showed that with four decimal digits, the pre-fixup results were 98.15% exact, 1.46% low, and 0.39% high.

A pragmatic approach to determining whether function results are correctly rounded is to try all possible values of the reduced f, and compare the computed square root with one computed in higher precision. The square roots of other x values computed by our algorithm differ only by an exact scale factor from those of f, so only the values of f need be considered.

In 32-bit IEEE 754 binary arithmetic, there are 24 bits in the significand, but the first bit is always one for normal numbers. Thus, we should expect $2^{23} = 8\,388\,608$ values of f for a given exponent. The interval $[\frac{1}{4}, 1)$ corresponds to two different exponent values, so there are twice that many f values, 16 777 216, to consider. The test program `rndsq1.c` implements a rounding-error test of `sqrtf()`, and can be run like this (here, with GNU/Linux on a 2.0 GHz AMD64 processor):

```
% cc rndsq1.c -L.. -lmcw && time ./a.out
Tests of rounding error in sqrtf(x)
Total tests      = 16777216
sqrtf() exact    = 12582107 (75.00%)
sqrtf() low      =  2093860 (12.48%)
sqrtf() high     =  2101249 (12.52%)
7.492u 0.000s 0:07.49 100.0%    0+0k 0+0io 0pf+0w
```

In the tests, care has to be taken to ensure that the compiler does not use a hardware instruction or a built-in function instead of the `sqrtf()` function from the `mathcw` library: compilation may require suppression of optimization, or a flag like the `gcc` compiler's `-fno-builtin` option.

Tests on systems with Alpha, AMD64, IA-64, MIPS, PA-RISC, PowerPC, and SPARC processors produced results identical to those shown in the sample run. However, tests on GNU/Linux IA-32 reported 87.59% correctly rounded, 6.14% low, and 6.27% high, and on Solaris 10 IA-32, 99.96% correctly rounded, none low, and 0.04% high. Those differences are likely due to use of 80-bit registers on IA-32; such differences were not observed on IA-64, which also has long registers, but more nearly correct rounding.

On an old Motorola 68040 system, the tests reported that 70.58% are correctly rounded, 25.40% are low, and 4.01% are high. The differences from other IEEE 754 systems are due to the compiler's generating only single-precision arithmetic instructions for intermediate expressions, instead of slightly more costly higher-precision instructions.

A test on the 36-bit DEC PDP-10 (hardware retired, but emulated in the KLH10 simulator) showed quite different results: 42.24% exact, none low, and 57.76% high. The larger 27-bit significand requires eight times as many f values as on 32-bit systems, and the test took 6765 seconds, or about 25.2 microseconds per call to a square-root function.

Test times on real hardware ranged from about 8 seconds on this author's fastest machine, to 385 seconds on the slowest. Clearly, a similar exhaustive numerical experiment to test the double-precision `sqrt()` function would be impractical: there are 2×2^{52} possible f values, so the experiment would take at least $8 \times 2^{53}/2^{24} \approx 4 \times 10^9$ seconds, or more than 136 years. An exhaustive test of the 128-bit `sqrtl()` would need much longer than the age of the universe.

For double and higher precision, testing would have to be done with random sampling, or perhaps with sparse uniform distributions, or using arguments of reduced precision. For the latter, we choose test arguments that are exactly representable squares of numbers with half as many bits. For example, with the IEEE 754 64-bit format and its 53-bit significand, we can use 26-bit numbers to form exact 52-bit squares. The test programs `rndsq3.c` and `rndsq4.c` use that technique to test `sqrt()` and `sqrtl()`, comparing the returned results with the known exact values. The test arguments for 64-bit and 128-bit arithmetic always have one or more trailing zero bits, which is undesirable because not all significand bit positions are tested with both zeros and ones. That deficiency can be remedied by adding a small perturbation, and computing the exact square root via a truncated Taylor series:

$$\sqrt{x^2 + \delta} = x + \delta/(2x) - \delta^2/(8x^3) + \delta^3/(16x^5) - \cdots.$$

Choosing $\delta = \frac{1}{2}\epsilon$ sets the last or second-last bit of the argument in the interval $[\frac{1}{4}, 1)$, and allows the series to be evaluated to machine precision from just the first two terms. Rounding errors from the division in $\delta/(2x)$ do not change the computed sum. Selecting $\delta = -\frac{1}{2}\epsilon$ inverts most of the trailing argument bits. The programs `rndsq5.c` and `rndsq6.c` implement such tests for `sqrtl()`.

Hull and Abrham's experiments with two extra decimal digits of working precision gave 98.15% exact results compared to our 75.00%, showing the benefits of a little more precision. Indeed, on most desktop computers, we could implement the single-precision square root with double-precision iterations at negligible cost, and almost always get a correctly rounded result. Tests of `sqrtf()` with that modification produced 99.96% exact, none low, and 0.04% high, using either `sqrt()` or `sqrtl()` as the comparison value. However, the double-precision square root would need even higher precision, and that is only cheaply available on AMD64, EM64T, IA-32, IA-64, IBM System/390 G5, and the now-obsolete Motorola 68000. For `sqrtl()`, extended precision in software is likely to be required.

Although higher intermediate precision helps, it still does not guarantee correct rounding of the square-root function, it is sometimes computationally expensive, and it is not universally available. In the next section, we show how for some platforms, at least, we can solve the problem properly.

8.1.2 An algorithm for correct rounding of the square root

The correctly rounded value of the square root is an exactly representable number $\bar{y}$ that lies within a half ulp of the infinite-precision result:

$$\sqrt{x} - \tfrac{1}{2}u \le \bar{y} \le \sqrt{x} + \tfrac{1}{2}u.$$

Because u is the spacing between consecutive floating-point numbers, the interval of width u normally contains only a single unique representable floating-point number. However, if the endpoints are themselves exactly representable, then it would appear that the interval contains two machine numbers, and that $\bar{y}$ must be precisely one of them. That cannot happen, because with a t-bit significand, it would require that $\sqrt{x} \pm \frac{1}{2}u$ be representable in t bits. That is only possible if $\sqrt{x}$ requires exactly $t + 1$ bits. Its square, x, then needs $2t + 2$ bits, contradicting the initial condition that x is exactly representable in t bits.

We only need to deal with reduced values in the range $[\frac{1}{4}, 1)$, so suitable values of u are a quarter of the machine epsilon for the range $[\frac{1}{4}, \frac{1}{2})$ and half the machine epsilon for the range $[\frac{1}{2}, 1)$.

We need to solve the inequality for $\bar{y}$, but unfortunately, we do not know $\sqrt{x}$, and we have only an approximate value y from the Newton–Raphson iteration that we found by experiment to be correctly rounded only 75% of the time. However, we know x exactly, so we can square and expand the terms of the inequality to get:

$$(x - u\sqrt{x} + \tfrac{1}{4}u^2) \le \bar{y}^2 \le (x + u\sqrt{x} + \tfrac{1}{4}u^2).$$

By the definition of u, we know that the terms $\frac{1}{4}u^2$ are negligible to machine precision, and because $\sqrt{x}$, y, and $\bar{y}$ are close (in IEEE 754 arithmetic, almost always within one ulp), we can simplify the relation to this:

$$x - uy \le \bar{y}^2 \le x + uy.$$

In order to achieve correct rounding, or equivalently, $y = \bar{y}$, if the square of the computed y is below $x - uy$, we need to increase y. If y is above $x + uy$, then y must be decreased. The catch is that those terms both require higher precision: without it, they each evaluate to x. In addition, they are to be compared against an almost identical value y^2: even a slight error in the comparison yields incorrect decisions.

The solution is to rearrange the computation so that the almost-identical terms that are subtracted are computed with an exact double-length product: in other words, a fused multiply-add operation. Before we display the code to do so, however, we need to address another issue. We know that the square root of an argument in $[\frac{1}{4}, 1)$ must lie strictly within the interval $[\frac{1}{2}, 1)$. The left endpoint is, of course, exactly representable, and the largest exactly representable value in that range is $1 - \epsilon/\beta$. Its exact square, $1 - 2\epsilon/\beta + (\epsilon/\beta)^2$, clearly lies within that range as well, although it is not a machine number. The iterative computation of y could have produced values just outside the interval $[\frac{1}{2}, 1 - \epsilon/\beta)$. We therefore need to clamp the computed y to that interval, and at the endpoints, no rounding-error adjustment is needed.

The code to implement the bounds restriction and the adjustment for correct rounding then looks like this, with the `FMA()` macro concealing the precision suffixes on the C99-style fused multiply-add library functions:

```
#define u              (FP_T_EPSILON / (fp_t)B)
#define LEFT_ENDPOINT  (ONE / (fp_t)B)

if (y <= LEFT_ENDPOINT)
    y = LEFT_ENDPOINT;
else if (y >= (ONE - u))
    y = ONE - u;
else                        /* y is in (1/B,1) */
{
    yy_f = FMA(y, y, -f);

    if (FMA(y, u, yy_f) < ZERO)
        y += u;
    else if (FMA(-y, u, yy_f) > ZERO)
        y -= u;
}
```

The tolerance and bounds are exactly representable, and are compile-time constants, except on GNU/LINUX on IA-64, which violates the 1999 ISO C Standard by having a run-time expression for `LDBL_EPSILON`, the `long double` value of `FP_T_EPSILON`. The entire adjustment costs at most five comparisons and three multiply-adds, compared to the five operations of each doubled Newton–Raphson iteration. That is acceptable as long as the multiply-add is done in hardware, but may be comparatively costly if it is entirely in software.

Our algorithm to ensure correct rounding is notably shorter than the Hull/Abrham approach with variable precision that the IBM decNumber library uses.

With GNU/LINUX on IA-64, exhaustive tests of `sqrtf()` with and without the fused multiply-add correction showed less than 1% difference in the test timing with debug compilation (`-g`), and about 5% difference with optimized compilation (`-O3`). With HP-UX on PA-RISC and IBM AIX on POWER, there was no significant timing difference. However, a similar experiment with Sun Microsystems SOLARIS 10 on IA-32 showed a 10.5 times slowdown from the software fused multiply-add, and on SPARC, a 2.0 times slowdown. If `sqrtf()` were the only computation, the penalty from software fused multiply-add would be larger.

We have assumed here that only a *single* adjustment is needed for y: that is experimentally true for IEEE 754 arithmetic, but almost certainly does not hold for older floating-point architectures. For those, we need to use a loop to adjust y until the rounding requirement is satisfied:

```
... as before ...
else                        /* y is in (1/B,1) */
{
#define MAXSTEP 5

    for (k = 1; k <= MAXSTEP; ++k)
    {
        yy_f = FMA(y, y, -f);
        if (FMA(y, u, yy_f) < ZERO)
            y += u;
        else if (FMA(-y, u, yy_f) > ZERO)
            y -= u;
        else                /* result is correctly rounded */
            break;
    }
}
```

In principle, the loop could be written as `for (;;)`, without needing a counter, because the `break` statement exits the loop when no adjustment is made. However, it is possible on especially aberrant historical systems that the adjustments would sometimes oscillate, producing an infinite loop. Compared to the cost of the loop body, testing and incrementing a loop counter is cheap, and provides an advisable level of safety.

Whether the host provides IEEE 754 arithmetic or not can be determined at compile time by various predefined architectural identifiers, or by the existence of the C99 preprocessor symbol `__STDC_IEC_559__`. If it does, and the rounding mode is known to always be the default *round-to-nearest* mode, then `MAXSTEP` *could* be defined as `1`. An optimizing compiler might then eliminate the loop entirely. However, on most IEEE 754 systems, the rounding mode can be changed dynamically, so the safest and most portable approach for library code like the mathcw package is to always use the loop, even though an extra iteration costs three more multiply-add operations that are almost always unnecessary. The extra steps are taken on average 25% of the time, based on the measurements reported earlier, so the total amortized cost for the code in `sqrtx.h` is about one extra multiply-add per call, which is relatively small.

When *correctly working* fused multiply-add support is available, the code in the file `sqrtx.h` produces correctly rounded results. That is practical only on certain current platforms: G5, IA-64, some MIPS processors, PA-RISC, POWER, and PowerPC. On others, fused multiply-add library support is either absent, incorrectly implemented, or supplied in (slow) software, as we described in **Section 4.17** on page 85. Here are sample runs on IA-64 of test programs to demonstrate that our algorithm works perfectly:

```
% gcc -fno-builtin -g rndsq1.c -L.. -lmcw -lm && time ./a.out
Tests of rounding error in sqrtf(x)
Total tests      = 16777216
sqrtf() exact    = 16777216 (100.00%)
sqrtf() low      =        0 (0.00%)
sqrtf() high     =        0 (0.00%)
29.626u 0.000s 0:29.62 100.0%   0+0k 0+0io 77pf+0w

% gcc -fno-builtin -g rndsq3.c -L.. -lmcw -lm && time ./a.out
Tests of rounding error in sqrt(x)
Total tests      = 33554432
sqrt() exact     = 33554432 (100.00%)
sqrt() low       =        0 (0.00%)
sqrt() high      =        0 (0.00%)
34.370u 0.000s 0:34.37 100.0%   0+0k 0+0io 75pf+0w

% gcc -fno-builtin -g rndsq4.c -L.. -lmcw -lm && time ./a.out
Tests of rounding error in sqrtl(x)
Total tests      = 2147483648
sqrtl() exact    = 2147483648 (100.00%)
sqrtl() low      =          0 (0.00%)
sqrtl() high     =          0 (0.00%)
2013.228u 0.016s 33:33.27 99.9% 0+0k 0+0io 75pf+0w

% gcc -fno-builtin -g rndsq5.c -L.. -lmcw -lm && time ./a.out
Tests of rounding error in sqrtl(x)
Total tests      = 2147483648
sqrtl() exact    = 2147483648 (100.00%)
sqrtl() low      =          0 (0.00%)
sqrtl() high     =          0 (0.00%)
2074.115u 0.021s 34:34.19 99.9% 0+0k 0+0io 75pf+0w

% gcc -fno-builtin -g rndsq6.c -L.. -lmcw -lm && time ./a.out
Tests of rounding error in sqrtl(x)
Total tests      = 2147483648
sqrtl() exact    = 2147483648 (100.00%)
sqrtl() low      =          0 (0.00%)
sqrtl() high     =          0 (0.00%)
2113.057u 0.099s 35:25.66 99.4% 0+0k 0+0io 75pf+0w
```

We also need to ask: what does our rounding correction do in the event that FMA(x,y,z) is incorrectly evaluated with separate multiply and add instructions. Here is a numerical experiment with GNU/LINUX on AMD64 with such a faulty library routine:

```
% cc -fno-builtin -g rndsq1.c  -L.. -lmcw -lm && time ./a.out
Tests of rounding error in sqrtf(x)
Total tests      = 16777216
sqrtf() exact    = 14023830 (83.59%)
sqrtf() low      =  1376926 (8.21%)
sqrtf() high     =  1376460 (8.20%)
9.587u 0.002s 0:09.59 99.8%     0+0k 0+0io 0pf+0w
```

That can be compared with the original code, which gave correctly rounded results in 75% of the exhaustive tests. Thus, although a simplistic multiply-add library function or hardware instruction, as on MIPS R10000, is undesirable, at least it does not make matters worse here.

When y is clamped to the interval endpoints, it is important not to do further adjustments. Here is a numerical experiment with hoc32 that demonstrates why, following the steps in our code to compute the square root of the floating-point number just below one:

```
hoc32> u = macheps(1)/2
hoc32> hexfp(u)
        +0x1.0p-24
hoc32> f = 1 - u
hoc32> hexfp(f)
        +0x1.fffffep-1
hoc32> f
        0.99999994
hoc32> y = 1
hoc32> yy_f = fma(y, y, -f)
hoc32> hexfp(yy_f)
        +0x1.0p-24
hoc32> yy_f
        5.96046448e-08
hoc32> fma(y, u, yy_f)
        1.1920929e-07
hoc32> fma(-y, u, yy_f)
        0
```

The second multiply-add is positive, so the upward adjustment of y would not be made. The third multiply-add is zero, so the downward adjustment of y would not be made either. The result is that the returned value is incorrect by one ulp. During development and testing, the report of that *single failure* from among the 16 777 216 tests led to incorporation of the bound checks on y.

8.1.3 Variant iterations for the square root

The iteration based on finding a root of $f(y) = y^2 - x$ that we use in the mathcw library is a good choice, but there are many other possible definitions of $f(y)$. The tests in **Section 8.1.1** on page 216 show that our standard algorithm produces results that are correctly rounded about 75% of the time, so we compare that with three other minimizing functions:

- Choosing $f(y) = 1/y^2 - 1/x$ gives an iteration that we can write in several mathematically, but *not computationally*, equivalent forms:

  $$\begin{aligned} y_{k+1} &= y_k(3x - y_k^2)/(2x) \\ &= y_k(3/2 - y_k^2/(2x)) \\ &= y_k + y_k(1/2 - y_k^2/(2x)), \\ &= y_k + \tfrac{1}{2}y_k(1 - y_k^2/x). \end{aligned}$$

  At convergence, y_{k+1} is the desired square root. Because x is a constant, the factor $1/(2x)$ can be precomputed to replace an expensive divide by a cheaper multiply in each iteration. However, $1/(2x)$ is unlikely to be exactly representable, so its computation introduces an extra rounding error, and the iteration is slightly less accurate. That can be corrected by restoring the division by $2x$ in the last iteration.

  For hexadecimal arithmetic, alter the iteration to $z = y_k(3/4 - y_k^2/(4x))$, with $1/(4x)$ computed outside the loop, and set $y_{k+1} = z + z$.

  Exhaustive tests in IEEE 754 32-bit binary arithmetic show that the first formula for y_{k+1} produces correctly rounded results in 67% of the tests when the factor $1/(2x)$ is used. Using the second formula with a division by $2x$ in the last iteration also gets a score of 67%. The third formula with a final division by $2x$ gets 77% correct.

- Let $x = 1 - y$, expand its square root in a Taylor series, and use the first three terms as an approximating function:

  $$\begin{aligned} \sqrt{x} &= \sqrt{1-y} \\ &= 1 - (1/2)y - (1/8)y^2 - (1/16)y^3 - (5/128)y^4 - \cdots, \\ g(y) &= 1 - (1/2)y - (1/8)y^2, && \textit{truncated Taylor series}, \\ g'(y) &= -(1/2) - (1/4)y, \\ f(y) &= x - (g(y))^2, \\ y_{k+1} &= y_k + (x - (g(y_k))^2)/(2g'(y_k)g(y_k)). \end{aligned}$$

  The final square root is $g(y_k)$, which should be computed in Horner form, producing a small correction to an exact constant. About 76% of the results are properly rounded.

- Proceed as in the last case, but use the first four terms as an approximating function:

  $$\begin{aligned} g(y) &= 1 - (1/2)y - (1/8)y^2 - (1/16)y^3, && \textit{truncated Taylor series}, \\ g'(y) &= -(1/2) - (1/4)y - (3/16)y^2, \\ f(y) &= x - (g(y))^2, \\ y_{k+1} &= y_k + (x - (g(y_k))^2)/(2g'(y_k)g(y_k)). \end{aligned}$$

  The target square root is again $g(y_k)$. However, because $g(y)$ is a closer approximation to the square root, we expect the iterations to converge more quickly. The tradeoff is that $g(y)$ and $g'(y)$ each require one extra add and one extra multiply. Test show that about 76% of the results are properly rounded, but the average iteration count decreases by only 0.2.

Timing tests show that the second and third algorithms are about 1.3 to 1.5 times slower than the first, so they offer no advantage. Computing the square root as the sum of a constant term and a small correction adds only about 2% to the fraction of results that are correctly rounded, so it does not provide an easy route to results that are almost always correctly rounded.

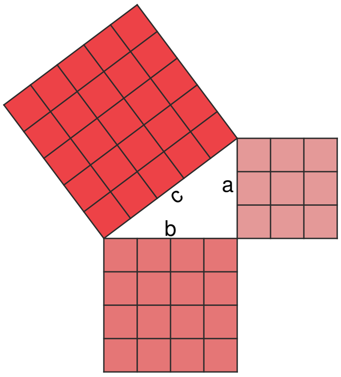
8.2 Hypotenuse and vector norms

The longest edge of a right triangle is called the *hypotenuse*, from a Greek phrase meaning *the side subtending the right angle*. The hypotenuse is often needed in many areas of engineering, science, and technology, where it occurs in problems of measurement, as well as in the polar form of complex numbers (see **Section 15.2** on page 443), and in the construction of rotation matrices.

Elementary-school pupils learn about the Greek mathematician and theologian Pythagoras and the discovery of the famous *Pythagoras' Theorem* that the sum of the squares of the lengths of the shorter edges of a right triangle is equal to the square of the length of the longest edge, usually shortened in mathematics to $a^2 + b^2 = c^2$. The square root of the left-hand side is the length of the longest edge. That value is also the shortest distance between two points on a flat plane, where it is called the *Euclidean distance*, after the Greek mathematician Euclid, who lived about two hundred years after Pythagoras.

To simplify calculation, schoolbook applications usually concentrate on the *Pythagorean Triples*, which are *integer* solutions that satisfy the Theorem. Integer multiples of Triples are also Pythagorean Triples, because $(na)^2 + (nb)^2 = n^2(a^2 + b^2) = n^2c^2 = (nc)^2$. However, the important thing to remember is that there is always a real solution c for which the Theorem holds for *any* real values a and b, and it satisfies the geometrically obvious *triangle inequality*: $|a - b| \leq c \leq a + b$.

Here is a graphical representation of the first Triple, and a list of the 16 scale-unrelated Triples with $c < 100$:



(3, 4, 5) (5, 12, 13) (7, 24, 25) (8, 15, 17)
(9, 40, 41) (11, 60, 61) (12, 35, 37) (13, 84, 85)
(16, 63, 65) (20, 21, 29) (28, 45, 53) (33, 56, 65)
(36, 77, 85) (39, 80, 89) (48, 55, 73) (65, 72, 97)

For software testing purposes, one can easily generate Triples, albeit possibly scaled upward by a common integer factor, like this:

$$\begin{aligned}
m, n &= \text{arbitrary positive integers with } m > n, \\
\boldsymbol{a = m^2 - n^2}, \qquad \boldsymbol{b} &\boldsymbol{= 2mn}, \qquad \boldsymbol{c = m^2 + n^2}, \\
a^2 + b^2 &= (m^4 - 2m^2n^2 + n^4) + 4m^2n^2, \\
&= m^4 + 2m^2n^2 + n^4, \\
&= (m^2 + n^2)^2, \\
&= c^2.
\end{aligned}$$

As an example of mathematical curiosity, Fermat found in 1643 the *smallest* Pythagorean Triple for which both c and $a + b$ are squares of integers (see [Kna92, pages 55–56, 119–122] for the history and derivation of those numbers):

$$\begin{aligned}
a &= 1\,061\,652\,293\,520, \\
b &= 4\,565\,486\,027\,761, \\
a + b &= 5\,627\,138\,321\,281 = 2\,372\,159^2, \\
c &= 4\,687\,298\,610\,289 = 2\,165\,017^2.
\end{aligned}$$

Euclid's result extends to higher dimensions in flat spaces, so the distance between two points in three-dimensional space is just $d = \sqrt{a^2 + b^2 + c^2}$, with each of the values a, b, and c being the corresponding differences in three-dimensional coordinates (x, y, z). For an n-element vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, the *Euclidean norm* is defined to be the square root of the sum of the squares, written $||\boldsymbol{x}|| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$. That norm is often called the *two-norm*, after the superscripts and root order: it can be generalized to arbitrary order p, of which $p = 1, 2$, and $+\infty$ are the most important:

$$\begin{aligned}
||\boldsymbol{x}||_p &= \sqrt[p]{|x_1|^p + |x_2|^p + \cdots + |x_n|^p}, && \textit{general } p\textit{-norm, for } p = 1, 2, 3, \ldots, \\
||\boldsymbol{x}||_1 &= |x_1| + |x_2| + \cdots + |x_n|, && \textit{one-norm}, \\
||\boldsymbol{x}||_2 &= \sqrt{|x_1|^2 + |x_2|^2 + \cdots + |x_n|^2}, && \textit{Euclidean norm, or two-norm}, \\
||\boldsymbol{x}||_\infty &= \lim_{p\to\infty} ||\boldsymbol{x}||_p \\
&= \max(|x_1|, |x_2|, \ldots, |x_n|), && \textit{infinity norm or inf-norm}.
\end{aligned}$$

When the norm subscript is omitted, then often, the Euclidean norm is assumed.

The first satisfactory published algorithm [Blu78] to compute the vector Euclidean norm was complicated, and it spurred further work [LHKK79] in the famous LINPACK and LAPACK Fortran libraries for numerical linear algebra. Those libraries provide a family of routines for computation of the two-norm, and `dnrm2()` is the most popular family member. The Fortran 2008 Standard [FTN10] adds `norm2(x)` to the language, and its Annex C notes that the one-norm and infinity-norm can be computed as `maxval(sum(abs(x)))` and `maxval(abs(x))`, using standard array intrinsics.

Unfortunately, when the Euclidean distance is needed, programmers too often write it in the obvious simple form

```
c = sqrt(a * a + b * b);
```

without realizing that the result is completely wrong when the squares overflow or underflow, or the sum overflows.[1] Indeed, the computation is invalid for roughly *half* of all possible floating-point numbers, so more care is called for.

Few programming languages address the need for an accurate implementation of that important operation, but it has been present on most UNIX systems since Version 7 in 1979. It is curiously absent from Ada, C89, C++, C#, Fortran (prior to the 2008 definition), Java, Pascal, and most scripting languages, but was finally added to the C99 Standard, which describes it like this:

> ***7.12.7.3 The hypot functions***
>
> ***Synopsis***
>
> ```
> #include <math.h>
> double hypot       (double x, double y);
> float hypotf       (float x, float y);
> long double hypotl (long double x, long double y);
> ```
>
> ***Description***
>
> *The `hypot` functions compute the square root of the sum of the squares of `x` and `y`, without undue overflow or underflow. A range error may occur.*
>
> ***Returns***
>
> *The `hypot` functions return $\sqrt{x^2 + y^2}$.*

In an Appendix, the C99 Standard specifies additional behavior:

> ***F.9.4.3*** *The `hypot` functions*
>
> — *`hypot(x, y)`, `hypot(y, x)`, and `hypot(x, -y)` are equivalent.*
>
> — *`hypot(x,` ± 0`)` is equivalent to `fabs(x)`.*
>
> — *`hypot(`$\pm\infty$`, y)` returns $+\infty$, even if y is a NaN.*

An explanation of the last requirement is found in the accompanying *C9X Rationale* document, which says

[1] The `gnuplot` utility used for many of the function error plots in this book has that serious flaw. This author repaired the error in a private patched version of that program.

> ***F.9.4.3*** *The* `hypot` *functions*
>
> *Note that* `hypot(INFINITY,NAN)` *returns* `+INFINITY`*, under the justification that* `hypot(INFINITY,y)` *is* $+\infty$ *for any numeric value* y*.*

A first attempt at reimplementing the function is based on the scaled forms $|x|\sqrt{1+(y/x)^2}$ and $|y|\sqrt{(x/y)^2+1}$, with a block that looks like this:

```
double q, r, result, s, t;

x = fabs(x);
y = fabs(y);

if (x < y)           /* reorder to ensure x >= y >= 0 */
{
    t = x;
    x = y;
    y = t;
}

q = y / x;
r = q * q;

result = x * sqrt(1.0 + r);
```

That resembles the code used in the original 1979 UNIX version of `hypot()` for the DEC PDP-11. The algorithm is frequently good enough, but we can, and should, do better in a library implementation. The problem is that the expression r takes twice as many digits as we have, and when it is small compared to one, we lose even more digits in forming $1+r$.

The Taylor series of the scaled hypotenuse kernel looks like this:

$$\sqrt{1+r} = 1 + r/2 - r^2/8 + r^3/16 - 5r^4/128 + \cdots .$$

We can use it to handle the case of small r, but for larger r, the rounding error in r itself from the division in its definition contaminates the argument to the square root.

We omit the straightforward initial code that handles Infinity, NaN, and zero arguments in the mathcw library implementation of `hypot()`, and show only the code that is most commonly used.

When a fast *fused multiply-add* operation is available, as described later in this book (see **Section 13.26** on page 388 through **Section 13.29** on page 402), we use the following code in `hypot()`. The comment block describes the tricky part of the computation:

```
#if defined(HAVE_FAST_FMA)

else
{
    fp_t t;

    x = QABS(x);
    y = QABS(y);

    if (x < y)              /* reorder to guarantee x >= y > 0 */
    {
        t = x;
        x = y;
        y = t;
    }

    if (x > y)
    {
        /************************************************************
           When we compute
                t = 1 + r*r
           the exact result needs twice as many bits as we have
           available.  When we then compute
                s = sqrt(t)
           we lose accuracy that we can readily recover as follows.
           Let the exact square root be the sum of the computed root,
           s, and a small correction, c:
                 sqrt(t)  = s + c           (in exact arithmetic)
                       t  = (s + c)**2      (in exact arithmetic)
                       t ~= s*s + 2*s*c     (discard tiny c**2 term)
                       c  = (t - s*s)/(2*s) (solve for correction)
           Although the correction is small, it makes a difference:
           without it, hypot(3,4) returns 5 + epsilon, instead of 5.
           With the correction, we not only get the correct value for
           hypot(3,4), we also repair any damage from an inaccurate
           sqrt() implementation.
        ************************************************************/

        fp_t c, s, r;

        r = y / x;
        t = ONE + r * r;
        s = SQRT(t);
        c = FMA(-s, s, t) / (s + s);
        result = FMA(x, s, x * c);
    }
    else    /* x == y */
        result = FMA(x, SQRT_TWO_HI, x * SQRT_TWO_LO);

    if (result > FP_T_MAX)
        result = SET_ERANGE(result);
}
```

Otherwise, we proceed somewhat differently with this code block that uses Taylor-series expansions for small ratios, and for larger ratios, a square root with an error correction:

```
#else /* FMA() is too slow */

#define QFMA(x,y,z)   ((x) * (y) + (z))

else
{
    fp_t h, q, r, t;
    static fp_t RCUT_4 = FP(0.0);
    static fp_t RCUT_8 = FP(0.0);
    static int do_init = 1;

    if (do_init)
    {
        RCUT_4 = SQRT(SQRT(      (FP(32768.0) / FP(1792.0)) * HALF * FP_T_EPSILON / (fp_t)B ));
        RCUT_8 = SQRT(SQRT(SQRT( (FP(32768.0) /  FP(715.0)) * HALF * FP_T_EPSILON / (fp_t)B )));
        do_init = 0;
    }

    x = QABS(x);
```

```
y = QABS(y);

if (x < y)                  /* reorder to guarantee x >= y > 0 */
{
    t = x;
    x = y;
    y = t;
}

q = y / x;
r = q * q;

if (r < RCUT_4)             /* 4th-order Taylor series region */
{
    fp_t sum;

    sum = FP(1792.0);
    sum = QFMA(sum, r, -FP( 2560.0));
    sum = QFMA(sum, r,  FP( 4096.0));
    sum = QFMA(sum, r, -FP( 8192.0));
    sum = QFMA(sum, r,  FP(32768.0));
    t = (r * (FP(1.0) / FP(65536.0))) * sum;
    h = x;
    h += x * t;
    result = h;
}
else if (r < RCUT_8)        /* 8th-order Taylor series region */
{
    fp_t sum;

    sum = FP(715.0);
    sum = QFMA(sum, r, -FP(  858.0));
    sum = QFMA(sum, r,  FP( 1056.0));
    sum = QFMA(sum, r, -FP( 1344.0));
    sum = QFMA(sum, r,  FP( 1792.0));
    sum = QFMA(sum, r, -FP( 2560.0));
    sum = QFMA(sum, r,  FP( 4096.0));
    sum = QFMA(sum, r, -FP( 8192.0));
    sum = QFMA(sum, r,  FP(32768.0));
    t = (r * (FP(1.0) / FP(65536.0))) * sum;
    h = x;
    h += x * t;
    result = h;
}
else
{
    h = x * SQRT(ONE + r);

    /*
    ** Now compute a correction such that
    **
    **  (h + d)**2 = x**2 + y**2
    **  (h**2 + 2*h*d) ~= x**2 + y**2           (drop d**2 term)
    **                d ~= (x**2 + y**2 - h**2)/(2*h)
    **                   = ((x/h)*x + (y/h)*y - h)/2
    */
```

```
        if (h <= FP_T_MAX)          /* apply only for finite h */
        {
            fp_t d, u, v;

            d = -h;
            u = x / h;
            d += u * x;
            v = y / h;
            d += v * y;
            d *= HALF;
            result = h + d;
        }
        else
            result = h;

        if (result > FP_T_MAX)
            result = SET_ERANGE(result);
    }
}

#endif /* defined(HAVE_FAST_FMA) */
```

The QFMA() macro expands inline here to separate multiply and add operations, but we use it to emphasize the structure of the Taylor-series computation. The series coefficients are scaled to integer values to eliminate rounding errors in their values. The outer rescaling of the series by $1/65\,536$ is exact for bases 2, 4, and 16. In tests with Pythagorean Triples, the final correction pushes the frequency of incorrect rounding below 10^{-7}.

8.3 Hypotenuse by iteration

The hypotenuse can be computed without requiring a square root with an iterative algorithm called pythag() [MM83] that is used in the EISPACK library for solving certain kinds of matrix-algebra problems. The original code was written before IEEE 754 arithmetic was widely available, and it goes into an infinite loop if either argument is Infinity or a NaN. Here is a hoc translation that handles those two special values as required by C99:

```
func pythag(a,b)                              \
{   ## Return sqrt(a**2 + b**2) without destructive underflow
    ## or overflow, and without an explicit square root

    if      (isinf(a)) p = fabs(a)            \
    else if (isinf(b)) p = fabs(b)            \
    else if (isnan(a)) p = a                  \
    else if (isnan(b)) p = b                  \
    else                                      \
    {
        a = fabs(a)
        b = fabs(b)
        p = fmax(a, b)

        if (p > 0)                            \
        {
            q = fmin(a, b)

            while (1)                         \
            {
                r = (q / p)**2

                if ((4 + r) == 4) break
                s = r / (4 + r)
                p += p * (s + s)
                q *= s
            }
        }
    }

    return (p)
}
```

Elimination of Infinity and NaN guarantees loop termination, which usually requires only three or four iterations in 64-bit arithmetic, and up to five in the 128-bit formats.

Moler[2] & Morrison present the `pythag()` function with little hint of its origin. In an article that follows theirs in the same journal issue, Dubrulle [Dub83] gives two derivations that are helpful in understanding the algorithm and its speedy convergence. Jamieson [Jam89] later generalized their results to arbitrarily high order.

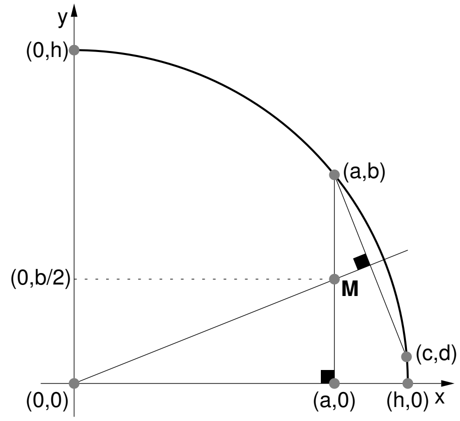
Dubrulle considers a sequence of points (x_k, y_k) on a quarter circle whose radius is the hypotenuse h that we seek. The initial point is $(x_0, y_0) = (x, y)$, where $h = \sqrt{x^2 + y^2}$. Each following point is closer to the final point $(x_n, y_n) = (h, 0)$, so that we have two monotone sequences:

$$\begin{aligned} x = x_0 < x_1 < x_2 < \cdots < x_{n-1} < x_n = h, \\ y = y_0 > y_1 > y_2 > \cdots > y_{n-1} > y_n = 0, \\ x_k^2 + y_k^2 = h^2, \qquad \text{for } k = 0, 1, 2, \ldots n. \end{aligned}$$

Each point on the arc satisfies $x_k \geq y_k > 0$, except for the last, where we permit $y_n = 0$. The algorithmic goal is to find relations between consecutive members of each sequence.

The first derivation of the `pythag()` algorithm is graphical, and to remove subscript clutter, we rename point (x_k, y_k) to (a, b), and its successor point (x_{k+1}, y_{k+1}) to (c, d). We then relate some algebraic formulas to a diagram, like this:

$$\begin{aligned} a^2 + b^2 &= h^2, \\ c^2 + d^2 &= h^2, \\ \boldsymbol{M} &= (a, b/2), \\ \vec{\boldsymbol{M}} \bullet \left(\overrightarrow{(c,d)} - \overrightarrow{(a,b)} \right) &= 0, \\ 2a(c-a) + b(d-b) &= 0. \end{aligned}$$



The two sums of squares are equal by definition. The point $\boldsymbol{M}$ lies exactly halfway between the x axis and the first point (a, b). The vector from the origin through $\boldsymbol{M}$, denoted $\vec{\boldsymbol{M}}$, intersects the middle of the line between (a, b) and (c, d), which is just their vector difference. Two vectors are perpendicular when their dot product is zero, and that fact is stated in the expansion of the dot product in the final equation.

Solutions of the equations relating the four values a, b, c, and d to find values of c and d are tedious to derive by hand, but Maple finds them easily:

[2] Cleve Moler is a famous American numerical analyst who has co-authored at least ten books on computation. Early in his career, he chaired the Computer Science Department at the University of New Mexico. He made important contributions to the EISPACK and LINPACK software projects for matrix algebra, and developed an interactive program prototype called `matlab` that provides a powerful and compact programming language for easy access to matrix operations by those libraries. He later co-founded *The MathWorks*, a large and successful company that further develops, markets, and supports MATLAB, Simulink®, and other software for general numerical computation, modeling, and simulation. A bibliography of his works is available in the *BibNet Project* archives at `http://www.math.utah.edu/pub/bibnet/authors/m`. Another bibliography covers MATLAB: `http://www.math.utah.edu/pub/tex/bib/index-table-m.html#matlab`.

```
% maple
> solve( { a*a + b*b = c*c + d*d, 2*a*(c - a) + b*(d - b) = 0 }, [c, d] );

                               2      2           3
                         a (3 b  + 4 a )         b
   [[c = a, d = b], [c = ---------------, d = ---------]]
                             2      2          2      2
                            b  + 4 a          b  + 4 a
```

The first solution is the uninteresting trivial one where the points (a, b) and (c, d) coincide. We simplify the second solution by introduction of new variables r and s to represent common subexpressions:

$$\begin{aligned}
r &= b^2/a^2, && \textit{convenience variable,}\\
s &= r/(4+r), && \textit{another convenience variable,}\\
c &= a(3b^2+4a^2)/(b^2+4a^2), && \textit{now divide bottom and top by } a^2\textit{,}\\
&= a(3r+4)/(r+4), && \textit{regroup,}\\
&= a\big((r+4)+2r\big)/(r+4), && \textit{divide by denominator,}\\
&= a\big(1+2r/(r+4)\big), && \textit{and substitute with } s\textit{,}\\
\boldsymbol{c} &\boldsymbol{= a+2as}, && \textbf{\textit{solution for first unknown,}}\\
d &= b^3/(b^2+4a^2), && \textit{divide bottom and top by } a^2\textit{,}\\
&= b\big((b^2/a^2)/(b^2/a^2+4)\big), && \textit{then substitute with } r\textit{,}\\
&= b\big(r/(r+4)\big), && \textit{and finally substitute with } s\textit{,}\\
\boldsymbol{d} &\boldsymbol{= sb}, && \textbf{\textit{solution for second unknown.}}
\end{aligned}$$

Dubrulle's highlighted solutions for c and d correspond exactly to Moler & Morrison's variables p and q in the `pythag()` function.

To see how quickly the algorithm converges, we relate the new estimate of the error in the hypotenuse to the old error:

$$\begin{aligned}
h - c &= h - a(3b^2+4a^2)/(b^2+4a^2), && \textit{then substitute } b^2 = h^2 - a^2\textit{,}\\
&= h - a(3h^2+a^2)/(h^2+3a^2), && \textit{then make common denominator,}\\
&= (h^3+3a^2h-3ah^2-a^3)/(h^2+3a^2), && \textit{then factor numerator polynomial,}\\
&= (h-a)^3/(h^2+3a^2).
\end{aligned}$$

The new error is proportional to the cube of the old error, so convergence in *cubic*. Once a is close to h, the number of correct digits in c *triples* with each iteration.

Here is a demonstration of how fast the `pythag()` algorithm converges, using the smallest Triple and the Fermat Triple with a modified function that prints the relative error and the current estimate of the hypotenuse:

```
hocd128> h = pythagx(3,4)
3.60e-01  4
5.47e-03  4.986_301_369_863_013_698_630_136_986_301_37
1.03e-08  4.999_999_974_188_252_149_492_661_061_886_531
6.88e-26  4.999_999_999_999_999_999_999_999_828_030_177
0.00e+00  5

hocd128> h = pythagx(1_061_652_293_520, 4_565_486_027_761)
5.13e-02  4_565_486_027_761
9.13e-06  4_687_277_220_292.340_969_469_158_891_647_832
4.75e-17  4_687_298_610_288.999_888_639_829_215_045_638
0.00e+00  4_687_298_610_289
```

Dubrulle's second derivation of the `pythag()` algorithm solves a function-minimization problem. We define

$$\begin{aligned} r &= b^2/a^2, && \textit{our original convenience variable,}\\ F(h) &= a^2 + b^2 - h^2, && \textit{a and b fixed constants, and h to be found,}\\ F'(h) &= -2h, && \textit{first derivative with respect to h,}\\ F''(h) &= -2, && \textit{second derivative with respect to h.} \end{aligned}$$

Given the function and its first two derivatives, we want to find the root h that satisfies $F(h) = 0$. The third-order Halley's method described in **Section 2.4** on page 9 says that, given a value h_k, we can find an improved estimate of the root like this:

$$\begin{aligned} h_{k+1} &= h_k - 2F(h_k)F'(h_k)/\big(2(F'(h_k))^2 - F(h_k)F''(h_k)\big), && \textit{Halley's formula,}\\ &= h_k - 2F(h_k)(-2h_k)/\big(2(2h_k)^2 - F(h_k)(-2)\big) && \textit{for our function $F(h)$.} \end{aligned}$$

Next, eliminate messy subscripts by substituting $h_k = a$ and $h_{k+1} = c$:

$$\begin{aligned} c &= a - 2F(a)(-2a)/\big(2(2a)^2 - F(a)(-2)\big),\\ &= a + 4aF(a)/\big(8a^2 + 2F(a)\big),\\ &= a + 2aF(a)/\big(4a^2 + F(a)\big). \end{aligned}$$

Evaluate the function, substitute into the last result for c, and continue simplification:

$$\begin{aligned} F(a) &= a^2 + b^2 - a^2,\\ &= b^2,\\ c &= a + 2ab^2/(4a^2 + b^2), && \textit{then divide bottom and top by a^2,}\\ &= a + 2a(b^2/a^2)/\big(4 + (b^2/a^2)\big), && \textit{then substitute with r,}\\ &= a + 2ar/(4 + r), && \textit{then substitute with s,}\\ &= a + 2as. \end{aligned}$$

The last result is identical to that in `pythag()` and the first solution in the graphical derivation. The solution for d is derived exactly as before: $d = sb$.

Moler, Morrison, and Dubrulle summarize the key features of the `pythag()` algorithm for finite nonzero arguments, and we augment their observations:

- The input variables can be replaced internally by their absolute values and then reordered so that $a \geq b > 0$.
- No magic constants or machine-specific parameters are needed.
- The code is short: each iteration requires only three adds, three multiplies, two divides, one comparison, and four or five intermediate variables. That is an advantage on older systems with small memories, or where the operands are not simple real scalars, but instead are complex, or software multiple-precision values, or matrices.
- On many modern systems, the entire loop can run entirely in the CPU with all variables in registers, and no memory accesses. Code-generation tests with optimizing compilers on several architectures show that the loop code may be reduced to as few as 14 instructions.
- The results of the two divides are needed immediately, so there is no possibility of overlapping their execution with other instructions until they complete. The software divide sequences in `pythag()` on IA-64 waste a third of the instruction-packet slots with `nop` (no-operation) instructions.
- Convergence is soon cubic.
- The convergence test is simple. The value r decreases on each iteration by at least $s^4 \leq (1/5)^4 \approx 0.0016$, and s falls by at least $r/4$ on each update, accelerating convergence, so that we soon have $\mathrm{fl}(4 + r) = 4$ to machine precision.

- The higher intermediate precision on some platforms is harmless, but its use in the convergence test may sometimes cause one extra iteration. That problem can be repaired by declaring the variable `r` to be `volatile` in C89 and C99, or by passing its address to an external routine. The tradeoff is then the cost of two or three storage references against one iteration requiring in-CPU arithmetic. The references are almost certain to be satisfied from cache memory, rather than main memory, and are therefore likely to be less costly than an additional iteration.

- The largest intermediate value produced never exceeds $h = \sqrt{a^2 + b^2} \leq \sqrt{2}a$. Overflow is avoided unless $a >$ (maximum normal number)$/\sqrt{2}$, in which case the overflow is inevitable because the exact result is too big to represent.

- If b is much smaller than a, intermediate computations can underflow harmlessly to subnormals or zero.

- Unfortunately, in `pythag()`, underflow is not always harmless if it is *abrupt*. In the first iteration, $r \leq 1$ and $s \leq 1/5$. Let μ be the smallest normal number. Then if $b \leq 5\mu$, and a is not much bigger, then at the end of the first iteration, we have $q < \mu$. If that value is abruptly set to zero, instead of becoming subnormal, the iterations terminate early, and the final result is $\approx a$. In particular, if $a = 4\mu$ and $b = 3\mu$, the function returns 4.98μ instead of the correct 5μ. The problem is most easily solved by exact upward scaling of the arguments, and we show how to do so shortly.

- A numerically robust computation of the vector Euclidean norm can be done with the help of `hypot()` or `pythag()` and simple code like this:

  ```
  e = ZERO;

  for (k = 0; k < n; ++k)
      e = HYPOT(e, v[k]);
  ```

  That is less efficient than alternatives like `dnrm2()`, but much easier to code correctly. Moler & Morrison observe that when a vector norm is needed, it is almost always outside other numerically intensive nested loops whose run times are $\mathcal{O}(n^2)$ or higher, so the inefficiency is unlikely to matter.

Despite its wide use in many other software packages, the original code for `pythag()` is not as accurate as our implementation of `hypot()`, and has problems with arguments that are tiny, Infinity, or NaN. Gradual underflow removes the problems with tiny arguments. The code *is* robust with respect to arguments near the floating-point overflow limits. Here are some numerical experiments with the 128-bit decimal version of hoc that compare the original `pythag()` with our `hypot()` function:

```
% hocd128
hocd128> load("opythag")
hocd128> __INDENT__ = "\t "

hocd128> MINSUBNORMAL * sqrt(2)
         1e-6176
hocd128> hypot(MINSUBNORMAL, MINSUBNORMAL)
         1e-6176
hocd128> pythag(MINSUBNORMAL, MINSUBNORMAL)
         1e-6176

hocd128> MINNORMAL * sqrt(2)
         1.414_213_562_373_095_048_801_688_724_209_698e-6143
hocd128> hypot(MINNORMAL, MINNORMAL)
         1.414_213_562_373_095_048_801_688_724_209_698e-6143
hocd128> pythag(MINNORMAL, MINNORMAL)
         1.414_213_562_373_095_048_801_688_724_209_698e-6143

hocd128> MAXNORMAL
         9.999_999_999_999_999_999_999_999_999_999_999e+6144
```

```
hocd128> hypot(MAXNORMAL/sqrt(2), MAXNORMAL/sqrt(2))
         9.999_999_999_999_999_999_999_999_999_999_999e+6144
hocd128> pythag(MAXNORMAL/sqrt(2), MAXNORMAL/sqrt(2))
         9.999_999_999_999_999_999_999_999_999_999_999e+6144

hocd128> hypot(3,4)
         5
hocd128> pythag(3,4)
         5

hocd128> hypot(8, 15)
         17
hocd128> pythag(8, 15)
         16.999_999_999_999_999_999_999_999_999_999_99
```

The Fortran version of `pythag()` in EISPACK differs slightly from the original published pseudocode in that p is updated from $(1 + 2s) \times p$ instead of with the addition $p + 2ps$, and q is eliminated in favor of updating r from its previous value. The first of those optimizations is unfortunate, because it reduces accuracy somewhat, and the results for most Pythagorean Triples exhibit rounding errors.

Because of the irregularities near the underflow limit when abrupt underflow to zero is in effect, internal scaling is required if the software is to be robust and portable. All that is needed is to move small arguments far enough away from the underflow limit so that none of the loop expressions underflows prematurely before convergence is reached. A suitable scale factor is the reciprocal of the machine epsilon. Before the main loop, we insert this code block:

```
static const fp_t EPS_INV = FP(1.0) / FP_T_EPSILON;
static const fp_t X_MIN  = FP_T_MIN / FP_T_EPSILON;
static const fp_t X_MAX  = FP_T_MAX * FP_T_EPSILON;
...

if (a < X_MIN)
{
    a *= EPS_INV;               /* EXACT scaling */
    b *= EPS_INV;               /* EXACT scaling */
    scale = FP_T_EPSILON;
}
else if (a > X_MAX)
{
    a *= FP_T_EPSILON;          /* EXACT scaling */
    b *= FP_T_EPSILON;          /* EXACT scaling */
    scale = EPS_INV;
}
else
    scale = ONE;
```

The block also scales large arguments away from the overflow limit to avoid problems with a tiny number of arguments where intermediate rounding could temporarily push values into the overflow region.

After the main loop, we undo the scaling with a single statement:

```
p *= scale;                     /* EXACT scaling */
```

The authors of the two cited papers about `pythag()` coded the loop iterations to run until a suitable convergence test is satisfied. However, they failed to ask whether the computed result is *correctly rounded*. Tests of `pythag()` with all representable Pythagorean Triples using the available precisions of both binary and decimal arithmetic on several architectures show that up to 22% of the results fail to be correctly rounded. The function values can be a few machine epsilons off because of rounding-error accumulation in the loop iterations. Unlike the Newton–Raphson iteration, the `pythag()` algorithm is *not* self correcting, and rounding errors are not just from the last update of the root.

To illustrate the rounding problem, here is what the single-precision decimal version of `pythag()` reports:

```
% hocd32
hocd32> load("pythag")
hocd32> pythag(3,4)
        5.000_006
hocd32> 5 + macheps(1)
        5.000_001
```

Even a school child can do better. The defect seems not to have been reported in the research literature, and it is likely to surprise a naive user.

What is needed is a final corrective code block to bring the computed result closer to the correct answer. That block is identical to that in the code for `hypot()` that begins in **Section 8.2** on page 225. Its cost is similar to doing one more iteration of the main loop.

We complete this section by noting that Dubrulle made some important extensions of `pythag()` by working out higher-order iterations for roots of more complicated functions of the form

$$F_k(x) = \big((h+x)^k - (h - x^k)\big)/(2h).$$

Higher powers are needed to prevent the higher derivatives becoming zero. Dubrulle's details are littered with subscripts and superscripts, but the good news is that the final formulas are reasonably simple, and tabulated for orders two through nine. The third-order iteration is just our normal `pythag()`, and the formulas can be divided into two classes — odd and even order — that require slightly different programs.

The file `pythagn.c` in the `exp` subdirectory of the `mathcw` distribution implements all of them, and tests on several current architectures show that the fifth-order function is almost always the fastest, but only by a few percent.

Higher-order approximations converge faster, so fewer iterations (only one or two) are needed. Dubrulle found that all of the formulas have the feature of `pythag()` that there are only *two* divisions in the loop. Thus, they are of significant value for multiple-precision arithmetic and matrix arithmetic, where division is a particularly expensive operation.

To give a flavor of the code required, here is the inner loop of the fifth-order algorithm that *quintuples* the number of correct digits on each iteration:

```
for (;;)
{
    fp_t p, s;

    q = y_k / x_k;
    r = q * q;

    if ((ONE + r) == ONE)
        break;

    p = FP(8.0) + FP(4.0) * r;                /* order dependent */
    s = r / (FP(16.0) + (FP(12.0) + r) * r);    /* order dependent */
    x_k += x_k * p * s;
    y_k *= r * s;                             /* order dependent */
}

return (scale * x_k);
```

Compared to `pythag()`, each iteration costs one more add and three more multiplies. The *only* code changes needed for other odd orders are replacement of the polynomials that define p and s, and adjustment of the power of r that appears in the final loop statement where y_k is updated.

8.4 Reciprocal square root

Among UNIX vendors, only Hewlett–Packard HP-UX on IA-64 and IBM AIX on PowerPC provide the reciprocal square-root function, `rsqrt()`. Neither C89 nor C99 defines it.

You might wonder why it is not sufficient to just compute the function as `1.0/sqrt(x)` or `sqrt(1.0/x)`. There are two reasons why a separate function is desirable:

- The division introduces an extra rounding error that can be avoided if $1/\sqrt{x}$ is computed independently of $\sqrt{x}$.
- We noted earlier in **Section 4.7.1** on page 69 that IEEE 754 requires that $\sqrt{-0} = -0$. That means that $1/\sqrt{-0} = -\infty$, whereas $\sqrt{1/(-0)} = \sqrt{-\infty} \to \mathrm{NaN}$. Thus, the mathematical relation $1/\sqrt{x} = \sqrt{1/x}$ does not hold in IEEE 754 arithmetic when x is a negative zero.

The `rsqrt()` function is defined to compute $1/\sqrt{x}$, so `rsqrt(-0)` returns $-\infty$.

Like the square-root function, the reciprocal square root is best computed by Newton–Raphson iteration, finding a solution $y = \mathrm{rsqrt}(x)$ to $f(y) = y^2 - 1/x = 0$ for a given fixed x. The iteration is $y_{k+1} = y_k - f(y_k)/f'(y_k) = \frac{1}{2}(y_k + 1/(xy_k))$.

Alternatively, use $f(y) = (1/y)^2 - x$, which gives the iteration $y_{k+1} = \frac{1}{2}y_k(3 - y_k^2 x)$. For hexadecimal arithmetic, compute it as $z = y_k(3/4 - y_k^2 x/4)$ followed by $y_{k+1} = z + z$, and replace the constants $3/4$ and $1/4$ by 0.75 and 0.25.

For x in $[\frac{1}{4}, 1)$, a $\langle 1/1 \rangle$ minimax polynomial produces a starting estimate of y_0 good to eight bits, and minimizes the total work for the number of iterations needed for the precisions of interest. The rational polynomial is scaled to get a unit high-order coefficient in the denominator, eliminating one multiply. The effects of wobbling precision in hexadecimal normalization do not matter for the computation of the starting guess.

As with the computation of the square root (see **Section 8.1** on page 215), the double-step iteration saves one multiply every two steps. Each double step takes the form $z = y_k + 1/(xy_k)$ followed by $y_{k+2} = \frac{1}{4}z + 1/(xz)$, and just two double steps produce about 143 correct bits.

When the intermediate computations are done in higher precision, such as in the 80-bit format used on IA-32 and IA-64 systems, the computed reciprocal square root is almost always correct to the last bit. Even without extra precision, the average error in the final result should normally not exceed 0.5 bits for rounding arithmetic, and 1 bit for truncating arithmetic. The test results reported later in **Chapter 24** on page 811 show that the average error is 0.00 bits on systems with IEEE 754 arithmetic.

8.4.1 Improved rounding of the reciprocal square root

The adjustment described in **Section 8.1.2** on page 217 to obtain `sqrt()` values that are always correctly rounded is readily adapted for the reciprocal square root. The correctly rounded value $\bar{y}$ satisfies this inequality:

$$1/\sqrt{x} - \tfrac{1}{2}u \le \bar{y} \le 1/\sqrt{x} + \tfrac{1}{2}u.$$

We then square the inequality, expand, and drop terms that are $\mathcal{O}(u^2)$, producing this relation:

$$1/x - u/\sqrt{x} \le \bar{y}^2 \le 1/x + u/\sqrt{x}.$$

The difference between the values y, $\bar{y}$, and $1/\sqrt{x}$ is $\mathcal{O}(u)$, and the terms $\pm u/\sqrt{x}$ are also $\mathcal{O}(u)$, so we can replace $1/\sqrt{x}$ by y to get this condition:

$$1/x - uy \le \bar{y}^2 \le 1/x + uy.$$

The bounds on the possible answers are determined as follows. The reduced argument, f, lies in the interval $[1/\beta^2, 1)$, so the result $\bar{y}$ lies in $(1, \beta]$ or $[1, \beta]$ (we soon show which of them is correct), and thus, $u = \epsilon$, the machine epsilon. The largest possible value of f is the machine number closest to the right endpoint, $1 - \epsilon/\beta$. The Taylor-series expansion for the reciprocal square root near that endpoint is

$$1/\sqrt{1-\delta} = 1 + (1/2)\delta + (3/8)\delta^2 + (5/16)\delta^3 + \cdots,$$

so $\mathrm{rsqrt}(1 - \epsilon/\beta) \approx 1 + \epsilon/(2\beta)$. To machine precision, that value rounds to exactly 1, so the computed y must be restricted to the interval $[1, \beta]$. Combining the bounds restrictions with the inequality leads to this adjustment code:

```
#define u FP_T_EPSILON

if (y <= ONE)
```

```
    y = ONE;
else if (y >= (fp_t)B)
    y = (fp_t)B;
else                        /* y is in (1,B) */
{
    yy_finv = FMA(y, y, -ONE/f);

    if (FMA( y, u, yy_finv) < ZERO)
        y += u;
    else if (FMA(-y, u, yy_finv) > ZERO)
        y -= u;
}
```

Without the adjustments, a test of `rsqrtf()` with the program `rndrs1.c` produces results like this on IA-64:

```
% gcc -fno-builtin -g rndrs1.c -L.. -lmcw -lm && time ./a.out
Tests of rounding error in rsqrtf(x)
Total tests       = 16777216
rsqrtf() exact    = 12437813 (74.14%)
rsqrtf() low      =  2169098 (12.93%)
rsqrtf() high     =  2170305 (12.94%)
26.813u 0.003s 0:43.43 61.7%    0+0k 0+0io 76pf+0w
```

With the adjustment and a correctly working fused multiply-add operation, the results on IA-64 are:

```
% gcc -fno-builtin -g rndrs1.c -L.. -lmcw -lm && time ./a.out
Tests of rounding error in rsqrtf(x)
Total tests       = 16777216
rsqrtf() exact    = 14594678 (86.99%)
rsqrtf() low      =  1091079 (6.50%)
rsqrtf() high     =  1091459 (6.51%)
28.829u 0.003s 0:46.73 61.6%    0+0k 0+0io 76pf+0w
```

That is an almost 13% improvement in the fraction of correctly rounded results, but is still not good enough.

The division $1/f$ is subject to rounding error, slightly contaminating the computed `yy_finv`, which in turn sometimes results in incorrect decisions in the conditionals. Scaling the inequality by f removes one source of rounding error, but introduces another. A test with that variant produced nearly the same results: 86.94% exact.

When the multiply-add is done without an exact product, as on most GNU/Linux systems, then the results look like this (on AMD64):

```
% gcc -fno-builtin -g rndrs1.c -L.. -lmcw -lm && time ./a.out
Tests of rounding error in rsqrtf(x)
Total tests       = 16777216
rsqrtf() exact    = 13245833 (78.95%)
rsqrtf() low      =  1765996 (10.53%)
rsqrtf() high     =  1765387 (10.52%)
9.271u 0.001s 0:09.28 99.8%     0+0k 0+0io 0pf+0w
```

That is about 8% worse than with a correctly functioning fused multiply-add operation.

More work clearly needs to be done to obtain results that are always correctly rounded. In the next section, we show how to approach that goal.

8.4.2 Almost-correct rounding of the reciprocal square root

The treatment of the adjustment of the computed reciprocal square root in the last section was deficient because we did not eliminate the error inherent in forming $1/x$. To remedy that, multiply the inequality in the previous section by x to obtain

$$1 - uxy \leq xy^2 \leq 1 + uxy.$$

Then subtract xy^2 from each term to obtain a form that is convenient for multiply-add operations:

$$-(uxy + (xy^2 - 1)) \leq 0 \leq uxy - (xy^2 - 1).$$

Because u is a power of the base, the product ux is exact, and thus ux and y supply the first two arguments of the fused multiply-add function which computes uxy exactly. However, exact computation of the quantity $xy^2 - 1$ requires three times as many bits as we have available, and the subtraction suffers massive cancellation, producing a result of size $\mathcal{O}(u)$. To solve that problem, we split y into two parts, $y = y_{\mathrm{hi}} + y_{\mathrm{lo}}$, each with half as many bits as the significand can represent. The term y_{hi}^2 is then exactly representable, and a term like $xy_{\mathrm{hi}}^2 - 1$ can then be computed accurately with a fused multiply-add operation. Because we know that y lies in $[1, \beta]$, with a t-digit significand, the split is easily done by adding and subtracting the constant $\beta^{\lceil t/2 \rceil}$, like this:

$$\begin{aligned} y_{\mathrm{hi}} &= \mathrm{fl}(\mathrm{fl}(y + \beta^{\lceil t/2 \rceil}) - \beta^{\lceil t/2 \rceil}), \\ y_{\mathrm{lo}} &= y - y_{\mathrm{lo}}. \end{aligned}$$

We then expand $xy^2 - 1$ in terms of y_{hi} and y_{lo}, factoring the terms to exploit fused multiply-add opportunities:

$$\begin{aligned} xy^2 - 1 &= x(y_{\mathrm{hi}} + y_{\mathrm{lo}})^2 - 1 \\ &= x(y_{\mathrm{hi}}^2 + 2y_{\mathrm{hi}}y_{\mathrm{lo}} + y_{\mathrm{lo}}^2) - 1 \\ &= (xy_{\mathrm{hi}}^2 - 1) + xy_{\mathrm{lo}}(2y_{\mathrm{hi}} + y_{\mathrm{lo}}) \\ &= \mathrm{fma}(x, y_{\mathrm{hi}}^2, -1) + x\,\mathrm{fma}(2y_{\mathrm{hi}}, y_{\mathrm{lo}}, y_{\mathrm{lo}}^2) \\ &= \mathrm{fma}(x, \mathrm{fma}(2y_{\mathrm{hi}}, y_{\mathrm{lo}}, y_{\mathrm{lo}}^2), \mathrm{fma}(x, y_{\mathrm{hi}}^2, -1)). \end{aligned}$$

The C code for the bounding and adjustment of y is clearer if we use two intermediate variables for the inner fused multiply-add operations:

```
#define u FP_T_EPSILON

if (y <= ONE)
    y = ONE;
else if (y >= (fp_)B)
    y = (fp_)B;
else                    /* y is in (1,B) */
{
    y_hi = purify + y;  /* purify is B**ceil(T/2) */
    y_hi -= purify;     /* y_hi now has only floor(T/2) digits */
    y_lo = y - y_hi;    /* y_lo has remaining ceil(T/2) digits */

    s = FMA(f, y_hi * y_hi, -ONE);
    t = FMA(y_hi + y_hi, y_lo, y_lo * y_lo);
    fyy_1 = FMA(f, t, s);              /* f*y*y - 1, accurately */

    if (fyy_1 > ZERO)                  /* y may be too high */
    {
        alt_fyy_1 = fyy_1 + u * u * f;
        alt_fyy_1 -= (u + u) * y * f;   /* f*(y - u)**2 - 1 */

        if (fyy_1 > QABS(alt_fyy_1))
            y -= u;
    }
    else if (fyy_1 < ZERO)             /* y may be too low */
    {
        alt_fyy_1 = fyy_1 + u * u * f;
        alt_fyy_1 += (u + u) * y * f;   /* f*(y + u)**2 - 1 */

        if ((-fyy_1) > QABS(alt_fyy_1))
```

```
            y += u;
    }
}
```

When y is possibly too high or too low, we compute a revised test value quickly, without further FMA() calls. If that produced an improvement, we then adjust the final y by one ulp.

Here are exhaustive test results of our code on IA-64 with a correct fused multiply-add operation, first for the 32-bit rsqrtf() function, and then for the 64-bit rsqrt() function:

```
% gcc -fno-builtin -g rndrs1.c ../libmcw.a && time ./a.out
Tests of rounding error in rsqrtf(x)
Total tests       = 16777216
rsqrtf() exact    = 16776743 (100.00%)
rsqrtf() low      =      234 (0.00%)
rsqrtf() high     =      239 (0.00%)
15.109u 0.002s 0:15.11 99.9%    0+0k 0+0io 0pf+0w

% gcc -fno-builtin -g rndrs2.c ../libmcw.a && time ./a.out
Tests of rounding error in rsqrt(x)
Total tests       = 16777216
rsqrt() exact     = 16773141 (99.98%)
rsqrt() low       =     2065 (0.01%)
rsqrt() high      =     2010 (0.01%)
16.111u 0.001s 0:16.11 100.0%   0+0k 0+0io 0pf+0w
```

Those results suggest that our improved rsqrtf() produces incorrect rounding only once in about 35 000 random arguments, and rsqrt() about once in 4000 arguments. However, the estimates are pessimistic, because tests of our reciprocal-square-root functions for both binary and decimal arithmetic using comparison values computed in higher precision in Maple show much lower rates of incorrect rounding.

8.5 Cube root

Berkeley UNIX 4.3BSD introduced the cube-root function, cbrt(), in 1987, and almost all UNIX vendors now include it, although some omit the float and long double companions. It is standardized in C99.

The cube-root function is faster and more accurate than pow(x,1.0/3.0), and like the square root and reciprocal square root, it too is computed by Newton–Raphson iteration. The algorithm finds a solution $y = \sqrt[3]{x}$ to $f(y) = y^3 - x$ for a given x with the iteration $y_{k+1} = y_k - f(y_k)/f'(y_k) = (2y_k + x/y_k^2)/3$. As with the other two functions, doubling the iteration steps and simplifying saves one operation, a divide. The double-step iteration looks like this:

```
f3 = f + f + f;             /* pre-iteration constant */
z = y + y + f / (y * y);
y = (z + z) / NINE + f3 / (z * z);
```

If the last statement is rewritten with a common denominator to save one division, the worst-case error reported by the test suite increases by about 0.7 bits.

In the mathcw implementation, a seven-bit initial approximation with a $\langle 1/1 \rangle$-degree minimax rational polynomial leads to an almost-correct IEEE 754 32-bit result in a double-step iteration at a cost of four adds, three divides, and two multiplies. Consecutive single and double steps provide full accuracy for the 64-bit IEEE 754 format, and two double steps suffice for the 80-bit and 128-bit formats. When the intermediate computations are done in higher precision, such as in the 80-bit format used on IA-32 and IA-64 systems, the computed cube root is almost always correct to the last bit. The test results reported later in Chapter 24 on page 811 show that the average error on systems with IEEE 754 arithmetic is 0.33 bits in the 32-bit format, and 0.00 bits in higher precisions.

8.5.1 Improved rounding of the cube root

As with the square root and its reciprocal, it is possible to improve the result of the Newton–Raphson iteration by application of a correction computed in higher precision with a fused multiply-add operation. The correctly rounded value $\bar{y}$ satisfies this inequality:

$$\sqrt[3]{x} - \tfrac{1}{2}u \le \bar{y} \le \sqrt[3]{x} + \tfrac{1}{2}u.$$

If we cube that inequality and expand, dropping terms that are $\mathcal{O}(u^2)$ or higher, we find:

$$x - \tfrac{3}{2}u(\sqrt[3]{x})^2 \le \bar{y}^3 \le x + \tfrac{3}{2}u(\sqrt[3]{x})^2.$$

Because the difference between the values y, $\bar{y}$, and $\sqrt[3]{x}$ is $\mathcal{O}(u)$, and the terms with the factor $(\sqrt[3]{x})^2$ are also $\mathcal{O}(u)$, we can safely replace $\sqrt[3]{x}$ by y to get this inequality:

$$x - \tfrac{3}{2}uy^2 \le \bar{y}^3 \le x + \tfrac{3}{2}uy^2.$$

Finally, to get a form that is convenient for the fused multiply-add operation, replace $\bar{y}$ by y, and divide the inequality by y to get:

$$x/y - \tfrac{3}{2}uy \le y^2 \le x/y + \tfrac{3}{2}uy.$$

Before we show how that relation can be used to adjust y, we need to examine the endpoint conditions. The reduced argument, f, lies in the interval $[1/\beta^3, 1)$, so its cube root must lie in the interval $[1/\beta, 1)$. The left endpoint is exactly representable, but the machine number closest to the right endpoint is $1 - \epsilon/\beta$. The Taylor-series expansion for the cube root near that endpoint is

$$\sqrt[3]{1-\delta} = 1 - (1/3)\delta - (1/9)\delta^2 - (5/81)\delta^3 - \cdots,$$

so $\operatorname{cbrt}(1 - \epsilon/\beta) \approx 1 - \epsilon/(3\beta)$. To machine precision, with default rounding, that value rounds to exactly 1, so the computed y must be clamped to the interval $[1/\beta, 1]$. We can readily turn that restriction and the inequality into adjustment code:

```
#define B_INVERSE (ONE / (fp_t)B)
#define u         (FP_T_EPSILON / (fp_t)B)
#define u15       (FP(1.5) * u)

if (y <= B_INVERSE)
    y = B_INVERSE;
else if (y >= ONE)
    y = ONE;
else                        /* y is in (1/B,1) */
{
    yy_xy = FMA(y, y, -x/y);

    if (FMA(y, u15, yy_xy) < ZERO)
        y += u;
    else if (FMA(-y, u15, yy_xy) > ZERO)
        y -= u;
}
```

The variables `u` and `u15` are compile-time constants, except on GNU/Linux on IA-64, which violates the 1999 ISO C Standard by having a run-time expression for `LDBL_EPSILON`, the `long double` value of `FP_T_EPSILON`.

Unfortunately, the computation of x/y itself introduces a rounding error that can change the outcome of the conditional tests. Without the adjustment code, the test program `rndcb1.c` produces results like this (on IA-64):

```
% gcc -fno-builtin -g rndcb1.c -L.. -lmcw -lm && time ./a.out
Tests of rounding error in cbrtf(x)
Total tests      = 25165824
cbrtf() exact    = 18694652 (74.29%)
cbrtf() low      =  3125325 (12.42%)
cbrtf() high     =  3345847 (13.30%)
45.188u 0.004s 1:13.32 61.6%    0+0k 0+0io 75pf+0w
```

With the adjustment code, the results look like this when a correct fused multiply-add operation is available:

```
% gcc -fno-builtin -g rndcb1.c -L.. -lmcw -lm && time ./a.out
Tests of rounding error in cbrtf(x)
Total tests     = 25165824
cbrtf() exact   = 23019920 (91.47%)
cbrtf() low     =  1072895 (4.26%)
cbrtf() high    =  1073009 (4.26%)
37.934u 0.002s 1:01.65 61.5%    0+0k 0+0io 76pf+0w
```

Almost identical results are obtained if the division by y is not done. Although the adjustment produces a 17% improvement in the fraction of exact results, it is still not good enough.

If instead the multiply-add is merely done in higher precision, then the results look like this for GNU/LINUX on AMD64:

```
% gcc -fno-builtin -g rndcb1.c -L.. -lmcw -lm && time ./a.out
Tests of rounding error in cbrtf(x)
Total tests     = 25165824
cbrtf() exact   = 22519809 (89.49%)
cbrtf() low     =  1341769 (5.33%)
cbrtf() high    =  1304246 (5.18%)
15.674u 0.000s 0:15.68 99.9%    0+0k 0+0io 0pf+0w
```

That is only 2% worse than with a correct fused multiply-add operation.

In the next section, we show how to make dramatic improvements, but still without a guarantee that results are always correctly rounded.

8.5.2 Almost-correct rounding of the cube root

The difficulty with the adjustments of the last section is that exact computation of the inequality requires three times as many bits as we have, because it needs an accurate value of $\bar{y}^3 - x$. To achieve that, we need to resort to a multiple-precision arithmetic package, or else split the computation into pieces, each of which can be computed exactly, or almost exactly.

If we split y into two parts, a top part with a third of the bits, and a bottom part with the remaining two-thirds of the bits, then the term $y_{\rm hi}^3 - x$ is exact because $y_{\rm hi}^3$ is exactly representable, and the terms have the same exponent. To accomplish the split, we can add and subtract the constant $\beta^{t-1-\lfloor t/3 \rfloor}$, where t is the number of bits in the significand, like this:

$$\begin{aligned} y_{\rm hi} &= \mathrm{fl}(y + \beta^{t-1-\lfloor t/3 \rfloor}) - \beta^{t-1-\lfloor t/3 \rfloor}, \\ y_{\rm lo} &= y - y_{\rm hi}. \end{aligned}$$

Next, we expand $y^3 - x$ in terms of $y_{\rm hi}$ and $y_{\rm lo}$, factoring the terms to expose multiply-add opportunities:

$$\begin{aligned} y^3 - x &= (y_{\rm hi} + y_{\rm lo})^3 - x \\ &= y_{\rm hi}^3 + 3y_{\rm hi}^2 y_{\rm lo} + 3y_{\rm hi} y_{\rm lo}^2 + y_{\rm lo}^3 - x \\ &= (y_{\rm hi}^3 - x) + y_{\rm lo}(3y_{\rm hi}^2 + 3y_{\rm hi} y_{\rm lo} + y_{\rm lo}^2) \\ &= (y_{\rm hi}^3 - x) + y_{\rm lo}\,\mathrm{fma}(3y_{\rm hi}, y_{\rm lo}, 3y_{\rm hi}^2 + y_{\rm lo}^2). \end{aligned}$$

The corresponding C code is easier to follow if we introduce two temporary variables and handle each fused multiply-add operation separately:

```
y_hi = purify + y;          /* purify == B**(T - 1 - floor(T/3)) */
STORE(&y_hi);
y_hi -= purify;             /* y_hi now has only floor(T/3) digits */
STORE(&y_hi);
y_lo = y - y_hi;            /* y_lo has remaining ceil(2T/3) digits */
STORE(&y_lo);
s = y_hi * y_hi * y_hi - f; /* exact */
```

```
t = FMA(THREE * y_hi, y_lo, THREE * y_hi * y_hi + y_lo * y_lo);
yyy_f = FMA(y_lo, t, s);    /* y**3 - f, accurately */

if (yyy_f > ZERO)           /* y may be too big */
{
    alt_yyy_f = -u * u * u;
    alt_yyy_f += -THREE * y * u * u;
    alt_yyy_f += -THREE * y * y * u;
    alt_yyy_f += yyy_f;     /* (y - u)**3 - f, accurately */

    if (yyy_f > QABS(alt_yyy_f))
        y -= u;
}
else if (yyy_f < ZERO)      /* y may be too small */
{
    alt_yyy_f = u * u * u;
    alt_yyy_f += THREE * y * u * u;
    alt_yyy_f += THREE * y * y * u;
    alt_yyy_f += yyy_f;     /* (y + u)**3 - f, accurately */

    if ((-yyy_f) > QABS(alt_yyy_f))
        y += u;
}
```

The cost of the correction is 2 multiply-adds, 3 comparisons, 8 or 9 adds, and 14 multiplies.

As we observed for the square-root function in **Section 8.1.2** on page 218, on historical floating-point architectures with deficient rounding practices, the correction block should be wrapped in a loop that takes at most a few iterations, with `else break` statements added to the two innermost `if` statements to allow early loop exit as soon as no further improvement can be found.

Tests of the adjusted algorithm for `cbrtf()` and `cbrt()` look like this on an IA-64 system:

```
% cc -fno-builtin -g rndcb1.c ../libmcw.a && time ./a.out
Tests of rounding error in cbrtf(x)
Total tests      = 25165824
cbrtf() exact    = 25152589 (99.95%)
cbrtf() low      =     6554 (0.03%)
cbrtf() high     =     6681 (0.03%)
28.352u 0.024s 0:28.37 100.0%   0+0k 0+0io 0pf+0w

% cc -fno-builtin -g rndcb2.c ../libmcw.a && time ./a.out
Tests of rounding error in cbrt(x)
Total tests     = 25165824
cbrt() exact    = 25159744 (99.98%)
cbrt() low      =     3032 (0.01%)
cbrt() high     =     3048 (0.01%)
30.166u 0.001s 0:30.17 99.9%    0+0k 0+0io 0pf+0w
```

Those results suggest that incorrect rounding happens about once in 1900 random arguments in `cbrtf()`, and once in 4000 arguments for `cbrt()`. However, tests of those functions, and `cbrtl()`, with high-precision comparison values in Maple show that incorrect rounding is much less frequent.

8.6 Roots in hardware

Several CPU architectures, including 68000, AMD64, IA-32, and PA-RISC, and later models of Alpha, MIPS, PowerPC, and SPARC, provide single hardware instructions to compute a correctly rounded square root in the IEEE 754 32-bit and 64-bit formats, and for some, also the 80-bit format.

IBM added hardware square-root instructions in the System/360 family for the 32-bit, 64-bit, and 128-bit hexadecimal formats in 1991. The later z-Series processors also support IEEE 754 binary arithmetic, with hardware square root for the three companion binary formats. There are, however, no equivalent instructions for z-Series decimal floating-point arithmetic.

No current commercially significant desktop architecture provides single hardware instructions for the cube root, or higher roots.

To permit graceful evolution of architectural extensions, a few operating systems provide run-time emulation of unimplemented instructions, so that code compiled on a newer processor can still work on earlier models at the same O/S level. However, most operating systems do not offer that convenience, so it is usually impractical to take advantage of new instructions at a user site until all of the local hardware has been upgraded to new models.

Some C compilers make it possible to generate inline assembly-language instructions, and our `sqrtx.h` algorithm file takes advantage of that feature, using code modeled on that shown later in **Section 13.26** on page 388. However, to permit independent testing and use of the software alternative, the hardware instructions are only exposed to the compiler when the preprocessor symbol `USE_ASM` is defined. In the best case, with compiler optimization, the entire body of the `SQRT()` function reduces to just two instructions: the square root, and a return.

The IA-64 architecture lacks single hardware instructions for floating-point divide and square root, but Markstein [Mar00, Chapter 9] shows how they can be implemented with fused multiply-add (FMA) operations using 82-bit registers with 64-bit significands to produce a final *correctly rounded* 53-bit significand in the IEEE 754 64-bit format.

The IA-64 executes bundles of up to three instructions simultaneously, although there are restrictions about how many of each of several instruction classes can appear in a single bundle. For example, bundles may contain at most one floating-point operation. In addition, bundles may contain, or end with, a *stop*, indicated by a double semicolon in assembly code, that forces a delay until the results of previous instructions are available. We display the computation of the square root as a transliteration of code shown in a vendor manual [IA6406, §6.3.3]. The actual code is a direct mapping of each of these assignments to single instructions, where each line contains independent instructions that can execute in parallel:

$$\begin{array}{rlll}
c &= 0.5 & g = \mathrm{frsqrta}(x) & ;; \\
h &= +c * g + 0 & g = +x * g + 0 & ;; \\
r &= -g * h + c & & ;; \\
h &= +r * h + h & g = +r * g + g & ;; \\
r &= -g * h + c & & ;; \\
h &= +r * h + h & g = +r * g + g & ;; \\
r &= -g * h + c & & ;; \\
h &= +r * h + h & g = +r * g + g & ;; \\
r &= -g * g + x & & ;; \\
\mathrm{result} &= r * h + g & &
\end{array}$$

The code cleverly interleaves an iterative algorithm for the square root with an iterative improvement of the estimate of the reciprocal square root. The code is correct *only* for nonzero finite x, so the special arguments of ± 0, quiet and signaling NaN, and $\pm\infty$ must be handled separately

We assume that x is available in a register at the start of the bundle sequence, and each of the four other variables is held in a register. Only five of the 128 floating-point registers are needed, and there are no memory accesses at all. The reciprocal square-root approximation instruction, $\mathrm{frsqrta}(x)$, computes $1/\sqrt{x}$ to slightly more than 8 correct bits. There are 10 instruction groups, 14 bundles, 13 FMA operations, and 27 wasted slots. The final FMA operation is a variant that simultaneously rounds the 82-bit computation to the 64-bit format. The square root in the 32-bit format can be handled with one less iteration by dropping the third and fourth stop groups, and the 80-bit format with one more iteration.

An IEEE 754 64-bit divide on the IA-64 has a similar sequence that fits in 8 stop groups and 10 bundles with an 8-bit reciprocal approximation, 9 FMA operations, and 20 wasted slots.

The IA-64 designers' intent is that compilers could become smart enough to map other nearby computations into the empty slots of *inline* divide and square-root sequences, a technique that is known as *software pipelining*.

Markstein's proofs of correct rounding for division and square root require both the higher intermediate precision, and the wider exponent range, of the 82-bit format on IA-64, so the instruction sequences are *not* directly usable on systems that provide only 64-bit FMA operations.

A subset of the architectures with hardware square root also provide instructions for approximating and iterating the computation of the reciprocal square root. However, we do not take advantage of that hardware support because of its uncertain rounding behavior. Markstein [Mar00, §9.4] reports that exhaustive testing of his compact algorithm for the reciprocal square root shows that it is always correctly rounded for the 32-bit format, but finding proofs of its rounding characteristics for the 64-bit and 80-bit formats is an open and unsolved problem.

8.7 Summary

The square root and inverse square root are common in computation, whereas the cube root, and higher roots, are rare. In this chapter, we have shown how effective algorithms for close approximations to roots lead to relatively short computer programs. However, achieving the goal of always-correct rounding of roots is difficult without access to arithmetic of higher precision. The fused multiply-add operation is a critical component for correctly rounded square root, and helpful, though not sufficient, for other roots.

The IA-64 algorithms for square root and divide are good examples of the value of the FMA instruction in practical computation. That operation is likely to be more widely available in future hardware platforms, and is required by IEEE 754-2008, so it is worthwhile to design new algorithms to be able to exploit it. The FMA was added to some AMD64 processors in late 2008, and to an implementation of the SPARC architecture in late 2009, although neither is available to this author. Sadly, the FMA is absent from the most widely deployed desktop CPU architecture, the IA-32 and AMD64, except in still-rare CPU models.

9 Argument reduction

REDUCTION, *n.*: (G) THE TRANSFORMATION OF AN ALGEBRAIC EXPRESSION INTO ANOTHER OF A SIMPLER KIND.

— *New Century Dictionary* (1914).

Elementary functions whose argument range is large can often be related to similar functions with diminished arguments, where those functions are more easily, and more accurately, computed. The ultimate accuracy of elementary functions depends critically on how well that argument reduction is carried out, and historically, that has been the weak point in many vendor libraries. Although the argument-reduction problem is usually easy to state, it is not easy to solve, because it often requires access to arithmetic of higher precision than is available.

The argument-reduction problem is most challenging for those trigonometric functions that are defined for arguments over the entire real axis, yet are periodic, repeating their behavior in a small interval about the origin in other intervals translated along the axis. The reduction problem also exists for logarithmic, exponential, and hyperbolic functions, but is usually less severe, either because the reduction can be carried out exactly by decomposition of the floating-point data format, or because the functions soon overflow or underflow, drastically limiting the argument range over which they can be represented in floating-point arithmetic.

In this chapter, we examine the reduction problem primarily for the worst case of the trigonometric functions. We describe how the problem can be simply solved for a sharply limited argument range, and then show how the range can be extended to cover most practical needs with only a modest programming effort. Algorithmic discoveries in the 1980s and 1990s provide a definitive solution of the argument-reduction problem, but at the cost of significant programming complexity. However, once such a portable implementation is available, the major difficulty in achieving high accuracy in the computation of some of the elementary functions is swept away. This book provides possibly the first detailed exposition and programming of exact reduction.

9.1 Simple argument reduction

The trigonometric functions are defined over the entire real axis, but their values are completely determined by their behavior over an interval $[0, \pi/2]$. We therefore need to be able to reduce an arbitrary argument x to the *nearest* integer multiple of π, plus a remainder, as $x = n\pi + r$, where r lies in $[-\pi/2, +\pi/2]$. Other reductions are possible as well, as shown in **Table 9.1** on the following page.

Because r can be negative, care must be taken to avoid loss of leading digits in table entries that require argument shifting. For example, accuracy loss in the argument of the entry $\sin(\pi/4 + r)$ can be handled as follows. Replace r by $f\pi/4$ to get $\sin(\pi/4 + r) = \sin((1 + f)\pi/4)$. When $r \approx -\pi/4$, we have $f \approx -1$, so computation of $1 + f$ to working precision requires both f and r accurate to about *twice* working precision. That problem is avoided if the reduction is done with respect to π or $\pi/2$.

Although the reduction looks trivial, it is not, for exactly the same reasons that the remainder functions are hard to compute (see **Section 6.15** on page 146): when x is large, the subtraction $r = x - n\pi$ suffers massive loss of leading digits. The value $n\pi$ needs at least t (the significant precision) more digits than there are in the integer part of the largest floating-point number. Thus, for the IEEE 754 80-bit and 128-bit binary formats, it appears that we need both n and π accurate to about 5000 decimal digits, where n is obtained from $\operatorname{round}(x/\pi)$. We investigate the minimum precision required more carefully in the next section on page 251.

The traditional solution to the problem is to pretend that it does not exist, either by arguing that it is the user's responsibility to do a sensible argument reduction, preferably by exact analytical means in the computational formulas that need the trigonometric functions, or by suggesting that higher precision should be used. For example, when $x > \beta^{t-1}$, x has no fractional digits at all in the available precision, so adjacent machine numbers sample the trigonometric function at most six times per period of 2π. We can see that effect with a high-precision calculation in Maple for IEEE 754 32-bit values in both binary and decimal formats by choosing x big enough to make $\epsilon = 1$:

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_9

Table 9.1: Trigonometric argument reduction as $x = n(\pi/k) + r$ for various choices of k. Larger k values increase the number of regions in which separate approximations, such as fits to rational polynomials, must be applied, but allow shorter polynomials and faster evaluation, and as described in the text, require r to higher precision.

	$x = n\pi + r$	
n mod 2	**0**	**1**
$\cos(x)$	$\cos(r)$	$-\cos(r)$
$\sin(x)$	$\sin(r)$	$-\sin(r)$

	$x = n\pi/2 + r$			
n mod 4	**0**	**1**	**2**	**3**
$\cos(x)$	$\cos(r)$	$-\sin(r)$	$-\cos(r)$	$\sin(r)$
$\sin(x)$	$\sin(r)$	$\cos(r)$	$-\sin(r)$	$-\cos(r)$

	$x = n\pi/4 + r$			
n mod 8	**0**	**1**	**2**	**3**
$\cos(x)$	$\cos(r)$	$\cos(\pi/4 + r)$	$-\sin(r)$	$-\sin(\pi/4 + r)$
$\sin(x)$	$\sin(r)$	$\sin(\pi/4 + r)$	$\cos(r)$	$\cos(\pi/4 + r)$
n mod 8	**4**	**5**	**6**	**7**
$\cos(x)$	$-\cos(r)$	$-\cos(\pi/4 + r)$	$\sin(r)$	$\sin(\pi/4 + r)$
$\sin(x)$	$-\sin(r)$	$-\sin(\pi/4 + r)$	$-\cos(r)$	$-\cos(\pi/4 + r)$

	$x = n\pi/8 + r$			
n mod 16	**0**	**1**	**2**	**3**
$\cos(x)$	$\cos(r)$	$\cos(\pi/8 + r)$	$\cos(\pi/4 + r)$	$\cos(3\pi/8 + r)$
$\sin(x)$	$\sin(r)$	$\sin(\pi/8 + r)$	$\sin(\pi/4 + r)$	$\sin(3\pi/8 + r)$
n mod 16	**4**	**5**	**6**	**7**
$\cos(x)$	$-\sin(r)$	$-\sin(\pi/8 + r)$	$-\sin(\pi/4 + r)$	$-\sin(3\pi/8 + r)$
$\sin(x)$	$\cos(r)$	$\cos(\pi/8 + r)$	$\cos(\pi/4 + r)$	$\cos(3\pi/8 + r)$
n mod 16	**8**	**9**	**10**	**11**
$\cos(x)$	$-\cos(r)$	$-\cos(\pi/8 + r)$	$-\cos(\pi/4 + r)$	$-\cos(3\pi/8 + r)$
$\sin(x)$	$-\sin(r)$	$-\sin(\pi/8 + r)$	$-\sin(\pi/4 + r)$	$-\sin(3\pi/8 + r)$
n mod 16	**12**	**13**	**14**	**15**
$\cos(x)$	$\sin(r)$	$\sin(\pi/8 + r)$	$\sin(\pi/4 + r)$	$\sin(3\pi/8 + r)$
$\sin(x)$	$-\cos(r)$	$-\cos(\pi/8 + r)$	$-\cos(\pi/4 + r)$	$-\cos(3\pi/8 + r)$

```
% maple
...
> Digits := 100:
> x := 2**23:
> for k from 0 to 5 do
>     printf("sin(x + %d * epsilon) = % .7f\n", k, sin(x + k))
> end do:
sin(x + 0 * epsilon) =  0.4322482
sin(x + 1 * epsilon) = -0.5252557
sin(x + 2 * epsilon) = -0.9998419
sin(x + 3 * epsilon) = -0.5551781
sin(x + 4 * epsilon) =  0.3999139
sin(x + 5 * epsilon) =  0.9873269

> x := 10**6:
> for k from 0 to 5 do
>     printf("sin(x + %d * epsilon) = % .7f\n", k, sin(x + k))
> end do:
sin(x + 0 * epsilon) = -0.3499935
sin(x + 1 * epsilon) =  0.5991474
sin(x + 2 * epsilon) =  0.9974350
sin(x + 3 * epsilon) =  0.4786854
sin(x + 4 * epsilon) = -0.4801653
sin(x + 5 * epsilon) = -0.9975543
```

Cody and Waite take the view that a modest range of $|x|$ somewhat greater than their reduced interval of $[0, \pi/2]$ should be supported. For example, if we allow $|x|$ up to about 100π, then by working in a precision extended by about three more decimal digits, we can often compute the remainder exactly.

If a sufficiently higher precision is available, then code like this

```
hp_t x_hp;
...
x_hp = (hp_t)x;
r = (fp_t)(x_hp - HP_ROUND(x_hp * HP_ONE_OVER_PI) * HP_PI);
```

does the job, with obvious definitions of the rounding function and the constants.

Otherwise, Cody and Waite first determine n by

```
fp_t n;
...
n = ROUND(x * ONE_OVER_PI);
```

where the constant is the closest machine number to $1/\pi$. They next introduce two constants c_1 and c_2 such that $c_1 + c_2$ approximates π to a few digits more than working precision. For binary arithmetic, they propose two pairs:

$$
\begin{aligned}
c_1 &= 201/64, && \textit{use when } t \le 32, \\
&= 3.140\,625, && \textit{exact decimal value}, \\
&= \texttt{0o3.11p0}, && \textit{exact octal value}, \\
&= \texttt{0x3.24p0}, && \textit{exact hexadecimal value}, \\
c_2 &= \texttt{9.676\,535\,897\,932e-04}, &&
\end{aligned}
$$

$$
\begin{aligned}
c_1 &= 3217/1024, && \textit{use when } t > 32, \\
&= 3.141\,601\,562\,5, && \textit{exact decimal value}, \\
&= \texttt{0o3.1104p0}, && \textit{exact octal value}, \\
&= \texttt{0x3.244p0}, && \textit{exact hexadecimal value}, \\
c_2 &= \texttt{-8.908\,910\,206\,761\,537\,356\,616\,720\,5e-06}. &&
\end{aligned}
$$

The first pair provides about eight additional bits of precision, and the second pair, about twelve extra bits.

If the floating-point design has a guard digit for subtraction, as modern systems do, then the reduction can be done like this:

```
volatile fp_t r;
/* ... code omitted ... */
r = ABS(x) - n * C1;
STORE(&r);
r -= n * C2;
STORE(&r);
```

When there is no guard digit, the reduction requires an additional step to split x into integer and fractional parts, to make it more likely that, if a trailing digit is lost during any alignment needed for the subtraction, then it is a zero digit:

```
volatile fp_t r;
fp_t x1, x2;
```

```
/* ... code omitted ... */
x1 = TRUNC(ABS(x));
x2 = ABS(x) - x1;
r = x1 - n * C1;
STORE(&r);
r += x2;
STORE(&r);
r -= n * C2;
STORE(&r);
```

In both cases, we have written the computation so as to avoid assuming that parentheses are obeyed, and to expose opportunities for fast fused multiply-add operations.

The `volatile` keyword and `STORE()` calls are essential to preserve the required evaluation order. However, the comma operator in C guarantees that its operands are evaluated in order, starting with the leftmost one, so we could use comma expressions instead like this:

```
fp_t r;
/* ... code omitted ... */
(r = ABS(x) - n * C1, r -= n * C2);      /* if guard digit */
(r = x1 - n * C1, r += x2, r -= n * C2); /* if no guard digit */
```

Many programmers find the comma operator uncomfortable in situations like those, so we henceforth stick with the more verbose code that makes clear that the evaluation order is being strictly controlled. The comma and comma-free versions are *not* computationally equivalent on some systems, because on architectures with long internal registers, the comma expression permits r to be computed in higher precision. That extra precision is helpful, and does no harm here.

Cody and Waite choose a limit `YMAX` with the value $\mathrm{trunc}(\pi\beta^{t/2})$ to ensure that these conditions are satisfied:

- n is exactly representable.
- Both nc_1 and $(n - \frac{1}{2})c_1$ are exactly representable.
- A rounding error in the last digit of $y = |x|$ should produce a relative error in $\sin(y)$ no larger than $\beta^{-t/2}$ for $y <$ `YMAX`.

If $|x| >$ `YMAX`, then at least one of those conditions fails to hold, and remedial action is called for. Cody and Waite suggest printing an error message, and then either returning 0.0, or accepting the reduced argument r as exact, and continuing normally. Some implementations of the trigonometric functions instead return a NaN when they can no longer do accurate argument reduction.

Although Cody and Waite do not do so, one can argue that if a return value of 0.0 is chosen for either $\cos(x)$ or $\sin(x)$, then the return value for the other should be 1.0, so that the mathematical identity for the sum of the squares of the cosine and sine is preserved.

Unfortunately, the Cody/Waite reduction is inaccurate when x is near a multiple of π. In the worst case, x is the floating-point value just below or just above the nearest machine number to that multiple. The subtraction that forms r loses all but the last digit of x, and their two-step reduction recovers only as many extra digits of $n\pi$ as the sum $c_1 + c_2$ represents: eight to twelve bits.

We can improve the reduction by representing π to at least *twice* working precision as a sum of terms, for all but the last of which the products nc_k are exact. A Maple procedure makes it easy to generate the needed terms as a C array declaration with exactly representable rational numbers as initial values:

```
asplit := proc(x, n, base, kmax, Cname)
    local b, c, d, j, k, sum:
    b := base**(n):
    sum := 0:
    printf("\n"):
    printf("static const fp_t C[] =\n{   /* sum(k) C[k] = %s */\n", Cname):
    printf("#define D_   FP(%a.0)\n", b):
```

```
    for k from 1 do

        if (evalb(k = kmax)) then
            printf("    /* %d */ FP(%.*g)\n", k - 1, min(35,Digits), x - sum):
            sum := x:
            break
        end if:

        d := evalf(b**k):
        c := evalf(round((x - sum) * d)):
        printf("    /* %d */ FP(%a) / (D_", k - 1, c):

        for j from 2 to k do printf(" * D_") end do:

        printf("),\n"):
        sum := sum + c / d:

        if (evalb(evalf(abs((x - sum)/x)) < 10**(-Digits))) then
            break
        end if

    end do:

    printf("#undef D_\n"):
    printf("};\n\n"):
end proc:
```

We can use that procedure to find expansions of π for the IEEE 754 32-bit binary and decimal formats like this:

```
> Digits := 40:

> asplit(Pi, 10, 2, 5, "pi"):

static const fp_t C[] =
{   /* sum(k) C[k] = pi */
#define D_   FP(1024.0)
    /* 0 */ FP(3217.) / (D_),
    /* 1 */ FP(-9.) / (D_ * D_),
    /* 2 */ FP(-350.) / (D_ * D_ * D_),
    /* 3 */ FP(134.) / (D_ * D_ * D_ * D_),
    /* 4 */ FP(-3.30279907448531851112350178e-13)
#undef D_
};

> asplit(Pi, 4, 10, 5, "pi"):

static const fp_t C[] =
{   /* sum(k) C[k] = pi */
#define D_   FP(10000.0)
    /* 0 */ FP(31416.) / (D_),
    /* 1 */ FP(-735.) / (D_ * D_),
    /* 2 */ FP(3590.) / (D_ * D_ * D_),
    /* 3 */ FP(-2068.) / (D_ * D_ * D_ * D_),
    /* 4 */ FP(3.8462643383279502884197e-17)
#undef D_
};
```

The mathcw library provides a general function to reduce a normal argument as $x = n \times C + r$, where n is returned via a pointer argument, r is the function value, C is represented by the array `c[n_c]`, and $1/C$ is supplied as `c_inverse`:

```
fp_t
REDUCE(fp_t x, fp_t c_inverse, int n_c, const fp_t c[/*n_c*/], int *pn, fp_t *perr)
{
    fp_t r, xn;
    volatile fp_t err;

    err = ZERO;
    xn = ROUND(x * c_inverse);

    if (xn == ZERO)             /* handle common case quickly */
        r = x;
    else
    {
        int k;
        fp_t c_hi, c_lo;
        volatile fp_t r_hi, r_lo, sum;

#if B == 2
        fp_t c_half;
#else
        volatile fp_t c_half;
#endif

#if defined(HAVE_GUARD_DIGIT)
        sum = x - xn * c[0];    /* EXACT */
        STORE(&sum);
#else
        {   /* compute sum = (x1 + x2) - n*c[0] accurately */
            volatile fp_t x1, x2;

            x1 = TRUNC(x);
            STORE(&x1);
            x2 = x - x1;        /* EXACT */
            STORE(&x2);
            sum = x1 - xn * c[0]; /* EXACT */
            STORE(&sum);
            sum += x2;
            STORE(&sum);
        }
#endif  /* defined(HAVE_GUARD_DIGIT) */

        c_hi = c[0];
        c_lo = ZERO;
        err = ZERO;

        for (k = 1; k < n_c; ++k)
        {
            fp_t t;
            volatile fp_t new_sum;

            t = -xn * c[k];     /* EXACT */
            new_sum = sum + t;
            STORE(&new_sum);
            err += sum - new_sum;
            STORE(&err);
            err += t;
            STORE(&err);
            sum = new_sum;
            c_lo += c[k];
        }

        r_hi = sum;
        r_lo = err;

        /*
        ** We now have computed the decomposition
        **
        **      x = n * C + r,
        **      r = r_hi + r_lo
        **
        ** where
        **
        **      C = sum(k=0:(n_c-1)) c[k]
        **
        ** However, if x is close to a half-integral multiple of C,
        ** perhaps we should instead have computed
        **
        **      x = (n + 1) * C + s
        ** or
        **      x = (n - 1) * C + t
        **
        ** We can tell whether one or the other of those adjustments
        ** is needed by checking whether r lies outside the interval
        ** [-C/2, +C/2].
        */

#if B == 2
        c_half = HALF * (c_hi + c_lo);
#else
        c_half = HALF * c_hi;
        STORE(&c_half);
        c_half += HALF * c_lo;
#endif

        r = r_hi + r_lo;
        err = r_hi - r;
        STORE(&err);
        err += r_lo;            /* (r + err) == (r_hi + r_lo) */

        if (r < -c_half)
        {
            xn--;
            r += c_hi;          /* EXACT */
            r += c_lo;

            if (r > c_half)
                r = c_half;
        }
        else if (c_half < r)
        {
            xn++;
            r -= c_hi;          /* EXACT */
            r -= c_lo;
            if (r < -c_half)
                r = -c_half;
        }
    }

    if (perr != (fp_t *)NULL)
        *perr = err;

    if (pn != (int *)NULL)
    {
        if ((fp_t)INT_MAX < B_TO_T)     /* INT_MAX + 1 is EXACT */
            *pn = (int)FMOD(xn, (fp_t)INT_MAX + ONE);
        else if ( (TWO_TO_23 < B_TO_T) && ((sizeof(int)/sizeof(char)) * CHAR_BIT > 23) )
            *pn = (int)FMOD(xn, TWO_TO_23);
        else
            *pn = (int)FMOD(xn, FP(32768.0));
    }

    return (r);
}
```

Because the reduction code is intended primarily for controlled internal use in the mathcw library, it makes no checks for Infinity and NaN arguments. The common case where the argument is already in the reduced range is detected by the test for a zero value of xn, and treated quickly without further unnecessary computation.

The code handles the first entry in the constant array, c[0], differently, depending on whether there is a guard digit or not. The loop accumulates the contributions of the remaining entries, c[k], to the function value, using the pair sum and err to represent the result to roughly twice working precision.

Tests of the original and extended five-term argument reductions for random x values logarithmically distributed in $[0, 1000]$ show that they produce correctly rounded r values about 97% of the time for 32-bit arithmetic, and 99% of the time for 64-bit arithmetic. However, in the remaining 1% to 3% of the tests where that is not the case, the original reduction can produce errors of thousands of ulps, whereas the extended version keeps the errors below one ulp.

It is clear that improving the accuracy of the machine representation of π helps, so the argument reduction should *always* be performed in the highest available hardware precision.

9.2 Exact argument reduction

About 1982, Mary Payne and Robert Hanek at DEC [PH83a, PH83b], and independently, Robert Corbett at the University of California, Berkeley [Ng92], proposed and implemented trigonometric argument reduction that almost always produces r values correct to the last digit, for *all* representable x values.

The obvious brute-force way to reach that goal is to use multiple-precision arithmetic, with the reduction constants, such as π or $\pi/2$, represented to at least t digits more than those of a number whose digits span the entire floating-point range. With a package developed by this author, the reduction then takes the form:

```
fp_t r, x;
mp_t r_mp, n_mp, x_mp;

MP_FROM_FP_T(&x_mp, x);                 /* convert x to m.p. */
MP_MUL(&n_mp, &x_mp, &one_over_pi);     /* n = x * (1/PI) */
MP_ROUND(&n_mp, &n_mp);                 /* n = round(n) */
MP_FMS(&r_mp, &n_mp, &pi, &x_mp);       /* r = n * PI - x */
r = -MP_TO_FP_T(&r_mp);                 /* r = x - n * PI */
```

Additional code is required to extract the low-order bit of n, which is needed to determine the correct sign of the result (see the first block of Table 9.1 on page 244).

A one-time initialization computes the two stored constants from the decimal representation of π:

```
static mp_t pi, one_over_pi;
mp_t one;

MP_FROM_STRING(&pi, PI_STR);          /* convert PI string to m.p. */
MP_FROM_FP_T(&one, FP(1.0));          /* convert 1.0 to m.p. */
MP_DIV(&one_over_pi, &one, &pi);      /* form 1/PI */
```

For t-digit arithmetic in base β, the number of base-β digits needed appears to be about EMAX $-$ EMIN $+ 2t$, where the two macros are the exponents of the largest and smallest normal numbers. We need $2t$, rather than t, extra digits to allow for subnormals, because their digits extend up to t more digits to the right in a fixed-point representation. However, the computations should be done with a few extra digits to allow for correct rounding, with high probability, of halfway cases. The macro PI_STR is therefore defined to be a string of about 220 digits for 32-bit arithmetic, 810 digits for 64-bit arithmetic, and 12 370 digits for the 80-bit and 128-bit formats. Those string lengths suffice for both binary and decimal arithmetic, as well as for most historical architectures (see **Table 4.2** on page 65, **Table D.1** on page 929, and **Appendix H** on page 947). However, a future 256-bit decimal format will require about 3 145 900 digits.

To make the required number of digits more precise, in unpublished notes dated March 1983 and titled *Minimizing* $q \times m - n$, Kahan shows how to compute the closest rational approximation, n/m, to an irrational number q, for m in a prescribed interval, using continued fractions (see **Section 2.7** on page 12). He implemented the idea in a BASIC program, now apparently lost. Fortunately, the code was reimplemented in C a few months later by graduate student Stuart McDonald in a program file nearpi.c that contains Kahan's notes in a leading comment block, and that file is available online [Kah83].

McDonald's program is written in old-style C specifically for the VAX architecture, and requires an input file with up to 2500 terms of the continued-fraction expansion of π, terminated with a negative value. Unfortunately, that input file is not included with the program. This author cleaned up the code for use with modern compilers and architectures to produce a companion file fixed-nearpi.c that is included with the mathcw library distribution. The missing data file can be created with just two UNIX command lines when Maple is available:

```
% printf 'convert(Pi, confrac, 2400);' | maple -q | sed -e 's/[][,]/\n/g' -e 's/ //g' > nearpi.dat
% printf "%d\n" -1 >> nearpi.dat
```

McDonald's code is complex because it simulates arithmetic of high precision without providing a general multiple-precision system. Also, it produces lengthy output that must be sorted to find the smallest residual, $q \times m - n$, and the corresponding values of m and n.

The work at DEC appeared only in a newsletter of small circulation and seems to have received little notice outside the numerical-analysis community, and the Berkeley work was never published. About decade later, Roger Smith at Motorola independently discovered and published the continued-fraction method for finding the worst cases for argument reduction [Smi95].

Jean-Michel Muller gives a much shorter, and more general, program in Maple [Mul97, pages 152–153] that finds the worst cases for argument reduction with respect to any constant C, because several other elementary functions also require accurate reduction. Muller also provides a good description of the continued-fraction analysis. A version of Muller's program modified to produce more compact output is included in the file jmmred.map in the mathcw library distribution. Muller's algorithm requires experimentation to find a suitable precision for the computation, so an extra procedure added to that file provides a way to repeat the job for a range of precisions, and yet another procedure encapsulates the computations at the precisions needed for several floating-point systems.

Table 9.2 on the next page summarizes the results of Muller's program for current and important historical architectures for reduction with respect to $\pi/2$ (a common choice in many implementations of the cosine and sine) and π. There are several points to note about the data in that table:

- Although the results for the smaller ranges are computed in a few seconds on a fast workstation, those for the widest ranges take about 15 minutes each.
- The computations must be carried out in a decimal precision that is a few percent larger than the number of decimal digits in the integer part of the maximum normal number.
- The worst-case argument for trigonometric reduction is not always a huge number, because four platforms in that table have negative exponents in the fourth column, and the PDP-10 36-bit case has a small exponent. For

Table 9.2: Exactly representable values near $n\pi/2$ and $n\pi$. These are the *worst cases* for trigonometric argument reduction.
`Digits` is the decimal precision required in Maple to determine the worst case, and t is the platform precision in base-β digits. The base is recorded in the power in the fourth column.
The floor of entries in the second-last column is roughly the number of leading zero base-β digits in the residual $r = x - n\pi/2$. See the text for examples.
The last column is the minimum number of decimal digits required in the representation of $\pi/2$. The digit count needed for reduction by π may differ by one from that number.
The largest exponent for the sample decimal calculator is 99, whereas that for the other systems is the expected `EMAX`. That value is available as the constants `FLT_MAX_EXP`, `DBL_MAX_EXP`, and `LDBL_MAX_EXP` in `<float.h>`, and similar ones in `<decfloat.h>`.
The worst cases for reduction with respect to π are double those for reduction by $\pi/2$, except for eight cases noted in the first column.

Platform	`Digits`	t	**worst-case** x		$-\log_\beta(r)$	$d(\pi/2)$
Decimal 10D calculator ($\pi/2$)	120	10	8 248 251 512	$\times 10^{-6}$	11.67	121
Decimal 10D calculator (π)	120	10	4 125 967 807	$\times 10^{14}$	11.40	121
IBM S/360 32-bit ($\pi/2$)	90	6	13 953 140	$\times 16^{37}$	7.38	91
IBM S/360 32-bit (π)	90	6	10 741 887	$\times 16^{3}$	6.97	91
IBM S/360 64-bit	100	14	14 635 640 253 018 919	$\times 16^{34}$	15.31	109
IBM S/360 128-bit ($\pi, \pi/2$)	120	28	5 148 095 012 591 940 008 617 327 381 075 435	$\times 16^{34}$	29.82	140
IEEE 754 32-bit	50	24	16 367 173	$\times 2^{72}$	29.21	72
IEEE 754 64-bit	330	53	6 381 956 970 095 103	$\times 2^{797}$	60.89	380
IEEE 754 80-bit	5000	64	17 476 981 849 448 541 921	$\times 2^{10531}$	75.54	5019
IEEE 754 128-bit	5000	113	8 794 873 135 033 829 349 702 184 924 722 639	$\times 2^{1852}$	123.25	5082
IEEE 754 32-bit dec. ($\pi, \pi/2$)	110	7	4 327 189	$\times 10^{42}$	9.72	113
IEEE 754 64-bit dec. ($\pi, \pi/2$)	410	16	8 919 302 781 369 317	$\times 10^{296}$	19.22	420
IEEE 754 128-bit dec. ($\pi, \pi/2$)	6200	34	9 308 532 438 209 917 461 067 659 354 862 169	$\times 10^{4639}$	37.68	6216
PDP-10 36-bit	50	27	133 620 862	$\times 2^{2}$	30.83	75
PDP-10 KA10 72-bit	140	54	12 822 055 925 551 548	$\times 2^{-48}$	60.49	111
PDP-10 KL10 72-bit D-floating	140	62	2 539 651 352 978 210 059	$\times 2^{-52}$	67.21	121
PDP-10 KL10 72-bit G-floating	330	59	557 088 013 743 460 028	$\times 2^{871}$	66.84	387
VAX 32-bit F-floating	50	24	16 367 173	$\times 2^{72}$	29.21	71
VAX 64-bit D-floating	60	56	25 644 111 851 103 096	$\times 2^{-49}$	60.49	113
VAX 64-bit G-floating	330	53	6 381 956 970 095 103	$\times 2^{797}$	60.89	379
VAX 128-bit H-floating	5000	113	8 794 873 135 033 829 349 702 184 924 722 639	$\times 2^{1852}$	123.25	5082

the latter, the reduction looks like this:

$$
\begin{aligned}
x &= 133\,620\,862 \times 2^2 \\
&= 534\,483\,448, \\
n &= \operatorname{round}(x/(\pi/2)) \\
&= 340\,262\,731, \\
r &= x - n\pi/2 \\
&= 534\,483\,448 - \\
&\quad 534\,483\,448.000\,000\,000\,522\,316\,639\,536\,881\,693\ \ldots \\
&= \qquad\qquad -0.000\,000\,000\,522\,316\,639\,536\,881\,693\ \ldots
\end{aligned}
$$

The worst case in the ten-digit calculator for reduction with respect to π involves a string of 25 consecutive 9's, and loss of 35 leading digits:

$$\begin{aligned} r &= 412\,596\,780\,700\,000\,000\,000\,000\ - \\ &\quad\ 412\,596\,780\,699\,999\,999\,999\,999.999\,999\,999\,995\,979\,186\,446\,461\ \ldots \\ &= \qquad\qquad\qquad\qquad\qquad 0.000\,000\,000\,004\,020\,813\,553\,538\ \ldots \end{aligned}$$

- Recovery of the residual $r = x - n\pi/2$ needs as many digits in $\pi/2$ as there are before the point in x, plus enough digits after the point to extract r to full accuracy. The decimal-digit counts in the last column are notably smaller than our earlier rough predictions, and do not include extra digits needed for correct rounding.

- The number of leading zeros in r predicted by the second-last column when $\beta = 2$ is not far from the sum of t and the exponent width. That allows us to make a reasonable estimate of the worst cases for the future 256-bit formats. Those cases are unlikely to be computable in Maple because of the much larger number of digits required for the computation, and the much greater exponent range to be investigated.

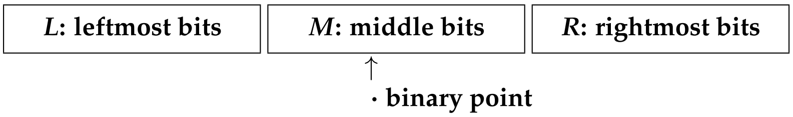
- For the narrower exponent ranges of older systems and shorter formats, a few dozen words can represent $\pi/2$ to sufficient accuracy, and even a brute-force multiple-precision reduction should be reasonably fast.

9.3 Implementing exact argument reduction

The Payne/Hanek algorithm [PH83a, PH83b] is difficult to understand. Its presentation occupies several pages of their articles, and they provide no information at all about its conversion into a computer program.

The critical observation that led to their discovery is that the reduction $r = x - nc$ involves subtraction from x of a product nc that possibly requires thousands of digits for current floating-point architectures, yet the result, r, needs at most a few dozen digits. Because of the relations in Table 9.1 on page 244, we do not require an exact value for n, but only knowledge of a few of its low-order bits. If we can avoid computing the digits that are discarded in the subtraction, then we have a fast algorithm for argument reduction.

The many-digit integer n is obtained from $\operatorname{round}(x/c)$, so the most difficult job is the multiplication of x by $1/c$. If we consider that product as a binary number, then we can split it into three nonoverlapping parts, like this:



The binary point lies somewhere inside M, and if we ensure that there are at least *three* bits in M to its left, then L, extended with m (≥ 3) zero bits to the position of the point, must be an integer divisible by eight. For $c = \pi$, $c = \pi/2$, and $c = \pi/4$, $L \times 2^m \times c$ is therefore a multiple of 2π. That value is the period of the cosine and sine, so $L \times 2^m \times c$, containing possibly thousands of digits, cannot affect the function values, and thus, need not be computed at all.

It is most convenient to compute r indirectly, by writing $x = nc + r = (n + f)c$, where $|f| \leq \frac{1}{2}$ and $r = fc$. Then $x/c = n + f$, and when f is positive, it is just the fraction whose bits follow the binary point in M, and continue into R. We defer consideration of negative f until later.

Smith [Smi95] splits x into the exact product of a t-bit integer, X, and a power of two, an operation for which we have standard library functions.

To determine how many bits are required in M to compute a correctly rounded value for f, we need to know how many zero bits there can be after the binary point, in the worst case. That question is answered by the results in the second-last column of Table 9.2 on the preceding page. From that table, for IEEE 754 32-bit arithmetic, we have at most $w = 29$ worst-case zero bits after the point. We need $t = 24$ more for the significand of r, plus a few more, say g, guard bits for correct rounding of f.

If we include $m + w + t + g$ bits in M, then f is determined to machine precision entirely from M, and the part R, which contains an infinite number of bits when c is irrational, can be ignored. Although we know g, t, and w, we do not yet know a precise value for m.

M is too big to represent in any standard integer or floating-point type, so we need to store it in a multiword array. It is most convenient to use the same representation for M, $1/c$, and X, as polynomials in a large base, D:

$$1/c = \sum_{k=0}^{\texttt{n\_c}-1} \texttt{cinv}[k] \times D^{-k},$$

$$M = 2^{\texttt{-nf}} \sum_{k=0}^{\texttt{n\_m}} \texttt{cm}[k] \times D^{k},$$

$$X = \sum_{k=0}^{\texttt{n\_x}} \texttt{cx}[k] \times D^{k}.$$

The number of fractional bits in M is `nf`. It determines the position of the binary point, but we defer its evaluation.

The coefficients are whole numbers lying in $[0, D)$, and they can be stored as either integer or floating-point values, with the constraints that D must be chosen such that sums of a few products of coefficients can be computed exactly, and that D is a power of the base β.

For example, we could pick $D = 2^{12} = 8^4 = 16^3 = 4096$ for the case of an `int` representation, because all current and historical implementations of C provide at least 16 bits for that data type. A product of two coefficients can then be represented in 24 bits, and C89 guarantees that a `long int` contains at least 32 bits. That type provides the extra bits needed for the sum of products.

For a floating-point representation of the coefficients, a reasonable choice for $\beta = 2$ and $\beta = 16$ is $D = 2^{20} = 16^5 = 1\,048\,576$, because that can be stored in a `float` on all architectures (the wobbling precision of IBM System/360 limits portable code to at most 21 bits). For $\beta = 8$, choose $D = 2^{21} = 8^7 = 2\,097\,152$ instead. A value of type `double` can then hold an exact product of two coefficients on all current and historical platforms (see **Table H.1** on page 948), except for the deficient Harris systems.

The more bits that can be held in a coefficient, the fewer the number of terms in the expansions of $1/c$, M, and X. Computation of $M \times X$ requires two nested loops to form the coefficient products, so we expect that doubling the number of bits in a coefficient should halve the loop limits, and make the loop pair run up to four times faster. However, the platform-dependent relative speed of integer and floating-point arithmetic, and the additional complication of compiler optimization and our limited flexibility in the choice of D, makes it impossible to predict which of the two data types is optimal. In the `mathcw` library, we therefore provide two families of functions for argument reduction, with these prototypes:

```
fp_t EREDUCE(fp_t x, fp_t c, int n_c, const fp_t cinv[/* n_c */], int *pn, fp_t *perr, int b);
fp_t ERIDUCE(fp_t x, fp_t c, int n_c, const int  cinv[/* n_c */], int *pn, fp_t *perr, int b);
```

The argument `c` is the reduction constant correct to working precision, and `n_c` is the number of coefficients stored in `cinv[]`, which represents $1/c$ to high precision. The eight low-order bits of n are returned through the pointer `pn`, and `b` is the maximum number of bits used in a coefficient, implicitly defining $D = 2^b$. The functions return the correctly rounded reduced argument, r.

If their last pointer argument is nonzero, the functions also provide a correction such that the sum of the return value and that correction represents the reduced argument to roughly twice working precision. The correction term can later be used for fine-tuning computed results to adjust for the rounding error in the reduced argument.

Extensive testing of those function families on all major desktop architectures shows that the integer versions are uniformly faster, by about 10% to 30%. In the following, we therefore present the code only for integer coefficients. The code for the functions with floating-point coefficients differs in about 20 of the 150 or so nonempty lines of code (excluding comments and assertions), due to the change in data types and operations needed to work on the data.

Although the argument reduction provided by our functions is primarily of interest for the trigonometric elementary functions, they might also be useful elsewhere, so we provide them as public functions, and take additional precautions in their coding to ensure that they work as intended.

Because the reduction code is lengthy, we present it as a semi-literate program, with descriptive prose followed by fragments of code. The algorithm depends upon exact extraction of powers of two, and is therefore usable only when the base is two, or an integral power thereof. Thus, although we provide decimal companions, their invocation generates a run-time error and job termination through a failing assertion:

```
fp_t
ERIDUCE(fp_t x, fp_t c, int n_c, const int cinv[/*n_c*/], int *pn, fp_t *perr, int b)
```

```
{
    fp_t err, result;
    int n;

#if !((B == 2) || (B == 8) || (B == 16))
    err = ZERO;
    n = 0;
    result = SET_EDOM(QNAN(""));
    assert((B == 2) || (B == 8) || (B == 16));
#else
```

With a suitable base, we can begin with the usual tests for NaN and Infinity arguments:

```
    fp_t xabs;

    err = ZERO;
    n = 0;

    if (ISNAN(x))
        result = SET_EDOM(x);
    else if (ISINF(x))
        result = SET_ERANGE(x);
```

Next, we compute the absolute value of x, and test for the important case where the argument is already in the reduced range $[-c/2, +c/2]$:

```
    else if ( (xabs = QABS(x), xabs <= HALF * c) )
        result = x;     /* no reduction necessary: x = 0*c + r */
```

The trigonometric functions make a similar check, and use a faster algorithm when x is not too large. However, because the code is available to users, we need the check here as well: assumptions in the next block require that n be nonzero. Compared to the work in the remainder of the code, the cost of the three initial tests is negligible.

A negative x is accounted for later, just before the function returns.

When $\beta \neq 2$, the expression HALF * c may not be computed exactly. On such systems, it is better to make the test externally with a correctly rounded value of $c/2$, and avoid calling the function unless $|x| > c/2$.

The real work begins in the next block, which starts with declarations of three argument-dependent constants:

```
    else
    {
        const fp_t D_ = (fp_t)((long int)1 << b);
        const fp_t D_inv = ONE / D_;
        const unsigned long int chunk_mask = (unsigned long int)(((long int)1 << b) - (long int)1);
```

We use D_ for the expansion base to avoid a conflict with the macro D that is used in the mathcw library header files for the decimal precision. Both D_ and D_inv depend on the argument b, but are constant in the remainder of the function. They are *exact* values, as long as b is reasonable; we check that requirement shortly.

The setting of chunk_mask uses type casts before the constants instead of the more compact 1L so that pre-C89 compilers accept the code.

Declarations of about two dozen local variables come next:

```
        fp_t D_to_k, sum, X;
        int c_first, cm[MAX_X_M], cx[MAX_X], e, k, k_first, k_last, m,
            M_first, M_last, nf, n_m, n_rbp, n_x, n_xm, q, shift_count;
        long int cxm[MAX_X_M];
        qp_t cc_inv, dd, dd_inv, ff, rr, xm_int;
        unsigned long int mask;
```

The array dimensions MAX_X and MAX_X_M are set in the header file erid.h. The coefficient arrays all reside on the stack, so they do not take up permanent storage outside the function, and it does not matter if their lengths are a bit larger than needed.

The data type qp_t is the highest available working precision, called quadruple precision on some systems. In deficient C implementations, it may be the same as double.

Those declarations are followed by definitions of two compile-time constants:

```
        static const int g = 30; /* number of guard bits */
        static const int t = FP_T_MANT_DIG;
```

The value of g could be reduced to obtain a small speedup, or increased to make incorrect rounding less likely. The chosen value suggests a rounding-error rate of about once in $2^{30} \approx 10^9$ random arguments, but we soon show that the rate is even lower.

The value of w is set from data in **Table 9.2** on page 252, and overestimating the worst case by a few bits has little or no cost. The guard bits provide additional insurance against an underestimate of w, such as for the case of the future 256-bit type, where it may not be feasible to determine the worst case using Muller's program:

```
#if T <= 27
        static const int w = 30; /* worst-case leading zeros in r */
#elif T <= 56
        static const int w = 61;
#elif T <= 64
        static const int w = 67;
#elif T <= 113
        static const int w = 123;
#elif T <= 237
        static const int w = 250; /* reasonable estimate */
#else
#error "eriduce() family not yet implemented for precision > 237"
#endif
```

We are now ready for some initial setup:

```
        n_x = (FP_T_MANT_DIG + b - 1) / b - 1;
        result = ZERO;
```

The value of n_x is one less than the number of entries available in the array cx[] for holding the polynomial expansion of X.

Next, we check that some critical properties hold in the remainder of the code:

```
        assert(8 <= b);
        assert(b < (int)bitsof(int));

#if defined(HAVE_LONG_LONG_INT)
        assert(b * 2 < ((int)bitsof(long long int) - 5));
#else
        assert(b < ((int)bitsof(long int) - 3));
#endif

        assert(b * 2 <= QP_T_MANT_DIG);
        assert((b / LOG2_B) * LOG2_B == b);
        assert(n_x < (int)elementsof(cx));
```

The bitsof() macro has a simple definition in the header file prec.h:

```
#define bitsof(type)        (CHAR_BIT * sizeof(type))
```

It localizes a potential portability problem for historical word-addressed machines, such as the early Cray systems, and the DEC PDP-10. The C99 Standard has this to say:

> *The* sizeof *operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is an integer. If the type of the operand is a variable length array type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant.*

> *When applied to an operand that has type `char`, `unsigned char`, or `signed char`, (or a qualified version thereof) the result is 1. When applied to an operand that has array type, the result is the total number of bytes in the array. When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding.*
>
> *The value of the result is implementation-defined, and its type (an unsigned integer type) is `size_t`, defined in `<stddef.h>` (and other headers).*

On word-addressed systems, packing characters into words may leave bits unused, so the Standard's assumption that larger types are multiples of the byte size may not hold. On the 36-bit PDP-10, bytes can be any size from 1 to 36 bits, and characters are normally seven-bit values stored five to a word, with the rightmost (low-order) bit of the word set to zero. Fortunately, the C implementations on that system by default store four nine-bit characters in a memory word, and set `CHAR_BIT` to 9. The early Cray systems have 64-bit words, with `CHAR_BIT` set to 8, so they pose no problems.

The six assertions test conditions that are always true when the function is invoked by other members of the mathcw library, but might not be when it is called by user code:

- We guarantee that the eight low-order bits of n can be reported to the caller through the argument `pn`. Ensuring that `b` is at least eight simplifies later code.

- Coefficient elements of type `int` must be able to hold a sign and `b` data bits.

- Sums of up to $\texttt{n\_x} + 1$ products of two coefficients must be representable in values of `long int`. Our test ensures that up to $2^5 = 32$ products can be summed exactly. That is enough for sizes up to the 128-bit format with $t = 113$: when $b = 8$, we have $\texttt{n\_x} + 1 = 15$. It also handles the future 256-bit type with $t = 237$, where the worst case of $b = 8$ has $\texttt{n\_x} + 1 = 30$.

- Two b-bit chunks must be exactly representable in the significand of type `qp_t`, so that we can extract the rightmost bits of the integer part, n.

- The value $D = 2^b$ must be a power of the base β. The asserted expression follows from the requirement $2^b = \beta^j$. Taking base-2 logarithms gives $b = j \log_2(\beta)$, which means that $b / \log_2(\beta)$ must be an integer value.

- The array `cx[]` must be large enough to contain all of the bits of the significand of x.

Job-terminating assertions are normally undesirable in library code. However, our library is designed to be free of direct I/O, and the only reasonable alternatives for indicating an argument-data error would be to return a NaN, or to introduce an additional error-code argument, both of which require post-call checks in user code. A NaN return provides no clue about what is wrong, and no other functions in the C library or the mathcw library use error-code arguments. If an assertion fails, the termination message includes a report of the unsatisfied condition, which should allow the programmer to correct the erroneous calling routine.

A library installer may exploit a requirement of the C language: defining the symbol `NDEBUG` at compile time disables all assertions, without requiring code modifications.

Three initializations clear the coefficient arrays:

```
(void)memset(cxm, 0, sizeof(cxm));
(void)memset(cx,  0, sizeof(cx));
(void)memset(cm,  0, sizeof(cm));
```

The `memset()` calls set the coefficient arrays to numerical zeros, whether they are integers, or binary, octal, or hexadecimal floating-point values. A pattern of all bits zero is not a canonical decimal floating-point zero (see Section D.3 on page 931), but decimal arithmetic is excluded by our base restriction.

The first computational job is to determine X from $|x|$:

```
(void)FREXP(xabs, &e);
X = LDEXP(xabs, t - e);
```

That guarantees that $X = |x| \times 2^{t-e}$.

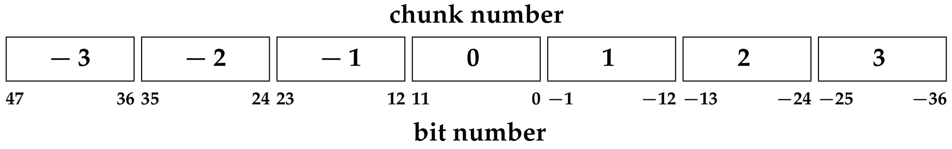
For $c = \pi/4$, $|x|$ exceeds $\pi/8 \approx 0.785 \times 2^{-1}$. The other choices of $c = \pi$ and $c = \pi/2$ have larger minimal $|x|$ values. For argument reduction in the exponential functions, we are likely to need values $c = \ln(\beta)$, the smallest of which is $\ln(2) \approx 0.693 \times 2^0$. We conclude that for the likely choices of c, the condition $e \geq -1$ always holds.

The next task is to split X into its coefficient array by solving for `cx[k]` in the expansion of X given earlier on page 254:

```
D_to_k = LDEXP(ONE, n_x * b);   /* D_**n_x */
assert(D_to_k < FP_T_MAX);      /* check for overflow */
sum = ZERO;

for (k = n_x; k >= 0; --k)
{
    cx[k] = (int)FLOOR((X - sum) / D_to_k);
    sum += (fp_t)cx[k] * D_to_k;
    D_to_k *= D_inv;
}
```

All operations inside the loop are *exact*, and cannot overflow.

Bits in M are numbered by their binary powers, like this, where b has the recommended value of 12:



Smith's results for the indexes of the first and last bits in M can be computed like this:

```
q = g + w + 1;
M_first = t - e + 1;
M_last = -(t + 1 + e + q);
```

We showed earlier that, for the likely choices of c, the smallest value of e is -1, for which `M_first` $= t + 2$. Larger values of x and e reduce the value of `M_first`. In the worst case, `M_first` is positive, and we need to handle negative chunk numbers, even though we don't store any data at negative coefficient indexes.

The obvious way to convert between bit numbers and coefficient chunk numbers requires different code for positive and negative numbers. Although we could handle that with a C ternary operator, a better solution is to index bits starting with a chunk far enough to the left that it will never be referenced. Smith's analysis shows that we do not require references to more than a couple of dozen negative chunks, specifically, $-\lceil (t+2)/b \rceil$. We therefore pick chunk -63 as our starting point, and count bits to the right from that. Two macros in the header file `erid.h` hide the indexing complexity:

```
#define LO_BITNUM_OF_WORDNUM(n) (-(n) * b)
#define WORDNUM_OF_BITNUM(n)    (-63 + ((int)(64 * (unsigned int)b) - 1 - n) / b)
```

Because b is in $[8, 13]$ on most systems, there is no possibility of integer overflow in the expansion of the macro `WORDNUM_OF_BITNUM()`. An optimizing compiler might recognize that an unsigned multiplication by 64 can be replaced by a faster six-bit left shift, but the integer division cannot be avoided because `b` is not a compile-time constant. In any event, there are only two evaluations of that macro for each call to `ERIDUCE()`.

The indexes `M_first` and `M_last` allow us to determine how many bits there are in M, which we need to know to estimate the cost of the argument reduction:

$$\begin{aligned}
\text{bitlength}(M) &= \texttt{M\_first} - \texttt{M\_last} + 1 \\
&= (t - e + 1) - (-(t + 1 + e + q)) + 1 \\
&= 2t + q + 3 \\
&= 2t + g + w + 4.
\end{aligned}$$

Because we have chosen $g = 30$, and **Table 9.2** on page 252 shows that w is approximately $t + 5$ to $t + 15$, we see that M has at most $3t + 49$ bits. In other words, even though we need $1/c$ to more than 16,000 bits for the 128-bit formats, and to about a half-million bits for a future 256-bit format, we never need to compute with bit strings longer than about 120, 210, 240, 390, and 760 bits in the five extended IEEE 754 binary formats. Because the complexity of schoolbook multiplication is proportional to the *square* of the number of digits, that is a huge savings — it is almost a million times faster for the 256-bit type.

For the limited precisions of older architectures, 200-bit computations could have provided exact argument reduction for elementary functions in all programming languages, had the Payne/Hanek discovery been made thirty years earlier, and more clearly described with freely available, published, and portable, working code. Instead, at the time of writing this, only a few compiler vendors have implemented exact argument reduction. That may be due to insufficient interest or understanding, or to lack of programmer awareness of the technique, or to the complexity of its original description.

The indexes of the coefficients in `cm[]` that contain the first and last bits of M are given by these variables:

```
k_first = WORDNUM_OF_BITNUM(M_first);
k_last = WORDNUM_OF_BITNUM(M_last);
assert(k_first <= k_last);
```

In the first word, we have to extract just the bits that we need before doing arithmetic, but any extra bits in the last word are just bonus guard bits that enhance the probability of correct rounding.

Of course, it is *possible*, if unlikely, that there are x values for which our choice of g is too small. Kahan wrote this remark about logarithm computations [Kah04a], and it applies equally well to the problem of argument reduction with respect to an irrational constant c:

> *No general way exists to predict how many extra digits will have to be carried to compute a transcendental expression and round it correctly to some preassigned number of digits. Even the fact (if true) that a finite number of extra digits will ultimately suffice may be a deep theorem.*

We are now ready to compute the limits for the arrays `cm[]` and `cxm[]`, and verify that the compile-time storage sizes are sufficient:

```
k = M_last - LO_BITNUM_OF_WORDNUM(k_last) + 1;
M_last -= k;
q += k;
n_m = k_last - k_first;
n_xm = n_x + n_m;

assert(0 <= n_m);
assert(n_m < (int)elementsof(cm));
assert(0 <= n_xm);
assert(n_xm < (int)elementsof(cxm));
assert(xabs == LDEXP(X, e - t));
```

We need a mask to remove any bits in the first chunk that lie before bit number `M_first`, and we make some more sanity checks:

```
shift_count = M_first - LO_BITNUM_OF_WORDNUM(k_first) + 1;
assert(0 < shift_count);
assert(shift_count <= b);
mask = ~((~((unsigned long int)0)) << shift_count);
```

Later computation is simplified if we copy the bit string from M into its own array, `cm[]`, taking care to provide zero values for negative indexes in `cinv[]`:

```
c_first = (k_first < 0) ? 0 : cinv[k_first];
c_first &= mask;

cm[k_last - k_first] = c_first;

for (k = k_first + 1; k <= k_last; ++k)
    cm[k_last - k] = (k < 0) ? 0 : cinv[k];
```

We are now ready to multiply X by M. This pair of loops is the *hot spot* of the code, taking most of the execution time:

```
    for (k = 0; k <= n_x; ++k)
    {
        long int cx_k;

        cx_k = cx[k];

        for (m = 0; m <= n_m; ++m)
        {
            long int hi, lo, sum;

            sum = cxm[k + m] + cx_k * (long int)cm[m];
            hi = sum >> b;
            lo = sum & chunk_mask;
            cxm[k + m] = lo;
            cxm[k + m + 1] += hi;
        }
    }

    if (cxm[n_xm + 1] != (long int)0)
        n_xm++;
```

The if statement that follows the loop pair increments the element count if one more coefficient is needed.

Profiling with logarithmically distributed random arguments shows that the element cx[k] is zero with probability $< 1/20\,000$, so we do not bother to skip the inner loop in that rare case. The integer multiply-add operation that forms sum generally produces a value outside the range $[0, D)$, so we split it into high and low parts. The low part is stored directly, but the high part increments the next higher coefficient.

The original prototype for the code delayed the split until after the loop pair, making the code slightly faster because the split is then done $\mathtt{n\_x} + 1$ fewer times. However, the final version of the loop pair is more complex, because we actually have three versions of it, one of which is dropped during preprocessing. The reason for the extra code is that we want to remove the restriction that a sum of products must fit in a long int. On historical and many current machines, that data type has at least 32 bits, forcing the restriction $b < 14$, and other constraints asserted above reduce it further to $b = 12$. If we could use the C99 long long int, which is mandated to be at least 64 bits, we could allow b to increase to 24, and on some systems, 28, making the loop pair potentially four times faster.

At the time of writing this, most C compilers on current systems provide a 64-bit integer type, although limitations of the underlying hardware may force it to be implemented in software. The actual code therefore tests the value of b, and when $b < 14$, uses a loop pair with a long int accumulator, and otherwise uses loops with an unsigned accumulator of data type UINT_LEAST64_T, which is the mathcw library name for C99's uint_least64_t defined in the C99 system header file <stdint.h>. That header file is included by our header file inttcw.h only in a C99 environment; otherwise, inttcw.h uses <limits.h> to determine whether a 64-bit type is available.

For older systems that have no support for a 64-bit integer type, we provide our own implementation as a pair of UINT_LEAST32_T values. That loop pair looks like this:

```
    for (k = 0; k <= n_x; ++k)
    {
        UINT_LEAST32_T cx_k;

        cx_k = (UINT_LEAST32_T)cx[k];

        for (m = 0; m <= n_m; ++m)
        {
            UINT_LEAST32_T result[2];
            long int hi, lo;

            umul64(result, cx_k, (UINT_LEAST32_T)cm[m]);
            uadd32(result, (UINT_LEAST32_T)cxm[k + m]);
            hi = (long int)((result[0] << (32 - b)) + (result[1] >> b));
```

```
                lo = (long int)(result[1] & chunk_mask);
                cxm[k + m] = lo;
                cxm[k + m + 1] += hi;
            }
        }
```

The inner loop invokes two private functions: `umul64()` produces a double-length product, and `uadd32()` adds a 32-bit integer to that product. The split into high and low parts is slightly more complex, because the high part is a *b*-bit chunk that overlaps both words of the sum of products. However, only *four* statements in the inner loop need to be aware of the software arithmetic, whereas had we kept the original code that renormalized the coefficients `cxm[]` outside the loop pair, the task would have been more difficult.

We do not address multiple-precision integer arithmetic elsewhere in this book, so we temporarily drop out of the discussion of the body of `ERIDUCE()` to display the code for two new functions, and a helper function.

Because we only need positive numbers for the inner loops of the argument reduction, we can avoid the complexity of signed arithmetic by using only unsigned data types. The multiplication to produce a two-word product is straightforward: split the input factors into 16-bit chunks, then accumulate their four 32-bit products in the `result[]` array:

```
#define MASK16  ((UINT_LEAST32_T)0xffff)

static void
umul64(UINT_LEAST32_T result[2], UINT_LEAST32_T a, UINT_LEAST32_T b)
{       /* result[0] = high product, result[1] = low product */
    UINT_LEAST32_T aa[2], bb[2];

    aa[0] = (a >> 16) & MASK16;
    aa[1] = a & MASK16;

    bb[0] = (b >> 16) & MASK16;
    bb[1] = b & MASK16;

    result[0] = aa[0] * bb[0];
    result[1] = aa[1] * bb[1];

    uacc16(result, aa[0] * bb[1]);
    uacc16(result, aa[1] * bb[0]);
}
```

Two of the products are easy, but the other two products overlap the array elements, so we handed them off to this helper function, which adds a shifted value to the double-length product:

```
#define MASK32  ((UINT_LEAST32_T)0xffffffff)

static void
uacc16(UINT_LEAST32_T result[2], UINT_LEAST32_T a)
{       /* return result + (a << 16) */
    UINT_LEAST32_T old, t;

    old = result[1];
    t = (a & MASK16) << 16;
    result[1] += t;

#if INT_MAX != 0x7fffffff
    result[1] &= MASK32;
#endif

    if ((result[1] < old) || (result[1] < t)) /* have carry bit */
        result[0]++;
    result[0] += (a >> 16) & MASK16;

#if INT_MAX != 0x7fffffff
    result[0] &= MASK32;
#endif

}
```

The problem is that the sums can overflow, and we discuss in **Section 4.10** on page 72 that integer overflow is unreported on many systems, neither causing an interrupt, nor setting a status flag. Fortunately, we can detect the overflow by exploiting a feature of unsigned arithmetic: an overflow bit in addition is simply dropped. Thus, if we form the unsigned sum $u + v$ and find that it is less than either u or v, overflow has occurred. In that case, we have a carry bit that must be added to the high part.

Our code also handles machines with word sizes other than 32 bits, such as the 36-bit PDP-10. The preprocessor then exposes two masking operations that truncate the two words of the `result[]` array to 32 bits.

The remaining function that adds a 32-bit value to a pair of 32-bit values also requires overflow detection, and masking to 32 bits when the word size differs from 32:

```
static void
uadd32(UINT_LEAST32_T result[2], UINT_LEAST32_T a)
{       /* return result + a */
    UINT_LEAST32_T old;

    old = result[1];
    result[1] += a;

#if INT_MAX != 0x7fffffff
    result[1] &= MASK32;
#endif

    if ((result[1] < old) || (result[1] < a)) /* have carry bit */
    {
        result[0]++;

#if INT_MAX != 0x7fffffff
        result[0] &= MASK32;
#endif

    }
}
```

You might wonder whether the extra complexity of three loop versions, and unsigned 64-bit software arithmetic, is worthwhile. Timing tests on several systems show that our code with `umul64()` and `uadd32()` is a fast as compiler-generated code that calls run-time library functions to implement 64-bit arithmetic, and our code also works on systems where such compiler support is absent. When native 64-bit hardware instructions are available, the entire argument reduction runs about 30% faster than when our 64-bit software arithmetic is used. Reductions with $b = 12$ and $b = 28$ run about the same speed when software arithmetic is needed. The important thing is that b can be made larger everywhere, reducing the storage requirements for the array `cinv[]` that holds $1/c$.

We now return to our discussion of the body of `ERIDUCE()`. The next task is to identify the location of the binary point in preparation for extracting the low-order bits of n, and the fraction f:

```
        nf = 2 * t + q;                 /* number of fractional bits */
```

The binary point may lie *between* two chunks, or *within* a chunk. The first case is easier to handle, because the low bits of n are just the chunk to the left of the point. The fraction is then just the scaled sum of the remaining chunks in M. By accumulating that sum in the highest available working precision from smallest to largest terms, we hope to get a correctly rounded fraction. Once again, there are sanity checks to be made to ensure that all is well:

```
n_rbp = nf / b - 1;             /* cxm[n_rbp] is RIGHT of binary point */
assert((n_rbp + 2) < (int)elementsof(cxm));
dd = QP_LDEXP(QP(1.0), -nf);
assert(dd > QP(0.0));           /* check for underflow disaster */
ff = (qp_t)cxm[0] * dd;

for (k = 1; k <= n_rbp; ++k)
{
    dd *= (qp_t)D_;             /* exact product */
    ff += (qp_t)cxm[k] * dd;    /* exact product, accurate sum */
}
```

The second case is more difficult, because we need to collect the integer and fractional bits separately. That is best done by converting a two-chunk sequence to a floating-point whole number, scaling it by a power of two to correctly position the binary point, and then using a member of the `modf()` family to extract the whole number and fraction. We must work with *two* chunks, rather than *one*, because we promised to deliver at least eight low-order bits of n, and the chunk containing the binary point may not have that many.

```
if ((nf % b) == 0)                      /* binary point BETWEEN coefficients */
    xm_int = (qp_t)cxm[n_rbp + 1];
else                                    /* binary point INSIDE coefficient */
{
    int nbits;

    nbits = nf - (n_rbp + 1) * b;       /* fractional bits left */
    assert(nbits >= 0);
    ff += QP_MODF(QP_LDEXP((qp_t)cxm[n_rbp + 2] * (qp_t)D_ +
                           (qp_t)cxm[n_rbp + 1], -nbits), &xm_int);
}
```

It is critical that the first argument of `QP_LDEXP()` be exactly representable. We need $2b$ bits, which the earlier assertion that $2b \leq$ `QP_T_MANT_DIG` guarantees. Because `qp_t` is equivalent to `double` on some current systems, that in practice prevents using $b = 28$, forcing us back to $b \leq 24$.

At this point, we have at least b low-order bits of n in `xm_int`, and the fraction in highest precision in `ff`, such that $|x| = (n + f)c$.

We assumed that n was computed by $\operatorname{round}(|x|/c)$, so that f lies in $[0, \frac{1}{2}]$. However, most of n lies in L, the big block of leading bits that we discarded. If the computed f lies outside its promised range, then we can bring it into range by incrementing n by one, and replacing f by $1 - f$. If f is close to one, then that subtraction suffers loss of leading bits, preventing us from computing a correctly rounded value of f.

To avoid subtraction loss, and ensure correct rounding, we need to compute $1 - f$ in more than twice the working precision, but we may lack a large-enough floating-point data type to do so. Here is how to solve that problem. To a mathematician, the values 1 and 0.99999... are equivalent, because they can be made arbitrarily close by appending more nines. That means that $1 - f$ is the same as $0.99999\ldots - f$. That subtraction can be done without digit borrowing by appending trailing zeros to f, and then subtracting each decimal digit of that extended f from nine. The subtraction is just the *nine's-complement* operation, the decimal analog of the *one's-complement* process described in **Appendix I.2.2** on page 972. We are not working with either binary or decimal numbers here, but instead with base-D numbers, where the *D's-complement* equivalent means that we can obtain $1 - f$ by subtracting each base-D digit of f from $D - 1$, and then converting to a floating-point value by adding the base-D digits with suitable scaling.

About half the time, we must therefore discard the value of `ff`, and recompute it in D's-complement arithmetic:

```
if (ff > QP(0.5))                        /* rounding needed */
{   /* produce 1 - f accurately with D_'s complement sum */
    long int DM1;

    xm_int++;

    DM1 = (long int)D_ - (long int)1;
```

```
        dd = QP_LDEXP(QP(1.0), -nf);
        ff = (qp_t)(D_ - cxm[0]) * dd;

        for (k = 1; k <= n_rbp; ++k)
        {
            dd *= (qp_t)D_;                  /* exact product */
            ff += (qp_t)(DM1 - cxm[k]) * dd; /* exact product */
        }

        if ((nf % b) != 0) /* binary point INSIDE coefficient */
        {
            int nbits;
            qp_t junk;

            nbits = nf - (n_rbp + 1) * b;
            ff += QP_MODF(QP_LDEXP((qp_t)(DM1 - cxm[n_rbp + 2]) * (qp_t)D_ +
                                   (qp_t)(DM1 - cxm[n_rbp + 1]), -nbits), &junk);
        }

        ff = -ff;
    }
```

Notice that the initial value of ff uses D, rather than $D - 1$; that is how we provide an unlimited number of trailing zeros in cxm[] that are implicitly subtracted from digits $D - 1$.

There are enough bits to produce an accurate value of $1 - f$ because we have nf $= 2t + q = 2t + g + w + 1 \approx 3t + g + 1$ fraction bits.

We do not require the whole number that is returned via the second argument to the modf() family. Unfortunately, the ISO C Standards do not specify whether a NULL pointer is acceptable for that argument, so we introduce the variable junk just to get a valid pointer.

The job of argument reduction is almost done. We merely need to form $r = fc$. Although we have f as ff in the highest-available precision, we only have c available to working precision. That multiplication would introduce another rounding error that we can likely avoid by instead using the high-precision value of $1/c$ in the array cinv[]:

```
    cc_inv = QP(0.0);
    dd_inv = (qp_t)D_inv;

    for (k = 1; k < n_c; ++k)
    {
        qp_t cc_inv_last;

        cc_inv_last = cc_inv;
        cc_inv += (qp_t)cinv[k] * dd_inv;

        if (cc_inv == cc_inv_last)
            break;

        dd_inv *= (qp_t)D_inv;
    }

    cc_inv += (qp_t)cinv[0];
    rr = ff / cc_inv;
```

Notice that the largest term is added last, after the loop has computed the remainder of the sum to working precision. Even though n_c is large, only a few terms, roughly $\lceil t/b \rceil$, are summed before the break statement is executed.

All that remains now is a final conversion of rr to working precision, correction of the sign of the result if x is negative, extraction and storage of the eight low-order bits of n, and a return from the function:

```
    result = (fp_t)rr;
```

```
        err = (fp_t)(rr - (qp_t)result);
        n = (int)QP_FMOD(xm_int, QP(256.0));

        if (x < ZERO)
            result = -result;
    }
#endif /* !((B == 2) || (B == 8) || (B == 16)) */

    if (perr != (fp_t *)NULL)
        *perr = err;

    if (pn != (int *)NULL)
        *pn = n;

    return (result);
}
```

When qp_t differs from fp_t, the conversion of rr to result introduces a second rounding whose probability is, alas, much larger than that predicted earlier for the reduction. The worst case in current architectures occurs when those types correspond to the IEEE 754 64-bit and 80-bit formats: the additional 11 bits in the significand of the larger type suggest a frequency of incorrect rounding of about $2^{-11} = 1/2048$.

9.4 Testing argument reduction

For the IEEE 754 32-bit format, it is feasible to enumerate every possible argument x in the range $[1, \mathtt{FLT\_MAX}]$, and compare the reduced arguments produced by the float and double exact reduction functions. The test programs exered.c and exerid.c take a few hours to run on several architectures, and report no differences, giving confidence in the accuracy of the reduction algorithm. The reductions for all of the worst cases in Table 9.2 on page 252 have also been verified against high-precision Maple computations.

Tests provided by the files timred*.c compare the relative times and accuracy of the ereduce(), eriduce(), and reduce() families for reductions with respect to $\pi/2$. On all of the major desktop CPU platforms, they show that the exact reductions with integer cinv[] data are 5 to 45 times slower in 32-bit arithmetic, 8 to 67 times slower in 64-bit arithmetic, 11 to 24 times slower in 80-bit arithmetic, and about 13 times slower in 128-bit arithmetic. The slowest results are on systems where the data type qp_t is implemented in software. The faster reduce() family is usable to about $100\pi/2$ in 32-bit arithmetic, $1000\pi/2$ in 64-bit arithmetic, and $1\,000\,000\pi/2$ in 80-bit and 128-bit arithmetic, so in most computations, exact reductions are unlikely to be needed.

9.5 Retrospective on argument reduction

Correctly rounded argument reduction is clearly an intricate process, and the Payne/Hanek algorithm is a significant contribution to the computation of elementary functions. Instead of time-consuming multiple-precision multiplication, rounding, and subtraction with thousands of digits to accurately compute $r = x - \mathrm{round}(x/c)c$, their algorithm permits a much smaller computation that, in the worst case, needs roughly *five times* working precision. Its cost is comparable to that of computing a trigonometric function once a reduced argument is available, and it need not be invoked at all for arguments that are already in the range $[-c/2, +c/2]$, or in a larger range where a simpler reduction, like our REDUCE() function family, can be used.

Kahan's continued-fraction analysis [Kah83] to find the worst cases for argument reduction is a critical component of the reduction algorithm because it provides a suitable value for w. The worst case depends only on the constant c, the precision t, and the exponent range $[-1, \mathtt{FP\_T\_MAX\_EXP}]$. They are all determined by mathematics and by the floating-point architecture, so the worst cases need to be found just once: they do not require run-time determination when the elementary functions are evaluated. The results that are collected in Table 9.2 on page 252 from Muller's Maple program for Kahan's method give a reasonable idea of the maximum number of leading zero bits to be expected in any argument reduction with respect to an irrational constant. When in doubt, just make g or w larger — an extra chunk or two in M does not cost much, and the reward of a correctly rounded reduction is large.

The payoff for correct argument reduction is substantial, because trigonometric functions are pervasive in science and engineering, and users should not have to take special precautions to avoid large arguments, especially because, in the absence of exact analytic reductions, they are likely to use inaccurate reduction techniques. In addition, as we record later in **Table 17.1** on page 479, trigonometric and hyperbolic functions are critical components needed for computation of several other elementary functions in complex arithmetic. The lack of exact argument reduction in most existing mathematical libraries has the possibly surprising side effect of making many functions in complex arithmetic unreliable for most of the floating-point range.

10 Exponential and logarithm

The formula $e^{i\pi}+1=0$ relates Euler's number e
to the imaginary value $i=\sqrt{-1}$, Archimedes' constant π,
and the digits of the binary number system.

The exponential function, and its inverse, the logarithm function, are two of the most fundamental elementary functions in all of mathematics, and in numerical computing. They also turn up hidden inside other important functions, such as the hyperbolic functions, the power function, and in the computation of interest and investment value, and lottery returns, in financial mathematics, a subject that we treat briefly at the end of this chapter.

Neither the exponential nor the logarithm is particularly difficult to compute when high accuracy is not a goal, but we show in this chapter that considerable care is needed to develop suitable computational algorithms when almost-always correctly rounded results are wanted. We also address the need for auxiliary functions for other bases, and for arguments near zero and one.

10.1 Exponential functions

The exponential function, $\exp(x)=e^x$, where $e\approx 2.718\,281\,828\ldots$ is Euler's[1] number, is one of the most important functions in all of mathematics. Euler proved in 1737 that e is *irrational*, and Hermite proved in 1873 that e is also *transcendental*. The exponential function is graphed over a small range in **Figure 10.1** on the next page.

Apart from the curious compact relation given in the epigraph to this chapter, the exponential function has the fundamental property that its slope at a given point (that is, its derivative) is the function value itself, and the area under the curve to the left of x is also the function value. That is, we have these relations:

$$\begin{aligned} d\exp(x)/dx &= \exp(x), \\ \int_{-\infty}^{x} \exp(x)\,dx &= \exp(x). \end{aligned}$$

Other properties follow easily from its definition as a power of e:

$$\begin{aligned} \exp(-\infty) &= 0, & \exp(2x) &= (\exp(x))^2, \\ \exp(-1) &= 1/e, & \exp(\tfrac{1}{2}x) &= \sqrt{\exp(x)}, \\ \exp(0) &= 1, & \exp(x+y) &= \exp(x)\exp(y), \\ \exp(1) &= e, & \exp(x\times y) &= (\exp(x))^y \\ \exp(+\infty) &= +\infty, & &= (\exp(y))^x. \end{aligned}$$

The exponential function has a Taylor series and a continued fraction (see **Section 2.7** on page 12), both valid for any finite complex argument z:

$$\begin{aligned} \exp(z) &= 1+z+z^2/2!+z^3/3!+\cdots+z^n/n!+\cdots, \\ \exp(z) &= 0+\frac{1}{1-}\,\frac{z}{1+}\,\frac{z}{2-}\,\frac{z}{3+}\,\frac{z}{2-}\,\frac{z}{5+}\,\frac{z}{2-}\,\frac{z}{7+}\,\frac{z}{2-}\,\frac{z}{9+}+\cdots. \end{aligned}$$

[1] The Swiss mathematician Leonhard Euler (pronounced *oiler*) (1707–1783) possessed a photographic memory, and was possibly the most prolific mathematician in human history. Euler produced almost half of his works by dictation to secretaries after he went blind in 1771, and publications of his articles continued for several decades after his death. He introduced into mathematics the notations e (base of natural logarithms), i ($\sqrt{-1}$), π (circle constant), Δy (for finite differences), $\sum$ (summation), $f(x)$ (function of x), and the modern names for trigonometric functions.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_10

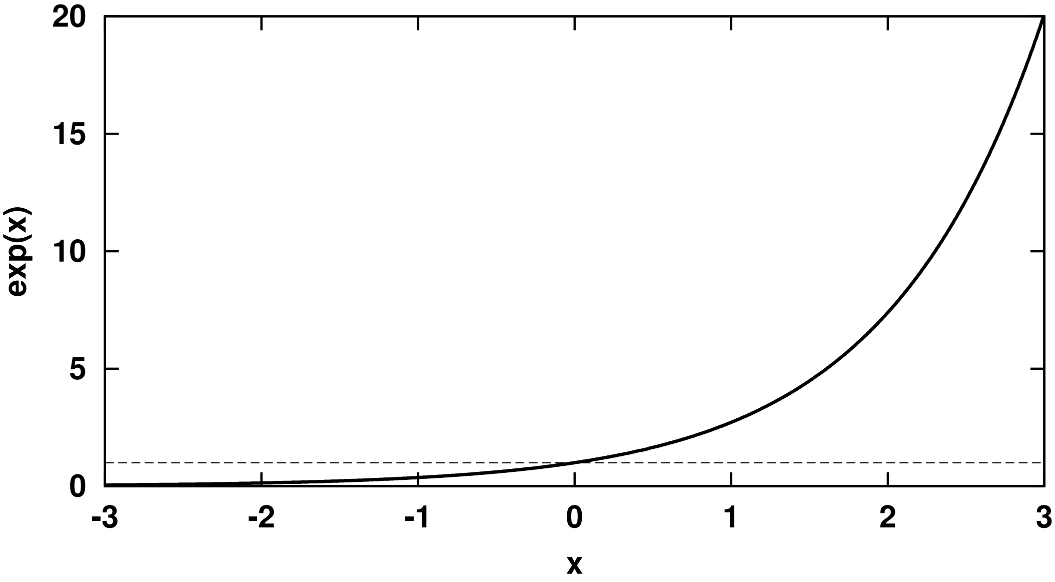


Figure 10.1: The exponential function, $\exp(x)$. The dashed line for $\exp(x) = 1$ intersects the curve at $x = 0$. The function grows rapidly for positive x, with $\exp(6.908) \approx 1000$, $\exp(13.816) \approx 1\,000\,000$, and $\exp(20.724) \approx 1\,000\,000\,000$.

For small x, the convergence of the series and the continued fraction is roughly the same. However, the latter is subject to serious loss of leading digits for large x. That happens because the fraction evaluated last in the backward direction has the form $1/(1+s)$, and for large $|x|$, we have $s \approx -1$.

The error-magnification formula (see Section 4.1 on page 61) for the exponential function looks like this:

$$\operatorname{errmag}(\exp(x)) = x.$$

Thus, the sensitivity of the function to argument errors is proportional to the argument, so when $|x|$ is large, we need high precision.

The number e can be defined by limits, and by a simple continued fraction that Euler first published in 1744:

$$\begin{aligned}
e &= \lim_{x\to\infty} (1 + 1/x)^x, \\
1/e &= \lim_{n\to\infty} \frac{(n!)^{1/n}}{n}, \\
e &= [2; 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, \ldots] \qquad \textit{coefficients } [b_0; b_1, b_2, b_3, \ldots] \\
&= 2 + \frac{1}{1+}\,\frac{1}{2+}\,\frac{1}{1+}\,\frac{1}{1+}\,\frac{1}{4+}\,\frac{1}{1+}\,\frac{1}{1+}\,\frac{1}{6+}\,\frac{1}{1+}\,\frac{1}{1+}\,\frac{1}{8+} + \cdots .
\end{aligned}$$

The two limits converge slowly: with $x = n = 10^{10}$, they produce only about 10 and 9 correct digits of e, respectively. By contrast, the simple continued fraction is computationally practical. The term b_k of the fraction is $2(k+1)/3$ if $k+1$ is a multiple of 3, and otherwise, is 1. Numerical experiments show that after a few terms, each additional term produces about one more decimal digit of e. There is also a recent algorithm, discovered only in 1995, that can produce the first n consecutive fractional decimal digits of e. It requires only hardware integer arithmetic, and uses $\mathcal{O}(n)$ storage and $\mathcal{O}(n^2)$ time [BB04, page 140]. The file `edigs.c` implements the algorithm. Symbolic-algebra systems make it easy to compute e, as in these short sessions:

```
% gp
? \p 60
   realprecision = 77 significant digits (60 digits displayed)
? eval(exp(1))
```

```
%1 = 2.71828182845904523536028747135266249775724709369995957496697

% maple
> evalf(exp(1), 60);
    2.71828182845904523536028747135266249775724709369995957496697

% math
In[1]:= N[E,60]
Out[1]= 2.71828182845904523536028747135266249775724709369995957496697

% maxima
(%i1) fpprec : 60$ bfloat(exp(1));
(%o2)   2.71828182845904523536028747135266249775724709369995957496697b0

% mupad
>> DIGITS := 60: float(E);
  2.71828182845904523536028747135266249775724709369995957496697

% reduce
1: on rounded$ precision 60$ E;
2.71828182845904523536028747135266249775724709369995957496697

% sage
sage: n(e,201)
2.71828182845904523536028747135266249775724709369995957496697
```

The book *e: The Story of a Number* [Mao94] chronicles the history of Euler's number, and the *Handbook of Mathematical Functions* [AS64, OLBC10, Chapter 4] summarizes many of the properties of the exponential function.

In floating-point arithmetic, the exponential function falls off so rapidly for negative x, and grows so rapidly for positive x, that it soon reaches the underflow and overflow limits. In 64-bit IEEE 754 binary arithmetic, underflow to subnormals happens for $x < -708.397$, and overflow for $+709.783 < x$.

In early work on computers, Hastings [Has55, pages 181–184] gave coefficients for polynomial approximations to the exponential function of the form

$$\exp(-x) = 1/(a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots)^4, \qquad \text{for } x \geq 0.$$

Taking the reciprocal of the formula extends coverage to the other half of the real axis. With coefficients up to $a_3 x^3$, he claimed about three correct decimal digits, and with terms up to $a_6 x^6$, almost seven correct decimal digits. Numerical and graphical checks of his formulas demonstrate that the errors grow dramatically outside the small interval $[0, 4]$, and his error plots are overly optimistic. This *must* be the case, because the exponential function grows faster than any finite polynomial of fixed degree.

We can find similar approximations with modern software technology using the Maple symbolic-algebra language, but restricting the interval of approximation to just $[0, 1]$:

```
% maple
> with(numapprox):
> Digits := 60:
> F := proc(x) return root(exp(-x), 4) end proc:
> G := proc(x) return (exp(-x) - 1 + x)/x**2 end proc:
> H := proc(x) return exp(-1/x) end proc:
>
> c := minimax(F(x), x = 0 .. 1, [0, 3], 1, 'maxerror'):
> printf("maxerror = %.2g\n", maxerror):
maxerror = 1.1e-06

> c := minimax(G(x), x = 0 .. 1, [3, 0], 1, 'maxerror'):
> printf("maxerror = %.2g\n", maxerror):
maxerror = 27.7e-06
```

```
> c := minimax(H(x), x = 0 .. 1, [3, 0], 1, 'maxerror'):
> printf("maxerror = %.2g\n", maxerror):
maxerror = 0.0044

> c := minimax(F(x), x = 0 .. 1, [3, 3], 1, 'maxerror'):
> printf("maxerror = %.2g\n", maxerror):
maxerror = 6.5e-14

> c := minimax(G(x), x = 0 .. 1, [3, 3], 1, 'maxerror'):
> printf("maxerror = %.2g\n", maxerror):
maxerror = 3.7e-12

> c := minimax(H(x), x = 0 .. 1, [3, 3], 1, 'maxerror'):
> printf("maxerror = %.2g\n", maxerror):
maxerror = 0.00029
```

The function $F(x)$ is suitable for Hastings-like formulas, but `minimax()` reports failure when the interval of approximation is widened. The function $G(x)$ is derived from the Taylor series, and $H(x)$ uses a reciprocal argument to map the entire positive axis $[0, \infty)$ to $(0, 1]$.

The key to accurate computation of the exponential function is to write it as the product of an *exact* power of the base and a small correction factor:

$$\begin{aligned}
\exp(x) &= \beta^n \exp(g), \\
x &= n \log(\beta) + g, && \textit{by taking logarithms,} \\
n &= \operatorname{round}(x / \log(\beta)), && \textit{nearest multiple of } \log(\beta) \\
g &= x - n \log(\beta), && g \textit{ in } [-\tfrac{1}{2}\log(\beta), \tfrac{1}{2}\log(\beta)].
\end{aligned}$$

The reduced argument g can be a small difference of two large numbers, so to compute it accurately, we must represent the constant $\log(\beta)$ to high precision. The `REDUCE()` family described in **Section 9.1** on page 243 finds g reliably. With $\beta = 2$, we have g in $[-0.347, +0.347]$, and we can use the same representation with a factor 2^n for bases of the form $\beta = 2^K$, where $K > 1$.

For a decimal base, Cody and Waite recommend reducing the argument with respect to a power of $\sqrt{10}$ rather than of 10, because that shrinks the range of g from $[-1.152, 1.152]$ to $[-0.576, 0.576]$:

$$\exp(x) = (\sqrt{10})^n \exp(g).$$

Better approximations than those of Hastings have fits to intermediate functions that are almost linear over the interval of approximation. Later books (see, for example, [HCL+68, page 104], [CW80, Chapter 6], and [Mos89, Chapter 4]) use an expression with a rational polynomial, $\mathcal{R}(g^2) = \mathcal{P}(g^2) / \mathcal{Q}(g^2)$, like this:

$$\begin{aligned}
\exp(g) &\approx \frac{\mathcal{Q}(g^2) + g\mathcal{P}(g^2)}{\mathcal{Q}(g^2) - g\mathcal{P}(g^2)} \\
&\approx \frac{1 + g\mathcal{R}(g^2)}{1 - g\mathcal{R}(g^2)} \\
&\approx \frac{1 - g\mathcal{R}(g^2) + 2g\mathcal{R}(g^2)}{1 - g\mathcal{R}(g^2)} \\
&\approx 1 + \frac{2g\mathcal{R}(g^2)}{1 - g\mathcal{R}(g^2)} \\
&\approx 1 + \frac{2g\mathcal{P}(g^2)}{\mathcal{Q}(g^2) - g\mathcal{P}(g^2)}, \\
\mathcal{R}(g^2) &\approx \frac{\exp(g) - 1}{g(\exp(g) + 1)}.
\end{aligned}$$

For $|g|$ in $[0, +0.347]$, $\mathcal{R}(g^2)$ decreases slowly, and lies entirely in the interval $[0.495, 0.500]$. The last equation for $\exp(g)$ writes it as the sum of an exact term and a possibly small correction. Numerical evaluation of that correction shows that it grows almost linearly over the range of g, and lies in $[0, 0.415]$.

For the function $\mathcal{R}(g^2)$, Maple achieves much better accuracy in the fit. It provides us with compact rational polynomials suitable for each of the extended IEEE 754 formats, and all of the historical formats, that we support in the mathcw library:

```
> R := proc(gg) local e, g: g := sqrt(gg): e := exp(g):
>              return ((e - 1)/(e + 1))/g
>      end proc:

> for pq in [ [1,1], [2,2], [2,3], [4,4], [8,8] ] do
>     c := minimax(R(gg), gg = 0 .. 0.121, pq,  1, 'maxerror'):
>     printf("%a : %.2g\n", pq, maxerror)
> end do:
[1, 1] : 2.7e-10
[2, 2] : 2.5e-18
[2, 3] : 1.3e-22
[4, 4] : 3.5e-36
[8, 8] : 6.1e-76
```

For the wider range of g needed for decimal arithmetic, the 128-bit and 256-bit formats need fits of degree $\langle 4/5 \rangle$ and $\langle 8/9 \rangle$.

Tests of an implementation of the Cody/Waite algorithm for the exponential function in IEEE 754 binary arithmetic show that it produces errors up to about 0.92 ulp, with about 1/70 of the results having errors larger than $\frac{1}{2}$ ulp. In decimal arithmetic, the peak error in `expdf()` is about 1.19 ulp, with about 1/40 of the results differing from correctly rounded values. See **Figure 10.2** on the next page for error plots.

Lowering the maximum error requires increasing the effective precision, and that is most easily done by decreasing the range of g, so that we have $\exp(g) = 1 + \textit{smaller correction}$. Tang [Tan89] published a complicated algorithm that relies on subtle properties of IEEE 754 and VAX binary arithmetic, and of the argument reduction for the exponential, to push the maximum error in `exp()` down to 0.54 ulp in the absence of underflow, and otherwise, down to 0.77 ulp if the result is subnormal. Detailed further analysis [Fer95] supports Tang's work.

With less complexity, more generality, and portability to all platforms supported by the mathcw library, we can do almost as well as Tang's algorithm. Implement a further reduction of g, like this:

$$\begin{aligned} g &= k/p + r, && \textit{for integer } k \textit{ and } p, \\ k &= \operatorname{round}(p \times g), && \textit{find } k \textit{ first,} \\ r &= g - k/p, && \textit{find exact } r \textit{ next,} \\ \exp(g) &= \exp(k/p)\exp(r), && \textit{factor exponential.} \end{aligned}$$

Here, p *must* be a power of the base, so that $p \times g$ and k/p are exact operations. The range of k depends on that of g. With the Cody/Waite reductions for binary arithmetic, k lies in $[-0.347p, +0.347p]$; for decimal arithmetic, k is in $[-0.576p, +0.576p]$. The values $\exp(k/p)$ are constants that are precomputed in high precision, and tabulated as sums of exact high and accurate low parts, so we need about $0.70p$ entries for binary arithmetic, and about $1.15p$ entries for a decimal base. The range of r is now $[-1/(2p), +1/(2p)]$. The maximal corrections c in $\exp(r) = 1 + c$ are small, and provide another three to six decimal digits of effective precision:

p	c	p	c
10	0.005 012	16	0.001 955
100	0.000 050	256	0.000 008

Tang's algorithm involves a similar representation with $p = 2^5 = 32$.

The final result is then reconstructed like this:

$$\exp(x) = \begin{cases} 2^n(\exp(k/p)_{\mathrm{hi}} + \exp(k/p)_{\mathrm{lo}})(1+c), & \\ 10^{n/2}(\exp(k/p)_{\mathrm{hi}} + \exp(k/p)_{\mathrm{lo}})(1+c), & \textit{n even,} \\ 10^{(n-1)/2}(\sqrt{10}_{\mathrm{hi}} + \sqrt{10}_{\mathrm{lo}})(\exp(k/p)_{\mathrm{hi}} + \exp(k/p)_{\mathrm{lo}})(1+c), & \textit{n odd.} \end{cases}$$

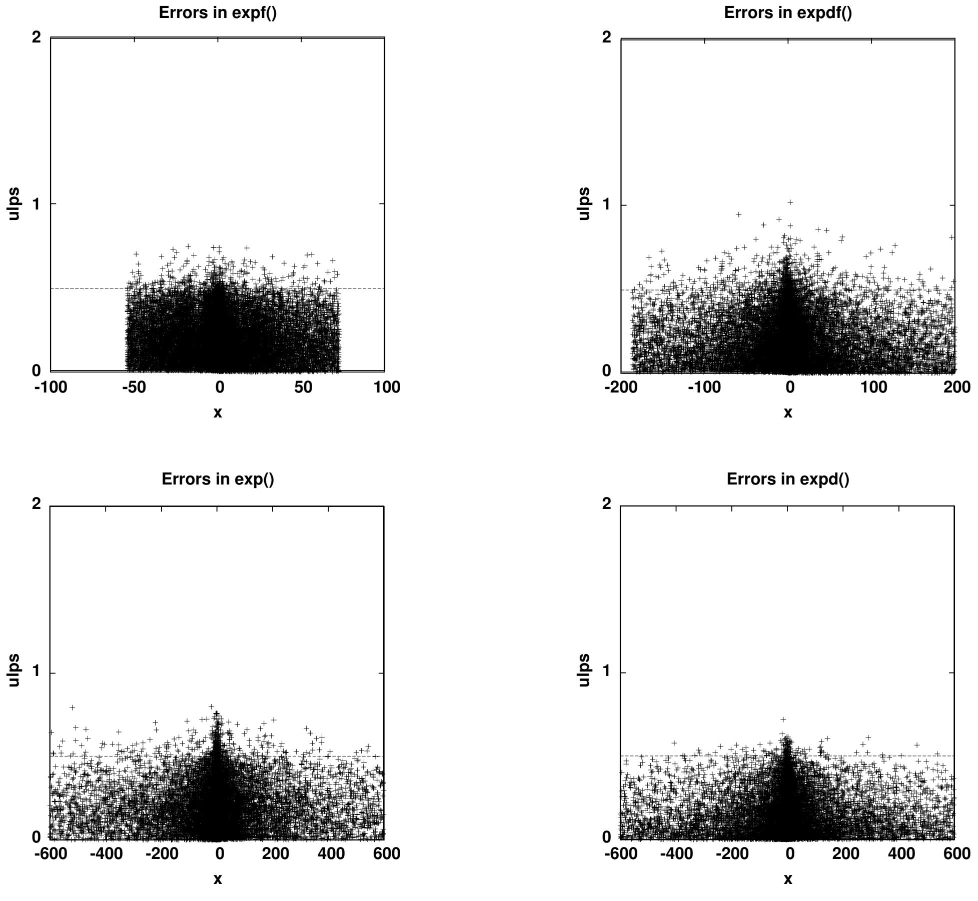


Figure 10.2: Errors in `EXP()` functions with the original Cody/Waite algorithm. The horizontal dotted line at 0.5 ulps marks the boundary below which results are correctly rounded.

The powers of 2 and 10 are supplied by the `ldexp()` family, and the products of sums of high and low parts are expanded and summed in order of increasing magnitude.

The files `exp.h` and `expx.h` contain preprocessor-selectable code for the Tang algorithm, the original Cody/Waite algorithm, and our extended algorithm for the small-table case ($p = 10$ and $p = 16$) and the default large-table case ($p = 100$ and $p = 256$). When the Tang algorithm is requested but is not applicable, the code reverts automatically to the default method. The file `exp.h` contains new polynomial fits for our extended algorithms to take advantage of the shorter argument range: a $\langle 1/2 \rangle$ rational polynomial provides 17 decimal digits in the small-table case, and a $\langle 1/1 \rangle$ fit gives that accuracy for the large tables.

Tang's algorithm includes a two-term Taylor series when x is sufficiently small, and uses his exponential-specific argument reduction scheme. The other three include code for two-term and five-term series for small x, and use the `REDUCE()` family for the reduction $x = n \log(\beta) + g$.

Tests show that the largest error in the small-table case is about 0.53 ulps, with 1/5650 of the results for random arguments differing from exact values rounded to working precision. For the default of large tables, the maximum error drops to 0.509 ulps in binary arithmetic with about 1/40 000 differing from correctly rounded exact results. For decimal arithmetic, the corresponding values are 0.501 ulps and 1/600 000. These results are close to those measured for Tang's algorithm: 0.508 ulps and 1/102 000 for `expf()`, and 0.502 ulps and 1/750 000 for `exp()`.

Figure 10.3 on the facing page shows the measured errors in four of the exponential functions. The improvements over the plots in **Figure 10.2** are evident.

The algorithms for the companion families `EXP2()`, `EXP8()`, `EXP10()`, and `EXP16()` are similar to that for the

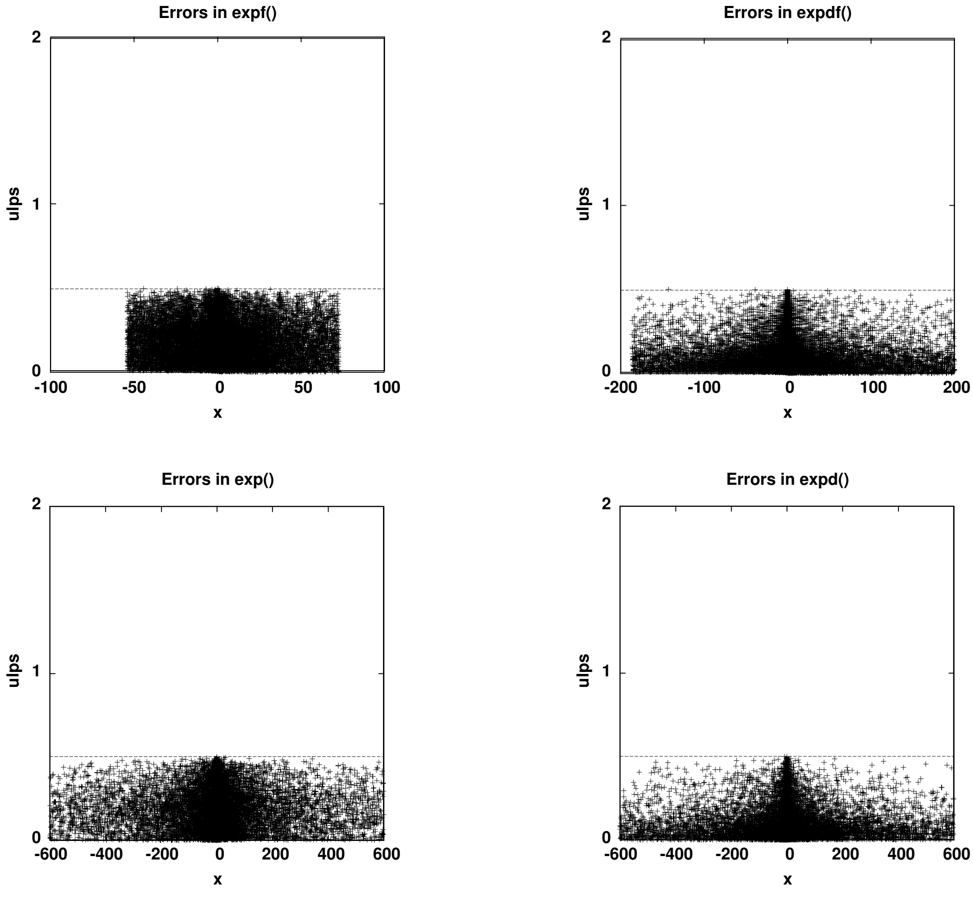


Figure 10.3: Errors in `EXP()` functions with the default large-table algorithm.

ordinary exponential function, although the cutoffs, Taylor series, and polynomial fits differ. Test show that those extra functions produce results that are almost always correctly rounded, so we do not show error plots for them.

10.2 Exponential near zero

Berkeley UNIX 4.3BSD introduced the `expm1()` function in 1987. It is defined as $\operatorname{expm1}(x) = \exp(x) - 1$, and handles the important case of small arguments. When x is small, it is clear from the Taylor series of $\exp(x)$ that subtracting one from it loses accuracy in binary arithmetic in the region $[\log(\frac{1}{2}), \log(\frac{3}{2})] \approx [-0.693, +0.405]$, so a different algorithm is called for.

One reasonable approach is to rewrite the function in terms of another that is likely to be available in standard programming-language libraries. Such a function is the hyperbolic tangent, which we treat later in this book in **Section 12.1** on page 341. We have

$$\begin{aligned}\tanh(x) &= (\exp(x) - \exp(-x))/(\exp(x) + \exp(-x)) \\ &= (1 - \exp(-2x))/(1 + \exp(-2x)).\end{aligned}$$

Solve the second equation for the exponential function, scale its argument by $-\frac{1}{2}$, and simplify using the symmetry relation $\tanh(-x) = -\tanh(x)$:

$$\exp(-2x) = (1 - \tanh(x))/(1 + \tanh(x)),$$

$$\exp(x) = (1 + \tanh(\tfrac{1}{2}x))/(1 - \tanh(\tfrac{1}{2}x)).$$

The exponential less one is then given by

$$\begin{aligned}\exp(x) - 1 &= (1 + \tanh(\tfrac{1}{2}x) - (1 - \tanh(\tfrac{1}{2}x)))/(1 - \tanh(\tfrac{1}{2}x)),\\ &= 2\tanh(\tfrac{1}{2}x)/(1 - \tanh(\tfrac{1}{2}x)).\end{aligned}$$

The subtraction in the denominator suffers leading digit loss only for x in $[1.098, \infty]$, well away from the region where direct computation of $\exp(x) - 1$ loses digits. If the hyperbolic tangent function is accurate, and not itself implemented with $\mathrm{expm1}(x)$, as is the case with the GNU glibc and Sun Microsystems fdlibm libraries, then the last formula is a quick way to program the computation of $\mathrm{expm1}(x)$ for x in $[-1, 1]$, although the computed value has at least three rounding errors.

We can do better with a modification of our algorithm for the exponential function. For sufficiently small x, we can sum a few terms of the Taylor series. Otherwise, we absorb the exponential growth into an exact power of the base, multiplied by a small correction factor, and write

$$\begin{aligned}\mathrm{expm1}(x) &= \beta^n \exp(g) - 1\\ &= \beta^n(\exp(g) - 1) + (\beta^n - 1),\\ x &= n\log(\beta) + g, && g \textit{ in } [-\tfrac{1}{2}\log(\beta), \tfrac{1}{2}\log(\beta)],\\ \exp(g) - 1 &= g + g^2/2! + g^3/3! + g^4/4! + \cdots, && \textit{Taylor series},\\ &\approx g + g^2/2 + g^3\mathcal{R}(g), && \textit{Taylor sum},\\ \mathcal{R}(g) &\approx (\exp(g) - 1 - g - g^2/2)/g^3, && \textit{fit to rational polynomial}.\end{aligned}$$

The term $\beta^n - 1$ is exactly representable for small n, but is negative for $n < 0$, hiding a subtraction in the final result. The term $\exp(g) - 1$ is given either as a Taylor series, or as the sum of three terms of decreasing magnitude, the first of which is exact, or nearly so, if we use the `REDUCE()` family for accurate argument reduction.

We could do a further reduction, $g = k/p + r$, as we did with the exponential functions of the preceding section, but we can do almost as well without it, provided that we evaluate the term sums accurately.

The code in `expm1x.h` starts with a one-time initialization block that computes cutoffs needed for later tests:

```
if (do_init)
{
    BIGX = LOG(FP_T_MAX);
    SMALLX = LOG(FP_T_MIN * FOURTH);
    XCUT_4 = SQRT(SQRT(FP(60.0) * FP_T_EPSILON));
    do_init = 0;
}
```

A few initial tests handle special cases and a short sum of the Taylor series:

```
if (ISNAN(x))
    result = SET_EDOM(x);
else if (x == ZERO)         /* only x for which expm1(x) is exact */
    result = x;             /* preserve sign of zero */
else if (x > BIGX)          /* set overflow flag only for finite x */
    result = ISINF(x) ? SET_ERANGE(x) : SET_ERANGE(INFTY());
else if (x < SMALLX)        /* exp(x) below negative machine epsilon */
{
    STORE(&tiny);
    result = tiny - ONE;    /* force inexact flag to be set */
}
else if (QABS(x) < XCUT_4)
{
    fp_t sum;

    sum = (FP(1.0) / FP(120.0)      ) * x;
    sum = (FP(1.0) / FP(24.0)  + sum) * x;
    sum = (FP(1.0) / FP(6.0)   + sum) * x;
    sum = (FP(1.0) / FP(2.0)   + sum) * x;
    result = x + x * sum;   /* FMA opportunity */
}
```

For sufficiently negative x, $\exp(x)$ is negligible compared to 1, but in order to handle rounding modes properly, and set the *inexact* flag, we force run-time evaluation of the expression `tiny - ONE`, where `tiny` is declared `volatile` and initialized to the smallest representable positive normal number.

Many compilers evaluate the rational numbers in the five-term sum in the Horner form of the Taylor series at compile time, eliminating the divisions. The terms are accumulated in order of increasing magnitude, and the final value of `sum` is sufficiently small that the rounding error in the final result is almost entirely from the last addition, and almost always zero.

The final `else` block of the function handles the general case, but tests of an initial implementation suggest a different approach for a decimal base, where the range of g is larger. The decimal code looks like this:

```
  else
  {
#if B == 10
    if ( (-FP(0.75) <= x) && (x <= FP(0.75)) ) /* digit loss region */
        result = expm1ts(x);
    else if ( (FP(2.3) <= x) && (x <= FP(2.4)) )
    {
        /*
        ** There is a slight inaccuracy in EXP(x) for x ~= log(10)
        ** that pushes errors in EXP() - 1 up to about 0.53 ulp.
        ** We can eliminate that error by special handling of this
        ** case with an accurate argument reduction.  We have
        **
        **     exp(x) - 1 = exp(1*log(10) + r) - 1
        **                = exp(log(10)) * exp(r) - 1
        **                = 10 * exp(r) - 1
        **                = 10 * (exp(r) - 1) + 9
        **
        ** The result is therefore 9 + (small correction)
        */

        fp_t r, sum;

        r = x - LOG_10_HI;      /* reduce x = 1 * log(10) + r */
        r -= LOG_10_LO;
        sum = expm1ts(r);
        result = NINE + TEN * sum;
    }
    else
        result = EXP(x) - ONE;
```

The summation of the general Taylor series is required in three places in our implementation, so we move the job to a separate private function, `expm1ts()`, whose code is straightforward, and thus, omitted.

The digit loss region for direct computation of $\exp(x) - 1$ in decimal arithmetic is $[\log(0.9), \log(2)]$, or about $[-0.105, +0.693]$. We widen that interval to $[-\frac{3}{4}, +\frac{3}{4}]$ and sum the general Taylor series. Otherwise, we just compute $\exp(x) - 1$ directly, except for the small region $[2.3, 2.4]$, where we do an additional exact argument reduction to get the result as a small correction to an exact value.

For bases of the form $\beta = 2^K$, the chief sources of error are the three-term summation to find $\exp(g) - 1$, the computation of $\frac{1}{2}g^2$, and the hidden subtraction of the term $\beta^n - 1$ when $n < 0$.

We can account for the errors of addition by a technique introduced later in this book at the start of **Chapter 13** on page 353, and made available in the mathcw library in the `VSUM()` function family, which returns the sum of an array, and an estimate of the error in the sum. We can recover the rounding error in multiplication by exploiting the fused multiply-add operation:

$$\text{error in } \text{fl}(a \times b) = \text{fma}(a, b, -\text{fl}(a \times b)).$$

The code for a nondecimal base in the final `else` block in `expm1x.h` continues like this, and does not invoke the exponential function at all:

```
#else /* B != 10 */
    if ( (FP(0.25) < QABS(x)) && (QABS(x) < FP(0.75)) )
        result = expm1ts(x);
    else
    {
        fp_t e1, e2, e3, err, g, gg, pg, qg, scale, sum, t, v[5];
        int n, nv;

        g = REDUCE(x, INV_C, NC, C, &n, (fp_t *)NULL);

        pg = POLY_P(p, g);
        qg = POLY_Q(q, g);

        gg = g * g;
        e1 = g;
        e2 = HALF * gg;
        e3 = g * gg * (pg / qg);

        /* recover error in e2 and add it to e3 */

        t = e2 + e2;
        err = FMA(g, g, -t);
        e3 += HALF * err;

        /*
        ** Reconstruct expm1(x) accurately, handling a few special
        ** cases of n explicitly to avoid the need to call LDEXP().
        */

        switch (n)  /* form sum = 2**n * (e1 + e2 + e3) + (2**n - 1) */
        {                   /* let s = e1 + e2 + e3 */
        case 0:             /* sum = s */
            nv = 3;
            v[2] = e1;
            v[1] = e2;
            v[0] = e3;
            break;

        case -1:            /* sum = s/2 - 1/2 */
            nv = 4;
            v[3] = -HALF;
            v[2] = HALF * e1;
            v[1] = HALF * e2;
            v[0] = HALF * e3;
            break;

        case 1:             /* sum = 2*s + 1 */
            nv = 4;
            v[3] = ONE;
            v[2] = e1 + e1;
            v[1] = e2 + e2;
            v[0] = e3 + e3;
            break;

        default:
            scale = LDEXP(ONE, n);
            nv = 5;

            if (n < 0)      /* move scale down in list */
            {
                v[4] = -ONE;
                v[3] = scale;
                v[2] = scale * e1;
                v[1] = scale * e2;
                v[0] = scale * e3;
            }
            else            /* move -1 down in list */
            {
                v[4] = scale;
                v[3] = scale * e1;
                v[2] = scale * e2;
                v[1] = scale * e3;
                v[0] = -ONE;
            }

            break;
        }

        sum = VSUM(&err, nv, v);
        sum += err;

        result = sum;
    }
#endif /* B == 10 */
}
```

In the region $[\frac{1}{4}, \frac{3}{4}]$, we use the Taylor series to avoid loss from the hidden subtraction in $\beta^n - 1$. Otherwise, we construct an array of terms of increasing magnitudes and pass it to the vector-sum routine. The corrections from the fused multiply-add and from the error estimate returned by VSUM() are small, but generally keep the total error under $\frac{1}{2}$ ulp. Without them, the worst-case errors rise by about 0.3 ulps.

Figure 10.4 on the next page shows the measured errors in four members of the EXPM1() family. The occasional errors above $\frac{1}{2}$ ulp in expm1(x) in the interval $[-1, 3]$ could be removed by adding code to make the additional argument reduction $g = k/p + r$, but we leave that improvement for future work.

The mathcw library also includes companion function families EXP2M1(), EXP8M1(), EXP10M1(), and EXP16M1() to compute $b^x - 1$, with $b = 2, 8, 10$, and 16. We can readily find the Taylor series with the help of Maple, and rewrite it in nested Horner form for efficient numerical evaluation:

```
% maple
> taylor(b^(z/log(b)) - 1, z = 0, 8);
         2        3          4           5           6            7     8
z + 1/2 z  + 1/6 z  + 1/24 z  + 1/120 z  + 1/720 z  + 1/5040 z + O(z )

> convert(convert(taylor(b^(z/log(b)) - 1, z=0, 8), polynom), horner);
/    /      /      /       /        /         z  \  \  \  \  \  \
|1 + |1/2 + |1/6 + |1/24 + |1/120 + |1/720 + ----| z| z| z| z| z| z
\    \      \      \       \        \        5040/  /  /  /  /  /
```

The intermediate variable $z = x \log(b)$ simplifies the series. The leading term introduces two rounding errors from the multiplication and the truncation of the transcendental number constant, $\log(b)$. When z is negative, there can also be subtraction loss in each pair of terms in the series.

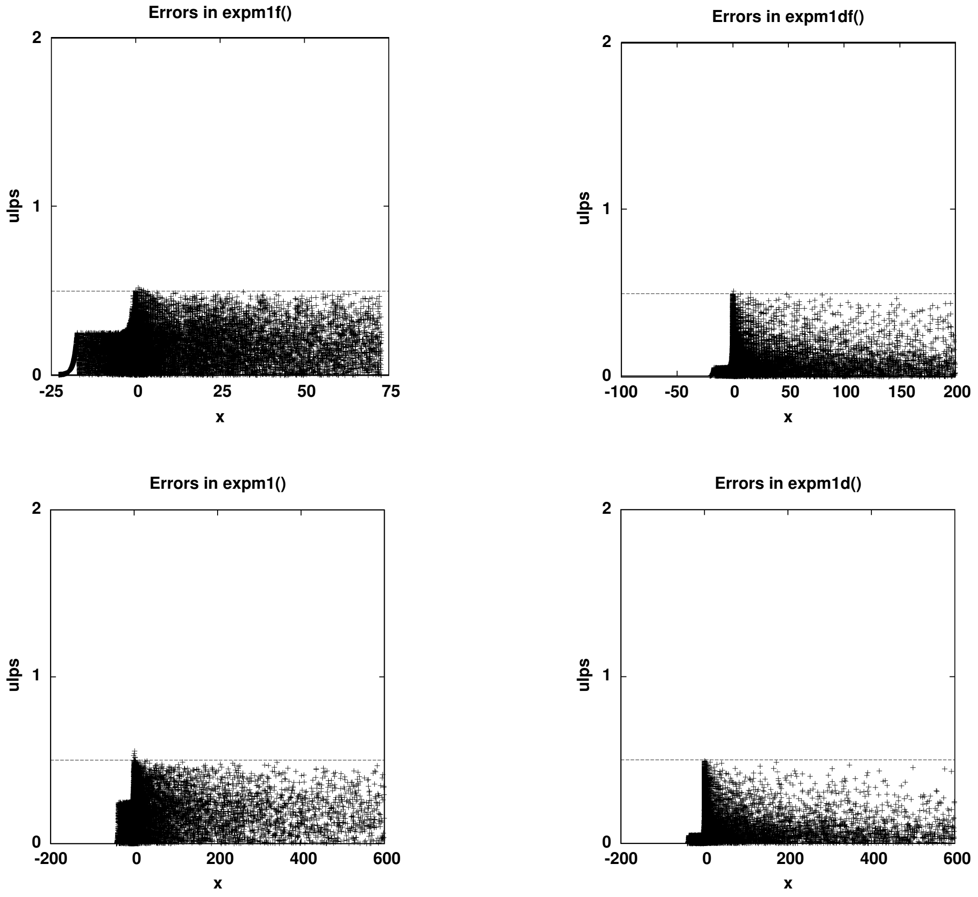


Figure 10.4: Errors in EXPM1() functions. The horizontal dotted line at 0.5 ulps marks the boundary below which results are correctly rounded.

For arguments near the underflow limit, we have $b^x - 1 \approx z$, but if we compute $z = x \times (\log(b)_{\rm hi} + \log(b)_{\rm lo})$, the product with the low part may be subnormal, or underflow to zero. The solution is an exact scaling of x away from the underflow region so that we can compute the products accurately, and then undo the scaling. A suitable scale factor is the cube of the machine epsilon, ϵ^3, and a short code block implements the computation:

```
else if (QABS(x) < XCUT_1)
{
    fp_t t1;

    t1 = SCALE * x;         /* EXACT */
    result = FMA(t1, LN_10_HI, t1 * LN_10_LO) * SCALE_INV;
}
```

The limit XCUT_1 is initialized to SCALE * FP_T_MIN. Instead of using a fused multiply-add operation to compute an accurate product, we could instead split t1 into high and low parts, and add the products in order of increasing magnitude, as we do later in the code. However, this block is expected to be entered rarely, and using FMA() simplifies the programming.

We can eliminate the subtraction loss in the series summation by making z smaller, using two argument reduction steps to the new variables g and r:

$$x = \begin{cases} n\log_2(2) + g, & \textit{when } \beta = 2^K \textit{ and } b = 2^M, \\ n\log_b(2) + g, & \textit{when } \beta = 2^K \textit{ and } b = 10, \\ n\log_b(10) + g, & \textit{when } \beta = 10, \end{cases}$$
$$g = k/p + r, \qquad \textit{where } p = \beta^N.$$

When $b = 2, 8$, or 16, and $\beta = 2^K$, or when $b = \beta = 10$, the reductions simplify to $x = n + g$, where $n = \operatorname{round}(x)$, and $g = x - n$ is computed exactly, and lies in $[-\frac{1}{2}, \frac{1}{2}]$.

Otherwise, we provide an accurate tabular representation of the logarithm constant and hand the reduction job off to the `REDUCE()` family. In that case, the range of g is $[-\frac{1}{2}\log_b(2), +\frac{1}{2}\log_b(2)]$ for $\beta = 2^K$, and $[-\frac{1}{2}\log_b(10), +\frac{1}{2}\log_b(10)]$ for a decimal base. The largest g is then $\frac{1}{2}\log_2(10) \approx 1.66$.

The second reduction variable, r, lies in $[-1/(2p), +1/(2p)]$, and can be made small by choosing p to be large.

We then have for the nondecimal case

$$\begin{aligned} b^x - 1 &= b^{m\log_b(2)+g} - 1 \\ &= 2^m b^g - 1 \\ &= 2^m(b^g - 1) + (2^m - 1). \end{aligned}$$

The decimal case is similar:

$$\begin{aligned} b^x - 1 &= b^{m\log_b(10)+g} - 1 \\ &= 10^m b^g - 1 \\ &= 10^m(b^g - 1) + (10^m - 1). \end{aligned}$$

Although subtractions are still present, the terms $2^m - 1$ and $10^m - 1$ can usually be computed exactly, or with at most one rounding error. We compute the reduced exponential, less one, as

$$\begin{aligned} b^g - 1 &= b^{k/p+r} - 1 \\ &= b^{k/p} b^r - 1 \\ &= b^{k/p}(b^r - 1) + (b^{k/p} - 1). \end{aligned}$$

The parenthesized expressions can have opposite signs if k is determined from $\operatorname{round}(p \times g)$, instead of from $\operatorname{trunc}(p \times g)$. With the latter choice, the range of r doubles, but a subtraction is avoided. Numerical tests with both variants show that errors are reduced slightly by using rounding instead of truncation.

The factor $b^r - 1$ is found either from its Taylor series, or from a polynomial approximation. For the latter, we use a rational polynomial, $\mathcal{R}(r)$, for which

$$\begin{aligned} z &= r\log(b), \\ b^r - 1 &\approx z + \tfrac{1}{2}z^2 + z^3\mathcal{R}(r), \\ \mathcal{R}(r) &\approx (b^r - 1 - z - \tfrac{1}{2}z^2)/z^3. \end{aligned}$$

Because r is small, the polynomial approximation provides a small correction to the first two terms, and there is no subtraction loss when r is negative.

The constant $\log(b)$ is split into the sum of an exact 12-bit (or four-decimal-digit) high part, and a low part accurate to working precision. The value r is split into the sum of an exact 9-bit (or three-decimal-digit) high part, and an exact low part. The factor $b^{k/p}$ is obtained by lookup in a table of exact high and approximate low parts, and $b^{k/p} - 1$ can be evaluated accurately from those parts.

After expanding the product sums, the leading term in the expansion of $b^r - 1$ is exact, and the remaining terms then provide a small correction. As in the `EXPM1()` family, we can get a slight improvement in accuracy by correcting for the errors of multiplication and addition with the help of the `VSUM()` family.

To show how that works, here is the code block with the argument reduction and computation of `exp10m1(x)` from the file `e10m1x.h`:

```
#if B == 10

  t = ROUND(x);
  g = x - t;              /* EXACT decomposition: g in [-1/2, +1/2] */
  n = (int)t;

  t = ROUND(g * FP(1.e1));
  k = (int)t;             /* overflow impossible */
  r = g - t * FP(1.e-1);  /* EXACT: r in [-1/20, +1/20] */

  scale = LDEXP(ONE, n);  /* EXACT: 10**n */
  f = FREXP(r, &m);       /* r = f * 10**n, with f in [1/10,1) */
  r_hi = LDEXP(TRUNC(FP(1.e3) * f) * FP(1.e-3), m);/* 3 upper digits */
  r_lo = r - r_hi;        /* remaining decimal digits of r */

#else /* B != 10 */

  g = REDUCE(x, ONE_OVER_LOG10_2, NC, C, &n, (fp_t *)NULL);

  t = ROUND(FP(16.0) * g);
  k = (int)t;             /* overflow impossible */
  r = g - t * FP(0.0625); /* EXACT product: r in [-1/32,+1/32] */

  scale = LDEXP(ONE, n);  /* EXACT: 2**n */
  f = FREXP(r, &m);       /* r = f * 2**n, with f in [1/2,1) */
  r_hi = LDEXP(TRUNC(FP(512.0) * f) / FP(512.0), m); /* 9 upper bits */
  r_lo = r - r_hi;        /* remaining bits of r */

#endif  /* B == 10 */

  p_hi = P10[k + P10_OFFSET][0];
  p_lo = P10[k + P10_OFFSET][1];

  z_hi  = r_hi * LN_10_HI;        /* EXACT product */
  z_lo  = r_lo * LN_10_LO;
  z_lo += r_hi * LN_10_LO;
  z_lo += r_lo * LN_10_HI;

  z = z_hi + z_lo;        /* best approximation to r * ln(10) */
  zz = z * z;

  pr = POLY_P(p, r);      /* exp10m1(r) = z + z**2/2 + z**3 * R(r) */
  qr = POLY_Q(q, r);      /* where z = r * ln(10) */
  pq = pr / qr;

  t3 = z * zz * pq;
  t2 = HALF * zz;
  t_hi = z_hi;
  t_lo = z_lo + t3;
  t_lo += t2;

  if (n == 0)
  {
      if (p_hi == ONE)
          sum = t_hi + t_lo;
      else
      {
          v[6] = -ONE;
          v[5] = p_hi;
          v[4] = p_hi * t_hi;
          v[3] = p_hi * t_lo;
          v[2] = p_lo;
          v[1] = p_lo * t_hi;
          v[0] = p_lo * t_lo;
          sum = VSUM(&err, 7, v);
          sum += err;
      }
  }
  else if (n > 0)
  {
      v[6] = p_hi;
      v[5] = -ONE / scale;        /* EXACT */
      v[4] = p_hi * t_hi;
      v[3] = p_hi * t_lo;
      v[2] = p_lo;
      v[1] = p_lo * t_hi;
      v[0] = p_lo * t_lo;
      sum = VSUM(&err, 7, v);
      sum += err;
      sum *= scale;               /* EXACT */
  }
  else /* n < 0 */
  {
      v[6] = -ONE / scale;        /* EXACT */
      v[5] = p_hi;
      v[4] = p_hi * t_hi;
      v[3] = p_hi * t_lo;
      v[2] = p_lo;
      v[1] = p_lo * t_hi;
      v[0] = p_lo * t_lo;
      sum = VSUM(&err, 7, v);
      sum += err;
      sum *= scale;               /* EXACT */
  }

  result = sum;
```

Here, we chose $p = 16$ for the nondecimal cases, and $p = 10$ for a decimal base.

The corresponding code for `exp8m1(x)` and `exp16m1(x)` is similar, except that `REDUCE()` is needed when $\beta = 10$, and `ROUND()` when $\beta = 2^K$.

For `exp2m1(x)`, our code omits the second reduction to r, and for a nondecimal base uses `VSUM()` to add the various contributions to $2^g - 1$, with code similar to that of `exp10m1(x)`. For a decimal base, the code sums the Taylor series for x in $[-0.155, +1]$, the approximate region where there is loss of leading decimal digits in the subtraction $2^x - 1$. For r in $[3.30, 3.42]$, the code uses the reduction $x = \log_2(10) + g$ to construct

$$\begin{aligned} 2^x - 1 &= 2^{\log_2(10)+g} - 1 \\ &= 10 \times 2^g - 1 \\ &= 10 \times (2^g - 1) + 9. \end{aligned}$$

It computes $2^g - 1$ from the Taylor series, multiplies that small value by 10 (an *exact* operation), and adds it to the larger value 9. Outside those two regions, the code for a decimal base just returns $\exp2(x) - 1$.

The results for the computation of $b^x - 1$ with our functions `exp2m1()`, `exp8m1()`, `exp10m1()`, and `exp16m1()` are almost-always correctly rounded, so we omit error plots for them.

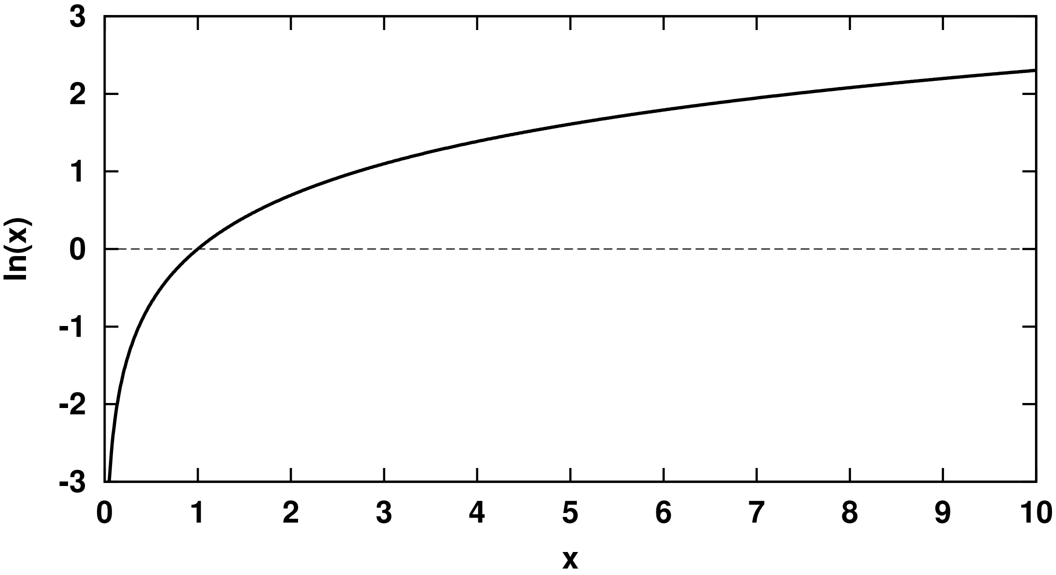


Figure 10.5: The natural logarithm function, $\ln(x)$. The dashed line for $\ln(x) = 0$ intersects the curve at $x = 1$. The function grows slowly for increasing x, with $\ln(1000) \approx 6.908$, $\ln(1\,000\,000) \approx 13.816$, $\ln(1\,000\,000\,000) \approx 20.724$, and $\ln(10^{100}) \approx 230.259$.

10.3 Logarithm functions

The Scottish mathematician, John Napier (1550–1617), is generally credited with the introduction of the logarithm into mathematics in 1614, although his definition was rather different from the modern one. For an arbitrary base β, we now define the logarithm in that base by

$$\begin{aligned} x &= y^p, && \text{where } x > 0 \text{ and } y > 0, \\ \log_\beta(x) &= p \log_\beta(y), && \beta > 1. \end{aligned}$$

That is, the logarithm is the inverse of a power function, scaled by $\log_\beta(y)$. In real arithmetic, the logarithm is *undefined* for negative arguments or bases.

When $y = \beta$, the scale factor disappears, and we have

$$\begin{aligned} x &= \beta^p, \\ \log_\beta(x) &= p. \end{aligned}$$

Logarithms are usually introduced in middle school, often only for the case $\beta = 10$, and the subscript that indicates the base is usually omitted. However, in higher mathematics, an omitted subscript on the logarithm generally means the default base of Euler's number, $e \approx 2.718\,281\,828\ldots$. The notation $\ln(x)$, for *natural* or *Napierian* logarithm, is a common synonym for $\log_e(x)$, and that function is graphed in Figure 10.5.

The chief importance of logarithms for manual computation is that they convert the hard problems of multiplication and division into the easier ones of addition and subtraction, because of these relations:

$$\begin{aligned} \log_\beta(x \times y) &= \log_\beta(x) + \log_\beta(y), \\ \log_\beta(x/y) &= \log_\beta(x) - \log_\beta(y). \end{aligned}$$

For hundreds of years, tables of logarithms and their inverses, called *antilogarithms*, were common in computation. Locate the logs of x and y in the logarithm table, manually add or subtract them, and then look up that result in the antilogarithm table to find the product or quotient of x and y.

Logarithms make it possible to represent extreme magnitudes with ease: the large number 10^{100} is called a *googol*, but its base-10 logarithm is only 100. The much larger number, the *googolplex*, $10^{\text{googol}} = 10^{10^{100}}$, would require more digits to write in full than there are particles in the universe,[2] but the logarithm of its logarithm, both in base 10, is still only 100.

Mechanical calculators capable of multiplication and division were first constructed late in the Seventeenth Century, but were not commonly available until after 1820, when the first mass production of calculators began. Nevertheless, they were bulky and expensive, and seldom available to students. This author would prefer to forget the distasteful drudgery of far too many study hours wasted with logarithm tables for lack of such a device, but that work produced his enthusiasm for programming digital computers.

The first commercial application of the first successful integrated circuit, the 4-bit Intel 4004 in 1971, was for an electronic calculator. Advances in electronics and large-volume manufacturing made such calculators affordable to many by the 1980s. At the time of writing this, advanced programmable calculators with many built-in functions and graphing facilities cost less than a textbook. As a result, students the world over are less familiar with logarithms than their parents were.

A little manipulation of the rules for logarithms shows that we can relate their values in two different bases α (Greek letter *alpha*) and β like this:

$$\log_\alpha(x) = \log_\beta(x) \times \log_\alpha(\beta).$$

For fixed α and β, the multiplier $\log_\alpha(\beta)$ is a constant that we can precompute, so we can find the logarithm in any base from one in a particular base, usually $\beta = e$, at the expense of one multiply, and one extra rounding error. In octal or hexadecimal arithmetic, because of wobbling precision, it may happen that $\log_\alpha(\beta)$ has leading zero bits, reducing its precision, whereas $\log_\beta(\alpha)$ has no leading zero bits. It is then more accurate to divide by the latter than to multiply by the former.

The relation between the logarithms in two different bases can be rewritten in two useful forms:

$$\begin{aligned}\log_\beta(x) &= \ln(x)/\ln(\beta),\\ \log_\alpha(\beta) \times \log_\beta(\alpha) &= 1.\end{aligned}$$

For the purposes of the `mathcw` library, we are interested in logarithms for $\beta = 2, e, 8, 10$, and 16, and the corresponding functions are called `log2(x)`, `log(x)`, `log8(x)`, `log10(x)`, and `log16(x)`. In computer-science applications, the base-2 logarithm is so common that the notation $\lg(x)$ has been widely adopted for it, although that name is absent from most programming languages, including the C-language family.

The standard representation of floating-point numbers as a fraction and an integer power of a base simplifies the logarithm considerably, because the representation provides an exact range reduction:

$$\begin{aligned}x &= (-1)^s \times f \times \beta^n, && \textit{either } f = 0 \textit{ exactly, or } f \textit{ is in } [1/\beta, 1),\\ \log_\alpha(|x|) &= \log_\alpha(f) + n\log_\alpha(\beta).\end{aligned}$$

Our primary concern is then accurate computation of $\log_\alpha(f)$, combined with an accurate adjustment of the result by $n\log_\alpha(\beta)$.

The error-magnification formula from **Table 4.1** on page 62,

$$\operatorname{errmag}(\log(x)) = 1/\log(x),$$

shows that the computation is only sensitive to argument errors for arguments near one, which is the only zero of the logarithm on the real axis.

We have these special limits and values for the logarithm:

$$\begin{aligned}\lim_{x\to+0} \log_\alpha(x) &\to -\infty,\\ \log_\alpha(1) &= 0,\\ \log_\alpha(\alpha) &= 1,\\ \lim_{x\to+\infty} \log_\alpha(x) &\to +\infty.\end{aligned}$$

[2] That number is estimated to be about 10^{80}, within a few powers of ten.

For extreme arguments, the approach to infinity is slow, and the logarithms of the largest and smallest representable floating-point numbers are of modest size. For the IEEE 754 64-bit format, the logarithm (to the base e) of the largest number is just below 709.79. The logarithm of the smallest normal number is just above -708.40, and that of the smallest subnormal number is about -744.44.

We should therefore expect any satisfactory implementation of the logarithm to be highly accurate, and to return finite values for any finite argument larger than zero. In IEEE 754 arithmetic, we should expect that `log(x)` returns a NaN when $x < 0$ or when x is itself a NaN, returns $-\infty$ when $x = 0$, and returns $+\infty$ only when $x = +\infty$.

10.3.1 Computing logarithms in a binary base

Cody and Waite take two approaches for the logarithm, depending on the floating-point base. When the base is a power of two, they choose $f < 1$, like this:

$$\begin{aligned}
\beta &= 2^K, && \textit{K = 1, 3, or 4 in practice},\\
x &= f \times \beta^N, && \textit{assume nonzero } f \textit{ in } [1/\beta, 1),\\
\log(x) &= \log(f) + N\log(\beta),\\
&= \log(f) + KN\log(2).
\end{aligned}$$

The exponent and fraction can be readily extracted with the functions `INTXP()` and `SETXP()` introduced by Cody and Waite, or with the traditional `FREXP()` family, or with the C99 `LOGB()` and `SCALBN()` functions. All of them produce *exact* results, not by numerical computation, but by decoding the bit representation of the native floating-point representation.

Next, when $\beta > 2$, Cody and Waite adjust f into the interval $[\frac{1}{2}, 1)$ with a code fragment like this:

```
n = K * N; while (f < 0.5) { f += f; n--; }
```

Although their doubling of f appears to be *inexact* if $\beta > 2$, that is *not* the case. When $\beta > 2$ and $f < \frac{1}{2}$, f has leading zero bits. The doubling then merely shifts nonzero bits into those positions, and zero bits into the low end of the fraction.

We then have

$$\log(x) = \log(f) + n\log(2), \qquad \textit{where } f \textit{ is exact, and in } [\tfrac{1}{2}, 1).$$

Now introduce a change of variable, and a rapidly convergent Taylor-series expansion of the logarithm in that new variable:

$$\begin{aligned}
z &= 2(f-1)/(f+1),\\
f &= (2+z)/(2-z),\\
\log(f) &= z + z^3/12 + z^5/80 + z^7/448 + z^9/2304 + z^{11}/11264 + \cdots .
\end{aligned}$$

The expansion suggests using a polynomial approximation of the form

$$\log(f) \approx z + z^3\mathcal{P}(z^2).$$

Here, $\mathcal{P}(z^2)$ could be a fit to a minimax rational polynomial, or to a Chebyshev polynomial expansion. Cody and Waite provide minimax approximations, and for the mathcw library, we extend them for more, and higher, floating-point precisions.

The range of z is $[-\frac{2}{3}, 0)$ for f in $[\frac{1}{2}, 1)$, but we can reduce the magnitude of z, and shorten the fitting polynomial, by one final upward adjustment in f. If $f \le \sqrt{\frac{1}{2}}$, *implicitly* double f and reduce n by one. For $\beta > 2$, if that were done explicitly, it would introduce unwanted rounding error into f; indeed, for $\beta = 16$, three nonzero low-order bits are lost. We show shortly how to accomplish the doubling of f without that loss.

Now we have f in $(\sqrt{\frac{1}{2}}, \sqrt{2}]$, with z in roughly $[-0.3432, +0.3432]$. Over that range of f, the term $|z^3\mathcal{P}(z^2)|$ never exceeds $|0.00999z|$, so it effectively extends the precision by about two decimal digits, and a rounding error in the polynomial evaluation is about a hundred times less likely to affect the computed logarithm.

The main contribution to $\log(f)$ is z, and care is needed in the evaluation of that value. Cody and Waite recommend that it be done like this:

$f > \sqrt{\frac{1}{2}}$: No implicit doubling of f is needed. Set $z_{\text{num}} = (f - \frac{1}{2}) - \frac{1}{2}$ and $z_{\text{den}} = \frac{1}{2}f + \frac{1}{2}$. In the numerator, the first subtraction, $f - \frac{1}{2}$, is *exact* because the two values have the same exponent. The second subtraction of $\frac{1}{2}$ may introduce a rounding error if there is no guard digit for subtraction, but otherwise, is exact. All modern systems have the needed guard digit. The denominator requires one more bit than we have available, so about half the time, it must be rounded. Record an estimate of the error in z_{den} as $\text{fl}(\text{fl}(\frac{1}{2} - z_{\text{den}}) + \frac{1}{2}f)$.

$f \leq \sqrt{\frac{1}{2}}$: Incorporate the implicit doubling of f by first reducing n by one, and then rewriting z like this:

$$\begin{aligned}
z(2f) &= 2 \times \frac{2f-1}{2f+1} \\
&= 4 \times \frac{f - \frac{1}{2}}{2f+1} \\
&= \frac{f - \frac{1}{2}}{\frac{1}{2}f + \frac{1}{4}} \\
&= \frac{f - \frac{1}{2}}{\frac{1}{2}(f - \frac{1}{2}) + \frac{1}{2}}.
\end{aligned}$$

Record these two values for later use:

$$\begin{aligned}
z_{\text{num}} &= f - \tfrac{1}{2}, \\
z_{\text{den}} &= \tfrac{1}{2}z_{\text{num}} + \tfrac{1}{2}.
\end{aligned}$$

Compute the denominator error estimate as $\text{fl}(\text{fl}(\frac{1}{2} - z_{\text{den}}) + \frac{1}{2}z_{\text{num}})$. We do not require f again, so there is no need to double it explicitly.

All divisions by two are, of course, replaced by multiplications by a half. Compute the logarithm of the reduced argument like this:

$$\begin{aligned}
z &= z_{\text{num}}/z_{\text{den}}, \\
g &= z^2, \\
\log(f) &\approx z + (z \times g \times \mathcal{P}(g)).
\end{aligned}$$

Obey the parentheses, and avoid factoring out z.

When $f \approx 1$, we can find its logarithm more quickly by summing a few terms of the Taylor series:

$$\begin{aligned}
f &= 1 + d, \qquad \text{where } |d| \ll 1, \\
\log(f) &= d - d^2/2 + d^3/3 - d^4/4 + d^5/5 - d^6/6 + \cdots .
\end{aligned}$$

The computation of d as $(f - \frac{1}{2}) - \frac{1}{2}$ is exact when guard digits are available. As usual, the truncated Taylor series is rewritten in nested Horner form, so that terms are summed from smallest to largest. With a six-term series, the largest acceptable value of $|d|$ in the IEEE 754 32-bit format is about 0.023, so the second term subtracts about 1% of the value of d, and there is no possibility of loss of leading bits when $d > 0$.

Error plots from an implementation of that algorithm have a broad peak around $x \approx 1$, reaching up to about 1.5 ulps. The problem is that in the polynomial region, the leading term, z, has two rounding errors. Although the numerator $f - 1$ can be computed exactly, the denominator $f + 1$ generally requires one more digit than is available, and thus must be rounded. The division introduces a second rounding error.

Improving the Cody/Waite recipe means eliminating the error in z. To do so, split z into the sum $z_1 + z_2$, where z_1 is larger than z_2, z_1 is exact, and z_2 effectively extends the precision of z. In addition, it is helpful if z_1 uses no more than half of the available bits, so that it can be doubled and squared exactly. With that decomposition, and a new value r, we can obtain the core logarithm with these steps:

$$\begin{aligned} g &= z^2 \\ &= (z_1 + z_2)^2 \\ &= z_1^2 + (2z_1z_2 + z_2^2), && z_1^2 \textit{ is exact,} \\ r &\approx g \times \mathcal{P}(g), && |r| < 0.01, \\ \log(f) &\approx z_1 + z_2 + (z_1 + z_2) \times r \\ &\approx z_1 + (z_1 \times r + (z_2 + z_2 \times r)). \end{aligned}$$

The parentheses indicate the preferred evaluation order. Because the polynomial term contributes less than 1% of z to the sum, we expand only one of its z factors.

The major sources of error are now in the additions, and in the multiplications needed for the products. We can account for the errors of addition with the VSUM() family that we used earlier in this chapter, and recover the errors of multiplication with the fused multiply-add operation. The sum of error terms then provides a small correction to the array sum, producing an improved value of $\log(f)$.

The final reconstruction of the complete logarithm requires adding $n\log(2)$. We can do that accurately by splitting the constant into the sum of exact high and approximate low parts:

$$\begin{aligned} \log(2) &= 0.693\,147\,180\,559\,945\,309\,417\,232\,121\,458\,176\,568\,075\,\ldots \\ &\approx C_{\mathrm{hi}} + C_{\mathrm{lo}}. \end{aligned}$$

The two parts of the constant are

$$\begin{aligned} C_{\mathrm{hi}} &= 355/512, && \textit{exact rational form requires just 9 bits,} \\ &= \texttt{0b1.0110\_0011p-1}, && \textit{binary form,} \\ &= \texttt{0o1.306p-1}, && \textit{octal form,} \\ &= \texttt{0x1.63p-1}, && \textit{hexadecimal form,} \\ &= 0.693\,359\,375, && \textit{exact decimal form,} \end{aligned}$$

and

$$\begin{aligned} C_{\mathrm{lo}} &= \log(2) - C_{\mathrm{hi}} \\ &\approx \texttt{-0x1.bd01\_05c6\_10ca\_86c3\_898c\_ff81\_a12a\_17e1\_979b...p-13} \\ &\approx -0.000\,212\,194\,440\,054\,690\,582\,767\,878\,541\,823\,431\,924\,499\,\ldots\,. \end{aligned}$$

Because the number of bits required for n is limited by the exponent size, the product nC_{hi} can be represented exactly, and often has trailing zero bits. We therefore obtain the final logarithm as

$$\log(x) = n \times C_{\mathrm{hi}} + (n \times C_{\mathrm{lo}} + \log(f)),$$

where the parentheses must be obeyed, and there are two opportunities for fused multiply-add operations.

Here is the code fragment from logx.h that implements the computation of $\log(x)$ in the region where the polynomial approximation is used, with error corrections to the sum:

```
z = z_num / (z_den + err_den); /* used only to compute z1 */

z1 = TRUNC(FP(4096.0) * FREXP(z, &m));
z1 = LDEXP(z1, m - 12);

z2 = (FMA(-z1, z_den, z_num) - z1 * err_den) / (z_den + err_den);

g = z2 * z2;
g += ((z1 + z1) * z2);
g += z1 * z1;              /* g = (z1 + z2)**2 */
```

```
pg = POLY_P(p, g);
qg = POLY_Q(q, g);
r = g * (pg / qg);          /* r = z**2 * P(z**2) / Q(z**2) */

/* form the expansion of (z1 + z2) + (z1 + z2) * r in t[] */

t[3] = z1;
t[2] = z1 * r;
t[1] = z2;
t[0] = z2 * r;

if (n == 0)
    sum = VSUM(&err, 4, t);
else
{
    fp_t xn;

    xn = (fp_t)n;
    t[5] = LN_2_HI * xn;    /* EXACT product */
    t[4] = LN_2_LO * xn;
    sum = VSUM(&err, 6, t);
    err += FMA(LN_2_LO, xn, -t[4]);
}

err += FMA(z1, r, -t[2]);
result = sum + err;
```

The terms in the array t[] are carefully arranged in order of increasing magnitude for optimal summation. The factor $4096 = 2^{12} = 4^6 = 8^4 = 16^3$ in the computation of z_1 ensures exact scaling for $\beta = 2, 4, 8$, and 16, and the fused multiply-add operation ensures that z_2 is accurate.

Although the constant splits *could* be adjusted for each floating-point format, in practice, it suffices to set them according to the worst case: the IBM System/360 hexadecimal format, with 7 exponent bits, and as few as 21 significand bits. The high parts of constants must therefore contain no more than 9 bits if they are needed in products with the 12-bit z_1, and at most 14 bits if they are to be multiplied by an exponent.

Testing of the library code showed that, although the Cody/Waite polynomials are sufficiently accurate for their original algorithm, they are deficient for our improved one. Fortunately, that problem is readily solved by reducing the digit limits in the preprocessor conditionals that select the polynomials in the file log.h, forcing use of a fit of higher degree.

For the base-10 logarithm in a binary base, instead of following the Cody/Waite recipe of computing $\log_{10}(x)$ from $\log(x) \times (1/\log(10))$, we eliminate the additional error of that scaling by computing the logarithm directly by the algorithm described in the next section, using new polynomial fits, and differing reductions to f and z.

The improvement over the original Cody/Waite recipe is dramatic, as illustrated in **Figure 10.6** on the following page. Our computed logarithms are almost always correctly rounded. To quantify that, a test with ten million arguments randomly selected from a logarithmic distribution over $[0.25, 1.75]$ in IEEE 754 arithmetic found only 13 arguments each of logf() and log() with errors above $\frac{1}{2}$ ulp. The largest error was 0.502 ulps. For the base-10 logarithm, 17 arguments of log10f() and 31 of log10() gave errors above $\frac{1}{2}$ ulp, the largest of which was 0.505 ulps. We therefore expect incorrect rounding in at most three of a million random arguments.

10.3.2 Computing logarithms in a decimal base

For a decimal base, the base-10 logarithm is the natural choice, and the decomposition of the argument into an exponent and a fraction gives us a decimal representation:

$$x = (-1)^s \times f \times 10^n, \qquad \textit{either } f = 0 \textit{ exactly, or } f \textit{ is in } [1/10, 1).$$

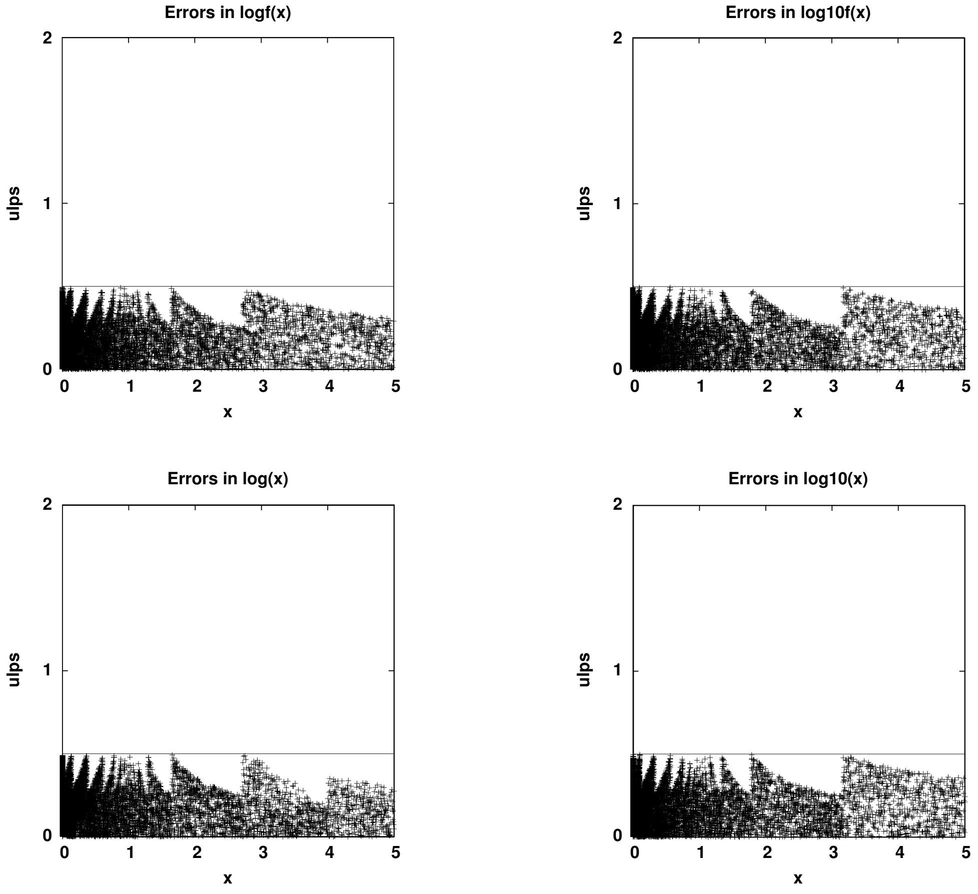


Figure 10.6: Errors in mathcw `LOG()` and `LOG10()` functions in a binary base. Outside the argument region show here, errors are *always* below $\frac{1}{2}$ ulp.

If $f \leq \sqrt{1/10}$, set $f = 10 \times f$ and $n = n - 1$, so that f is now in the interval $(\sqrt{1/10}, \sqrt{10}]$. Then introduce a change of variable, a Taylor-series expansion, and a polynomial representation of that expansion:

$$\begin{aligned}
z &= (f-1)/(f+1),\\
f &= (1+z)/(1-z),\\
D &= 2\log_{10}(e)\\
&= 2/\log(10),\\
\log_{10}(f) &= D \times (z + z^3/3 + z^5/5 + z^7/7 + z^9/9 + z^{11}/11 + \cdots)\\
&\approx D \times z + z^3\mathcal{Q}(z^2), \qquad \textit{polynomial fit incorporates } D \textit{ in } \mathcal{Q}(z^2).
\end{aligned}$$

For f in $(\sqrt{1/10}, \sqrt{10}]$, we have z in roughly $[-0.5195, +0.5195]$. The wider range of z requires longer polynomials compared to the binary case, and also makes the correction term $z^3\mathcal{Q}(z^2)$ relatively larger. Its magnitude does not exceed $|0.35z|$, so it provides barely one extra decimal digit of precision, instead of two. Accurate computation of z is easier than in the binary case: just set $z = \mathrm{fl}(\mathrm{fl}(f - \frac{1}{2}) - \frac{1}{2}) / \mathrm{fl}(f+1)$.

For improved accuracy, the constant D can be represented with a split like this:

$$\begin{aligned}
D &= D_{\mathrm{hi}} + D_{\mathrm{lo}}\\
&= 0.868\,588\,963\,806\,503\,655\,302\,257\,837\,833\,210\,164\,588\,794\,\ldots,\\
D_{\mathrm{hi}} &= 0.868\,588,
\end{aligned}$$

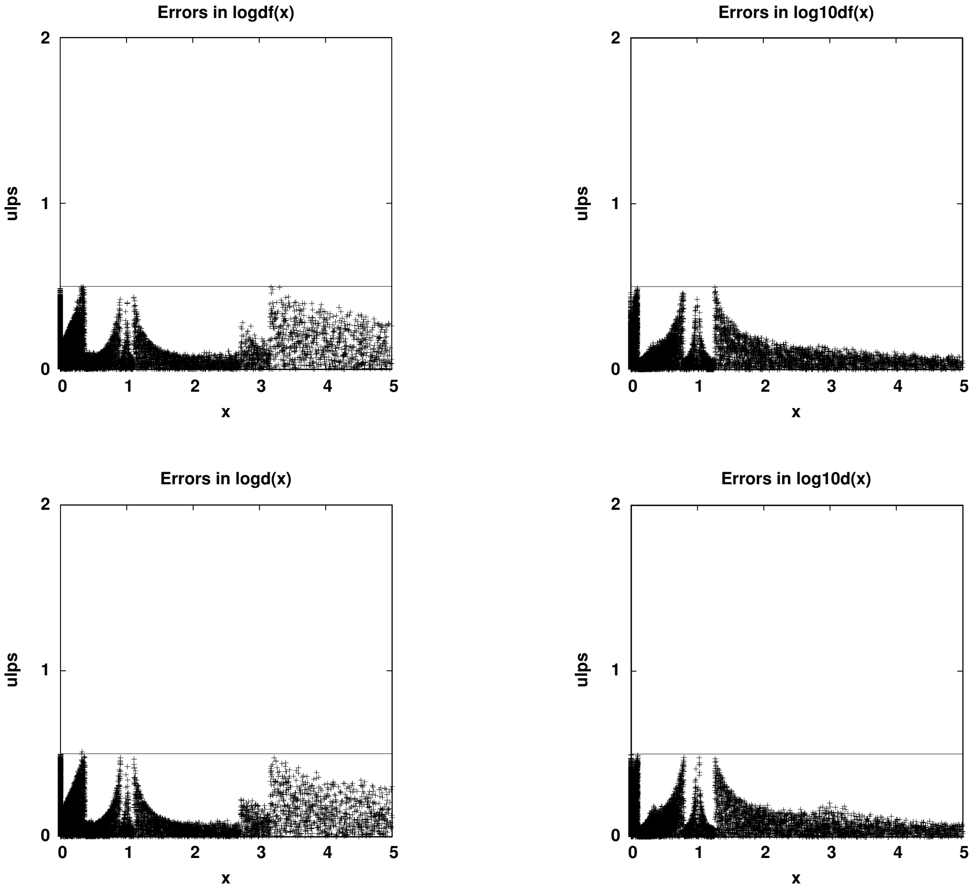


Figure 10.7: Errors in mathcw `LOG()` and `LOG10()` functions in a decimal base. Outside the argument region show here, errors are *always* below $\frac{1}{2}$ ulp.

$$D_{\mathrm{lo}} \approx 0.000\,000\,963\,806\,503\,655\,302\,257\,837\,833\,210\,164\,588\,794\ldots.$$

The base-10 logarithm of f is then computed in this order:

$$\begin{aligned}\log_{10}(f) &\approx D_{\mathrm{hi}} \times z + (D_{\mathrm{lo}} \times z + z^3 \mathcal{Q}(z^2))\\ &\approx \mathrm{fma}(D_{\mathrm{hi}}, z, \mathrm{fma}(D_{\mathrm{lo}}, z, z^3 \mathcal{Q}(z^2))).\end{aligned}$$

Rewriting the computation with fused multiply-add operations takes advantage of the higher precision offered by the split constant.

We can now compute the final result:

$$\log_{10}(|x|) = \log_{10}(f) + n.$$

The mathcw library code for the logarithm in a decimal base improves on the Cody/Waite procedure by splitting z into a sum of two parts, and then using a function from the `VSUM()` family to compute the sum of contributions to $\log(x)$ and an estimate of the error in that sum. The error term is then added to the sum to get the final value.

Figure 10.7 shows the measured errors in our implementations of the natural and base-10 logarithm by the algorithms described in this section.

When the function families `LOG2()`, `LOG8()`, and `LOG16()` are implemented by scaling of the natural logarithm, plots show errors up to about 1 ulp in IEEE 754 arithmetic. The mathcw library implementations of those functions therefore compute each of them directly with code that differs by only a few lines from that used for `LOG()` and `LOG10()`, but, of course, with separate polynomial approximations. That brings the errors of the base-2, -8, and -16 logarithms down below $\frac{1}{2}$ ulp, so we do not show error plots for them.

10.4 Logarithm near one

In 1987, Berkeley UNIX 4.3BSD introduced the `log1p()` function, defined as $\operatorname{log1p}(x) = \log(1+x)$. As with `expm1()`, the Taylor series (see **Section 2.6** on page 10) shows serious accuracy loss in $\log(1+x)$ for small x if log is used directly.

One reasonable computational procedure arises from the answer to the question: *if we know an **accurate** value of* $\log(w)$ *for a value w that is near $1+x$, can we compute* $\log(1+x)$ *from it?* The argument w is the result of forming $w = \operatorname{fl}(1+x)$ in finite-precision computer arithmetic. The value w is exactly representable, but differs from the exact mathematical $1+x$. The Maple symbolic-algebra system readily finds this series expansion for $w \neq 1$, and $a = \operatorname{fl}(w-1)$:

$$\log(1+x) = \log(w)(x/a)[1 + (1/(w\log(w)) - 1/a)(x-a) + \cdots]$$

The value $(x-a)$ can be either positive or negative, and its magnitude is $\mathcal{O}(\epsilon)$. The factor that multiplies it in the second term increases monotonically from $-\frac{1}{2}$ at $w=1$, to about -0.28 at $w=2$, then to -0 as $w \to +\infty$. Thus, the first two terms may sum to $1-\mathcal{O}(\frac{1}{2}\epsilon)$, 1, or $1+\mathcal{O}(\frac{1}{2}\epsilon)$. With the IEEE 754 default of *round to nearest*, the sum is either $1-\frac{1}{2}\epsilon$ or 1. When $\beta = 2$, those two are adjacent representable numbers, but in larger bases, they enclose additional machine numbers. Most libraries that supply the $\operatorname{log1p}(x)$ function include only the first term of the series, but for correct rounding in all bases and rounding modes, we need to use both terms.

Accuracy is limited primarily by that of the logarithm function for arguments near one. That requirement is *not* satisfied by arbitrary implementations of `log()`, but it is for the improved Cody/Waite algorithm used in the mathcw package.

If $\operatorname{fl}(w) = 1$ to machine precision, the denominator $w-1$ is zero, but we can then evaluate $\log(1+x)$ instead by its Taylor series.

If we omit the usual tests for special arguments, and ignore all but the first term in the series expansion, the implementation of `log1p()` is then just two statements in C:

```
w = ONE + x;
return (w == ONE) ? x : (LOG(w) * (x / (w - ONE)));
```

This code is correct for *round to nearest* arithmetic, but wrong for other rounding modes, including historical systems with truncating arithmetic. The code has at least these problems:

- When $|x|$ is smaller than half the machine epsilon, the computation $w = \operatorname{fl}(1+x)$ in non-default rounding modes may produce $1-\epsilon/\beta$, 1, or $1+\epsilon$. As a result, the Taylor-series code might never be selected, even though it is simpler and faster.
- When the Taylor series is selected, it is not sufficient to include just its first term in the small-argument case. Two terms of the series $\log(1+x) = x - x^2/2 + x^3/3 - \cdots$ are needed to get the right answer for some rounding modes.
- When $w \neq 1$, the second term in the bracketed series expansion is needed for correct rounding.
- On base-16 systems, the computation $1+x$ loses about three bits when x is small. Such expressions should be rewritten as $2(\frac{1}{2}+\frac{1}{2}x)$, because $\frac{1}{2}$ has no leading zero bits. For the same reason, the truncated series, $x - \frac{1}{2}x^2$, should be evaluated as `x - HALF*x*x`, *not* as `x*(ONE - HALF*x)`.

An implementation of that algorithm is quickly done, and testing finds errors up to about 2 ulps. If you need the `log1p()` function in a programming language that doesn't provide it, you can use that recipe to supply the missing function.

However, in the previous section, we saw that careful handling of the argument reduction eliminates almost all of the rounding error in the logarithm functions. The implementations of the `log1p()` function family in the mathcw library achieve comparable accuracy by a modification of the argument reduction, and reuse of the same polynomial approximations from the `log()` family.

If $|x|$ is sufficiently small, we use the six-term Taylor series to evaluate $\log(1+x)$ directly. Otherwise, we use a rational polynomial approximation. For any base b, we have

$$\begin{aligned}\log_b(1+x) &= \log_b(f \times \beta^N), && \textit{where } f \textit{ is in } [1/\beta, 1), \\ &= \log_b(f) + N\log_b(\beta).\end{aligned}$$

The problem is that explicit evaluation of $1+x$ loses trailing digits when x is small, so f is inaccurate. Nevertheless, the decomposition does give us an *exact* value of N.

For argument reduction in a nondecimal base, $\beta = 2^K$, we have

$$\begin{aligned}\log_b(1+x) &= \log_b(f) + N\log_b(2^K) \\ &= \log_b(f) + N \times K \times \log_b(2) \\ &= \log_b(f) + n \times \log_b(2), \\ n &= N \times K,\end{aligned}$$

and as we did for $\beta > 2$, we can adjust f upward by successive doubling to move it into the interval $[\frac{1}{2}, 1)$, decrementing n each time. We then make one additional adjustment to f and n, if necessary, to move f into the interval $[\sqrt{\frac{1}{2}}, \sqrt{2})$.

The task is now to compute z without using the inaccurate value of f in numeric computation. If we did not make the final doubling in f, follow these steps:

$$\begin{aligned}s &= 2^n, \\ 1+x &= f \times s, \\ z_{\mathrm{num}} &= f - 1, \\ Z_{\mathrm{num}} &= s \times z_{\mathrm{num}} \\ &= f \times s - s \\ &= (1+x) - s \\ &= (1-s) + x, && \textit{computational form,} \\ z_{\mathrm{den}} &= \tfrac{1}{2}f + \tfrac{1}{2}, \\ Z_{\mathrm{den}} &= s \times z_{\mathrm{den}} \\ &= f \times \tfrac{1}{2}s + \tfrac{1}{2}s \\ &= \tfrac{1}{2}(1+x) + \tfrac{1}{2}s \\ &= (\tfrac{1}{2} + \tfrac{1}{2}s) + \tfrac{1}{2}x, && \textit{computational form,} \\ z &= z_{\mathrm{num}}/z_{\mathrm{den}} \\ &= Z_{\mathrm{num}}/Z_{\mathrm{den}}.\end{aligned}$$

The scaled numerator and denominator are computed directly from the exact s and x, avoiding f entirely. However, unlike the case of the ordinary logarithm, we now in general have rounding errors in both the numerator and denominator that must be accounted for when we compute the split $z = z_1 + z_2$. We have

$$\begin{aligned}\mathrm{Err}_{\mathrm{num}} &= \mathrm{fl}(\mathrm{fl}(\mathrm{fl}(1-s) - Z_{\mathrm{num}}) + x), \\ \mathrm{Err}_{\mathrm{den}} &= \mathrm{fl}(\mathrm{fl}(\mathrm{fl}(1+s) - Z_{\mathrm{den}}) + x), \\ z_2 &= z - z_1 \\ &= (\mathrm{fma}(-z_1, Z_{\mathrm{den}}, Z_{\mathrm{num}}) + (\mathrm{Err}_{\mathrm{num}} - z_1 \times \mathrm{Err}_{\mathrm{den}}))/(Z_{\mathrm{den}} + \mathrm{Err}_{\mathrm{den}}).\end{aligned}$$

If we *did* make the final doubling of f, then, omitting intermediate steps, we have

$$\begin{aligned}Z_{\mathrm{num}} &= \mathrm{fl}(\mathrm{fl}(1-s) + x), \\ Z_{\mathrm{num}} &= \mathrm{fl}(\mathrm{fl}(\tfrac{1}{2} + \tfrac{1}{2}s) + \tfrac{1}{2}x), \\ \mathrm{Err}_{\mathrm{num}} &= \mathrm{fl}(\mathrm{fl}(\mathrm{fl}(1-s) - Z_{\mathrm{num}}) + x), \\ \mathrm{Err}_{\mathrm{den}} &= \mathrm{fl}(\mathrm{fl}(\mathrm{fl}(\tfrac{1}{2} + \tfrac{1}{2}s) - Z_{\mathrm{den}}) + \tfrac{1}{2}x),\end{aligned}$$

and we compute the split of z as before.

For a decimal base, we require a different intermediate variable, $z = (f-1)/(f+1)$, producing different results for the values of z_1 and z_2. We leave the details of the decimal computation to the source code file `logx.h` in the code block selected when the base defined by the preprocessor variable `B` is 10.

For bases $b \neq e$, the Taylor series must be scaled by a suitable split of the constant $1/\log(b)$, and that introduces a small problem. For arguments near the underflow limit, the computation of products of powers of x with the high and low parts of the split constant may require subnormals, introducing unwanted additional error. The solution is to scale the terms of the series by a power of the base to move the intermediate values into the range of normal numbers. The inverse cube of the machine epsilon is a suitable scale factor, and the Taylor-series block looks like this:

```
fp_t sum, x1, x2;
static const fp_t SCALE = FP(1.0) / (FP_T_EPSILON * FP_T_EPSILON * FP_T_EPSILON);
static const fp_t SCALE_INV = FP_T_EPSILON * FP_T_EPSILON * FP_T_EPSILON;

sum = (-SCALE / FP(6.0)      ) * x;
sum = ( SCALE / FP(5.0) + sum) * x;
sum = (-SCALE / FP(4.0) + sum) * x;
sum = ( SCALE / FP(3.0) + sum) * x;
sum = (-SCALE / FP(2.0) + sum) * x;

x2 = x * sum;           /* scaled partial sum */
x1 = SCALE * x;         /* scaled leading term */

sum  = x2 * INV_LN_B_LO;
sum += x2 * INV_LN_B_HI;
sum += x1 * INV_LN_B_LO;
sum  = FMA(x1, INV_LN_B_HI, sum);

result = SCALE_INV * sum; /* EXACT scaling */
```

Because `SCALE` is fixed, a compiler can replace the rational coefficients with constants, eliminating the divisions. Without that scaling, the measured errors in arguments near the underflow limit can reach several ulps.

Testing of that algorithm for the function $\log_2(1+x)$ showed errors slightly above 0.5 ulps in two small regions, $[-0.298, -0.293]$ and $[0.414, 0.426]$, where the function values are close to $\mp 1/2$, respectively. Code was therefore added to `l21px.h` to compute the functions for the `float` and `double` data types with Taylor-series expansions in those regions. For improved accuracy, the expansions compute the small corrections to be added to $\mp 1/2$, and the corrections have the same sign as those constants.

Figure 10.8 on the facing page shows measured errors in the base-e functions. Error plots for the companion families `log21p()`, `log81p()`, `log101p()`, and `log161p()` are similar, and thus not shown.

10.5 Exponential and logarithm in hardware

Two architecture families, the Intel IA-32 and Motorola 68000, provide hardware instructions summarized in Table 10.1 on page 294 for evaluation of selected exponential and logarithm functions.

When there is compiler support for inline assembly code (see the discussion in Section 13.26 on page 388), the instruction is available on the host platform, the preprocessor symbol `USE_ASM` is defined, and the argument is in range, the mathcw library code uses the hardware instructions. As an example, the value of 2^x can be computed from the expansions acceptable to the widely available `gcc` compiler family of these macro definitions:

```
#define IA32_HW_EXP2(result, x)                             \
  { __asm__ __volatile__("f2xm1" : "=t" (result) : "0" (x));  \
    result += ONE; }                    /* valid ONLY for |x| <= 1 */

#define M68K_HW_EXP2(result, x)                             \
  { __asm__ __volatile__("ftwotoxx %1, %0": "=f" (result): "f" (x)); }
```

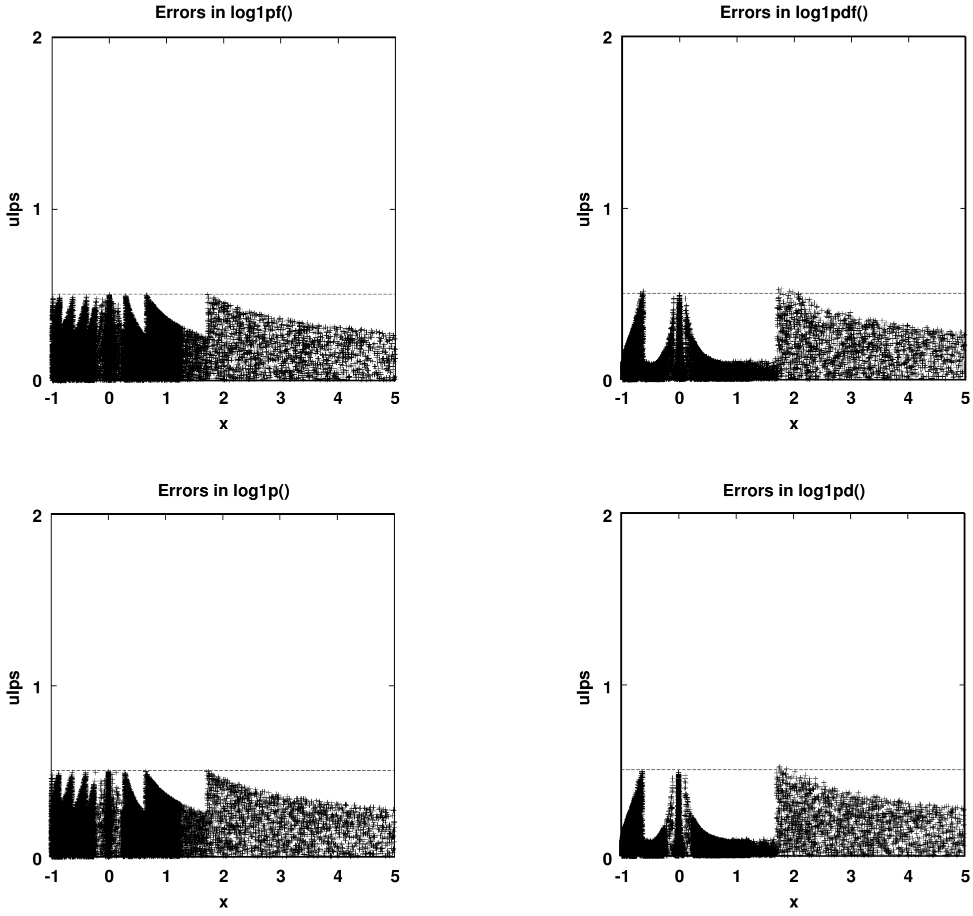


Figure 10.8: Errors in `LOG1P()` functions in binary and decimal bases.

The additional `x` on the Motorola instruction indicates that the operands are in the 80-bit format. The compiler handles the needed format conversions automatically for the input and output operands.

Tests on several Intel CPUs show no significant speedup from those hardware instructions compared to our software algorithms, but accuracy improves slightly.

Intel manuals do not describe the algorithms used to implement those instructions, nor do they document the instruction accuracy. Sometimes, that information can be found in research papers, such as the description of the transcendental functions in the AMD K5 processor which aims for errors below 1 ulp [LAS^{+}95]. Tests on recent processors suggest that the results are almost always correctly rounded. However, because of the large number of CPU models, and hardware vendors, in that long-lived architecture family, testing of the comparative accuracy of the software and hardware alternatives in the `mathcw` library code is advisable at end-user sites.

Motorola manuals note that the instructions use a 67-bit significand internally, providing three additional bits, and suggest that the worst-case error is about 64 ulps in the final 64-bit significand in the 80-bit format. Results for the 32-bit and 64-bit formats are therefore likely to be almost always correctly rounded. The only 68000 system available to this author for testing during the `mathcw` library development does not provide a usable `long double` data type, so testing of the Motorola hardware instructions has only been possible with the 32-bit `float` and 64-bit `double` data types.

The IA-32 logarithm instructions include a multiplier, y, to allow use of the relation $\log_{\alpha}(x) = \log_{\alpha}(2) \times \log_{2}(x)$ to find logarithms in other bases. To make that easier for common cases, the IA-32 hardware includes instructions `fld1`, `fldlg2`, and `fldln2` to load the constants 1, $\log_{10}(2)$, and $\log_{e}(2)$. However, we noted earlier in this chapter that high accuracy demands a representation of such constants to more than working precision, so we only use the

Table 10.1: Hardware instructions for exponential and logarithm functions.

Function	IA-32	68000	Conditions
e^x	—	`fetox`	
$e^x - 1$	—	`fetoxm1`	
2^x	—	`ftwotox`	
$2^x - 1$	`f2xm1`	—	$\lvert x\rvert \le 1$
10^x	—	`ftentox`	
$\log(x)$	—	`flogn`	
$\log(1+x)$	—	`flognp1`	
$\log_2(x)$	—	`flog2`	
$y\log_2(x)$	`fyl2x`	—	
$y\log_2(1+x)$	`fyl2xp1`	—	$\lvert x\rvert \le 1 - 1/\sqrt{2} \approx 0.293$
$\log_{10}(x)$	—	`flog10`	

logarithm instructions for computing the base-2 logarithm functions, for which the multiplier $y = 1$ is exact.

10.6 Compound interest and annuities

The functions of this chapter provide accurate solutions to two problems of concern to many: compound interest on loans, and investment returns. The `mathcw` library includes two families of functions that solve those problems with numerically stable algorithms, and Sun Microsystems SOLARIS includes those functions in the `-lsunmath` library. There is no mention of them in the ISO programming-language standards.

A loan has risks for the lender, and thus has associated costs, the largest of which is usually interest. The amount of the loan is called the principal, P, and for an interest rate r for a fixed period of time, the amount owed for that period is $P \times (1+r)$. If the loan is held for a second period, the debt is now $(1+r)$ times that value, or $P \times (1+r)^2$. When the loan is compounded for n periods, the principal grows by a factor $(1+r)^n$.

Interest rates are usually quoted as annual percentages, and it is imperative to adjust them to actual rates by dividing by 100, and to the period by dividing by the number of periods per year. Thus, an annual rate of 6% corresponds to $r = 0.06$ for one year, $r = 0.06/4$ for one quarter, and $r = 0.06/12$ for one month.

The computational problem is thus to define a function `compound(r,n)` that computes $(1+r)^n$ accurately. We can rewrite that expression as follows, using the substitution $x = \exp\big(\log(x)\big)$:

$$\begin{aligned}
\texttt{compound(r,n)} &= (1+r)^n \\
&= \exp\big(\log\big((1+r)^n\big)\big) \\
&= \exp\big(n\log(1+r)\big) \\
&= \exp\big(n\,\mathrm{log1p}(r)\big).
\end{aligned}$$

The code in `compound()` then only needs to check that the rate is finite and larger than -1, and if so, call `log1p()` and `exp()` to complete the job. Otherwise, it returns a NaN.

The error-magnification formulas (see Section 4.1 on page 61) look like this:

$$\begin{aligned}
C(r,n) &= (1+r)^n, \\
r(dC(r,n)/dr)/C(r,n) &= rn/(r+1), \\
n(dC(r,n)/dn)/C(r,n) &= n\log(1+r).
\end{aligned}$$

On the right-hand side, r and $\log(1+r)$ are usually small, so for most applications of `compound()`, argument sensitivity is not a serious problem unless n is large.

An *annuity* is an investment device that pays an annual return once it has matured. These payments normally continue until both the principal and the interest have been repaid. Many people buy annuities to provide assured retirement income, and a fixed-deposit bank savings account is essentially an annuity. Maturity may be reached either by the buyer's making regular payments until an agreed-upon principal has been paid, or the buyer may pay the entire principal immediately. The *present value of an annuity* of unit value is determined by working backward through n periods of interest compounding at rate r, from which we obtain this formula:

$$\begin{aligned}
\texttt{annuity(r,n)} &= \sum_{k=1}^{n} (1+r)^{-k} \\
&= \frac{1-(1+r)^{-n}}{r} \\
&= \frac{1-\exp\big(\log\big((1+r)^{-n}\big)\big)}{r} \\
&= \frac{1-\exp\big(-n\log(1+r)\big)}{r} \\
&= \frac{-\operatorname{expm1}\big(-n\operatorname{log1p}(r)\big)}{r}.
\end{aligned}$$

The second equation is just the series rewritten in closed form. Clearly, if r is small, the subtraction in the numerator suffers loss of leading digits, decreasing the precision of the computed result. Straightforward application of the second equation is not advisable unless the working precision is high enough to absorb the digit loss. However, the last equation provides a stable way to compute the annuity using our implementations of the two functions in the numerator, with an expected error below *four ulps*.

The error-magnification formulas for the annuity function take these forms:

$$\begin{aligned}
A(r,n) &= \frac{1-(1+r)^{-n}}{r}, \\
r(dA(r,n)/dr)/A(r,n) &= \frac{nr}{(1+r)(\texttt{compound(r,n)}-1)} - 1, \\
n(dA(r,n)/dn)/A(r,n) &= \frac{n\log(1+r)}{r\times\texttt{compound(r,n)}\times\texttt{annuity(r,n)}}.
\end{aligned}$$

Numerical evaluation of those expressions shows that their magnitudes remain below 1.0 for typical values of r and n, so argument sensitivity is small.

Here are some case studies of compound interest and annuities to illustrate the use of our library functions:

- The *future value of an annuity* is the sum of the future value of each payment P, where the payment is made at the end of each period. The first payment therefore has $n-1$ future periods in which to grow to $P(1+r)^{n-1}$. The second payment has only $n-2$ future periods, and grows to $P(1+r)^{n-2}$. The last payment brings the annuity to maturity, but has no time left to grow. We therefore have these relations:

$$\begin{aligned}
\text{future value of annuity} &= P(1+r)^{n-1} + P(1+r)^{n-2} + \cdots + P(1+r)^{0} \\
&= P \times \sum_{k=0}^{n-1} (1+r)^{k} \\
&= P \times \frac{(1+r)^{n}-1}{r} \\
&= P \times (1+r)^{n} \times \frac{1-(1+r)^{-n}}{r} \\
&= P \times \texttt{compound(r,n)} \times \texttt{annuity(r,n)}.
\end{aligned}$$

The sum is just a geometric series with the well-known closed form given in the third equation, and suitable factoring then leads to the simple product of our two functions.

For example, an annual payment of \$1000 into a 5% college tuition-fund annuity for fifteen years has a total payment of $15 000, but grows to

$$\begin{aligned}\text{future value} &= \$1000 \times \texttt{compound(0.05, 15)} \times \texttt{annuity(0.05, 15)}\\ &\approx \$1000 \times 2.078\,028 \times 10.379\,638\\ &\approx \$21\,579.\end{aligned}$$

Of course, that growth is reduced by inflation, which from 1900 to 2000 averaged about 3.10% annually in the US, and about 4.25% annually in the UK, from historical consumer price index data. Yearly fluctuations can be large, and both countries experienced years in that century with annual inflation rates above 20%. Repeating the computation with the effective rate reduced to $5\% - 3.10\% = 1.9\%$ produces a lower future value of $17 169.

- A mortgage for a borrower is essentially an annuity for the lender. If the lender provides a loan principal P, at rate r for n payment periods, then the borrower must pay it back in equal installments of size $P/\texttt{annuity(r, n)}$. For example, a loan at an annual rate of 6% for 20 years requires a monthly repayment given by

  $$P/\texttt{annuity(0.06 / 12, 20 * 12)} \approx 0.007\,164P,$$

  or quarterly payments of

  $$P/\texttt{annuity(0.06 / 4, 20 * 4)} \approx 0.021\,548P.$$

  If annual interest is 12%, the monthly repayment is about $0.011\,010P$.

  These computations lead to a handy rule of thumb for typical house mortgages: the monthly payment is about 1% of the loan principal. For a five-year vehicle loan, similar calculations produce a monthly payment estimate of about 2% of the principal.

  It is important to understand how much of the payment goes towards interest, enriching the lender, and how much towards the principal, reducing the borrower's debt. At the end of the first period, the payment must cover the interest on the principal, $r \times P$. For our example of a 20-year mortgage at 6%, paid annually, the interest is $0.06P$, and the payment is $P/\texttt{annuity(0.06,20)} \approx 0.087P$. This means that only a fraction of approximately $0.087 - 0.06 = 0.027$ of the principal is paid, giving the borrower an *equity* increase of just 2.7% of the loan. The first payment is about 69% interest and 31% principal. A longer loan life decreases the payment each period, but increases the total interest paid, and reduces the amount of principal paid in any period. If the loan terms, and personal finances, permit, it is advantageous for the borrower to make additional principal payments to increase equity, reduce debt and total interest paid, and shorten the loan life, unless the additional payments instead could be separately invested at a notably higher rate of return.

- Is it better for the borrower to make monthly, quarterly, or annual payments on a loan? We take the same example of 6% annually for 20 years, and compute the total amount repaid annually, as follows:

  $$\begin{aligned}\text{total (monthly payments)} &= 12P/\texttt{annuity(0.06 / 12, 20 * 12)}\\ &\approx 0.085\,971P,\\ \text{total (quarterly payments)} &= 4P/\texttt{annuity(0.06 / 4, 20 * 4)}\\ &\approx 0.086\,193P,\\ \text{total (one payment)} &= P/\texttt{annuity(0.06, 20)}\\ &\approx 0.087\,184P.\end{aligned}$$

  Monthly payments are clearly better for the borrower. The more frequent the loan payments, the sooner the lender recovers the investment (and can then reinvest it), and the faster the borrower reduces the debt.

- A savings deposit of a principal P earning annual interest of 6% grows to $P \times \texttt{compound(0.06, 10)} \approx 1.790P$ after a decade, and doubles in about 11.9 years.

  Credit cards carry higher risk for the lender because they are not backed by fixed assets, and thus, have higher interest rates. The US average 19% annual interest rate for credit cards at the time of writing this doubles the lender's investment in just four years. This helps to explain why credit cards, once a convenience available only to the wealthy, are now easily available to consumers in many countries, and have placed many users of such cards in permanent debt when they are unable to pay the entire monthly balance.

Higher-risk payday loans are even worse for the consumer: a recent newspaper report in this author's area cited annual rates of up to 450%, doubling the debt in just over 21 weeks.

- The US stock market growth rate for common *stocks*,[3] adjusted for inflation, was about 10.7% annually from 1926 to 2001. A diversified investment of one unit would grow in that time to `compound(0.107, 75)`, or about 2047. The corresponding returns for 25 and 50 years are about 13 and 161.

  The important lessons of interest compounding are that longer terms and higher rates are better for the investor or lender, and worse for the borrower.

- A lottery-jackpot prize of $270 000 000 was offered to the winner in annual payments of $13 500 000 for twenty years, or as an immediate, but reduced, single payment of $164 000 000. Which option should the winner choose, ignoring inflation and taxes?

  The immediate payment conservatively invested in government-backed *bonds*[4] at 5% per year would grow by a factor of `compound(0.05,20)` ≈ 2.65 in twenty years, to about $435 141 000, and an annual rate of just 2.525% would return the full prize. At the historical growth rate of the stock market, the principal could rise by `compound(0.107,20)` ≈ 7.64, reaching about $1 252 500 000.

  Alternatively, if the annual payments are chosen and reinvested, the formula for the future value of an annuity shows that the principal could grow in twenty years to

$$\$13\,500\,000 \times \texttt{compound(0.05, 20)} \times \texttt{annuity(0.05, 20)} \approx \$446\,390\,380 \qquad \textit{safe bonds},$$
$$\$13\,500\,000 \times \texttt{compound(0.107, 20)} \times \texttt{annuity(0.107, 20)} \approx \$837\,444\,239 \qquad \textit{risky stocks}.$$

  The newspaper that carried that lottery-jackpot story reported that the winner chose an immediate single payment, as our simple projections recommend.

  For much more on this subject, including its history, the many kinds of state lotteries in the US, and an elementary description of how the chances of winning are calculated, see *The Lottery Book* [Cat03].

- If a lottery winner accepts n annual payments instead of a lump sum, how much money does the lottery have to set aside in a separate fund pool to be able to meet that long-term obligation? Also, how does it determine the lump-sum amount?

  The Lottery Book gives without derivation a complicated formula that determines the pool amount required for the jackpot [Cat03, page 92]:

$$\text{annuity pool for } n \text{ payments} = \big(\frac{(1+r)^{n-1}-1}{r(1+r)^{n-1}} + 1\big) \times \frac{\text{jackpot}}{n}$$

  Here, the value of n is fixed by the rules of the lottery, usually at 20, 25, or 30 years, but the rate r is determined from a complex mixture of the rates of bonds with different maturity periods in an attempt to model future behavior of the bond market. The pool formula looks unpleasant until we remove a common factor in the first fraction, and relate it to the `annuity()` function:

$$\begin{aligned}\text{annuity pool for } n \text{ payments} &= \big(\frac{1-(1+r)^{-(n-1)}}{r} + 1\big) \times \frac{\text{jackpot}}{n} \\ &= \frac{\texttt{annuity(r,n - 1)} + 1}{n} \times \text{jackpot}\end{aligned}$$

  Let us again assume that the yearly rate is 5%, and compute the multiplier of the jackpot needed to determine the annuity pool amount:

$$\begin{aligned}M(r,n) &= \frac{\texttt{annuity(r,n - 1)} + 1}{n}, \\ M(0.05, 20) &\approx 0.654\,266, \\ M(0.05, 25) &\approx 0.591\,946, \\ M(0.05, 30) &\approx 0.538\,036.\end{aligned}$$

[3] *Stocks* are usually shares of a company, and thus carry no time limit. They may or may not include voting rights in company business decisions. Their value depends on the company's financial health and success, and also on their perceived worth to other investors.

[4] *Bonds*, *notes*, and *bills* are investments of decreasing, but fixed, term lengths that carry a high probability of repayment at a rate promised at the time of purchase, usually as a matter of law. They are normally issued by governments and large stable companies.

Thus, at a 5% rate, between 53% and 66% of the jackpot must be reserved to pay the annuity that grows to match it. However, the lottery could just give the winner that fraction of the jackpot in a single payment. That is why the lump-sum option generally pays only about $\frac{1}{2}$ to $\frac{2}{3}$ of the full jackpot.

A conservative winner who sticks to the bond market is likely to find both payment choices to be of similar value, but a less-cautious winner who accepts a lump sum and puts it into a diversified stock portfolio may do much better.

For the lottery win that began our discussion, we have

$$\begin{aligned} M(r, 20) &= 164\,000\,000/270\,000\,000 \\ &\approx 0.607\,407. \end{aligned}$$

There is no simple solution of the general equation for $M(r, n)$ to find the rate, but a numerical search finds it to be $r \approx 6.011\%$ when $n = 20$.

10.7 Summary

The exponential and logarithm are important elementary functions, and Cody and Waite show that one can obtain reasonable accuracy for them with relatively little code. Improving on their algorithms requires more accurate argument reduction, careful splitting of the contributions to the function values into sums of exact high and accurate low parts, and accurate summation of error-compensation terms and products of those parts.

The related functions $\mathrm{expm1}(x)$ and $\mathrm{log1p}(x)$ simplify important applications where direct computation of $\exp(x) - 1$ and $\log(1 + x)$ for small x would suffer serious accuracy loss. The software-library functions `expm1(x)` and `log1p(x)` deserve to be available in more programming languages, and we showed how simple versions of them can be implemented quickly, although without achieving the high accuracy of our final code for them.

Once accurate implementations of $\mathrm{expm1}(x)$ and $\mathrm{log1p}(x)$ are available, compound interest and annuities can be computed reliably. Buying, selling, borrowing, and lending are common in most human societies, and are the foundation of modern economies. Having `compound(r,n)` and `annuity(r,n)` functions readily accessible makes it easy to evaluate the costs of borrowing, and the returns on investments. Many economies have experienced downturns, and even collapse, when financial arithmetic, opportunities, and risks, have not been sufficiently understood.

11 Trigonometric functions

SOH-CAH-TOA: MNEMONIC DEVICE FOR THE
TRIGONOMETRIC FUNCTIONS IN A RIGHT TRIANGLE.
SOH : sin = OPPOSITE/HYPOTENUSE,
CAH : cos = ADJACENT/HYPOTENUSE,
TOA : tan = OPPOSITE/ADJACENT.

Trigonometry, from the Greek words for *triangle* and *measurement*, has been of interest to (at least some) humans for more than four millennia, with early evidence of use in Egypt, India, Mesopotamia, and Greece. The common method of measuring angles in degrees, minutes, and seconds comes from the Babylonian *sexagesimal* (base-60) number system, although in modern mathematics and computer software, circular measure of *radians*, where 2π radians is the angle of one complete rotation through a circle, is preferred.

The work of Fourier[1] on the expansions of functions in trigonometric series increased the importance of trigonometry in mathematics, engineering and science. Later independent discoveries of fast ways to compute the coefficients of Fourier series (see footnote in **Appendix I** on page 969) are revolutionary for numerical computation, although they are outside the scope of this book. If you are interested in learning more about their use on computers, consult almost any textbook on numerical analysis or signal processing, and particularly, [CHT02, KMN89, Van92]. For more on Fourier's place in the history of mathematics, see Hawking's comments and an English translation of some of Fourier's writing [Haw05, pages 491–562].

The mnemonic in the epigraph at the start of this chapter makes it evident that the tangent has a simple relation to the sine and cosine, as $\tan(\theta) = \sin(\theta)/\cos(\theta)$, where θ is the Greek letter *theta* that is commonly used in mathematics for angles. However, the range of the sine and cosine is $[-1, +1]$, and both can be zero, although never for the same argument, so the range of the tangent is $[-\infty, +\infty]$. That range difference is significant for floating-point computation, so most programming languages that offer mathematical functions provide three basic trigonometric functions: `cos(x)`, `sin(x)`, and `tan(x)`. Accurate computation of those three functions, and their inverses, is the subject of the remainder of this chapter.

Although you may have learned in school about the secant, cosecant, and cotangent, which are the reciprocals of the cosine, sine, and tangent, respectively, modern practice is to ignore those additional functions because of their simple relation to the other three.

11.1 Sine and cosine properties

The sine and cosine are periodic functions defined over the entire real axis, and they are graphed over a short interval in **Figure 11.1**. Their reciprocals, the cosecant and secant, are shown in **Figure 11.2**. Two mathematical notations, $\csc(x)$ and $\operatorname{cosec}(x)$, are common for the cosecant function.

The sine and cosine functions satisfy these periodicity and symmetry relations, where n is an integer:

$$\begin{aligned} \cos(x) &= \cos(x + 2n\pi), & \cos(x) &= -\cos(x + (2n+1)\pi), \\ \sin(x) &= \sin(x + 2n\pi), & \sin(x) &= -\sin(x + (2n+1)\pi), \\ \cos(x) &= \cos(-x), & \sin(x) &= -\sin(-x). \end{aligned}$$

Those relations allow us to reduce the computation to an interval of width no more than 2π, and we only need to consider nonnegative x values. Additional symmetry that is evident in the graph permits the interval of computation to be further reduced to $[0, \frac{1}{2}\pi]$.

[1] Jean Baptiste Joseph Fourier (1768–1830) was a French scientist who made important discoveries in mathematics and physics. The *Fourier series*, an expansion of a function as an infinite series of sines and cosines, is named after him, as are the *Fourier operator* and the various *Fourier transforms*. He also developed an analytical theory of heat, and in that field, the *Fourier number* or *Fourier modulus* is the ratio of the rate of heat conduction to the rate of thermal-energy storage.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_11

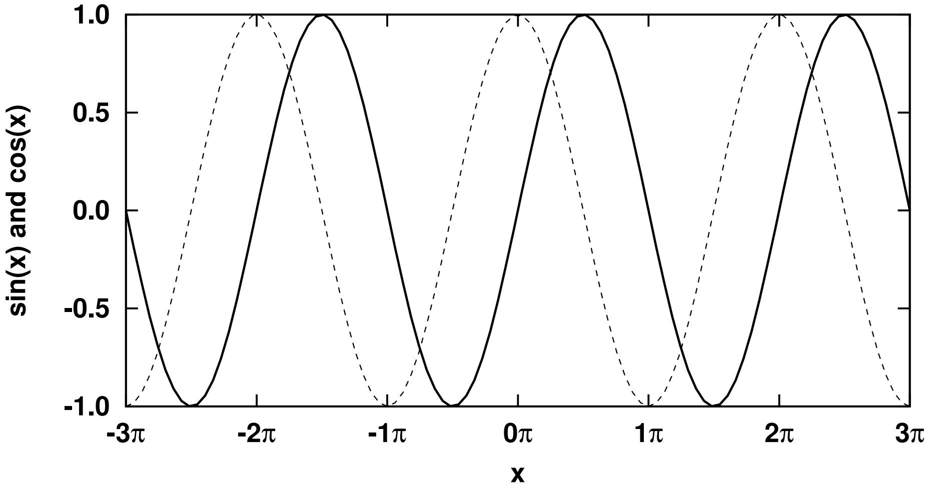


Figure 11.1: Three periods of the sine (solid) and cosine (dashed) functions.

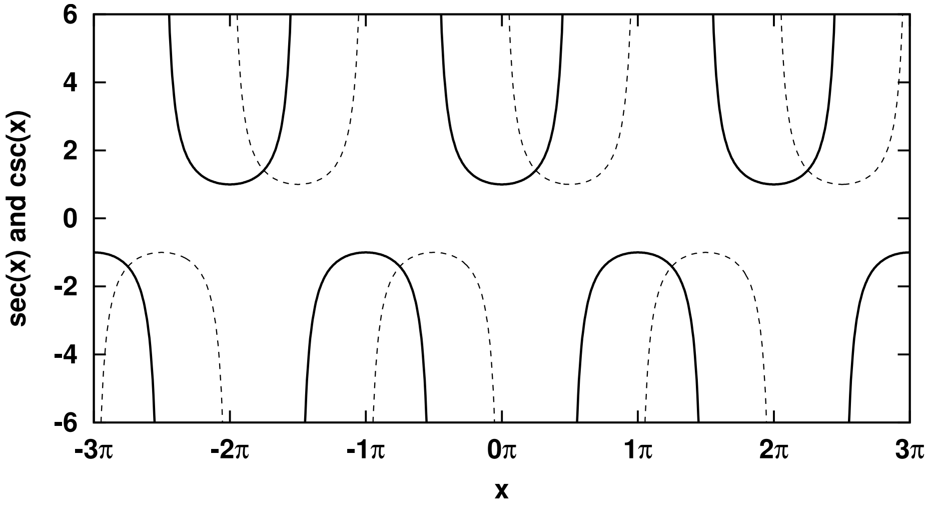


Figure 11.2: Three periods of the secant (solid) and cosecant (dashed) functions.

Addition formulas relate trigonometric functions of sums and differences of angles to functions of a single angle:

$$\cos(x \pm y) = \cos(x)\cos(y) \mp \sin(x)\sin(y),$$
$$\sin(x \pm y) = \sin(x)\cos(y) \pm \cos(x)\sin(y).$$

Those relations can be iterated to find functions of any integer multiple of x:

$$\cos(nx) = 2\cos((n-1)x)\cos(x) - \cos((n-2)x),$$
$$\sin(nx) = 2\sin((n-1)x)\cos(x) - \sin((n-2)x).$$

The mathematical formulas for arguments $x \pm y$ and nx are only useful for computation when there is no subtraction loss.

Although it is common in mathematics texts to drop parentheses around a simple argument of a trigonometric function, and to move a power before the argument, this book replaces the compact, but ambiguous, notation $\sin^n x$ with $(\sin(x))^n$.

From the multiple-angle relations, one can easily derive special cases that prove useful in further argument reduction, and for testing implementations of the functions:

$$\begin{aligned}
(\cos(x))^2 + (\sin(x))^2 &= 1, \\
\cos(x + \tfrac{1}{2}\pi) &= -\sin(x), \\
\sin(x - \tfrac{1}{2}\pi) &= -\cos(x), \\
\cos(\tfrac{1}{2}x) &= \pm\sqrt{\tfrac{1}{2}(1 + \cos(x))}, \\
\sin(\tfrac{1}{2}x) &= \pm\sqrt{\tfrac{1}{2}(1 - \cos(x))}, \\
\cos(2x) &= (\cos(x))^2 - (\sin(x))^2 \\
&= 2(\cos(x))^2 - 1 \\
&= 1 - 2(\sin(x))^2 \\
&= \frac{1 - (\tan(x))^2}{1 + (\tan(x))^2}, \\
\sin(2x) &= 2\sin(x)\cos(x) \\
&= \frac{2\tan(x)}{1 + (\tan(x))^2}, \\
\cos(3x) &= 4(\cos(x))^3 - 3\cos(x), \\
\sin(3x) &= 3\sin(x) - 4(\sin(x))^3.
\end{aligned}$$

For the half-angle formulas, consult the function graphs to determine the correct sign.

There are formulas that define sums and differences of trigonometric functions as products of other functions [AS64, §4.3.34–§4.3.42] [OLBC10, §4.21], which may be helpful in reducing subtraction loss in computer arithmetic:

$$\begin{aligned}
\cos(x) - \cos(y) &= -2\sin(\tfrac{1}{2}(x - y))\sin(\tfrac{1}{2}(x + y)), \\
\cos(x) + \cos(y) &= 2\cos(\tfrac{1}{2}(x - y))\cos(\tfrac{1}{2}(x + y)), \\
\sin(x) - \sin(y) &= 2\sin(\tfrac{1}{2}(x - y))\cos(\tfrac{1}{2}(x + y)), \\
\sin(x) + \sin(y) &= 2\sin(\tfrac{1}{2}(x + y))\cos(\tfrac{1}{2}(x - y)), \\
(\cos(x))^2 - (\cos(y))^2 &= -\sin(x + y)\sin(x - y), \\
(\cos(x))^2 - (\sin(y))^2 &= \cos(x + y)\cos(x - y), \\
(\sin(x))^2 - (\sin(y))^2 &= \sin(x + y)\sin(x - y).
\end{aligned}$$

In some applications, including one that we encounter later in this chapter, we need to evaluate $1 - \cos(x)$ or $1 - \sin(x)$ when the values of the trigonometric functions are near 1. They are just special cases of the difference formulas where we first substitute $x \to \frac{1}{2}\pi$, and then $y \to x$:

$$\begin{aligned}
1 - \cos(x) &= 2\big(\sin(\tfrac{1}{2}x)\big)^2 & \qquad 1 - \sin(x) &= 2\big(\cos(\tfrac{1}{2}x + \tfrac{1}{4}\pi)\big)^2 \\
&= 2\big(\cos(\tfrac{1}{2}x + \tfrac{1}{2}\pi)\big)^2, & &= 2\big(\sin(\tfrac{1}{2}x - \tfrac{1}{4}\pi)\big)^2.
\end{aligned}$$

We can also find those equations by squaring and rearranging the half-angle formulas, and using an argument shift by $\pi/2$ to convert a cosine to a sine.

As a historical note, the function $1 - \cos(x)$ is called the *versine*, and $1 - \sin(x)$ is the *coversine*. Those functions, and half their values, called the *haversine* and the *cohaversine*, were often included in books of numerical tables because accurate values for them are required in navigational formulas in spherical trigonometry. The need for them disappeared as machines took over such computations, and they are rarely mentioned in modern mathematical texts.

The first derivatives have simple relations to the functions themselves, and are also restricted to the range $[-1, +1]$:

$$d\cos(x)/dx = -\sin(x), \qquad d\sin(x)/dx = \cos(x).$$

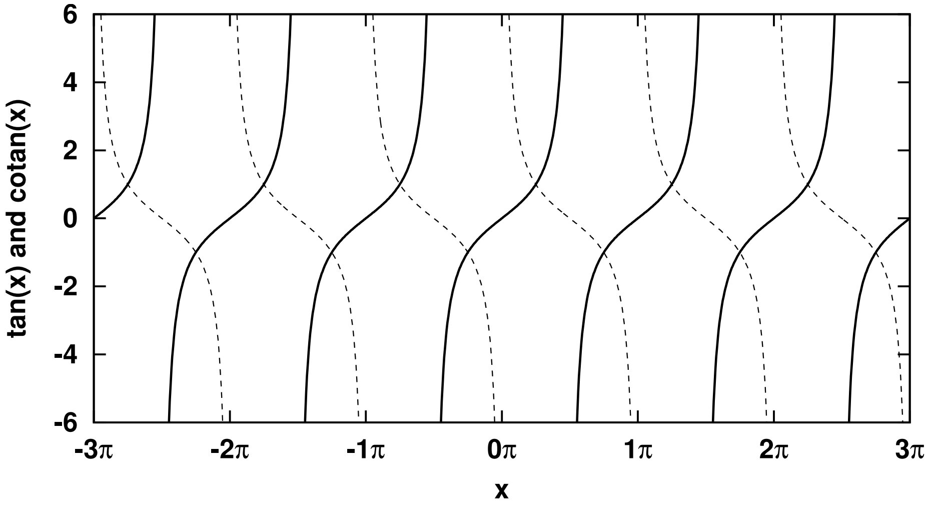


Figure 11.3: Six periods of the tangent (solid) and cotangent (dashed) functions.

The Taylor series of the cosine and sine converge rapidly:

$$\begin{aligned}\cos(x) &= 1 - (1/2)x^2 + (1/24)x^4 - (1/720)x^6 + (1/40\,320)x^8 - \\ &\quad (1/3\,628\,800)x^{10} + \cdots + (-1)^n(1/(2n)!)x^{2n} + \cdots, \\ \sin(x) &= x - (1/6)x^3 + (1/120)x^5 - (1/5040)x^7 + (1/362\,880)x^9 - \\ &\quad (1/39\,916\,800)x^{11} + \cdots + (-1)^n(1/(2n+1)!)x^{2n+1} + \cdots.\end{aligned}$$

Those series produce the handy rules of thumb that, for small x, $\cos(x) \approx 1$ and $\sin(x) \approx x$.

11.2 Tangent properties

The tangent function, which is just the ratio of the sine and cosine, is periodic over an interval of width π, instead of the width 2π for the other two functions. Three mathematical notations for its reciprocal, the cotangent, are common: $\cot(x)$, $\operatorname{cotan}(x)$, and $\operatorname{ctn}(x)$. The tangent and cotangent are graphed over a small interval in Figure 11.3. For the tangent, there are asymptotes at odd multiples of $\frac{1}{2}\pi$ where the function goes to $+\infty$ from the left, and to $-\infty$ from the right. The asymptotes of the cotangent occur at even multiples of $\frac{1}{2}\pi$.

The tangent satisfies these periodicity and symmetry relations, where n is an integer:

$$\tan(x) = \tan(x + n\pi), \qquad \tan(x) = -\tan(-x).$$

Addition formulas connect tangents of sums and differences of angles to trigonometric functions of single angles:

$$\begin{aligned}\tan(x \pm y) &= \frac{\tan(x) \pm \tan(y)}{1 \mp \tan(x)\tan(y)}, \\ &= \frac{\sin(x)\cos(y) \pm \cos(x)\sin(y)}{\cos(x)\cos(y) \mp \sin(x)\sin(y)}.\end{aligned}$$

The second form avoids numerical problems from infinite operands when Infinity is not representable, and from significance loss in a hexadecimal base for a denominator of the form $1 + \mathit{small}$, although that particular problem can be easily solved by halving the numerator and denominator.

The addition formulas readily lead to these special cases:

$$\tan(\tfrac{1}{2}x) = \pm\sqrt{\frac{1-\cos(x)}{1+\cos(x)}} \qquad \tan(2x) = \frac{2\tan(x)}{1-(\tan(x))^2},$$

$$= \frac{1-\cos(x)}{\sin(x)} \qquad\qquad \tan(3x) = \frac{3\tan(x)-(\tan(x))^3}{1-3(\tan(x))^2},$$

$$= \frac{\sin(x)}{1+\cos(x)}, \qquad\qquad \tan(nx) = \frac{\tan((n-1)x)+\tan(x)}{1-\tan((n-1)x)\tan(x)}.$$

For the first half-angle formula, consult the function graph to determine the correct sign.

Sums and differences of tangents and cotangents have representations that may help reduce subtraction loss in computation:

$$\tan(x) \pm \tan(y) = \frac{\sin(x \pm y)}{\cos(x)\cos(y)},$$
$$\operatorname{cotan}(x) \pm \operatorname{cotan}(y) = \frac{\sin(x \pm y)}{\sin(x)\sin(y)}.$$

The first derivative of the tangent has a simple relation to the tangent itself:

$$d\tan(x)/dx = 1 + (\tan(x))^2.$$

Thus, the slope of the tangent function is never less than $+1$.

Unlike the sine and cosine, where the Taylor-series coefficients have simple forms, the Taylor series of the tangent involves the famous, but complicated, *Bernoulli numbers*, B_{2n} [AS64, Table 23.2, p. 804], [AW05, §5.9], [AWH13, §12.2], [OLBC10, §24.2]. They are rational numbers that alternate in sign, grow factorially, and hide some deep mathematics. For example, they have curious relations to prime numbers, and to the famous *Riemann zeta function*.[2] We also encounter the Bernoulli numbers later in this book in series for the hyperbolic tangent (**Section 12.1** on page 342), the gamma function (**Section 18.1** on page 525), and the psi function (**Section 18.2.6** on page 542):

$$\begin{aligned}\tan(x) = x &+ (1/3)x^3 + (2/15)x^5 + (17/315)x^7 + (62/2835)x^9 + \\ &(1382/155\,925)x^{11} + (21\,844/6\,081\,075)x^{13} + \cdots + \\ &(B_{2n}(-4)^n(1-4^n)/(2n)!)x^{2n-1} + \cdots.\end{aligned}$$

Clearly, for small x, we have $\tan(x) \approx x$.

Table 11.1 on the following page lists the first two dozen or so nonzero Bernoulli numbers, and we describe their calculation in **Section 18.5** on page 568 and **Section 18.5.1** on page 574. They are difficult to compute accurately with limited-precision arithmetic, and they soon reach the overflow limit of most floating-point systems. Fortunately, we can often get by with just a few of the low-order Bernoulli numbers, expressing them directly in the code as exact rational numbers that optimizing compilers can convert to accurate floating-point equivalents.

To help investigate how closely the computed tangent function can approach the asymptotes, here is the Taylor series of its reciprocal:

$$\begin{aligned}1/\tan(x) &= z + (1/3)z^3 + (2/15)z^5 + (17/315)z^7 + \cdots, \\ z &= \tfrac{1}{2}\pi - x.\end{aligned}$$

Because $\frac{1}{2}\pi \approx 1.570\,796\ \ldots$ is a transcendental number, it cannot be represented exactly as a floating-point number. Instead, it lies *between* two adjacent machine numbers that are separated by the machine epsilon, ϵ. The closest machine number may be just below, or just above, $\frac{1}{2}\pi$, depending on the value of the base and the precision, as shown in **Table 11.2** on page 305. The reciprocals of the numbers in the second column of that table are roughly the largest values that $\tan(x)$ can reach in finite-precision arithmetic, although rarely, for large integer multiples of $\frac{1}{2}\pi$, it is possible to get closer to an asymptote. Those reciprocals are far from the largest representable number on most of the systems listed, so the software designer must decide whether `HALF_PI` should be treated as $\frac{1}{2}\pi$, which is probably what *humans* expect, or as a nearby exact machine number. The tangents of those two values are quite different.

[2] Bernhard Riemann's (1826–1866) hypothesis (1859) about the locations of the complex roots of the zeta function, $\zeta(z) = \sum_{n=1}^{\infty} n^{-z}$,where ζ is the Greek letter *zeta*, is possibly the greatest unsolved problem in all of mathematics [CJW06, Clay09, Dev02, Der03, Lap08, dS03, Roc06, Sab03]. Riemann also made fundamental contributions to the study of the geometry of curved spaces, and Albert Einstein (1879–1955) later built on Riemann's work in the *General Theory of Relativity* (1916), which accurately describes the structure and evolution of the Universe. Riemann died of tuberculosis at the age of 39.

Table 11.1: Bernoulli numbers of even order. Those of odd order are all zero, except for $B_1 = -\frac{1}{2}$. Growth is rapid: the numerator of B_{100} has 83 digits, that of B_{200} has 222 digits, and that of B_{500} has 743 digits.

2*n*	B_{2n}	**2*n***	B_{2n}
0	+1	28	−23 749 461 029/870
2	+1/6	30	+8 615 841 276 005/14 322
4	−1/30	32	−7 709 321 041 217/510
6	+1/42	34	+2 577 687 858 367/6
8	−1/30	36	−26 315 271 553 053 477 373/1 919 190
10	+5/66	38	+2 929 993 913 841 559/6
12	−691/2730	40	−261 082 718 496 449 122 051/13 530
14	+7/6	42	+1 520 097 643 918 070 802 691/1806
16	−3617/510	44	−27 833 269 579 301 024 235 023/690
18	+43 867/798	46	+596 451 111 593 912 163 277 961/282
20	−174 611/330	48	−5 609 403 368 997 817 686 249 127 547/46 410
22	+854 513/138	50	+495 057 205 241 079 648 212 477 525/66
24	−236 364 091/2730	52	−801 165 718 135 489 957 347 924 991 853/1590
26	+8 553 103/6	...	.../...

Similarly, the Taylor series of $\cos(x)$ and $\sin(x)$ near $x = \frac{1}{2}\pi$ are

$$\begin{aligned}\cos(x) &= z - (1/6)z^3 + (1/120)z^5 - \cdots,\\ \sin(x) &= 1 - (1/2)z^2 + (1/24)z^4 - \cdots,\\ z &= \tfrac{1}{2}\pi - x,\end{aligned}$$

so depending on software-design decisions, we may get a nonzero value for cos(`HALF_PI`), but we *should* find that fl(`sin(HALF_PI)`) = 1 in a *round-to-nearest* mode when $\frac{1}{2}z^2 \ll \epsilon/\beta$ (the little epsilon).

11.3 Argument conventions and units

In the introduction to this chapter, we noted that computer software usually follows the mathematical practice that arguments of trigonometric functions are in radians. However, some languages, and some implementations of C, provide companion functions with different argument units.

For example, Sun Microsystems SOLARIS includes the functions `cospi(x)`, `sinpi(x)`, and `tanpi(x)`, where x is in units of π. Thus, `cospi(x)` is defined to be $\cos(\pi x)$, which can be rewritten as $\cos((2\pi)(\frac{1}{2}x))$ and then with the substitution $\frac{1}{2}x = n + r$, replaced by $\cos(2\pi r)$. The value r is the fractional part of $\frac{1}{2}x$, which is recovered exactly, when `HALF * x` is exact, by either of these statements:

```
r = HALF * x - TRUNC(HALF * x);
r = MODF(HALF * x, NULL);
```

However, we note in Section 9.3 on page 264 that a `NULL` argument in the `modf()` family is not portable.

SOLARIS also offers three slightly different functions, `cosp(x)`, `sinp(x)`, and `tanp(x)`, that use these argument reductions:

$$\begin{aligned}x &= n\,\mathrm{fl}(2\pi) + r, && \textit{for cosine and sine},\\ x &= n\,\mathrm{fl}(\pi) + r, && \textit{for tangent}.\end{aligned}$$

That choice does not eliminate the argument-reduction problem, but it simplifies it, because the reduction can then be done *exactly* with a call to a standard library function to compute either `fmod(x,TWO_PI)` or `fmod(x,PI)`. We show in Section 6.15 on page 146 how to implement that function to provide an exact remainder operation. Unfortunately, use of those functions introduces a portability problem, because $\mathrm{fl}(\pi)$ depends on the precision, and may differ from working precision if the library code, or hardware, uses higher internal precision for that constant.

Table 11.2: Distances of closest machine numbers, `HALF_PI`, to $\frac{1}{2}\pi$ for t-digit base-β floating-point formats.

t	$\frac{1}{2}\pi$ − `HALF_PI`	**Sample architectures**
		$\beta = 2$
13	−4.5e−06	Lawrence Livermore National Laboratory S-1
24	1.6e−08	DEC VAX, **IEEE 754**
27	9.9e−10	DEC PDP-10, S-1, General Electric 600, Honeywell 6000
48	−1.7e−15	CDC 6000 and 7000, Cray 1
53	−5.0e−17	DEC VAX, **IEEE 754**
54	5.7e−18	DEC PDP-10 KA10
56	5.7e−18	DEC VAX
57	−1.2e−18	S-1
59	5.2e−19	DEC PDP-10 KL10
62	8.3e−20	DEC PDP-10 KL10
63	−2.5e−20	General Electric 600, Honeywell 6000
64	−2.5e−20	**IEEE 754**
96	8.5e−32	CDC 6000 and 7000
113	4.3e−35	**IEEE 754**
237	−4.5e−73	extended **IEEE 754**
		$\beta = 8$
13	7.4e−13	Burroughs B5700, B6700, B7700
26	6.4e−25	Burroughs B5700, B6700, B7700
		$\beta = 10$
7	2.7e−08	**IEEE 754**
16	1.9e−17	**IEEE 754**
34	4.2e−35	**IEEE 754**
70	3.1e−72	extended **IEEE 754**
		$\beta = 16$
6	1.6e−08	IBM System/360
14	5.7e−18	IBM System/360
28	4.3e−35	IBM System/360

OSF/1 and SOLARIS provide another three functions with arguments in degrees: `cosd(x)`, `sind(x)`, and `tand(x)`. Their names conflict with those of the new functions for decimal floating-point arithmetic, so they might be renamed to use a longer prefix or suffix, such as `degcos(x)` or `cosdeg(x)`. We use the latter form. The argument reduction for degrees can be done exactly with a standard function: `fmod(x,360.0)`.

In some military organizations, yet another angular unit is common: *mils*. The *British mil* is defined by 1000 British mils $=$ 1 radian, but the *NATO mil* differs slightly: 6400 NATO mils $= 2\pi$ radians. We therefore have 1 NATO mil $\approx 0.981\,748\ldots$ British mil. To add to the confusion, there is also a *US mil* defined by 1000 US mils $= 90^\circ \approx 1.570\,796$ radians.

Here is an explanation of the origin of the mil unit. The length of an arc of a circle of radius r is $s = r\theta$ when θ is in radians. Thus, the range of a target of width w (in any convenient units) that subtends a small angle of b British mils is just $r \approx (1000/b)w$, providing an easy way to convert angles to distances for gunnery settings.

This author has never encountered a programming language with trigonometric functions for mil arguments, but in view of the problem that we discuss in the next section, and in **Section 11.6** on page 313, it is *not* sufficient to simply define functions with scaled arguments, such as with preprocessor statements like these

```
#define cosbritishmil(x)        cos((x) / 1000.0)
#define cosnatomil(x)           cos((TWO_PI / 6400.0) * (x))
#define cosusmil(x)             cos((HALF_PI / 1000.0) * (x))
```

if accuracy is to be maintained for large arguments that lie well outside the principal ranges (in radians) of $[-\pi, \pi]$ and $[0, \frac{1}{2}\pi]$.

11.4 Computing the cosine and sine

After the special arguments of Infinity, NaN, and signed zero have been handled, and the argument magnitude has been suitably reduced to the interval $[0, \frac{1}{2}\pi]$, the Cody/Waite algorithm for the cosine and sine uses the relation $\cos(x) = \sin(x + \frac{1}{2}\pi)$ to further simplify the computation to just that of the sine. For tiny $|r|$, they use the first term of the Taylor series, $\sin(r) \approx r$. Otherwise, they evaluate the sine by a polynomial approximation of the form

$$g = r^2, \qquad \sin(r) \approx r + rg\mathcal{P}(g)/\mathcal{Q}(g),$$

valid for r in the range $[0, \frac{1}{2}\pi]$. For that range of r, the factor $g\mathcal{P}(g)/\mathcal{Q}(g)$ lies in $[0, -0.364]$, providing a modest correction to an exact value. That negative correction is small enough that there is no significance loss in the implicit subtraction that happens for all $r > 0$, because $|\sin(r)| \leq |r|$. Nevertheless, tests of that algorithm with decimal arithmetic show that `sindf(HALF_PI)` and `sind(HALF_PI)` are smaller than the exact value by ϵ/β. We therefore extend the Cody/Waite algorithm to switch to the formula

$$\sin(r) = 1 - 2\big(\sin(\tfrac{1}{2}r - \tfrac{1}{4}\pi)\big)^2$$

when $\sin(|r|) > 0.9$, or $|r| > \sin^{-1}(0.9) \approx 1.12$. That choice extends the effective precision on the right-hand side by one or more decimal digits, and produces the exact value of the sine of `HALF_PI` in both binary and decimal formats.

Argument reduction for the cosine requires additional care to avoid subtraction loss when $x \approx -\frac{1}{2}\pi$. We clearly cannot just compute the argument of the sine as $x + \frac{1}{2}\pi$, but we can do so analytically as follows:

$$|x| = n\pi + r, \qquad |x| + \tfrac{1}{2}\pi = (n + \tfrac{1}{2})\pi + r = N\pi + (r - \tfrac{1}{2}\pi).$$

Determine the integer N like this:

$$N = \operatorname{round}((|x| + \tfrac{1}{2}\pi)/\pi) = \operatorname{round}(|x|/\pi + \tfrac{1}{2}) = \operatorname{trunc}(|x|(1/\pi) + 1).$$

The reduced argument of the sine is then

$$r = |x| - (N - \tfrac{1}{2})\pi.$$

As long as $N - \frac{1}{2}$ is exactly representable, we can recover an accurate value of r by the stepwise reduction described on page 245.

Cody and Waite derive their polynomial approximation from earlier work by Hart and others [HCL$^+$68], and set $\mathcal{Q}(g) = 1$, eliminating division. Rational approximations are usually more accurate than single polynomials of the same total degree, and numerical experimentation for the sine function shows that about six bits of accuracy can be gained at the expense of one division. Our code normally uses the Cody/Waite polynomials for the precisions for which they are suitable, and otherwise, uses rational approximations. However, testing shows that their $\langle 3/0 \rangle$ polynomial for 24-bit precision can produce errors slightly above 0.5 ulps, so we replace it with a new $\langle 4/0 \rangle$ approximation.

The key improvement that we make upon the Cody/Waite recipe is replacement of their argument-reduction step by the more accurate `reduce()` and `eriduce()` routines that we describe in **Chapter 9**. That avoids accuracy loss for arguments that are near multiples of $\frac{1}{2}\pi$, and allows us to eliminate their restriction to arguments $|x| < \operatorname{trunc}(\pi\beta^{t/2})$, extending accurate coverage to the entire real axis. Because the reductions need long expansions of irrational constants, we conserve memory and reduce code complexity by moving the job to a separate function, defined in file `rpx.h`, that can be used by several others. We present it as a semi-literate program in parts, beginning with its simple interface:

```
fp_t
RP(fp_t x, int *pn, fp_t *perr)
{
    /*
    ** Reduce FINITE x (unchecked!) with respect to PI, returning a
    ** result in [-PI/2, PI/2] such that
    **
```

```
   **     r = (return value) + (*perr)
   **
   ** satisfies x = n*PI + r to roughly twice working precision,
   ** and (*perr) is no larger than one ulp of the return value.
   **
   ** A NULL value for perr indicates the error term is not needed.
   */
```

The first task is to declare and initialize some local variables, and then quickly dispense with the common cases where the result would be identical to that computed on the last call, or where no reduction is needed:

```
   fp_t err, result, xabs;
   int n;
   static fp_t last_x = FP(0.);
   static fp_t last_result = FP(0.);
   static fp_t last_err = FP(0.);
   static int last_n = 0;

   err = ZERO;
   n = 0;
   xabs = QABS(x);

   if (x == last_x)    /* return cached results from last call */
   {
       result = last_result;
       err = last_err;
       n = last_n;
   }
   else if (xabs <= PI_HALF)
   {
       result = x;
       err = ZERO;
       n = 0;
   }
```

For decimal arithmetic, we use the `REDUCE()` family routine of highest precision when the argument magnitude is not too large, or else we produce a quiet NaN:

```
   else
   {

#if B == 10

       /* ereduce() and eriduce() do not work for a decimal base, so
          do the reduction as accurately as we can */
       if (xabs < XREDMAX)
       {
           decimal_long_long_double r, r_lo;

           r = _reddll((decimal_long_long_double)x, ONE_OVER_PI_DLL, NC_PI_DLL, C_PI_DLL, &n, &r_lo);
           result = (fp_t)r;
           err = (fp_t)((r - (decimal_long_long_double)result) + r_lo);
       }
       else
           result = SET_EDOM(QNAN(""));
```

The cutoff value, `XREDMAX`, depends on the number of digits supplied in the arrays that hold the expansion of the reduction constant, through the requirement described in **Section 9.1** on page 243 that nc_k be exactly representable.

For decimal arithmetic, XREDMAX is 10^8. For other bases, it is 8000 when there are at least 53 bits in the significand, and otherwise, it is 1000.

Because we have a higher-precision representation of π, we can, with negligible cost, compute a correction to the function value and later return the correction via the final pointer argument. The sum of the function value and the correction then provides an argument reduction with double the working precision, and that can be exploited to further reduce errors in computed functions.

For nondecimal arithmetic, a member of the REDUCE() family of at least type double does the job for smaller arguments, and the slower exact-reduction routine, ERIDUCE(), handles larger arguments:

```
#else  /* B != 10 */

        if (xabs < XREDMAX)     /* use simple and fast reduction
                                   in at least double precision */
        {

#if defined(HAVE_FP_T_SINGLE)
            double r;

            r = _red((double)x, ONE_OVER_PI_DBL, NC_PI_DBL, C_PI_DBL, &n, (fp_t *)NULL);
            result = (fp_t)r;
            err = (fp_t)(r - (double)result);
#else
            result = REDUCE(x, ONE_OVER_PI, NC_PI, C_PI, &n, &err);
#endif /* defined(HAVE_FP_T_SINGLE) */

        }
        else            /* use complex and much-slower reduction */
            result = ERIDUCE(x, PI, NC_ONE_OVER_PI, C_ONE_OVER_PI, &n, &err, bits_per_chunk);

#endif /* B == 10 */

    }
```

All that remains is possible storage of the error term and the multiplier n, preservation of the final results in static variables, and a return to the caller:

```
    if (perr != (fp_t *)NULL)
        *perr = err;

    if (pn != (int *)NULL)
        *pn = n;

    last_n = n;         /* cache results for possible later use */
    last_result = result;
    last_x = x;

    return (result);
}
```

A similar function, RPH(), in file rphx.h provides reductions with respect to $\frac{1}{2}\pi$. The functions RP() and RPH() are intended only for internal library use, so they are *not* declared in mathcw.h, and their macro names expand to the implementation-reserved forms _rp() and _rph() with the usual type suffixes.

The Cody/Waite argument reduction for the cosine is applicable only in a limited range, and we want to handle all finite arguments, and still be able to reuse the polynomial approximations for the sine. The solution to that problem is to use the RPH() family to reduce the cosine argument with respect to $\frac{1}{2}\pi$ instead of π. We can then use the formulas in the second block of **Table 9.1** on page 244 and the relation $\cos(x) = \sin(\frac{1}{2}\pi - x)$ to further reduce the computation to the sine of a reduced argument. The four cases are handled by code in cosx.h like this:

```
else
{
    hp_t f_abs, s, v[4];
    int negate;

    f_abs = QABS(f);
    negate = 0;

    switch (((n < 0) ? -n : n) & 0x3)
    {
    default:
    case 2: /* cos(x) = -cos(f) = -cos(|f|) = -sin(PI/2 - |f|) */
        negate = 1;
        /* FALL THROUGH */

    case 0: /* cos(x) = cos(f) = cos(|f|) = sin(PI/2 - |f|) */
        v[0] = (f < HP(0.0)) ? f_err : -f_err;
        v[1] = (hp_t)PI_HALF_LO;
        v[2] = -f_abs;
        v[3] = (hp_t)PI_HALF_HI;
        f = HP_VSUM(&f_err, 4, v);

        if (negate)
        {
            f = -f;
            f_err = -f_err;
        }

        break;

    case 1: /* cos(x) = -sin(f) = sin(-f) */
        f = -f;
        f_err = -f_err;
        break;

    case 3: /* cos(x) = sin(f) */
        break;
    }

    f_abs = QABS(f);

    /* ... code omitted ... */
}
```

The value of f at the start of the `switch` statement is restricted to $[-\frac{1}{4}\pi, \frac{1}{4}\pi]$, and argument symmetry allows replacement of $\cos(-|f|)$ by $\cos(|f|)$, so the subtraction $\frac{1}{2}\pi - |f|$ cannot suffer loss of leading bits. However, we need a value of $\frac{1}{2}\pi$ to twice working precision in order to recover an accurate difference, and the `VSUM()` family that we used in **Section 10.2** on page 273 does the job. We include the error term in the vector sum, and recover a new `f` and `f_err` representing the further-reduced argument to twice working precision.

The remainder of the code then uses a two-term Taylor series for $\sin(f)$ if $|f|$ is small, and otherwise, computes a rational approximation. Just as we did in `sinx.h`, for $|f| > \sin^{-1}(0.9)$, we use the alternate formula that computes $\sin(f)$ as a small negative correction to the exact value 1. In both cases, the `VSUM()` family allows accurate computation of the reduced argument $\frac{1}{2}|f| - \frac{1}{4}\pi$ as $\frac{1}{2}(|f| - \frac{1}{2}\pi)$.

From **Table 4.1** on page 62, the error-magnification factor for the sine is $x/\tan(x)$, and its value lies in $[1, 0]$ for x in $[0, \frac{1}{2}\pi]$. We therefore expect high accuracy for the sine function over that range.

When argument reduction is needed, we have $x = n\pi + r$, and the value returned by `RP()` is $\mathrm{fl}(r)$, differing from the exact r by at most one rounding error. We then compute the sine of that rounded value, when instead, we want

$\sin(r)$. `RP()` returns an estimate of the argument error, so we need to determine how to use it to correct the values of $\sin(\mathrm{fl}(r))$ and $\cos(\mathrm{fl}(r))$. The Taylor series once again provide answers:

$$\cos(r_{\mathrm{hi}} + r_{\mathrm{lo}}) = \cos(r_{\mathrm{hi}}) - \sin(r_{\mathrm{hi}})r_{\mathrm{lo}} - \tfrac{1}{2}\cos(r_{\mathrm{hi}})r_{\mathrm{lo}}^2 + \mathcal{O}(r_{\mathrm{lo}}^3),$$
$$\sin(r_{\mathrm{hi}} + r_{\mathrm{lo}}) = \sin(r_{\mathrm{hi}}) + \cos(r_{\mathrm{hi}})r_{\mathrm{lo}} - \tfrac{1}{2}\sin(r_{\mathrm{hi}})r_{\mathrm{lo}}^2 - \mathcal{O}(r_{\mathrm{lo}}^3).$$

When r is tiny, $\cos(r_{\mathrm{hi}}) \approx 1$, so the correction to the sine of the accurate reduced argument is almost exactly r_{lo}, which is enough to perturb the computed answer by as much as one ulp. Similarly, when r is near an odd integer multiple of $\frac{1}{2}\pi$, $\sin(r_{\mathrm{hi}}) \approx \pm 1$, and the correction to the cosine can be about one ulp.

We expect the argument reduction to contribute no more than *one* additional rounding error. That means that if our polynomial approximation for x on $[-\frac{1}{2}\pi, \frac{1}{2}\pi]$ is good enough, and we have *round-to-nearest* arithmetic, then we expect errors below 0.5 ulps on that central interval, and no more than 1 ulp for all other possible values of x over the entire floating-point range.

Those expectations are confirmed by error plots from a version of our code that uses working precision for the internal computations. However, on all modern systems, there is relatively little difference in performance between `float` and `double` types, and for the common Intel (and compatible) processor families, between those and `long double`. Thus, we implement the kernel of the sine and cosine functions in the next higher precision, `hp_t`, although the polynomial approximations are still selected according to the working precision, `fp_t`. A compile-time definition of the macro `USE_FP_T_KERNEL` forces the kernel computation back to working precision, should that be desirable for performance reasons on some systems.

The error-magnification factor for $\cos(x)$ is $x\tan(x)$. For x in $[-1, 1]$, that factor lies in $[0, 1.558]$, but it rises sharply to infinity for $|x|$ near $\frac{1}{2}\pi$. We therefore expect larger errors in the cosine than in the sine, and that is indeed observed when the kernel code is in working precision.

The errors in our implementation of the cosine and sine functions are presented in **Figure 11.4** on the next page and **Figure 11.5** on page 312. Although the argument range in those plots is restricted to allow the behavior in the reduction-free region to be seen, plots over a wider range are qualitatively similar.

11.5 Computing the tangent

The Cody/Waite procedure for the tangent reduces the argument to the range $[-\frac{1}{4}\pi, +\frac{1}{4}\pi]$ using a technique similar to that for the sine and cosine described on page 245. They handle tiny arguments with the first term of the Taylor series, $\tan(r) \approx r$. Otherwise, they use an approximation of the form

$$g = r^2, \qquad \tan(r) \approx r\mathcal{P}(g)/\mathcal{Q}(g),$$

valid for r in $[0, \frac{1}{4}\pi]$. The corresponding range of the factor $\mathcal{P}(g)/\mathcal{Q}(g)$ is about $[1, 1.28]$.

That form differs from that of their approximation for $\sin(r)$ by the absence of a leading term r. The reason for that change is that it makes it possible to provide a separate routine for the cotangent that simply computes $\mathrm{cotan}(r) \approx \mathcal{Q}(g)/(r\mathcal{P}(g))$. That avoids introduction of an additional rounding error if the cotangent were computed as $1/\tan(r)$, and allows the same polynomial approximations to be used for both functions.

Unfortunately, the fact that the function value is not obtained as the sum of an exact value and a small correction produces larger errors than we would like. Error plots for an implementation of their algorithm show peaks of up to 1.5 ulps, with occasional errors over 2.5 ulps. The error-magnification factor for the tangent (see **Table 4.1** on page 62) grows with increasing x: for x on $[0, \frac{1}{4}\pi]$, its range is $[1, \frac{1}{2}\pi]$, but for x on $[\frac{1}{4}\pi, \frac{1}{2}\pi]$, its range is $[\frac{1}{2}\pi, \infty]$. Higher precision is essential for reducing that unavoidable mathematical error magnification. We therefore entirely replace the Cody/Waite algorithm by a similar one, with a two-term Taylor series for tiny arguments, and a polynomial approximation of the form used for the sine:

$$g = r^2, \qquad \tan(r) \approx r + rg\mathcal{P}(g)/\mathcal{Q}(g).$$

We also use the Taylor series to find out how to incorporate a correction from the argument reduction:

$$\begin{aligned}\tan(r_{\mathrm{hi}} + r_{\mathrm{lo}}) = \tan(r_{\mathrm{hi}}) + (1 + \tan(r_{\mathrm{hi}}))r_{\mathrm{lo}} + {}\\ \tan(r_{\mathrm{hi}})(1 + (\tan(r_{\mathrm{hi}}))^2)r_{\mathrm{lo}}^2 + \mathcal{O}(r_{\mathrm{lo}}^3).\end{aligned}$$

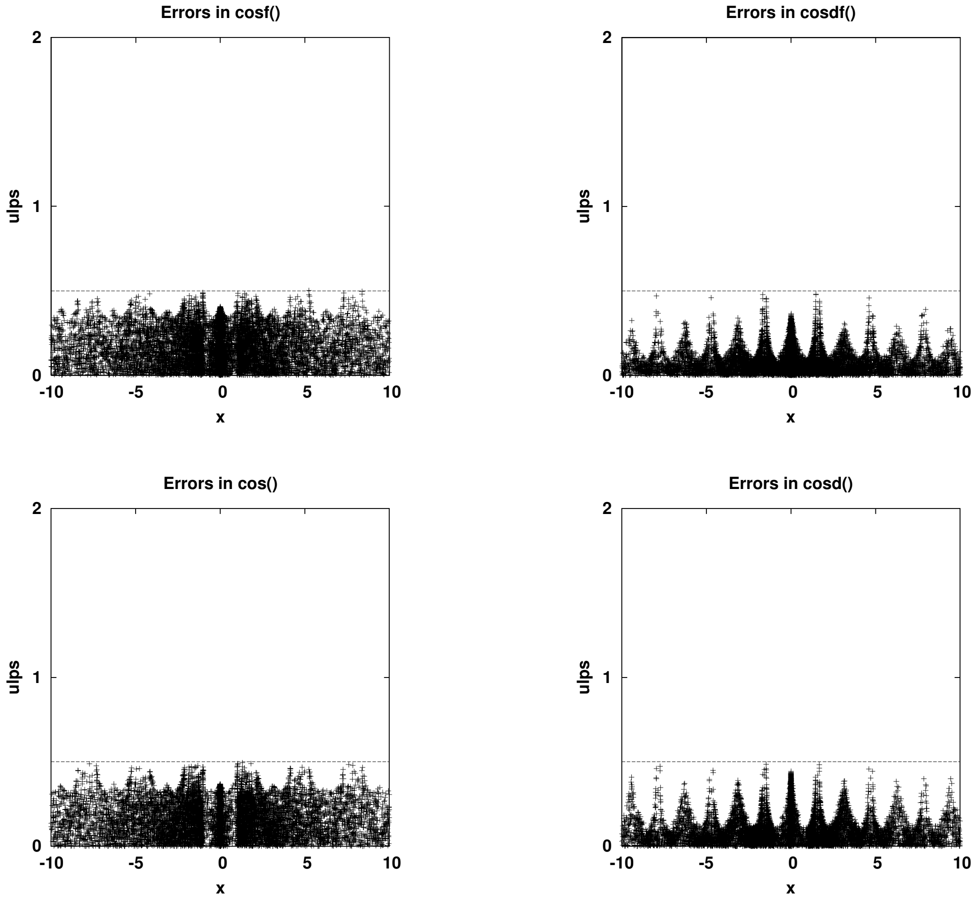


Figure 11.4: Errors in `COS()` functions.

We use the next higher precision for the argument reduction and evaluation of the polynomial approximation. The core of our tangent routine, after arguments of NaN, Infinity, and signed zero have been handled, looks like this:

```
hp_t a, f, f_err;
int n;
volatile hp_t t, u;

f = HP_RPH(x, &n, &f_err);

if (ISNAN((fp_t)f))             /* |x| too big to reduce accurately */
    result = SET_EDOM((fp_t)f);
else
{
    if (QABS((fp_t)f) < EPS)    /* tan(f) = f + f**3/3 for tiny |f| */
        u = f * f * f / THREE;
    else                        /* tan(f) = f + f * g * P(g)/Q(g) */
    {
        hp_t g, pg_g, qg;

        g = f * f;
        pg_g = POLY_P(p, g) * g;
        qg   = POLY_Q(q, g);
```

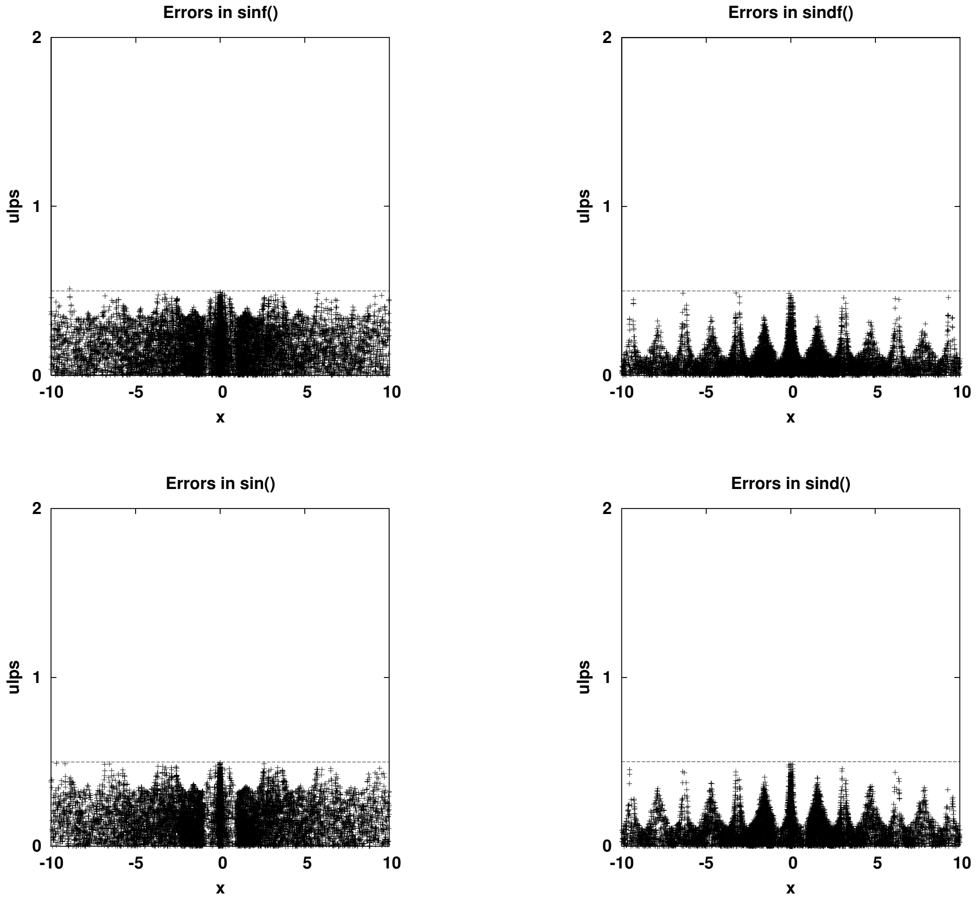


Figure 11.5: Errors in `SIN()` functions.

```
        u = f * (pg_g / qg);
    }

    STORE((fp_t *)&u);
    a = f + u;                  /* initial estimate of tan(f) */

    if (QABS(a) < HP(1.0))
    {
        t = u + a * f_err;
        STORE((fp_t *)&t);
        t += f_err;
        STORE((fp_t *)&t);
    }
    else                      /* reverse terms to add largest last */
    {
        t = u + f_err;
        STORE((fp_t *)&t);
        t += a * f_err;
        STORE((fp_t *)&t);
    }
```

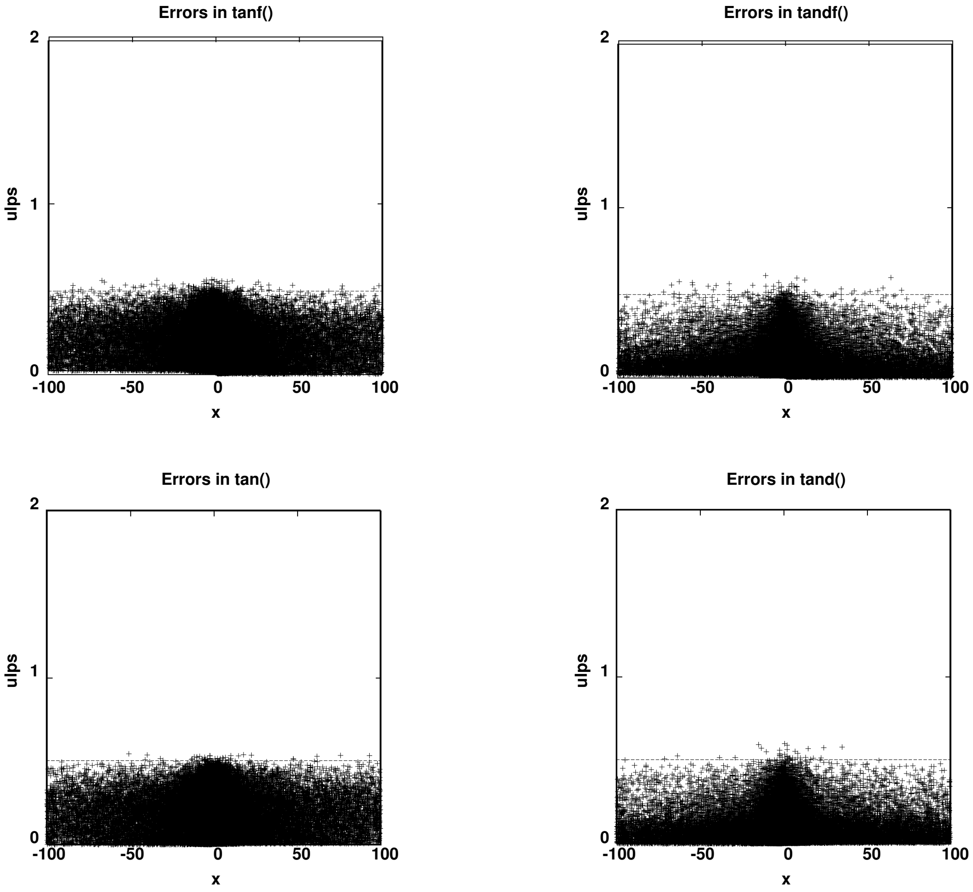


Figure 11.6: Errors in `TAN()` functions.

```
        t += f;                  /* refined value of tan(f) */
        result = (fp_t)(IS_EVEN(n) ? t : -HP(1.0) / t);
    }
```

We also provide library routines for the cotangent. The code in `cotanx.h` differs from that in `tanx.h` only in the handling of the case $x = \pm 0$, and in using the reciprocals of the final expressions before the cast to `fp_t` is applied. The errors in our implementation of the tangent are shown in **Figure 11.6**. Plots of the errors in the cotangent are similar, and thus, not shown.

Tests of our code for the tangent and cotangent against high-precision computations in Maple with logarithmically distributed random arguments show that about one in 2400 results fails to be correctly rounded, with a worst-case error of about 0.62 ulps.

11.6 Trigonometric functions in degrees

We observed in **Section 11.3** on page 304 that arguments in degrees can be exactly reduced to a single period with the help of the `fmod()` function family. It might appear that all that is then required is to call the corresponding normal trigonometric function with an argument scaled from degrees to radians by multiplication with $\pi/180$. Unfortunately, that introduces a problem that we discuss in relation to **Table 11.2** on page 305: trigonometric arguments in degrees can exactly locate zeros, inflection points, and poles, but nonzero arguments in radians cannot. We can see that in numerical experiments with the 32-bit binary and decimal versions of hoc:

```
% hoc32
hoc32> for (x = 0; x <= 360; x += 45) \
hoc32>     printf("%3d  % .9g\n", x, sin(x * (PI / 180)))
  0   0
 45   0.707106769
 90   1
135   0.707106769
180  -8.74279067e-08
225  -0.707106709
270  -1
315  -0.707106888
360   1.74855813e-07

% hocd32
hocd32> for (x = 0; x <= 360; x += 45) \
hocd32>     printf("%3d  % .7g\n", x, sin(x * (PI / 180)))
  0   0
 45   0.7071067
 90   1
135   0.7071071
180   6.535898e-07
225  -0.7071062
270  -1
315  -0.7071076
360  -1.30718e-06
```

The expected output is $0, \sqrt{\frac{1}{2}}, 1, \sqrt{\frac{1}{2}}, 0, -\sqrt{\frac{1}{2}}, -1, -\sqrt{\frac{1}{2}}, 0$, but we got tiny numbers instead of zeros, and we lost important symmetry relations.

The solution to that problem is special handling of arguments at zeros, poles, and inflection points, and in general, arguments that are multiples of 45°. In addition, for both the cosine and sine, we reduce the computation to evaluation of a sine function, so that we can ensure that $\sin(x) \approx x$ when x is small.

We show here only the code for `COSDEG()`, because the code required for the other functions is analogous:

```
fp_t
COSDEG(fp_t x)
{
    fp_t result;

    if (ISNAN(x))
        result = SET_EDOM(x);
    else if (ISINF(x))
        result = SET_EDOM(QNAN(""));
    else
    {
        fp_t r, xabs;

        xabs = FABS(x);
        r = (xabs < FP(360.0)) ? xabs : FMOD(xabs, FP(360.0));

        if (r == ZERO)
            result = ONE;
        else if (r == FP(45.0))
            result = SQRT_HALF;
        else if (r == FP(90.0))
            result = ZERO;
        else if (r == FP(135.0))
            result = -SQRT_HALF;
        else if (r < FP(180.0))
```

```
            result = -(fp_t)HP_SIN(DEG_TO_RAD * ((hp_t)r - HP(90.0)));
        else if (r == FP(180.0))
            result = -ONE;
        else if (r == FP(225.0))
            result = -SQRT_HALF;
        else if (r == FP(315.0))
            result = SQRT_HALF;
        else
            result = (fp_t)HP_SIN(DEG_TO_RAD * ((hp_t)r - HP(270.0)));
    }

    return (result);
}
```

Argument scaling uses higher precision to make correct rounding more likely, and the arguments are shifted relative to the zeros of the cosine at 90° and 270° before scaling. We take the simpler approach of using higher-precision sine functions, but with a bit more code, we could use working precision, and make a small correction for the argument error.

11.7 Trigonometric functions in units of π

We mentioned in **Section 11.3** on page 304 that a few vendors provide trigonometric functions whose arguments are units of π. They are defined like this:

$$\begin{aligned} \mathrm{cospi}(x) &= \cos(\pi x), & \mathrm{cotanpi}(x) &= \mathrm{cotan}(\pi x), \\ \mathrm{sinpi}(x) &= \sin(\pi x), & \mathrm{tanpi}(x) &= \tan(\pi x). \end{aligned}$$

Functions of that kind are uncommon, but they do show up in some applications, notably, the gamma and psi functions that we treat in **Chapter 18** on page 521. Like their companions with degree arguments, they can exactly locate zeros, inflection points, and poles.

Their accurate computation contains several pitfalls, so we describe here how they are computed in the mathcw library.

The first point to note is that the difficult problem of trigonometric argument reduction is completely eliminated in those functions. No matter how large x is, we can always use the `MODF()` family to split it into an *exact* sum of a whole number and a fraction, for which we have two flavors of argument reduction:

$$\begin{aligned} \tfrac{1}{2}x &= n + r & x &= n + r, \\ x &= 2(n + r), & & \\ \mathrm{cospi}(x) &= \cos(2\pi(n + r)) & \mathrm{cotanpi}(x) &= \mathrm{cotan}(\pi(n + r)) \\ &= \cos(2\pi r) & &= \mathrm{cotan}(\pi r) \\ &= \mathrm{cospi}(2r), & &= \mathrm{cotanpi}(r), \\ \mathrm{sinpi}(x) &= \sin(2\pi r) & \mathrm{tanpi}(x) &= \tan(\pi(n + r)) \\ &= \mathrm{sinpi}(2r), & &= \mathrm{tanpi}(r). \end{aligned}$$

The catch for the left-hand set is that multiplication and division by two must always be exact, and that is only the case in binary arithmetic with $\beta = 2$. There is no such problem for the cotangent and tangent: there, the base does not matter.

For arguments above β^t, the last of the contiguous sequence of positive integers starting at zero that is exactly representable in floating-point arithmetic, we have $x = 2n$ or $x = n$, and $r = 0$. We then have special cases that allow early exit for large argument magnitudes:

$$\begin{aligned} \mathrm{cospi}(x) &= 1, & \mathrm{cotanpi}(x) &= \mathrm{copysign}(\infty, x), \\ \mathrm{sinpi}(x) &= \mathrm{copysign}(0, x), & \mathrm{tanpi}(x) &= \mathrm{copysign}(0, x). \end{aligned}$$

Three of them are odd functions, so they acquire the sign of their argument. The same relations hold for smaller $x = n$ when $r = 0$, except that the cosine is then $(-1)^n$. For $x \geq \beta^t$, n is even because all practical computer-arithmetic systems have even values of β.

By using the function symmetry relations, we can restrict ourselves now to considering only the case of r in $(0, 1)$. It might appear then that all we need to do after the initial reduction is to call a standard function with a scaled argument, like this: `tan(PI * r)`. Unfortunately, that produces poor results, as we can see from simple numerical calculations in decimal arithmetic with three precisions in hoc, and with higher precision in Maple:

```
% hocd128
hocd128> x = 0.999_999_9
hocd128> single(tan(single(single(PI) * x))); \
         double(tan(double(double(PI) * x))); \
         tan(PI * x)
 3.464_102e-07
-3.141_592_652_384_73e-07
-3.141_592_653_589_896_592_718_244_382_322_819e-07

hocd128> x = 0.500_000_1
hocd128> single(tan(single(single(PI) * x))); \
         double(tan(double(double(PI) * x))); \
         tan(PI * x)
-1_485_431
-3_183_098.861_617_03
-3_183_098.861_837_801_995_622_555_604_656_751

% maple
> printf("%.45e\n", evalf(tan(Pi * 0.9999999), 60));
-3.141_592_653_589_896_592_718_244_386_760_350_269_051_308_781_e-07

> printf("%.38f\n", evalf(tan(Pi * 0.5000001), 60));
-3_183_098.861_837_801_995_622_555_606_986_643_448_569_747_78
```

The value of x is one seven-digit ulp below 1, or above $\frac{1}{2}$. The function value at the first x is tiny, and at the second, is close to an asymptote, and thus large.

For the first test, single-precision computation gets the correct magnitude, but the sign is wrong, and only the leading digit is correct. The 16-digit format gets the sign right, but only the first ten digits are correct. The 34-digit format has only 27 correct digits. Results for the second test are hardly better.

Those errors are far too large, and we have to work hard to reduce them.

11.7.1 Cosine and sine in units of π

Despite the absence of asymptotes in the cosine and sine functions, `cospi()` and `sinpi()`, it is hard to produce correctly rounded results for them without access to higher precision. This author experimented with several different function approximations, all of which had errors above 1 ulp, until, finally, a set was found that is reasonably satisfactory.

The first step is exact reduction of the argument to the interval $[0, 1]$ using the `MODF()` family and the symmetry relations $\cos(-|x|) = \cos(|x|)$ and $\sin(-|x|) = -\sin(|x|)$. We then need to evaluate the standard trigonometric functions only on the radian interval $[0, \pi]$. However, the constant π has two leading zero bits in systems with hexadecimal normalization, introducing further unwanted argument error. The leading zero bits can be eliminated if we work instead with $\frac{1}{4}\pi$, and use the private auxiliary functions

$$\operatorname{cospi4}(w) = \cos(\tfrac{1}{4}\pi w), \qquad \operatorname{sinpi4}(w) = \sin(\tfrac{1}{4}\pi w), \qquad \text{for } w \text{ on } [0, 1].$$

We then require the reductions

$$\begin{aligned} x &= m + f, && \text{for whole number } m \text{ and fraction } f \text{ in } (0, 1], \\ \cos(\pi x) &= \cos(\pi(m + f)), \end{aligned}$$

$$
\begin{aligned}
&= \cos(\pi m)\cos(\pi f) - \sin(\pi m)\sin(\pi f), && \textit{by addition rule,}\\
&= (-1)^m \cos(\pi f), && \textit{because } \sin(\pi m) = 0,\\
&= (-1)^m \operatorname{cospi4}(4f),\\
\sin(\pi x) &= \sin(\pi(m+f)),\\
&= \sin(\pi m)\cos(\pi f) + \cos(\pi m)\sin(\pi f), && \textit{by addition rule,}\\
&= (-1)^m \sin(\pi f), && \textit{because } \sin(\pi m) = 0,\\
&= (-1)^m \operatorname{sinpi4}(4f).
\end{aligned}
$$

The multiplications by four are exact only when $\beta = 2$ or $\beta = 4$. For other bases, we require yet another reduction:

$$
\begin{aligned}
4f &= u + v, && \textit{for whole number } u \textit{ in } [0,3] \textit{ and fraction } v \textit{ in } (0,1],\\
f &= u/4 + d, && \textit{for } d \textit{ in } [0, \tfrac{1}{4}],\\
v &= 4d, && \textit{exact in any integer base.}
\end{aligned}
$$

In practice, we employ that reduction for all bases, because it produces a useful simplification that we discover shortly.

Another application of the trigonometric argument addition rules allows further decomposition, because for $u = 0, 1, \ldots, 7$, the factor $\sin(\frac{1}{4}\pi u)$ takes on the values 0, $\sqrt{1/2}$, 1, $\sqrt{1/2}$, 0, $-\sqrt{1/2}$, -1, and $-\sqrt{1/2}$. Those constants then repeat cyclically for larger values of u. Similarly, the factor $\cos(\frac{1}{4}\pi u)$ cycles through the values 1, $\sqrt{1/2}$, 0, $-\sqrt{1/2}$, -1, $-\sqrt{1/2}$, 0, and $\sqrt{1/2}$. With some careful study, and more applications of the addition rules, the constants can be eliminated to produce these results:

$$
\cos(\tfrac{1}{4}\pi(u+v)) = \begin{cases}
+\cos(\frac{1}{4}\pi v), & \textit{if } u \bmod 8 = 0,\\
+\sin(\frac{1}{4}\pi(1-v)), & \textit{if } u \bmod 8 = 1,\\
-\sin(\frac{1}{4}\pi v), & \textit{if } u \bmod 8 = 2,\\
-\cos(\frac{1}{4}\pi(1-v)), & \textit{if } u \bmod 8 = 3,\\
-\cos(\frac{1}{4}\pi v), & \textit{if } u \bmod 8 = 4,\\
-\sin(\frac{1}{4}\pi(1-v)), & \textit{if } u \bmod 8 = 5,\\
+\sin(\frac{1}{4}\pi v), & \textit{if } u \bmod 8 = 6,\\
+\cos(\frac{1}{4}\pi(1-v)), & \textit{if } u \bmod 8 = 7,
\end{cases}
$$

$$
\sin(\tfrac{1}{4}\pi(u+v)) = \begin{cases}
+\sin(\frac{1}{4}\pi v), & \textit{if } u \bmod 8 = 0,\\
+\cos(\frac{1}{4}\pi(1-v)), & \textit{if } u \bmod 8 = 1,\\
+\cos(\frac{1}{4}\pi v), & \textit{if } u \bmod 8 = 2,\\
+\sin(\frac{1}{4}\pi(1-v)), & \textit{if } u \bmod 8 = 3,\\
-\sin(\frac{1}{4}\pi v), & \textit{if } u \bmod 8 = 4,\\
-\cos(\frac{1}{4}\pi(1-v)), & \textit{if } u \bmod 8 = 5,\\
-\cos(\frac{1}{4}\pi v), & \textit{if } u \bmod 8 = 6,\\
-\sin(\frac{1}{4}\pi(1-v)), & \textit{if } u \bmod 8 = 7.
\end{cases}
$$

Our decision to use the reduction $4f = u + v$ for all bases reduces u from the range $[0,7]$ to $[0,3]$, so we only need the first four of each of the eight cases. However, we may require either cospi4() or sinpi4() with exact arguments v or $1 - v$ to find cospi(x), and similarly for sinpi(x).

Here is how we evaluate the two auxiliary functions:

$$
\begin{aligned}
r &= \tfrac{1}{4}\pi x, && \textit{scaled argument in radians},\\
\operatorname{cospi4}(x) &= \cos(r)\\
&\approx \begin{cases} 1 + r^2\mathcal{P}(x^2), & \textit{for } x \textit{ in } [0, \frac{1}{2}],\\ 25/32 + \mathcal{Q}(x^2), & \textit{for } x \textit{ in } [\sqrt{\frac{1}{5}}, 1],\end{cases}\\
\operatorname{sinpi4}(x) &= \sin(r)
\end{aligned}
$$

$$\approx \begin{cases} r + r^3\mathcal{R}(x^2), & \text{for } x \text{ in } [0,1], \\ r - r^3/6 + r^5/120 - r^7/5040 + r^9/362\,880, & \text{Taylor series.} \end{cases}$$

The polynomial approximations $\mathcal{P}(x^2)$, $\mathcal{Q}(x^2)$, and $\mathcal{R}(x^2)$ are Chebyshev fits, for which the 32-bit floating-point formats require only four or five terms. They are defined in terms of x^2, rather than r^2, to avoid introducing additional error into their arguments. Each approximation formula computes the function as the sum of an exact term and a correction that is never much bigger than about 10% of the result, and the sum is free of subtraction loss.

There is a small overlap in the two regions for cospi4(x) because $\sqrt{\frac{1}{5}} \approx 0.447$. That peculiar endpoint was selected to simplify the conversion to the Chebyshev polynomial variable and reduce its error. The Chebyshev variable $t = \frac{5}{2}x - \frac{3}{2}$ lies on $[-1, +1]$. The correction to the first term is about twice as large in the overlap region with the second formula, so we use it only for $x > \frac{1}{2}$.

The scaled argument r is represented as a sum of two parts computed like this:

$$\begin{aligned} \mathrm{fl}(r) &= \mathrm{fma}((\tfrac{1}{4}\pi)_{\mathrm{hi}}, x, (\tfrac{1}{4}\pi)_{\mathrm{lo}} \times x), \\ \delta &= \mathrm{fma}((\tfrac{1}{4}\pi)_{\mathrm{hi}}, x, -\mathrm{fl}(r)) + (\tfrac{1}{4}\pi)_{\mathrm{lo}} \times x. \end{aligned}$$

As usual, the constant is split so that the high part is exact to working precision, and the low part is rounded to working precision, so that their sum represents $\frac{1}{4}\pi$ to twice working precision. The computed value $\mathrm{fl}(r)$ is then almost certainly the floating-point number closest to the exact product.

The correction $\delta \leq \epsilon r$ is included only to first order in each of the remaining formulas for powers of r up to three:

$$r \approx \mathrm{fl}(r) + \delta, \qquad r^2 \approx \mathrm{fl}(r)^2 + 2\delta\,\mathrm{fl}(r), \qquad r^3 \approx \mathrm{fl}(r)^3 + 3\delta\,\mathrm{fl}(r)^2.$$

Sums of terms are computed in order of increasing magnitudes to minimize the final rounding error.

The Taylor series is used for small arguments, and evaluated in Horner form, but modified to include the δ corrections. In addition, because r and its higher powers can be subnormal, or underflow to zero, when x is tiny but normal, terms in the Taylor series are scaled by $1/\epsilon^3$ to keep the numerically important parts of the computation away from the underflow region. That ensures accurate results after unscaling by ϵ^3, even when x is tiny or subnormal.

Numerical experiments show that a five-term Taylor series for $\sin(r)$ is more accurate for small arguments than the polynomial alternative, and using fewer than five terms results in errors above the target of $\frac{1}{2}$ ulp.

The files `cospix.h` and `sinpix.h` contain the code that handles the usual argument checking, and does the outer argument reductions. Their header files include the shared header file `scpi4.h` that defines the Chebyshev polynomial coefficients, and the code for the private functions `cospi4()` and `sinpi4()`. Those functions are defined only for a restricted argument range, and do no argument checking, so they are not directly callable from user code.

Extensive testing against high-precision computation in Maple shows that the results produced by our `COSPI()` and `SINPI()` family are correctly rounded in binary arithmetic, except for about 1 in 2000 random arguments taken from a logarithmic distribution in $[0,1]$. In decimal arithmetic, the error rate is about $1/200$ for the sine, but the cosine appears to be perfectly rounded. The maximum error is below 0.82 ulps. Error plots are therefore omitted.

11.7.2 Cotangent and tangent in units of π

For the tangent, we can find approximating functions on the interval $(0,1)$ by first examining series expansions analytically and numerically:

```
% maple
> taylor(tan(Pi * r), r = 0, 8);
           3          5           7
         Pi   3   2 Pi   5   17 Pi   7      9
  Pi r + --- r  + ----- r  + ------ r  + O(r )
          3        15         315


> evalf(%, 6);
                     3            5            7      9
3.14159 r + 10.3354 r  + 40.8026 r  + 163.000 r  + O(r )
```

```
> series(tan(Pi * (1/2 + h)), h = 0, 8);
                          3          5        7
   1    -1    Pi      Pi   3   2 Pi   5   Pi    7      9
- ---- h   + ---- h + --- h  + ----- h  + ---- h  + O(h )
   Pi         3      45        945       4725

> evalf(%, 6);
            -1                         3            5
  -0.318310 h   + 1.04720 h + 0.689026 h  + 0.647656 h  +

              7      9
     0.639212 h  + O(h )
```

The series coefficients are complicated by Bernoulli numbers, and not easily generated by a recurrence.

The first series is usable for tanpi(x) for $x \approx 0$, and also for $x \approx 1$, because we can use the angle-sum formula with $r = 1 + d$ to find

$$\begin{aligned}\tan(\pi(1+d)) &= \frac{\tan(\pi) + \tan(\pi d)}{1 + \tan(\pi)\tan(\pi d)} \\ &= \tan(\pi d) \\ &\approx \pi d + d^3\mathcal{G}(d^2), &&\text{\textit{first polynomial fit.}}\end{aligned}$$

The expansion coefficients in the first series are large, so direct summation using a stored table of coefficients is unwise.

The coefficients in the second series do not decrease much, even at high order, so direct series summation is inadvisable. Instead, we use the approximation

$$\tan(\pi(\tfrac{1}{2} + h)) \approx \frac{-1}{\pi h} + h\mathcal{H}(h^2), \qquad \text{\textit{second polynomial fit,}}$$

with $h = r - \frac{1}{2}$, which is also exact.

A Chebyshev fit of the function $\mathcal{H}(h^2)$ for h on the interval $[-\frac{1}{4}, \frac{1}{4}]$ needs only 4 terms for 7-digit accuracy, 9 terms for the 16-digit format, and at most 40 terms for the 70-digit format. Minimax fits could do even better, but need to be customized for each precision, whereas a single Chebyshev table, suitably truncated, serves for all precisions and bases.

Alternatively, we can exploit an identity to reuse the first polynomial fit, at the cost of an extra division:

$$\begin{aligned}\tan(\pi(\tfrac{1}{2} + h)) &= -1/\tan(\pi h) \\ &\approx -1/(\pi h + h^3\mathcal{G}(h^2)), &&\text{\textit{first polynomial fit.}}\end{aligned}$$

However, that gives us an extra rounding error that we wish to avoid, so we use two Chebyshev fits instead of one.

Away from the regions of $x \approx 0$ and $x \approx 1$, the tangent of a scaled argument can be safely used, and we do so in the code in `tanpix.h`. However, by choosing the ranges of d and h each to be $[-\frac{1}{4}, +\frac{1}{4}]$, the two polynomial fits cover the complete range of r on $(0, 1)$, and with the argument reduction from x, and the symmetry relation $\tan(-x) = -\tan(x)$, we can extend coverage to the entire real axis.

The rounding errors associated with representing the transcendental constant $\pi = \pi_{\text{hi}} + \pi_{\text{lo}}$, and products with it, require careful handling in finite-precision arithmetic. For x near the underflow limit, scaling is required to prevent the lower product $\pi_{\text{lo}}x$ from underflowing into the subnormal region, or to zero. The code in `tanpix.h` for a three-term series expansion of $\tan(\pi x)$ looks like this:

```
static fp_t
tan_0_series(fp_t x)
{   /* tan(Pi * x) = tan(Pi * (1 + x)) for tiny |x| */
    fp_t err_scaled, g, gg, g_scaled, result, sum, x_scaled;
    static int do_init = 1;
    static fp_t SCALE = FP(1.0);
    static fp_t SCALE_INV = FP(1.0);

    if (do_init)
    {   /* some systems do not allow these as compile-time constants */
        SCALE_INV = FP_T_EPSILON * FP_T_EPSILON * FP_T_EPSILON;
        SCALE = ONE / SCALE_INV;
        do_init = 0;
    }

    x_scaled = x * SCALE;           /* EXACT */
    g_scaled = FMA(PI_HI, x_scaled, PI_LO * x_scaled);
    err_scaled = FMA(PI_HI, x_scaled, -g_scaled) + PI_LO * x_scaled;    /* scaled rounding error in g */
    g = g_scaled * SCALE_INV;       /* EXACT */
    gg = g * g;

    sum =            FP(62.0) / FP(2835.0);
    sum = sum * gg + FP(17.0) / FP( 315.0);
    sum = sum * gg + FP( 2.0) / FP(  15.0);
    sum = sum * gg + FP( 1.0) / FP(   3.0);
    sum = sum * gg;
    sum = sum * g_scaled + err_scaled;
    result = (g_scaled + sum) * SCALE_INV;

    return (result);
}
```

Error compensation is needed in the second series expansion as well, but we omit the code here. Extensive testing against high-precision computation in Maple shows that the `TANPI()` family appears to produce results that are always correctly rounded in the decimal functions. In the binary functions, only about one in 60 000 results is incorrectly rounded, and no error exceeds 0.53 ulps.

For the cotangent, instead of doing a new set of series expansions and polynomial fits, we can leverage two identities:

$$\begin{aligned} \operatorname{cotanpi}(r) &= -\operatorname{tanpi}(r \pm \tfrac{1}{2}), && \textit{true for both signs in the argument,} \\ &= 1/\operatorname{tanpi}(r). \end{aligned}$$

When $r \geq \frac{1}{4}$, the difference $r - \frac{1}{2}$ is exact, so the first formula requires only a negation of the tangent. For $r < \frac{1}{4}$, the argument in the first formula requires more precision than we have, but the second formula does not; the final division introduces a single rounding error. We can largely eliminate that error by computing the tangent and its reciprocal in the next higher precision. That means that our single-precision cotangent of units-of-π argument, and on many systems, our double-precision one as well, are almost always correctly rounded.

11.8 Computing the cosine and sine together

Because the cosine and sine are often required for the same argument, and because their computations are closely related, it may be desirable, and efficient, to have a function to return both of them. SOLARIS provides two such function families, with typical prototypes like these:

```
void sincos  (double x, double *s, double *c);
void sincosd (double x, double *s, double *c);
```

The GNU `-lm` math library provides just the `sincos()` family, with the same prototypes as on SOLARIS. OSF/1 provides functions of the same name, but returns the pair as a structure value:

```
double_complex sincos  (double x);
double_complex sincosd (double x);
```

Here too, for the functions with arguments in degrees, there are conflicts with the naming conventions of the new decimal floating-point functions.

A straightforward implementation of the sincos() function might look like this:

```
void
SINCOS(fp_t x, fp_t *s, fp_t *c)
{
    if (c != (fp_t *)NULL)
        *c = COS(x);

    if (s != (fp_t *)NULL)
        *s = SIN(x);
}
```

However, it is interesting to consider how the two functions could be computed more quickly together, because they share the same checks for Infinity, NaN, and signed zero arguments, and they require similar argument reductions.

The angle-addition formulas

$$\cos(x+d) = \cos(x)\cos(d) - \sin(x)\sin(d),$$
$$\sin(x+d) = \sin(x)\cos(d) + \cos(x)\sin(d),$$

provide a way to compute both functions with much of the work shared. If we tabulate nearly evenly spaced, and exactly representable, values x_k together with high-precision values of their sines and cosines, then once we have a reduced argument $r = |x| - n\pi$, we can quickly find the interval $[x_k, x_{k+1}]$ that contains r. If that interval is small, then the calculation $d = r - x_k$ is exact, and d is small enough that two short polynomial approximations can be used to compute $\sin(d)$ and $\cos(d)$. For the sine, we have $d \geq 0$, and both terms of the addition formula are positive. However, for the cosine, the condition $d > 0$ produces a subtraction loss. The solution, if r is not at a tabulated point, is to make d negative by moving into the following interval $[x_{k+1}, x_{k+2}]$, and that requires storage of one extra table entry.

There is, however, an additional problem: if $\sin(x_{k+1})$ is negative, then subtraction loss creeps back in. We can resolve that problem by reducing the interval $[0, \frac{1}{2}\pi]$ to $[0, \frac{1}{4}\pi]$ according to the formulas in **Table 9.1** on page 244:

```
r = RPH(x < ZERO ? -x : x, &n, &err);
rabs = QABS(r);
```

Once the argument reduction has been done, the sine and cosine can be obtained by evaluating two truncated Taylor series when $|r|$ is tiny, or by using the addition formulas. In both cases, we make use of the correction to the reduced argument.

The Taylor-series code adds the correction only to first order:

```
if (rabs < EPS)
{
    fp_t rr;

    rr = r * r;

    the_sin =                FP( 1.0) / FP(120.0);
    the_sin = the_sin * rr + FP(-1.0) / FP(  6.0);
    the_sin = the_sin * rr * r + delta;
    the_sin += r;

    the_cos =                FP(-1.0) / FP(720.0);
    the_cos = the_cos * rr + FP( 1.0) / FP( 24.0);
    the_cos = the_cos * rr + FP(-1.0) / FP(  2.0);
    the_cos = the_cos * rr - r * delta;
    the_cos += ONE;
}
```

The code for the addition formula looks like this:

```
else
{
    hp_t d, sum, xx;    /* xx is used in rcos() and rsin() macros */
    int k;

    k = (int)FLOOR(rabs * FOUR_OVER_PI_TIMES_N);

    if (rabs < x_sin_cos[k].x)
        k--;
    else if (rabs >= x_sin_cos[k+1].x)
        k++;

    d = (hp_t)rabs - x_sin_cos[k].x;                /* EXACT */
    d += (r < ZERO) ? -delta : delta;

    if ((d < HP(0.0)) && (k > 0))           /* rarely taken branch */
    {
        k--;
        d = (hp_t)rabs - x_sin_cos[k].x;    /* EXACT */
        d += (r < ZERO) ? -delta : delta;
    }

    sum = (hp_t)rsin(d) * C(k);
    sum += (hp_t)rcosm1(d) * S(k);
    sum += S(k);
    the_sin = (fp_t)sum;

    if (r < ZERO)
        the_sin = -the_sin;

    if ( (d != HP(0.0)) && (k < (N + 1)) )  /* maybe advance k */
    {
        if (rabs <= x_sin_cos[k+1].x)
        {   /* almost-always taken branch that makes |d| smaller  */
            k++;
        }

        d = (hp_t)rabs - x_sin_cos[k].x;    /* EXACT */
        d += (r < ZERO) ? -delta : delta;

        if ((d > HP(0.0)) && (k < (N + 1))) /* rarely taken branch */
        {
            k++;
            d = (hp_t)rabs - x_sin_cos[k].x;        /* EXACT */
            d += (r < ZERO) ? -delta : delta;
        }
    }

    sum  = -(hp_t)rsin(d) * S(k);
    sum += (hp_t)rcosm1(d) * C(k);
    sum += C(k);
    the_cos = (fp_t)sum;
}
```

The initial value of k may be off by 1 when $|r|$ is close to a tabulated x_k, so it may require adjustment up or down, and that change may need to be repeated after the correction to the reduced argument is applied.

The tabulated sines and cosines are in the next higher precision, and intermediate computations are done in that precision.

The macros rcos(), rcosm1(), and rsin() expand inline to short polynomials. For the limited argument range in which they are valid, numerical fits find no advantage of rational polynomials over single polynomials, so we can avoid a division. Our implementation uses $N = 64$, for which polynomials of orders 2, 5, and 11 produce 15, 33, and 72 correct digits, respectively. There are $N + 2$ values of x in the table, along with their sines and cosines. The first $N + 1$ values of x are at intervals of approximately $(\frac{1}{4}\pi)/N$, but rounded to working precision, so each host precision requires its own table. The last x value is $\mathrm{fl}(\frac{1}{4}\pi)$, so the interval $[x_N, x_{N+1}]$ is generally much smaller than the preceding ones.

The final function results are then easily generated from one of four cases, and a possible negation of the sine value:

```
switch (n % 4)
{
default:
case 0: result_cos =  the_cos; result_sin =  the_sin; break;
case 1: result_cos = -the_sin; result_sin =  the_cos; break;
case 2: result_cos = -the_cos; result_sin = -the_sin; break;
case 3: result_cos =  the_sin; result_sin = -the_cos; break;
}

if (x < ZERO)
    result_sin = -result_sin;
```

Extensive tests of SINCOS() against values computed in high-precision arithmetic in Maple show that the results are almost always correctly rounded for any finite representable argument, so we omit error plots. They look similar to those in Figure 11.4 on page 311 and Figure 11.5 on page 312.

Timing tests of SINCOS() relative to separate computation of SIN() and COS() show considerable variation across systems, and also a strong dependence on the argument range. In most cases, the combined approach wins.

11.9 Inverse sine and cosine

The Taylor series for the inverse sine and cosine functions are

$$\begin{aligned}\mathrm{asin}(x) &= x + (1/6)x^3 + (3/40)x^5 + (5/112)x^7 + \cdots + \\ &\qquad \big((2n)!/(4^n(n!)^2(2n+1))\big)x^{2n+1} + \cdots,\\ \mathrm{acos}(x) &= \tfrac{1}{2}\pi - \big(x + (1/6)x^3 + (3/40)x^5 + (5/112)x^7 + \cdots + \\ &\qquad \big((2n)!/(4^n(n!)^2(2n+1))\big)x^{2n+1} + \cdots\big).\end{aligned}$$

Some programming languages use the names arcsin(x) and arccos(x) instead, and mathematics literature often writes them as $\sin^{-1}(x)$ and $\cos^{-1}(x)$. The functions are graphed in Figure 11.7 on the following page.

The argument range of $\mathrm{acos}(x)$ is $[-1, +1]$, and the corresponding function range is $[\pi, 0]$. The argument range of $\mathrm{asin}(x)$ is $[-1, +1]$, and the function range is $[-\frac{1}{2}\pi, +\frac{1}{2}\pi]$. The limited argument range means that there is no serious argument-reduction problem for those functions.

The symmetry relations are

$$\begin{aligned}\mathrm{acos}(-x) &= \tfrac{1}{2}\pi + \mathrm{asin}(x),\\ \mathrm{asin}(-x) &= -\mathrm{asin}(x).\end{aligned}$$

The two functions are related by equations that allow the inverse cosine to be computed from the inverse sine:

$$\begin{aligned}\mathrm{acos}(x) &= 2\,\mathrm{asin}\Big(\sqrt{\tfrac{1}{2}(1-x)}\Big), && \textit{use for } x > \tfrac{1}{2},\\ &= \pi - 2\,\mathrm{asin}\Big(\sqrt{\tfrac{1}{2}(1+x)}\Big), && \textit{use for } x < -\tfrac{1}{2},\\ &= \tfrac{1}{2}\pi - \mathrm{asin}(x), && \textit{use for } |x| \le \tfrac{1}{2},\end{aligned}$$

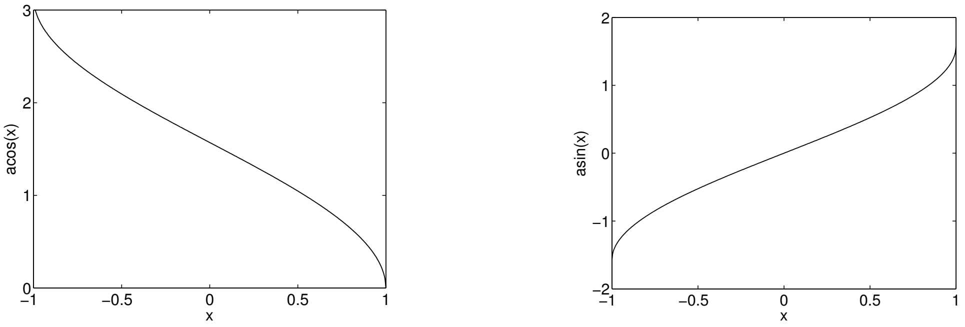


Figure 11.7: The inverse cosine and sine functions.

$$\operatorname{asin}(x) = \tfrac{1}{2}\pi - 2\operatorname{asin}\left(\sqrt{\tfrac{1}{2}(1-x)}\right), \qquad \textit{use for } |x| > \tfrac{1}{2}\textit{, but see text.}$$

The factors $(1 \pm x)$ lose leading digits in the regions where they are used, but as we describe shortly, they can nevertheless be computed exactly. The subtractions from π and $\frac{1}{2}\pi$ also suffer digit loss, and require considerably more care to program.

The error-magnification factors (see **Table 4.1** on page 62) for the two functions are proportional to $1/\sqrt{1-x^2}$, so both are sensitive to argument errors for $x \approx \pm 1$.

For a zero argument, Cody and Waite set $\operatorname{asin}(x) = x$ to preserve the sign of zero, and set $\operatorname{acos}(x) = \frac{1}{2}\pi$. For tiny $|x|$, they use just the leading term of the Taylor series, but we extend that to two terms, $\operatorname{asin}(x) \approx x + x^3/6$, to set the *inexact* flag and guarantee correct rounding. For $|x| \le \frac{1}{2}$, they use a rational polynomial approximation of the form

$$g = x^2, \qquad \operatorname{asin}(|x|) \approx |x| + |x|\mathcal{P}(g)/\mathcal{Q}(g),$$

for which the range of the factor $\mathcal{P}(g)/\mathcal{Q}(g)$ is about $[0, 0.048]$, giving a small correction to an exact value.

Our extensions for higher precision use a slightly different form for improved accuracy with the same computational effort:

$$\operatorname{asin}(|x|) \approx |x| + |x|g\mathcal{P}(g)/\mathcal{Q}(g).$$

For arguments $|x| > 1/2$, the inverse sine reduction requires computing $\sqrt{\frac{1}{2}(1-|x|)}$, which can be done accurately when there are guard digits for subtraction and multiplication, as is the case for all modern systems, by setting

$$g = \tfrac{1}{2}(1-|x|), \qquad r = \sqrt{g}.$$

The polynomial variable g is computed exactly in binary and decimal bases, and loses at most one bit in a hexadecimal base. With our accurate square root function, the reduced argument r should almost always be correctly rounded in binary and decimal bases.

When there are no guard digits, the computation of g requires more care:

$$g = \tfrac{1}{2}\left(\tfrac{1}{2} + \left(\tfrac{1}{2} - |x|\right)\right).$$

That assignment must be coded to prevent the compiler from replacing the constants $\frac{1}{2}$ by their sum at compile time.

Because $|x|$ is known to lie in $(\frac{1}{2}, 1]$, the innermost subtraction does not require a normalizing shift, so no digits are lost before the difference is computed. If there is a guard digit for multiplication, the division by two can be replaced by a multiplication by a half. Historical systems vary in their provision of guard digits, so for them, our code assumes the worst case, and does the outer division.

The constant $\frac{1}{2}\pi$ loses three bits of precision in a hexadecimal base, so we replace expressions of the form $\frac{1}{2}\pi + z$ by $\frac{1}{4}\pi + (\frac{1}{4}\pi + z)$, because $\frac{1}{4}\pi$ has no leading zero bits with hexadecimal normalization.

To avoid further accuracy loss, Cody and Waite preserve the exact computation of g by making the code for the two functions be simple wrappers that invoke a common routine with a distinguishing additional argument:

```
fp_t
ACOS(fp_t x)
{
    return (ACOSSIN(x, 1));
}


fp_t
ASIN(fp_t x)
{
    return (ACOSSIN(x, 0));
}
```

The common code in `ACOSSIN()` is intended for internal library use only, so its prototype is absent from the `mathcw.h` header file, and the macro name `ACOSSIN()` expands to an implementation-reserved name, `__acs()`, with the usual type suffix letters.

Testing of an implementation of the Cody/Waite algorithm for the inverse sine and cosine shows peaks in the error plots up to about 2 ulps when $|x| \approx \frac{1}{2}$, and investigations of alternatives lead to small, but significant, changes in the original algorithm:

- Reduce the error peaks by extending the interval of the polynomial approximation from $[0, \frac{1}{2}]$ to $[0, \sqrt{\frac{1}{2}}] \approx [0, 0.707]$, using the representation $\operatorname{asin}(r) \approx r + rg\mathcal{P}(g)/\mathcal{Q}(g)$. The larger range increases the total polynomial degree by at most two.
- Do not form $\operatorname{asin}(r)$ directly. Instead, keep the terms r and $rg\mathcal{P}(g)/\mathcal{Q}(g)$ separate as the high and low parts of what is effectively a value that is more accurate than working precision by about four bits.
- Because $\operatorname{acos}(0) = \frac{1}{2}\pi$ cannot be represented exactly, set the *inexact* flag by returning `PI_HALF + TINY`. That change also accommodates rounding modes of other than the IEEE 754 default.
- For the intervals where we compute $\operatorname{asin}(\sqrt{\frac{1}{2}(1 + |x|)})$, borrow a technique from the Sun Microsystems fdlibm library to recover the rounding error made in setting $r = \sqrt{g} \approx \operatorname{fl}(\sqrt{g})$. Represent the exact r as a sum of an exact high part and an approximate low part, and then solve for those parts, like this:

$$\begin{aligned}
r &= r_{\mathrm{hi}} + r_{\mathrm{lo}}, \\
r_{\mathrm{hi}} &= \text{high } \lfloor \tfrac{1}{2}t \rfloor \text{ digits of } r, \\
r_{\mathrm{lo}} &= r - r_{\mathrm{hi}} \\
&= (r - r_{\mathrm{hi}}) \frac{r + r_{\mathrm{hi}}}{r + r_{\mathrm{hi}}} \\
&= (r^2 - r_{\mathrm{hi}}^2)/(r + r_{\mathrm{hi}}) \\
&= (g - r_{\mathrm{hi}}^2)/(r + r_{\mathrm{hi}}).
\end{aligned}$$

  The computation of r_{hi} is accomplished by a simple technique described in **Section 13.11** on page 359 that requires addition and subtraction of a known constant, converting low-order digits of r to zero.

  Because r_{hi}^2 can be represented exactly in t-digit arithmetic, and is close to g, the product and difference in the numerator of the last expression for r_{lo} are exact, and the sum $r_{\mathrm{hi}} + r_{\mathrm{lo}}$ represents $\sqrt{g}$ more accurately than $\operatorname{fl}(\sqrt{g})$.
- Split the constants π and $\frac{1}{4}\pi$ into high and low parts where the high part has one less decimal digit, or three fewer bits, than the native precision. The reduced precision in the high part allows it to be added to a term of comparable size without causing an additional rounding error.
- When a fast higher-precision type is available, use it to construct the results from the split constants and the expansion of $\operatorname{asin}(r)$, summing the smallest terms first.

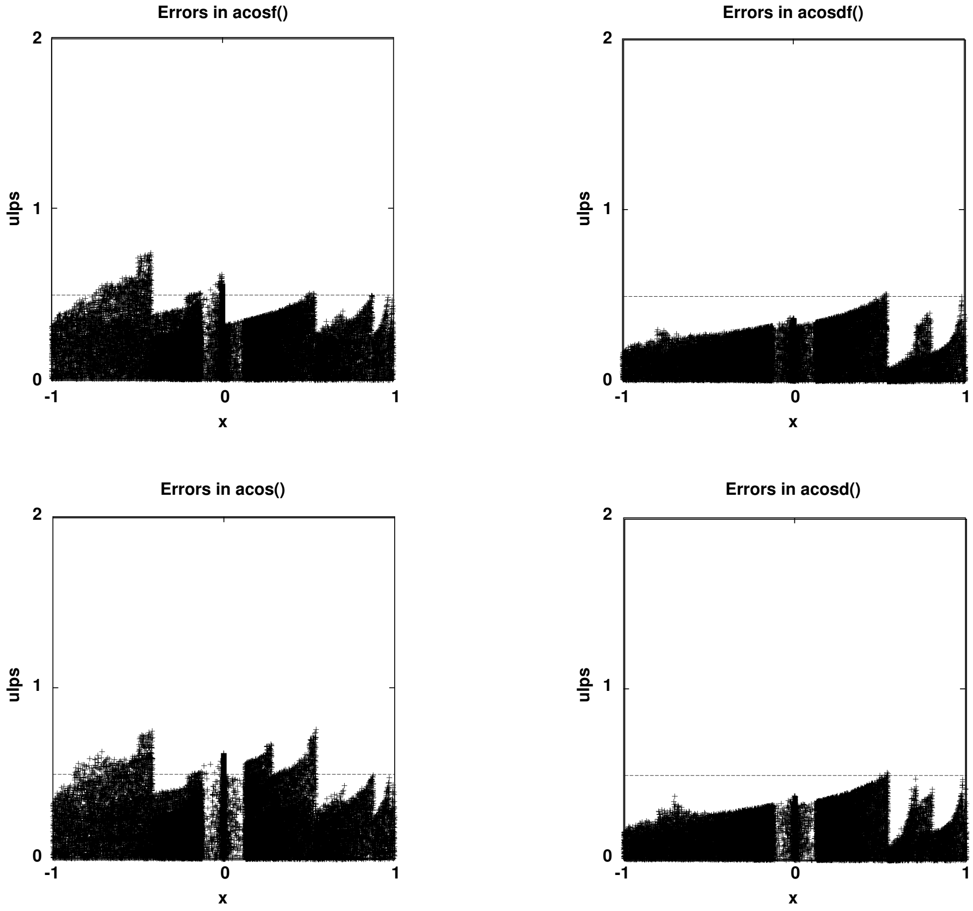


Figure 11.8: Errors in `ACOS()` functions.

- Otherwise, use the simplest of the Kahan–Møller summation formulas that we introduce at the start of Chapter 13 on page 353 to recover the error, and carry it over to the next term, where it is incorporated into the larger of the next two terms to be summed. The code is simplified by using the macro

  ```
  #define SUM(x,y) (_x = x,             \
                    _y = y,             \
                    sum = _x + _y,      \
                    err = _x - sum,     \
                    err += _y,          \
                    sum)
  ```

  that is defined in the `asin.h` header file. The `SUM()` macro uses the C comma expression to control evaluation order, avoiding the need for the `volatile` attribute and calls to the `STORE()` macro. The first argument of `SUM()` must be that of larger magnitude.

Implementing those changes adds several dozen lines of code, but pushes the peaks in the error plots sharply down, so they reach only about 0.75 ulps for `acosf()`, with the function results over most of the argument range $[-1, +1]$ expected to be correctly rounded, as illustrated in Figure 11.8 and Figure 11.9 on the next page. The results for decimal arithmetic are superior because rounding in addition and subtraction is less frequent in a larger base.

The code for our final version of `ACOSSIN()` looks like this:

```
fp_t
```

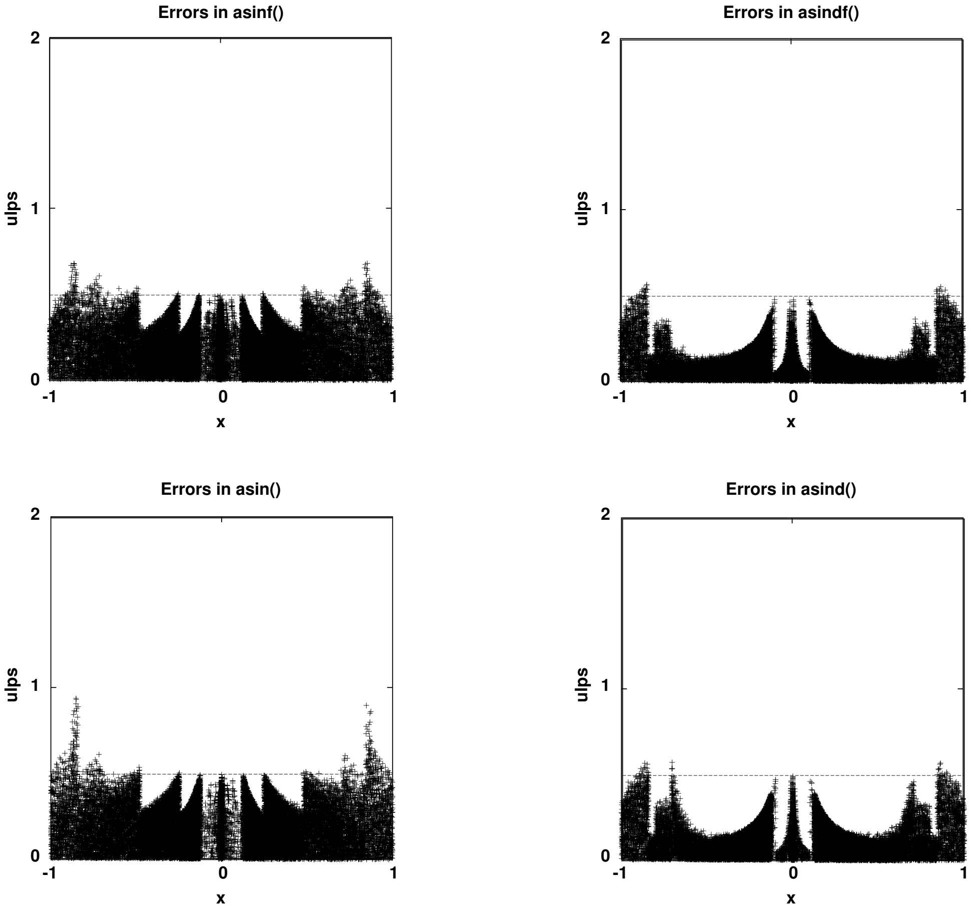


Figure 11.9: Errors in ASIN() functions.

```
ACOSSIN(fp_t x, int flag)
{   /* common code: flag = 0 for asin(), flag = 1 for acos() */
    static fp_t EPS;
    static int do_init = 1;
    static volatile fp_t TINY;
    volatile fp_t result;

    if (do_init)
    {                                    /* once-only initialization */
        EPS = B_TO_MINUS_HALF_T;
        TINY = FP_T_MIN;
        STORE(&TINY);
        do_init = 0;
    }

    result = FP_T_MAX;                   /* inhibit use-before-set warning */

    if ((flag != 0) && (flag != 1))      /* sanity check */
        result = SET_EDOM(QNAN(""));
    else if (ISNAN(x))
        result = SET_EDOM(x);
    else if (ISINF(x))
        result = SET_EDOM(QNAN(""));
    else if ((x < -ONE) || (ONE < x))
        result = SET_EDOM(QNAN(""));
    else if (QABS(x) == ONE)
    {
        if (flag == 0)
            result = COPYSIGN(PI_HALF, x);
        else
            result = (x == ONE) ? ZERO : PI;
    }
    else if (x == ZERO)                 /* asin(0) or acos(0) */
        result = (flag == 0) ? x : (PI_HALF + TINY);
    else
    {
        fp_t err, f, f_hi, f_lo, g, pg_g, qg, r, s, sum, _x, _y;
        int done, i;

        done = 0;
        f_hi = f_lo = pg_g = qg = s = FP_T_MAX; /* inhibit warnings */
        r = FABS(x);

        if (r > SQRT_HALF)
        {
            i = 1 - flag;

#if defined(HAVE_GUARD_DIGIT)
            g = (ONE - r) * HALF;
#else
            g = HALF - r;
            g += HALF;
            g /= TWO;
#endif

            s = SQRT(g);
            r = -s;
            r += r;                     /* r = -2 * SQRT(g) */
        }
        else                            /* r <= SQRT_HALF */
        {
            i = flag;

            if (r < EPS)                /* use 2-term Taylor series */
            {
                g = ZERO;               /* avoid compiler warnings */
                f_lo = r * r * r / SIX;
                f_hi = r;
                done = 1;
            }
            else
                g = r * r;
        }

        if (!done)
        {
            pg_g = POLY_P(p, g) * g;
            qg   = POLY_Q(q, g);
            /* f = f_hi + f_lo (kept separate), where
               f = asin(|x|) (cases 0 and 3), or
               f = -2*asin(sqrt((1-|x|)/2)) (cases 1 and 2) */

            f_lo = r * pg_g / qg;
            f_hi = r;
        }

        switch (2*flag + i)
        {
        default:                        /* asin(x) = +/-asin(|x|) */
        case 0:                         /* x in [-sqrt(1/2),+sqrt(1/2)] */
            f = f_hi + f_lo;
            result = (x < ZERO) ? -f : f;
            break;

        case 1:                         /* x in [-1,-sqrt(1/2)) or (sqrt(1/2),1] */

#if defined(HAVE_FAST_HP_T)
            {
                volatile hp_t t, u;

                t = (hp_t)PI_QUARTER_HI + (hp_t)f_hi;
                STORE((fp_t *)&t);
                t += (hp_t)PI_QUARTER_HI;
                u = (hp_t)PI_QUARTER_LO + (hp_t)PI_QUARTER_LO;
                STORE((fp_t *)&u);
                u += (hp_t)r * (hp_t)pg_g / (hp_t)qg;
                result = (fp_t)(t + u);
            }
#else
            result = SUM(f_lo, PI_QUARTER_LO + PI_QUARTER_LO);
            STORE(&result);
            result = SUM(f_hi + err, result);
            STORE(&result);
            result = SUM(PI_QUARTER_HI + err, result);
            STORE(&result);
            result = SUM(PI_QUARTER_HI + err, result);
            STORE(&result);
#endif /* defined(HAVE_FAST_HP_T) */

            if (x < ZERO)
                result = -result;
            break;

        case 2:                         /* x in [-1,-sqrt(1/2)) or (sqrt(1/2),1] */
            if (x < ZERO)               /* acos(x) = PI - 2*asin(sqrt((1-|x|)/2)) */
            {

#if defined(HAVE_FAST_HP_T)
                hp_t t, u;

                t = (hp_t)PI_HI + (hp_t)f_hi;
                u = (hp_t)PI_LO + (hp_t)r * (hp_t)pg_g / (hp_t)qg;
                result = (fp_t)(t + u);
                STORE(&result);
```

```
#else
                result = SUM(f_lo, PI_LO);
                STORE(&result);
                result = SUM(f_hi + err, result);
                STORE(&result);
                result = SUM(PI_HI + err, result);
                STORE(&result);
#endif /* defined(HAVE_FAST_HP_T) */

            }
            else                        /* acos(x) = 2*asin(sqrt((1-|x|)/2)) */
            {                           /* result = -f (but accurately!) */
                fp_t r_lo, t;
                volatile fp_t r_hi;

                r = s;                  /* sqrt(g) */
                r_hi = r + B_TO_CEIL_HALF_T;
                STORE(&r_hi);
                r_hi -= B_TO_CEIL_HALF_T;
                r_lo = (g - r_hi * r_hi) / (r + r_hi);
                t = pg_g / qg;
                result = SUM(r_lo, r_lo * t);
                STORE(&result);
                result = SUM(r_hi * t + err, result);
                STORE(&result);
                result = SUM(r_hi + err, result);
                STORE(&result);
                result += result;
                STORE(&result);
            }
            break;

        case 3:                         /* x in [-sqrt(1/2),+sqrt(1/2)] */
            if (x < ZERO)               /* acos(x) = PI/2 - asin(x) */
            {                           /*         = PI/2 + asin(|x|) */
                result = PI_QUARTER_LO + PI_QUARTER_LO + f_lo;
                STORE(&result);
                result += PI_QUARTER_HI;
                STORE(&result);
                result += f_hi;
                STORE(&result);
                result += PI_QUARTER_HI;
                STORE(&result);
            }
            else                        /* acos(x) = PI/2 - asin(x) */
            {

#if defined(HAVE_FAST_HP_T)
                hp_t t, u;

                u = (hp_t)PI_QUARTER_LO - (hp_t)f_lo;
                u += (hp_t)PI_QUARTER_LO;
                t = (hp_t)PI_QUARTER_HI - (hp_t)f_hi;
                t += (hp_t)PI_QUARTER_HI;
                result = (fp_t)(t + u);
                STORE(&result);
#else
```

```
                    result = SUM(-f_lo, PI_QUARTER_LO);
                    STORE(&result);
                    result = SUM(PI_QUARTER_LO + err, result);
                    STORE(&result);
                    result = SUM(PI_QUARTER_HI + err, result);
                    STORE(&result);
                    result = SUM(-f_hi + err, result);
                    STORE(&result);
                    result = SUM(PI_QUARTER_HI + err, result);
                    STORE(&result);
#endif /* defined(HAVE_FAST_HP_T) */

                }
                break;
            }
        }

        return (result);
}
```

The control flow in ACOSSIN() is complex, with eight subcases treated in the switch statement. A few early assignments are done solely to eliminate superfluous warnings from compilers that attempt to detect errors of use before definition, following our policy of providing warning-free compilation of the entire mathcw library.

The code contains two instances of sums of the low parts of $\frac{1}{4}\pi$. They are written that way, rather than replaced by the corresponding low part of $\frac{1}{2}\pi$, because that would then place a requirement on the split of $\frac{1}{2}\pi$ that we prefer to avoid. There are multiple versions of those splits, one for each supported host precision, and particularly for a decimal base, the separate splits of $\frac{1}{2}\pi$ and $\frac{1}{4}\pi$ may not differ by an exact factor of two. Optimizing compilers can evaluate the sums at compile time, because the values added are constants.

In the second case of the switch statement, we reevaluate $rg\mathcal{P}(g)/\mathcal{Q}(g)$, computing the factors in higher precision to gain a bit more accuracy. Although that duplicates earlier code that uses working precision, it is done only for the argument region $[-1, -\sqrt{\frac{1}{2}}]$, and is minor compared to the rest of the code.

11.10 Inverse tangent

The Taylor series for the inverse tangent function is:

$$\operatorname{atan}(x) = x - (1/3)x^3 + (1/5)x^5 - (1/7)x^7 + (1/9)x^9 - \cdots.$$

Section 23.9 on page 801 introduces some additional series expansions for that function, but we do not require them here.

The argument range of $\operatorname{atan}(x)$ is $(-\infty, +\infty)$, and the function range is $[-\frac{1}{2}\pi, +\frac{1}{2}\pi]$. The function is graphed in Figure 11.10 on the next page.

The symmetry relation

$$\operatorname{atan}(-x) = -\operatorname{atan}(x)$$

allows us to compute with only positive arguments.

The infinite argument range of the inverse tangent requires further argument reduction, through these relations:

$$\begin{aligned}
r &= \big((\sqrt{3})x - 1\big)/(x + \sqrt{3}), && \textit{use if } \beta \neq 16,\\
&= \big((((\sqrt{3} - 1)x - \tfrac{1}{2}) - \tfrac{1}{2}) + x\big)/(x + \sqrt{3}), && \textit{use if } \beta = 16,\\
\operatorname{atan}(x) &= \tfrac{1}{2}\pi - \operatorname{atan}(1/x), && \textit{use if } |x| > 1,\\
&= \tfrac{1}{6}\pi + \operatorname{atan}(r), && \textit{use if } |x| > 2 - \sqrt{3}.
\end{aligned}$$

For arguments of large magnitude, $\operatorname{atan}(1/x)$ is a small correction to $\frac{1}{2}\pi$, so we expect that the correctly rounded inverse tangent of large $|x|$ should be easy to compute.

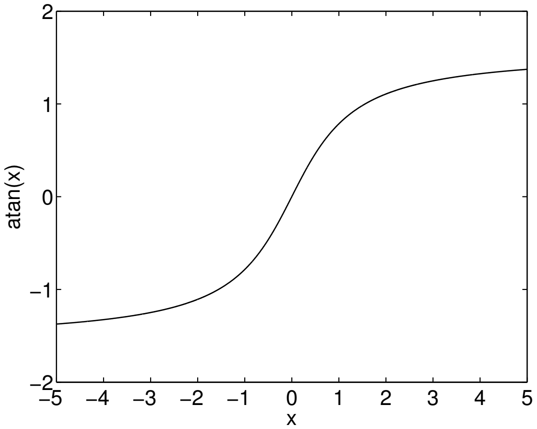


Figure 11.10: The inverse tangent function. It is defined for arguments over the entire real axis, with limits $\mp\frac{1}{2}\pi$ as x approaches $\mp\infty$.

For a zero argument, we set $\operatorname{atan}(x) = x$ to preserve the sign of zero. For tiny arguments, Cody and Waite use only the first term of the Taylor series, but we use two terms, $\operatorname{atan}(x) \approx x - x^3/3$, for correct rounding and setting of the *inexact* flag. Otherwise, they use a rational polynomial approximation

$$g = x^2, \qquad \operatorname{atan}(|x|) \approx |x| + |x| g\mathcal{P}(g)/\mathcal{Q}(g),$$

valid for $|x|$ in the interval $[0, 2-\sqrt{3}]$, or about $[0, 0.268]$. In that interval, the range of the factor $g\mathcal{P}(g)/\mathcal{Q}(g)$ is $[0, -0.023]$, and its magnitude is never bigger than 9.5% of $|x|$, so the inverse tangent is computed from the sum of an exact value and a small correction.

The computation of r suffers loss of leading digits as $|x|$ nears $\sqrt{1/3}$, causing r and $\operatorname{atan}(r)$ to approach zero. Cody and Waite argue that the accuracy loss is compensated by the addition of $\frac{1}{6}\pi$, so the loss can be ignored. However, error plots for a direct implementation of their algorithm show that $\operatorname{atan}(x)$ is correctly rounded for almost all arguments, *except* for the intervals $[-\frac{1}{2}, -\frac{1}{4}]$ and $[+\frac{1}{4}, +\frac{1}{2}]$, where some errors rise to about two ulps. Tracing the computational steps with an argument that produces one of the largest errors reveals the reason for the peaks in the error plots: a one-ulp error in the polynomial evaluation sometimes coincides with a one-ulp error in the subtraction from $\frac{1}{6}\pi$, doubling the error in the inverse tangent. Computing r in the next higher precision helps, but still leaves peak errors of about 1.5 ulps in the two problem regions.

A solution to that problem is to extend the Cody/Waite algorithm with an additional argument-reduction formula that avoids both the irrational constant and a subtraction from that constant. Use the angle-sum formula for the tangent (see **Section 11.2** on page 302) to find the inverse tangent as the sum $y = c + d$, like this:

$$\begin{aligned}
y &= \operatorname{atan}(x), && \textit{the function value that we seek,}\\
x &= \tan(y),\\
&= \tan(c+d), && \textit{c a chosen constant,}\\
&= \big(\tan(c) + \tan(d)\big)/\big(1 - \tan(c)\tan(d)\big),\\
r &= \tan(d),\\
&= \big(x - \tan(c)\big)/\big(x\tan(c) + 1\big), && \textit{by solving x for } \tan(d),\\
d &= \operatorname{atan}(r).
\end{aligned}$$

Thus, for a fixed value of c, $\tan(c)$ is a constant, and r can be computed with just four arithmetic operations. We then use the polynomial approximation for the inverse tangent to find d, and then add that value to c to recover $\operatorname{atan}(x)$.

We want c to be exactly representable, and we need to ensure that the reduced argument, r, lies in the range where the polynomial approximation is valid. A suitable choice is

$$c = 1/4, \qquad \tan(c) = 0.255\,341\,921\,221\,036\,266\,504\,482\,236\,490\,473\,678\,\ldots\,.$$

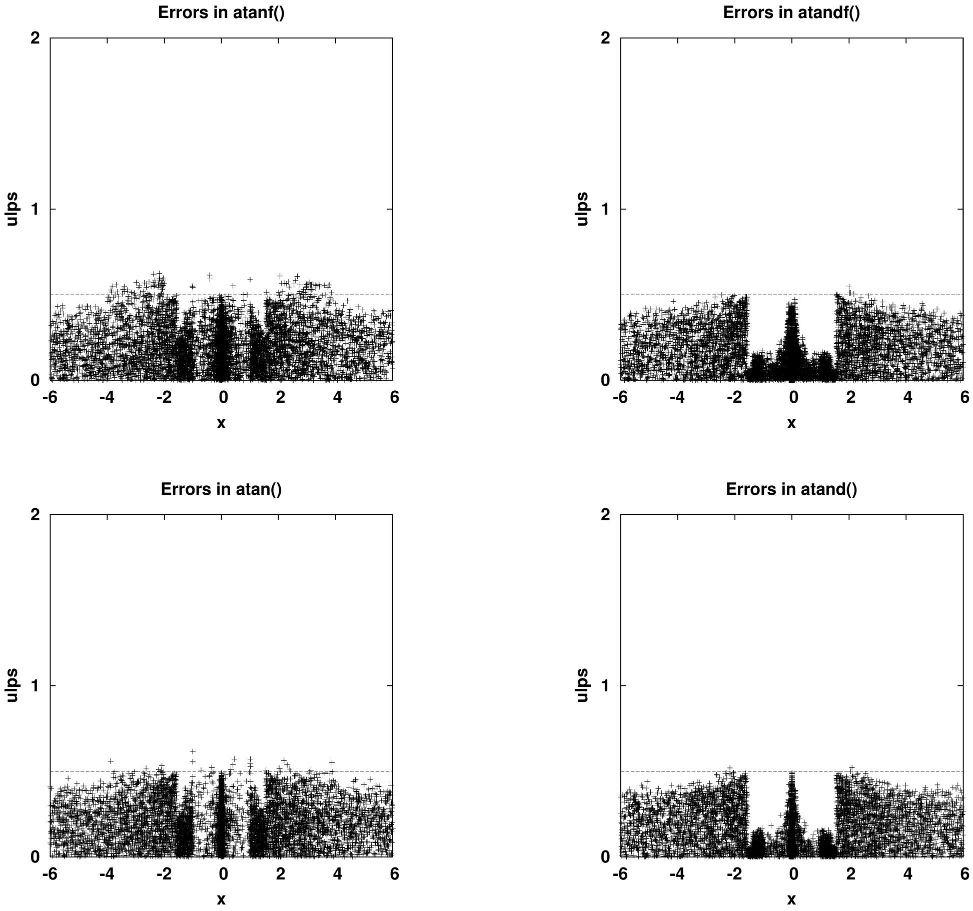


Figure 11.11: Errors in `ATAN()` functions. Outside the interval shown in the plots, the errors remain well below $\frac{1}{2}$ ulp, indicating correct rounding. For `atanf(x)`, fewer than 0.6% of the results differ from the rounded exact result for x in $[16, 1000]$. About 7% differ for x in $[1, 16]$, and about 11% differ for x in $[\frac{1}{4}, 1]$.

For x in $[2 - \sqrt{3}, \frac{1}{2}] \approx [0.268, \frac{1}{2}]$, we have r inside $[0.011, 0.217]$, which is contained in the polynomial-approximation interval. For that range of r, the value $d = \operatorname{atan}(r)$ lies within $[0.011, 0.214]$. In binary arithmetic, for d in the interval $[\frac{1}{8}, \frac{1}{4}]$, the sum $y = \frac{1}{4} + d$ is computed exactly, and otherwise, it should be correctly rounded.

When we compute r, the numerator $x - \tan(c)$ suffers loss of leading bits near the left end of the interval of x, but we can recover those bits by computing it as $(x - \frac{1}{4}) - (\tan(c) - \frac{1}{4})$, where $x - \frac{1}{4}$ is exact, and $\tan(c) - \frac{1}{4}$ is split into exact high and approximate low parts.

We gain additional accuracy by representing the constants that are fractions of π as a sum of high and low parts, where the high part is carefully chosen to use as many digits as can be represented exactly for the given precision. The low part of each split constant is added before the high part.

Error plots show a distinct improvement from computing the reduced argument r in the next higher precision, so we include that refinement. **Figure 11.11** shows the measured errors in our final implementation, whose code looks like this:

```
fp_t
ATAN(fp_t x)
{
    volatile fp_t result;

    if (ISNAN(x))
```

```
            result = SET_EDOM(x);
        else if (x == ZERO)
            result = x;                     /* preserve sign of zero argument */
        else
        {
            fp_t r, xabs;
            int n;
            static const fp_t EPS = MCW_B_TO_MINUS_HALF_T;

            n = 0;
            xabs = FABS(x);
            r = xabs;

            if (r > ONE)
            {
                r = ONE / r;
                n = 2;
            }

            if (r > TWO_MINUS_SQRT_THREE)
            {
                hp_t rr;
                volatile hp_t tt;

                if (r >= HALF)
                {
                    rr = (n == 2) ? (HP(1.0) / (hp_t)xabs) : (hp_t)r;

#if defined(HAVE_WOBBLING_PRECISION)
                    tt = rr * HP_SQRT_THREE_MINUS_ONE - (hp_t)HALF;
                    STORE(&tt);
                    tt -= (hp_t)HALF;
                    STORE(&tt);
                    tt += rr;
#else
                    tt = rr * HP_SQRT_THREE - (hp_t)ONE;
#endif

                    r = (fp_t)(tt / (rr + HP_SQRT_THREE));
                    n++;
                }
                else
                {
                    rr = (n == 2) ? (HP(1.0) / (hp_t)xabs) : (hp_t)r;
                    tt = rr - HP(0.25);
                    STORE(&tt);
                    tt -= (hp_t)TAN_QUARTER_MINUS_QUARTER_LO;
                    STORE(&tt);
                    tt -= (hp_t)TAN_QUARTER_MINUS_QUARTER_HI;
                    r = (fp_t)(tt / (HP(1.0) + rr * HP_TAN_QUARTER));
                    n += 4;
                }
            }

            if (QABS(r) < EPS)              /* use two-term Taylor series */
            {
                result = -r * r * r / THREE;
                result += r;
            }
            else
            {
                fp_t g, pg_g, qg;

                g = r * r;
                pg_g = POLY_P(p, g) * g;
                qg   = POLY_Q(q, g);
                result = FMA(r, pg_g / qg, r);
            }

            switch (n)
            {
            default:
            case 0:                         /* |x| in [0, 2 - sqrt(3)] */
                break;                      /* atan(|x|) = atan(r) */

            case 1:                         /* |x| in [1/2, 1] */
                result += PI_SIXTH_LO;
                result += PI_SIXTH_HI;
                break;                      /* atan(|x|) = PI/6 + atan(r) */

            case 2:                         /* 1/|x| in [0, 2 - sqrt(3)] */
                result = -result;
                result += PI_HALF_LO;
                result += PI_HALF_HI;
                break;                      /* atan(|x|) = PI/2 - atan(1/|x|) */

            case 3:                         /* 1/|x| in [1/2, 1] */
                result = -result;
                result += PI_THIRD_LO;
                result += PI_THIRD_HI;
                break;                      /* atan(|x|) = PI/2 - (PI/6 + atan(1/|x|)) */

            case 4:                         /* (|x| in (2 - sqrt(3), 1/2)) OR
                                               (1/|x| in (2 - sqrt(3), 1/2)) */
                result += FOURTH;
                break;                      /* atan(|x|) = 1/4 + atan(r) */

            case 6:                         /* 1/|x| in (2 - sqrt(3), 1/2) */
                result += FOURTH;
                result = -result;
                result += PI_HALF_LO;
                result += PI_HALF_HI;
                break;                      /* atan(|x|) = PI/2 - (1/4 + atan(r)) */
            }

            if (x < ZERO)
                result = -result;
        }

        return (result);
    }
```

11.11 Inverse tangent, take two

The `atan()` function family is accompanied in several programming languages by a two-argument form, `atan2(y,x)`, that preserves important sign information, and avoids the need for an explicit Infinity for its argument. The arguments of `atan2(y,x)` can be considered the opposite and adjacent sides of a right triangle defining the positive (counterclockwise) angle of rotation about the origin.

The C99 Standard defines that function briefly as follows:

> ***7.12.4.4 The atan2 functions***
> ***Synopsis***
>
> ```
> #include <math.h>
> double atan2       (double y, double x);
> float atan2f       (float y, float x);
> long double atan2l (long double y, long double x);
> ```
>
> ***Description***
> *The `atan2()` functions compute the value of the arc tangent of `y/x`, using the signs of both arguments to determine the quadrant of the return value. A domain error may occur if both arguments are zero.*
> ***Returns***
> *The `atan2()` functions return* $\arctan y/x$ *in the interval* $[-\pi, +\pi]$ *radians.*

The Standard later places requirements on the function's behavior in about two dozen special cases:

> ***F.9.1.4 The atan2 functions***
>
> - $\mathrm{atan2}(\pm 0, -0)$ *returns* $\pm\pi$. *[Footnote:* $\mathrm{atan2}(0,0)$ *does not raise the* invalid *floating-point exception, nor does* $\mathrm{atan2}(y, 0)$ *raise the* divbyzero *floating-point exception.]*
> - $\mathrm{atan2}(\pm 0, +0)$ *returns* ± 0.
> - $\mathrm{atan2}(\pm 0, x)$ *returns* $\pm\pi$ *for* $x < 0$.
> - $\mathrm{atan2}(\pm 0, x)$ *returns* ± 0 *for* $x > 0$.
> - $\mathrm{atan2}(y, \pm 0)$ *returns* $-\frac{1}{2}\pi$ *for* $y < 0$.
> - $\mathrm{atan2}(y, \pm 0)$ *returns* $\frac{1}{2}\pi$ *for* $y > 0$.
> - $\mathrm{atan2}(\pm y, -\infty)$ *returns* $\pm\pi$ *for finite* $y > 0$.
> - $\mathrm{atan2}(\pm y, +\infty)$ *returns* ± 0 *for finite* $y > 0$.
> - $\mathrm{atan2}(\pm\infty, x)$ *returns* $\pm\frac{1}{2}\pi$ *for finite* x.
> - $\mathrm{atan2}(\pm\infty, -\infty)$ *returns* $\pm\frac{3}{4}\pi$.
> - $\mathrm{atan2}(\pm\infty, +\infty)$ *returns* $\pm\frac{1}{4}\pi$.

We discuss that function, and portability issues from its incomplete definition in various programming languages and floating-point hardware designs, in **Section 4.7.3** on page 70. The most troublesome case is that where both arguments are zero, even though that situation is addressed in the C89 and C99 Standards.

Cody and Waite give an algorithm for $\mathrm{atan2}(y, x)$, although they do not address the issue of signed zeros or subnormal arguments, because their book preceded the IEEE 754 design. Our code for that function follows the C99 specification, and looks like this:

```
fp_t
ATAN2(fp_t y, fp_t x)
{
    fp_t result;

    if (ISNAN(y))
        result = SET_EDOM(y);
    else if (ISNAN(x))
        result = SET_EDOM(x);
    else if (y == ZERO)
    {
```

```
        if (x == ZERO)
            result = (COPYSIGN(ONE,x) >= ZERO) ? y : COPYSIGN(PI, y);
        else if (x < ZERO)
            result = COPYSIGN(PI, y);
        else
            result = COPYSIGN(ZERO, y);
    }
    else if (x == ZERO)
        result = (y < ZERO) ? -PI_HALF : PI_HALF;
    else if (ISINF(x))
    {
        if (ISINF(y))
        {
            if (x < ZERO)
                result = COPYSIGN(THREE_PI_QUARTER, y);
            else
                result = COPYSIGN(PI_QUARTER, y);
        }
        else
            result = (x < ZERO) ? COPYSIGN(PI, y) : COPYSIGN(ZERO, y);
    }
    else if (ISINF(y))                  /* x is finite and nonzero */
        result = (y < ZERO) ? -PI_HALF : PI_HALF; /* y / x overflows */
    else                                /* x and y are finite and nonzero */
    {
        fp_t xabs, yabs;

        xabs = FABS(x);
        yabs = FABS(y);

        if ((yabs > xabs) && (xabs < ONE) && (yabs > (xabs*FP_T_MAX)))
            result = (y < ZERO) ? -PI_HALF : PI_HALF; /* y / x ofl */

#if defined(HAVE_IEEE_754)
        else if ( (yabs < xabs) && (yabs < (xabs * FP_T_MIN)) )
#else
        else if ( (yabs < xabs) && (xabs > ONE) && (yabs < (xabs * FP_T_MIN)) )
#endif

        {                               /* y / x underflows to subnormal or zero */

#if defined(HAVE_IEEE_754)
            result = y / x;             /* set underflow and inexact */
#else
            result = ZERO;
#endif

            if (x < ZERO)
                result = PI - result;   /* set inexact */

            if (y < ZERO)
                result = -result;
        }
        else                            /* normal case: atan() completes the job */
            result = ATAN(y / x);       /* y / x is finite and nonzero */
    }
```

Table 11.3: Hardware instructions for trigonometric functions.

Function	IA-32	68000
inverse cosine	—	facos
inverse sine	—	fasin
inverse tangent	fpatan	fatan
cosine	fcos	fcos
sine	fsin	fsin
cosine & sine	fsincos	—
tangent	fptan	ftan

```
    return (result);
}
```

The constants involving fractions of π are all stored values to ensure that they are correct to the last digit. The first half of the code handles the special cases required by C99. The second half, the final outer `else` block, treats the cases where y/x would overflow or underflow. The tests are complicated because they check for overflow and underflow without normally causing those conditions.

The value `FP_T_MIN` is the smallest *normal* number, and no standard names are available in C for the smallest and largest subnormals. If y is subnormal, y/x can be subnormal for $|x|$ values just below one. In that case, the expression `xabs * FP_T_MIN` underflows to a subnormal or zero, likely setting the *underflow* and *inexact* exception flags prematurely. However, the next statement executed in each block then computes y/x, which sets the same flags.

For other arithmetic systems, ensuring that $|x| > 1$ before evaluating the product `xabs * FP_T_MIN` prevents premature underflow, and if y/x would underflow, we assign a zero value to `result`, avoiding the underflow.

The `atan2()` function has more than two dozen nested logical tests, making its flow control among the most complex in the entire C library. Extensive validation is therefore essential, and the test files `chkatan2.c` and `chkatan2d.c` perform more than 40 checks on each member of the function family, supplementing the accuracy tests made by the ELEFUNT test suite.

11.12 Trigonometric functions in hardware

Given the considerable complexity of computing trigonometric functions, it is perhaps surprising that two architecture families, the Intel IA-32 and the Motorola 68000, provide them in hardware. The instructions listed in Table 11.3 include argument reduction, but the reduced argument is correctly rounded only in a limited range. Motorola documentation is vague about that range, but Intel manuals note that the instruction fails unless the magnitude of the argument operand is smaller than 2^{63}. For the inverse tangent, the Intel instruction corresponds to the two-argument form, $\mathrm{atan2}(y, x)$, but the Motorola instruction computes only $\mathrm{atan}(x)$.

The `mathcw` library code uses the trigonometric hardware instructions under these conditions:

- the host architecture provides the instructions;
- the compiler supports the `gcc` syntax for inline assembly code;
- special arguments (Infinity, NaN, and signed zero) are handled separately;
- the many special cases of the two-argument inverse tangent function, $\mathrm{atan2}(y, x)$, have been dealt with according to the C language standards;
- accurate argument reduction has already been completed;
- the preprocessor symbol `USE_ASM` is defined.

The hardware instructions provide only a partial solution to the trigonometric function computation, because they replace only the code block that computes a polynomial approximation.

As an example of how a trigonometric hardware instruction can be used, after the reduction $r_{\text{hi}} + r_{\text{lo}} = x - n(\frac{1}{2}\pi)$, the cosine and sine can be computed together on IA-32 and compatible systems like this:

```
long double cos_r_hi, rr, sin_r_hi;

rr = (long double)r_hi;

__asm__ __volatile__("fsincos" : "=t" (cos_r_hi), "=u" (sin_r_hi) : "0" (rr));

the_cos = (fp_t)(cos_r_hi - sin_r_hi * (long double)r_lo);
the_sin = (fp_t)(sin_r_hi + cos_r_hi * (long double)r_lo);
```

The syntax of the inline assembly-code directive is discussed in **Section 13.26** on page 388, so we do not describe it further here. The high part of the reduced argument is first converted exactly to the 80-bit format used by the floating-point hardware and stored in the variable `rr`. In the absence of precision control, the `fsincos` instruction produces results in the 80-bit format for $\cos(r_{\text{hi}})$ and $\sin(r_{\text{hi}})$. Those values are used in the corrections for the argument error, and the two expressions are then converted to working precision. The computed cosine and sine values are, with high probability, correctly rounded in the 32-, 64-, and 80-bit formats, although in the shorter formats, there is a double rounding that may sometime cause an error of $\frac{1}{2}$ ulp in the default *round-to-nearest* mode.

It is likely that implementations of the trigonometric instructions on differing CPU designs or architecture generations vary in accuracy. Extensive testing of the code with logarithmically distributed random arguments on Intel Pentium and Xeon, and AMD Opteron, CPUs finds no examples of incorrect rounding in up to 10^7 tests. However, there are so many different chip vendors and models of CPUs in the IA-32 family that further testing is advisable.

Speed tests of the hardware instructions against our software code show considerable variation across platforms, and on some systems, the hardware approach is found to be slower. The main benefit of trigonometric hardware instructions seems to be enhanced function accuracy.

11.13 Testing trigonometric functions

The data for the error plots shown in this chapter, and elsewhere in this book, are produced with test programs that invoke two versions of the function, one at working precision, and the other at the next higher precision. Test arguments are randomly selected from a logarithmic distribution in each of several regions where the function is defined, with special attention to trouble spots, and regions where different algorithms are used. The errors in ulps with respect to the higher-precision values are recorded in an output file for graphing.

By controlling the order of source files and load libraries, the two functions can be chosen from different, and independent, packages. That is a useful test, because our use in the mathcw library of a single algorithm file for all floating-point types could mask a systematic error in the computational recipes.

The ELEFUNT test suite checks particular known values of the functions, as well as arguments that are out-of-range, or that might be expected to cause underflows and overflows. It also uses logarithmically distributed random arguments to verify how well certain carefully chosen mathematical identities are satisfied. The selected identities are unlikely to be used in the computational algorithm, and usually require the arguments to be purified to avoid introducing additional rounding errors into the tests.

For example, for the sine and cosine, ELEFUNT tests the identities

$$\begin{aligned}
\sin(x) &\equiv 3\sin(\tfrac{1}{3}x) - 4(\sin(\tfrac{1}{3}x))^3, \\
\cos(x) &\equiv 4(\cos(\tfrac{1}{3}x))^3 - 3\cos(\tfrac{1}{3}x), \\
\sin(x) &\equiv -\sin(-x), \\
\cos(x) &\equiv \cos(-x), \\
\sin(x) &\equiv x, && \textit{for sufficiently small } x.
\end{aligned}$$

For the tests with arguments of $\frac{1}{3}x$, the random values are adjusted so that $\frac{1}{3}x$ and $3(\frac{1}{3}x)$ are exactly representable, with code like this:

```
x = randab(a,b);                        /* log-distributed random value on [a,b] */
y = x / FP(3.0);                        /* form y = (x + x/3) - x, carefully */
```

```
y += x;
y -= x;
x = FP(3.0) * y;                        /* now x is exactly 3*y */
```

For the tangent and cotangent, these identities are tested:

$$\begin{aligned}\tan(x) &\equiv \frac{2\tan(\frac{1}{2}x)}{1-(\tan(\frac{1}{2}x))^2},\\ \operatorname{cotan}(x) &\equiv \frac{(\cot(\frac{1}{2}x))^2-1}{2\cot(\frac{1}{2}x)}.\end{aligned}$$

The inverse sine, cosine, and tangent are tested against their Taylor series expansions. The ELEFUNT suite also checks these identities for the inverse tangent:

$$\begin{aligned}\operatorname{atan}(x) &\equiv \operatorname{atan}(1/16)+\operatorname{atan}((x-1/16)/(1+x/16)),\\ 2\operatorname{atan}(x) &\equiv \operatorname{atan}(2x/(1-x^2)).\end{aligned}$$

11.14 Retrospective on trigonometric functions

There are three main difficulties in programming the trigonometric functions: their variety, the complexity of their argument reduction, and achieving high accuracy over the complete argument range.

In this chapter, we covered the most important functions in that family that are provided by the C library. The mathcw library offers several convenient extensions, some of which are also provided by a few vendor libraries, including the asindeg(), asinp(), asinpi(), sindeg(), sinp(), and sinpi() families, and their companions for the cosine, cotangent, and tangent, plus sincos().

Chapter 9 covers argument reduction, showing that there is a fast way for small arguments, and a slow way for large ones, as provided by the function families REDUCE(), EREDUCE(), and ERIDUCE(). To hide that problem, we introduced in this chapter the private function families RP() and RPH() to handle reductions with respect to π and $\frac{1}{2}\pi$.

Cody and Waite generally do not require more than working precision in their algorithms, although they do mention that the accuracy of their recipe for trigonometric-argument reduction can be improved by using higher precision. We found that some limited use of higher precision in part of the code is essential to push the peaks in the error plots below one ulp, especially for functions with large mathematical error magnification, such as the tangent function. At the highest available precision, which is just double on some systems, accuracy unavoidably decreases. Libraries that solve that problem generally do so by using software extended precision, with considerable code complexity.

The two-argument form of the inverse tangent, atan2(), requires care in handling of the special cases required by the C Standards, and we discuss in Section 4.7.3 on page 70 some of the hardware and language portability issues related to signed zeros. Our implementation of that function passes the tests of our validation programs, but all vendor libraries that have been subjected to those tests fail several of them. Tests of several symbolic algebra systems show that they too disagree in many of the special cases. The lessons are that complexity in language specifications does not lead to reliable code, and that users of the atan2() function family cannot expect consistency across compilers and platforms. There is an even greater problem when translating code between different languages that supply a two-argument tangent, because language specifications of the two-argument inverse tangent, and the misimplementations of it, differ.

12 Hyperbolic functions

OUT, HYPERBOLICAL FIEND! HOW VEXEST THOU THIS MAN!

— SHAKESPEARE'S *Twelfth Night* (1602).

PERFORMANCES ARE LARGELY SATISFACTORY, EVEN IF THE LEAD ACTRESS TENDS TO GO HYPERBOLIC AT TIMES.

— FILM REVIEW FOR *Khoey Ho Tum Kahan* (2001).

The hyperbolic functions are often treated with the trigonometric functions in mathematical texts and handbooks [AS64, OLBC10, Chapter 4], because there are striking similarities in their series expansions, and the relations between family members, even though their graphs are quite different.

IBM 709 Fortran provided the hyperbolic tangent function in 1958, and that remained the only family member through the first Fortran Standard in 1966. The 1977 Fortran Standard [ANSI78] adds the hyperbolic cosine and sine, but even the 2004 Standard [FTN04a] fails to mention the inverse hyperbolic functions, although a few vendor libraries do provide them.

Fortran versions of the hyperbolic functions and their inverses are available in the PORT library [FHS78b]. The FNLIB library [Ful81b, Ful81a] provides only the inverse hyperbolic functions. Both libraries were developed in the 1970s by researchers at AT&T Bell Laboratories. We discuss their accuracy in the chapter summary on page 352.

Neither Java nor Pascal has any hyperbolic functions in its mathematical classes or libraries, but C# supplies the hyperbolic cosine, sine, and tangent.

Common Lisp requires the hyperbolic functions and their inverses, but the language manual notes that direct programming of mathematical formulas may be inadequate for their computation [Ste90, page 331]. Tests of those functions on two popular implementations of the language suggest that advice was not heeded.

The C89 Standard requires the hyperbolic cosine, sine, and tangent, and the C99 Standard adds their inverse functions. In this chapter, we show how those functions are implemented in the mathcw library.

12.1 Hyperbolic functions

The hyperbolic companions of the standard trigonometric functions are defined by these relations:

$$\begin{aligned}
\cosh(x) &= (\exp(x) + \exp(-x))/2, \\
\sinh(x) &= (\exp(x) - \exp(-x))/2, \\
\tanh(x) &= \sinh(x)/\cosh(x) \\
&= (\exp(x) - \exp(-x))/(\exp(x) + \exp(-x)).
\end{aligned}$$

They are connected by these equations:

$$\begin{aligned}
(\cosh(x))^2 - (\sinh(x))^2 &= 1, \\
\tanh(x) &= \sinh(x)/\cosh(x).
\end{aligned}$$

Those functions exist for all real arguments, and vary smoothly over the ranges $[1, \infty)$ for $\cosh(x)$, $(-\infty, +\infty)$ for $\sinh(x)$, and $[-1, +1]$ for $\tanh(x)$. **Figure 12.1** on the following page shows plots of the functions.

The hyperbolic functions have simple reflection rules:

$$\begin{aligned}
\cosh(-|x|) &= +\cosh(|x|), \\
\sinh(-|x|) &= -\sinh(|x|), \\
\tanh(-|x|) &= -\tanh(|x|).
\end{aligned}$$

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_12

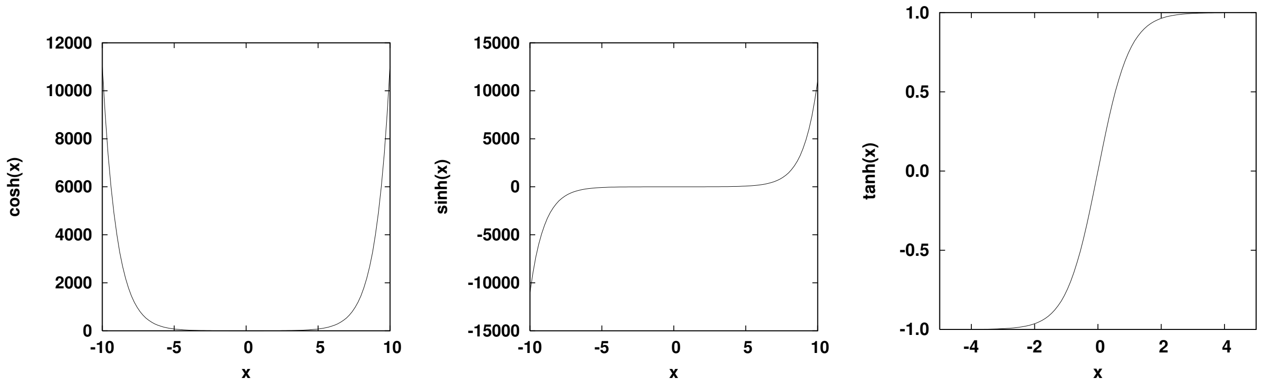


Figure 12.1: The hyperbolic functions.

We can guarantee those symmetries by computing the functions only for the absolute value of the argument, and then, for $\sinh(x)$ and $\tanh(x)$, if the argument is negative, inverting the sign of the computed result. That practice is common in mathematical software, but is incorrect for rounding modes other than the default of *round to nearest*. Preservation of mathematical identities is usually more important than how inexact results are rounded, but that would not be true for computation with interval arithmetic.

The Taylor series of the hyperbolic functions for $x \approx 0$ are:

$$\begin{aligned}
\cosh(x) &= 1 + (1/2!)x^2 + (1/4!)x^4 + \cdots + (1/(2n)!)x^{2n} + \cdots, \\
\sinh(x) &= x + (1/3!)x^3 + (1/5!)x^5 + \cdots + (1/(2n+1)!)x^{2n+1} + \cdots, \\
\tanh(x) &= x - (1/3)x^3 + (2/15)x^5 - (17/315)x^7 + (62/2835)x^9 + \cdots \\
&= \sum_{k=1}^{\infty} \frac{4^k(4^k-1)B_{2k}}{(2k)!} x^{2k-1}.
\end{aligned}$$

The first two converge quickly, and have simple coefficients, but the series for the hyperbolic tangent converges more slowly, and involves the *Bernoulli numbers*, B_{2k}, that we first met in the series expansion of the tangent (see **Section 11.2** on page 302), and that we discuss further in **Section 18.5** on page 568.

Like trigonometric functions of sums of two angles, hyperbolic functions with argument sums can be related to functions of each of the arguments:

$$\begin{aligned}
\cosh(x+y) &= \cosh(x)\cosh(y) + \sinh(x)\sinh(y), \\
\sinh(x+y) &= \sinh(x)\cosh(y) + \cosh(x)\sinh(y), \\
\tanh(x+y) &= \frac{\tanh(x) + \tanh(y)}{1 + \tanh(x)\tanh(y)}.
\end{aligned}$$

Those relations are used in the ELEFUNT tests of the hyperbolic functions.

The exponential function of a real argument is always positive, and satisfies the reflection rule $\exp(-|x|) = 1/\exp(|x|)$, so it would appear that the hyperbolic functions could be computed from a single exponential and at most four additional elementary operations.

There are three serious computational problems, however:

- The argument at which overflow happens in $\cosh(x)$ and $\sinh(x)$ is larger than that for $\exp(x)$. Thus, it is impossible to compute the two hyperbolic functions correctly by direct application of their definitions for arguments in the region where the hyperbolic functions remain finite, but $\exp(x)$ overflows.
- The subtractions in the definitions of $\sinh(x)$ and $\tanh(x)$ produce serious significance loss for $x \approx 0$.
- Hexadecimal arithmetic requires special consideration to combat the accuracy loss from wobbling precision.

We address those problems by dividing the argument interval $[0, \infty)$ into five consecutive regions:

$[\mathbf{0}, \boldsymbol{x}_{\mathbf{TS}}]$: use two terms of the Taylor series;

$(\boldsymbol{x}_{\mathbf{TS}}, \boldsymbol{x}_{\mathbf{loss}}]$: use rational polynomial approximation where there is bit loss from subtraction in $\sinh(x)$ and $\tanh(x)$, and otherwise, use the mathematical definition of $\cosh(x)$;

$(\boldsymbol{x}_{\mathbf{loss}}, \boldsymbol{x}_{\boldsymbol{c}}]$: use the mathematical definitions;

$(\boldsymbol{x}_{\boldsymbol{c}}, \boldsymbol{x}_{\mathbf{ovfl}}]$: special handling near function overflow limits;

$(\boldsymbol{x}_{\mathbf{ovfl}}, \boldsymbol{\infty})$: $\cosh(x) = \sinh(x) = \infty$, and $\tanh(x) = 1$.

Cody and Waite observe that premature overflow can be avoided by suitable translation of the argument, using the relation $\exp(x)/2 = \exp(x) \times \exp(-\log(2)) = \exp(x - \log(2))$. Unfortunately, because $\log(2)$ is a transcendental number, we have to approximate it, thereby introducing an argument error. Because the error-magnification factor of $\exp(x)$, and thus also of $\cosh(x)$ and $\sinh(x)$, is proportional to x (see **Table 4.1** on page 62), and x is large, that translation introduces large errors in the function, and is unsatisfactory.

Instead, we write $\exp(x)/2 = \exp(x)/2 \times (v/v) = v \exp(x - \log(v))/2$, and choose $\log(v)$ to be a constant slightly larger than $\log(2)$, but having the property that the number of integer digits in x plus the number of digits in $\log(v)$ does not exceed the significand size. The subtraction is then *exact*, and the argument translation introduces no additional error. Of course, we then have an additional error from the multiplication by $v/2$, but that should be at most one ulp, and independent of the size of x.

For bases that are powers of two, the value 45427/65536 is a suitable choice for the constant $\log(v)$. That value needs only 16 bits, and the overflow limit for the exponential in single-precision arithmetic requires at most 8 integer bits for current and historical architectures, so their sum can be represented in the 24 significand bits available. We need these stored constants:

$$\begin{aligned}
\log(v) &= 45427/65536, \\
v/2 - 1 &= 1.383\,027\,787\,960\,190\,263\,751\,679\,773\,082\,023\,374\ \ldots \times 10^{-5}, \\
1/v^2 &= 0.249\,993\,085\,004\,514\,993\,356\,328\,792\,112\,262\,007\ \ldots .
\end{aligned}$$

For base 10, assuming at least six significand digits, we use these values:

$$\begin{aligned}
\log(v) &= 0.6932, \\
v/2 - 1 &= 5.282\,083\,502\,587\,485\,246\,917\,563\,012\,448\,599\,540\ \ldots \times 10^{-5}, \\
1/v^2 &= 0.249\,973\,591\,674\,870\,159\,651\,646\,864\,161\,934\,786\ \ldots .
\end{aligned}$$

It is imperative to use the value $v/2 - 1$ directly, instead of $v/2$, because the latter loses about five decimal digits.

The PORT library authors recognize the problem of premature overflow, and compute the hyperbolic cosine with simple Fortran code like this:

```
t = exp(abs(x / 2.0e0))
cosh = t * (0.5e0 * t) + (0.5e0 / t) / t
```

The argument scaling is only error free in binary arithmetic, and the factorization introduces for all arguments two additional rounding errors that could easily be avoided. The PORT library hyperbolic sine routines use a similar expression with a subtraction instead of an addition, but do so only for $|x| \geq 1/8$. Otherwise, the PORT library code sums the Taylor series to convergence, starting with the largest term, thereby losing another opportunity to reduce rounding error by adding the largest term last.

In order to preserve full precision of the constants, and avoid problems from wobbling precision in hexadecimal arithmetic, for arguments near the function overflow limit, we compute the hyperbolic cosine and sine like this:

$$\begin{aligned}
u &= \exp(x - \log(v)),\\
y &= u + (1/v^2)(1/u),\\
z &= u - (1/v^2)(1/u),\\
\cosh(x) &= (v/2)\big(u + (1/v^2)(1/u)\big)\\
&= y + (v/2 - 1)y,\\
\sinh(x) &= (v/2)\big(u - (1/v^2)(1/u)\big)\\
&= z + (v/2 - 1)z.
\end{aligned}$$

For hexadecimal arithmetic, those formulas should be used even for moderate $|x|$ values, because if $\exp(x)$ has leading zero bits from hexadecimal normalization, $\exp(x - \log(v))$ is about half that size, has no leading zero bits, and retains full accuracy. When $\cosh(x)$ or $\sinh(x)$ have leading zero bits, they are computed from an exponential with a similar number of leading zero bits, so accuracy of the two hyperbolic functions is no worse. That special handling is easily done by reducing the limit x_{ovfl} for $\beta = 16$.

For large-magnitude arguments, $\tanh(x)$ quickly approaches its limits of ± 1. We could simply return those values as soon as $\exp(x) \pm \exp(-x)$ is the same as $\exp(x)$ to machine precision. However, to produce correct rounding, and setting of the *inexact* flag, at the limits, the return value must be computed as $\pm(1 - \tau)$ (Greek letter *tau*) for a suitable tiny number τ, such as the machine underflow limit, and that computation must be done at run time, rather than at compile time.

To find the cutoff for the limits in $\tanh(x)$, multiply the numerator and denominator by $\exp(-x)$, and then add and subtract $2\exp(-2x)$ in the numerator to find:

$$\begin{aligned}
\tanh(x) &= (1 - \exp(-2x))/(1 + \exp(-2x))\\
&= (1 + \exp(-2x) - 2\exp(-2x))/(1 + \exp(-2x))\\
&= 1 - 2/(\exp(2x) + 1).
\end{aligned}$$

The right-hand side is 1 to machine precision when the second term falls below half the negative machine epsilon, ϵ/β, where $\epsilon = \beta^{1-t}$ for t-digit arithmetic:

$$\begin{aligned}
\tfrac{1}{2}\epsilon/\beta &= \tfrac{1}{2}(\beta^{1-t}/\beta)\\
&= \tfrac{1}{2}\beta^{-t}\\
&= 2/(\exp(2x_c) + 1)\\
&\approx 2/\exp(2x_c),\\
\exp(2x_c) &= 4\beta^t,\\
x_c &= \tfrac{1}{2}(\log(4) + t\log(\beta)).
\end{aligned}$$

Similar considerations for $\cosh(x)$ and $\sinh(x)$ show that the $\exp(-x)$ terms can be ignored for x values larger than these:

$$x_c = \begin{cases} \tfrac{1}{2}(\log(2) + (t-1)\log(\beta)), & \textit{for } \cosh(x),\\ \tfrac{1}{2}(\log(2) + t\log(\beta)), & \textit{for } \sinh(x). \end{cases}$$

The cutoff values, x_c, can be easily computed from stored constants. Our result for $\tanh(x)$ differs slightly from the result $\frac{1}{2}(\log(2) + (t+1)\log(\beta))$ given by Cody and Waite [CW80, page 239], but the two are identical when $\beta = 2$. For other bases, their cutoff is slightly larger than needed.

For small-magnitude arguments, $\cosh(x)$ can be computed stably like this:

$$\begin{aligned}
w &= \begin{cases} \exp(|x|), & \textit{usually,}\\ \exp(-|x|), & \textit{sometimes when } \beta = 16\textit{: see text,} \end{cases}\\
\cosh(x) &= \tfrac{1}{2}(w + 1/w).
\end{aligned}$$

We have $\exp(|x|) \geq 1$, so the reciprocal of that value is smaller, and the rounding error of the division is minimized. However, when $\beta = 16$, we need to be concerned about wobbling precision. Examination of the numerical values of $\exp(|x|)$ show that it loses leading bits unless $|x|$ lies in intervals of the form $[\log(8 \times 16^k), \log(16 \times 16^k)]$, or roughly $[2.08, 2.77]$, $[4.85, 5.55]$, $[7.62, 8.32]$, ... Outside those intervals, the computed value of $\exp(-|x|)$ is likely to be more accurate than that of $\exp(|x|)$. However, the growth of the exponential soon makes the first term much larger than the second, so in our code, we use a block that looks similar to this:

```
#if B == 16
    if ( (FP(2.079442) < xabs) && (xabs < FP(2.772589)) )
        z = EXP(xabs);
    else
        z = EXP(-xabs);
#else
    z = EXP(xabs);
#endif

    result = HALF * (z + ONE / z);
```

For tiny arguments, the three functions can be computed from their Taylor series. From the first two terms of the series, and the formulas for the machine epsilons, we can readily find these cutoffs:

$$x_{\mathrm{TS}} = \begin{cases} \sqrt{6}\beta^{-t/2} & \textit{for } \sinh(x) \textit{ and } \tanh(x), \\ \sqrt{2\beta}\beta^{-t/2} & \textit{for } \cosh(x). \end{cases}$$

For simplicity, Cody and Waite use a smaller value, $\beta^{-t/2}$, for each of those cutoffs, and they keep only the first term of each of the series. However, in the mathcw library implementation, we sum two terms to set the *inexact* flag and obey the rounding mode.

For arguments of intermediate size, because of significance loss in the subtraction, $\sinh(x)$ and $\tanh(x)$ require rational polynomial approximations. In binary arithmetic, bit loss sets in as soon as $\exp(-x)$ exceeds $\frac{1}{2}\exp(x)$, which we can rearrange and solve to find

$$\begin{aligned} x_{\mathrm{loss}} &= \tfrac{1}{2}\log(2) \\ &\approx 0.347. \end{aligned}$$

However, Cody and Waite base their rational approximations on the wider interval $[0, 1]$, so we do as well. The polynomials provide as much accuracy as the exponential function, and are faster to compute, although they could be slightly faster if they were fitted on the smaller interval $[0, 0.347]$, reducing the total polynomial degree by about one.

If the macro USE_FP_T_KERNEL is not defined when the code is compiled, the next higher precision is normally used for $\frac{1}{2}(\exp(x) \pm \exp(-x))$ in order to almost completely eliminate rounding errors in the computed results, which for x in $[1, 10]$ otherwise can approach one ulp with *round-to-nearest* arithmetic. Experiments with Chebyshev and minimax polynomial fits over that interval demonstrate that polynomials of large degree defined on many subintervals would be necessary to achieve comparable accuracy, at least until we can identify a better approximating function that can be easily computed to effective higher precision using only working-precision arithmetic. A variant of the exponential function that returns a rounded result, along with an accurate estimate of the rounding error, would be helpful for improving the accuracy of the hyperbolic functions.

We omit the code for the hyperbolic functions, but Figure 12.2 on the following page shows measured errors in the values returned by our implementations. The largest errors are 0.95 ulps (cosh()), 1.00 ulps (sinhf()), and 1.21 ulps (tanh()), but the results from the decimal functions tanhdf() and tanhd() are almost always correctly rounded.

12.2 Improving the hyperbolic functions

Plots of the errors in the hyperbolic cosine and sine functions show that the difficult region is roughly $[1, 10]$, where the roundings from the reciprocation and addition or subtraction in $\frac{1}{2}(\exp(x) \pm 1/\exp(x))$ have their largest effect.

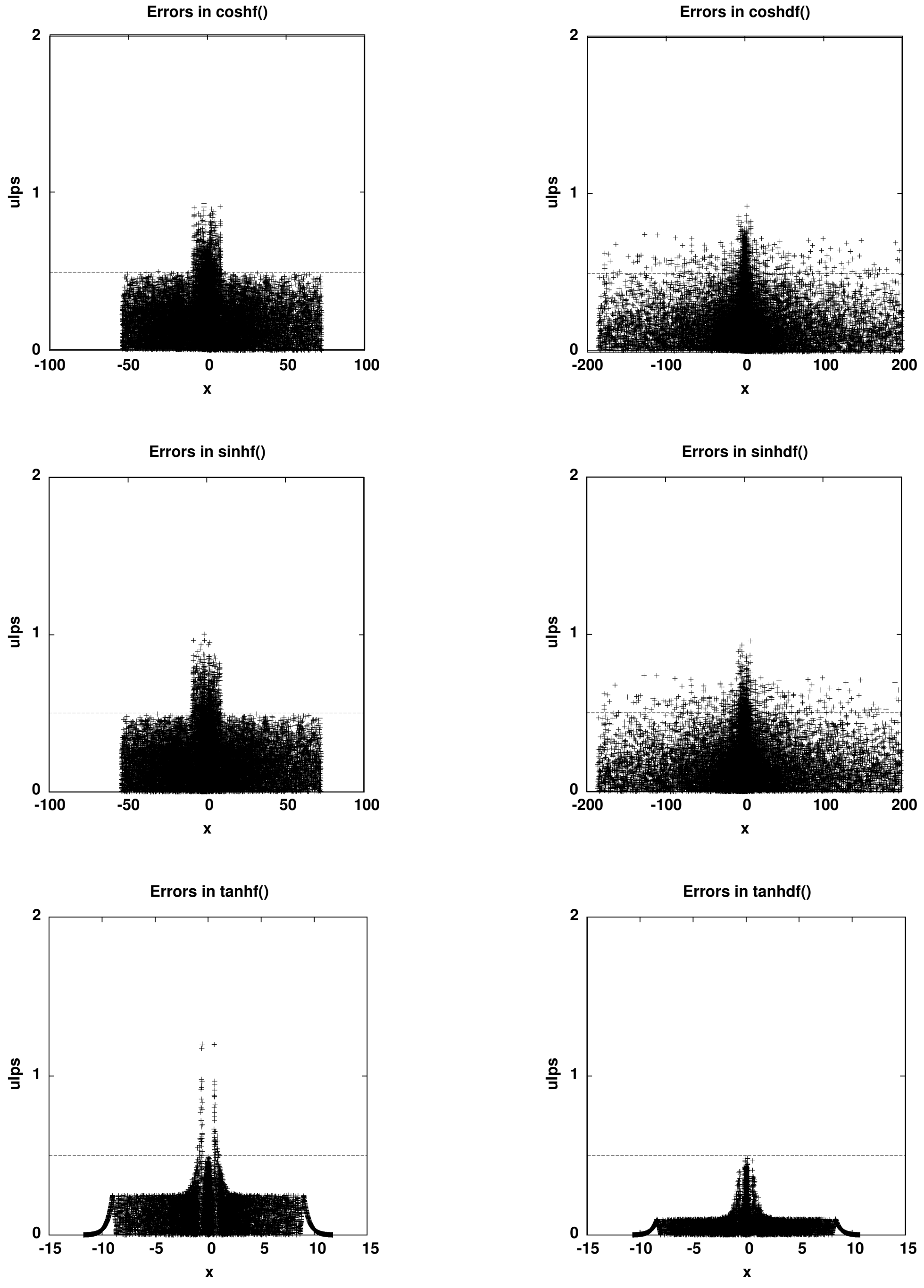


Figure 12.2: Errors in the single-precision hyperbolic functions for binary and decimal arithmetic *without* use of intermediate higher precision. Plots for the double-precision functions are similar, and thus, not shown.

The mathcw library code for $\cosh(x)$ and $\sinh(x)$ normally uses the next higher precision for the exponentials. When that is possible, for randomly chosen arguments, fewer than 1 in 10^7 function results are incorrectly rounded.

When higher precision is available, those errors are negligible. However, some platforms do not supply a usable `long double` type in C, and for all platforms, we would like to improve the accuracy of the functions at the highest available precision. Doing so requires computing the exponential function in even higher precision.

Because the Taylor series of the exponential function begins $1 + x + x^2/2! + \dots$, when x is small, we can achieve that higher precision by delaying addition of the leading term. Fortunately, we do not need to reimplement the exponential function, because we already have `expm1(x)` to compute the value of $\exp(x) - 1$ accurately. For larger x values, we can obtain small arguments as differences from tabulated exact arguments like this:

$$\begin{aligned}
e^x &= e^{x-c_k}e^{c_k} \\
&= (1 + \operatorname{expm1}(x - c_k)) \times (\exp(c_k)_{\rm hi} + \exp(c_k)_{\rm lo}) \\
&= \exp(c_k)_{\rm hi} + (\operatorname{expm1}(x - c_k) \times (\exp(c_k)_{\rm hi} + \exp(c_k)_{\rm lo}) + \exp(c_k)_{\rm lo}) \\
&= \exp(c_k)_{\rm hi} + \operatorname{fma}(\operatorname{expm1}(x - c_k), \exp(c_k)_{\rm hi}, \operatorname{fma}(\operatorname{expm1}(x - c_k), \exp(c_k)_{\rm lo}, \exp(c_k)_{\rm lo})) \\
&= \exp(x)_{\rm hi} + \exp(x)_{\rm lo}.
\end{aligned}$$

We need to choose enough c_k values on $[1, 10]$ to ensure that $1 + \operatorname{expm1}(x - c_k)$ represents $\exp(x - c_k)$ to a few extra digits, and so that k and c_k can be found quickly. Evenly spacing the c_k values $1/8$ unit apart means that $|x - c_k| < 1/16$, for which $\operatorname{expm1}(x - c_k)$ lies in $[-0.061, +0.065]$, giving us roughly two extra decimal digits at the expense of storing a table of 144 high and low parts of $\exp(c_k)$. For decimal arithmetic, we use a more natural spacing of $1/10$. The fused multiply-add operations allow accurate reconstruction of high and low parts of $\exp(x)$, such that their sum gives us about two more digits. That technique does not scale well: to get one more digit, we need a spacing of $1/64$, and a table of 1152 parts.

Once we have the higher-precision exponential, we can find the hyperbolic functions by careful evaluation of these expressions:

$$\begin{aligned}
H &= \exp(x)_{\rm hi}, \\
L &= \exp(x)_{\rm lo}, \\
\cosh(x) &= \tfrac{1}{2}((H + L) + 1/(H + L)), \\
\sinh(x) &= \tfrac{1}{2}((H + L) - 1/(H + L)).
\end{aligned}$$

We can recover the error, e, in the division like this:

$$\begin{aligned}
D &= 1/(H + L), && \textit{exact}, \\
d &= \operatorname{fl}(D), && \textit{approximate}, \\
D &= d + e, && \textit{exact}, \\
e &= D - d, && \textit{exact}, \\
&= (D \times (H + L) - d \times (H + L))/(H + L), && \textit{exact}, \\
&= D \times (1 - d \times (H + L)), && \textit{exact}, \\
&\approx d \times (1 - d \times (H + L)), && \textit{approximate}, \\
&\approx d \times (\operatorname{fma}(-d, H, 1) - d \times L).
\end{aligned}$$

Finally, we reconstruct the hyperbolic functions

$$\begin{aligned}
\cosh(x) &= \tfrac{1}{2}(H + (d + (L + e))), \\
\sinh(x) &= \tfrac{1}{2}(H + (-d + (L - e))).
\end{aligned}$$

using the accurate summation function family, `VSUM()`, to sum the four terms in order of increasing magnitudes.

Although that technique does not require a higher-precision data type, it gains only about two extra decimal digits in the exponential. We therefore use it only for the hyperbolic functions of highest available precision.

Tests of the code that implements that algorithm shows that it lowers the incidence of incorrect rounding of the hyperbolic cosine and sine to about 10^{-4}, with a maximum error of about 0.53 ulps. By contrast, using the next higher precision lowers the rate of incorrect rounding to below 10^{-7}, with a maximum error of about 0.51 ulps.

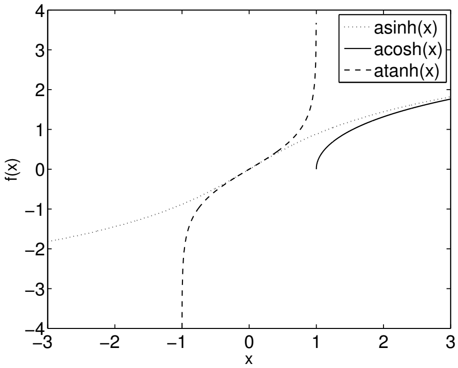


Figure 12.3: Inverse hyperbolic functions near the origin. An artifact of the MATLAB graphing software prevents display of the upper part of atanh(x).

12.3 Computing the hyperbolic functions together

Just as we found it efficient in **Section 11.8** on page 320 to compute the trigonometric cosine and sine together, we can do the same thing with their hyperbolic companions. The file `shchx.h` defines the function `sinhcosh(x, &sh, &ch)` and its companions with the usual suffixes for other data types.

The code in `shchx.h` is a straightforward interleaving of that from `coshx.h` and `sinhx.h`. They use the same algorithms, guaranteeing identical function results. When both the hyperbolic cosine and sine are needed, `sinhcosh()` is the recommended, and faster, way to compute them.

12.4 Inverse hyperbolic functions

The inverse hyperbolic functions, illustrated in **Figure 12.3** through **Figure 12.6** on page 350, all bear simple relations to the logarithm function that make them straightforward to compute:

$$\begin{aligned}
\operatorname{acosh}(x) &= \log(x + \sqrt{x^2 - 1}), && \text{for } x \geq 1, \\
\operatorname{asinh}(x) &= \operatorname{sign}(x)\log(|x| + \sqrt{x^2 + 1}), && \text{for } x \text{ in } (-\infty, +\infty), \\
\operatorname{atanh}(x) &= \tfrac{1}{2}\log(1 + 2x/(1 - x)), && \text{for } x \text{ in } [-1, +1].
\end{aligned}$$

The primary computational issues are premature overflow of x^2, and loss of significance from subtractions in the square roots and from evaluation of the logarithm for arguments near 1.0. For the latter, the log1p(x) function provides the required solution.

The inverse hyperbolic functions have these Taylor series:

$$\begin{aligned}
\operatorname{acosh}(1 + d) &= \sqrt{2d}\,(1 - (1/12)d + (3/160)d^2 - (5/896)d^3 + \\
&\qquad (35/18\,432)d^4 - (63/90\,112)d^5 + \cdots) \\
\operatorname{asinh}(x) &= x - (1/6)x^3 + (3/40)x^5 - (5/112)x^7 + (35/1152)x^9 - \\
&\qquad (63/2816)x^{11} + (231/13\,312)x^{13} - \cdots \\
\operatorname{atanh}(x) &= x + (1/3)x^3 + (1/5)x^5 + \cdots + (1/(2k+1))x^{2k+1} + \cdots
\end{aligned}$$

For a truncated series, we can reduce rounding error by introducing a common denominator to get exactly representable coefficients. For example, a five-term series for the inverse hyperbolic tangent can be evaluated with this code fragment from `atanhx.h`:

```
/* atanh(x) ~= (((945+(315+(189+(135+105 x^2)x^2)x^2)x^2)x^2)x)/945 */
```

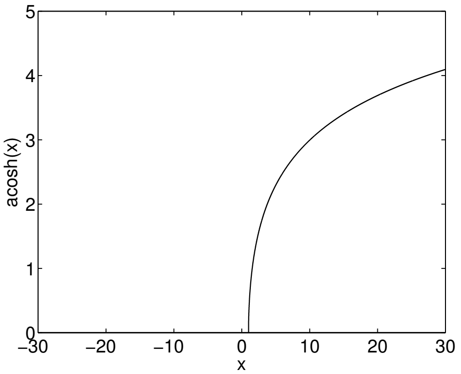


Figure 12.4: Inverse hyperbolic cosine. The function $\mathrm{acosh}(x)$ is defined only for x on the interval $[+1, \infty)$, and is slow to approach the limit $\lim_{x \to \infty} \mathrm{acosh}(x) \to \infty$. In the IEEE 754 32-bit format, the largest finite $\mathrm{acosh}(x)$ is less than 89.5, in the 64-bit format, about 710.5, and in the 80-bit and 128-bit formats, just under 11357.3.

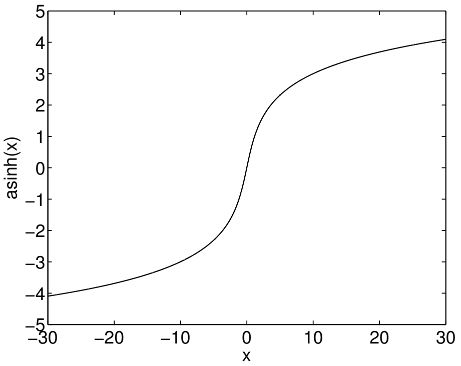


Figure 12.5: Inverse hyperbolic sine. The function $\mathrm{asinh}(x)$ is defined for x on the interval $(-\infty, +\infty)$, and is slow to approach the limit $\lim_{x \to \infty} \mathrm{asinh}(x) \to \infty$. In the IEEE 754 32-bit format, the largest finite $\mathrm{asinh}(x)$ is less than 89.5, in the 64-bit format, about 710.5, and in the 80-bit and 128-bit formats, just under 11357.3.

```
x_sq = x * x;
sum = (FP(105.0)      ) * x_sq;
sum = (FP(135.0) + sum) * x_sq;
sum = (FP(189.0) + sum) * x_sq;
sum = (FP(315.0) + sum) * x_sq;
sum *= xabs / FP(945.0);
result = xabs + sum;
```

Cody and Waite do not treat the inverse hyperbolic functions, so we base our code for those functions on the approach used in the Sun Microsystems' fdlibm library, but we also use the Taylor-series expansions to ensure correct behavior for small arguments. After the argument-range limits have been checked, the algorithms look like this for base β and a t-digit significand:

$$\mathrm{acosh}(x) = \begin{cases} \log(2) + \log(x) & \text{for } x > \sqrt{2\beta^t}, \\ \log(2x - 1/(\sqrt{x^2 - 1} + x)) & \text{for } x > 2, \\ \mathrm{log1p}(s + \sqrt{2s + s^2}) & \text{for } s = x - 1, \end{cases}$$

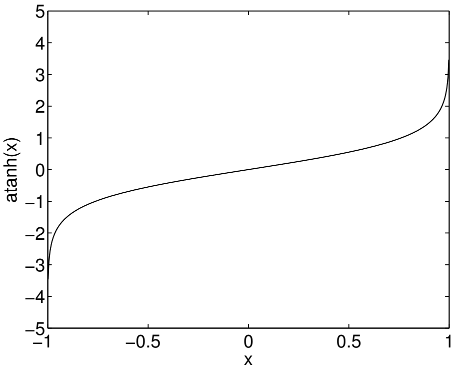


Figure 12.6: Inverse hyperbolic tangent. The function $\operatorname{atanh}(x)$ is defined only for x on the interval $[-1, +1]$, and has poles at $x = \pm 1$ that are only approximated in this plot.

$$\operatorname{asinh}(x) = \begin{cases} \operatorname{sign}(x)(\log(2) + \log(x)) & \textit{for } x > \sqrt{2\beta^{t-1}}, \\ \operatorname{sign}(x)\log(2|x| + 1/(|x| + \sqrt{x^2+1})) & \textit{for } |x| > 2, \\ x & \textit{if } \mathrm{fl}(x^2+1) = 1, \\ \operatorname{sign}(x)\operatorname{log1p}(|x| + x^2/(1+\sqrt{1+x^2})) & \textit{otherwise}, \end{cases}$$

$$\operatorname{atanh}(x) = \begin{cases} -\operatorname{atanh}(-x) & \textit{for } x < 0, \\ \frac{1}{2}\operatorname{log1p}(2x/(1-x)) & \textit{for } x \geq 1/2, \\ \frac{1}{2}\operatorname{log1p}(2x + 2x^2/(1-x)) & \textit{for } x \textit{ in } [0, 1/2). \end{cases}$$

Notice that for `acosh()` and `asinh()`, $\log(2x)$ is computed as $\log(2) + \log(x)$ to avoid premature overflow for large x. The cutoff $\sqrt{2\beta^t}$ is the value above which $\mathrm{fl}(x^2 - 1) = \mathrm{fl}(x^2)$ to machine precision, and $\sqrt{2\beta^{t-1}}$ is the corresponding cutoff that ensures $\mathrm{fl}(x^2+1) = \mathrm{fl}(x^2)$.

At most seven floating-point operations are needed beyond those for the logarithm and square-root functions, so as long as those two functions are accurate, the inverse hyperbolic functions are expected to be as well.

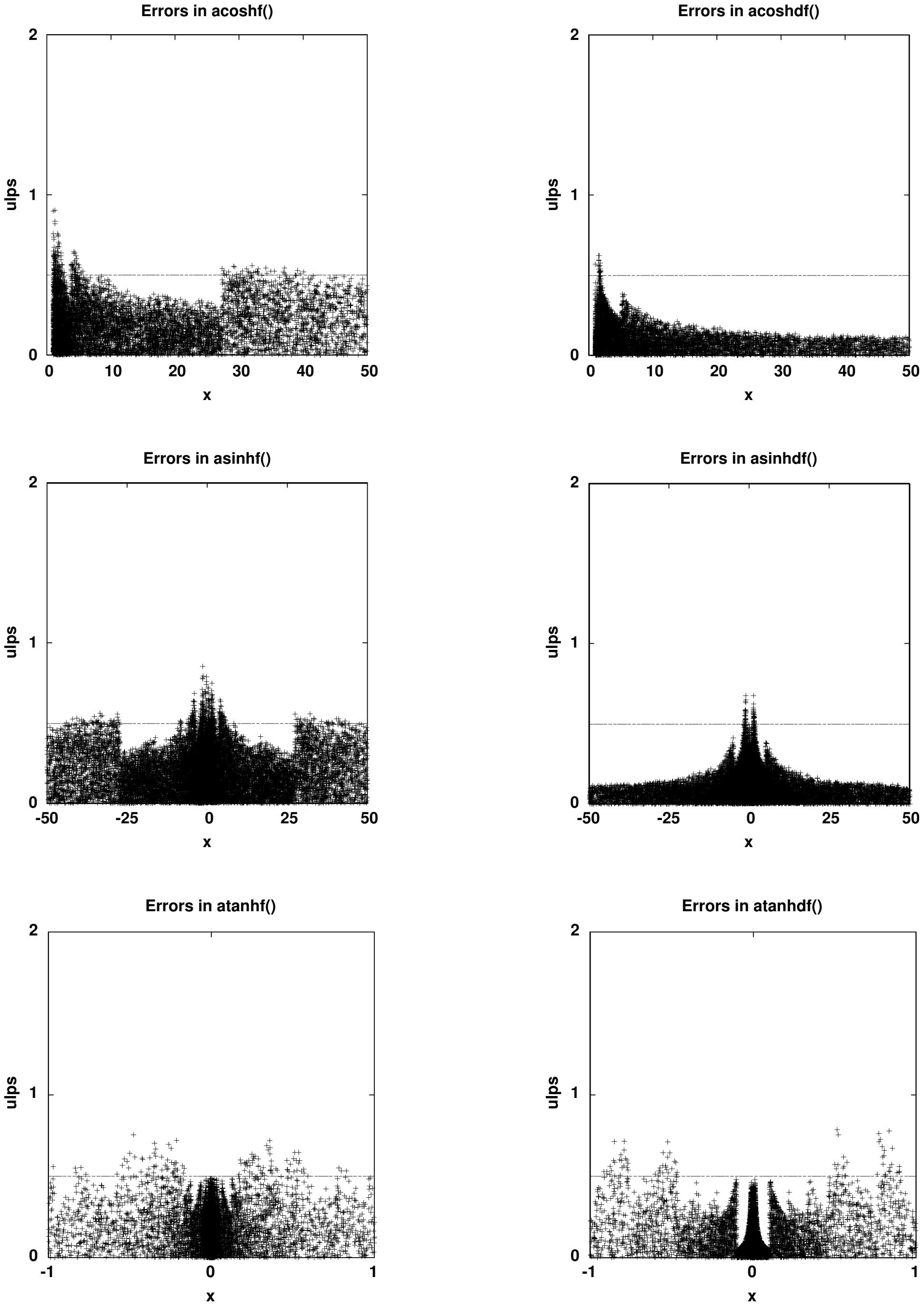
Figure 12.7 shows measured errors in the values returned by our implementations of the inverse hyperbolic functions when arithmetic is restricted to working precision. The largest errors found are 0.91 ulps (`acoshf()`), 0.88 ulps (`asinh()`), and 0.85 ulps (`atanhd()`). For the `ATANH()` family, computing just the argument of `LOG1P()` in the interval $[x_{\mathrm{TS}}, \frac{1}{2})$ in higher precision, and then casting it to working precision, reduces the maximum error by about 0.10 ulps. Code to do so is present in the file `atanhx.h`, but is disabled because we can do better.

The errors can be almost completely eliminated by excursions to the next higher precision in selected argument ranges. When possible, we do so for acosh() and asinh() for argument magnitudes in $(2, 70]$ for a nondecimal base, and in $[x_{\mathrm{TS}}, 2]$ for all bases. We do so as well for atanh() for argument magnitudes in $[x_{\mathrm{TS}}, 1)$. The Taylor-series region covers 16% to 21% of the argument range for the single-precision functions `atanhf()` and `atanhdf()`. As with the trigonometric functions and the ordinary hyperbolic functions, compile-time definition of the `USE_FP_T_KERNEL` macro prevents the use of higher precision.

When higher precision is not available, the code in `atanhx.h` uses separate rational polynomial approximations of the form $\operatorname{atanh}(x) \approx x + x^3\mathcal{R}(x^2)$ on the intervals $[0, \frac{1}{2}]$ and $[\frac{1}{2}, \frac{3}{4}]$ to further reduce the errors. In binary arithmetic, that has better accuracy than the logarithm formulas. In decimal arithmetic, the results are almost always correctly rounded.

12.5 Hyperbolic functions in hardware

Although the Intel IA-32 architecture provides limited hardware support for the square root, exponential, logarithm, and trigonometric functions, it has no specific instructions for the hyperbolic functions. However, they can be computed with hardware support from their relations to logarithms tabulated at the start of **Section 12.4** on page 348, as



Errors in acoshf()

Errors in acoshdf()

Errors in asinhf()

Errors in asinhdf()

Errors in atanhf()

Errors in atanhdf()

Figure 12.7: Errors in the single-precision hyperbolic functions for binary and decimal arithmetic *without* use of intermediate higher precision. Plots for the double-precision functions are similar, and thus, not shown.
The mathcw library code for those functions normally uses the next higher precision for the logarithms and square roots. When that is possible, for randomly chosen arguments, fewer than 1 in 10^7 function results are incorrectly rounded.

Table 12.1: Maximum errors in units in the last place (ulps), and percentages of results that are correctly rounded, for hyperbolic functions in the PORT library on AMD64.

Function	32-bit		64-bit	
	max err	OK	max err	OK
`acosh()`	512.4	95.9%	5619.8	89.7%
`asinh()`	4.7	94.3%	4.8	94.5%
`atanh()`	2.2	80.8%	3.8	76.5%
`cosh()`	1.9	72.1%	1.9	70.2%
`sinh()`	10.7	68.3%	9.3	66.9%
`tanh()`	3.5	82.6%	3.7	81.6%

Table 12.2: Maximum errors in units in the last place (ulps), and percentages of results that are correctly rounded, for inverse hyperbolic functions in the FNLIB library on AMD64.

Function	32-bit		64-bit	
	max err	OK	max err	OK
`acosh()`	10^9	0.0%	1790.0	94.8%
`asinh()`	10^7	7.3%	1.1	96.6%
`atanh()`	10^5	34.4%	1.0	85.6%

long as the argument x is not so large that x^2 overflows. A better approach is to employ the three alternate forms involving logarithms of modified arguments. The mathcw library code does not use inline IA-32 assembly code for the hyperbolic functions, but it can do so for the needed logarithms, at the slight expense of an additional function call.

The Motorola 68000 has `fatanh`, `fcosh`, `fsinh`, and `ftanh` instructions. No single instructions exist for the inverse hyperbolic cosine and sine, but they can be computed with the help of the 68000 hardware instructions `flogn` and `flognp1` for the logarithms. The mathcw library uses those hardware instructions when it is built on a 68000 system with a compiler that supports inline assembly code, and the preprocessor symbol `USE_ASM` is defined. The only 68000 machine available to this author during the library development lacks support for the `long double` data type, so it has been possible to test the hyperbolic-function hardware support only for the 32-bit and 64-bit formats.

12.6 Summary

Simplistic application of the mathematical definitions of the hyperbolic functions and their inverses in terms of the exponential, logarithm, and square root does not lead to acceptable accuracy in software that uses floating-point arithmetic of fixed precision.

Plots of the errors of those functions as implemented in the PORT library [FHS78b] show reasonable accuracy, except for the `acosh()` pair, as summarized in Table 12.1. Errors in the `acosh()` functions rise sharply above 2 ulps for x in $[1, 1.1]$, and the errors increase as x decreases.

Results of tests of the inverse hyperbolic functions in the earlier FNLIB library [Ful81b, Ful81a] are shown in Table 12.2. Although 85% or more of the function values are correctly rounded, the tests find huge errors for $x \approx 1$ in the inverse hyperbolic cosine and sine, and for $x \approx 0.5$ in the inverse hyperbolic tangent.

Tests with the newer software technology in the GNU math library show worst case errors of 1.32 ulps in `tanhf()` and `tanh()`, and 1.11 ulps for the other hyperbolic functions. From 88% (`tanhf()`) to 98% (`acosh()`) of the results are correctly rounded. Tests of the fdlibm library show similar behavior.

Our implementations of the hyperbolic functions and their inverses provide results that are almost always correctly rounded, and have accuracy comparable to our code for the related exponential and logarithm functions. Achieving that goal requires considerable care in algorithm design, and in programming.

13 Pair-precision arithmetic

> THE COMPUTATION OF `a*b - c*d` IN JAVA WILL PRODUCE THE SAME RESULT EVERYWHERE, BUT IT MAY BE THE SAME VERY WRONG RESULT EVERYWHERE!
>
> — FRED G. GUSTAVSON, JOSÉ E. MOREIRA AND ROBERT F. ENENKEL (1999).

In a few places in the mathcw library, we need to work with an intermediate precision having at least twice as many digits as storage precision. In almost all historical floating-point architectures, for single-precision computation, the double-precision format satisfies that requirement (see **Table H.1** on page 948). However, for double-precision work, higher precision may be unavailable, as is the case in many programming languages, or it may be insufficiently precise, such as the IEEE 754 80-bit format compared to the 64-bit format.

The solution to that problem is to represent floating-point numbers as the sum of two values, a high-order part, and a low-order part. That technique is sometimes referred to as *doubled-double arithmetic*, but that term can be confusing, and in any event, we want to use that idea for any machine precision. A better description seems to be the phrase *pair-precision arithmetic* that we use in the title of this chapter.

In 1965, Kahan [Kah65] and Møller [Møl65b] independently reported that the rounding error in floating-point summation can be recovered like this:

```
sum = x + y;
err = (x - sum) + y;
```

The parentheses indicate the order of computation, and the expression requires that $|x| \geq |y|$, for otherwise, the computed error may be zero, when the true error is nonzero.

Mathematically, the error in summation is always exactly zero. Computationally, because of finite precision, with numbers of similar sign and magnitude, and *round-to-nearest* addition, the computed error is nonzero about half the time.

Kahan's one-column note reports that the error term can be computed as indicated only on certain machines, such as the IBM models 704, 709, 7040, 7044, 7090, 7094, and System/360, which normalize results before rounding or truncating. The computation fails to produce a correct error term on those that round or truncate before normalization, such as the CDC 3600, the IBM 620 and 1650, and the Univac 1107. Most CPU architectures produced since the early 1970s fall into the first group, and IEEE 754 arithmetic certainly does.

In his much longer paper, after a detailed analysis of floating-point error, Møller gives a more complex correction that he terms 'monstrous':

```
err = (y - (sum - x)) +
      (x - (sum - (sum - x))) +
      (y - ((sum - x) + (y - (sum - x))));
```

The term in the first line matches that of Kahan, and is the largest of the three.

Møller credits his insight to earlier work by Gill in 1951 in fixed-point computation of solutions of ordinary differential equations on the EDSAC computer at the Cambridge University Mathematical Laboratory [Gil51]. However, although Gill carried an error term in his computations, it is not clear whether he realized how it could be computed, and no simple formula for the error term is evident in his paper.

In 1971, Dekker [Dek71] took the simpler Kahan/Møller formula, and developed portable algorithms for add, subtract, multiply, divide, and square root, without needing to decode the floating-point format. He wrote the code in Algol 60, but the translation to C is simple.

This chapter describes implementations of Dekker's primitives, and more than a dozen others, that allow programming with pair-precision arithmetic. In one important case, we use results of later work that broadens the applicability of Dekker's algorithms (see **Section 13.16** on page 368).

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_13

Near the end of this chapter, we include an important routine for accurate summation, based on the Kahan/Møller idea, because the results `(sum,err)` are nicely representable in pair-precision arithmetic. That routine then serves as the basis for accurate computation of dot products, which lie at the core of algorithms for arithmetic with complex numbers, for calculating the roots of quadratic equations, and for computing the areas of thin triangles [GME99]. The simplest case to which the dot-product code applies is mentioned in the epigraph that begins this chapter.

13.1 Limitations of pair-precision arithmetic

Pair-precision arithmetic is not ideal. What we would much prefer to have is hardware-assisted arithmetic of any user-specifiable precision, with range increasing with precision. Increased range is essential if we are to be able to compute products without danger of overflow or underflow. Pair-precision arithmetic doubles precision, but leaves the exponent range the same, so we still have to deal with issues of scaling to prevent destructive overflow and underflow. The economic realities of computer-chip manufacturing make it unlikely that such hardware support will be available in the foreseeable future, so we have to make do with software solutions.

Pair-precision arithmetic can lead to computational surprises. For example, the machine epsilon can normally be obtained by this C code:

```
eps = 1.0;

while ((1.0 + eps/BASE) != 1.0)
    eps /= BASE;
```

Besides the many uses of the machine epsilon in numeric computations, the floor of the negative base-10 logarithm of the machine epsilon is the number of decimal digits that the arithmetic supports. Although the C and C++ languages provide standard names in system header files for the machine epsilons in each precision, most languages do not. It is therefore useful to be able to compute the machine epsilon dynamically, instead of relying on a platform-dependent compile-time constant.

In pair-precision arithmetic, after several iterations of the loop, the high part is exactly 1.0, and the low part holds the value `eps/BASE`. The low part remains nonzero until `eps` reaches the smallest representable floating-point value, that is, the underflow limit. That value is *much* smaller than the machine epsilon, and the loop computation therefore produces a wildly wrong, or at least unexpected, answer. The basic problem is that there is no constraint on the relative sizes of the high and low parts, as there would be if we had a single exponent and a double-length significand. We show an alternate algorithm for computing the pair-precision machine epsilon in **Section 23.2** on page 779.

Code similar to the machine-epsilon loop exists in many iterative solvers, which continue computation until a computed tolerance is negligible to machine precision, using code with this outline:

```
do
{
  ... some computation ...
  tolerance = ... estimate of error ...
}
while (1.0 != (1.0 + tolerance))
```

In ordinary floating-point arithmetic, with reasonable rounding behavior, the loop terminates once the tolerance falls below the machine epsilon. In pair-precision arithmetic, it may take much longer to complete, or it may fail to terminate if the computed tolerance never reaches zero.

For sufficiently small values, the low component can underflow to a subnormal number, or abruptly to zero. Low-order bits are gradually or precipitously lost, and the precision of the number can be reduced to half its normal value. That happens in base β and t-digit precision as soon as the number comes within β^t of the smallest normal number. If a proper double-length significand with a single exponent were used, loss of trailing bits would not happen until numbers below the smallest normal number were reached. That misfeature of pair-precision arithmetic strikes in the computation of elementary functions (see **Chapter 23** on page 777), where normally exact scaling operations become inexact. Test programs for the elementary functions have to take that abnormal precision loss into account.

Another issue with pair-precision arithmetic is that the signs of the two parts may differ, because they are obtained independently by floating-point computations. The approach that we adopt in this chapter when the sign is needed is to test the sign of the sum of the two parts.

Signed zero, Infinity, and NaN components also pose problems: if the high part is one of those special values, then the low part should be as well, but might not be, because of its separate computation. When we need to generate explicitly a pair-precision number that has the value of a signed zero, Infinity or NaN, we set both parts to the same value.

Finally, comparisons of pair-precision numbers need careful handling. It is not sufficient to compare just the high parts, because even if they are equal, the low parts could have opposite signs. We therefore compare the difference against zero.

Pair-precision arithmetic was used in some historical systems to implement double precision when the hardware supported only single-precision computation. Among systems available to the author during the development of the `mathcw` library, pair-precision arithmetic is used for the C `long double` data type on IBM AIX PowerPC and Silicon Graphics IRIX MIPS systems.

13.2 Design of the pair-precision software interface

Modern programming practice is to provide a software interface that supplies an extensive, or better, complete, set of operations on data, without exposing implementation details to the user. Once the interface is specified, then internal implementation details can be modified without impacting user software. If the data are more than just simple scalars represented with standard data types, then new data types are called for. The data types should be *opaque*, providing type names, but hiding implementation details. Type names and function names should be related, so that the programmer can learn how to use the package more easily. In addition, the programmer should be able to program with the new data types without knowledge of their internal representation. That requires provision of functions for at least creation, destruction, initialization, assignment, member access, and type conversion. I/O support is also highly desirable. For numerical data, it should be possible to overload the arithmetic operators for add, subtract, multiply, and divide, as well as the relational operators, so the programmer can write conventional numeric and logical expressions, instead of a series of function calls.

We cannot meet all of those goals without restricting ourselves to a handful of newer programming languages. Ada, C++, and Fortran 90 offer the needed capabilities, but C does not. Nevertheless, as several appendices of this book demonstrate, by writing code in C, we can make it available to other programming languages, and portable to all major computing platforms.

The interface to the pair-precision functions is defined in a single header file, `paircw.h`, that is one of several such files provided for general use when the `mathcw` library is built and installed. That header file declares prototypes of all of the functions, and definitions of the opaque data types.

The type names are made by suffixing the underlying numeric type name with the word `pair`, and then converting spaces to underscores. Thus, we provide types named `float_pair`, `double_pair`, `long_double_pair`, and `long_long_double_pair`. Hewlett–Packard HP-UX IA-64 systems have two more: `extended_pair` and `quad_pair`, with synonyms `__float80_pair` and `__float128_pair`, respectively. When compiler support for decimal types is available, we also have the types `decimal_float_pair`, `decimal_double_pair`, `decimal_long_double_pair`, and `decimal_long_long_double_pair`.

Table 13.1 on the next page shows a complete summary of the functions, which are all named with the prefix `p`, for *pair*. Because few programming languages support the returning of multiple function values, the functions return the pair result in the *first* argument. That design choice is analogous to assignment statements: in almost all programming languages, the destination precedes the source.

Following the conventions used throughout this book, we show implementation code that is independent of the precision, and we name functions with uppercase macros that expand to lowercase names with one of the conventional precision suffixes. Thus, the macro `PADD()` represents the functions `paddf()`, `padd()`, and `paddl()` for the three C types `float`, `double`, and `long double`, respectively. There are additional companion functions for extended data types supported by some compilers.

The function implementations use private generic types `fp_pair_t` and `fp_t`. Those types expand to suitable public pair-precision and scalar data types, as defined and used in the header file `paircw.h`.

With the exception of four functions, the pair-precision functions are all of type `void`: their actions are recorded

Table 13.1: Pair-precision primitives. Only the `double` versions are listed here, but variants for all other supported precisions are available, with standard suffixes. Functions with an initial argument `r` are of type `void`, and return their result in that argument. Arguments `a`, `b`, `c`, and `d` are scalar floating-point values. Arguments `v` and `w` are `n`-element floating-point vectors. Arguments `r`, `x`, and `y` are scalar pair-precision values.

`pabs(r,x)`	absolute value	`pmul2(r,a,b)`	multiply scalars
`padd(r,x,y)`	add	`pmul(r,x,y)`	multiply
`pcbrt(r,x)`	cube root	`pneg(r,x)`	negate
`pcmp(x,y)`	compare	`pprosum(r,a,b,c,d)`	product and sum
`pcopy(r,x,y)`	copy	`pset(r,x,y)`	set
`pdiv(r,x,y)`	divide	`psplit(r,x)`	split
`pdot(r,n,v,w)`	vector dot product	`psqrt(r,x)`	square root
`peval(x)`	evaluate	`psub(r,x,y)`	subtract
`phigh(x)`	high part	`psum2(r,x,y)`	sum and error
`plow(x)`	low part	`psum(r,n,v)`	vector sum

entirely through their result arguments. The exceptions are `PCMP()`, which returns an `int` value, and `PEVAL()`, `PHIGH()`, and `PLOW()`, which return a value of the generic type `fp_t`.

Because we want to be able to interface the `mathcw` library to other languages, pair-precision types are implemented as two-element arrays, which almost all programming languages can easily handle. The C language guarantees that array arguments are passed *by address*, rather than *by value*, and that practice is almost universal among other programming languages. For efficiency, we use array-element indexing inside most of the functions. However, user code should avoid doing so, and call the accessor functions instead.

We guarantee that the result argument can overlap any of the input arguments, because that often proves convenient in a chain of pair-precision operations, and avoids unnecessary data copying in user code.

Except for the result argument, all arguments that are not simple scalar types are declared with the `const` qualifier, to make it clear that their contents are not modified in the pair-precision routines. That is, of course, not true if any input argument coincides with the result argument.

13.3 Pair-precision initialization

The constructor function `PSET()` initializes the components of a pair-precision value from scalar values:

```
void
PSET(fp_pair_t result, fp_t hi, fp_t lo)
{
    /* set result from high and low parts */

    result[0] = hi;
    result[1] = (lo == ZERO) ? COPYSIGN(lo, hi) : lo;
}
```

The special handling of the sign of a zero low part ensures correct behavior later.

The `PSET()` function receives extensive use in the other pair-precision routines, because it enhances data abstraction and shortens code.

Our code for `PSET()` does *not* enforce the requirement of some of Dekker's algorithms that the magnitude of the low part not exceed an ulp of the high part. The `PSUM2()` function that we describe later does that job, and it should be used when the relative magnitudes of the two parts are unknown.

`PSET()` is the type-cast function for converting ordinary floating-point values to pair-precision numbers. Its inverse is the `PEVAL()` function presented in the next section.

13.4 Pair-precision evaluation

We define the floating-point value of a pair-precision number to be the sum of its components:

```
fp_t
PEVAL(const fp_pair_t x)
{
    /* return the value of a pair-precision number as its sum of parts */

    return (x[0] + x[1]);
}
```

The PEVAL() function serves as the type-cast function for converting a pair-precision number to an ordinary floating-point number. Its inverse operation is PSET().

13.5 Pair-precision high part

The accessor function PHIGH() returns the high part of a pair-precision number:

```
fp_t
PHIGH(const fp_pair_t x)
{
    /* return the high part of x */

    return (x[0]);
}
```

For reasons of low-level efficiency, we do not invoke accessor functions internally in the pair-precision routines, but user code should do so to avoid exposing details of the implementation of the pair types as two-element arrays. Such access should be rare in any event, because the representation of pair-precision numbers is seldom of interest to user code.

13.6 Pair-precision low part

The accessor function PLOW() returns the low part of a pair-precision number:

```
fp_t
PLOW(const fp_pair_t x)
{
    /* return the low part of x */

    return (x[1]);
}
```

User code should use the PLOW() function family instead of explicit array-element access.

13.7 Pair-precision copy

The code to copy one pair-precision value to another is simple:

```
void
PCOPY(fp_pair_t result, const fp_pair_t x)
{
    /* return a copy of the pair-precision value of x in result */

    result[0] = x[0];
    result[1] = x[1];
}
```

For data abstraction and brevity, we use `PCOPY()` several times in the pair-precision routines.

13.8 Pair-precision negation

We sometimes need to negate a pair-precision number. That simple operation is defined by $-(x+y) = -x - y$, and the code is obvious:

```
void
PNEG(fp_pair_t result, const fp_pair_t x)
{
    /* return the pair-precision negative of x in result */

    PSET(result, -x[0], -x[1]);
}
```

13.9 Pair-precision absolute value

The absolute value of a pair-precision number is not simply the absolute value of each component, because we cannot guarantee that the signs of the components are the same. Instead, we use the comparison function, `PCMP()` (see **Section 13.15** on page 368), and then either copy or negate the value, depending on the sign of the comparison result:

```
void
PABS(fp_pair_t result, const fp_pair_t x)
{
    /* return the absolute value of x in result */

    static fp_pair_t zero = { FP(0.0), FP(0.0) };

    if (PCMP(x, zero) >= 0)
        PCOPY(result, x);
    else                      /* negative or unordered */
        PNEG(result, x);
}
```

As long as the magnitude of the input high part exceeds that of the low part, the high part of the result is positive, but the low part need not be.

13.10 Pair-precision sum

The error-corrected sum of two scalar floating-point values is one of three central operations in pair-precision arithmetic. The Kahan/Møller summation formula introduced at the beginning of this chapter looks trivial, in that it can be expressed in just two simple statements, once the magnitudes of the summands are known.

The simplicity of the summation formula hides peril: if the compiler uses higher-precision arithmetic for the computation, or 'optimizes' the computation by discarding the error term, the results are incorrect, and many of the pair-precision routines are then also wrong. Thus, it is unwise to code the summation formula inline, as is commonly done in numerical software. Instead, we provide it as a function that hides the details.

As we discuss elsewhere in this book, two solutions to the problem of higher intermediate precision are available to the C programmer. With C89 and C99 compilers, the `volatile` qualifier forces the compiler to keep the variable in memory. With older compilers, and in other programming languages, it is necessary to force the compiler to move the variable from the CPU to memory by calling a routine that receives the *address* of the variable. On return from the storage routine, the value must then be loaded from memory into the CPU when it is next required. By compiling the storage routine separately, we conceal from the compiler of `PSUM2()` the fact that the routine does nothing, and thereby prevent the compiler from optimizing away the call. Because almost all C compilers now support the `volatile` qualifier, the default definition of `STORE()` is an empty expansion.

Here is the code for the `PSUM2()` procedure:

```
void
PSUM2(fp_pair_t result, fp_t x, fp_t y)
{
    /* return the pair sum and error for x + y in result */

    volatile fp_t err, sum;

    sum = x + y;
    STORE(&sum);

    if (FABS(x) >= FABS(y))
    {
        err = x - sum;
        STORE(&err);
        err += y;
        STORE(&err);
    }
    else
    {
        err = y - sum;
        STORE(&err);
        err += x;
        STORE(&err);
    }

    PSET(result, sum, err);
}
```

The reader should note the difference between that routine and the `PSET()` routine. `PSUM2()` alters the two components so that the high part is large, and the low part is a correction that is smaller by a factor about the size of the machine epsilon. By contrast, `PSET()` simply stores the two parts in the result, and adjusts the sign of a zero low part.

When the relative sizes of the components are unknown, use `PSUM2()` instead of `PSET()` for initialization.

In recent research articles on floating-point arithmetic, the `PSUM2()` operation is often called `twosum()`. A companion function, `quicktwosum()`, is sometimes defined as well; it assumes that the first argument has a larger magnitude than the second, without checking whether that is true or not. Having two similar functions available to users is likely to be confusing and error prone, and the presence of the phrase *quick* in the function name is likely to encourage its use, even when it is not valid. We therefore omit such a function from the `mathcw` library.

Conditional tests are undesirable in high-performance code, because they can flush and refill the instruction pipeline, delaying processing. There is a variant of the `PSUM2()` operation that does more computation, but avoids the conditionals. It is based on code like this:

```
sum = x + y;
tmp = sum - x;
err = (x - (sum - y)) + (y - tmp);
```

As usual, parentheses must be obeyed, and higher intermediate precision suppressed. Depending on the compiler and the platform, it may be faster or slower than the `PSUM2()` code. We do not provide a separate function for the conditional-free algorithm because two different functions for the same purpose would confuse users.

13.11 Splitting numbers into pair sums

> DOUBLE ROUNDING IS HARMFUL.
>
> — EXPERIMENTAL EVIDENCE
> IN THIS SECTION (2006).

Table 13.2: Splitting numbers into sums of parts. We take a simple case of precision $t = 5$, and display the binary representations of all numbers in $[1, 2)$, splitting them into sums and differences. The bold entries show the cases with the minimal number of bits in the low part. The two bold cases marked with the arrow (⇐) have a low part of 0.0100, but that in reality requires only one or two bits, because the floating-point exponent can be adjusted upward to represent 0.0010 and 0.0001.

1.0000	=	**1.0000 + 0.0000**	=	1.1000 − 0.1000
1.0001	=	**1.0000 + 0.0001**	=	1.1000 − 0.0111
1.0010	=	**1.0000 + 0.0010**	=	1.1000 − 0.0110
1.0011	=	**1.0000 + 0.0011**	=	1.1000 − 0.0101
1.0100	=	1.0000 + 0.0100	=	**1.1000 − 0.0100** ⇐
1.0101	=	1.0000 + 0.0101	=	**1.1000 − 0.0011**
1.0110	=	1.0000 + 0.0110	=	**1.1000 − 0.0010**
1.0111	=	1.0000 + 0.0111	=	**1.1000 − 0.0001**
1.1000	=	**1.1000 + 0.0000**	=	10.000 − 0.1000
1.1001	=	**1.1000 + 0.0001**	=	10.000 − 0.0111
1.1010	=	**1.1000 + 0.0010**	=	10.000 − 0.0110
1.1011	=	**1.1000 + 0.0011**	=	10.000 − 0.0101
1.1100	=	1.1000 + 0.0100	=	**10.000 − 0.0100** ⇐
1.1101	=	1.1000 + 0.0101	=	**10.000 − 0.0011**
1.1110	=	1.1000 + 0.0110	=	**10.000 − 0.0010**
1.1111	=	1.1000 + 0.0111	=	**10.000 − 0.0001**

The key idea in Dekker's operations is that floating-point arithmetic can be *exact*, provided that we can avoid producing more digits than we can store, and provided that the underlying floating-point system is not badly botched, as was unfortunately the case on some systems before IEEE 754 arithmetic was developed. We can do that by splitting each number into the exact sum of two parts. The high part has no more than half the digits, and the low part has the remaining digits.

Dekker made this critical observation: if the low part is permitted to be negative as well as positive, then it can be represented with *one less bit than otherwise*, as long as the floating-point base is 2. That is not obvious, but Dekker gave a mathematical proof that we can give a flavor of by looking at the five-bit numerical experiments shown in **Table 13.2**. In each of the bold entries in the table, the high part is represented in two bits, and the low part also in two bits, instead of the expected three. The fifth bit has not disappeared: it has just moved into the sign bit of the low part.

That discovery means that products of two high parts, or of a high and a low part, or of two low parts, are then *exactly representable*. That in turn makes it possible to compute exact double-length products as the sum of four ordinary floating-point values that can be closely represented by a sum of two values.

In integer arithmetic, we can extract the high part of an n-digit value by first shifting right by $\mathrm{ceil}(n/2)$ digits, discarding low-order digits, and then shifting left by the same amount, inserting zeros. For example, in a five-digit decimal integer system, 12345 is reduced to 12000 by the three-digit shift operations. The low part is then easily recovered by subtraction: $12345 - 12000 = 345$. The original number has thus been split into an exact sum of high and low parts: $12345 = 12000 + 345$.

In floating-point arithmetic, shift operations are not available, but we can simulate them by multiplication by an integral power of the base, an operation that is always exact, *as long as underflow and overflow are avoided*. For a general t-digit floating-point system with base β, we can split a number into high and low parts like this:

$$\begin{aligned} s &= \beta^{\lceil t/2 \rceil}, \\ x_{\mathrm{hi}} &= (sx + x) - sx \\ &= (s+1)x - sx, \\ x_{\mathrm{lo}} &= x - x_{\mathrm{hi}}. \end{aligned}$$

Given precomputed values of $s + 1$ and s, the formula for the high part requires two multiplies and one add. However, we can change a slower multiply into a faster add by adding and subtracting x in the first equation for the high part:

$$\begin{aligned}
x_{\text{hi}} &= (sx + x) - sx + x - x \\
&= (s + 1)x - ((s + 1)x - x) \\
&= (s + 1)x + (x - (s + 1)x), \\
p &= (s + 1)x, \\
q &= x - p, \\
x_{\text{hi}} &= p + q, \\
x_{\text{lo}} &= x - x_{\text{hi}}.
\end{aligned}$$

Experiments show that the two alternatives do not always produce the same splits when the number of significand bits is even. For example, the value `0x1.ffe73cp-5` is split into `0x1.000000p-4 - 0x1.8c4000p-17` with the first algorithm, and into `0x1.ffe000p-5 + 0x1.cf0000p-19` with the second. Such differences are harmless and rare.

In most programming languages, for a given floating-point data type, the precision and base are architecture-dependent constants. That means that the split value, $s + 1$, can be a compile-time constant. The split cost is one multiply, three adds, five loads, and four stores. On modern systems, the loads and stores are likely to be the most expensive part.

Dekker notes three important restrictions on the applicability of the splitting algorithm:

- The floating-point base must be 2.
- Multiplication must be "faithful", rounding up or down, with at most a one-ulp error.
- Addition and subtraction must be "optimal", producing a result with at most a half-ulp error.

With those limitations, then the low part has the required sign that permits it to be represented with one less bit.

Those restrictions can be severe. The algorithm does not work properly with decimal arithmetic, or with IBM System/360 hexadecimal arithmetic, or in rounding modes other than the default in IEEE 754 arithmetic. It *does* produce a split into high and low parts, but for t-digit precision, the low part has ceil$(t/2)$ digits instead of ceil$(t/2) - 1$ digits, and the product of two low parts is then exact only if t is even. The IEEE 754 specification of decimal floating-point arithmetic has precisions of $t = 7, 16$, and 34, so for the first of them, splitting does not lead to exact products of low parts. We show later that the third restriction on the accuracy of addition and subtraction can be loosened.

Dekker's mathematical proof of the splitting algorithm [Dek71, pages 234–235] is complex, and we omit proofs in this book. However, we can see what happens by doing the split for just one of the entries in **Table 13.2** on the facing page. Once again we work in binary arithmetic:

$$\begin{aligned}
x &= 1.1101, \\
s &= 1000, \\
p &= (s + 1)x \\
&= 10000.0101 \\
&= 10000 && \textit{round or truncate to five bits,} \\
q &= x - p \\
&= 1.1101 - 10000 \\
&= -1110.00110 \\
&= -1110.0 && \textit{round to five bits,} \\
x_{\text{hi}} &= p + q \\
&= 10000 + -1110.0 \\
&= 10.000, \\
x_{\text{lo}} &= x - x_{\text{hi}} \\
&= 1.1101 - 10.000, \\
&= -0.0011,
\end{aligned}$$

$$x = 10.000 - 0.0011.$$

Although binary arithmetic is simple, it is less familiar to humans than decimal arithmetic. The UNIX `bc` calculator makes it easy to do the needed computations with input and output in binary instead of decimal:

```
% bc
ibase = 2
obase = 2
x = 1.1101
s = 1000
(s + 1) * x
10000.01010000000000
x - 10000
-1110.00110000000000
...
```

We then just have to do the rounding manually.

It is imperative that the splitting computation *not* be carried out in higher intermediate precision, as is common on computer systems based on the 68000, AMD64, EM64T, IA-32, and IA-64 CPU architectures, or on systems where the compiler uses fused multiply-adds to compute $ab + c = (s + 1)x + q$ with only a single rounding. In C89 and C99, we can ensure that does not happen by declaring the high part with the `volatile` keyword, and computing it in steps. In older C implementations, and most other programming languages, we need to resort to the `STORE()` routine subterfuge that we have used several times earlier in this book.

13.12 Premature overflow in splitting

The splitting code suffers from premature overflow when $|x|$ is large. It might appear that the solution is to first scale x by a suitable power of the base, such as s^{-2}, compute p, and then rescale that value by s^2. Unfortunately, that does not work. For example, in IEEE 754 arithmetic, when x is near the largest normal number, the scaled algorithm computes a larger x_{hi}, and a negative x_{lo}, producing the pair $(\infty, -\infty)$. Subsequent use of the split in multiplication is almost certain to produce Infinity and NaN. The appearance of those special values in the split of large numbers is unacceptable, so for large values, we use a scaled version of the original algorithm:

$$\begin{aligned}
x_{\text{scaled}} &= x/s^2, \\
x_{\text{hi}} &= ((s+1)x_{\text{scaled}} - s x_{\text{scaled}})s^2, \\
x_{\text{lo}} &= x - x_{\text{hi}}.
\end{aligned}$$

The algorithm produces components of the same sign, and avoids overflow. That split slightly reduces the accuracy of multiplication and division with pair-precision numbers near the overflow limit, but that seems to be unavoidable.

The cost of scaling to avoid premature overflow is two extra multiplies, both exact. The additional expense means that we should do the scaling only when it is essential. We can compare the magnitude of x with an exact compile-time constant that is the largest normal floating-point value divided by s^2: if x is larger, then we may need the scaling.

The revised scaling algorithm looks simple, but extensive testing on multiple platforms exposes a problem that arises from the double rounding that occurs on systems with extended-precision CPU registers. In a *single case*, the largest representable number in the IEEE 754 64-bit format, the split unexpectedly produces a pair $(+\infty, -\infty)$. Here is how that happens:

$$\begin{aligned}
s &= \texttt{+0x1.0p+27}, \\
x &= \texttt{+0x1.ffff\_ffff\_ffff\_fp+1023}, \\
x_{\text{scaled}} &= \texttt{+0x1.ffff\_ffff\_ffff\_fp+969}, \\
(s+1)x_{\text{scaled}} &= \texttt{+0x1.0000\_001f\_ffff\_f7ff\_ffffp+997} && \textit{exact} \\
&= \texttt{+0x1.0000\_001f\_ffff\_f800p+997} && \textit{round to 64} \\
&= \texttt{+0x1.0000\_0020\_0000\_0p+997} && \textit{round to 53},
\end{aligned}$$

$$\begin{aligned}
sx_{\text{scaled}} &= \texttt{+0x1.ffff\_ffff\_ffff\_fp+996}, \\
(s+1)x_{\text{scaled}} - sx_{\text{scaled}} &= \texttt{+0x1.0000\_0040\_0000\_0p+970}, \\
x_{\text{hi}} &= \texttt{+0x1.0000\_0040\_0000\_0p+1024} \qquad\qquad \textit{overflow} \\
&= +\infty, \\
x_{\text{lo}} &= x - x_{\text{hi}} \\
&= -\infty.
\end{aligned}$$

The double rounding from exact to 80-bit format to 64-bit format produced two upward roundings, leading to overflow in x_{hi}. That behavior is reproducible on AMD64, IA-32, IA-64, and Motorola 68000 systems. The error disappears when the compiler generates 64-bit instructions, but that depends on the particular compiler or optimization level, and such instructions are not available at all on IA-32 systems.

The `volatile` type qualifier in the code does not prevent that problem, because both roundings happen before the result leaves the CPU register. Higher intermediate precision is frequently beneficial, but it forces two roundings before data are stored, and that can introduce subtle and expected surprises, as it does in the split.

On systems that do not use higher precision for intermediate results, the computation produces a correct split:

$$\begin{aligned}
(s+1)x_{\text{scaled}} &= \texttt{+0x1.0000\_001f\_ffff\_fp+997}, \\
sx_{\text{scaled}} &= \texttt{+0x1.ffff\_ffff\_ffff\_fp+996}, \\
(s+1)x_{\text{scaled}} - sx_{\text{scaled}} &= \texttt{+0x1.ffff\_ff80\_0000\_0p+969}, \\
x_{\text{hi}} &= \texttt{+0x1.ffff\_ff80\_0000\_0p+1023}, \\
x_{\text{lo}} &= \texttt{+0x1.ffff\_ffc0\_0000\_0p+997}.
\end{aligned}$$

Although the erroneous split happens only for a single value, and only in the 64-bit format, the value for which it occurs is a special one that may well be used in software, as it was in the test programs that revealed the error. One way to mask it is to use precision control (see **Section 5.11** on page 123) to reduce the working precision from 64 bits to 53 bits. Another way would be to alter the rounding direction for that single case. However, those features may not be available on all systems. A satisfactory alternative splitting algorithm is not obvious, so the only choice appears to be architecture-dependent code. That is unpleasant, but it is one of the few instances in the entire mathcw library where such a platform dependence appears.

The code for the split operation is short enough that it is often programmed inline, but usually without consideration of premature overflow, or protection from unwanted higher intermediate precision, or explanation of the meaning and origin of the magic splitting constants. We prefer robustness, and hiding of details, so we provide a function to do the job:

```
void
PSPLIT(fp_pair_t result, fp_t x)
{
    /* split x into (hi,lo) pair in result */

    if ((x == ZERO) || ISINF(x) || ISNAN(x))
        PSET(result, x, x);
    else
    {
        volatile fp_t hi, lo;

        if (((x < ZERO) ? -x : x) > XBIG)
        {
#if B == 2
            volatile fp_t x_scaled;

#if defined(HAVE_FP_T_DOUBLE) &&                         \
    ( defined(FP_ARCH_AMD64) || defined(FP_ARCH_IA32) || \
      defined(FP_ARCH_IA64)  || defined(FP_ARCH_M68K) )
            if (x == FP_T_MAX)
```

```
                hi = ldexp(67108863.0, 998);
            else if (x == -FP_T_MAX)
                hi = -ldexp(67108863.0, 998);
            else
            {
                x_scaled = x * split_square_inverse;
                STORE(&x_scaled);
                hi = one_plus_split * x_scaled;
                STORE(&hi);
                hi -= split * x_scaled;
                STORE(&hi);
                hi *= split_square;
            }
#else
            x_scaled = x * split_square_inverse;
            STORE(&x_scaled);
            hi = one_plus_split * x_scaled;
            STORE(&hi);
            hi -= split * x_scaled;
            STORE(&hi);
            hi *= split_square;
#endif

#elif B == 10

            fp_t s;
            int e;

            s = FREXP(x, &e);
            hi = LDEXP(TRUNC(s * S_SCALE), e - E_SHIFT);
#else
#error "psplit() family not yet implemented for this base"
#endif  /* B == 2 */

            STORE(&hi);
        }
        else            /* |x| <= XBIG */
        {
            volatile fp_t p, q;

            p = x * one_plus_split;
            STORE(&p);
            q = x - p;
            STORE(&q);
            hi = p + q;
            STORE(&hi);
        }

        lo = x - hi;

        if (lo == ZERO)
            lo = COPYSIGN(lo, hi);

        PSET(result, hi, x - hi);
    }
}
```

We must handle signed zero, Infinity, and NaN arguments specially, so as to preserve their sign and values.

Otherwise, direct application of our splitting formulas to a negative zero gives the pair $(+0, -0)$, and to an argument that is either Infinity and NaN, produces the pair $(\mathrm{NaN}, \mathrm{NaN})$. The special handling of a zero low part before the last call to `PSET()` ensures correct behavior later.

The necessary compile-time architecture constants are defined in `pspli.h` as macros prefixed with `MCW` in private `mathcw` header files included by the header file `const.h`, and we assign them at compile time to `const`-qualified meaningful names. Regrettably, the constants cannot be derived from constant expressions involving other constants in the C system header files, nor can the constant expressions contain other `const`-qualified names. C++ permits such constant expressions, but C does not.

The `STORE()` calls can be dispensed with if the `volatile` keyword is properly supported, as is now common in C compilers. The `store.h` header file therefore normally defines the `STORE()` macro to have an empty expansion, but that can be changed by defining the symbol `HAVE_STORE` at compile time, in which case, the `STORE()` macro generates a call to a suitable storage function. At run time, that call forces the value of `hi` out of the CPU and into memory before the call; `hi` is reloaded into the CPU the next time that it is needed. Only two other functions in the elementary operations of the pair-precision family, `PMUL2()` and `PSUM2()`, require the `volatile` keyword and `STORE()` calls.

For the decimal case, arguments near the overflow limit require special handling. No simple algorithm seems to do the job, so we fall back to splitting the argument into a significand and a power of ten, then scale and truncate the significand, and construct the high part. For example, in the case of 32-bit decimal arithmetic, for the largest representable finite number, the normal algorithm produces the incorrect split

$$\begin{aligned} 9.999\,999\mathrm{e}96 &= 10.000\,000\mathrm{e}96 + (-0.000\,001\mathrm{e}96) \\ &= \infty + (-0.000\,001\mathrm{e}96), \end{aligned}$$

whereas the special-case code gives the correct answer:

$$9.999\,999\mathrm{e}96 = 9.99\mathrm{e}96 + 0.009\,999\mathrm{e}96.$$

Timing tests on several systems show that the cost of the `volatile` keyword in `PSPLIT()` is a slowdown by a factor of 1.1 to 4.0, with 1.5 to 2.0 being typical. We pay a penalty for portability on those systems where `volatile` might not be needed. However, on systems with extra-length registers, or fused multiply-add instructions, the fast code is wrong, and thus, useless.

13.13 Pair-precision addition

The addition operation in Dekker's primitives is just the simple form of the Kahan/Møller summation formula given at the start of this chapter:

```
sum = x + y;
err = (x - sum) + y;
```

with the restriction $|x| \geq |y|$.

Dekker [Dek71] cites several necessary conditions on the floating-point system for his pair-precision arithmetic operations to be correct:

- The floating-point base must be 2.
- For the pair-precision addition operation, the host floating-point addition must be "optimal" (rounding to nearest, with at most a half-ulp error), and subtraction must be "faithful", rounding up or down, with at most a one-ulp error. Error bounds are improved if arithmetic is "optimal".

  Implementations of IEEE 754 arithmetic satisfy that in the default rounding mode, although some (IA-32 and IA-64) do not fully conform in those rounding requirements. DEC VAX and IBM System/360 architectures also satisfy the requirements, but some models of now-retired CDC and Cray machines do not.
- The functions require a specific evaluation order, indicated by parentheses in expressions. Parentheses are obeyed in C89, but *not* in earlier implementations of the language.
- The functions assume that `abs(hi)` $\geq$ `abs(lo)`, but place no further constraint on the relative sizes of `hi` and `lo`. In particular, their exponents are not linked by a constant offset.

The restriction to a binary base can be eliminated if one more digit is available for addition and subtraction; however, in practice, that may not be feasible.

The precise conditions for correct operation of pair-precision arithmetic have been analyzed in more detail by Bohlender et al [BWKM91], with later corrections by Boldo and Daumas [BD03a]. In his book section titled *Accuracy of Floating Point Arithmetic*, Knuth treats the problem in great detail [Knu97, §4.2.2, p. 229].

Although we do not do so in the mathcw library, it is worth noting that one can certainly go further than pair-precision arithmetic, without going all the way to a general multiple-precision arithmetic package. Hida, Li, and Bailey extend pair-precision arithmetic to quad-precision, using four numbers of decreasing magnitude, instead of two [HLB00].

Because the mathcw library is designed to work on a wide range of systems, our implementation does not rely on use of parentheses to control evaluation order, but instead, computes the expressions in multiple steps.

Dekker's pair-precision addition algorithm computes the high part of the result as the sum of the input high parts, r, plus a small correction, s. That correction is computed starting with the high part of larger magnitude. The low part of the result is then recovered as a Kahan/Møller error term using the PSUM2() primitive:

```
PADD(fp_pair_t result, const fp_pair_t x, const fp_pair_t y)
{
    /* return the pair-precision sum x + y in result */

    fp_t r, s;

    r = x[0] + y[0];

    if (ISINF(r))
        PSET(result, r, r);
    else
    {
        if (FABS(x[0]) > FABS(y[0]))
        {
            /* s = (((x[0] - r) + y[0]) + y[1]) + x[1]; */
            s = x[0] - r;
            s += y[0];
            s += y[1];
            s += x[1];
        }
        else
        {
            /* s = (((y[0] - r) + x[0]) + x[1]) + y[1]; */
            s = y[0] - r;
            s += x[0];
            s += x[1];
            s += y[1];
        }

        if (s == ZERO)
            s = COPYSIGN(s, r);

        PSUM2(result, r, s);
    }
}
```

The declaration of s does not require a volatile qualifier. Indeed, it is helpful if the compiler can make use of any available higher intermediate precision, because that reduces the error in s. However, the code requires a check for Infinity, because otherwise, the expression x[0] - r produces an unwanted NaN from $\infty - \infty$.

The sign of a zero low part is set to that of the high part to ensure proper handling of negative zero.

In **Section 13.20** on page 379, we measure and discuss the accuracy of the pair-precision primitives.

13.14 Pair-precision subtraction

Dekker's algorithm for pair-precision subtraction is a straightforward modification of the code in PADD(), inverting the sign of the second input argument:

```
void
PSUB(fp_pair_t result, const fp_pair_t x, const fp_pair_t y)
{
    /* return the pair-precision difference x - y in result */

    fp_t r, s;

    r = x[0] - y[0];

    if (ISINF(r))
        PSET(result, r, r);
    else
    {
        if (FABS(x[0]) > FABS(y[0]))
        {
            /* s = (((x[0] - r) - y[0]) - y[1]) + x[1]; */
            s = x[0] - r;
            s -= y[0];
            s -= y[1];
            s += x[1];
        }
        else
        {
            /* s = (((-y[0] - r) + x[0]) + x[1]) - y[1]; */
            s = -y[0] - r;
            s += x[0];
            s += x[1];
            s -= y[1];
        }

        if (s == ZERO)
            s = COPYSIGN(s, r);

        PSUM2(result, r, s);
    }
}
```

Dekker's original primitives lacked our negation primitive. With it, the PSUB() code can be written more simply as

```
void
PSUB(fp_pair_t result, const fp_pair_t x, const fp_pair_t y)
{
    fp_pair_t t;

    PNEG(t, y);
    PADD(result, x, t);
}
```

but the mathcw library uses the first version.

13.15 Pair-precision comparison

The comparison of pair-precision values is best defined as a test for unordered data, in case either value is a NaN, followed by a test of the sign of their difference. The result of the comparison is a small integer that can be tested by the caller. The code is not difficult:

```
int
PCMP(const fp_pair_t x, const fp_pair_t y)
{
    /* Pair-precision comparison: return
        -2 if either x or y is a NaN,
        -1 if x < y,
         0 if x == y, and
        +1 if x > y. */

    int status;

    if ( ISNAN(x[0]) || ISNAN(y[0]) ||
         ISNAN(x[1]) || ISNAN(y[1]) )
        status = -2;
    else
    {
        fp_pair_t result;

        PSUB(result, x, y);             /* |result[0]| >> |result[1]|, or both are zero */

        if (result[0] < ZERO)
            status = -1;
        else if (result[0] == ZERO)
            status = 0;
        else
            status = 1;
    }

    return (status);
}
```

In principle, we could test only the high components for a NaN, but we prefer to be thorough. Infinity arguments pose no problems, because they obey the same ordering relations as finite values.

13.16 Pair-precision multiplication

Dekker defines two primitives for pair-precision multiplication. The first function, PMUL2(), does the easier job of producing a pair-precision representation of the double-length product of two scalars. The second function, PMUL(), handles the more complex task of multiplying two pair-precision numbers.

The product xy of scalars x and y is handled by using PSPLIT() to split each of them into two parts, such that

$$\begin{aligned} x &= x_0 + x_1, \\ y &= y_0 + y_1, \\ xy &= x_0 y_0 + x_0 y_1 + x_1 y_0 + x_1 y_1. \end{aligned}$$

From the discussion of the splitting algorithm in **Section 13.11** on page 359, in a binary floating-point system with reasonable constraints on the accuracy of addition and subtraction, and on the range of the arguments, each of the four terms is *exact*. For other bases, or less accurate arithmetic, the fourth term may be rounded. Dekker's algorithm then computes

$$\begin{aligned} p &= x_0 y_0, \\ q &= x_0 y_1 + x_1 y_0, \\ r &= x_1 y_1, \\ xy &= p + q + r. \end{aligned}$$

using the Kahan/Møller summation formula to form $\texttt{hi} = p + q$ and $\texttt{lo} = (p - \texttt{hi}) + q$. Because the low parts require one less bit than expected, each of the products $x_0 y_1$ and $x_1 y_0$ requires one bit less than is available, and their sum can use that extra bit, so q is exact. The algorithm then adds the fourth term, r, to `lo`, and applies the summation formula once more to obtain the final result.

If Dekker's conditions on the arithmetic system for the split operation are not met, then p may be exact, but q and r each require one or two more digits than are available, and thus, have to be rounded.

When the conditions *are* met, then the sum of the three values $p + q + r$ is exact, but the reduction to a sum of two values in the pair-precision representation introduces additional rounding errors. Thus, in general, the pair-precision result is *not* correctly rounded, but it should be within one or two pair-precision ulps of the exact answer.

The code for `PMUL2()` follows the above analysis reasonably closely, but it replaces the variable r by its value, the low-order product:

```
void
PMUL2(fp_pair_t result, fp_t x, fp_t y)
{
    /* return the nearly double-length product x * y in pair precision in result */

    volatile fp_t hi;
    fp_t lo, p, q;
    fp_pair_t xx, yy;

    PSPLIT(xx, x);
    PSPLIT(yy, y);

    p = xx[0] * yy[0];
    q = xx[0] * yy[1] + xx[1] * yy[0];

    hi = p + q;
    STORE(&hi);

    /* lo = ((p - hi) + q) + xx[1] * yy[1]; */
    lo = p - hi;
    lo += q;
    lo += xx[1] * yy[1];

    if (lo == ZERO)
        lo = COPYSIGN(lo, hi);

    /* Apply extra correction (Dekker's paper, eq. 5.14, pages
       233--234), to reduce error by factor of 2 or 3. */

    PSUM2(result, hi, lo);
}
```

In recent research literature, the `PMUL2()` algorithm is sometimes called `twoproduct()`. That function, and its companions `PSUM2()` (or `twosum()`) and `PSPLIT()`, form the basis of all pair-precision arithmetic.

If a fast *correct* fused multiply-add operation is available, a faster and more compact `PMUL2()` can be written like this:

```
void
```

```
PMUL2(fp_pair_t result, fp_t x, fp_t y)
{
    /* return the nearly double-length product x * y in pair precision in result */
    volatile fp_t xy_neg;

    xy_neg = -x * y;
    STORE(&xy_neg);
    PSET(result, -xy_neg, FMA(x, y, xy_neg));
}
```

We computed the negative of the product so that the FMA() operation appears to have positive arguments. That tricks compilers that recognize `x * y + z` as a fused multiply-add opportunity, but handle `x * y - z` as a separate product and a sum.

We do not provide the fused multiply-add variant of PMUL2() in the mathcw library, because among CPU architectures current at the time of writing this, only IA-64, PA-RISC, POWER, PowerPC, and IBM System/390 G5 have such instructions. Also, POWER provides only the 64-bit version, and PA-RISC, PowerPC, and G5 only the 32-bit and 64-bit ones. On other systems, the fused multiply-add operation is either unavailable, or is implemented in software multiple-precision arithmetic. On the GNU/LINUX operating system, which can run on all of the major CPU types, the library version is wrong, because it just computes `x * y + z` inline; on most systems, that gives a single-length product and suffers two roundings. A correct fused multiply-add operation requires an exact double-length product and a single rounding.

Living with serious library errors, and relying on compiler (mis)behavior, is certainly not a recipe for writing portable code!

The conditions for Dekker's algorithms mean that we can use that function only for systems with rounding arithmetic, and that excludes three of the four IEEE 754 rounding modes, all of which are easily accessible in C99 or with the mathcw library. Fortunately, a decade later, Linnainmaa [Lin81] showed how to extend the pair-precision primitives under less strict conditions, allowing correct operation under all four IEEE 754 rounding modes. In the mathcw library, we use his algorithm for PMUL2(). Its code looks like this:

```
void
PMUL2(fp_pair_t result, fp_t x, fp_t y)
{
    /* return the nearly double-length product x * y in pair precision in result, using
       the algorithm from Seppo Linnainmaa's TOMS 7(3) 272--283 (1981) paper, p. 278 */

    volatile fp_t hi;
    fp_t lo;
    fp_pair_t xx, yy, zz;

    PSPLIT(xx, x);
    PSPLIT(yy, y);
    PSPLIT(zz, yy[1]);

    hi = x * y;
    STORE(&hi);
    lo =  xx[0] * yy[0] - hi;
    lo += xx[0] * yy[1];
    lo += xx[1] * yy[0];
    lo += zz[0] * xx[1];
    lo += zz[1] * xx[1];

    if (lo == ZERO)
        lo = COPYSIGN(lo, hi);

    PSET(result, hi, lo);
}
```

Dekker's algorithm for multiplication of two pair-precision values starts out as before, except that the high and low parts are already available, so PSPLIT() is not required:

$$\begin{aligned}x &= x_0 + x_1,\\ y &= y_0 + y_1,\\ xy &= x_0y_0 + x_0y_1 + x_1y_0 + x_1y_1.\end{aligned}$$

Now, however, *none* of the product terms is exact, because each component is a full-length number. We could therefore apply PMUL2() four times to the products, and then use PSUM2() to track the sum and error. Dekker instead economized the computation, discarding the last term, x_1y_1, on the grounds that dropping it produces a relative error in xy that is $\mathcal{O}(\beta^{-2t})$, or about the same size as the pair-precision ulp. Similarly, the middle terms x_0y_1 and x_1y_0 affect the final product with relative error of $\mathcal{O}(\beta^{-3t})$, so they are computed with ordinary arithmetic. Their sum then contributes a relative error of $\mathcal{O}(\beta^{-3t})$ to the product. The net result is that Dekker accepts six rounding errors in the final product, but they have sharply decreasing weights: two of $\mathcal{O}(\beta^{-2t})$, three of $\mathcal{O}(\beta^{-3t})$, and one of $\mathcal{O}(\beta^{-4t})$. The net effect is that we expect about one or two ulps of error. We discuss experimental measurements of the accuracy of pair-precision arithmetic later in **Section 13.20** on page 379.

The code for PMUL() adds the second and third terms to the error in the first term, drops the fourth term, and then uses PSUM2() to produce a pair-precision result:

```
void
PMUL(fp_pair_t result, const fp_pair_t x, const fp_pair_t y)
{
    /* return the pair-precision product x * y in result */

    fp_pair_t c;
    fp_t tmp;

    PMUL2(c, x[0], y[0]);

    /* c[1] = ((x[0] * y[1]) + (x[1] * y[0])) + c[1]; */

    tmp = x[0] * y[1];
    tmp += x[1] * y[0];
    c[1] += tmp;

    if (c[1] == ZERO)
        c[1] = COPYSIGN(c[1], c[0]);

    PSUM2(result, c[0], c[1]);
}
```

As in PADD() and PSUB(), higher intermediate precision in the computation of c[1] is beneficial, and volatile modifiers are not needed.

13.17 Pair-precision division

The most complex of the pair-precision primitives is division, because it requires iterative computation of successive approximations to the quotient, using a good starting value obtained from x_0/y_0. Convergence is rapid, so only one or two iterations are needed.

However, Dekker proceeded differently. His algorithm makes use of the expansion $1/(a+b) = (1/a)(1 - b/a + (b/a)^2 - \cdots)$, and is derived as follows. First, obtain a pair-precision formula for the full quotient:

$$\begin{aligned}c_0 &= x_0/y_0,\\ x/y &= (x_0/y_0) + (x/y - x_0/y_0)\\ &= c_0 + (x/y - c_0)\end{aligned}$$

$$
\begin{aligned}
&= c_0 + (x - c_0 y)/y \\
&= c_0 + (x_0 + x_1 - c_0 y_0 - c_0 y_1)/y \\
&= c_0 + \big((x_0 + x_1 - c_0 y_0 - c_0 y_1)/y_0\big)\big(1 - y_1/y_0 + (y_1/y_0)^2 - \cdots\big) \\
&= c_0 + c_1.
\end{aligned}
$$

Next, simplify the low-order part by dropping all but the first term in the denominator, because the dropped terms contribute a relative error of $\mathcal{O}(\beta^{-2t})$ to the full quotient:

$$
\begin{aligned}
c_1 &= \big((x_0 + x_1 - c_0 y_0 - c_0 y_1)/y_0\big)\big(1 - y_1/y_0 + (y_1/y_0)^2 - \cdots\big) \\
&\approx (x_0 + x_1 - c_0 y_0 - c_0 y_1)/y_0.
\end{aligned}
$$

Finally, regroup and evaluate the high-order product in pair precision:

$$
\begin{aligned}
c_1 &= (x_0 - c_0 y_0 + x_1 - c_0 y_1)/y_0, \\
u &= c_0 y_0 \\
&= u_0 + u_1 && \textit{compute with } \texttt{PMUL2()}, \\
c_1 &= \big((((x_0 - u_0) - u_1) + x_1) - c_0 y_1\big)/y_0.
\end{aligned}
$$

The code in `PDIV()` is a straightforward translation of the formulas that we derived for c_0 and c_1:

```
void
PDIV(fp_pair_t result, const fp_pair_t x, const fp_pair_t y)
{   /* return the pair-precision quotient of x / y in result */
    fp_pair_t c, u;

    if (y[0] == ZERO)
    {
        if ( (x[0] == ZERO) || ISNAN(x[0]) )
            PSET(result, QNAN(""), QNAN(""));
        else
            PSET(result, INFTY(), INFTY());
    }
    else
    {
        c[0] = x[0] / y[0];

        if (ISINF(c[0]))
            PSET(result, c[0], c[0]);
        else
        {
            PMUL2(u, c[0], y[0]);

            /* c[1] = ((((x[0] - u[0]) - u[1]) + x[1]) - c[0] * y[1]) / y[0]; */

            c[1] = x[0] - u[0];
            c[1] -= u[1];
            c[1] += x[1];
            c[1] -= c[0] * y[1];
            c[1] /= y[0];

            if (c[1] == ZERO)
                c[1] = COPYSIGN(c[1], c[0]);

            PSUM2(result, c[0], c[1]);
        }
    }
}
```

We handle a zero divisor explicitly, producing Infinity for a finite numerator, and otherwise a NaN. After an explicit division, a test for Infinity is essential to avoid generating spurious NaNs from later subtractions of Infinity. No additional NaN tests are needed, because if either x or y is a NaN, the NaN correctly propagates into the result.

As in `PADD()`, `PSUB()`, and `PMUL()`, higher intermediate precision in the computation of `c[1]` is helpful. The `PSUM2()` routine produces a final result that incorporates the sum $c_0 + c_1$ in the high component, and a small error term in the low component.

Because of the dropped terms, and the fact that the sum of terms, the term $c_0 y_1$, and the division by y_0, are computed in ordinary precision, the final value for c_1 suffers about six rounding errors, but as with multiplication, the errors differ sharply in size. At first glance, the total error would appear to be dominated by the error in the initial term, $x_0 - u_0$. However, because $u_0 \approx c_0 y_0 \approx x_0$, the initial term suffers massive subtraction loss, and is $\mathcal{O}(u_1)$. Nevertheless, the *error* in that subtraction is normally zero. Indeed, if both x_0 and u_0 have the same exponent, as is often the case, the floating-point subtraction is *exact*, because it is just the scaled difference of two integers, and integer subtraction is exact. If the exponents differ, there can be a rounding error of $\mathcal{O}(\beta^{-t} x_0)$. That means that Dekker's pair-precision division algorithm is about as accurate as that for multiplication. See **Section 13.20** on page 379 for more on the subject.

13.18 Pair-precision square root

Dekker's algorithm for the square root of a pair-precision value is essentially one step of the Newton–Raphson iteration, with a starting estimate obtained from the ordinary square root of the high component. We discuss the square-root algorithm in much more detail in **Section 8.1** on page 215, but for the purposes of this description, we only need to borrow a bit of material from that section.

Given a starting estimate, y_0, of the solution of $y = \sqrt{x}$ for a given fixed x, we apply Newton–Raphson iteration to find a zero of $F(y) = y^2 - x$. That produces an improved estimate, $y_1 = \frac{1}{2}(y_0 + x/y_0)$. Dekker turned that into an effective algorithm for the pair-precision square root, $c_0 + c_1 = \sqrt{x_0 + x_1}$, as follows:

$$\begin{aligned}
y_0 &= \sqrt{x_0}, \\
y_1 &= \tfrac{1}{2}(y_0 + x/y_0), \\
c_0 &= y_0 && \textit{high part is initial estimate,} \\
c_1 &= y_1 - y_0 && \textit{low part is correction to initial estimate} \\
&= \tfrac{1}{2}(y_0 + x/y_0) - y_0 && \textit{one step of Newton–Raphson iteration} \\
&= \tfrac{1}{2}(x/y_0 - y_0) \\
&= (x - y_0^2)/(2y_0), \\
y_0^2 &= u \\
&= u_0 + u_1 && \textit{compute with } \texttt{PMUL2()}, \\
c_1 &= (x - u)/(2y_0) \\
&= (x_0 - u_0 - u_1 + x_1)/(2y_0) \\
&= (x_0 - u_0 - u_1 + x_1)/(2c_0).
\end{aligned}$$

Notice that in the expansion of c_1 in the middle of that display, we eliminated the term x/y_0 by factoring out the divisor y_0. That is necessary, because Dekker does not define a primitive to compute a pair-precision result for the division of a pair-precision value by a scalar.

In the last two equations for the expansion of c_1, terms have been arranged so that the two high-order terms occur first, and they should be evaluated from left to right, using ordinary floating-point arithmetic.

To conform to IEEE 754 conventions for negative, signed zero, Infinity, and NaN arguments of the square-root function, we have to extend Dekker's algorithm with an enclosing `if` statement that handles those arguments specially. The original code forms the body of the innermost `else` block, and closely follows the algorithm that we derived from the Newton–Raphson iteration:

```
void
PSQRT(fp_pair_t result, const fp_pair_t x)
```

```
{   /* return the pair-precision square root of x in result */

    fp_t x_val;

    x_val = PEVAL(x);

    if (ISNAN(x_val))
        PSET(result, SET_EDOM(x[0]), x[0]);
    else if (x_val == ZERO)             /* preserve sign of zero */
        PSET(result, x[0], x[0]);
    else if (x_val < ZERO)
    {
        fp_t q;

        q = SET_EDOM(QNAN(""));
        PSET(result, q, q);
    }
    else if (ISINF(x[0]))
        PSET(result, SET_ERANGE(x[0]), x[0]);
    else                                /* finite x_val > ZERO */
    {
        fp_pair_t c, u;

        c[0] = SQRT(x[0]);
        PMUL2(u, c[0], c[0]);

        /* c[1] = ((((x[0] - u[0]) - u[1]) + x[1]) * HALF) / c[0]; */

        c[1] = x[0] - u[0];
        c[1] -= u[1];
        c[1] += x[1];
        c[1] = (c[1] / c[0]) * HALF;

        if (c[1] == ZERO)
            c[1] = COPYSIGN(c[1], c[0]);

#if B == 10

        /* A second iteration improves the decimal square root slightly, reducing scattered errors
           from about 0.5 ulp to 0.25 ulp, and the average error from 0.09 ulps to 0.07 ulps. */

        c[0] = PEVAL(c);
        PMUL2(u, c[0], c[0]);

        /* c[1] = ((((x[0] - u[0]) - u[1]) + x[1]) * HALF) / c[0]; */

        c[1] = x[0] - u[0];
        c[1] -= u[1];
        c[1] += x[1];
        c[1] = (c[1] / c[0]) * HALF;

        if (c[1] == ZERO)
            c[1] = COPYSIGN(c[1], c[0]);

#endif /* B == 10 */

        PSUM2(result, c[0], c[1]);
    }
}
```

As with earlier pair-precision primitives, higher intermediate precision in the computation of `c[1]` is desirable. Notice that in the final assignment to `c[1]`, we computed the right-hand side as `(c[1] / c[0]) * HALF` instead of the expected `(c[1] * HALF) / c[0]`. That avoids unnecessary precision loss on systems with wobbling precision, such as the hexadecimal arithmetic on the IBM System/360 mainframe family. Multiplication by a half is only exact when the floating-point base is 2. In a hexadecimal base, that multiplication can cause the loss of three bits, or almost one decimal digit. Although we may suffer that loss anyway, it is better to first compute the quotient `c[1] / c[0]` to full precision.

Because of the self-correcting feature of the Newton–Raphson iteration (see **Section 2.2** on page 8), the errors in the computed square root arise from the four inexact operations in forming c_1, plus a small, and usually negligible, error in computing the low-order part in `PSUM2()`. The net effect should be an error of at most two ulps in rounding arithmetic, and four ulps in truncating arithmetic.

Instead of using Dekker's decomposition for the square root, we can code the Newton–Raphson iteration more simply by using pair-precision arithmetic directly:

```
void
PSQRT(fp_pair_t result, const fp_pair_t x)
{   /* return the pair-precision square root of x in result */
    static const fp_pair_t zero = { FP(0.), FP(0.) };

    if (PISNAN(x))
    {
        (void)SET_EDOM(ZERO);
        PCOPY(result, x);
    }
    else if (PCMP(x, zero) == 0)
        PCOPY(result, x);
    else if (PCMP(x, zero) < 0)
    {
        (void)SET_EDOM(ZERO);
        PSET(result, QNAN(""), QNAN(""));
    }
    else if (PISINF(x))
    {
        (void)SET_ERANGE(ZERO);
        PCOPY(result, x);
    }
    else
    {
        fp_pair_t t, u, y_k;
        int k;

#if B != 2
        static const fp_pair_t half = { FP(0.5), FP(0.) };
#endif

        PSET(y_k, SQRT(PEVAL(x)), ZERO); /* accurate estimate */

        for (k = 0; k < 2; ++k)   /* two Newton--Raphson iterations */
        {                         /* y[k+1] = (y[k] + x / y[k]) / 2 */
            PDIV(t, x, y_k);
            PADD(u, y_k, t);

#if B == 2
            y_k[0] = u[0] * HALF;
            y_k[1] = u[1] * HALF;
```

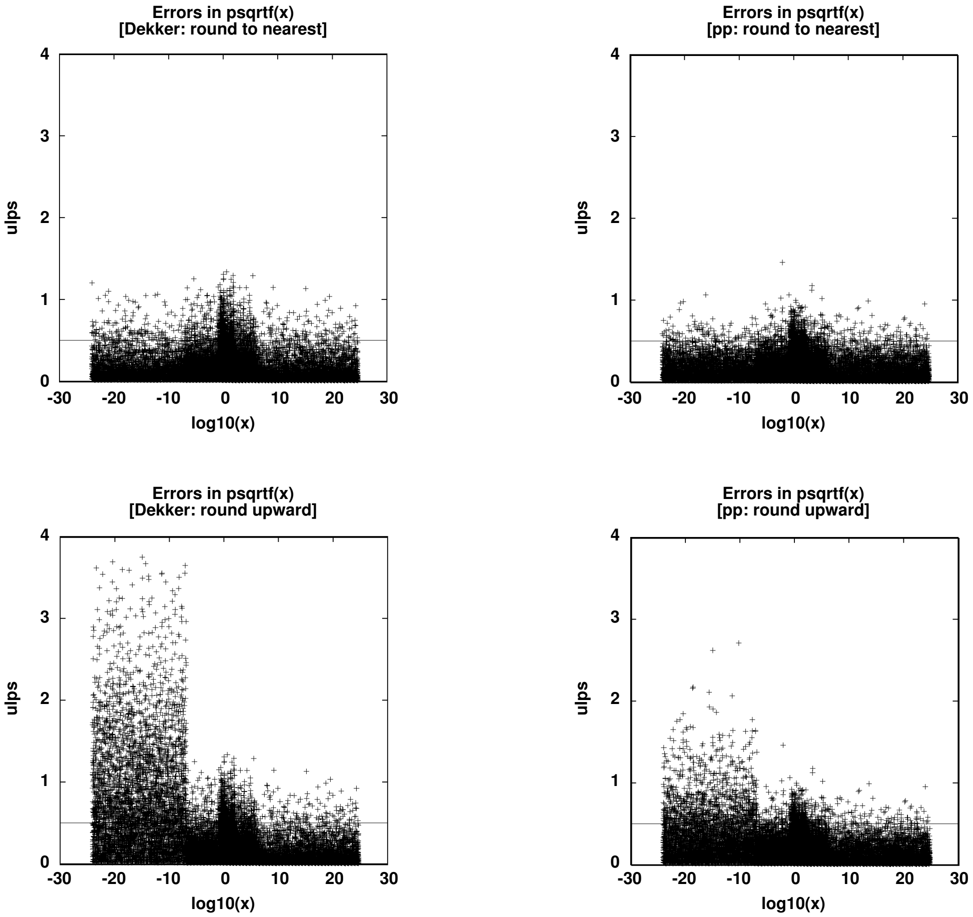
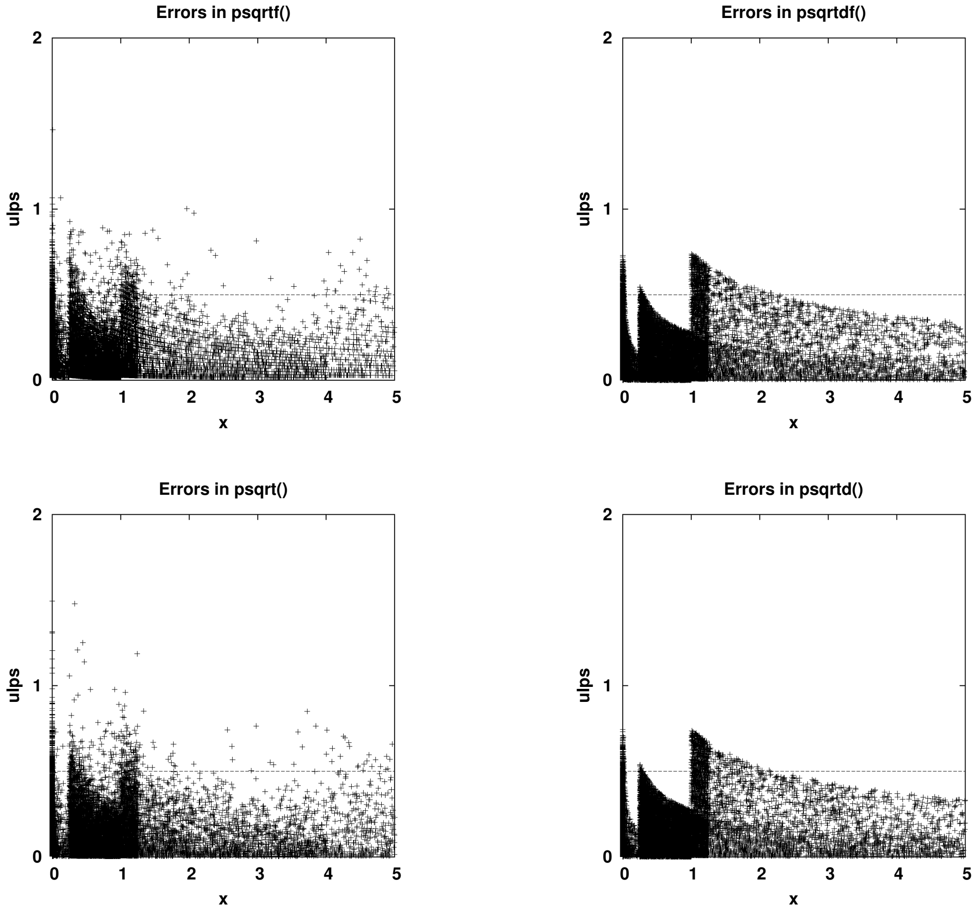


Figure 13.1: Errors in pair-precision square-root functions. The horizontal dotted line at 0.5 ulps marks the boundary below which results are correctly rounded. The plots on the left show the measured errors for Dekker's algorithm for the square root, and those on the right for the algorithm using direct pair-precision arithmetic.

```
#else
            PMUL(y_k, u, half);
#endif

        }

        PCOPY(result, y_k);
    }
}
```

The two approaches are compared on a logarithmic argument scale in **Figure 13.1** using two different rounding modes. Errors for the *round-to-zero* and *round-downward* modes are smaller than for the *upward* rounding mode, and thus, are not shown. The direct pair-precision algorithm is clearly superior, and is the default in the file `psqrtx.h`. However, the code for the original Dekker algorithm is retained in that file, and can be selected by defining the symbol `USE_DEKKER_SQRT` at compile time.

Figure 13.2 on the next page shows errors in four pair-precision square root functions with the default algorithm and a linear argument scale. Our error estimates are pessimistic: the measured errors are usually below one ulp in binary arithmetic, and the function results are often correctly rounded in decimal arithmetic.


Errors in psqrtf()

Errors in psqrtdf()

Errors in psqrt()

Errors in psqrtd()

Figure 13.2: Errors in pair-precision square-root functions in binary (left) and decimal (right) arithmetic with the default (non-Dekker) algorithm.

13.19 Pair-precision cube root

Dekker does not define a primitive for the pair-precision computation of the cube root. Because cube-root functions are provided in C99, we implement an algorithm modeled on Dekker's procedure for the square root. There is much more detail on the computation of cube roots in **Section 8.5** on page 237, but here, we only need the formula for the iteration from that section.

Given an initial estimate, y_0, of the solution of $y = \sqrt[3]{x}$ for a given fixed x, the Newton–Raphson iteration applied to the problem of finding a zero of $F(y) = y^3 - x$ produces an improved estimate $y_1 = (2y_0 + x/y_0^2)/3$. To find a pair-precision cube root, we proceed as follows:

$$\begin{aligned}
y_0 &= \sqrt[3]{x_0}, \\
y_1 &= (2y_0 + x/y_0^2)/3, \\
c_0 &= y_0 && \textit{high part is initial estimate,} \\
c_1 &= y_1 - y_0 && \textit{low part is correction to initial estimate} \\
&= (2y_0 + x/y_0^2)/3 - y_0 && \textit{one step of Newton–Raphson iteration} \\
&= (x/y_0^2 - y_0)/3 \\
&= (x - y_0^3)/(3y_0^2),
\end{aligned}$$

$$\begin{aligned}
y_0^2 &= u \\
&= u_0 + u_1 && \textit{compute with } \texttt{PMUL2()}.
\end{aligned}$$

We need an accurate value for the cube of y_0, but that would seem to require triple-precision computation. We therefore compute the cube in steps, using two reductions with `PMUL2()`:

$$\begin{aligned}
y_0^3 &= y_0 u \\
&= v + w, \\
v &= y_0 u_0 \\
&= v_0 + v_1 && \textit{compute with } \texttt{PMUL2()}, \\
w &= y_0 u_1 \\
&= w_0 + w_1 && \textit{compute with } \texttt{PMUL2()}, \\
c_1 &= (x - v - w)/(3c_0^2) \\
&= (x_0 - v_0 + x_1 - v_1 - w_0 - w_1)/(3c_0^2).
\end{aligned}$$

As with the square-root algorithm, the value of c_1 is to be computed from left to right in the term order shown in the last line of the display. The w terms appear last because they are much smaller than the x and v terms.

We handle NaN, Infinity, and zero arguments first. The final `else` block is a straightforward implementation of the computation of c_0 and c_1:

```
void
PCBRT(fp_pair_t result, const fp_pair_t x)
{
    fp_pair_t c, u, v, w;
    fp_t p, q;

    p = PEVAL(x);

    if (ISNAN(p))
    {
        q = SET_EDOM(QNAN(""));
        PSET(result, q, q);
    }
    else if (ISINF(p))
    {
        q = SET_ERANGE(x[0]);
        PSET(result, q, q);
    }
    else if (p == ZERO)
        PSET(result, x[0], x[0]);
    else
    {
        c[0] = CBRT(x[0]);
        PMUL2(u, c[0], c[0]);
        PMUL2(v, c[0], u[0]);
        PMUL2(w, c[0], u[1]);

        /* c[1] = (((((x[0] - v[0]) + x[1]) - v[1]) - w[0]) - w[1]) / (THREE * c[0] * c[0]); */

        c[1] = x[0] - v[0];
        c[1] += x[1];
        c[1] -= v[1];
        c[1] -= w[0];
        c[1] -= w[1];
        c[1] /= THREE * c[0] * c[0];
        if (c[1] == ZERO)
            c[1] = COPYSIGN(c[1], c[0]);

#if B == 10

        c[0] = PEVAL(c);
        PMUL2(u, c[0], c[0]);
        PMUL2(v, c[0], u[0]);
        PMUL2(w, c[0], u[1]);

        /* c[1] = (((((x[0] - v[0]) + x[1]) - v[1]) - w[0]) - w[1]) / (THREE * c[0] * c[0]); */

        c[1] = x[0] - v[0];
        c[1] += x[1];
        c[1] -= v[1];
        c[1] -= w[0];
        c[1] -= w[1];
        c[1] /= THREE * c[0] * c[0];

        if (c[1] == ZERO)
            c[1] = COPYSIGN(c[1], c[0]);

#endif /* B == 10 */

        PCOPY(result, c);
    }
}
```

Higher intermediate precision in the computation of `c[1]` is welcome.

As with the square root, the self-correcting behavior of Newton–Raphson iteration means that the error in the computed cube root comes entirely from the term c_1, which involves eight inexact operations. With rounding arithmetic, we therefore expect a worst-case error of about four ulps, and with truncating arithmetic, eight ulps.

Instead of using a Dekker-like algorithm for the cube root, we can program the Newton–Raphson iteration directly in pair-precision arithmetic. The code is similar to that for the square root, so we do not display it here. **Figure 13.3** on the next page compares the two approaches in two rounding modes, and shows that the direct algorithm is a better choice. Plots for the other two rounding modes show smaller errors than for *upward* rounding, and are therefore omitted. The alternative algorithm in the file `pcbrtx.h` can be selected by defining the preprocessor symbol `USE_DEKKER_CBRT` at compile time. **Figure 13.4** on page 381 shows the errors in four of our final implementations of cube-root functions in binary and decimal arithmetic.

13.20 Accuracy of pair-precision arithmetic

Dekker derives theoretical estimates of the errors in his pair-precision primitives [Dek71, pages 237–238] for binary floating-point arithmetic. For multiplication, division, and square root, where there is no possibility of massive significance loss in subtraction, the errors are a small constant, not much larger than one, times 2^{-2t}. The pair-precision ulp is 2^{-2t+1}, so the predicted errors are about a half ulp.

As long as leading digits are not lost in subtractions, the predicted errors in addition and subtraction are about four times as large, or about two ulps.

Nevertheless, it is also essential to perform numerical tests of the accuracy, for at least these reasons:

- The theoretical estimates depend on details of floating-point arithmetic that vary between systems, especially for historical architectures.
- Even though modern systems with IEEE 754 arithmetic should behave almost identically for pair-precision arithmetic, some do not obey rounding rules entirely correctly, and the Standard allows rounding behavior

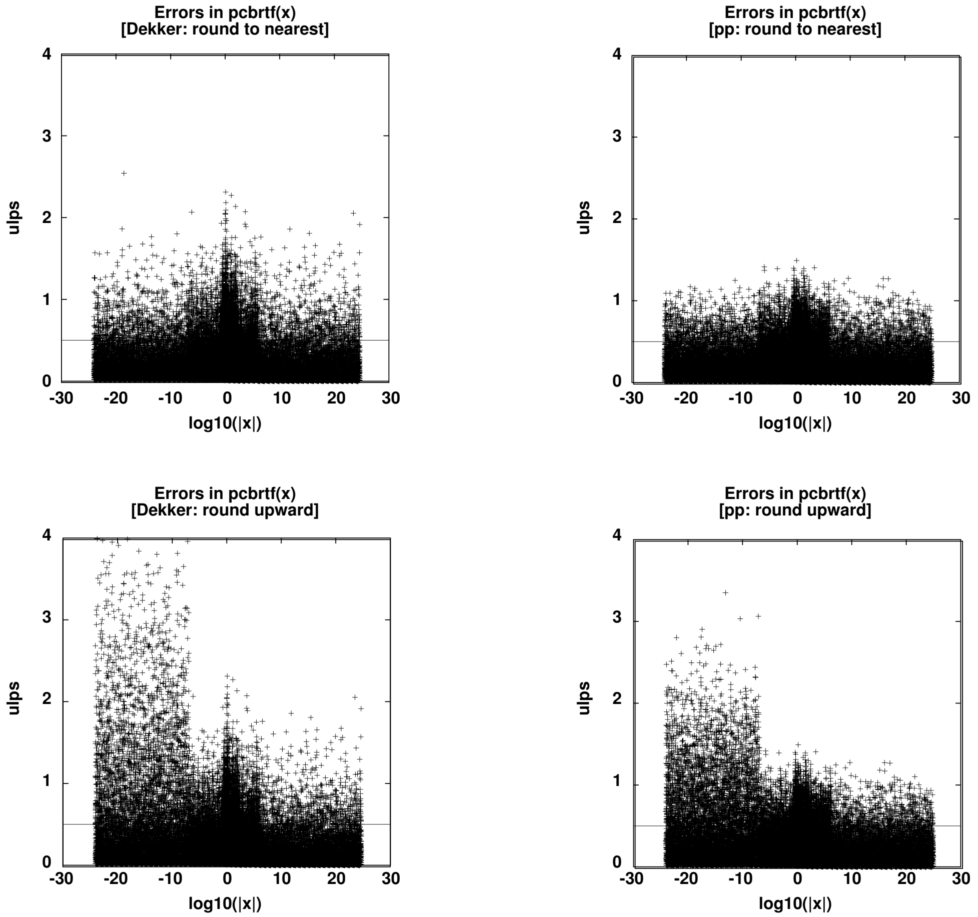


Figure 13.3: Errors in pair-precision cube-root functions. The horizontal dotted line at 0.5 ulps marks the boundary below which results are correctly rounded. The plots on the left show the measured errors for a Dekker-style algorithm for the cube root, and those on the right for the algorithm using direct pair-precision arithmetic.

near underflow and overflow limits to be implementation dependent. See the discussion in Section 4.6 on page 68 for details.

- We have seen several times in this chapter that the transformation from algorithm to code can be tricky, so there could be coding errors.

- Our code is further processed by a compiler, so there could be translation errors, especially from code rearrangement, optimization, and use of higher intermediate precision where it cannot be tolerated.

- For historical systems now implemented with virtual machines, the architects of those virtual machines sometimes have difficulty finding precise specifications of how the floating-point instructions in the often no-longer-running hardware actually worked, or discover that the floating-point hardware behavior varied across different models of computer systems that were shipped to customers.

Severe deviations from the theoretical predictions indicate implementation errors or compiler errors, or perhaps even the effects of floating-point hardware designs that fail to meet Dekker's conditions. Detection and reporting of such deviations must be part of the software validation suite. A significant failure of the test suite strongly suggests that the software should not be installed until the problem is found, repaired, and the validation tests subsequently passed.

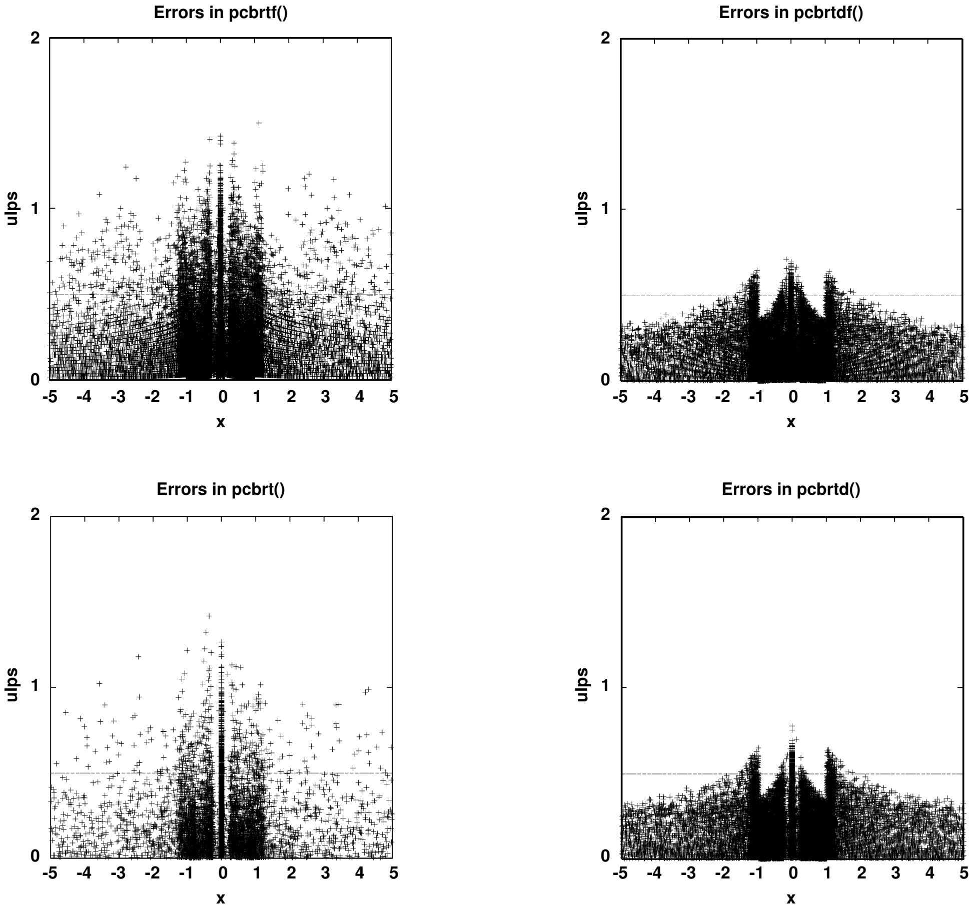


Figure 13.4: Errors in pair-precision cube-root functions in binary (left) and decimal (right) arithmetic with the default (non-Dekker) algorithm.

During development, this author found it helpful to insert assertions at critical points in the code to cause abrupt termination with an error report if the tested condition is found to be false. Those assertions twice caught typos in the code that were quickly fixed. However, they entail additional runtime overhead in time-critical primitives, so they have been removed from the final code on the grounds that their job can be handled at build time by the validation suite. Of course, that does not help in the detection of intermittent hardware errors on a particular system, but such errors are rare.

Table 13.3 on the following page and **Table 13.4** show the results of numerical experiments to measure the accuracy of those pair-precision primitives that involve significant computation. **Table 13.5** and **Table 13.6** show the distribution of errors for the collected primitives.

In **Table 13.4**, the error determination requires independent computation of the function values in IEEE 754 128-bit arithmetic. That is implemented in software on the test systems, and is thus substantially slower than hardware arithmetic. The number of test values was therefore reduced a hundredfold compared to the experiments for the `float_pair` primitives. That in turn means that the search for maximum errors is a hundred times less thorough, and consequently, those errors underestimate what could be found with more computation.

The high parts of the test values are random values from the unit interval, possibly scaled to a few units wide. The low parts are similarly selected, but are smaller by a factor of 2^{-t}, and have random signs. In addition, they are purified so that the sum of the high and low parts is exactly representable. The two parts are determined by code like this:

Table 13.3: Accuracy of `float_pair` primitives. These results were generated on an AMD64 system, and closely reproduced on an IA-64 system, with 100 million random test values for each of the high and low parts, and random signs for the low parts. Random test values were taken from a uniform distribution on $[0, 1)$.
The ulp value is $2^{-47} \approx 7.105 \times 10^{-15}$.
The **Exact** column shows the percentage of results that have zero error.
The σ column is the standard deviation of the arithmetic mean.

Function	**Exact**	**Average error (ulps)**	**Maximum error (ulps)**	**σ (ulps)**
Round to nearest (IEEE 754 default)				
`paddf()`	28.38%	0.124	1.370	0.140
`psubf()`	26.08%	0.162	1.669	0.182
`pmulf()`	3.67%	0.252	2.850	0.234
`pdivf()`	4.22%	0.311	3.904	0.344
`pcbrtf()`	7.60%	0.193	3.396	0.236
`pcbrt2f()`	7.97%	0.155	1.895	0.169
`psqrtf()`	16.33%	0.109	1.491	0.141
Round to $-\infty$				
`paddf()`	17.15%	0.334	2.644	0.326
`psubf()`	18.87%	0.355	2.745	0.346
`pmulf()`	0.60%	0.640	5.229	0.478
`pdivf()`	1.64%	0.775	7.121	0.809
`pcbrtf()`	2.28%	0.443	4.979	0.404
`pcbrt2f()`	1.97%	0.525	4.452	0.467
`psqrtf()`	5.53%	0.343	3.556	0.332
Round to 0				
`paddf()`	17.33%	0.321	2.644	0.324
`psubf()`	18.82%	0.348	2.721	0.348
`pmulf()`	1.40%	0.531	5.229	0.468
`pdivf()`	1.30%	0.911	8.129	0.948
`pcbrtf()`	3.43%	0.343	4.951	0.370
`pcbrt2f()`	1.95%	0.525	4.452	0.467
`psqrtf()`	5.50%	0.336	3.556	0.333
Round to $+\infty$				
`paddf()`	17.93%	0.349	2.655	0.330
`psubf()`	18.77%	0.383	2.749	0.346
`pmulf()`	0.36%	0.667	4.145	0.401
`pdivf()`	2.02%	0.567	5.449	0.571
`pcbrtf()`	2.38%	0.974	9.840	1.027
`pcbrt2f()`	0.51%	1.916	9.509	1.355
`psqrtf()`	3.40%	0.696	5.568	0.626

```
exact_t x10;

u = UNIRAN();
x0 = u * ONE;

u = UNIRAN();
u *= RANSIGN();
x1 = x0 * u * EPS_1 * FP(0.5);

x10 = (exact_t)x1 + (exact_t)x0;
x1 = (fp_t)(x10 - (exact_t)x0);         /* purify */
```

The `UNIRAN()` call returns a uniformly distributed random value on $[0, 1)$. The `RANSIGN()` call returns $+1$ or -1 with equal probability. The final adjustment on `x1` handles the purification, so that `x0 + x1` is exact in the precision `exact_t`, which must have at least twice the precision of the type `fp_t`.

For the subtraction tests, the second operand, $y = y_0 + y_1$, is further constrained by the choice

```
u = UNIRAN();
```

Table 13.4: Accuracy of `double_pair` primitives. The results presented here were found to be identical on Sun Microsystems SPARC and Hewlett–Packard PA-RISC systems. A Hewlett–Packard IA-64 system produced largely similar results.
There were one million random test values for each of the high and low parts, and random signs for the low parts. Random values were taken from a uniform distribution on $[0, 1)$.
The ulp value is $2^{-105} \approx 2.465 \times 10^{-32}$.
The **Exact** column shows the percentage of results that have zero error.
The σ column is the standard deviation of the arithmetic mean.

Function	**Exact**	**Average**	**Maximum**	σ
		error (ulps)	**error (ulps)**	**(ulps)**
Round to nearest (IEEE 754 default)				
`padd()`	39.69%	0.065	0.484	0.078
`psub()`	24.73%	0.162	1.780	0.182
`pmul()`	0.89%	0.249	2.168	0.227
`pdiv()`	1.10%	0.310	3.853	0.343
`pcbrt()`	3.02%	0.174	2.808	0.205
`pcbrt2()`	3.06%	0.153	2.005	0.167
`psqrt()`	4.24%	0.145	1.904	0.190
Round to $-\infty$				
`padd()`	25.64%	0.208	0.989	0.211
`psub()`	18.35%	0.370	2.949	0.346
`pmul()`	0.13%	0.659	4.280	0.470
`pdiv()`	0.41%	0.798	7.926	0.806
`pcbrt()`	0.73%	0.476	3.605	0.403
`pcbrt2()`	0.78%	0.558	4.265	0.474
`psqrt()`	0.96%	0.369	2.333	0.296
Round to 0				
`padd()`	26.08%	0.169	0.988	0.172
`psub()`	18.34%	0.361	2.678	0.347
`pmul()`	0.36%	0.539	4.280	0.462
`pdiv()`	0.32%	0.935	8.174	0.949
`pcbrt()`	1.70%	0.301	3.328	0.312
`pcbrt2()`	0.78%	0.558	4.265	0.474
`psqrt()`	0.99%	0.490	3.653	0.496
Round to $+\infty$				
`padd()`	26.37%	0.209	0.991	0.212
`psub()`	18.36%	0.371	2.916	0.346
`pmul()`	0.08%	0.657	3.540	0.397
`pdiv()`	0.53%	0.557	5.705	0.572
`pcbrt()`	0.83%	0.696	6.667	0.711
`pcbrt2()`	0.04%	1.896	9.490	1.355
`psqrt()`	0.24%	0.599	3.766	0.472

```
  y0 = u * x0 * FP(0.5);
  u = UNIRAN();
  y1 = y0 * u * EPS_1 * FP(0.5);
```

That makes the high components differ by at least a factor of two, preventing all loss of leading digits in the subtraction. Without that constraint, the errors in subtraction can be much larger, and they hide the trailing-digit errors that we want to measure.

The standard deviations from the mean that are given in the σ column of the tables identify the spread of the results around the mean. From introductory statistics, or Table 19.3 on page 616, only about 0.3% of the results lie more than 3σ away from the mean, and fewer than 0.6 in a million are more than 5σ away.

The good news from the data in the two tables is that when the rounding mode is the IEEE 754 default of *round to nearest*, the pair-precision primitives are, on average, *accurate to better than a half ulp*, which is almost the same as correctly rounded to the representable value closest to the exact value.

In the three other rounding directions supported by IEEE 754, the average errors are still below one ulp, and for

Table 13.5: Error distribution for `float_pair` primitives. Results are cumulative for the seven primitives.

Rounding mode	Frequency of errors (ulps)						
	$[0, 1/2)$	$[1/2, 1)$	$[1, 2)$	$[2, 3)$	$[3, 4)$	$[4, 5)$	$[5, \infty)$
to nearest	91.17%	7.72%	1.06%	0.04%			
to $-\infty$	59.89%	27.31%	11.23%	1.27%	0.24%	0.05%	0.01%
to 0	63.61%	23.90%	10.45%	1.53%	0.39%	0.11%	0.02%
to $+\infty$	49.48%	25.85%	14.31%	5.47%	2.92%	1.27%	0.70%

Table 13.6: Error distribution for `double_pair` primitives. Results are cumulative for the seven primitives.

Rounding mode	Frequency of errors (ulps)						
	$[0, 1/2)$	$[1/2, 1)$	$[1, 2)$	$[2, 3)$	$[3, 4)$	$[4, 5)$	$[5, \infty)$
to nearest	91.78%	7.16%	1.02%	0.04%			
to $-\infty$	61.01%	26.54%	10.90%	1.26%	0.24%	0.05%	0.01%
to 0	63.61%	23.90%	10.45%	1.53%	0.39%	0.11%	0.02%
to $+\infty$	50.66%	25.01%	13.88%	5.50%	2.96%	1.28%	0.72%

add and subtract, are below a half ulp, or almost correctly rounded.

In all rounding modes, the maximum errors, however, can reach several ulps, which would not be expected of properly implemented hardware arithmetic, if it were available. The occasional large errors complicate the error analysis of code that uses pair-precision arithmetic.

The `pcbrt2f()` and `pcbrt2()` results are for special variants of the normal cube-root routine that we describe in Section 13.19 on page 377. In the variant routines, the final `else` block contains two Newton–Raphson iteration steps. That roughly doubles the amount of work, and the measurements for the default case of *round to nearest*, show that, although there is little effect on the average error, it reduces the maximum error and standard deviation by a third. However, for the other three IEEE 754 rounding directions, the average error, the maximum error, and the standard deviation, are made worse by the extra computation.

13.21 Pair-precision vector sum

The Kahan/Møller summation error term is easy to incorporate into a loop that sums the elements of a vector. Instead of adding a vector element to a scalar sum in the loop body, add the element and the current error estimate, and then compute a new error estimate. On completion of the loop, the last sum and error term provide a pair-precision result.

Here is code to compute the pair-precision sum of a vector of ordinary floating-point values:

```
void
PSUM(fp_pair_t result, int n, const fp_t x[/* n */])
{
    /* compute x[0] + ... + x[n-1] with Kahan/Møller error
       compensation to produce the pair-precision result */

    int k;
    fp_pair_t sum;

    PSET(sum, ZERO, ZERO);

    for (k = 0; k < n; ++k)
        PSUM2(sum, sum[0], x[k] + sum[1]);

    PCOPY(result, sum);
}
```

The dimension precedes the array in the argument list to conform to the requirement of declaration before use in the new C99 support of run-time dimensioning of arrays, a feature that has been available in Fortran for decades.

The argument array is not referenced if the loop count is zero or negative, and the result array is not referenced until the loop completes, so that it can overlap with the input `x[]` array if that proves convenient.

The loop body contains only the ordinary arithmetic inside `PSUM2()`, not further calls to pair-precision arithmetic routines. Apart from the loop-counter and function-call overhead, each iteration requires three additions and one subtraction, instead of the single addition of a simple vector sum. We therefore expect the loop to be about four times slower, but the improvement in accuracy can be significant.

Although more accurate vector-summation algorithms have been developed since the Kahan/Møller work, they are considerably more complex to program, and to understand. The bibliography of floating-point arithmetic[1] lists about 100 publications on that topic; the state of the art at the time of writing this is represented by the work of Priest [Pri91], Demmel and Hida [DH04], Ogita, Rump, and Oishi [ORO05, ROO08a, ROO08b, Rum09, Rum12], Zhu and Hayes [ZH10], and Kornerup and others [KLLM12, CBGK13, KGD13, BCD$^+$14, AND15, CDGI15, ML15, Nea15, AND16, KGD16, Lef16, OOO16]. Priest's work shows how to extend the idea of pair-precision arithmetic to arrays of n components of decreasing magnitude, using only ordinary floating-point operations for the componentwise arithmetic.

In an amazing feat of lengthy and complicated analysis, Rump, Ogita, and Oishi show how to reduce the mathematics to algorithms in MATLAB, each of which fills less than a page. Their `AccSum()`, `AccSumK()`, `FastAccSum()`, and `NearSum()` functions compute sums that provably return the nearest floating-point number to the exact value, and have the desirable properties that they require only one IEEE 754 floating-point format, need no unpacking of exponent or significand, contain no performance-sapping branches in the inner loops, and compute easy sums quickly, and difficult ones more slowly. Translations of their algorithms to C are straightforward, but they do require dynamic-memory allocation of internal arrays. That feature violates the design requirements of the `mathcw` library, so we do not discuss them further here.

13.22 Exact vector sums

It is worth thinking about how one might implement an exact floating-point summation algorithm by brute force. The problem is most simply illustrated by considering the three-term sum $M + m - M$, where M is the maximum normal number, and m is the minimum subnormal number. The first and last terms exactly cancel, and the result is the tiny middle term. If we have a fixed-point accumulator that is wide enough to hold any of the representable numbers in the floating-point range, we can dispense with floating-point exponents and do the computation in exact fixed-point arithmetic. The only rounding is that incurred when the value in the accumulator is finally converted to storage precision. We require accumulators of width at least $\texttt{EMAX} - \texttt{EMIN} + t$ bits. For the five extended IEEE 754 binary formats, the widths are 277, 2098, 32 829, 32 878, and 524 522 bits, respectively. In practice, several more bits are needed to accommodate sums with large numbers of terms: increasing the width by 50 bits can handle sums with up to 10^{15} terms.

Several proposals to build long accumulators in hardware have been published (see [Knö91, MRR91, MRR96, SvG12, BK15] and their references to earlier work), but no mainstream CPU designs offer such a facility. The challenge is to make the common case of much more limited range run about as fast as straightforward floating-point summation, then to provide a standard interface to the long accumulator in major programming languages, train compiler writers and programmers how to use it, and convince them to do so. Recent software solutions are sketched in journals [Rum09, ZH10], but are complicated to program.

13.23 Pair-precision dot product

The dot product of two n-element vectors is defined by

$$\mathbf{x} \cdot \mathbf{y} = \sum_{k=0}^{n-1} x_k y_k.$$

[1] See `http://www.math.utah.edu/pub/tex/bib/index-table-f.html#fparith`.

Its accurate computation is difficult, especially if some terms are positive, and others are negative, because severe subtraction loss is then possible. As we observed in the previous section, a brute-force approach would require exact products and sums in a wide accumulator, and support for two-element products roughly doubles the accumulator-width requirements, because of the doubled exponent range.

Pair-precision arithmetic provides a reasonable way to compute the dot product, when we realize that splitting x[k] and y[k] into pairs with PSPLIT() leads to a four-term sum for the product in each iteration. As long as we have a base-2 floating-point system, and the values are away from the underflow and overflow limits, each of the four products is *exact* (see **Section 13.11** on page 359). If we represent the accumulating dot product in pair precision, then one application of PSUM() reduces the inner sum to a single pair, ready for the next iteration.

The code for the dot-product operation is then clear and simple:

```
void
PDOT(fp_pair_t result, int n, const fp_t x[], const fp_t y[])
{
    /* compute vector dot product x * y in pair precision
       in result */

    fp_t t[6];
    fp_pair_t xx, yy;
    int k;

    PSET(t, ZERO, ZERO);

    for (k = 0; k < n; ++k)
    {
        PSPLIT(xx, x[k]);
        PSPLIT(yy, y[k]);
        t[2] = xx[0] * yy[0];
        t[3] = xx[0] * yy[1];
        t[4] = xx[1] * yy[0];
        t[5] = xx[1] * yy[1];
        PSUM(t, 6, t);
    }

    PCOPY(result, t);
}
```

We arrange to keep the accumulating dot product in the first two elements of t[], and put the four intermediate products in the next four elements. After the call to PSUM(), the pair sum of the six elements is in the first two elements, ready for the next loop iteration. That is only possible because of our design specification that allows overlap of the input and output arguments of the pair-precision primitives.

13.24 Pair-precision product sum

Although PDOT() makes it possible to evaluate an n-element dot product, there is one important special case that appears in a few places in the mathcw library code: the expression $ab + cd$. If the terms are of opposite sign, then severe subtraction loss is possible. By using pair-precision arithmetic, we can work with almost exact products ab and cd, and the subtraction results in a nearly correct answer, as long as only a few leading digits are lost in the subtraction.

The product-sum routine PPROSUM() makes it convenient to compute an accurate value of $ab + cd$ in pair precision, after which PEVAL() can reduce the result to working precision. The code is straightforward, and free of loops and conditionals:

```
void
PPROSUM(fp_pair_t result, fp_t a, fp_t b, fp_t c, fp_t d)
{
    /* Compute a*b + c*d in pair precision in result */

    fp_t x[4];
    fp_pair_t aa, bb, cc, dd, ab, cd;

    PSPLIT(aa, a);
    PSPLIT(bb, b);
    PSPLIT(cc, c);
    PSPLIT(dd, d);

    x[0] = aa[0] * bb[0];
    x[1] = aa[0] * bb[1];
    x[2] = aa[1] * bb[0];
    x[3] = aa[1] * bb[1];

    PSUM(ab, 4, x);

    x[0] = cc[0] * dd[0];
    x[1] = cc[0] * dd[1];
    x[2] = cc[1] * dd[0];
    x[3] = cc[1] * dd[1];

    PSUM(cd, 4, x);

    x[0] = ab[0];
    x[1] = cd[0];
    x[2] = ab[1];
    x[3] = cd[1];

    PSUM(result, 4, x);
}
```

The call to `PSPLIT()`, followed by collection of each of the products in descending magnitude in the vector `x[]`, produces four *exact* terms (see **Section 13.11** on page 359). A call to `PSUM()` then accumulates an accurate sum of the products. After doing that for both ab and cd, we again have four terms to sum, and a final call to `PSUM()` produces an accurate result for $ab + cd$.

13.25 Pair-precision decimal arithmetic

The base restrictions in some of Dekker's algorithms prevent their use for decimal floating-point arithmetic. Nevertheless, the general ideas of pair-precision arithmetic still apply, as long as we can provide suitable implementations of the basic operations of add, subtract, multiply, and divide. Most of the other pair-precision functions that we discuss in this chapter, and in **Chapter 23** on page 777, are independent of the base, or are already parameterized to work with any reasonable base.

From the parameters of decimal floating-point arithmetic summarized in **Table D.6** on page 937, we see that the precision more than doubles when the length of the data type increases. That means that we can represent products exactly in the next higher precision, and we can do a reasonably accurate job for sums, differences, and quotients. At the highest available precision, we have to do something different, but there, we can fall back to the software implementation of decimal arithmetic provided by the underlying decNumber library [Cow07]. The major deficiency lies in splitting of single-precision decimal values into high and low parts: with seven digits, the low part has four digits, so products of two low parts can no longer be exactly represented. The higher precisions have even numbers of digits, eliminating that problem.

Accuracy tests of the decimal versions of the four basic operations show that the average relative error is below 0.01 ulps, and the maximum relative error does not exceed 0.50 ulps, considerably better than is obtainable with the binary versions. For square root and cube root, the average relative error is below 0.11 ulps, and the maximum relative error is below 4.85 ulps.

13.26 Fused multiply-add with pair precision

The discussion of exact summation in Section 13.22 on page 385 and the C99 requirement that the fused multiply-add operation to compute $xy + z$ be done with an exact product xy and a single rounding from the sum suggests that a software implementation is likely to be difficult.

The pair-precision arithmetic primitives allow the computation to be done as follows:

$$\begin{aligned} x &= x_{\mathrm{hi}} + x_{\mathrm{lo}} && \textit{compute with } \texttt{PSPLIT()}, \\ y &= y_{\mathrm{hi}} + y_{\mathrm{lo}} && \textit{compute with } \texttt{PSPLIT()}, \\ xy + z &= x_{\mathrm{hi}}y_{\mathrm{hi}} + x_{\mathrm{hi}}y_{\mathrm{lo}} + x_{\mathrm{lo}}y_{\mathrm{hi}} + x_{\mathrm{lo}}y_{\mathrm{lo}} + z. \end{aligned}$$

The four componentwise products can each be done exactly in binary floating-point arithmetic, and they each require one less bit than is available. The sum $x_{\mathrm{hi}}y_{\mathrm{lo}} + x_{\mathrm{lo}}y_{\mathrm{hi}}$ involves terms with the same exponent, so that sum is exact. However, it is not obvious that the remaining three sums can be done without introducing multiple rounding errors. What we do know is that, as long as the products are nonzero, the last product is almost negligible compared to the first, so it is unlikely to affect the rounding. The sum of the high-low products is larger, but still small compared to the high-high product, so it too should have little effect on the rounded sum of products. The sign and size of z relative to xy is unknown, and in the worst case, the final sum could have a large effect on the outcome.

We thus expect that computation of the fused multiply-add operation with pair-precision arithmetic should often be accurate, but we certainly have no proof that the computation always satisfies the single-rounding requirement.

Test programs for `float`, `double`, and `long double` versions of a prototype of a pair-precision arithmetic algorithm for the fused multiply-add operation are amazingly successful:

- On current systems with native fused multiply-add hardware (G5, IA-64, PA-RISC, and PowerPC), the tests find *not a single instance* where the error of the pair-precision code is as large as a half ulp, conforming to the C99 requirement.

  The tests are done in all four IEEE 754 rounding modes, and use up to 10^8 arguments from logarithmically distributed random samples over the entire range of floating-point numbers, except for the region of small arguments where the low parts become subnormal.

- Tests on systems with library software implementations of the operation, including Hewlett–Packard HP-UX `cc` on PA-RISC, Intel `icc` on AMD64 and IA-32, and Sun Microsystems SOLARIS `cc` on IA-32 and SPARC, are similarly successful.

- On SOLARIS SPARC, the tests report 'errors' of up to 180 ulps, but that reveals inaccuracies in the vendor library. Recomputation of the worst case in 100-digit precision in Maple shows exact agreement with our `FMA()` code.

Except for some subtleties that we discuss in the remainder of this chapter, we conclude that the following implementation of the fused multiply-add is usually correct, and we therefore use code similar to it in the mathcw library:

```
fp_t
FMA(fp_t x, fp_t y, fp_t z)
{

#if defined(__GNUC__) && defined(HAVE_FP_T_SINGLE) && \
      defined(FP_ARCH_PA_RISC)

    fp_t result;

    __asm__("fmpyfadd,sgl %1, %2, %3, %0" : "=f" (result) :
            "f" (x), "f" (y), "f" (z));

    return (result);

#elif defined(__GNUC__) && defined(HAVE_FP_T_SINGLE) && \
      defined(FP_ARCH_IA64)
    fp_t result;

    __asm__("fma.s %0 = %1, %2, %3" : "=f" (result) :
            "f" (x), "f" (y), "f" (z));

    return (result);

#elif defined(__GNUC__) && defined(HAVE_FP_T_SINGLE) && \
      defined(FP_ARCH_MIPS) && defined(HAVE_CORRECT_MADD_S)
      /* CODE DISABLED! */

    fp_t result;

    /* The madd.s instruction does not exist on the MIPR R4000
       series, and is implemented incorrectly as a separate
       multiply and add on the R10000, sigh...  This code is
       therefore disabled on all MIPS processors, because we
       cannot tell at compile time which one we have, and the
       code would produce CPU-dependent results if we ignored
       processor differences. */

    /* "madd.s fd,fz,fx,fy" produces x * y + z in fd */
    __asm__("madd.s %0, %3, %1, %2" : "=f" (result) :
            "f" (x), "f" (y), "f" (z));

    return (result);

#elif defined(__GNUC__) && defined(HAVE_FP_T_SINGLE) && \
      defined(FP_ARCH_POWERPC)

    fp_t result;

    __asm__("fmadds %0, %1, %2, %3" : "=f" (result) :
            "f" (x), "f" (y), "f" (z));

    return(result);

#elif defined(__GNUC__) && defined(HAVE_FP_T_SINGLE) && \
      defined(FP_ARCH_S390)

    /* MADBR rz,rx,ry overwrites rz with x * y + z */
    __asm__("maebr %1, %2, %3" : "=f" (z) :
            "0" (z), "f" (x) , "f" (y));

    return (z);

#elif defined(__GNUC__) && defined(HAVE_FP_T_SINGLE) && \
      defined(FP_ARCH_SPARC) && defined(__FP_FAST_FMAF__)

    fp_t result;

    __asm__("fmadds %1, %2, %3, %0" : "=f" (result) :
            "f" (x), "f" (y), "f" (z));

    return(result);
```

```
#elif defined(__GNUC__) && defined(HAVE_FP_T_DOUBLE) && \
      defined(FP_ARCH_PA_RISC)

    fp_t result;

    __asm__("fmpyfadd,dbl %1, %2, %3, %0" : "=f" (result) :
            "f" (x), "f" (y), "f" (z));

    return (result);

#elif defined(__GNUC__) && defined(HAVE_FP_T_DOUBLE) && \
      defined(FP_ARCH_IA64)

    fp_t result;

    __asm__("fma.d %0 = %1, %2, %3" : "=f" (result) :
            "f" (x), "f" (y), "f" (z));

    return (result);

#elif defined(__GNUC__) && defined(HAVE_FP_T_QUADRUPLE) && \
      defined(FP_ARCH_IA64)

    fp_t result;

    __asm__("fma %0 = %1, %2, %3" : "=f" (result) :
            "f" (x), "f" (y), "f" (z));

    return (result);

#elif defined(__GNUC__) && defined(HAVE_FP_T_EXTENDED) && \
      defined(FP_ARCH_IA64)

    fp_t result;

    __asm__("fma %0 = %1, %2, %3" : "=f" (result) :
            "f" (x), "f" (y), "f" (z));

    return (result);

#elif defined(__GNUC__) && defined(HAVE_FP_T_DOUBLE) && \
      defined(FP_ARCH_MIPS) && defined(HAVE_CORRECT_MADD_D)
      /* CODE DISABLED! */

    fp_t result;

    __asm__("madd.d %0, %3, %1, %2" : "=f" (result) :
            "f" (x), "f" (y), "f" (z));

    return (result);

#elif defined(__GNUC__) && defined(HAVE_FP_T_DOUBLE) && \
      defined(FP_ARCH_POWER)

    fp_t result;

    __asm__("fmadd %0, %1, %2, %3" : "=f" (result) :
            "f" (x), "f" (y), "f" (z));

    return(result);

#elif defined(__GNUC__) && defined(HAVE_FP_T_DOUBLE) && \
      defined(FP_ARCH_S390)

    /* MADBR rz,rx,ry overwrites rz with x * y + z */
    __asm__("madbr %1, %2, %3" : "=f" (z) :
            "0" (z), "f" (x) , "f" (y));

    return (z);

#elif defined(__GNUC__) && defined(HAVE_FP_T_DOUBLE) && \
      defined(FP_ARCH_SPARC) && defined(__FP_FAST_FMA__)

    fp_t result;

    __asm__("fmaddd %1, %2, %3, %0" : "=f" (result) :
             "f" (x), "f" (y), "f" (z));

    return(result);

#elif defined(HAVE_FP_T_SINGLE)            && \
      (DBL_MANT_DIG >= (2 * FLT_MANT_DIG)) && \
      (DBL_MAX_EXP  >= (2 * FLT_MAX_EXP))

    return ((fp_t)((double)x * (double)y + (double)z));

#elif defined(HAVE_FP_T_DOUBLE) && defined(HAVE_LONG_DOUBLE) && \
      !defined(HAVE_BROKEN_LONG_DOUBLE)                   && \
      (LDBL_MANT_DIG >= (2 * DBL_MANT_DIG))               && \
      (LDBL_MAX_EXP  >= (2 * DBL_MAX_EXP))

    return ((fp_t)((long double)x * (long double)y +
            (long double)z));

#else

    /* code presented later in this section */

#endif

}
```

The code for FMA() in file fmaxxx.h is exceptional in the mathcw library in that it uses inline assembly code to access hardware instructions, where available. That is not only platform dependent, but also compiler dependent, because few compilers, other than GNU gcc, provide an interface to assembly code. However, the performance gain is definitely worthwhile.

The code is intentionally written with only a single level of preprocessor directives and complex conditional expressions, because it is then easier to see the precise conditions that select a block of code for compilation.

Hardware support for the fused multiply-add operation is regrettably incomplete. For each platform, we need code for the float, double, and long double function types, but POWER lacks the float variant, and only IA-64 has the long double instruction.

Although the FMA() code has conditional blocks for the MIPS processors, they are normally disabled by the final test in the selector expressions. The madd.d and madd.s instructions were introduced with the MIPS IV architecture specification in 1994, and a vendor architecture manual describes a correct fused multiply-add operation. However, tests on the R5000 and R10000 CPUs show that the instructions instead perform separate multiply and add operations with two roundings, making them useless for their intended purpose. It is uncertain whether that flaw exists in other CPUs in the MIPS IV family, so it is unsafe to use the fused multiply-add instructions in an environment with mixed CPU models. However, we leave the code in place so that it can be enabled by compile-time definition of the symbols `HAVE_CORRECT_MADD_D` and `HAVE_CORRECT_MADD_S` in more restricted environments where the CPUs are known to implement the specification correctly.

In 2007, the SPARC architecture definition was extended to include 32-bit and 64-bit fused multiply-add instructions. That hardware support is not yet available to this author, but recent releases of the native compilers can generate those instructions when requested by command-line options. When they do so, they define the macros `__FP_FAST_FMAF__` and `__FP_FAST_FMA__`, and we therefore test for their presence before selecting inline code. As that CPU family has evolved, the vendor has provided run-time compatibility by trapping and emulating instructions that are unimplemented on some processor models. Library code built with the fused multiply-add instructions should then work on all SPARC systems that have the required software assist in the operating system.

The `gcc __asm__()` macro is complex, and instruction operand order may disagree with hardware manuals. It is not as well documented as one might wish, and not used often enough to ensure that the documentation matches the compiler support. The macro argument takes the form of three colon-separated fields:

- The first provides a list of one or more assembly-code instructions to be generated, with operands indicated by percent and a digit, and numbered from zero.
- The second field describes the output of the instruction (operand `%0`). In each case, `"=f"` means that a value is produced in a floating-point register associated with the parenthesized variable name following the code string.
- The third string contains a comma-separated list of items describing the input operands (`%1`, `%2`, ...). The code `"f"` means an input operand in a floating-point register, and `"0"` means an input register operand that is overwritten with the result. Each code string is followed by the corresponding variable name in parentheses.

The compiler ensures that, before the instruction begins, all operands have been properly initialized from their associated variables, and that, on completion, the result is stored in the output variable. In practice, the optimizer removes redundant loads and stores. On IA-64, PA-RISC, PowerPC, and SPARC, with optimization selected, the entire body of `FMA()` compiles into just two instructions: the fused multiply-add, and a return, and memory is not referenced at all. On MIPS, when the code is enabled, only the `fma()` function reduces to two instructions.

When higher precision with at least twice as many digits is available, the code uses a single inline multiply and add. That is faster than pair-precision arithmetic, and because the product xy is exact, the correct result is usually, but not always, obtained with only a single rounding. We examine it more carefully in the next section.

We delayed presentation in the code for `FMA()` of the final block that uses pair-precision arithmetic when neither hardware, nor adequate higher precision, are available. Here is what it looks like:

```
fp_t result;

if (ISNAN(x))
    result = x;
else if (ISNAN(y))
    result = y;
else if (ISNAN(z))
    result = z;
else if (ISINF(x) || ISINF(y) || ISINF(z))
    result = x * y + z;
else if (z == ZERO)
{
    volatile fp_t xy;

    xy = x * y;
    STORE(&xy);

    if ( (x != ZERO) && (y != ZERO) && (xy == ZERO) )
```

```
            result = xy;                    /* sign independent of z */
        else                                /* x == 0 && y == 0 */
            result = xy + z;                /* sign depends on z */
    }
    else if ( (x == ZERO) || (y == ZERO) )  /* and z is nonzero */
        result = z;
    else
    {
        fp_pair_t rr, t00, t01, t10, t11, xx, yy, zz;

        PSPLIT(xx, x);
        PSPLIT(yy, y);
        PMUL2(t00, xx[0], yy[0]);
        PMUL2(t01, xx[0], yy[1]);
        PMUL2(t10, xx[1], yy[0]);
        PMUL2(t11, xx[1], yy[1]);
        PSET(zz, z, ZERO);
        PADD(rr, t11, zz);
        PADD(rr, rr, t10);
        PADD(rr, rr, t01);
        PADD(rr, rr, t00);

        result = PEVAL(rr);
    }

    return (result);
```

Because pair-precision arithmetic often produces a NaN when Infinity is present, we supply special handling for NaN, Infinity, and zero arguments.

If any argument is a NaN, we return that NaN.

Otherwise, if any argument is Infinity, we let the hardware compute `x * y + z`: the result may be a signed Infinity, or a NaN.

All arguments are now known to be finite. However, we need special treatment for a zero z because straightforward computation of `x * y + z` can produce a result of the wrong sign. The culprit here is underflow:

- Higher intermediate precision must be prevented by the usual subterfuge of the `volatile` qualifier for the product, or the `STORE()` function.
- If either factor in the product is exactly zero, then the product is an *exact* zero of suitable sign, and the addition of a signed zero produces the correct sign, which depends on those signs, and on the current rounding direction.
- If both factors are nonzero, but their product underflows, then the result must be just `x * y`, because that would be nonzero in exact arithmetic, and the subsequent addition of a signed zero cannot change the sign of the result.

Now z is known to be nonzero, and if x or y is zero, we can avoid further arithmetic entirely.

Otherwise, all three arguments are finite and nonzero, and the last braced block contains the pair-precision arithmetic that does the job in a dozen statements.

Despite the encouraging test results that we reported earlier, two problems remain: the tricky case of rounding results that lie almost exactly between two adjacent representable numbers, and intermediate overflow and underflow in the products because of inadequate exponent range. The next two sections describe how we can handle them.

13.27 Higher intermediate precision and the FMA

When a higher precision type with at least twice as many significand digits, and with sufficiently extended exponent range, is available, the simple statement

```
return ((fp_t)((hp_t)x * (hp_t)y + (hp_t)z));
```

similar to that used in the code of the preceding section is almost always correct. The product is exact, and there is only a single rounding in computing the sum. However, the easy-to-overlook point is that there is a *second rounding* in the outer cast to working precision.

Here is an example in decimal arithmetic that shows what happens in a bad case. We assume 7-digit single-precision operands, but work in 16-digit double precision:

```
hocd64> x = 15
hocd64> y = 3733333
hocd64> z = 1e-8

hocd64> x * y + z
        55_999_995.000_000_01

hocd64> x * y - z
        55_999_994.999_999_99

hocd64> single(x * y + z) ; single(x * y - z)
        56_000_000  # CORRECT: just above halfway case, so rounded up
        55_999_990  # CORRECT: just below halfway case, so rounded down
```

Here, the product lies exactly halfway between two single-precision numbers, and the sum of a small value then decides the rounding direction, producing results that depend on the sign of the addend.

However, if z is a bit smaller, the sum is no longer exactly representable in the higher precision. The sum must then be rounded, and the second rounding to single precision makes one of the results incorrect:

```
hocd64> z = 1e-9
hocd64> single(x * y + z) ; single(x * y - z)
        56_000_000
        56_000_000  # WRONG: should be 55_999_990
```

The `fma()` function in hoc handles even the tiniest possible z correctly:

```
hocd32> x = 15; y = 3733333; z = MINSUBNORMAL
hocd32> z; fma(x, y, z); fma(x, y, -z)
        1e-101
        5.6e+07
        5.599_999e+07
```

Clearly, we need additional code that checks for the halfway case, and takes remedial action to avoid the double rounding that produces the wrong answer. Here is a private function from `fmaxxx.h` that does the fused multiply-add more carefully:

```
static fp_t
hp_fma(fp_t x, fp_t y, fp_t z)
{   /* return x * y + z computed in hp_t, with special handling of halfway rounding */
    fp_t result;
    hp_t xxyy, xxyyzz, zz;

    zz = (hp_t)z;
    xxyy = (hp_t)x * (hp_t)y;

    xxyyzz = xxyy + zz;         /* x * y + z in higher precision */

    if ( (xxyy == xxyyzz) && (zz != HP_ZERO) )
    {   /* check for worst case of x * y = +/-ddd..ddd500...000 */
        hp_t ff, rr;
        int nn;
```

```
            /* rr is difference between exact and rounded product */
            rr = xxyy - (hp_t)((fp_t)xxyy);
            rr = HP_COPYSIGN(rr, xxyy);
            ff = HP_FREXP(rr, &nn); /* significand of |rr| is in [1/B,1) */

            if (QABS(ff) == HP_HALF)/* worst case of trailing 500...000 */
            {
                if (HP_SIGNBIT(xxyy) == HP_SIGNBIT(zz))
                    rr *= HP_HALF;  /* change to +/-250...000 to force xxyy magnitude to round up */
                else
                    rr *= -HP_HALF; /* change to -/+250...000 to force xxyy magnitude to round down */

                /* adjust trailing digits to force correct rounding */
                result = (fp_t)(xxyy + rr);
            }
            else  /* easy case: type cast is safe and correctly rounded */
                result = (fp_t)xxyy;
        }
        else      /* easy case: type cast is safe and correctly rounded */
            result = (fp_t)xxyyzz;

        return (result);
    }
```

If z is nonzero and $\mathrm{fl}(x \times y + z)$ is identical to $\mathrm{fl}(x \times y)$, then z is tiny and rounding must have occurred. The larger block of the `if` statement then checks for the halfway case, and adds or subtracts a reduced remainder to obtain correct rounding. Otherwise, we are not at the halfway case, or the initial computation of $\mathrm{fl}(x \times y + z)$ is exact. In both those cases, the final type cast is correct, and safe.

It is possible for the final result to be finite, yet the product $\mathrm{fl}(x \times y)$ exceeds the exponent range of working precision. The higher precision must therefore offer at least one extra bit in the exponent. The IEEE 754 formats satisfy that requirement, as do a few hardware architectures, such as the floating-point registers of the Motorola 68000 and the Intel IA-64. Many historical architectures, however, have the same exponent range in all precisions, so for them, premature overflow or underflow in our fused multiply-add code invalidate the results, unless we supply intermediate scaling.

13.28 Fused multiply-add without pair precision

The fused multiply-add operation is of such importance that it is worthwhile to revisit its implementation, this time without using our pair-precision arithmetic primitives. A few key design points lead to a workable implementation that we present shortly:

- If a *correct* hardware fused multiply-add instruction is available, use it.
- Otherwise, if a higher-precision type provides at least twice as many significand bits, and at least twice the exponent range, then use the `hp_fma(x, y, z)` function of the preceding section.

  Except for some embedded systems where all floating-point types are treated as equivalent to a single hardware type, that solution is fast, and nearly optimal for the `float` data type, and may be usable for the `double` data type on some platforms.

  For decimal arithmetic in software, as long as the decimal library provides user-settable precision, the single statement can implemented with about a dozen library calls. We show how to do that later in **Section 13.29** on page 402.
- Otherwise, decompose x and y into sums of terms of descending magnitudes, $x = \sum_i x_i$ and $y = \sum_j y_j$, where each of the terms requires no more than half the number of significand bits. Term products $x_i y_j$ can then be computed *exactly* with ordinary floating-point arithmetic, provided that they are representable.

- If we can compute an exact sum of two terms as a rounded high part, plus a tiny low part that is no larger than half a unit in the last place of the high part, then we should be able to sum a small number of terms almost exactly to machine precision.
- The value of $xy + z = (\sum_i \sum_j (x_i y_j)) + z$ can then be computed with a *single* significant rounding in the final addition. For best accuracy, the double sum should accumulate the products in order of increasing magnitudes.
- Intermediate overflow, underflow, and digit-depleting subnormals, must be avoided by suitable scaling.
- Higher intermediate precision must be prevented, and the problem of double rounding on such machines must be recognized and accepted, or eliminated with dynamic precision control.

The first task is to provide a function that splits a number into an exact sum of terms. With the exceptions of two aberrant historical architectures (the CDC 1700 and the Harris systems listed in **Table H.1** on page 948), and some embedded systems, all floating-point architectures provide at least twice as many significand bits in the `double` data type as there are in the `float` type, so a compact representation can use the shorter type for the terms. We therefore introduce a type definition that makes it easy to treat the term representation as a single object that is scaled away from the underflow and overflow limits:

```
typedef struct
{   /* Represent x as B**e * (typeof(x))(sum(k = 0 : n-1) f[k]) */
    int n;              /* number of elements used in f[] */
    int e;              /* exponent of base B */
    float f[10];        /* enough for 256-bit long long double */
} split_t;
```

The splitting function then just needs a loop that repeatedly extracts the higher bits until the remainder is zero:

```
static void
SPLIT(split_t *s, fp_t x)
{   /* x MUST be finite and nonzero (not NaN or Infinity): unchecked */
    fp_t t;

    t = FREXP(x, &s->e);    /* t in [1/B,1) */

    for (s->n = 0; (s->n < (int)elementsof(s->f)) && (t != ZERO); s->n++)
    {
        float hi;

        hi = (float)t;
        t -= (fp_t)hi;
        s->f[s->n] = hi;
    }
}
```

The scaling from x to t is done first, so that the terms are representable in the smaller range (on most systems) of the `float` data type. We make no effort to check for Infinity, NaN, or zero arguments, because they are assumed to be handled separately. The `SPLIT()` function is not robust enough for general use, and is therefore declared `static` to make it invisible outside the source file in which it is defined.

For IEEE 754 arithmetic, three terms suffice for the 64-bit and 80-bit types, five for the 128-bit type, and ten for the 256-bit type, but our loop test allows early exit as soon as no bits are left. The loop is not entered at all for a zero argument, so the sign of zero is lost: zero arguments are therefore excluded.

Because the loop limit is known at compile time for a given floating-point architecture and data type, we could replace the entire function with macros that provide the split with inline code, such as this definition for `double` arithmetic:

```
#if DBL_MANT_DIG <= 3 * FLT_MANT_DIG

#define SPLIT(s,x)                                      \
  do                                                      \
  {                                                       \
      double t;                                           \
      float hi;                                           \
      t = frexp(x, &(s)->e);                              \
      hi = (float)t; t -= (double)hi; (s)->f[0] = hi;    \
      hi = (float)t; t -= (double)hi; (s)->f[1] = hi;    \
      hi = (float)t; t -= (double)hi; (s)->f[2] = hi;    \
      (s)->n = 3;                                         \
  } while (0)

#endif /* DBL_MANT_DIG <= 3 * FLT_MANT_DIG */
```

The one-trip loop is necessary in C to allow the macro to be used as the true branch of an `if–else` statement. Although we do not require that protection here, it is good practice to define statement blocks in macros that way.

The next task is writing a function for computing an exact sum. Here is a variant of our pair-precision sum given by Boldo and Muller [BM05, BM11], enhanced with mandatory protection against higher intermediate precision:

```
static void
exact_add(double result[/* 2 */], double a, double b)
{   /* a + b == result[0] + result[1] == HI + LO EXACTLY */
    /* Property: |LO| <= 0.5 * ulp(HI) */
    volatile double c, d, err, err_a, err_b, sum;

    sum   = a + b;      STORE(&sum);
    d     = sum - a;    STORE(&d);
    c     = sum - d;    STORE(&c);
    err_a = a - c;      STORE(&err_a);
    err_b = b - d;      STORE(&err_b);
    err   = err_a + err_b;
    result[0] = sum;
    result[1] = err;
}
```

The following fused multiply-add function is long, and contains several subtleties that we need to discuss. However, it is devoid of all knowledge of the bit layout of floating-point data. Except for some constants that define floating-point limits, the code is also independent of floating-point data type, precision, and range. Because of its length, we present it in pieces, as a semi-literate program.

We begin by handling the simple case where type casting, higher precision, extended exponent range, and care for the halfway cases, do the job. As usual, we use generic types and uppercase macro wrappers for function names:

```
fp_t
FMA(fp_t x, fp_t y, fp_t z)
{

#if (2 * FP_T_MANT_DIG <= HP_T_MANT_DIG)   && \
    (2 * FP_T_MAX_EXP  <= HP_T_MAX_EXP)    && \
     !defined(HAVE_BROKEN_LONG_FP_T)

    return (hp_fma(x, y, z));

#else
```

Otherwise, we have the hard case that occupies the rest of the function body.

We handle NaN, Infinity, and zero arguments as described earlier in the final conditional block for the `FMA()` function.

All arguments are now known to be finite and nonzero, but may be subnormal. The code must work properly in all rounding modes, and be able to correctly handle the case of intermediate overflow with a finite normal result, such as in the call `FMA(FP(2.0), FP_T_MAX, -FP_T_MAX)`. It must also handle the case of intermediate underflow with a nonzero normal result, such as in the call `FMA(FP(0.5), FP_T_MIN, -1.5 * FP_T_MIN)`.

We begin the final outer `else` block by splitting x and y, and scaling z with the scale factor from the split of the product xy:

```
    else
    {
        split_t sx, sy;
        volatile fp_t z_scaled;

        SPLIT(&sx, x);
        SPLIT(&sy, y);
        z_scaled = LDEXP(z, -(sx.e + sy.e));
```

The new variable `z_scaled` can overflow or underflow. If it overflows, the magnitude of the result may be Infinity, but in a nondefault rounding mode, it may instead be the largest finite floating-point number. We therefore check whether it is outside the range (`-FP_T_MAX`, `FP_T_MAX`), and if so, we compute the result as the sum of z and a tiny number having the sign of xy. That ensures correct rounding in nondefault modes, and is acceptably accurate because the product of the scaled x and y is nonzero and in the small interval $[1/\beta^2, 1)$:

```
        if (QABS(z_scaled) >= FP_T_MAX)
        {       /* |x * y| << |z|: result is sign(x * y)*tiny + z */
            int nz;

            z_scaled = FREXP(z, &nz);
            z_scaled += COPYSIGN(ONE, x) * COPYSIGN(FP_T_MIN, y);
            STORE(&z_scaled);
            result = LDEXP(z_scaled, nz);
        }
```

If `z_scaled` underflows, the result is just xy, rather than $xy + z$, because the latter introduces two rounding errors, instead of the single error permitted in the fused multiply-add operation. The underflowed value may be zero, or a subnormal, or in nondefault rounding modes, the smallest representable normal magnitude with the sign of z:

```
        else if (QABS(z_scaled) <= FP_T_MIN)
            result = x * y;     /* |x * y| >> |z|: allow 1 rounding */
```

Notice that we do not need to call `ISSUBNORMAL()`, because a simple range test does the job.

Most calls to `FMA()` reach the inner `else` block, which handles the case where all three scaled variables are away from the underflow and overflow limits:

```
        else            /* normal case: z_scaled finite and nonzero */
        {
            fp_t err, pair[2], sum, txi;
            int i, j, n;

            err = ZERO;
            sum = ZERO;

            for (i = sx.n - 1; i >= 0; --i)
            {
                txi = sx.f[i];

                for (j = sy.n - 1; j >= 0; --j)
                {
                    EXACT_ADD(pair, txi * sy.f[j], sum);
                    err += pair[1];
                    sum = pair[0];
                }
            }
```

The nested loops run backwards so that the smallest products are computed first. The variable sum is a correctly rounded approximation to the double sum, but the correction term, err, may have rounding errors from the direct addition of pair[1]. Because there are only a few terms in the double sum (9 to 100, depending on the data type), the final correction term may be in error by a few units in the last place. The probability of that error affecting the final result is small, but if desired, we could reduce it substantially by using another EXACT_ADD() call to represent the correction as a pair.

On completion of the loops, the scaled product xy is now accurately represented as sum + err, and it lies in $[1/\beta^2, 1)$. We can now safely add z_scaled with EXACT_ADD(), form sum + err to get correct rounding behavior, and then undo the scaling to get the final result:

```
            EXACT_ADD(pair, sum, z_scaled);
            err += pair[1];
            sum = pair[0];
            result = sum + err; /* only significant rounding */
            n = sx.e + sy.e;
            result = LDEXP(result, n);
```

Although it appears that we have completed the job, there is one final subtlety. If the result is subnormal after scaling, then that action introduced an additional, and unwanted, rounding when low-order bits are lost in normalization. We therefore check for that case, adjust the correction term by the scaling error, and update the final result accordingly:

```
            if (QABS(result) < FP_T_MIN)
            {   /* subnormal result: adjust for scaling error */
                double err_result, sum_plus_err;
                volatile double err_sum_plus_err;

                sum_plus_err = sum + err;
                err_result = sum_plus_err - LDEXP(result, -n);
                err_sum_plus_err = sum - sum_plus_err;
                STORE(&err_sum_plus_err);
                err_sum_plus_err += err;
                result += LDEXP(err_sum_plus_err + err_result, n);
            }
        }
    }
```

The rest of the code just needs to return the result computed in one of the if-statement branches:

```
    return (result);

#endif /* (2 * FP_T_MANT_DIG < HP_T_MANT_DIG) && ... */

}
```

Extensive tests of float, double, and long double versions of that code against the fused multiply-add hardware instructions on IA-64 have been done in all four IEEE 754 rounding modes, with multiple compilers and different optimization levels. The tests include specially selected difficult arguments, arguments drawn from uniform distributions, and arguments taken from logarithmic distributions covering the entire floating-point range. The results show exact agreement in thousands of millions of tests.

Similar tests of the software against the 32-bit and 64-bit hardware instructions on PowerPC with GNU/LINUX and MAC OS X find exact agreement.

Tests against the SOLARIS SPARC native software implementations of the fmaf(), fma(), and fmal() functions show perfect agreement.

With one exception discussed next, tests on all of the modern operating systems and CPU architectures available to this author against versions of the fused multiply-add functions implemented with multiple-precision arithmetic (see **Section 13.29** on page 401) are similarly successful.

Unfortunately, tests on IA-32 systems with various operating systems show instances of one-ulp errors in the default *round-to-nearest* mode, but not in other modes, for arguments and results well away from overflow and underflow limits. Further investigation shows that the culprit is the *double rounding* that plagues architectures with extended internal precision. Here is an example, simplified from one of the test reports, that shows the effect:

```
hoc80> proc show(mode, sum, err) \
hoc80> {
hoc80>     status = fesetround(mode)
hoc80>     println hexfp(sum + err)
hoc80>     println hexfp(double(sum + err))
hoc80>     println ""
hoc80> }

hoc80> sum = 0x1.0000_0000_0000_bp0

hoc80> err = 0x1.fffp-54

hoc80> show(FE_TONEAREST,  sum, err)
+0x1.0000_0000_0000_b8p+0
+0x1.0000_0000_0000_cp+0

hoc80> show(FE_UPWARD,     sum, err)
+0x1.0000_0000_0000_b8p+0
+0x1.0000_0000_0000_cp+0

hoc80> show(FE_DOWNWARD,   sum, err)
+0x1.0000_0000_0000_b7fep+0
+0x1.0000_0000_0000_bp+0

hoc80> show(FE_TOWARDZERO, sum, err)
+0x1.0000_0000_0000_b7fep+0
+0x1.0000_0000_0000_bp+0
```

The result in the default *to-nearest* rounding disagrees in the last digit with an exact computation:

```
hoc128> show(FE_TONEAREST,  sum, err)
+0x1.0000_0000_0000_b7ff_cp+0
+0x1.0000_0000_0000_bp+0
```

Results for that example in the other three rounding directions are identical in 80-bit and 128-bit arithmetic.

The first rounding in the 80-bit format comes in the initial addition, producing a result that is rounded upward, and the second rounding on storage to a 64-bit result introduces another upward rounding that gives a wrong answer.

Without precision control, it is difficult to prevent that unwanted double rounding. Here is how to introduce that control in hoc:

```
hoc80> status = fesetprec(FE_DBLPREC)

hoc80> show(FE_TONEAREST,  sum, err)
+0x1.0000_0000_0000_bp+0
+0x1.0000_0000_0000_bp+0
```

In our C code, we can wrap the final addition like this:

```
#if defined(FP_ARCH_IA32) && defined(HAVE_FP_T_DOUBLE)
            {   /* final, and only significant, rounding */
               int old_prec;

               old_prec = fegetprec();
```

```
                (void)fesetprec(FE_DBLPREC);
                result = sum + err;
                (void)fesetprec(old_prec);
            }
#else
            result = sum + err;
#endif /* defined(FP_ARCH_IA32) && defined(HAVE_FP_T_DOUBLE) */
```

The conditional limits the variant to the 64-bit `fma()` on IA-32. All of the compilers tested on systems with IA-64 and 68000 CPUs generate 64-bit instructions, instead of 80-bit ones, for arithmetic with operands of data type `double`, so no precision control is needed for them. Although the mathcw library provides the necessary, but nonstandard, interface functions, they may be absent from native libraries on some platforms.

13.29 Fused multiply-add with multiple precision

Yet another way to implement a correct fused multiply-add operation is to resort to multiple-precision arithmetic.

Although several such libraries are available, *GNU MP: The GNU Multiple Precision Arithmetic Library* [GSR$^+$04] is particularly noteworthy, because it is highly optimized and well tested, and it sees wide use, including in the kernels of versions 10 (2005) and later of the Maple symbolic-algebra system, and version 5 (2003) and later of Mathematica.

A second library, *MPFR: The Multiple Precision Floating-Point Reliable Library* [MPFR04, FHL$^+$07], builds on the first to provide correct rounding for binary arithmetic.

Those libraries are usable only on modern systems with IEEE 754 arithmetic, but are otherwise highly portable and easy to build from source code. Recent distributions of GNU / LINUX for several CPU architectures include them as native libraries, and `gcc` uses them internally for accurate compile-time conversions.

This author's major extension of hoc was done before the mathcw library development, so hoc uses the MPFR library for its fused multiply-add functions when they are unavailable, or incorrectly implemented, in native libraries. The code is relatively short, and with a support routine, looks like this for one of the functions:

```
#if defined(HAVE_GMP_H) && defined(HAVE_MPFR_H) && \
    defined(HAVE_MPFR_RND_T)

#include <gmp.h>
#include <mpfr.h>

static mpfr_rnd_t
get_mpfr_rounding(void)
{
    mpfr_rnd_t gmp_rounding_mode;

    switch (fegetround())
    {
    case FE_DOWNWARD:
        gmp_rounding_mode = GMP_RNDD;
        break;

    case FE_TOWARDZERO:
        gmp_rounding_mode = GMP_RNDZ;
        break;

    case FE_UPWARD:
        gmp_rounding_mode = GMP_RNDU;
        break;

    case FE_TONEAREST:
    default:
        gmp_rounding_mode = GMP_RNDN;
        break;
    }

    return (gmp_rounding_mode);
}

#if !defined(HAVE_USABLE_FMA)

#define MP (2 * DBL_MANT_DIG)

double
(fma)(double x, double y, double z)
{
    mpfr_rnd_t gmp_rounding_mode;
    static int first = 1;
    static mpfr_t xx, yy, zz, rr;

    if (first)
    {
        mpfr_init2(xx, MP);
        mpfr_init2(yy, MP);
        mpfr_init2(zz, MP);
        mpfr_init2(rr, MP);
        first = 0;
    }

    gmp_rounding_mode = get_mpfr_rounding();

    (void)mpfr_set_d(xx, x, gmp_rounding_mode);
    (void)mpfr_set_d(yy, y, gmp_rounding_mode);
    (void)mpfr_set_d(zz, z, gmp_rounding_mode);

    (void)mpfr_mul(rr, xx, yy, gmp_rounding_mode);
    (void)mpfr_add(rr, rr, zz, gmp_rounding_mode);

    return ((double)mpfr_get_d(rr, gmp_rounding_mode));
}

#endif /* !defined(HAVE_USABLE_FMA) */

#endif /* defined(HAVE_GMP_H) && defined(HAVE_MPFR_H) && ... */
```

On the first call to `fma()`, the `if` block creates and initializes four `static` data structures that are preserved across calls. The call to the private function `get_mpfr_rounding()` returns the GMP rounding mode that matches the current IEEE 754 rounding mode. The three arguments for the fused multiply-add operation are then converted to multiple-precision format. Although the conversion calls include the rounding mode, it has no effect because both formats are binary, and the target format has higher precision. Two further calls handle the multiply and add, and a final call converts the multiple-precision result back to a `double` value. In those three calls, the rounding mode matters.

For decimal arithmetic at the higher precisions, the mathcw library implementation of the fused multiply-add operation uses the IBM decNumber library, with code like this for one of them:

```
#elif defined(HAVE_FP_T_DECIMAL_LONG_DOUBLE)

  fp_t result;
  decContext set;
  decNumber dn_result, dn_x, dn_y, dn_z;
  decimal128 d128_result, d128_x, d128_y, d128_z;
```

```
    decContextDefault(&set, DEC_INIT_DECIMAL128);
    set.digits *= 2;

    host_to_ieee_128(x, &d128_x);
    host_to_ieee_128(y, &d128_y);
    host_to_ieee_128(z, &d128_z);

    (void)decimal128ToNumber(&d128_x, &dn_x);
    (void)decimal128ToNumber(&d128_y, &dn_y);
    (void)decimal128ToNumber(&d128_z, &dn_z);

    (void)decNumberMultiply(&dn_result, &dn_x, &dn_y, &set);
    (void)decNumberAdd(&dn_result, &dn_result, &dn_z, &set);

    set.digits /= 2;
    (void)decimal128FromNumber(&d128_result, &dn_result, &set);

    ieee_to_host_128(d128_result, &result);

    return (result);
```

That code does not handle decimal rounding modes, because they are not yet supported in the early implementations of decimal arithmetic available during the development of the mathcw library.

Newer versions of the decNumber library provide optimized FMA routines that operate directly on the 64-bit and 128-bit decimal data types. Timing tests show that they run about twice as fast as the older code with its multiple format conversions. With the new routines, the last code block simplifies to just two function calls:

```
  #elif defined(HAVE_FP_T_DECIMAL_LONG_DOUBLE) && \
        defined(HAVE_DECNUMBER_FMA)

    decContext set;

    decContextDefault(&set, DEC_INIT_DECIMAL128);
    (void)decQuadFMA(&result, &x, &y, &z, &set);

    return (result);
```

13.30 Fused multiply-add, Boldo/Melquiond style

In 2004, Sylvie Boldo and Guillaume Melquiond reported an investigation of a never-before-implemented rounding mode: *round to odd* [BM04]. They used an automatic proof checker to validate their work, and showed that such a mode could be used to eliminate the nasty problem of double rounding from extended CPU formats to shorter storage formats. That discovery is largely of theoretical interest, because the hardware in the huge installed base of systems that provide 32-bit, 64-bit, and 80-bit formats cannot be changed.

However, those authors later formally proved [BM08, BM11] that the new rounding mode leads to a compact and fast algorithm for a correct fused multiply-add in a binary base, *provided that* the product and sum do not overflow or underflow. Their pseudocode for `fma(x, y, z)` is short, just four lines:

$$\begin{aligned}
(u_{\mathrm{hi}}, u_{\mathrm{lo}}) &\leftarrow \mathrm{ExactMult}(x, y), \\
(t_{\mathrm{hi}}, t_{\mathrm{lo}}) &\leftarrow \mathrm{ExactAdd}(z, u_{\mathrm{hi}}), \\
v &\leftarrow \mathtt{RO}(t_{\mathrm{lo}} + u_{\mathrm{lo}}), && \textit{round to odd,} \\
\mathtt{result} &\leftarrow \mathtt{RN}(t_{\mathrm{hi}} + v), && \textit{round to nearest, with ties to even.}
\end{aligned}$$

Any algorithm that computes an exact product or sum as a pair-precision result can be used in the first two lines.

The unusual rounding operation can be implemented by this pseudocode:

```
         $d \leftarrow$ RD(x + y),          round down (to $-\infty$),
         $u \leftarrow$ RU(x + y),          round up (to $+\infty$),
         $e \leftarrow$ RN(d + u),          round to nearest, with ties to even,
         $f \leftarrow e \times 0.5$,             exact in binary base,
         $g \leftarrow$ RN(u - f),
    result $\leftarrow$ RN(g + d),          completion of RO(x + y).
```

Turning their pseudocode into a practical algorithm in C is straightforward, and implemented in the file `bm-fma.c`, and its companions for other IEEE 754 binary-format precisions. Here, we show only the essential parts.

The first task is to supply the split operation needed for the exact multiplication:

```
#define HI result[0]
#define LO result[1]

static void
safe_split(fp_t result[/* 2 */], fp_t x)
{
    volatile fp_t p, q;

    p = one_plus_split * x;
    STORE(&p);
    q = x - p;
    STORE(&p);
    HI = p + q;
    LO = x - HI;
}


static void
split(fp_t result[/* 2 */], fp_t x)
{
    if (QABS(x) > split_overflow_limit)
    {
        fp_t f, s;
        int n;

        f = FREXP(x, &n);
        safe_split(result, f);
        s = LDEXP(ONE, n);
        HI *= s;
        LO *= s;
    }
    else
        safe_split(result, x);
}
```

The exact-multiplication function looks like this:

```
static void
exact_mul(fp_t result[/* 2 */], fp_t x, fp_t y)
{
    fp_t p, q, r, rx[2], ry[2];

    split(rx, x);
    split(ry, y);
    /* Form exact (when base is 2) x * y as p + q + r */

    p = rx[0] * ry[0];
    q = rx[1] * ry[0] + rx[0] * ry[1];
    r = rx[1] * ry[1];

    HI = p + q;
    LO = p - HI;
    LO += q;
    LO += r;
}
```

One possibility for the exact-add function is given earlier in **Section 13.10** on page 359, and another in **Section 13.28** on page 397, so we do not repeat their code here.

Implementation of *round-to-odd* addition requires C99 access to IEEE 754 rounding-mode control:

```
static fp_t
round_odd_sum(fp_t x, fp_t y)
{
    volatile fp_t d, e, f, g, result, u;

#pragma FENV_ACCESS ON

    (void)fesetround(FE_DOWNWARD);
    d = x + y;                          /* RD(x + y) */
    STORE(&d);

    (void)fesetround(FE_UPWARD);
    u = x + y;                          /* RU(x + y) */
    STORE(&u);

    (void)fesetround(FE_TONEAREST);
    e = d + u;                          /* RN(d + u) */
    STORE(&e);

#pragma FENV_ACCESS OFF

    f = e * FP(0.5);                    /* exact because beta == 2 */
    g = u - f;                          /* RN(u - f) */
    STORE(&g);
    result = g + d;                     /* RN(g + d) */
    STORE(&result);

    return (result);
}
```

On return, the function leaves the rounding mode in the IEEE 754 default of *round-to-nearest*. To eliminate that assumption, just two more lines of code could save and restore the current rounding mode.

A more cautious implementation, also included in `bm-fma.c`, but not shown here, checks the return codes from `fesetround()`, because a failing return code invalidates the algorithm.

The final fused multiply-add function matches the pseudocode closely:

```
fp_t
BM_FMA(fp_t x, fp_t y, fp_t z)
{
    fp_t t[2], u[2], v, result;

    exact_mul(u, x, y);
    exact_add(t, z, u[0]);
    v = round_odd_sum(u[1], t[1]);
    result = t[0] + v;

    return (result);
}
```

Although we implemented and tested the Boldo/Melquiond algorithm, we have not used it in the mathcw library for `fma(x, y, z)` because the algorithm applies only in IEEE 754 arithmetic when $\beta = 2$, *round-to-nearest* is in effect at entry, and C99 access to rounding control is available. The latter constraint is unlikely to be satisfied for software implementations of the 128-bit and 256-bit formats. The algorithm also needs to be extended to scale sums and products away from the overflow and underflow regions. Nevertheless, we introduced it here because it is novel and unexpected, and likely to encourage further research into fast ways to compute the fused multiply-add operation in software.

Timing tests on several CPU platforms show that `BM_FMA()` is about 35% slower than an incorrect version that implements `round_odd_sum(x,y)` as `RN(x,y)`. That difference could be entirely eliminated if floating-point hardware supplied the new rounding mode. Regrettably, the discovery of the utility of *round-to-odd* came too late for its incorporation in the revised IEEE 754-2008 Standard. It would then have made sense also to supply the companion, but currently nonexistent, modes *round-to-even* and *round-to-nearest-ties-to-odd*.

13.31 Error correction in fused multiply-add

Just as it is possible to compute the error in floating-point addition or multiplication, we can recover the error made in rounding an infinite-precision fused multiply-add operation to working precision. However, the code to do so is not obvious, and little known, so it is worth recording here. The algorithm may have first been published in the previously cited work of Boldo and Muller [BM05, BM11]. We enhance it with protection against higher intermediate precision:

```
static fp_t
FMA_AND_ERR(fp_t result[/* 2 */], fp_t x, fp_t y, fp_t z)
{   /* x * y + z == r1 + r2 + r3 == FMA + HI + LO
    ** r1 is FMA value, result[0] = r2, result[1] = r3
    ** FMA == fma(x, y, z)
    ** |HI + LO| <= 0.5 * ulp(FMA)
    ** |LO| <= 0.5 * ulp(|HI|)
    */

    fp_t alpha[2], beta[2], r1, u[2];
    volatile fp_t g;

    r1 = FMA(x, y, z);
    EXACT_MUL(u, x, y);
    EXACT_ADD(alpha, z, u[1]);
    EXACT_ADD(beta, u[0], alpha[0]);
    g = beta[0] - r1;
    STORE(&g);
    g += beta[1];
    STORE(&g);
    EXACT_ADD(result, g, alpha[1]);

    return (r1);
}
```

The code requires this private helper function for recovering the error in multiplication:

```
static void
EXACT_MUL(double result[/* 2 */], double a, double b)
```

```
{   /* a * b == result[0] + result[1] == HI + LO EXACTLY */
    /* Property: |LO| <= 0.5 * ulp(HI) */
    result[0] = a * b;
    result[1] = FMA(a, b, -HI);
}
```

The helper function is equivalent to the mathcw library `FMUL()` family, except that its implementation here assumes that a fused multiply-add function is available.

The mathcw library does not provide public functions that implement the `FMA_AND_ERR()` family.

A hoc translation of our C code is straightforward, if we follow our convention of returning two-part results in global variables. We can exercise it with the example from **Section 13.27** on page 393:

```
hocd32> load("fma_and_err"}
hocd32> x = 15; y = 3733333; z = 1e-9
hocd32> fma_and_err(x, y, z); __HI__; __LO__
         5.6e+07
        -5
         1e-09
```

The output is $56\,000\,000 - 5 + 10^{-9} = 55\,999\,995.000\,000\,001$, in exact agreement with $x \times y + z$. Further tests with smaller z values show that the function correctly handles such values down to the smallest possible, `MINSUBNORMAL`.

13.32 Retrospective on pair-precision arithmetic

Near the beginning of this chapter, in **Section 13.1** on page 354, we noted several limitations of pair-precision arithmetic. Now, having seen the internal code for a score of routines, we can better judge the difficulty of programming with pair-precision arithmetic. Here are several points to consider:

- Dekker's original five primitives (add, subtract, multiply, divide, and square root) are inadequate for practical programming. We found a need to introduce fifteen more routines, and we still have not provided any additional support for I/O of pair-precision data.

- Although the tables and discussion in **Section 13.20** on page 379 show that the routines are *on average* correctly rounded, they are not always so. It would be better to have implementations that can be guaranteed to be always correctly rounded, because that facilitates error analysis and prediction. Achieving correct rounding is difficult, because it requires attention to many subtle details at the bit level [Omo94, Chapter 6], [EL04a, Chapter 8], [MBdD$^+$10, Chapters 8 and 10], [BZ11, Chapter 3], as IEEE 754 hardware does.

- Because the six core operations (Dekker's five, plus cube root) are not "faithful" (having at most a one-ulp error), we cannot use them to define arithmetic of even higher precision, using pairs of pairs, triples of pairs, and so on. The cited works of Hida, Li, and Bailey [HLB00], and of Priest [Pri91], point the way to extensions to higher precision. However, it might be better to use a portable general multiple-precision package, such as *GNU MP: The GNU Multiple Precision Arithmetic Library* [GSR$^+$04], or its extension, *MPFR: The Multiple Precision Floating-Point Reliable Library* [MPFR04, FHL$^+$07], because that has the significant advantage of offering dynamic choice of precision, and a much wider exponent range.

- For values within $1/\beta^2$ of the overflow limit, we had to change the splitting algorithm to avoid production of Infinity and NaN, and thereby lost the property that allows the product of two numbers to be represented by an exact sum of four exact products of pairs.

- For magnitudes below β^t times the smallest normal number, the split produces a low component that is subnormal or zero. In the worst case, that *halves* the expected precision.

- The exponent range of pair-precision arithmetic is the same as that of the base type, and that range is often grossly inadequate for the higher precision. The exponent sizes allocated for the four IEEE 754 formats grow by about four bits for each doubling of precision, suggesting that the exponent range should increase by at least a factor of 16. Current packages for decimal arithmetic and multiple-precision arithmetic have even wider exponent ranges, because in practice, the exponent range can be as much a barrier to increasing accuracy as the precision is.

- Only two elementary functions, square root and cube root, are provided in this chapter. There are several additional elementary functions for pair-precision arithmetic in the mathcw library, but we delay their presentation until algorithms for their computation have been described later; see **Chapter 23** on page 777.

- Every nontrivial numerical constant must be represented in pair precision in such a way that the high part is exactly representable. That makes each such constant dependent on the host floating-point system, possibly limiting portability. For example, we might require code like this to record the value of π for pair precision on a system with 64-bit IEEE 754 arithmetic:

```
double_pair PI;
pset(PI, 28296951008113760.0 /  9007199254740992.0,
          9935515200043022.0 / (9007199254740992.0 * 9007199254740992.0));
```

  The peculiar constant that occurs repeatedly in the denominators is 2^{53}. Expressions in rational form with exactly representable numerators and denominators, where the denominators are powers of the base, are essential to guarantee exact conversion.

  To compute the numerators and denominators, we need access to arithmetic with higher than pair precision. For example, in Maple, we can calculate them like this:

```
% maple
> Digits := 40;

> 2**53;
        9007199254740992

> pi_high := round((Pi/4) * 2**53) / ((1/4) * 2**53 ):

> pi_low := round((Pi - pi_high) * 2**106) / 2**106:

> pi_high * 2**53;
        28296951008113760

> pi_low * 2**106;
        9935515200043022
```

  Notice that we had to use the value of $\pi/4$ in the computation of the high part to ensure that we get an exactly representable numerator. The file `split.map` in the mathcw package `maple` subdirectory contains a Maple function to simplify the task: the call `split(53,Pi)` reports high and low components in rational integer form.

  A similar calculation with the 128-bit version of hoc looks like this:

```
% hoc128
hoc128> pi_high = round((PI/4) * 2**53) / ((1/4) * 2**53)

hoc128> pi_low = round((PI - pi_high) * 2**106) / 2**106

hoc128> 2**53
        9007199254740992

hoc128> pi_high * 2**53
        28296951008113760

hoc128> pi_low * 2**106
        9935515200043022
```

  With a bit more difficulty, we can coax the answers out of the standard UNIX multiple-precision calculator, `bc`, using the relation $\operatorname{atan}(1) = \pi/4$, and checking the relative error of the pair sum:

```
% bc -l
scale = 40

p = 4 * a(1)

p * 2^53
28296951008113761.1030637736600980854978913107503539552256

h = 28296951008113760 / 2^53

l = p - h

l * 2^106
9935515200043021.7570353844131295999164978322961210540032

l = 9935515200043022 / 2^106

(h + l - p) / p
.0000000000000000000000000000000009532649
```

Variables and built-in functions in bc are limited to a single letter; $a(x)$ is the arc tangent of x. The calculator lacks rounding functions, so we have to do that job manually.

If a C99 compiler is available, we can express the constants in hexadecimal floating point, and check the relative accuracy of their sum:

```
hoc128> hexfp(pi_high)
        +0x1.921fb54442d18p+1

hoc128> hexfp(pi_low)
        +0x1.1a62633145c07p-53

hoc128> (+0x1.921fb54442d18p+1 + +0x1.1a62633145c07p-53 - PI) / PI
        9.80868066233368547287978295469524658e-34
```

The pair-precision initialization code is then shorter, but probably equally puzzling to a human reader:

```
pset(PI, +0x1.921fb54442d18p+1, +0x1.1a62633145c07p-53);
```

If there are lots of constants to express in pair precision, splitting them into high and low parts is tedious and error prone. If the constants are not sensibly named, the reader cannot easily tell from their rational representation in pair precision what they are supposed to be, and whether they are correct or not.

The alternative to that messy specification of pair-precision floating-point constants is accurate run-time conversion from lengthy hexadecimal and decimal representations. That is what most existing packages for higher-precision arithmetic provide, and our pair-precision primitives should clearly be augmented to offer something similar. Programming such a conversion routine is a nontrivial task, and made even more difficult by the irregular rounding-error behavior of pair-precision arithmetic. We show a partial solution to that problem in **Section 23.4** on page 788.

- Without operator overloading, coding with pair-precision arithmetic routines, or any other multiple-precision package, is mind numbing.
- Most seriously, as the algorithms for at least divide, square root, and cube root show, correct handling of the programming of arithmetic with high and low parts is difficult. Subtle issues of component ordering, and sometimes-unwanted higher intermediate precision, make programming mistakes likely. Testing in multiple environments, including systems that support higher intermediate precision, and systems that do not, and using a variety of compilers and optimization options, are essential.

The last point suggests that large-scale programming with pair-precision arithmetic routines is unwise, unless the details of the arithmetic can be largely abstracted and hidden away in suitable primitives. The vector sum and vector dot-product routines, PSUM() and PDOT(), are good examples of how clear and easily understandable code can be written using pair-precision arithmetic primitives.

A major challenge in software design is finding the right set of library primitives. The fused multiply-add function, fma(), was introduced in C99 almost a decade after it became available in hardware on the IBM POWER architecture. That operation has since been shown to be of great value in diverse numerical software; see Markstein's book [Mar00] for many examples. The pair-precision routines PSUM2(), PMUL2(), PSPLIT(), PSUM(), and PDOT(), are in that category as well, especially if the internals of the latter two can be replaced by code that does the job exactly.

Elsewhere in the mathcw library code, we need pair-precision arithmetic in only a few places, and the required code is clean and simple enough that we can have confidence in its correctness.

If you need to extend existing software in C or C++ to use multiple-precision arithmetic, one promising approach is automated conversion to operator-overloaded C++ [SC08]; such conversions have been successfully applied to two large mathematical software libraries.

14 Power function

TO UNDERCREST YOUR GOOD ADDITION
TO THE FAIRNESS OF MY POWER.

— SHAKESPEARE'S *Coriolanus* (1608).

The 1999 ISO C Standard describes the behavior of the power function like this:

> ***7.12.7.4 The pow functions***
>
> ***Synopsis***
>
> ```
> #include <math.h>
> double pow       (double x, double y);
> float powf       (float x, float y);
> long double powl (long double x, long double y);
> ```
>
> ***Description***
>
> *The pow functions compute x raised to the power y. A domain error occurs if x is finite and negative and y is finite and not an integer value. A domain error may occur if x is zero and y is less than or equal to zero. A range error may occur.*
>
> ***Returns***
>
> *The pow functions return* x^y.

ISO Standards for other programming languages, such as Fortran and Pascal, provide even less information about the behavior of that function.

Fortran and a few other programming languages use an operator notation for the power function: `x**y`. Some symbolic-algebra systems, and the TEX typesetting system, use the caret operator instead: `x^y`. However, double asterisk and caret are assigned other meanings in the C-language family, so those languages have always used a function call for the power function. Programmers need not be concerned about the relative efficiency of inline operators versus calls to external functions: hardware does not implement the power function as a single instruction.

A few programming languages provide a restricted version of the power function that only permits powers of positive or negative whole numbers. That is unfortunate, because the power function is so common in numerical computation that users of those languages are then forced to implement their own versions, and as we shall see, their results are likely to be poor.

14.1 Why the power function is hard to compute

The power function has a simple mathematical definition:

$$x^y = \exp(y \ln(x)).$$

The exponential and logarithm functions are available in standard mathematical software libraries, so one might erroneously assume that the power function should be a simple one-line routine. As often happens, apparent mathematical simplicity obscures substantial computational difficulty. Cody and Waite [CW80, Chapter 7] devote the longest chapter of their book to the power function, and the code that results from their algorithm is by far the most complex of the routines whose implementations they describe. In the mathcw library, the code in the algorithm files for the power function, `powx.h` and `pxyx.h`, is about six times longer than that for most other functions, with about a dozen instances of differing code for decimal and nondecimal floating-point systems.

The Cody/Waite algorithm for the power function contains more computational subtleties than any other in their book, and unfortunately, their description of it sometimes leaves out critical details. Worse, the careful initial implementation of their algorithm in the mathcw library revealed several errors in their presentation (see **Appendix E**),

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_14

and uncovered unnecessary numerical deficiencies. The treatment in this book provides the additional detail that this important function deserves.

Although the ELEFUNT tests for most of the other elementary functions that are discussed in **Chapter 24** on page 811 show worst-case losses of one or two bits, the `long double` version of the power function loses up to eight bits when computed by the original Cody/Waite algorithm. That is unacceptably high, and we show later how to do much better.

Why is the computation of the power function so hard? The answer lies in the error-magnification factor that we discussed in **Section 4.1** on page 61:

$$\begin{aligned} d(x^y)/dx &= yx^{y-1}, && \textit{derivative,} \\ \delta(x^y)/(x^y) &= y(\delta x/x), && \textit{error magnification.} \end{aligned}$$

That says that the relative error in the computed power function value is the relative error in x magnified by y.

If we use the mathematical definition to compute the power function, we find

$$\begin{aligned} z &= \exp(y\ln(x)), \\ w &= y\ln(x), \\ dz/dw &= \exp(w), && \textit{derivative,} \\ \delta w &= \delta z/\exp(w) \\ &= \delta z/z, && \textit{error magnification.} \end{aligned}$$

The last result means that the *relative error* in the computed power function value is approximately the *absolute error* in the product of y and $\ln(x)$. Thus, when either x or $|y|$ is large, the relative error in the power function is large.

The *only way* to decrease the relative error in the power function is either to keep w small, or to increase its precision. The former is impossible, because we require w over many orders of magnitude between the underflow and overflow limits of $\exp(w)$.

To show the severity of the problem, from **Table 4.2** on page 65 and **Table D.1** on page 929, we observe that in the `long double` versions of the power function in IEEE 754 arithmetic, the range of w is about $[-11\,433, +11\,356]$, and in the companion type for decimal arithmetic, the range is about $[-14\,145, +14\,150]$. For a 256-bit decimal type, w ranges over approximately $[-3\,621\,810, +3\,621\,656]$. Thus, direct computation of the power function via the exponential and logarithm functions can lose up to five decimal digits in the 128-bit formats, and seven in the 256-bit formats.

We conclude that higher-precision computation of the power function is *essential* if we are to obtain a low relative error. Cody and Waite show that most of that need can be met without access to a data type of higher precision by careful computation of critical steps with variables consisting of sums of exact high- and approximate low-order parts. Nevertheless, there are limits to what can be achieved that way, as the ELEFUNT test results show.

Because we are interested in accuracy as well as reliability and portability of the mathcw library code, we deviate from the Cody/Waite algorithm when it is necessary to do so to improve accuracy. In particular, our power function is one of the few in the library where a higher-precision data type is used internally for small parts of the computation.

Markstein's algorithm for the power function [Mar00, Chapter 12] uses the formula with the exponential and logarithm directly, but exploits features of the IA-64 architecture to compute both of those functions in higher precision in software, with just enough extra bits to obtain results that are almost always correctly rounded. That approach is not easily employed in portable code.

14.2 Special cases for the power function

The mathematical definition of the power function does not apply when x is zero or negative, because in real arithmetic, the logarithm is undefined for those cases. However, it is perfectly reasonable to raise a negative value to an integer power, and the ISO C Standards require support for that case. Historical Fortran Standards and Fortran implementations consulted during this work are silent on the issue of powers of negative x.

In IEEE 754 arithmetic, we know that special handling is almost always required for Infinity, NaN, and signed zero arguments, and sometimes, also for subnormal arguments. However, for the power function, there are additional cases that it is desirable to treat separately, either for speed, or for accuracy, or to ensure important mathematical identities, or even to conform to the sometimes peculiar requirements of C99. We summarize more than two

Table 14.1: Special cases in the power function, in the order in which they must be tested for. Nested simple `if` statements reduce the number of outermost tests compared to code using complex Boolean expressions implied by the conditions at the right.

It should be noted that most existing implementations of the power function in various programming languages handle only a subset of those special cases, with the result that the functions lose accuracy unnecessarily, or produce computational or mathematical anomalies.

Input	Output	Condition
x^y	NaN	either x or y is a NaN (exceptions below)
$(-\infty)^y$	-0	$y < 0$ and y is an odd whole number
$(+\infty)^y$	$+0$	$y < 0$
$(\pm\infty)^y$	$+1$	$y = \pm 0$
$(-\infty)^y$	$-\infty$	$y > 0$ and y is an odd whole number
$(-\infty)^y$	$+\infty$	$y > 0$
$(+\infty)^y$	$+\infty$	$y > 0$
$x^{+\infty}$	$+1$	$\lvert x\rvert = 1$
$x^{+\infty}$	$+\infty$	$\lvert x\rvert > 1$
$x^{+\infty}$	$+0$	$\lvert x\rvert < 1$
$x^{-\infty}$	$+1$	$\lvert x\rvert = 1$
$x^{-\infty}$	$+0$	$\lvert x\rvert > 1$
$x^{-\infty}$	$+\infty$	$\lvert x\rvert < 1$
$x^{\pm 0}$	$+1$	all x, *including* ± 0 and NaN
$(+1)^y$	$+1$	all y, *including* NaN
$(-1)^y$	-1	y is an odd whole number
$(-1)^y$	$+1$	y is an even whole number
$(-1)^y$	NaN	y is not a whole number
$(-0)^y$	-0	$y > 0$ and y is an odd whole number
$(\pm 0)^y$	$+0$	$y > 0$
$(-0)^y$	$-\infty$	$y < 0$ and y is an odd whole number
$(\pm 0)^y$	$+\infty$	$y < 0$
$x^{1/2}$	`sqrt`(x)	all finite or infinite $x > 0$
$x^{-1/2}$	`rsqrt`(x)	all finite or infinite $x > 0$
$(\pm\beta)^y$	exact	β is base and y is a whole number
x^y	$x \times x \times \cdots \times x$	$y > 0$ and y is a limited whole number
x^y	$1/(x \times x \times \cdots \times x)$	$y < 0$ and y is a limited whole number
$(-\lvert x\rvert)^y$	$-(\lvert x\rvert^y)$	y is an odd whole number
$(-\lvert x\rvert)^y$	$+(\lvert x\rvert^y)$	y is an even whole number
$(-\lvert x\rvert)^y$	NaN	y is not a whole number

dozen of them in **Table 14.1** and **Table 14.2** on the next page, and also note a detailed study of the special cases that was published after this book was completed [SF16].

Whenever the result is a NaN, the code must also set a domain error. If the result is $\pm\infty$ (or the signed largest representable magnitude when infinities are not supported) and the arguments are finite, the code must also set a range error. However, a range error is not set if the result is $\pm\infty$ and either argument is itself an infinity. If the result underflows, the code must set the underflow and inexact flags, which is most easily done by returning the *run-time* square of a tiny number, such as the smallest nonzero normal floating-point number.

The handling of 0^0 is the most hotly debated special case, and although it is mathematically undefined, and thus should produce a NaN, reasonable arguments have been published for other return values. However, all of the many C implementations tested by this author return $0^0 = 1$, so our code does as well. The C89 Standard does not discuss that case, but an appendix in the C99 Standard says that `pow(x,0)` must return 1 for any x, including a NaN. C99 also requires a return value of $+1$ for 1^{NaN}. Those two requirements are indefensible in this author's view, because NaNs should propagate in computations, and the code would be simpler, and easier for humans to understand, if it could return a NaN if either argument is a NaN. Nevertheless, our code follows C99, but notice that the similar case of $(-1)^{\mathrm{NaN}}$ produces a NaN.

Special handling for $x = \pm 0$, $x = \pm 1$, and $x = \pm\beta$ is important because the results have a simple form that can be computed quickly, either by direct assignment, or by calling a function in the `ldexp()` family. Novice programmers are likely to call the power function for common factors like $(-1)^n$, 2^n, and 10^n. Most implementations of the

Table 14.2: More on special cases in the power function. Shaded entries indicate abnormalities required by C99. Except for those entries, the return value for a NaN argument is the second argument if it is a NaN, and otherwise, is the first argument.

$x \setminus y$	**−0**	**+0**	**−1**	**+1**	**−∞**	**+∞**	**−QNaN**	**+QNaN**	**−SNaN**	**+SNaN**
−0	+1	+1	−∞	−0	+∞	+0	−QNaN	+QNaN	−SNaN	+SNaN
+0	+1	+1	+∞	+0	+∞	+0	−QNaN	+QNaN	−SNaN	+SNaN
−1	+1	+1	−1	−1	+1	+1	−QNaN	+QNaN	−SNaN	+SNaN
+1	+1	+1	+1	+1	+1	+1	**+1**	**+1**	**+1**	**+1**
−∞	+1	+1	−0	−∞	+0	+∞	−QNaN	+QNaN	−SNaN	+SNaN
+∞	+1	+1	+0	+∞	+0	+∞	−QNaN	+QNaN	−SNaN	+SNaN
−QNaN	**+1**	**+1**	−QNaN	−QNaN	−QNaN	−QNaN	−QNaN	+QNaN	−SNaN	+SNaN
+QNaN	**+1**	**+1**	+QNaN	+QNaN	+QNaN	+QNaN	−QNaN	+QNaN	−SNaN	+SNaN
−SNaN	**+1**	**+1**	−SNaN	−SNaN	−SNaN	−SNaN	−QNaN	+QNaN	−SNaN	+SNaN
+SNaN	**+1**	**+1**	+SNaN	+SNaN	+SNaN	+SNaN	−QNaN	+QNaN	−SNaN	+SNaN

power function do not provide special handling for those cases, with the unexpected result that exactly representable powers can be incorrect by a few ulps. For reasons discussed later in Section 14.6 on page 423, the original Cody/ Waite algorithm has that defect when $x = 1$ or $x = \beta$.

The cases of $y = \pm\frac{1}{2}$ are diverted to `sqrt(x)` and `rsqrt(x)`, respectively, because their implementations in the `mathcw` library guarantee always, or almost-always, correctly rounded results, and are faster than the general power-function algorithm.

We treat integer powers specially, as described in the next two sections.

The additional special-case handling that we provide compared to other implementations of the power function adds a bit of overhead on each call, but it is modest compared to the work involved in the rest of the computation, and its significant benefits outweigh the small cost of extra argument checks.

14.3 Integer powers

Powers that are small integers are worth special treatment: they are common, there are fast ways to compute them, and we can produce results that are almost always correctly rounded. Base conversion for input and output needs exact powers and the functions described in this section provide a solution that does not require large tables of precomputed powers.

For maximum accuracy, we handle negative powers by first computing the positive power, and then taking the reciprocal of the result. That introduces only a single extra rounding error, but does mean that $x^{|n|}$ can overflow or underflow prematurely even though the reciprocal $x^{-|n|}$ is representable. In IEEE 754 arithmetic, reciprocals of numbers near the overflow limit are subnormal, and thus lose precision, or might even be flushed to zero. Reciprocals of values near the underflow limit are normal, and pose no problems. The ranges of some older floating-point designs are less symmetrical; see in particular Appendix H.1 on page 951.

We can avoid the issues of premature underflow and overflow by computing a negative power as $(1/x)^{|n|}$, introducing an error in $1/x$ that is magnified by $|n|$ in the final result. That error can be made negligible by computing the reciprocal and its powers in higher precision. In IEEE 754 arithmetic, only premature underflow is of concern in the power computation, and we can handle its rare occurrence by checking for a zero result, and then redoing the computation with the reciprocal.

The obvious way to compute x^n as a product of n values can easily be improved on by repeated squaring. For example, to find x^{16}, compute x^2 as $x \times x$, square that to get x^4, square again to produce x^8, and square once more to arrive at the final answer with just four multiplications instead of 15. In general, x^n can be computed that way with just $\mathrm{ceil}(\log_2 n)$ multiplications.

In order to program that algorithm, we need to be able to decompose n into a sum of powers of two. For example, $25 = 2^4 + 2^3 + 2^0 = 16 + 8 + 1$, so $x^{25} = x^{16} \times x^8 \times x$ needs six multiplications if we store the intermediate powers. The decomposition into powers of two is easy when we recall that the binary representation of n supplies the answer.

For our example, $25_{10} = 11\,001_2$, and the 1-bits select the required power of two. We only need to store two values if we loop over the bits of n from right to left: one value records the developing result, and the other holds x^{2^k} at bit position $k = 0, 1, 2, \ldots$, counting from the right.

Although we could do the bit extraction by shifting a one-bit mask left on each iteration, AND'ing it with n, and testing for a nonzero result, doing so requires a loop count equal to the number of bits in a word. A better approach is to abandon the mask, and instead, test the rightmost (low-order) bit of n, shift n one bit to the right, and exit the loop as soon as that shift produces a zero.

Here is the integer-power implementation in hoc code:

```
func ipow(x, n)       \
{   # return x**n, for integer n
    v = 1
    k = fabs(int(n))
    xpow = x

    while (k > 0)     \
    {
        if (k & 1) v *= xpow

        k >>= 1

        if (k > 0) xpow *= xpow
    }

    if (n < 0)        \
    {
        v = 1 / v

        if (v == 0) v = ipow(1 / x, -n)
    }

    return (v)
}
```

The total number of multiplications is at least $\mathrm{ceil}(\log_2 n)$, and at most $2\,\mathrm{ceil}(\log_2 n) - 1$. The minimal count happens when n is a power of two, and thus, has only a single nonzero bit. The maximal count is reached for n of the form $2^m - 1$, which has m nonzero bits.

The check for a positive k value before the final squaring in the loop adds to the cost, because it is executed on every iteration, and saves at most one multiplication for the entire loop. However, it prevents premature setting of the *overflow* or *underflow* exception flags in IEEE 754 arithmetic.

Zero arguments are handled by defining $0^0 \equiv 1$, which is what our function produces. There is no consensus among numerical programmers for what should be done for that special case. Our choice is simple to explain, needs no extra code to check for zero arguments, and conforms to C99 requirements.

Negative values of x need no special treatment, because the power operation for integer exponents involves only multiplications, not logarithms.

If underflow or overflow happens, the function result is subnormal, zero, or Infinity, as expected.

In the rare case where $1/v$ underflows to zero for a negative power, we recursively recompute the power as $(1/x)^{|n|}$, sacrificing some accuracy to possibly obtain a tiny nonzero result.

When the power is an integer type in two's-complement arithmetic, negation of the most negative integer produces that number, so a bit more code is needed to check for that case in the handling of negative powers. When x is near 1, such large powers may well be finite.

No checks for Infinity or NaN arguments are needed, because any such arguments simply propagate to the returned function value, as expected. However, our handling of zero powers means that we also define $\pm\infty^0 \equiv 1$, and $\pm\mathrm{NaN}^0 \equiv 1$. That decision could be controversial, but it is easy for users to remember the design choice that $x^0 \equiv 1$ for *all* possible floating-point values of x. That practice agrees with C99, but conflicts with the general rule that NaNs should always propagate.

The binary-decomposition-and-squaring algorithm is reasonably easy to program, and keeps the multiplication count low. However, it is not always optimal, and extensive mathematical analysis of integer-power computation [Knu97, Section 4.6.3] show that it is sometimes possible to do better. The first case where that is so is x^{15}. The binary decomposition computes that power as $x^8 \times x^4 \times x^2 \times x$ with six multiplications, but the result can be obtained with only *five* multiplications from $x^3 = x \times x \times x$, $x^6 = (x^3)^2$, and $x^{15} = x^6 \times x^6 \times x^3$. The cited reference gives a minimal-power table that shows how any power up to $n = 100$ can be computed with at most nine multiplications.

For higher speed and inline loop-free computation, we can use a `switch` statement to quickly select particular cases from the minimal power table, with code that looks like this:

```
switch ((n < 0) ? -n : n)
{

/* cases 0 to 14 omitted */

case 15:
    xxx = x * x * x;
    r = xxx * xxx;
    r *= r * xxx;
    break;

/* cases 16 to 100, and default, omitted */

}
```

Examination of the assembly-language code generated by optimizing compilers on several architectures shows that the `switch` statement uses an indirect branch through a table of addresses to reach the code for the selected case, and the intermediate variables are stored in registers. Each `case` code block consists exclusively of multiply instructions, followed by a single branch instruction to break out of the `switch` statement.

In order to guarantee correctly rounded integer powers, we need to solve the problem of accumulation of rounding errors. As long as x^n is exactly representable in working precision, the computation is exact, but that is only possible when both x and n are small. For example, in IEEE 754 64-bit arithmetic, the powers 2^{53}, 3^{33}, 4^{26}, 5^{22}, …, 29^{10}, …, 100^7, …, 1000^5, …, $10\,000^3$, … are exactly representable.

Extensive numerical experiments with 10^8 random arguments in all four rounding modes of IEEE 754 arithmetic show that at most five multiplications can be done in working precision before the computed result sometimes differs from the exactly rounded power. When a floating-point type of higher precision is available, we use variables of that type for the product accumulation. Otherwise, a preprocessor conditional disables the inline cases that require more than five multiplications, reducing the number of cases from 101 to 22.

For the application of the integer-power function to base conversion, traditional code uses lookup in a table of correctly rounded powers of ten when the exponent range is small, as was the case in most historical architectures. However, the wider exponent range of the binary IEEE 754 80-bit and 128-bit formats would need nearly 10 000 table entries, and the future 256-bit format, more than 630 000. One solution is to store a small table of powers of 5^{2^k} or 10^{2^k}. Just 13 table entries suffice for the 128-bit format, and only 18 for the 256-bit format, and then 10^n can then be computed from products of table entries with a procedure similar to the code in the sample `ipow()` function. Unfortunately, numerical tests show that a few powers of ten with values near the underflow and overflow limits have errors larger than $\frac{1}{2}$ ulp, and thus fail to be correctly rounded. That happens even in a version of the code where the table entries are split into pairs of exact high and accurate low parts.

The table approach is of no use for computing integer powers of general x, so the general solution that we adopt is to use the code from the `ipow()` prototype, with either a higher-precision floating-point type, or else inline pair-precision arithmetic, for powers that are not handled by the `switch` statement.

Here is what the pair-precision implementation of the integer power function looks like:

```
static fp_t
unchecked_ipow (fp_t x, int n)
{
    fp_t result;
    int k;
    fp_t the_power_hi, the_power_lo, xx_hi, xx_lo;
    /* Compute the powers with inline pair-precision arithmetic */

    k = (n < 0) ? -n : n;
    the_power_hi = ONE;
    the_power_lo = ZERO;
    xx_hi = x;
    xx_lo = ZERO;

    while (k > 0)
    {
        if (k & 1)    /* compute the_power *= xx in pair precision */
            PPMUL(the_power_hi, the_power_lo, the_power_hi, the_power_lo, xx_hi, xx_lo);

        k >>= 1;

        if (k > 0)    /* compute xx *= xx in pair precision */
            PPMUL(xx_hi, xx_lo, xx_hi, xx_lo, xx_hi, xx_lo);
    }

    if (ISNAN(the_power_hi) || ISNAN(the_power_lo))
    {
        if (n < 0)
            result = (IS_ODD(n) && (x < ZERO)) ? COPYSIGN(ZERO, -ONE) : ZERO;
        else
            result = (IS_ODD(n) && (x < ZERO)) ? SET_ERANGE(-INFTY()) : SET_ERANGE( INFTY());
    }
    else if (n < 0)
    {
        fp_t one_hi, one_lo;

        one_hi = ONE;
        one_lo = ZERO;
        PPDIV(the_power_hi, the_power_lo, one_hi, one_lo, the_power_hi, the_power_lo);

        if (ISNAN(the_power_hi) || ISNAN(the_power_lo))
        {
            if (QABS(x) >= ONE)
                result = (IS_ODD(n) && (x < ZERO)) ? COPYSIGN(ZERO, -ONE) : ZERO;
            else
                result = (IS_ODD(n) && (x < ZERO)) ? SET_ERANGE(-INFTY()) : SET_ERANGE( INFTY());
        }
        else
        {
            result = the_power_hi + the_power_lo;

            if (result == ZERO) /* possible premature underflow */
                result = unchecked_ipow(ONE / x, -n);
        }
    }
    else
        result = the_power_hi + the_power_lo;

    return (result);
}
```

That function is available only to its caller, a library-internal function _IPOW() that includes the switch statement for fast processing of small powers. That function in turn is called by the user-level function IPOW(), which also handles

Infinity, NaN, and signed zero arguments, as well as the special case where the power is the most negative integer, which cannot be negated in two's-complement arithmetic. The intermediate function `_IPOW()` can be used elsewhere in the library when it is already known that the first argument is finite.

The complexity of pair-precision arithmetic is concealed in the `PPDIV()` and `PPMUL()` macros, but their definitions are omitted here. Both invoke the macro `FFMUL()` to compute the double-length product of its last two arguments. That macro has two reasonable definitions. One uses a fused multiply-add to recover the low part of the product:

```
#define FFMUL(hi, lo, a, b)           \
        {                             \
            hi = a * b;               \
            lo = FMA(a, b, -hi);      \
        }
```

The second uses a mathcw library primitive described later in this chapter (see **Section 14.10** on page 430):

```
#define FFMUL(hi, lo, a, b)           \
        {                             \
            fp_t ab[2];               \
            FMUL(ab, a, b);           \
            hi = ab[0];               \
            lo = ab[1];               \
        }
```

As we observe in the retrospective on pair-precision arithmetic (see **Section 13.32** on page 407), there is accuracy loss if any of the digits in the low component of a pair descends into the subnormal region. That means that, when pair-precision arithmetic is used instead of a higher-precision native data type, powers with values near the underflow limit are computed less accurately than we would like. We can prevent that accuracy loss by instead computing an exactly scaled power away from the underflow region, and then rescale that result. When $|x| > 1$ and $n < 0$, we can write

$$\begin{aligned}
q &= \lfloor 2t/n \rfloor, && \textit{$q < 0$ and t is significand precision,} \\
y &= x\beta^{q}, && \textit{no overflow possible, because $|y| < |x|$,} \\
&= \operatorname{scalbn}(x, q), && \textit{exact scaling,} \\
y^{n} &= x^{n}\beta^{nq}, && \textit{compute with \texttt{\_IPOW(y,n)},} \\
x^{n} &= y^{n}\beta^{-nq} \\
&= \operatorname{scalbn}(y^{n}, -nq).
\end{aligned}$$

We can also approach the underflow limit when $|x| < 1$ and $n > 0$. A little thought shows that a similar scaling works, except that we now must define q by

$$q = \lceil 2t/n \rceil, \qquad q > 0.$$

There is one final piece that we have not yet presented: reduction of powers of integers. Consider the power 10^{22} that might be required for base conversion. Its exact value takes 73 bits to represent, which is longer than the 53-bit significand size of type `double` in IEEE 754 arithmetic. However, 5^{22} can be exactly represented in only 52 bits, and because $10^{22} = 5^{22} \times 2^{22}$, we can compute the larger power *exactly* as `LDEXP(_IPOW(5, 22), 22)` without needing higher precision. In general, to compute m^n, if we can represent m exactly as $k\beta^r$, then `_IPOW(m,n)` can be replaced by `SCALBN(_IPOW(k,n), n * r)`. When that is possible, we effectively gain $n \times r$ extra bits of precision in the power computation, making a correctly rounded result more likely.

The needed decomposition is provided by this function, which removes powers of the base from the first argument, returning the power count via the final pointer argument, and the reduced m as the function value:

```
static long int
factor (long int m, int base, int *power)
{   /* factor m = f * base**(*power), and return f and *power */
    int k;
    long int f;
    f = (m < 0) ? -m : m;

    for (k = 0; (k < (int)(CHAR_BIT*sizeof(long int))) && (f >= base); ++k)
    {
        long int q;

        q = f / base;

        if (q * base != f)
            break;

        f = q;
    }

    *power = k;

    return ((m < 0) ? -f : f);
}
```

Profiling a test of _IPOW() with a large number of random integer arguments shows that the inner loop in factor() executes roughly twice on average, and the time spent there is negligible compared to the rest of the code.

The private factor() function is used in _IPOW() in the switch statement for the default case that is chosen when none of the inline minimal-power computations is selected:

```
default:
    r_set = 0;

    if ( (x == TRUNC(x)) && ((fp_t)LONG_MIN < x) && (x <= (fp_t)LONG_MAX) )
    {
        fp_t f;
        int k;

        k = 0;                          /* keep optimizers happy */
        f = (fp_t)factor((long int)x, B, &k);

        if (k == 0)
            result = unchecked_ipow(x, n);
        else
        {
            if (IS_MUL_SAFE(n, k))
                result = SCALBN(unchecked_ipow(f, n), n * k);
            else
                result = unchecked_ipow(x, n);
        }
    }
    else
        result = unchecked_ipow(x, n);
```

The determination of whether x is a whole number is based on the outcome of the equality test x == TRUNC(x), rather than a test like x == (int)x. As we discuss in **Section 4.10** on page 72, conversion of floating-point numbers to integer values is unsafe and nonportable without range tests.

Our treatment of the integer power function shows that the major cause of inaccuracy in the function result is the accumulated rounding error from the multiplications needed to compute the power. We showed how to minimize the operation count, and how, with suitable attention to scaling near the underflow limit, we can use pair-precision arithmetic to make the rounding errors negligible.

By providing special handling of integer powers from within the general power function, we can guarantee fast execution and high accuracy, and we extend the C99 library by making the ipow() family available to the user. We also avoid the anomalies seen in some Fortran implementations, where the computed results for mathematically equivalent expressions like `x**3` and `x**3.0` can disagree, because many implementations of that language treat integer and floating-point exponents in the power operator differently. Even worse, when x is negative, `x**3` succeeds, but `x**3.0` fails with a *negative-argument error* report from the logarithm function that the user did not even call.

14.4 Integer powers, revisited

Long after the preceding section was written, an important new paper with the title *Computing correctly rounded integer powers in floating-point arithmetic* appeared [KLL+10]. Its authors set out to answer the question of how much additional precision is needed for that job, by finding worst cases for integer powers. The worst cases arise when the exact power lies almost exactly halfway between two machine numbers.

Only x values in $[1, \beta)$ need to be tested in the search, because all other finite floating-point numbers in the same representation can differ from those values only by an integer power of the base. Thus, if $y = \beta^p x$, then $y^n = \beta^{pn} x^n$, so the powers x^n and y^n have the same significand. Despite that simplification, the effort required is substantial. The researchers report that a search using all integer powers from 3 to 733 consumed nearly *75 CPU years* on a network of processors! The search time grows with the powers, and the largest powers each require almost a CPU month.

For the IEEE 754 64-bit binary format, the worst case found by their exhaustive search is the power $n = 458$ of the value

$$\begin{aligned} x &= \texttt{0x1.0f38\_cfaa\_cb71\_ap0} \\ &= 1.059\_460\_620\_115\_209\_028\_568\_870\_053\_277\_350\_962\_162\_017\_822\_265\_625 \qquad \textit{exact decimal value,} \end{aligned}$$

for which

$$\begin{aligned} x^{458} &= \texttt{+0x1.1f0b\_0876\_ba02\_5800\_0000\_0000\_0000\_3bbf\_}\cdots\texttt{p+38} \\ &= \texttt{+0x1.1f0b\_0876\_ba02\_6p+38} \qquad \textit{correctly rounded.} \end{aligned}$$

The 1-bit following the last stored bit is the rounding bit, and forms the halfway case. It is followed by 61 0-bits before another 1-bit is met that forces upward rounding of the halfway case.

That result shows that correctly rounded integer powers up to $n = 733$ of all IEEE 754 64-bit values can be computed if we generate intermediate products with at least $53 + 1 + 61 + 1 = 116$ bits of precision. Unfortunately, that is more than we have with the 64-bit significand of the 80-bit format, or with pair-precision arithmetic ($53 + 53 = 106$ bits), or with the 113-bit significand of the IEEE 754 128-bit format.

The paper's authors examined the lengths of runs of 0-bits following the rounding bit, and showed that the counts of worst-case powers that produce a particular run length drop rapidly as those lengths increase, suggesting that powers $n > 733$ are unlikely to produce much longer runs. However, it is at present computationally impractical to continue the search for worst cases with even higher powers.

They then go on to consider various alternatives for computing integer powers with sufficient precision to guarantee correctly rounded results, including tricks on particular hardware platforms, and computation of x^n from $\exp(n \log(x))$.

Because we want our library code to work on a wide range of architectures, the simplest implementation choice seems to be to implement our bitwise-reduced power algorithm using pair-precision arithmetic of the *highest available* precision. On systems that provide at least one precision beyond the normal `float` and `double` offerings of C89, we can then guarantee correct rounding for all integer powers of those two basic types up to $n = 733$, and with high probability, for even higher powers. On those deficient systems that provide only two floating-point formats, we cannot guarantee that integer powers (up to $n = 733$) are always correctly rounded, but using normal pair-precision arithmetic reduces the incidence of incorrect rounding to negligible levels for most users.

The revised code in `ipowx.h` that implements the new algorithm is therefore selected by default, but can be suppressed by a suitable compile-time definition of a preprocessor symbol.

Extensive numerical tests of our new code with more than 10^{12} random arguments x in $[1, 2)$, and random integer powers in $[0, 6145]$, compared to high-precision symbolic algebra computations, find *no* instances of incorrect rounding.

The paper's authors do not address negative powers. We handle them by computing the corresponding positive power in extended pair-precision arithmetic and then reciprocating that result with error correction.

14.5 Outline of the power-function algorithm

The key to accurate computation of the power function is a product representation of the function where most of the factors can be computed exactly. We begin by rewriting the function in terms of the exponential and logarithm of a number n related to the base of the floating-point system:

$$\begin{aligned} z &= \mathrm{pow}(x, y) \\ &= x^y \\ &= n^{y \log_n x} \\ &= n^w, \\ w &= y \log_n x. \end{aligned}$$

Define n by $\beta = n^K$. For $\beta = 2$, 4, 8, and 16, choose $n = 2$ so that $K = 1$, 2, 3, and 4. For $\beta = 10$, let $n = 10$ so that $K = 1$.

Next, assume that the cases of $x = \infty$, x is a NaN, and $x \leq 0$ have been handled according to **Table 14.1** on page 413, so that we can henceforth assume that x is finite, positive, and nonzero. Then decompose x into a fraction and a power of the base:

$$x = f\beta^m \qquad \textit{where } f \textit{ lies in } [1/\beta, 1).$$

That is an exact operation for which we can use either this C-style code

```
f = FREXP(x, &m);
```

or the Cody/Waite primitives:

```
m = INTXP(x);
f = SETXP(x, 0);
```

In all floating-point architectures used for high-performance computation, the number of bits in an exponent field is smaller than the number in a default-sized integer, so the variable `m` can be of type `int`.

Because we want to handle different bases of floating-point systems, introduce four new integers, C, p, q, and r defined as follows. First, pick C like this:

$$C = \begin{cases} 10^q & \textit{for a decimal base,} \\ 16^q & \textit{for nondecimal bases.} \end{cases}$$

We discuss the choice of q later in **Section 14.13** on page 438. Cody and Waite do not consider values of C other than 10 or 16, so they always have $q = 1$.

By a procedure described in **Section 14.6** on page 423, choose p in $[0, C]$ and r in $[0, K)$ such that

$$\begin{aligned} a &= n^{-p/C} && \textit{exact by lookup in a } (C+1)\textit{-entry table,} \\ g &= fn^r && \textit{exact product,} \\ g &\text{ in } [1/n, 1) && \textit{implicit definition of } r. \end{aligned}$$

For $\beta = 2$ and 10, we always have $r = 0$ and $g = f$. For $\beta = 4$, 8, and 16, we have $r = 0$, 1, 2, and 3, and $r > 0$ is possible only when $f < \frac{1}{2}$.

Although it is not immediately evident, the computation of g is always exact, because when $r > 0$, g has leading zero bits, and scaling by $n^r = 2$, 4, or 8 merely results in shifting the fraction bits to the left, introducing zero bits on the right. No bits are lost by that process. The code to find r is simple, and not even required when $\beta = 2$ or 10:

```
g = f;
r = 0;

while (g < HALF)        /* move g from [1/beta,1) to [1/2,1) */
{
    g += g;
    r++;
}
```

Next, rearrange the expansion of f to read

$$\begin{aligned} f &= gn^{-r} \\ &= (g/a)n^{-r-p/C} \end{aligned}$$

and take the base-n logarithm of both sides to find

$$\log_n f = \log_n(g/a) - r - p/C.$$

We compute the logarithm on the right by a polynomial approximation, and the other two terms are exactly representable because r and p are small integers, and C is a small power of the base β.

We then have

$$\begin{aligned} w &= \log_n z \\ &= y \log_n x \\ &= y \log_n(f\beta^m) \\ &= y(\log_n f + m \log_n \beta) \\ &= y(\log_n f + m \log_n(n^K)) \\ &= y(\log_n f + mK) \\ &= y(Km - r - p/C + \log_n(g/a)) \\ &= y(u_1 + u_2), \end{aligned}$$

where the two critical quantities to be computed are

$$\begin{aligned} u_1 &= Km - r - p/C, && \textit{exactly representable,} \\ u_2 &= \log_n(g/a), && \textit{via polynomial approximation.} \end{aligned}$$

and $|u_1|$ is strictly greater than $|u_2|$, so that the digits of those two values cannot overlap.

Finally, split the computed w into a sum of two parts:

$$\begin{aligned} w &= w_1 + w_2, \\ w_1 &= Km' - r' - p'/C && \textit{exactly representable,} \\ w_2 &\text{ in } (-1/C, 0]. \end{aligned}$$

Here as well, the digits of w_1 and w_2 cannot overlap.

The new variables in the definition of w_1 are whole numbers defined by these conditions:

$$m' \geq 0, \qquad p' \text{ in } [0, C], \qquad r' \text{ in } [0, 3].$$

As written here, m' could be too large to represent as an integer, but we show later that we can delay computing its value until w_1 is known to be sufficiently small that an `int` data type suffices for m', p', and r'.

The accurate computation of w, w_1, and w_2 is the hardest part of the algorithm, but we delay the details until **Section 14.11** on page 433.

Finally, we recover the power function like this:

$$\begin{aligned} z &= \text{pow}(x, y) \\ &= n^{w_1 + w_2} \\ &= n^{Km' - r' - p'/C + w_2} \\ &= (n^K)^{m'} n^{-r' - p'/C + w_2} \\ &= \beta^{m'}(n^{-r'} n^{-p'/C} n^{w_2}). \end{aligned}$$

The factor $n^{-r'}$ is exact, and obtained by lookup in a small table. The factor $n^{-p'/C}$ is not exactly representable, but is correctly rounded to working precision and obtained by table lookup. The final factor n^{w_2} is not exactly representable either, but is computed by another polynomial approximation. None of the magnitudes of those three factors exceeds one, and all are close to that number, so there is no possibility of underflow in their product. We therefore compute their product first, and then carry out the final exact scaling by $\beta^{m'}$, not by direct multiplication, but instead by exponent adjustment using a function from the `ldexp()` or `setxp()` families.

The final scaling by $\beta^{m'}$ can overflow or underflow, but that merely indicates that the power is too large or too small to represent. The function return value is then either an infinity (or the largest representable number, if infinities are not supported), or zero. We show later how to detect the overflow and underflow conditions before they happen, allowing early exit before even computing the three powers of n.

If we compute the three powers of n on the right to higher than working precision, their product will almost always be correctly rounded. For example, if we have k extra digits, then we expect a rounding error only about once in β^k times. To see that, take $\beta = 10$ and $k = 2$: the extra k digits are then $00, \ldots, 49, 50, 51, \ldots, 99$. Only when the extra digits are 50 is the rounding in question: we then need to know even more digits to find out whether they are all zero, in which case the current rounding rule decides whether we round up or down. Otherwise, one or more of the trailing digits is nonzero, and we round up. From that example, we see that only once in $\beta^k = 100$ times does correct rounding require more than $k = 2$ extra digits.

On floating-point architectures that support both single- and double-precision data types (see **Table H.1** on page 948), there is sufficient extra precision in the latter that, by using it, we can guarantee correctly rounded results for the single-precision power function for at least all but one in 10^8 to 10^{10} random argument pairs.

Of the four factors in the product that defines z, all but n^{w_2} are either exact, or known to be correct to working precision. Thus, the accuracy of the power function depends critically on that of n^{w_2}, and it is precisely that step where most algorithms for the power function, including that of Cody and Waite, fail to deliver the needed accuracy.

14.6 Finding a and p

From the definition of $\log_n f = \log_n(g/a) - r - p/C$, we want to make the magnitude of the $\log_n(g/a)$ term as small as possible, because the two exact terms then dominate the sum when their contribution is nonzero. In such a case, we have effectively extended the precision of $\log_n f$ beyond working precision. The logarithm on the right has small magnitude when $g/a \approx 1$.

We have table values $A[p] = n^{-p/C}$ whose entries decrease from $A[0] = 1$ to $A[C] = 1/n$. There are no leading zero bits in any entry when $\beta = 4, 8,$ or 16, because the entries all lie in $[\frac{1}{2}, 1]$, so there are no problems from wobbling precision. We need to find the index p for which choosing $a = A[p]$ makes g/a closest to one. Because $f < 1$, we know that $g < 1$, so we have $g < A[0]$. If we can find an interval such that

$$A[k] > g \geq A[k+1] \qquad\qquad k \text{ in } [0, C),$$

then $p = k$ or $p = k+1$. We make the choice between them by comparing $1 - g/A[k]$ with $g/A[k+1] - 1$. Both expressions are nonnegative, but require two relatively costly divides, so we multiply them by $A[k]A[k+1]$ and instead compare $A[k+1](A[k] - g)$ with $A[k](g - A[k+1])$. If the first is smaller, set $p = k$; otherwise, set $p = k+1$.

A linear search in a table of N entries requires on average $\lceil N/2 \rceil$ comparisons, but because we know that $A[0] > g \geq A[C]$, we need not compare g with either endpoint, so we have an average of only $\lceil (N-2)/2 \rceil = \lceil (C-1)/2 \rceil$ comparisons.

A binary search for the containing interval in an *ordered* table of N entries takes at most $\lceil \log_2 N \rceil$ comparisons. We describe it later in this section.

We show in **Section 14.8** on page 426 that we require $A[k]$ to higher than working precision, so we represent it as two tables, such that $A[k] = A_1[k] + A_2[k]$, where $A_1[k]$ is correctly rounded to working precision, and $A_2[k]$ is a small correction that may be either positive, zero, or negative. Only $A_2[0]$ and $A_2[C]$ are zero, because only $A_1[0] = 1$ and $A_1[C] = 1/n$ are exact.

It is critical that the entries in the tables $A_1[\,]$ and $A_2[\,]$ be represented *exactly* to working precision, and because their computation would need more than double the normal working precision, we cannot compute them on-the-fly in the initialization code. Instead, we compute them to high accuracy in a symbolic-algebra system and store their values as compile-time initializers in header files. Here is a fragment of that code for decimal arithmetic:

```
#define TEN_TO_7__      FP(10000000.0)

static const fp_t A1[] =
{
    /* exact: A1[p] == 10^(-p/10) rounded to T digits */
    /*  0 */ FP( 10000000.0) / TEN_TO_7__,
    /*  1 */ FP(  7943282.0) / TEN_TO_7__,
...
    /*  9 */ FP(  1258925.0) / TEN_TO_7__,
    /* 10 */ FP(  1000000.0) / TEN_TO_7__
};

static const fp_t A2[] =
{
    /* correction: A2[p] == 10^(-p/10) - A1[p] */
    /*  0 */ ( FP(        0.0) / TEN_TO_7__ ) / TEN_TO_7__,
    /*  1 */ ( FP(  3472428.0) / TEN_TO_7__ ) / TEN_TO_7__,
...
    /*  9 */ ( FP(  4117942.0) / TEN_TO_7__ ) / TEN_TO_7__,
    /* 10 */ ( FP(        0.0) / TEN_TO_7__ ) / TEN_TO_7__
};
```

Each value is defined in rational form, such that both numerator and denominator are exactly representable, and the denominator is a power of the base. Their compile-time evaluation is then exact.

We need those tables for each supported host precision, so there are several different sets: at the time of writing this, four for decimal arithmetic, and eleven for binary arithmetic.

Cody and Waite choose $C = 10$ or $C = 16$. However, they also introduce a small storage economization that keeps only odd-numbered entries in $A_2[\,]$. Their search condition then requires finding $A[k] > g \geq A[k+2]$ where k is even, and they then choose $p = k + 1$. The storage savings of five or eight words, even with computer memories of a few hundred kilobytes in the 1970s, is not much, and anyway, might be offset in increased code size from more complicated indexing of the arrays. Unfortunately, the storage parsimony introduces a serious problem. When g matches a table entry $A[k]$, the most important of which is $g = A[C] = 1/n$, their algorithm chooses $a = A[k-1]$, making g/a as far as possible from 1, so subsequent errors in the computation of $\log_n(g/a)$ can result in powers of the base, and of one, being slightly incorrect. The optimal choice is $a = A[k]$, for which $\log_n(g/a) = 0$ exactly. With that choice, powers of the base and of one are always exact. Our new code handles those two cases separately for speed (see Table 14.1 on page 413), but also enhances accuracy when the old algorithm is used.

We improve on the Cody/Waite algorithm by always choosing the optimal a value, and thus, we must store all entries of $A_2[\,]$. No special indexing of that table is then required.

14.7 Table searching

For $C = 10$, a linear search averages five comparisons. By contrast, a binary search would need at most four, but requires more complex code. Therefore, for $C = 10$, Cody and Waite pick the simpler linear search, but unfortunately, they make an error. They use a search equivalent to this code fragment:

```
k = 0;

while (g <= A1[k+2])
    k += 2;

p = k + 1;
```

The loop exits when $g > A_1[k+2]$. That works correctly for all but values of g identical to some $A_1[k]$, in particular, $g = A_1[C] = 1/10$. In that case, the last iteration has $k = C - 2$ with the test $g \leq A_1[C]$, which is true, so k advances to C. The next iteration compares g with the nonexistent table entry $A_1[C+2]$, and the loop continues until the bounds violation is detected and the job is terminated, or unpredictable storage contents treated as a floating-point number cause the loop to exit. The value of p is then certainly incorrect. That error was caught by `assert` calls placed in our early code for the decimal power function, and together with other errors found in their algorithm for the decimal case, suggests that the power function for decimal arithmetic was never actually implemented and tested.

Here is how to do the linear search correctly and find the optimum a:

```
assert((A1[0] > g) && (g >= A1[C]));

k = 0;

while (A1[k+1] > g)
    k++;

assert((A1[k] > g) && (g >= A1[k+1]));

p = ((A1[k+1]*(A1[k] - g)) < (A1[k]*(g - A1[k+1]))) ? k : k + 1;
a = A1[p];
```

When $C = 16$, a linear search takes seven comparisons on average, and a binary search takes at most four. Binary search in an ordered table is a divide-and-conquer algorithm: split a big problem into two roughly equal parts, one of which can easily be determined to contain the solution, and the other discarded. Then repeat the split on the now half-size problem, until the problem can be solved by inspection. As long as the splits produce equal-sized parts, the computation terminates after $\lceil \log_2 N \rceil$ steps for a problem of size N. For binary search, we compare g with the middle element, and then continue with whichever half of the table is known to contain g.

Although the description of binary search makes it appear simple, it is not easy to program. The key to correct code for binary search is to maintain a *loop invariant*, namely that, given two table indices `lo` and `hi` marking the left and right ends of the array section to be searched, we then guarantee that the conditions

$$(\text{lo} < (\text{hi} - 1)) \text{ AND } (A[\text{lo}] > g) \text{ AND } (g \geq A[\text{hi}])$$

hold during each iteration of the loop. The loop terminates as soon as any one of them fails.

This function does the job of finding p such that g lies in $(A[p], A[p+1]]$:

```
static int
binsearch (fp_t g, const fp_t A[], int C)
{   /* return p such that (A[p] > g) && (g >= A[p+1]) */
    int lo, hi, middle;     /* array indices in [0,C] */

    /* verify initial condition */
    assert((C > 0) && (A[0] > g) && (g >= A[C]));

    lo = 0;
    hi = C;

    while (lo < (hi - 1))   /* first loop invariant */
    {
        middle = lo + (hi - lo)/2;

        if (A[middle] > g)
            lo = middle;
        else
            hi = middle;
        /* remaining invariants: (A[lo] > g) && (g >= A[hi]) */
    }

    return (lo);
}
```

The requirement that `lo` and `hi` be separated by at least two when the loop body is entered ensures that `middle` differs from both, forcing the index range to shrink on each iteration, and guaranteeing loop termination. The computation of `middle` as written avoids a possible uncaught integer overflow from the traditional form `(lo + hi)/2`, as described in **Appendix I.5** on page 976. For the special case $C = 16$, Cody and Waite unroll the function and its loop into three inline tests, but we add a fourth and final test to find the precise interval:

```
p = 0;

if (A1[p + 8] > g) p += 8;
if (A1[p + 4] > g) p += 4;
if (A1[p + 2] > g) p += 2;
if (A1[p + 1] > g) p += 1;
```

Because C is in practice a compile-time constant, larger values of C can be handled similarly, as long as C is a power of two. Start with the first test like this:

```
if (A1[p + C/2] > g) p += C/2;
```

Then repeat it with successive halving of the constant $C/2$ until the last test is against $A_1[p+1]$. However, for larger values of C, we are more likely to use the general `binsearch()` function instead.

14.8 Computing $\log_n(g/a)$

The first approximation that we need to make in the computation of the power function is finding $\log_n(g/a)$, where $g/a \approx 1$. We discuss that problem in **Section 10.4** on page 290, where we show that it is computationally much better to use the companion `log1p()` function: $\log(g/a) = \log(1 + (g/a - 1)) = \operatorname{log1p}(g/a - 1)$, as long as we can compute the argument $g/a - 1$ accurately. However, that function has only recently been included in the C library, and is absent from most other programming languages.

Cody and Waite recognize the difficulty of accurate computation of the logarithm of arguments near one. They prefer to keep the elementary functions largely independent of one another, and because a faster algorithm is possible when the argument range is small, they handle the problem by first introducing a change of variable:

$$\begin{aligned} s &= (g/a - 1)/(g/a + 1) \\ &= (g - a)/(g + a), \\ g/a &= (1 + s)/(1 - s), && \textit{inverse relation.} \end{aligned}$$

Digit loss in the subtraction $g - a$ could produce an inaccurate s, but we can prevent that by observing that because our choice of g and a guarantees that they have the same floating-point exponent, the subtraction is exact, and the loss is entirely due to insufficient precision in a. We therefore compute s with code like this:

```
s = g - A1[p];          /* exact (rounding-free) subtraction */
s -= A2[p];
s /= (g + A1[p]);
```

Because $a = A_1[p] + A_2[p]$ to twice the working precision, the two-stage subtraction is correct to at least working precision.

In terms of the new variable s, the Taylor-series expansion of $\log_n(g/a)$ about $s = 0$ (that is, $g/a = 1$), is

$$\begin{aligned} \log_n(g/a) &= \log_n((1 + s)/(1 - s)) \\ &= (2/\ln(n))(s + s^3/3 + s^5/5 + s^7/7 + \cdots) \\ &= (2/\ln(n))(s + s^3(1/3 + s^2/5 + s^4/7 + \cdots)) \\ &= (2/\ln(n))s + s^3((2/\ln(n))(1/3 + s^2/5 + s^4/7 + \cdots)). \end{aligned}$$

Because g/a is close to one, s is small, and the series converges quickly. The inner series is almost a straight line, and is therefore likely to be well-represented by a low-order fit to a polynomial, or a ratio of polynomials. We therefore compute the needed logarithm like this:

$$\begin{aligned} \log_n(g/a) &= (2/\ln(n))s + s^3 R_n(s^2), && \textit{default computation,} \\ &= s \times \operatorname{fma}(s^2, R_n(s^2), (2/\ln(n))), && \textit{improved computation.} \end{aligned}$$

Table 14.3: Variable limits in the computation of $\log_n(g/a)$. The range of s is symmetric: $[-s_{\max}, +s_{\max}]$. For $n = 2$, $\zeta = 2s$ and $t = \zeta^2$. For $n = 10$, $v = s^2$.

n	C	$s_{\max}$	$t_{\max}$
2	16	1.09473e-02	4.79371e-04
2	256	6.77360e-04	1.83526e-06
2	4096	4.23081e-05	7.15991e-09
n	C	$s_{\max}$	$v_{\max}$
10	10	6.07959e-02	3.69615e-03
10	100	5.78953e-03	3.35187e-05
10	1000	5.75978e-04	3.31750e-07
10	10000	5.75679e-05	3.31407e-09

We introduce another change of variable, $v = s^2$, which is even smaller than s, and guaranteed to be nonnegative, and then solve for $R_n(v)$:

$$\begin{aligned} R_n(v) &= (\log_n((1+s)/(1-s)) - (2/\ln(n))s)/s^3 \\ &= (\log_n((1+\sqrt{v})/(1-\sqrt{v})) - (2/\ln(n))\sqrt{v})/(v\sqrt{v}). \end{aligned}$$

In order to avoid a divide-by-zero condition during the polynomial fit for $R_n(v)$, we need to know the value of $R_n(0)$, which is just the first term of the inner series:

$$\lim_{v\to 0} R_n(v) = 2/(3\ln n).$$

For the fit, we also need to know the range of v, which must have the form $[0, v_{\max}]$. To investigate that, we solve for the g value, g_c, at the point where the definition of a changes:

$$\begin{aligned} 1 - g_c/A[p] &= g_c/A[p+1] - 1, \\ g_c &= 2A[p]A[p+1]/(A[p] + A[p+1]). \end{aligned}$$

The value g_c produces the g/a ratio furthest from one, and that is where we have the maximum value of $|s|$:

$$\begin{aligned} s_{\max} &= -(g_c - A[p])/(g_c + A[p]) \\ &= (1 - A[1])/(3A[1] + 1). \end{aligned}$$

We see that $s_{\max}$ is independent of p, but depends on the constants n and C. The range of s is $[-s_{\max}, +s_{\max}]$, so the range of v is $[0, s^2_{\max}]$. Sample values are shown in Table 14.3.

As we have seen with other functions approximated by polynomials, we cannot predict the polynomial degree needed to achieve a particular accuracy. Instead, we make several different fits, and then tabulate the accuracy obtained for each polynomial degree. For $\log_n(g/a)$, some useful results are given in Table 14.4 on the next page. From that table, it should now be clear why we generalized the original Cody/Waite choice of C from 10 and 16 to powers of those numbers. Larger C values mean bigger storage requirements for the tables $A_1[\,]$ and $A_2[\,]$, but they also reduce the polynomial degree required to achieve a specified accuracy, and therefore make the computation faster. Memory sizes on historical machines could not justify larger tables, but modern machines can easily handle them.

Because we need to support a broad range of precisions in the `mathcw` library, we include data for several different polynomial fits in the header files `pxy*.h`. The more accurately that we can compute $u_2 = \log_n(g/a)$, the better job we can do for w_2, and for the power function.

For nondecimal bases, Cody and Waite proceed slightly differently, and make yet another change of variable: $\zeta = 2s$. We show the reason for that substitution at the end of this section. To counteract the effect of wobbling precision, compute ζ like this:

$$\zeta = \begin{cases} s + s & \text{when } \beta = 2, \\ ((g - A_1[p]) - A_2[p])/(\tfrac{1}{2}g + \tfrac{1}{2}A_1[p]) & \text{when } \beta = 4, 8, \text{ or } 16. \end{cases}$$

Table 14.4: Accuracy in decimal digits of rational polynomial fits of degree $\langle p/q \rangle$ to the auxiliary functions $R(v)$ and $\mathcal{R}(t)$ needed to compute $\log_n(g/a)$.

		Degree							
C	$v_{\max}$	**⟨1/1⟩**	**⟨1/2⟩**	**⟨2/2⟩**	**⟨3/3⟩**	**⟨5/5⟩**	**⟨7/7⟩**	**⟨9/9⟩**	**⟨9/10⟩**
10	3.69615e-03	10	14	18	25	40	54	69	72
100	3.35187e-05	17	22	28	39	62	85	108	113
1000	3.31750e-07	23	30	38	53	84	115	146	153
10000	3.31407e-09	29	38	48	67	106	145	184	193
		Degree							
C	$t_{\max}$	**⟨1/1⟩**	**⟨1/2⟩**	**⟨2/2⟩**	**⟨3/3⟩**	**⟨5/5⟩**	**⟨7/7⟩**	**⟨9/9⟩**	**⟨9/10⟩**
16	4.79371e-04	13	17	22	31	49	67	85	101
256	1.83526e-06	20	27	34	48	75	103	131	138
4096	7.15991e-09	27	37	46	64	102	139	177	186

The factor $\frac{1}{2}$ has no leading zero bits for $\beta = 16$, whereas 2 has two leading zero bits, and could unnecessarily reduce the accuracy of ζ if we just computed $2s$ or $s + s$ directly.

The logarithm then becomes:

$$\begin{aligned}
\log_n(g/a) &= \log_n((1 + (\zeta/2))/(1 - (\zeta/2))) \\
&= \log_n((2 + \zeta)/(2 - \zeta)) \\
&= (1/\ln n)(\zeta + (1/12)\zeta^3 + (1/80)\zeta^5 + (1/448)\zeta^7 + \cdots) \\
&= (\log_n e)(\zeta + (1/12)\zeta^3 + (1/80)\zeta^5 + (1/448)\zeta^7 + \cdots) \\
&= (\log_n e)(\zeta + \zeta^3((1/12) + (1/80)\zeta^2 + (1/448)\zeta^4 + \cdots)) \\
&= (\log_n e)(\zeta + \zeta^3\mathcal{R}_n(\zeta^2)), \\
\mathcal{R}_n(\zeta^2) &= ((\ln n)\log_n((2 + \zeta)/(2 - \zeta)) - \zeta)/\zeta^3.
\end{aligned}$$

With another variable transformation, $t = \zeta^2$, the new approximating function to which we fit rational polynomials is then defined by:

$$\mathcal{R}_n(t) = ((\ln n)\log_n((2 + \sqrt{t})/(2 - \sqrt{t})) - \sqrt{t})/(t\sqrt{t}).$$

This time, the outer logarithmic constant is not factored into the approximating function.

The value of $\mathcal{R}_n(0)$ is needed to prevent a zero divide during the polynomial fit, and from the first term of the inner series, we find this limit:

$$\lim_{t \to 0} \mathcal{R}_n(t) = 1/12.$$

There is one final issue to be dealt with: the outer constant multiplier, $\log_n e \approx 1.44269\ldots$, has up to three leading zero bits in binary bases when $\beta > 2$, and thus effectively reduces overall accuracy by almost one decimal digit. Cody and Waite therefore split the constant into two parts: $1 + L$, where $L = \log_n e - 1$ is a stored constant with only one leading zero bit when $\beta > 2$. We can then find the logarithm like this:

$$\begin{aligned}
\log_n(g/a) &= (\zeta + \zeta^3\mathcal{R}_n(\zeta^2)) + L(\zeta + \zeta^3\mathcal{R}_n(\zeta^2)) \\
&= (\zeta + \zeta t\mathcal{R}_n(t)) + L(\zeta + \zeta t\mathcal{R}_n(t)) \\
&= \zeta + L\zeta + (L\zeta t\mathcal{R}_n(t) + \zeta t\mathcal{R}_n(t)), \\
b &= \zeta t\mathcal{R}_n(t), \\
\log_n(g/a) &= \zeta + (L\zeta + (Lb + b)), && \textit{default computation,} \\
&= \zeta + \mathrm{fma}(L, \zeta, \mathrm{fma}(L, b, b)), && \textit{improved computation.}
\end{aligned}$$

In that form, the leading term is exact, and that is why Cody and Waite made the variable change $\zeta = 2s$.

The fused multiply-add operations should be used when available, and otherwise, the computation must obey the parentheses in the preceding expression. The variable ζ must *not* be factored out of the right-hand side, because that would put wobbling precision back into the computation in the second factor of $\zeta(1 + \cdots)$.

14.9 Accuracy required for $\log_n(g/a)$

The part of the power-function algorithm that has the largest effect on the overall accuracy is the computation and splitting of the product w, where

$$\begin{aligned} w &= y(u_1 + u_2) \\ &= w_1 + w_2, \\ w_1 &= Km' - r' - p'/C \qquad \text{\textit{exactly representable,}} \\ w_2 &\text{ in } (-1/C, 0]. \end{aligned}$$

Exact computation of the sum $u_1 + u_2$ in general requires more than working precision, and its exact product with y thus needs more than twice working precision. We show later that when we require the decomposition of w_1, it has at most as many digits before the point as there are digits in the floating-point exponent, and we need w_2 to full precision, and preferably, correctly rounded.

As a concrete example, consider the IEEE 754 32-bit decimal format, for which the exponent range is $[-101, +96]$, and the precision is seven decimal digits. Then w_1 takes the form `ddd.d` and w_2 looks like `0.0ddddddd`. The last digit in w_2 has value $d \times 10^{-8}$. The power function $x^y = n^w$ overflows for $w \geq 97$, and underflows for $w < -101$. The smallest nonzero magnitude of $\log_{10}(g/a) = \log_{10}((1+s)/(1-s))$ is obtained with $s = 10^{-6}$, where the Taylor series expansion shows that $u_2 \approx 10^{-6}$, and the last stored digit of u_2 has value $d \times 10^{-12}$. The largest possible y that does not cause overflow is thus $y \approx 10^2/10^{-6} \approx 10^8$, so the product yu_2 shifts digits of u_2 left *eight* places, and the low-order digit of u_2 affects w_2 in the *fourth* decimal place. Because we must determine w_2 to eight decimal places, we need at least *four* more digits in u_2.

More generally, consider base β and t digits of precision in w_2, and T digits of precision in u_2, where T is to be determined. Let w_2 have $j > 0$ leading zero digits after the point, and u_2 have $k > 0$ such digits. The value of a unit last digit in w_2 is then $\beta^{-(t+j)}$, and that of a unit final digit in u_2 is $\beta^{-(T+k)}$. Further, let there be e digits before the fractional point in y. The product yu_2 then shifts the effect of the last digit of u_2 left by e places, and we want that to be in the last digit of w_2. We then compare digit values to find T like this:

$$\begin{aligned} -(t+j) &= -(T+k) + e \\ T &= t + j - k + e \end{aligned}$$

Evidently, T is maximal when j is large and $k = 1$, minimal when $j = 1$ and k is large, and in the common case where $j = 1$ and $k = 1$, we have $T = t + e$.

From the Taylor series expansion, the smallest nonzero magnitude of u_2 is $(2/\ln(n))s = (2/\ln(n))\beta^{1-t}$, so we see that $k < t$ because the logarithm factor is $2/\ln(n) = 2.885\ldots$ for $n = 2$, and $0.868\ldots$ for $n = 10$. The smallest value of T is therefore $t + 1 - (t-1) + e = e + 2$, and when w_2 has $j = t - 1$ leading fractional zero digits, the maximal value of T is $2t - 2 + e$. Because we have $e \geq 2$ in all reasonable floating-point systems, *we require* $T > 2t$ *digits in* u_2.

Our rough analysis leads to the inescapable conclusion that we are doomed to an inaccuracy of a few trailing digits in results from the power function unless we can compute u_2 with *at least twice* the working precision. The Cody and Waite algorithm does not do so, but in the `mathcw` library, we do. The design of that library has, for each function, a single algorithm file with a parametrized working-precision data type, `fp_t`, and a secondary type, `hp_t`, that represents the next higher precision, when available, and otherwise, is identical to `fp_t`. The simplest solution that makes higher-precision computation of u_2 possible, at least for the shorter data types, is to make visible the internal function that handles the computation of x^y for finite positive nonzero x, after all of the special cases have been handled. That function is normally a private one with code that looks like this:

```
static fp_t
powpy (fp_t x, fp_t y) /* POW(positive nonzero x, non-NaN y) */
{
    /* ... body omitted ... */
}
```

We then move the body into another function in the separate algorithm file, `pxyx.h`, and replace the body of `powpy()` with this statement:

```
    return ((fp_t)HP_POWPY((hp_t)x, (hp_t)y));
```

The macro HP_POWPY() is defined to be a name like _pxy(), where the leading underscore indicates that it is a private internal function that must not be called directly by user code.

Thus, calls to the private powpy() function elsewhere in the algorithm file are trapped, the arguments are promoted to hp_t, a higher-precision function compiled from one of the companion files, pxy*.c, is called to do the work, and the result is then coerced back to fp_t and returned to the caller.

When fp_t is the highest available precision, HP_POWPY() is simply defined to be the name of the function at that precision, for example, _pxyll(). The computation is then done at working precision, and is therefore less accurate than we would like. One way to repair that problem is to implement an even higher precision in software, but we show an easier way later in **Section 14.13** on page 438.

Once that change was made to the library, the ELEFUNT test results improved dramatically, reporting zero bit loss for all but the highest available precision.

The need for an excursion into higher precision for part of the computation, and the coding practices of the mathcw library, mandate the separation of the power-function computation into two algorithm files, powx.h and pxyx.h. That is the only instance in the mathcw library where that has been necessary.

14.10 Exact products

In the next section, and elsewhere in the mathcw library, we need to compute exact products. Although many early computer architectures provided instructions for computing the $2t$-digit product of two t-digit numbers, such hardware support is rare on modern systems. A software solution is therefore required to produce the result as an exact sum of high and low parts.

When a floating-point format of at least twice the working precision, and extended exponent range, is available, then type promotion provides an easy implementation. If the product magnitude exceeds the overflow limit of the working precision, then scaling must also be provided. In our applications, product overflow is not expected, so we ignore the scaling problem.

When a fused multiply-add operation is available, and the arguments are finite and nonzero, the code is short:

```
#define HI      result[0]
#define LO      result[1]

void
FMUL(fp_t result[/* 2 */], fp_t x, fp_t y)
{   /* x * y = result[0] + result[1] (exact if no over/underflow) */
    HI = x * y;
    LO = (ISINF(HI) || (HI == ZERO)) ? HI : FMA(x, y, -HI);
}
```

That works because the high part is the product rounded to working precision, and the low part is the rounding error in the product. Extra code must be supplied to handle the cases of zero, Infinity, and NaN arguments, but we delay its presentation until our second implementation of FMUL() later in this section.

In the absence of FMA() support, we need to reduce the double-length product to a sum of terms, each of which can be computed exactly, except possibly for the term of smallest magnitude. That requires splitting each operand into a sum of high and low parts, where the high part has $\lfloor t/2 \rfloor$ digits, and the low part has the remaining $\lceil t/2 \rceil$ digits.

If the number to be split is far from the overflow limit, then the split can be done with private code like this:

```
static void
safe_split(fp_t result[/* 2 */], fp_t x)
{   /* exact split of x as result[0] + result[1] (if no overflow) */
    static const fp_t one_plus_split = FP(1.0) + MCW_B_TO_CEIL_HALF_T;        /* 1 + beta**ceil(t/2) */
    volatile fp_t p, q;

    p = one_plus_split * x;
    STORE(&p);
    q = x - p;
    STORE(&q);
    HI = p + q;
    LO = x - HI;
}
```

We discuss that operation, and the conditions under which it is valid, in more detail in **Section 13.11** on page 359.

Our public code is more general, supplying checks for special arguments, and avoiding premature overflow:

```
void
FSPLIT(fp_t result[/* 2 */], fp_t x)
{   /* return exact split of x as result[0] + result[1] */
    static int do_init = 1;
    static fp_t one_minus_tad = FP(0.);

    if (do_init)
    {
        one_minus_tad = ONE - LDEXP(ONE, -(T / 2));
        split_overflow_limit = FP(0.75) /* fudge factor */ *
            FP_T_MAX / (FP(1.0) + MCW_B_TO_CEIL_HALF_T);
        do_init = 0;
    }

    if (ISNAN(x) || ISINF(x) || (x == ZERO))
        HI = LO = x;
    else
    {
        if (QABS(x) > split_overflow_limit)
        {
            fp_t f, s;
            int n;

            f = FREXP(x, &n);       /* x = f * BASE**n */
            safe_split(result, f);

            if (n >= FP_T_MAX_EXP)  /* rare case: prevent overflow */
            {
                if (QABS(HI) == ONE) /* special case needs new split */
                {
                    /* e.g., for BASE = 10, T = 7, and f = HI + LO = 1,
                       we have HI = 1 - 0.001 = 0.999, LO = 0.001 */
                    HI = COPYSIGN(one_minus_tad, HI);
                    STORE(&HI);
                    LO = f - HI;
                    STORE(&LO);
                }

                HI = LDEXP(HI, n);
                LO = LDEXP(LO, n);
            }
            else            /* common case needs only one LDEXP() */
            {               /* call and overflow is impossible */
                s = LDEXP(ONE, n);
                HI *= s;    /* exact */
                LO *= s;    /* exact */
            }

        }
        else
            safe_split(result, x);
        if (LO == ZERO)
            LO = COPYSIGN(LO, HI);
    }
}
```

The one-time initialization block computes the value $1 - \beta^{-\lfloor t/2 \rfloor}$, and a large cutoff somewhat below the limit where the safe split would suffer overflow.

If x is Infinity, NaN, or a signed zero, we return that argument in the high and low parts with no further computation, and no error indication. Including a test for zero ensures consistent handling of its sign, so that $-0 = (-0) + (-0)$ and $+0 = (+0) + (+0)$.

If x is large enough to cause overflow in the split, we use FREXP() to reduce it to the exact product of a fraction f in $[1/\beta, 1)$, and a power of the base, β^n. The split of f is then safe, and we can usually scale the two parts exactly by β^n computed with LDEXP(). However, when the exponent is at its maximum and f is close to one, the split returns $f = 1 - \delta$. Rescaling the high part would then overflow. The solution is to check for that rare case, and resplit so that the high part has the required number of digits, and is less than one.

The code in FSPLIT() is nontrivial, and useful elsewhere, so it is supplied as a standard part of the mathcw library.

Programming the exact double-length product function is now straightforward, provided that we take care to check for several special cases where the arguments are Infinity, NaN, or signed zero. The code is adapted from that of our final PMUL2() function (see **Section 13.16** on page 370), but adds internal scaling to prevent premature overflow and underflow:

```
void
FMUL(fp_t result[], fp_t x, fp_t y)
{   /* x * y = result[0] + result[1] (exact if no overflow) */
    if (ISNAN(x))       /* NaN * any -> NaN */
        HI = LO = x;
    else if (ISNAN(y))  /* any * NaN -> NaN */
        HI = LO = y;
    else if (ISINF(x) || ISINF(y))
    {   /* Inf * nonNaN -> Inf (with correct sign) or NaN */
        if ( (x == ZERO) || (y == ZERO) ) /* Inf * 0 -> NaN */
            HI = LO = QNAN("");
        else            /* Inf * nonzero -> Inf (with correct sign) */
            HI = LO = (SIGNBIT(x) ^ SIGNBIT(y)) ? -INFTY() : INFTY();
    }
    else if ( (x == ZERO) || (y == ZERO) )
        HI = LO = (SIGNBIT(x) ^ SIGNBIT(y)) ?
                  COPYSIGN(ZERO, -ONE) : ZERO;
    else                /* x and y are finite and nonzero */
    {
        fp_t lo, xx[2], yy[2], zz[2];
        int nx, ny, nxy;
        volatile fp_t hi;

        x = FREXP(x, &nx);
        y = FREXP(y, &ny);
        nxy = nx + ny;                  /* exponent of product */

        FSPLIT(xx, x);
        FSPLIT(yy, y);
        FSPLIT(zz, yy[1]);

        hi = x * y;
        STORE(&hi);
        lo =  xx[0] * yy[0] - hi;
        lo += xx[0] * yy[1];
        lo += xx[1] * yy[0];
        lo += zz[0] * xx[1];
        lo += zz[1] * xx[1];

        if (lo == ZERO)
            lo = COPYSIGN(lo, hi);

        HI = LDEXP(hi, nxy);
        LO = ( ISINF(HI) || (HI == ZERO) ) ? HI : LDEXP(lo, nxy);
    }
}
```

For reasons discussed in **Section 13.11** on page 359, the decomposition of $x \times y$ as a sum of terms is exact only when $\beta = 2$ and none of the intermediate products underflows. Our initial scaling prevents product underflow, but if there is underflow in the final scaling, then trailing digits are lost. The programmer is therefore advised to work with suitably scaled x and y when `FMUL()` is used.

The best solution for nonbinary bases, $\beta > 2$, is the `FMA()` approach, because that guarantees a two-part split of the product with at most one rounding error.

14.11 Computing w, w_1 and w_2

In **Section 14.9** on page 429, we showed that u_2 must be accurate to more than working precision to compute the product $w = y(u_1 + u_2)$ satisfactorily. It is now time to examine how to do that.

Because the digits of u_1 and u_2 do not overlap, their sum requires even more precision, so roughly, we need nearly triple the working precision to find w. Cody and Waite suggest that when a higher-precision data type is available, then the simplest approach is to promote the operands to that precision, do the add and multiply, and then split the product into the sum $w_1 + w_2$. Unfortunately, that is still not good enough, because typical floating-point designs allocate only slightly more than twice the number of significand digits in each higher precision, and we need almost three times as many. Indeed, if the number of extra digits for u_2 cited in the preceding section is taken into account, none of the IEEE 754 formats is adequate, unless we promote from a 32-bit format to a 128-bit format, but the most widely available CPUs at the time of writing this provide no more than the 80-bit format, and some systems or compilers offer nothing beyond the 64-bit format.

Two alternatives present themselves. The first, and most obvious, is to supplement the elementary-function library with a multiple-precision arithmetic library, and do the computation and split of w that way. Few existing libraries take that approach, partly because it reduces performance, and partly because multiple-precision packages are neither standardized, nor particularly portable across a wide range of historical and current architectures.[1] For decimal floating-point arithmetic, code in `pxyx.h` employs the 128-bit data type for internal computations in the single-precision case, and for the other three precisions, uses IBM `decNumber` library routines [Cow07] to compute w, w_1, and w_2 in triple the working precision before converting them back to working precision. That ensures values of w_2 that are always correctly rounded, and because the early support for decimal arithmetic is implemented in software anyway, provides enhanced performance over alternative algorithms.

The second approach, and the one used by Cody and Waite, is to simulate the extra precision with inline code to handle just the single add and multiply needed for w. That is neither easy, nor obvious, and their code is particularly inscrutable, and offered without derivation or explanation. It is worthwhile to dissect their algorithm, so that we can see why, despite their care, it is still impossible to avoid loss of a few trailing digits in x^y for large y.

The critical operation that Cody and Waite introduce is called `REDUCE()`. Its job is to discard fractional digits after the first from its argument, where a digit is in base C, with $C = 10^q$ or 16^q for all practical floating-point architectures. They observe that there are multiple ways to do that, one of which is low-level bit manipulation on the floating-point representation. That way is unsatisfactory in a portable library, because it requires special code for each precision, each floating-point design, and even for different byte-addressing conventions.

The simplest approach based on tools that we already have is also portable: multiply the argument by C, convert to a whole number, and then multiply the result by $1/C$. For example, when $C = 10$, those operations convert

[1] Although the GNU `GMP` multiple-precision library for binary arithmetic and the IBM `decNumber` library for decimal arithmetic are generally easy to build and install on any system with IEEE 754 arithmetic, they would require major revisions for historical architectures, and those revisions are unlikely to be accepted or supported by the library developers.

123.456 to 1234.56 to 1234 to 123.4. That number has the same exponent as the original argument, and thus, we can subtract the two to recover the low-order part exactly: $123.456 - 123.4 = 0.056$. The catch is that the argument can be too large to represent as an integer data type, so we need to avoid a type conversion, and instead use a definition like this:

```
#define REDUCE(v)     (TRUNC(C * (v)) * ONE_OVER_C)
```

Thus, we recover the high and low parts *exactly* like this:

```
v1 = REDUCE(v);
v2 = v - v1;
```

Armed with that reduction, there are three important observations:

- Sums of high parts cannot introduce digits into the low part, but the reverse is not true: sums of low parts can produce a carry into the high part, requiring a second split.
- Products of high parts can spill into the low part, requiring a second split, but products of low parts cannot produce a high part.
- The two-part representation is the key to organizing the computation, because it allows us to keep the parts separate: the sum of all of the low parts determines w_2, although any carry has to be moved into the sum of all of the high parts that contribute to w_1.

With those points in mind, here is a code equivalent to the opaque Cody/Waite computation of w:

```
#define SPLIT(x1,x2,x)  (x1 = REDUCE(x), x2 = x - x1)

/* compute w = w1 + w2
             = y * (u1 + u2)
             = (y1 + y2) * (u1 + u2)
             = y1 * u1 + y2 * u1 + y1 * u2 + y2 * u2
             = y1 * u1 + y2 * u1 + y * u2
*/
SPLIT(y1, y2, y);

tmp = y * u2 + y2 * u1; /* three of four needed terms */
SPLIT(a1, a2, tmp);     /* a1 + a2 is sum of three terms */

tmp = a1 + y1 * u1;     /* add the fourth term to high part */
SPLIT(b1, b2, tmp);     /* b1 + b2 holds high part */

tmp = b2 + a2;          /* sum of low parts (carry possible) */
SPLIT(c1, c2, tmp);     /* c1 is carry, c2 is low part */

w1 = b1 + c1;           /* collect high parts */
w2 = c2;                /* copy low part */
iw1 = TRUNC(C * w1);    /* high part scaled to whole number */
```

The four splits are exact, but alas, the first two assignments to `tmp` involve products whose trailing bits are lost, because they are computed only to working precision. Thus, although the code recovers w_1 and w_2 reasonably well, the latter still suffers unacceptable accuracy loss.

Although the `pxyx.h` algorithm file in the `mathcw` library contains code equivalent to that, the code is no longer used unless it is enabled by defining a certain preprocessor symbol. Instead, we improve upon it by eliminating the trailing-digit loss in the three products. To do so, we employ our library primitive `FMUL()` to compute the exact double-length product as a sum of high- and low-order terms: $xy = r_1 + r_2$. That split is not quite what we need for computing w_1 and w_2, because the high part, r_1, has full precision, whereas w_1 has only one base-C fractional digit, but we can use `FMUL()` with this wrapper function:

```
static void
fmul2(fp_t result[], fp_t x, fp_t y)
{   /* split exact x * y into exact high and accurate, but approximate, low parts in result[] */
    fp_t r[2], s[2], t[2];

    FMUL(r, x, y);     /* x * y = r[0] + r[1] (exact) */

    if (ISNAN(r[0]) || ISINF(r[0]))     /* r[0] and r[1] identical */
        result[0] = result[1] = r[0];
    else
    {
        split(s, r[0]);
        split(t, r[1]);
        split(result, s[1] + t[1]);
        result[0] += s[0] + t[0];
    }
}
```

That function in turn needs this `split()` function:

```
static void
split (fp_t result[], fp_t x)
{   /* split x exactly into high and low parts in result[] */
    /* with just one base-C digit in high part */

    if (QABS(x) >= (FP_T_MAX / C))
    {   /* fast split would suffer intermediate overflow: rescale */
        if (ISINF(x))
            result[0] = result[1] = x;
        else
        {
            x *= C_INVERSE;                     /* exact */
            result[0] = REDUCE_POW(x);
            result[1] = x - result[0];
            result[0] *= C;                     /* exact */
            result[1] *= C;                     /* exact */
        }
    }
    else
    {
        result[0] = REDUCE_POW(x);
        result[1] = x - result[0];
    }
}
```

The function to compute the split of w is then straightforward:

```
static void
wsplit (fp_t *w1, fp_t *w2, fp_t y, fp_t u1, fp_t u2)
{   /* compute (w1 + w2) = y * u1 + y * u2 accurately */
    fp_t yu2[2];

    fmul2(yu2, y, u2);

    if (u1 == ZERO)
    {
        *w1 = yu2[0];
        *w2 = yu2[1];
    }
    else
    {
        fp_t r[2], yu1[2];

        fmul2(yu1, y, u1);
        split(r, yu1[1] + yu2[1]);
        *w1 = yu2[0] + r[0];  /* add smaller high parts first */
        *w1 += yu1[0];        /* add largest high part last */
        *w2 = r[1];
    }
}
```

We include special handling for the common case $u_1 = 0$, which happens with small-magnitude y arguments, because the computation is then simpler, faster, and more accurate. Otherwise, the job is straightforward bookkeeping of high and low parts of two double-length multiplications. ELEFUNT test results for the power function with that new code for computing w report bit loss smaller by a fraction of a digit compared to the original simpler algorithm.

Having computed w_1 and w_2, we can now easily determine whether $z = x^y = n^w$ would overflow or underflow. We just have to compare w against two limits, BIGW and SMALLW. The first is just $\log_n$(largest finite), and the second is $\log_n$(smallest subnormal), or if subnormal numbers are not supported, $\log_n$(smallest normal). On some systems, they could be precomputed constants defined at compile time, but on others, whether subnormals are available or not depends on compiler options or on library calls (see **Section 4.12** on page 78), so we compute them at run time in an initialization block that is executed only on the first call to the power function, and depends on a reasonably accurate logarithm function. However, because handling of subnormals may be suspect on some systems, we compute the logarithm of the smallest subnormal stepwise from that of the smallest normal. Here is what part of the initialization code looks like:

```
BIGW = LOGN(FP_T_MAX);   /* overflow  if n**w has w > BIGW */
SMALLW = LOGN(FP_T_MIN); /* underflow if n**w has w < SMALLW */
TINY = FP_T_MIN;
STORE(&TINY);
xmin = FP_T_MIN;

while ((xmin * BASE_INVERSE) > ZERO) /* we have subnormals */
{
    xmin *= BASE_INVERSE;
    SMALLW -= ONE;
}
```

TINY is used to generate a run-time underflow from the expression TINY * TINY. The variable is declared with the volatile qualifier, or storage is forced with the STORE() call, to prevent compile-time evaluation of the underflowing product.

At this point, we assume that w is in the range [SMALLW, BIGW], so underflow and overflow are excluded, or at least can only happen for w values that differ from those limits by only a few ulps. Values of w outside that range are handled by direct assignment of underflowed or overflowed values to the final result, avoiding further unnecessary computation.

The split of w can produce a positive w_2, but our polynomial fit requires w_2 in $(-1/C, 0]$, so we check for that case, and adjust accordingly:

```
iw1 = (int)TRUNC(C * w1);     /* integer overflow impossible */

if (w2 > ZERO)
{
    iw1++;
    w1 += C_INVERSE;          /* exact */
    w2 -= C_INVERSE;          /* approximate */
}

assert( (-C_INVERSE < w2) && (w2 <= ZERO) );  /* sanity check */
```

Table 14.5: Accuracy in decimal digits of rational polynomial fits of degree $\langle p/q \rangle$ to $S(w) = (n^w - 1)/w$. The range of w is $(-1/16, 0]$ when $n = 2$, and $(-1/10, 0]$ when $n = 10$.

	Degree								
n	**⟨2/2⟩**	**⟨3/3⟩**	**⟨4/4⟩**	**⟨5/5⟩**	**⟨7/7⟩**	**⟨9/9⟩**	**⟨11/11⟩**	**⟨13/13⟩**	**⟨14/14⟩**
2	8	13	19	25	39	53	68	83	93
10	9	14	19	24	35	46	57	69	76

The last bit of code that we need to describe in this section is that for the decomposition $w_1 = Km' - r' - p'/C$. When $\beta = 2$ or 10, we have $K = 1$ and $r' = 0$, so the code is simple, and the computations are exact:

```
i = (iw1 < 0) ? 0 : 1;
m_prime = (int)TRUNC(w1) + i;   /* in [SMALLW + i, BIGW + i] */
p_prime = (int)((long int)C * (long int)m_prime -
                (long int)iw1); /* in [0,C] */
```

Notice that higher-precision computation is essential for `p_prime`, even though its final value is small. The `i` term eliminates the need for separate handling of positive and negative w_1 values.

Otherwise, when $K > 1$, we need slightly more complex code:

```
i = (iw1 < 0) ? 0 : 1;
N = (int)TRUNC(w1) + i;
m_prime = N / K + i;            /* in [SMALLW/K + i,BIGW/K + i] */
p_prime = (int)((long int)C * (long int)N - (long int)iw1);
                                /* in [0,C] */
r_prime = (int)((long int)K * (long int)m_prime - (long int)N);
                                /* in [0,K) */
```

14.12 Computing n^{w_2}

Because w_2 is known to be in the small interval $(-1/C, 0]$, the value of n^{w_2} is near one, and as with $\log_n(g/a)$, there is danger of significance loss in its computation. We eliminate the problem by instead computing the difference, $n^{w_2} - 1$, with an accurate polynomial approximation. The Taylor series expansion looks like this:

$$\begin{aligned} n^w - 1 &= \ln(w)w + (1/2!)(\ln(w)w)^2 + (1/3!)(\ln(w)w)^3 + \cdots \\ &= w(\ln(w) + (1/2!)(\ln(w))^2 w + (1/3!)(\ln(w))^3 w^2 + \cdots) \\ &= w\mathcal{S}(w). \end{aligned}$$

We therefore require a polynomial approximation to this function:

$$\mathcal{S}(w) = (n^w - 1)/w.$$

Table 14.5 shows the accuracy of fits to $\mathcal{S}(w)$.

We can mitigate the effect of wobbling precision in computing $n^{w_2} = 1 + w_2 S(w_2)$ by combining the final two factors in the computation of the power function:

$$\begin{aligned} n^{-p'/C} n^{w_2} &= A_1[p'] n^{w_2} \\ &= A_1[p'](1 + w_2 S(w_2)) \\ &= A_1[p'] + A_1[p'] w_2 S(w_2), && \textit{default computation} \\ &= \mathrm{fma}(A_1[p'], w_2 S(w_2), A_1[p']), && \textit{improved computation.} \end{aligned}$$

Because w_2 is always negative, it might appear that there is possible subtraction loss here. For the nondecimal case, $2^w - 1$ lies in $(-0.0424, 0]$, and from the rule in Section 4.19 on page 89, subtraction loss is not possible in implicitly forming $1 + (2^w - 1)$.

For the decimal case, $10^w - 1$ lies in $(-0.206, 0]$, and the implicit computation of $1 - (10^w - 1)$ potentially loses *one* decimal digit. The solution is to either compute $n^{w_2} - 1$ in higher precision, or to increase C (see **Section 14.13**), or to use the fused multiply-add operation. For the mathcw library, we choose the latter. Most systems are likely to have only software implementations of decimal floating-point arithmetic, and the `fma()` function might even be faster than calls to separate functions for the multiply and add operations, as it is in the mathcw library. IBM z-Series systems from about 2005 have hardware support for decimal floating-point arithmetic, although fused multiply-add instructions are provided only for the 32-bit and 64-bit formats. The IBM PowerPC chips introduced in 2007 have similar hardware support.

14.13 The choice of q

In **Section 14.5** on page 421, we defined $C = 10^q$ for a decimal base, and $C = 16^q$ for nondecimal bases, and noted that Cody and Waite only considered the case $q = 1$. We need $C + 1$ entries in each of the tables $A_1[\,]$ and $A_2[\,]$, and we observed in **Section 14.8** on page 426 and **Section 14.12** on the previous page that larger C values have the virtue of reducing the interval of approximation for $\log_n(g/a)$ and for n^{w_2}, allowing use of smaller and faster rational polynomials.

There are no loops that depend on C, apart from the table search needed to find $A[p]$. We showed in **Section 14.6** on page 423 that only four or five comparisons are needed when $q = 1$. Using binary search for larger tables keeps the number of comparisons small: 14 for $q = 4$ in the decimal case, and 12 for $q = 3$ in the nondecimal case.

There is another reason, however, for picking $q > 1$. The `usplit()` and `wsplit()` operations that produce u_1, u_2, w_1, and w_2 move a base-C digit from the low part to the high part when q is increased by one. That has two effects:

- The accuracy of u_2 is largely independent of C because our polynomial approximations for $\log_n(g/a)$ are tailored to produce full accuracy on the interval determined by the choice C. There may, of course, be a slight improvement in accuracy in the subsequent computations $(2/\ln(n))s + s^3 R_n(s^2)$ and $\zeta + \zeta^3 \mathcal{R}_n(\zeta^2)$ because the results in **Table 14.3** on page 427 show that s^2 and ζ^2 drop by about two orders of magnitude for a unit increase of q, effectively adding about two decimal digits of precision.

- The bigger, and more important, effect, however, is that the product yu_2 is made smaller by a factor of 10 or 16, with the result that the effect of the last digit of u_2 on w_2 moves right by one decimal digit, or four bits, improving the accuracy of w_2. The improvement on accuracy is dramatic, as shown in **Table 14.6** on the facing page. The corresponding memory requirements are given in **Table 14.7**.

As a result of those measurements, the `pxy.h` file selects default table sizes to keep the average loss well below 1.00 bit or digit on modern systems, but the installer can override the defaults by compile-time definition of special symbols. The defaults always have $q > 1$, and they ensure that the accuracy of our power functions is often better than that of power functions in vendor-provided or other libraries, and competitive with the accuracy of the other elementary functions in the mathcw library.

Our use of $q > 1$ is thus an essential improvement over the original Cody/Waite algorithm, and our analysis shows that the only way to reduce q while preserving accuracy is to use almost triple-precision software arithmetic for the computation of w. However, ELEFUNT testing shows that it is still advantageous to use larger table sizes, even for decimal floating-point arithmetic, where our code for `wsplit()` produces always-correct w_2 values.

14.14 Testing the power function

Our tests of the power function are an extended version of the ELEFUNT tests, which check four important identities with suitably purified arguments selected from logarithmic distributions over the indicated argument ranges:

- Compare x^y with x for $y = 1$ and random x in $[1/\beta, 1]$. That test finds no errors if the case $y = 1$ is treated specially, as it is in the mathcw library, but is not in the original Cody/Waite algorithm. Because y is small and exactly representable, there is no error magnification. We should have $w_1 = u_1$ and $w_2 = u_2$, and inaccuracies in the normal decomposition of w should be absent. Any small errors found therefore reflect inaccuracies in the computation of $u_2 = \log_n(g/a)$.

Table 14.6: Effect of q on power-function accuracy, from ELEFUNT measurements for the functions `powl()` and `powdl()`.

		Bit or digit loss: average(worst)			
		q			
CPU	β	1	2	3	4
Alpha	2	0.40 (3.10)	0.00 (0.99)	0.00 (0.93)	n/a
AMD64	2	4.28 (6.83)	0.33 (2.86)	0.00 (0.99)	n/a
IA-32	2	4.28 (6.83)	0.33 (2.86)	0.00 (0.99)	n/a
MIPS	2	1.12 (3.84)	0.00 (1.00)	0.00 (0.96)	n/a
PowerPC	2	0.40 (3.10)	0.00 (0.99)	0.00 (0.93)	n/a
SPARC	2	4.04 (6.47)	0.31 (3.04)	0.00 (1.00)	n/a
AMD64	10	2.41 (3.14)	1.48 (2.24)	0.53 (1.32)	0.00 (1.00)

Table 14.7: Effect of q on power-function memory size for the private internal functions `_pxyl()` and `_pxydl()` that compute x^y for finite arguments when $x > 0$. The sizes include compiler-generated symbol tables and the `static` functions `binsearch()`, `find_a()`, `fmul2()`, `split()`, `usplit()`, and `wsplit()`.

		Code and data (KB)			
		q			
CPU	β	1	2	3	4
AMD64	2	5	12	132	n/a
Alpha	2	5	9	69	n/a
IA-32	2	4	10	100	n/a
MIPS	2	8	15	137	n/a
PowerPC	2	3	7	67	n/a
SPARC	2	20	28	148	n/a
AMD64	10	8	11	39	320

- Compare $(x \times x)^{3/2}$ with x^3 and random x in $[1/\beta, 1]$, purified so that the product $x \times x$ is exactly representable.

  The purification is easily done as follows. If t is the working precision, then given a random x, compute $\sigma = x\beta^{\lfloor (t+1)/2 \rfloor}$. That is an exact scaling as long as we precompute the power of β by direct multiplication, or else use the exact `ldexp()` or `setxp()` primitives. Then replace x with $(x + \sigma) - \sigma$. As usual, obey the parentheses, and force storage of intermediate results to thwart machines with higher-precision registers.

  There is then at most one rounding error in the computation of x^3, but that value should be the one closest to the exact cube in a *round-to-nearest* floating-point system, in which case it can be regarded as exact to working precision.

  Because $y = 3/2$, it is less likely to be detected by special-case code, and computation of w will require a decomposition where the major source of error is again in the computation of $u_2 = \log_n(g/a)$.

- Make the same test as the preceding one, but this time, select random x values from the interval $[1, (\text{maximum normal})^{1/3}]$. The larger x values mean larger u_1 and w, increasing the error magnification.

- Compare $(x \times x)^{y/2}$ with x^y, with random x on $[1/100, 10]$, and random y on $[-Y, +Y]$, where Y is the largest number that avoids both underflow and overflow in x^Y. Purify x as in the first test, so that x^2 is exact. Purify y so that $y/2$ is exact like this: compute $a = y/2$ and $b = (a - y) + y$ (obey parentheses and force storage of $(a - y)$), and then set $y = b + b$.

  That is the most demanding test, because y is large, and unlikely to have multiple trailing zeros. It tests the accuracy of the decomposition $w_1 + w_2 = y(u_1 + u_2)$ more than the earlier tests, because the trailing digits of both y and u_2 affect the final digits of w_2, and because y can be large, error magnification is also large.

We augment the four ELEFUNT tests in response to the observation of the effect of a suboptimal choice of g (see Section 14.6 on page 424) on the computation of powers of 1 and β. Those powers are exactly representable in all floating-point systems, but are likely to be slightly inaccurate if the computation of $\log_n(g/a)$ does not handle them carefully. We therefore make these extra tests:

- For all possible exponents n in the floating-point representation, compare $(-1)^n$ with $+1$ (n even) or -1 (n odd), and 1^n with 1.
- For all possible exponents n in the floating-point representation, compare $(-\beta)^n$, $(\beta)^n$, $(-1/\beta)^n$, and $(1/\beta)^n$ with exact n-term products.

If the power function handles $x = \pm 1$ and $x = \pm\beta$ separately when y is a whole-number power, the first, and part of the second, test should find no errors. The powers of $x = 1/\beta$ are unlikely to receive special handling, and should force a normal path through the power function, even though the result can be represented exactly.

14.15 Retrospective on the power function

We devoted an entire chapter to the power function, which is the most difficult elementary function in the mathcw library repertoire, primarily because its accurate computation needs almost triple the working precision, and few programming languages provide floating-point arithmetic with a precision that can be chosen arbitrarily at run time.

We therefore have to simulate the higher precision by representing several different values in the computation as the sum of two components, where the high part is exact, and where the exact sum would require more than working precision. Decomposing a computation into a sum or product of parts, some of which can be computed exactly, is a valuable technique that deserves to be more widely used.

The power function requires two sets of polynomial approximations, one for $\log_n(g/a)$, and another for $n^w - 1$, and we must be able to compute the first of them in higher than working precision.

Finally, we demonstrated that generalization of the algorithm to support larger values of C permits lower-order, and thus faster, polynomial approximations for $\log_n(g/a)$, at the expense of storage of larger tables $A_1[\,]$ and $A_2[\,]$. Indeed, several papers on the computation of elementary functions that have been published since the influential book by Cody and Waite improve both speed and accuracy by trading storage for performance. Serendipitously, by increasing C, we are then able to reach a level of accuracy competitive with other elementary functions in the mathcw library.

15 Complex arithmetic primitives

GAUSS[1] ESTABLISHED THE MODERN THEORY OF NUMBERS,
GAVE THE FIRST CLEAR EXPOSITION OF COMPLEX NUMBERS,
AND INVESTIGATED THE FUNCTIONS OF COMPLEX VARIABLES.

— *The Columbia Encyclopedia* (2001).

Apart from Fortran and symbolic-algebra systems, few programming languages support complex arithmetic. The 1998 ISO C++ Standard adds a standard header file, `<complex>`, to provide a `complex` data-type template, and prototypes of complex versions of a dozen or so elementary functions. The 1999 ISO C Standard goes further, offering a standard header file, `<complex.h>`, and a new language keyword, `_Complex`. When `<complex.h>` is included, the name `complex` can be used as a synonym for the new keyword, allowing declaration of objects of type `float complex`, `double complex`, and `long double complex`. The header file declares nearly two dozen function prototypes for each of those three data types.

The C99 Standard also specifies a built-in pure imaginary data-type modifier, called `_Imaginary`, and a new keyword, `_Imaginary_I`, representing the constant $i = \sqrt{-1}$ as one solution of the equation $i^2 = -1$. The `<complex.h>` header file defines macro synonyms `imaginary` for the type and `I` for the constant. However, the Standard makes the imaginary type optional: it is available if, and only if, the macros `imaginary` and `_Imaginary_I` are defined. If the imaginary type is not supported, then `I` is defined to be the new keyword `_Complex_I`, which has the value of the imaginary unit, and type of `float _Complex`. Because of its optional nature, we avoid use in the `mathcw` library of the imaginary type modifier; the complex data types are sufficient for our needs.

The IEEE *Portable Operating System Interface (POSIX) Standard* [IEEE01] requires the same complex arithmetic support as C99, but defers to that Standard in the event of differences.

Annex G of the C99 Standard contains this remark:

> *A complex or imaginary value with at least one infinite part is regarded as an* infinity *(even if its other part is a NaN). A complex or imaginary value is a* finite number *if each of its parts is a finite number (neither infinite nor NaN). A complex or imaginary value is a* zero *if each of its parts is a zero.*

That first statement is surprising, because it implies different computational rules that might ignore, or fail to propagate, the NaN. That is contrary to the usual behavior of NaNs in the operations and functions of real arithmetic, and in this author's view, likely to be more confusing than useful.

The C99 Standard requires implementations of the complex data type to satisfy this condition [C99, §6.2.5, ¶13, p. 34]:

> *Each complex type has the same representation and alignment requirements as an array type containing exactly two elements of the corresponding real type; the first element is equal to the real part, and the second element to the imaginary part, of the complex number.*

The mandated storage layout follows the practice in Fortran, and means that, in C99, a complex argument passed by address can be received as a pointer to a two-element array. That simplifies interlanguage communication.

The C89 Standard adds support for `struct` return values, and the C99 Standard allows complex return values from functions. However, the C language has never permitted arrays to be returned from functions, nor does it support operator overloading. The only solution seems to be implementation of the type as a `struct` instead of an array. Otherwise, it is *impossible* to retrofit full support for complex types into older C implementations without modifying the compiler itself.

[1] The German scientist Carl Friedrich Gauss (1777–1855) was one of the most influential mathematicians in history, with important contributions in algebra, astronomy, complex analysis, geodesy, geometry, magnetism, number theory, numerical quadrature, probability, statistics, and telegraphy. The unit of magnetic induction is named after him. He was a child prodigy, a mental calculating savant, and a polyglot. He also invented the *heliotrope*, a device for using mirrors to reflect sunlight over long distances in land surveying.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_15

For that reason, the complex-arithmetic primitives that we describe in this chapter take a different approach: when a complex function result is required, it appears as the *first* argument. The function is then declared to be of type void, and the code guarantees that the result argument can overlap with an input argument, because that often proves convenient in a chain of complex-arithmetic operations.

In addition, to avoid collisions with the C99 function names, we use prefix letters cx instead of the standard prefix letter c. Thus, our cxabs() corresponds to C99's cabs().

Our implementation of support for complex arithmetic then provides a *completely portable* set of functions. When the compiler can support the C99-style extensions, as indicated by the compile-time definition of the standard macro __STDC_IEC_559_COMPLEX__, it is easy to provide the new functions as wrappers around calls to our own set. Although the C99 Standard requires the enabling macro to be defined by the language, at the time of writing this, some implementations that claim at least partial support of that Standard require <complex.h> to be included to have the symbol defined. That is backwards, because the symbol should indicate the availability of the header file, not the reverse. Our code caters to the problem by using instead a private compile-time symbol, HAVE_COMPLEX, to enable code that references the complex data type. The <complex.h> header file is then expected to be available, and the mathcw library version of that file, complexcw.h, includes the standard file, and then declares prototypes for our additional functions.

At the time of writing this, the GNU gcc and Intel icc compilers do not permit combining the complex modifier with decimal floating-point types. Compilation of the complex decimal functions is therefore suppressed by omitting them from the decimal source-file macros in the mathcw package Makefile.

In the following sections, after we define some macros and data types, we present in alphabetical order the dozen or so primitives for portable complex arithmetic, along with their C99 counterparts.

15.1 Support macros and type definitions

To simplify, and hide, the representation of complex-as-real data, we define in the header file cxcw.h public types that correspond to two-element arrays of each of the supported floating-point types:

```
typedef float                  cx_float                 [2];
typedef double                 cx_double                [2];
typedef decimal_float          cx_decimal_float         [2];
typedef decimal_double         cx_decimal_double        [2];
typedef long double            cx_long_double           [2];
typedef __float80              cx_float80               [2];
typedef __float128             cx_float128              [2];
typedef long_long_double       cx_long_long_double      [2];
typedef decimal_long_double    cx_decimal_long_double   [2];
typedef decimal_long_long_double cx_decimal_long_long_double [2];
```

Unlike Fortran, which has built-in functions for creating a complex number from two real numbers, C99 instead uses expressions involving the imaginary value, I. That value is defined in <complex.h> as a synonym for either the new keyword _Imaginary_I, if the compiler supports pure imaginary types, or else the new keyword _Complex_I. To clarify the creation of complex numbers from their component parts, the header file complexcw.h defines the constructor macro

```
#define CMPLX(x,y)      ((x) + (y) * I)
```

that we use in the remainder of this chapter. That header file also defines the macro

```
#define CTOCX_(result,z) CXSET_(result, CREAL(z), CIMAG(z))
```

for converting from native complex to the complex-as-real type fp_cx_t.

Our C99 complex functions use a new type, fp_c_t, for *floating-point complex* data. It is defined with a typedef statement to one of the standard built-in complex types.

The header file cxcw.h provides a few public macros for inline access to the components of complex-as-real data objects, and their conversion to native complex data:

```
#define CXCOPY_(z,w)    CXSET_(z, CXREAL_(w), CXIMAG_(w))
#define CXIMAG_(z)      (z)[1]
#define CXREAL_(z)      (z)[0]
#define CXSET_(z,x,y)   (CXREAL_(z) = (x), CXIMAG_(z) = (y))
#define CXTOC_(z)       CMPLX(CXREAL_(z), CXIMAG_(z))
```

We use them extensively to eliminate explicit array subscripting in all of the complex functions.

Experiments on several platforms with multiple compilers show that code-generation technology for complex arithmetic is immature. The simple constructor function

```
double complex
cmplx(double x, double y)
{
    return (x + y * I);
}
```

can compile into a dozen or more instructions, including a useless multiplication by one in the imaginary part. With high optimization levels, some compilers are able to reduce that function to a single `return` instruction, when the input argument and the output result occupy the same registers.

For the common case of conversion of `fp_cx_t` data to `fp_c_t` values, we can use the storage-order mandate to reimplement the conversion macro like this:

```
#define CXTOC_(z)       (*(fp_c_t *)(&(CXREAL_(z))))
```

Tests show that our new version eliminates the useless multiply, and produces shorter code.

On most architectures, access to the imaginary or real parts requires only a single load instruction, and the assignments in `CXSET_()` can sometimes be optimized to two store instructions, and on some platforms, to just one double-word store instruction. The conversion to native complex data by `CXTOC_()` can often be reduced to two store or register-move instructions, or one double-word store instruction.

Because most of the functions defined in this chapter are short and time critical, we use the underscore-terminated macros to get inline code. For slower code that requires function calls and allows debugger breakpoints, any of the macros that end with an underscore can be replaced by their companions without the underscore.

15.2 Complex absolute value

A complex number $z = x + yi$ can be represented as a point in the plane at position (x, y), as illustrated in **Figure 15.1** on the following page. The notation $x + yi$ is called the *Cartesian form*. Using simple trigonometry, we can write it in *polar form* as

$$\begin{aligned} z &= r\cos(\theta) + r\sin(\theta)i, \\ &= r\exp(\theta i), && \textit{polar form,} \end{aligned}$$

where r is the distance of the point (x, y) from the origin $(0, 0)$, the angle θ is measured *counterclockwise* from the positive x axis, and the exponential is obtained from the famous *Euler formula* for the imaginary exponential:

$$\boldsymbol{\exp(\theta i) = \cos(\theta) + \sin(\theta)i,} \qquad \textit{Euler formula.}$$

The combination of trigonometric functions on the right-hand side is so common that many textbooks give it the name $\operatorname{cis}(\theta)$. In this book, we prefer the exponential form on the left-hand side.

The substitution $\theta = \pi$ in the Euler formula produces

$$\begin{aligned} \exp(\pi i) &= \cos(\pi) + \sin(\pi)i \\ &= -1 + 0i \\ &= -1. \end{aligned}$$

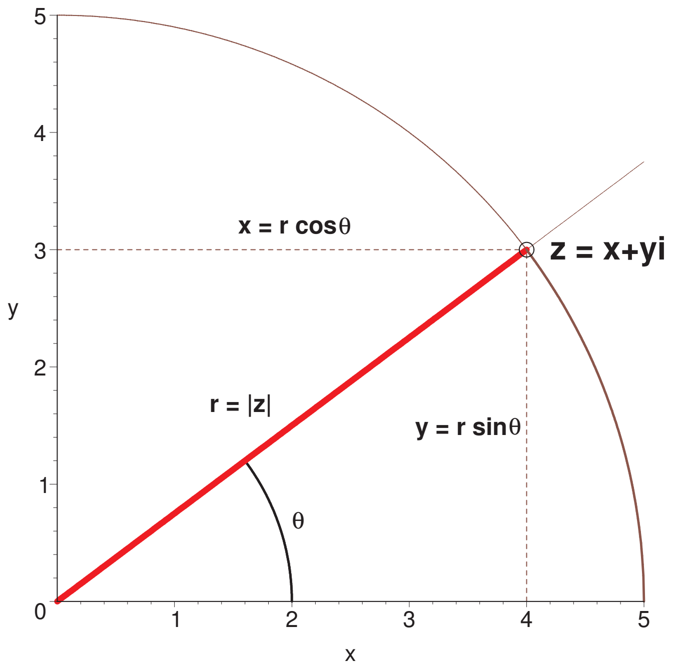


Figure 15.1: Cartesian and polar forms of a point in the complex plane. The angle θ is positive when measured counterclockwise from the positive x axis.

That can be rearranged to produce the *Euler identity* that connects the two most important transcendental numbers in mathematics with the complex imaginary unit and the digits of the binary number system:

$$e^{\pi i} + 1 = 0, \qquad \textit{Euler identity.}$$

The absolute value of a complex number $x + yi$ is the *Euclidean distance* of the point (x, y) from the origin: $|z| = r = \sqrt{x^2 + y^2}$. Because the standard hypot() function already provides that computation, the C99 Standard mandates use of that function, or equivalent code:

```
fp_t
CXABS(const fp_cx_t z)
{   /* complex absolute value: return abs(z) */
    /* WARNING: this function can overflow for component magnitudes
       larger than FP_T_MAX / sqrt(2): rescale carefully! */

    return (HYPOT(CXREAL_(z), CXIMAG_(z)));
}
```

No special handling of Infinity and NaN components in the complex number is needed, because HYPOT() does that work for us.

The commented warning about possible overflow is significant, because the real function family ABS() is *never* subject to overflow. Whenever an algorithm requires the complex absolute value, it is essential to provide suitable scaling to prevent premature overflow.

The code for the C99 complex functions, and their function prototypes, is bracketed with preprocessor conditionals that emit code only if HAVE_COMPLEX is defined. However, we omit those conditionals in the code presented in this book.

With a few exceptions where efficiency is imperative, and the code is simple, we implement the C99-style functions in terms of our portable functions, as here for the complex absolute value:

```
fp_t
CABS(fp_c_t z)
{   /* complex absolute value: return abs(z) */
    /* WARNING: this function can overflow for component magnitudes
       larger than FP_T_MAX / sqrt(2): rescale carefully! */
    fp_cx_t zz;

    CTOCX_(zz, z);

    return (CXABS(zz));
}
```

15.3 Complex addition

The addition operation for complex numbers is simple: just sum the real and imaginary parts separately:

```
void
CXADD(fp_cx_t result, const fp_cx_t x, const fp_cx_t y)
{  /* complex addition: result = x + y */
    CXSET_(result, CXREAL_(x) + CXREAL_(y), CXIMAG_(x) + CXIMAG_(y));
}
```

The code works correctly even when `result` is the same as either, or both, of `x` or `y`.

The C99-style companion function for complex addition is not likely to be used, but we include it for completeness, because it facilitates machine-assisted translation to C99 code from code that uses the portable complex arithmetic primitives:

```
fp_c_t
CADD(fp_c_t x, fp_c_t y)
{   /* complex addition: return x + y */
    return (x + y);
}
```

15.4 Complex argument

The *argument*, or *phase*, of a complex number $x + iy$ is the angle in radians between the positive x axis and a line from the origin to the point (x, y). That is exactly what the two-argument arc tangent function computes, so the code is easy:

```
fp_t
CXARG(const fp_cx_t z)
{   /* complex argument: return argument t of z = r * exp(i*t) */
    return (ATAN2(CXIMAG_(z), CXREAL_(z)));
}
```

The C99 function is a simple wrapper that calls the portable function:

```
fp_t
CARG(fp_c_t z)
{   /* complex argument: return argument (angle in radians) of z */
    fp_cx_t zz;

    CTOCX_(zz, z);

    return (CXARG(zz));
}
```

From the polar form of complex numbers, it is easy to see that the argument of a product is the sum of the arguments:

$$\arg(w \times z) = \arg(w) + \arg(z).$$

The argument of a quotient is the difference of the arguments, and the argument of a reciprocal is the negation of the argument:

$$\begin{aligned}\arg(w/z) &= \arg(w) - \arg(z),\\ \arg(1/z) &= -\arg(z).\end{aligned}$$

15.5 Complex conjugate

The *conjugate* of a complex number $x + iy$ is just $x - iy$, so the code is easy:

```
void
CXCONJ(fp_cx_t result, const fp_cx_t z)
{   /* complex conjugate: result = complex_conjugate(z) */
    CXSET_(result, CXREAL_(z), -CXIMAG_(z));
}
```

We implement the C99 function with inline code:

```
fp_c_t
CONJ(fp_c_t z)
{   /* complex conjugate: return complex_conjugate(z) */
    return (CMPLX(CREAL(z), -CIMAG(z)));
}
```

15.6 Complex conjugation symmetry

There are two common notations for complex conjugation of a variable or expression in mathematical texts: a superscript star, $z^\star$, or an overbar, $\bar{z}$. In this chapter, we use the star notation because it is easier to see.

The conjugate of a complex number is the reflection of its point on the complex plane across the real axis. In the Cartesian form of complex numbers, we have

$$\begin{aligned} w &= u + vi, & z &= x + yi, && \textit{for real } u, v, x, \textit{ and } y,\\ w^\star &= u - vi, & z^\star &= x - yi, && \textit{complex conjugate}.\end{aligned}$$

In the equivalent polar form, we have

$$\begin{aligned} z &= r\exp(\theta i), && \textit{for real } r \textit{ and } \theta,\\ z^\star &= r\exp(-\theta i), && \textit{by reflection across the real axis}.\end{aligned}$$

The operation of complex conjugation appears in an important symmetry relation for many complex functions of a single variable, $f(z)$:

$$f(z^\star) = \big(f(z)\big)^\star, \qquad \textit{symmetry under complex conjugation}.$$

To understand the origin of that special symmetry of some complex functions, we look first at how complex conjugation behaves with the low-level operations of complex arithmetic:

$$\begin{aligned}(-z)^\star &= (-x - yi)^\star\\ &= (-x + yi)\\ &= -(x - yi)\end{aligned}$$

$$\begin{aligned}
&= -(z^\star), && \textit{symmetry under negation},\\
(w+z)^\star &= \bigl((u+x)+(v+y)i\bigr)^\star \\
&= (u+x)-(v+y)i \\
&= (u-vi)+(x-yi) \\
&= w^\star + z^\star, && \textit{symmetry under addition},\\
(w\times z)^\star &= \bigl((ux-vy)+(uy+vx)i\bigr)^\star \\
&= (ux-vy)-(uy+vx)i \\
&= (u-iv)(x-iy) \\
&= w^\star \times v^\star, && \textit{symmetry under multiplication},\\
(1/z)^\star &= \bigl(1/(x+yi)\bigr)^\star \\
&= \bigl((x-yi)/(x^2+y^2)\bigr)^\star \\
&= (x+yi)/(x^2+y^2) \\
&= 1/(x-yi) \\
&= 1/(z^\star), && \textit{symmetry under reciprocation},\\
(w/z)^\star &= w^\star/z^\star, && \textit{symmetry under division}.
\end{aligned}$$

The last relation follows by combining the symmetry rules for multiplication and reciprocation, but can also be derived by tedious expansion and rearrangement of the numerator and denominator. Thus, the operation of complex conjugation distributes over negations, sums, differences, products, and quotients. Because it holds for products, we conclude that it also holds for integer powers:

$$(z^n)^\star = (z^\star)^n, \qquad \textit{for } n = 0, \pm 1, \pm 2, \ldots.$$

Because it applies for products and sums, it is also valid for polynomials and convergent series with *real* coefficients, p_k:

$$\begin{aligned}
\bigl(\mathcal{P}(z)\bigr)^\star &= (p_0 + p_1 z + p_2 z^2 + p_3 z^3 + \cdots)^\star \\
&= (p_0)^\star + (p_1 z)^\star + (p_2 z^2)^\star + (p_3 z^3)^\star + \cdots \\
&= p_0^\star + p_1^\star z^\star + p_2^\star (z^\star)^2 + p_3^\star (z^\star)^3 + \cdots \\
&= p_0 + p_1 z^\star + p_2 (z^\star)^2 + p_3 (z^\star)^3 + \cdots \\
&= \mathcal{P}(z^\star).
\end{aligned}$$

If a function with a single complex argument has a convergent Taylor-series expansion about some complex point z_0 given by

$$f(z) = c_0 + c_1(z-z_0) + c_2(z-z_0)^2 + c_3(z-z_0)^3 + \cdots,$$

then as long as the coefficients c_k are real, we have the conjugation symmetry relation $\bigl(f(z)\bigr)^\star = f(z^\star)$. The elementary functions that we treat in this book have that property, and it is of utmost importance to design computational algorithms to ensure that the symmetry property holds for all complex arguments z, whether finite or infinite. However, NaN arguments usually require separate consideration.

The product of a complex number with its conjugate is the square of its absolute value. We can show that in both Cartesian form and in polar form:

$$\begin{aligned}
zz^\star &= (x+yi)\times(x-yi) & \qquad zz^\star &= r\exp(\theta i)\times r\exp(-\theta i) \\
&= x^2 - xyi + yxi + y^2 & &= r^2\exp(\theta i - \theta i) \\
&= x^2 + y^2 & &= r^2 \\
&= |z|^2, & &= |z|^2.
\end{aligned}$$

Rearranging the final identity produces a simple formula for the complex reciprocal that is helpful for converting complex division into complex multiplication and a few real operations:

$$\frac{1}{z} = \frac{z^\star}{|z|^2}.$$

From the Cartesian form, these further relations are evident:

$$\begin{aligned} z &= z^\star, && \textit{if, and only if, } z \textit{ is real}, \\ \mathrm{real}(z) &= \tfrac{1}{2}(z + z^\star), \\ \mathrm{imag}(z) &= -\tfrac{1}{2}(z - z^\star)i. \end{aligned}$$

15.7 Complex conversion

The type-conversion macros defined on page 442 make it easy to provide companion functions:

```
void
CTOCX(fp_cx_t result, fp_c_t z)
{   /* convert native complex z to complex-as-real */
    CTOCX_(result, z);
}


fp_c_t
CXTOC(fp_cx_t z)
{   /* convert complex-as-real z to native complex */
    return (CXTOC_(z));
}
```

Those macros and functions reduce the need to reference the imaginary and real parts separately, shorten code that uses them, and make the type conversions explicit.

15.8 Complex copy

Because C does not support array assignment, we need a primitive for the job, so that user code can avoid referring to array subscripts, or individual components:

```
void
CXCOPY(fp_cx_t result, const fp_cx_t z)
{   /* complex copy: result = z */
    CXCOPY_(result, z);
}
```

The C99-style companion is unlikely to be needed, except for machine-assisted code translation, but we provide it for completeness:

```
fp_c_t
CCOPY(fp_c_t z)
{   /* complex copy: return z */
    return (z);
}
```

15.9 Complex division: C99 style

The most difficult operation in the complex primitives is division. If $x = a + ib$ and $y = c + id$, then complex division is defined by introducing a common factor in the numerator and denominator that reduces the complex denominator to a real number that can then divide each component:

$$\begin{aligned} x/y &= (a+bi)/(c+di) \\ &= \big((a+bi)(c-di)\big)/\big((c+di)(c-di)\big) \\ &= \big((ac+bd)+(bc-ad)i\big)/(c^2+d^2) \\ &= \big((ac+bd)/(c^2+d^2)\big)+\big((bc-ad)/(c^2+d^2)\big)i. \end{aligned}$$

In the last result, we can readily identify two serious problems for implementation with computer arithmetic of finite precision and range: significance loss in the additions and subtractions in the numerator, and premature overflow and underflow in both the numerator and the denominator.

There are more difficulties lurking, however. We also have to consider the possibility that one or more of the four components a, b, c, and d are Infinity, a NaN, or zero.

Infinities introduce problems, because IEEE 754 arithmetic requires that subtraction of like-signed Infinities, and division of Infinities, produce a NaN. Thus, even though we expect mathematically that $(1+i)/(\infty+i\infty)$ should evaluate to zero, the IEEE 754 rules applied to the definition of division produce a NaN result:

$$\begin{aligned} (1+i)/(\infty+i\infty) &= ((\infty+\infty)+(\infty-\infty)i)/(\infty^2+\infty^2) \\ &= (\infty+\mathrm{NaN}i)/\infty \\ &= \mathrm{NaN}+\mathrm{NaN}i. \end{aligned}$$

Division by zero is also problematic. Consider a finite numerator with positive parts. We then have three different results, depending on whether we divide by a complex zero, a real zero, or an imaginary zero:

$$\begin{aligned} (a+bi)/(0+0i) &= 0/0+(0/0)i \\ &= \mathrm{NaN}+\mathrm{NaN}i, \\ (a+bi)/0 &= a/0+(b/0)i \\ &= \infty+\infty i, \\ (a+bi)/(0i) &= b/0-(a/0)i \\ &= \infty-\infty i. \end{aligned}$$

Thus, a complex-division routine that checks for zero real or imaginary parts to simplify the task to two real divides gets different answers from one that simply applies the expansion of x/y given at the start of this section.

Experiments with complex division in C99 and Fortran on various platforms show that their handling of Infinity is inconsistent. The ISO standards for those languages offer no guidance beyond a recommended algorithm for complex division in an informative annex of the C99 Standard [C99, §G.5.1, p. 469]. However, that annex notes in its introduction:

> *This annex supplements annex F to specify complex arithmetic for compatibility with IEC 60559 real floating-point arithmetic. Although these specifications have been carefully designed, there is little existing practice to validate the design decisions. Therefore, these specifications are not normative, but should be viewed more as recommended practice.*

The possibility of Infinity and NaN components could require extensive special casing in the division algorithm, as well as multiple tests for such components. That in turn makes the algorithm slower for all operands, even those that require no special handling.

To eliminate most of the overhead of special handling, the algorithm suggested in the C99 Standard follows the policy of *compute first, and handle exceptional cases later*, which the IEEE 754 nonstop model of computation easily supports. Older architectures may require additional coding, however, to achieve documented and predictable results for complex division.

Here is our implementation of the C99 algorithm for complex division:

```
void
CXDIV(fp_cx_t result, const fp_cx_t x, const fp_cx_t y)
{   /* complex division: result = x / y */
    fp_t a, b, c, d, logb_y, denom;
    fp_pair_t ac_bd, bc_ad;
    volatile fp_t e, f;
    int ilogb_y;

    ilogb_y = 0;
    a = CXREAL_(x);
    b = CXIMAG_(x);
    c = CXREAL_(y);
    d = CXIMAG_(y);

    logb_y = LOGB(FMAX(QABS(c), QABS(d)));

    if (ISFINITE(logb_y))
    {
        ilogb_y = (int)logb_y;
        c = SCALBN(c, -ilogb_y);
        d = SCALBN(d, -ilogb_y);
    }

    denom = c * c + d * d;
    fast_pprosum(ac_bd, a, c, b, d);
    fast_pprosum(bc_ad, b, c, -a, d);

    e = SCALBN(PEVAL(ac_bd) / denom, -ilogb_y);
    STORE(&e);
    f = SCALBN(PEVAL(bc_ad) / denom, -ilogb_y);
    STORE(&f);

    if (ISNAN(e) && ISNAN(f))
    {
        fp_t inf;

        /* Recover infinities and zeros that computed as NaN +
           I*NaN. The only cases are nonzero/zero, infinite/finite,
           and finite/infinite */
        if ((denom == ZERO) && (!ISNAN(a) || !ISNAN(b)))
        {
            inf = INFTY();
            e = COPYSIGN(inf, c) * a;
            f = COPYSIGN(inf, c) * b;
        }
        else if ((ISINF(a) || ISINF(b)) && ISFINITE(c) && ISFINITE(d))
        {
            inf = INFTY();
            a = COPYSIGN(ISINF(a) ? ONE : ZERO, a);
            b = COPYSIGN(ISINF(b) ? ONE : ZERO, b);
            e = inf * (a * c + b * d);
            f = inf * (b * c - a * d);
        }
        else if (ISINF(logb_y) && ISFINITE(a) && ISFINITE(b))
        {
            c = COPYSIGN(ISINF(c) ? ONE : ZERO, c);
            d = COPYSIGN(ISINF(d) ? ONE : ZERO, d);
            e = ZERO;
            f = ZERO;
            e *= (a * c + b * d);
            f *= (b * c - a * d);
        }
    }

    CXSET_(result, e, f);
}
```

The complex division algorithm is complicated, and several subtle points are worth noting:

- Premature overflow and underflow are avoided by scaling the denominator and the numerator. Mathematically, that could be done by dividing each component by the larger magnitude, $|c|$ or $|d|$. However, that division introduces into the scaled components at least one rounding error each, and for older architectures, even more, because division was often less accurate than addition and multiplication. The C99 algorithm trades execution time for better accuracy by scaling by the power of the base near the larger of $|c|$ and $|d|$. That scaling is *exact*, so no additional rounding error is introduced. The C99 `LOGB()` and `SCALBN()` functions provide the needed tools, and we can use them on all systems because they are also members of the `mathcw` library.

- Even though division is more expensive than multiplication, the computation of the result components, e and f, uses two divisions, rather than precomputing the reciprocal of `denom` and then using two multiplications. Doing so would introduce additional rounding errors that are unnecessary if we pay the cost of an extra division.

- The biggest loss of accuracy in the division comes from the product sums $ab + bd$ and $bc - ad$. The algorithm recommended by the C99 Standard computes them directly, but we replace that computation by calls to the functions `PPROSUM()` and `PEVAL()` for enhanced accuracy. We comment more on that problem later in **Section 15.13** on page 455 and **Section 15.16** on page 458.

- If the result components are finite or Infinity, or just one of them is a NaN, no further computation is needed.

- Otherwise, both components are a NaN, and three separate cases of corrective action are required, each of which involves from two to six property checks with the `ISxxx()` functions.

C99 does not provide a function for complex division, because that operation is built-in. For completeness, we provide a C99-style function that uses the code in `CXDIV()`:

```
fp_c_t
CDIV(fp_c_t x, fp_c_t y)
{   /* complex division: return x / y */
    fp_cx_t xx, yy, result;

    CTOCX_(xx, x);
    CTOCX_(yy, y);
    CXDIV(result, xx, yy);

    return (CXTOC_(result));
}
```

15.10 Complex division: Smith style

The first published algorithm for complex division known to this author is Robert L. Smith's *ACM Algorithm 116* [Smi62], which addresses the overflow and underflow problems by scaling the denominator by the larger of its two components. If $|c| \geq |d|$, we rewrite the division given at the start of the previous section with two intermediate variables r and s like this:

$$x/y = (a + bi)/(c + di)$$

$$
\begin{aligned}
&= \big((ac+bd)/(c^2+d^2)\big) + \big((bc-ad)/(c^2+d^2)\big)i,\\
&= \big((a+bd/c)/(c+d^2/c)\big) + \big((b-ad/c)/(c+d^2/c)\big)i,\\
r &= d/c,\\
s &= dr+c,\\
x/y &= \big((a+br)/s\big) + \big((b-ar)/s\big)i.
\end{aligned}
$$

Otherwise, when $|c| < |d|$, similar steps produce

$$
\begin{aligned}
r &= c/d,\\
s &= cr+d,\\
x/y &= \big((ar+b)/s\big) + \big((br-a)/s\big)i,
\end{aligned}
$$

The total floating-point work is 2 `fabs()` operations, 1 compare, 3 adds, 3 divides, and 3 multiplies.

The inexact scaling contaminates both components of the result with an additional rounding error. With somewhat more work, that scaling can be made exact [LDB$^+$00, Appendix B, page 61], but we do not discuss it here because we can do even better.

The fused multiply-add operation was not invented until much later, but it clearly could be of use here, speeding the calculations, and largely eliminating subtraction loss.

15.11 Complex division: Stewart style

About two decades after *ACM Algorithm 116* [Smi62], G. W. Stewart revisited Smith's method and pointed out that premature overflow and underflow can be made less likely by rearranging the computation to control the size of intermediate products [Ste85]. The revised algorithm requires additional control logic, and looks like this when rewritten to use the same variables for real and imaginary parts as we have in `CXDIV()`:

```
#include <stdbool.h>

#define SWAP(x,y)       (temp = x, x = y, y = temp)

void
stewart_cxdiv(fp_cx_t result, const fp_cx_t z, const fp_cx_t w)
{   /* complex-as-real division: set result = z / w */
    fp_t a, b, c, d, e, f, s, t, temp;
    bool flip;

    a = CXREAL_(z);
    b = CXIMAG_(z);
    c = CXREAL_(w);
    d = CXIMAG_(w);
    flip = false;

    if (QABS(d) > QABS(c))
    {
        SWAP(c, d);
        SWAP(a, b);
        flip = true;
    }

    s = ONE / c;
    t = ONE / (c + d * (d * s));

    if (QABS(d) > QABS(s))
        SWAP(d, s);
    if (QABS(b) >= QABS(s))
        e = t * (a + s * (b * d));
    else if (QABS(b) >= QABS(d))
        e = t * (a + b * (s * d));
    else
        e = t * (a + d * (s * b));

    if (QABS(a) >= QABS(s))
        f = t * (b - s * (a * d));
    else if (QABS(a) >= QABS(d))
        f = t * (b - a * (s * d));
    else
        f = t * (b - d * (s * a));

    if (flip)
        f = -f;

    CXSET_(result, e, f);
}
```

In the first `if` block, Stewart exploits the complex-conjugation symmetry rule for division:

$$\begin{aligned}\big((a+bi)/(c+di)\big)^{\star} &= (a+bi)^{\star}/(c+di)^{\star} \\ &= (a-bi)/(c-di), && \textit{then multiply by } i/i, \\ &= (b+ai)/(d+ci).\end{aligned}$$

If necessary, the real and imaginary components are swapped to ensure that $|c| \geq |d|$, and the `flip` variable records that action. Subsequent operations then compute the conjugate of the desired result, and the sign is inverted in the final `if` statement.

Stewart's code requires parentheses to be obeyed, but as we recorded in **Section 4.4** on page 64, that was not true in C before the 1990 ISO Standard.

The floating-point operation count is 8 to 12 `QABS()` tests, 4 to 6 compares, 3 adds, 2 divides, 8 multiplies, and possibly 1 negation. Unless divides are exceptionally slow, Stewart's algorithm is likely to be somewhat slower than Smith's, but it has better numerical behavior at the extremes of the floating-point range.

Extensive tests of both Smith's and Stewart's algorithms on several platforms against more accurate code for complex division show that, with millions of random arguments, the relative error of the quotient lies below 3.0 ulps. However, it is certainly possible with specially chosen components to exhibit cases that suffer catastrophic subtraction loss.

Although we do not show the needed code, special handling of Infinity and NaN arguments is required, because both Smith's and Stewart's algorithms produce NaN results, instead of Infinity, for ∞/finite and zero for finite/∞.

15.12 Complex division: Priest style

Two decades more passed before complex division was again revisited. In a lengthy article with complicated numerical analysis [Pri04], Douglas Priest shows that the inexact scaling in the Smith [Smi62] and Stewart [Ste85] methods can be eliminated *without* the overhead of the `logb()` and `scalbn()` calls used in the C99 algorithm, provided that the programmer is willing to commit to a particular floating-point format, and grovel around in the bits of the floating-point representation. Priest's detailed examination shows that there is some flexibility in the choice of scale factor, that absolute-value operations can be eliminated by bit masking, and that a two-step scaling can eliminate *all* premature underflow and overflow.

The major difficulty for the programmer is decoding the floating-point representation and choosing suitable bit masks and constants to accomplish the scaling. Priest exhibits code only for IEEE 754 64-bit arithmetic, and hides the architecture-dependent byte storage order by assuming that `long long int` is a supported 64-bit integer data type. With the same variable notation as before, here is what his code looks like:

```
void
priest_cxdiv(cx_double result, const cx_double z, const cx_double w)
{   /* set result = z / w */
    union
    {
        long long int i;    /* must be 64-bit integer type */
        double d;           /* must be 64-bit IEEE 754 type */
    } aa, bb, cc, dd, ss;
    double a, b, c, d, t;
    int ha, hb, hc, hd, hz, hw, hs; /* components of z and w */

    a = CXREAL_(z);
    b = CXIMAG_(z);
    c = CXREAL_(w);
    d = CXIMAG_(w);
    aa.d = a; /* extract high-order 32 bits to estimate |z| and |w| */
    bb.d = b;
    ha = (aa.i >> 32) & 0x7fffffff;
    hb = (bb.i >> 32) & 0x7fffffff;
    hz = (ha > hb) ? ha : hb;
    cc.d = c;
    dd.d = d;
    hc = (cc.i >> 32) & 0x7fffffff;
    hd = (dd.i >> 32) & 0x7fffffff;
    hw = (hc > hd) ? hc : hd;   /* compute the scale factor */

    if (hz < 0x07200000 && hw >= 0x32800000 && hw < 0x47100000)
    {                    /* |z| < 2^-909 and 2^-215 <= |w| < 2^114 */
        hs = (((0x47100000 - hw) >> 1) & 0xfff00000) + 0x3ff00000;
    }
    else
        hs = (((hw >> 2) - hw) + 0x6fd7ffff) & 0xfff00000;

    ss.i = (long long int)hs << 32; /* scale c & d, & get quotient */
    c *= ss.d;
    d *= ss.d;
    t = ONE / (c * c + d * d);
    c *= ss.d;
    d *= ss.d;

    CXSET_(result, (a * c + b * d) * t, (b * c - a * d) * t);
}
```

The variables ha through hd hold the top 32 bits of the four components with the sign bits masked to zero, and the larger of each pair are then used to determine hs, which has the top 32 bits of the scale factor. That scale factor is an exact power of the base, and is constructed in the structure element ss.i, and used as its memory overlay ss.d. The scale-factor selection is intricate and takes three journal pages to describe; see [Pri04, §2.2] for the details.

Determining the scale factor requires only fast 32-bit and 64-bit integer operations, and once it is available, the final result is constructed with 3 adds, 1 divide, and 12 multiplies.

Priest observes that the product-sums in the last statement are subject to catastrophic subtraction loss, but does not attempt to correct that problem.

Instead of laboriously deriving new scale-factor code for a float version of Priest's method for complex division, it is more sensible to promote the float operands to double and call priest_cxdiv(), since all internal products are then exact, and subtraction loss is eliminated. The long double type is more difficult to handle, since it too needs new scale-factor code, and there are 80-, 96-, and 128-bit storage conventions for the 80-bit type, a paired-double 128-bit format, and a separate 128-bit representation, plus differing byte-addressing practices.

Priest claims that his algorithm also correctly handles the case of complex infinite operands: when the exact result is infinite, at least one component of the result is Infinity, and the other may be a NaN, as permitted by C99 and noted at the beginning of this chapter.

Timing tests on several architectures with arguments of random signs, and magnitudes drawn from both uniform and logarithmic distributions, show that Priest's algorithm is always faster than Stewart's, and is much faster than the C99 algorithm that we present in **Section 15.9** on page 449.

15.13 Complex division: avoiding subtraction loss

The problem of catastrophic subtraction loss remains in the four algorithms for complex division ([C99, §G.5.1, p. 469], [Smi62], [Ste85], and [Pri04]) that we have presented in the preceding sections. It is a requirement of IEEE 754 arithmetic that results of the five basic operations of real arithmetic are always correctly rounded. Even though complex arithmetic is more difficult than real arithmetic, a library implementation of the basic complex operations should guarantee *relative errors* that are no worse than a few units in the last place for *all possible* operands.

In each algorithm, subtraction loss lurks in the expression forms $ab + cd$ and $ab + c$. In **Section 15.9** on page 449, we proposed handling the first form with our pair-precision product-sum function family, `PPROSUM()`. In **Section 15.10** on page 451, we suggested using the `FMA()` fused multiply-add family for the second form.

When we recall that each product can be represented *exactly* as a sum of pairs, then we can apply our `VSUM()` primitive for accurate vector summation:

```
fp_t v[4], result;
v[3] = a * b;                     /* hi(a * b) */
v[2] = c * d;                     /* hi(c * d) */
v[1] = FMA(a, b, -v[3]);          /* lo(a * b) */
v[0] = FMA(c, d, -v[2]);          /* lo(c * d) */
result = VSUM((ft_t)NULL, 4, v);  /* a * b + c * d, accurately */
```

When a fast `fma()` operation is available, the problem expression can, and should, be computed that way. However, when the `fma()` function is comparatively slow, it is better to use it only when subtraction loss is known to happen: in a binary base, the terms must be of opposite sign, and ratio of their magnitudes must lie in $[\frac{1}{2}, \frac{3}{2}]$ (see **Section 4.19** on page 89).

We therefore replace the call to `PPROSUM()` with a call to this faster version:

```
static void
fast_pprosum(fp_pair_t result, fp_t a, fp_t b, fp_t c, fp_t d)
{   /* compute result = a * b + c * d accurately and quickly */
    fp_t ab, ab_abs, cd, cd_abs;

    ab = a * b;
    cd = c * d;
    result[1] = ZERO;

    if ((ab >= ZERO) && (cd >= ZERO))           /* same signs */
        result[0] = ab + cd;
    else if ((ab < ZERO) && (cd < ZERO))        /* same signs */
        result[0] = ab + cd;
    else                                        /* opposite signs */
    {
        ab_abs = QABS(ab);
        cd_abs = QABS(cd);

        if ( ((cd_abs + cd_abs) < ab_abs) || ((ab_abs + ab_abs) < cd_abs) )
            result[0] = ab + cd;
        else                                    /* certain loss */
        {
            fp_t err_ab;
            err_ab = FMA(a, b, -ab);
            result[0] = FMA(c, d, ab);
            result[1] = err_ab;
        }
    }
}
```

In that code, PSET_() is a macro that expands inline to set both components of its first argument, without worrying about the sign of zero in the second component, as the function PSET() does.

15.14 Complex imaginary part

The imaginary part of a complex number is just its second component, so retrieving it is simple:

```
fp_t
CXIMAG(const fp_cx_t z)
{   /* complex imaginary part: return imag(z) */
    return (CXIMAG_(z));
}
```

The C99 companion function exploits the storage mandate cited earlier on page 441 to cast a complex-value pointer to an array-element pointer via our conversion macro:

```
fp_t
CIMAG(fp_c_t z)
{   /* complex imaginary part: return imag(z) */
    return (CXIMAG_((fp_t *)&z));
}
```

15.15 Complex multiplication

After division, the next most difficult operation in the complex primitives is multiplication. If $x = a + ib$ and $y = c + id$, then complex multiplication is defined like this:

$$\begin{aligned} xy &= (a+ib)(c+id) \\ &= (ac-bd) + i(ad+bc). \end{aligned}$$

That looks straightforward, but as happens with division, the problems of significance loss and premature overflow and underflow, and the introduction of spurious NaN results from subtraction and division of Infinity, must be dealt with.

Our algorithm follows the procedure recommended in a non-binding annex of the C99 Standard [C99, §G.5.1, p. 468], and like the division algorithm, it computes first, and handles exceptional cases later:

```
void
CXMUL(fp_cx_t result, const fp_cx_t x, const fp_cx_t y)
{   /* complex multiply: result = x * y */
    fp_t a, b, c, d;
    fp_pair_t ac_bd, ad_bc;

    a = CXREAL_(x);
    b = CXIMAG_(x);
    c = CXREAL_(y);
    d = CXIMAG_(y);

    PPROSUM(ac_bd, a, c, -b, d);
    PPROSUM(ad_bc, a, d, b, c);
    CXSET_(result, PEVAL(ac_bd), PEVAL(ad_bc));

    if (ISNAN(CXREAL_(result)) && ISNAN(CXIMAG_(result)))
    {
        int recalc;

        recalc = 0;

        if ( ISINF(a) || ISINF(b) )     /* x is infinite */
        {   /* Box the infinity and change NaNs in other factor to 0 */
            a = COPYSIGN(ISINF(a) ? ONE : ZERO, a);
            b = COPYSIGN(ISINF(b) ? ONE : ZERO, b);

            if (ISNAN(c))
                c = COPYSIGN(ZERO, c);

            if (ISNAN(d))
                d = COPYSIGN(ZERO, d);

            recalc = 1;
        }

        if  ( ISINF(c) || ISINF(d) )    /* y is infinite */
        {    /* Box infinity and change NaNs in other factor to 0 */
            c = COPYSIGN(ISINF(c) ? ONE : ZERO, c);
            d = COPYSIGN(ISINF(d) ? ONE : ZERO, d);

            if (ISNAN(a))
                a = COPYSIGN(ZERO, a);

            if (ISNAN(b))
                b = COPYSIGN(ZERO, b);

            recalc = 1;
        }

        if (!recalc && (ISINF(a * c) || ISINF(b * d) || ISINF(a * d) || ISINF(b * c)))
        {   /* Recover infinities from overflow: change NaNs to zero */
            if (ISNAN(a))
                a = COPYSIGN(ZERO, a);

            if (ISNAN(b))
                b = COPYSIGN(ZERO, b);

            if (ISNAN(c))
                c = COPYSIGN(ZERO, c);

            if (ISNAN(d))
                d = COPYSIGN(ZERO, d);

            recalc = 1;
        }

        if (recalc)
        {
            fp_t inf;
            inf = INFTY();
            CXSET_(result, inf * ( a * c - b * d ),
                           inf * ( a * d + b * c ));
        }
    }
}
```

The code deals with the common case quickly, but when both components of the result are found to be NaNs, further testing, and possible computation of properly signed infinities, is needed. We improve upon the recommended algorithm by using `PPROSUM()` and `PEVAL()` for the computation of $ac - bd$ and $ad + bc$. In practice, we can use our `fast_pprosum()` private function (see **Section 15.13** on page 455) instead of `PPROSUM()`.

As with division, C99 does not provide a function for complex multiplication, because that operation too is built-in. Following our practice with `CDIV()`, we provide a C99-style function that uses the code in `CXMUL()`:

```
fp_c_t
CMUL(fp_c_t x, fp_c_t y)
{   /* complex multiply: return x * y */
    fp_cx_t xx, yy, result;

    CTOCX_(xx, x);
    CTOCX_(yy, y);
    CXMUL(result, xx, yy);

    return (CXTOC_(result));
}
```

15.16 Complex multiplication: error analysis

The accuracy of multiplication with complex arithmetic has been studied with detailed mathematical proofs that occupy about ten journal pages [BPZ07]. That work has been recently extended to algorithms using fused multiply-add operations [JKLM17], slightly improving the bounds cited here. The authors show that for floating-point base β under the conditions

- subnormals, overflow, and underflow are avoided,
- *round-to-nearest* mode is in effect for real arithmetic operations,
- the number of significand bits is at least five, and
- expressions of the form $ab \pm cd$ are computed accurately,

then complex multiplication has a maximum relative error below $\frac{1}{2}\sqrt{5}\beta^{1-t}$, where t is the number of significand digits. For binary IEEE 754 arithmetic, their bound corresponds to 1.118 ulps. Importantly, their analysis leads to simple test values that produce the *worst-case* errors:

$$
\begin{aligned}
\beta &= 2,\\
t &= 24 \text{ or } 53,\\
e &= \tfrac{1}{2}\beta^{1-t},\\
z_0 &= 3/4 + \big((3(1-4e))/4\big)i, && \textit{32-bit IEEE 754 format,}\\
&= \frac{3}{4} + \frac{12\,582\,909}{16\,777\,216}i\\
&= \texttt{0x1.8p-1 + 0x1.7fff\_fap-1 * I},\\
z_1 &= \big(2(1+11e)\big)/3 + \big((2(1+5e))/3\big)i
\end{aligned}
$$

$$
\begin{aligned}
&= \frac{5\,592\,409}{8\,388\,608} + \frac{5\,592\,407}{8\,388\,608}i \\
&= \texttt{0x1.5555\_64p-1 + 0x1.5555\_5cp-1 * I,} \\
\text{exact } z_0 z_1 &= (5e + 10e^2) + (1 + 6e - 22e^2)i \\
&= \frac{41\,943\,045}{140\,737\,488\,355\,328} + \frac{140\,737\,538\,686\,965}{140\,737\,488\,355\,328}i \\
&= \texttt{0x1.4000\_028p-22 + 0x1.0000\_05ff\_ffeap+0 * I,} \\
w_0 &= \big(3(1+4e)\big)/4 + (3/4)i, \qquad \textit{64-bit IEEE 754 format,} \\
&= \frac{6\,755\,399\,441\,055\,747}{9\,007\,199\,254\,740\,992} + \frac{3}{4}i \\
&= \texttt{0x1.8000\_0000\_0000\_3p-1 + 0x1.8p-1 * I,} \\
w_1 &= \big(2(1+7e)\big)/3 + \big(2(1+e)/3\big)i \\
&= \frac{3\,002\,399\,751\,580\,333}{4\,503\,599\,627\,370\,496} + \frac{3\,002\,399\,751\,580\,331}{4\,503\,599\,627\,370\,496}i \\
&= \texttt{0x1.5555\_5555\_5555\_ap-1 +} \\
&\ \texttt{0x1.5555\_5555\_5555\_6p-1 * I,} \\
\text{exact } w_0 w_1 &= (5e + 14e^2) + (1 + 6e + 2e^2)i \\
&= \frac{22\,517\,998\,136\,852\,487}{40\,564\,819\,207\,303\,340\,847\,894\,502\,572\,032} + \\
&\ \frac{40\,564\,819\,207\,303\,367\,869\,492\,266\,795\,009}{40\,564\,819\,207\,303\,340\,847\,894\,502\,572\,032}i \\
&= \texttt{0x1.4000\_0000\_0000\_1cp-51 +} \\
&\ \texttt{0x1.0000\_0000\_0000\_3000\_0000\_0000\_008p+0 * I.}
\end{aligned}
$$

Despite the factors of $2/3$ in the expressions for z_1 and w_1, all of the components are *exactly representable* in binary arithmetic.

Using those worst-case values for the `float` and `double` formats, this author wrote two short test programs, `tcmul2.c` and `tcmul3.c`, in the `exp` subdirectory. The first uses native complex arithmetic and is run with the native math library. The second replaces the complex multiplications by calls to our `CMUL()` family members, and thus requires the `mathcw` library. The programs were then run on several architectures, including GNU/LINUX (Alpha, AMD64, IA-32, IA-64, PowerPC, and SPARC), FREEBSD (IA-32), OPENBSD (IA-32), and SOLARIS (AMD64 and SPARC), with multiple C99-level compilers.

All of the tests show relative errors below 0.539 ulps for the 32-bit native complex multiply. For the 64-bit native complex multiply, all systems produce a correctly rounded imaginary part, but *almost all* of the test systems lose 49 of the 53 significand bits for the real part of the product! Only on the FREEBSD IA-32 and SOLARIS AMD64 tests, and with one uncommon commercial compiler on GNU/LINUX AMD64, is the real part correctly rounded. By contrast, the test program that uses the mathcw library routines produces correctly rounded results for those tests on all platforms. The lesson is that *complex arithmetic in* C *is not yet trustworthy*.

15.17 Complex negation

The negative of a complex number $x + iy$ is just $-x - iy$, so the implementation is simple:

```
void
CXNEG(fp_cx_t result, const fp_cx_t z)
{   /* complex negation: result = -z */
    CXSET_(result, -CXREAL_(z), -CXIMAG_(z));
}
```

There is no C99 Standard function for complex negation, because the operation is built-in, but we provide a companion function that uses inline code, rather than calling `CXNEG()`:

```
fp_c_t
CNEG(fp_c_t z)
{   /* complex negation: return -z */
    return (-z);
}
```

15.18 Complex projection

The projection function may be unique to C99, and the Standard describes it this way [C99, §7.3.9.4, p. 179]:

> *The `cproj()` functions compute a projection of `z` onto the Riemann*[2] *sphere: `z` projects to `z` except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If `z` has an infinite part, then `cproj(z)` is equivalent to*
>
> *`INFINITY + I * copysign(0.0, cimag(z))`.*

That description readily leads to obvious code:

```
void
CXPROJ(fp_cx_t result, const fp_cx_t z)
{   /* complex projection of z onto Riemann sphere: result = proj(z) */
    if (ISINF(CXREAL_(z)) || ISINF(CXIMAG_(z)))
        CXSET_(result, INFTY(), COPYSIGN(ZERO, CXIMAG_(z)));
    else
        CXSET_(result, CXREAL_(z), CXIMAG_(z));
}
```

The C99 companion function uses `CXPROJ()` for the real work:

```
fp_c_t
CPROJ(fp_c_t z)
{   /* complex projection of z onto Riemann sphere: return proj(z) */
    fp_cx_t zz, result;

    CTOCX_(zz, z);
    CXPROJ(result, zz);

    return (CXTOC_(result));
}
```

If a sphere of radius one is centered at the origin on the complex plane (see **Figure 15.2**), then a line from any point $w = (u, v)$ on the plane outside the sphere to the North Pole of the sphere intersects the sphere in exactly two places, like an arrow shot through a balloon. Thus, every finite point (u, v) has a unique image point on the sphere at the first intersection. As we move further away from the origin, that intersection point approaches the North Pole. The two intersections with the sphere coincide only for points where one or both components of the point on the plane are infinitely far away. That is, all complex infinities project onto a single point, the North Pole, of the Riemann sphere.

15.19 Complex real part

The real part of a complex number is just its first component, so retrieving it is easy:

```
fp_t
CXREAL(const fp_cx_t z)
{   /* complex real part: return real(z)  */
    return (CXREAL_(z));
}
```

[2] See the footnote in **Section 11.2** on page 303.

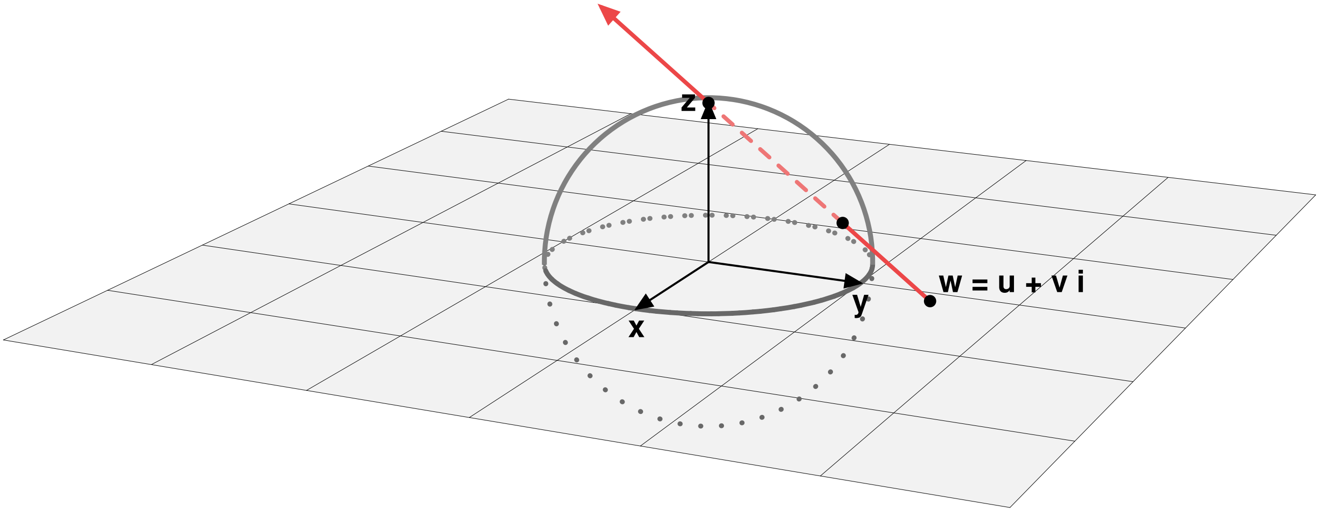


Figure 15.2: Projecting a complex point onto the Riemann sphere.

As with `CIMAG()`, the corresponding C99 function for the real part uses the storage requirement to convert a complex-value pointer to an array-element pointer:

```
fp_t
CREAL(fp_c_t z)
{   /* complex real part: return real(z) */
    return (CXREAL_((fp_t *)&z));
}
```

15.20 Complex subtraction

The subtraction operation for complex numbers simply requires computing the difference of their real and imaginary components, so the code is obvious:

```
void
CXSUB(fp_cx_t result, const fp_cx_t x, const fp_cx_t y)
{   /* complex subtraction: result = x - y */
    CXSET_(result, CXREAL_(x) - CXREAL_(y), CXIMAG_(x) - CXIMAG_(y));
}
```

There is no C99 function for complex subtraction, because it is a built-in operation, but we can easily provide a C99-style function. Because the operation is so simple, we code it directly, rather than calling `CXSUB()` to do the work:

```
fp_c_t
CSUB(fp_c_t x, fp_c_t y)
{   /* complex subtraction: return x - y */
    return (x - y);
}
```

15.21 Complex infinity test

The peculiar C99 definition of complex infinite values cited on page 441 suggests that we provide a primitive for testing for a complex infinity, even though the ISO Standard does not specify test functions for complex types. The code for our two families of complex types is short:

```
int
ISCXINF(const fp_cx_t z)
{   /* return 1 if z is a complex Infinity, else 0 */
    return (ISINF(CXREAL_(z)) || ISINF(CXIMAG_(z)));
}
```

```
int
ISCINF(fp_c_t z)
{   /* return 1 if z is a complex Infinity, else 0 */
    return (ISINF(CREAL(z)) || ISINF(CIMAG(z)));
}
```

15.22 Complex NaN test

The C99 definition of complex infinite values complicates NaN tests, so we extend the ISO Standard with test functions to hide the mess. Their code looks like this:

```
int
ISCXNAN(const fp_cx_t z)
{   /* return 1 if z is a complex NaN, else 0 */
    fp_t x, y;
    int result;

    x = CXREAL_(z);
    y = CXIMAG_(z);

    if (ISINF(x) || ISINF(y))
        result = 0;
    else if (ISNAN(x) || ISNAN(y))
        result = 1;
    else
        result = 0;

    return (result);
}
```

```
int
ISCNAN(fp_c_t z)
{   /* return 1 if z is a complex NaN, else 0 */
    fp_cx_t zz;

    CTOCX_(zz, z);
    return (ISCXNAN(zz));
}
```

15.23 Summary

At the time of writing this, support for complex arithmetic in compilers for the C language family is limited, and on many systems, of doubtful quality. Consequently, the functions for complex arithmetic described in this chapter have not received extensive use.

Most of the functions are simple, and good candidates for inline expansion by compilers. We encouraged such expansion by using the macros `CXIMAG_()`, `CXREAL_()`, `CXSET_()`, `CXTOC_()` and `CTOCX_()` to produce inline code instead of a function call, and conceal all array-element references.

The functions for complex absolute value and complex argument are numerically troublesome, but can easily be expressed in terms of the `HYPOT()` and `ATAN2()` families which are carefully implemented to produce high accuracy.

The functions for multiplication and division are subject to premature underflow and overflow, and to massive subtraction loss. We discussed four different algorithms for complex division, and showed how careful handling of expressions of the forms $a \times b + c \times d$ and $a \times b + c$ with the occasional help of fused multiply-add operations can eliminate subtraction loss in both complex division and complex multiplication. Since the mathcw package and this book were completed, several additional papers on the problem of accurate complex multiplication and division have appeared [BS12, WE12, JLM13b, JLM13a, Mul15, Jea16, JLMP16, JKLM17], but comparison with our code remains a project for future study.

The last of those papers also points out an issue that we have not treated in this chapter: computer algorithms for complex multiplication may not obey the expected mathematical property of commutativity $(w \times z \equiv z \times w)$, and thus, may incorrectly produce a nonzero imaginary part in the computation $z \times z^\star = (x + yi) \times (x - yi) = x^2 - x \times yi + y \times xi + y^2 = x^2 + y^2$, a value that, mathematically, is purely real.

The difficulties in producing fast and correctly rounded complex multiply and divide cry out for a hardware solution. If hardware provided multiple functional units with extended range and double-width results, then complex absolute value, complex division, and complex multiplication could be done quickly, and without premature underflow or overflow, or unnecessary subtraction loss. Sadly, only a few historical machines seem to have addressed that need:

- Bell Laboratories Model 1 through Model 4 relay computers with fixed-point decimal arithmetic (1938–1944) (ten digits, only eight displayed) [Sti80],
- Bell Laboratories Model 5 relay computer with seven-digit floating-point decimal arithmetic (1945) [Sti80], and
- Lawrence Livermore National Laboratory S-1 (see **Appendix H.7**),

No current commercially significant computer architecture provides complex arithmetic in hardware, although there are several recent papers in the chip-design literature on that subject.

Unlike Fortran, C99 does not offer any native I/O support for complex types. The real and imaginary parts must be handled explicitly, forcing complex numbers to be constructed and deconstructed by the programmer for use in calls to input and output functions for real arithmetic. Maple provides a useful extension in `printf()` format specifiers that could be considered for future C-language standardization: the letter `Z` is a numeric format modifier character that allows a single format item to produce two outputs, like this:

```
% maple
> printf("%Zg\n", 1 + 2*I);
1+2I
> printf("%5.1Zf\n", 1 + 2*I);
  1.0 +2.0I
```

The `_Imaginary` data type is a controversial feature of C99, and its implementation by compilers and libraries was therefore made optional. That decision makes the type practically unusable in portable code. Although mathematicians work with numbers on the real axis, and numbers in the complex plane, they rarely speak of numbers that are restricted to the imaginary axis. More than four decades of absence of the imaginary data type from Fortran, the only widely used, and standardized, language for complex arithmetic, suggests that few programmers will find that type useful. The primary use of an imaginary type may be to ensure that an expression like `z * I` is evaluated without computation as `-cimag(z) + creal(z) * I` to match mathematics use, instead of requiring an explicit complex multiplication that gets the wrong answer when one or both of the components of `z` is a negative zero, Infinity, or NaN.

We provided complex companions for the real `ISxxx()` family only for Infinity and NaN tests, because they are nontrivial, and likely to be programmed incorrectly if done inline. Most of the relational operations do not apply to complex values, and it is unclear whether tests for complex finite, normal, subnormal, and zero values are useful. We have not missed them in writing any of the complex-arithmetic routines in the `mathcw` library. Should they prove desirable in some applications, they could easily be generated inline by private macros like these:

```
#define ISCFINITE(z)     ISFINITE(CREAL(z))      && ISFINITE(CIMAG(z))
#define ISCNORMAL(z)     ISNORMAL(CREAL(z))      && ISNORMAL(CIMAG(z))
#define ISCSUBNORMAL(z)  ISSUBNORMAL(CREAL(z))   && ISSUBNORMAL(CIMAG(z))
#define ISCZERO(z)       (CREAL(z) == ZERO)      && (CIMAG(z) == ZERO)

#define ISCXFINITE(z)    ISFINITE(CXREAL_(z))    && ISFINITE(CXIMAG_(z))
#define ISCXNORMAL(z)    ISNORMAL(CXREAL_(z))    && ISNORMAL(CXIMAG_(z))
#define ISCXSUBNORMAL(z) ISSUBNORMAL(CXREAL_(z)) && ISSUBNORMAL(CXIMAG_(z))
#define ISCXZERO(z)      (CXREAL_(z) == ZERO)    && (CXIMAG_(z) == ZERO)
```

However, the programmer needs to decide what should be done about cases where one component is, say, subnormal, and the other is not. For example, it might be desirable to change && to `||` in the definition of `ISCXSUBNORMAL()`.

We defer presentation of the computation of complex versions of the elementary functions required by C99 to Chapter 17 on page 475, because now is a good time to apply the primitives of this chapter to one of the simplest problems where complex arithmetic is required: solution of quadratic equations, the subject of the next chapter.

16 Quadratic equations

> THE FIRST KNOWN SOLUTION OF A QUADRATIC EQUATION IS THE ONE GIVEN IN THE BERLIN PAPYRUS FROM THE MIDDLE KINGDOM (CA. 2160–1700 BC) IN EGYPT.
>
> — ERIC W. WEISSTEIN
> WOLFRAM *MathWorld*™ (2006).

We mentioned in Chapter 2 the need for solving quadratic equations, but we deferred further discussion of how to do so until we could develop some essential tools. It is now time to tackle the problem.

16.1 Solving quadratic equations

> THE QUADRATIC EQUATION IS ONE OF THE SIMPLEST MATHEMATICAL ENTITIES AND IS SOLVED ALMOST EVERYWHERE IN APPLIED MATHEMATICS. ITS ACTUAL USE ON A COMPUTER MIGHT BE EXPECTED TO BE ONE OF THE BEST UNDERSTOOD OF COMPUTER ALGORITHMS. INDEED IT IS NOT, . . .
>
> — GEORGE E. FORSYTHE
> *Solving a Quadratic Equation on a Computer* (1969).

In Section 2.3 on page 9, we found a need for the roots of a quadratic equation

$$Ax^2 + Bx + C = 0$$

arising from a truncated second-order Taylor-series approximation. In grade school, you learned its solution:

$$x = (-B \pm \sqrt{B^2 - 4AC})/(2A).$$

In our Taylor-series application, the coefficient A depends on the second derivative, which may be small, or hard to compute accurately. With the schoolbook formula for the roots, inaccuracies in A therefore contaminate both roots, whatever the values of B and C may be.

If we multiply the numerator and denominator by a common factor, and simplify, we obtain an alternate formula for the roots:

$$\begin{aligned} x &= (-B \mp \sqrt{B^2 - 4AC})(-B \pm \sqrt{B^2 - 4AC})/ \\ &\quad ((-B \mp \sqrt{B^2 - 4AC})(2A)) \\ &= \big(B^2 - (B^2 - 4AC)\big)/\big((-B \mp \sqrt{B^2 - 4AC})(2A)\big) \\ &= 2C/(-B \mp \sqrt{B^2 - 4AC}). \end{aligned}$$

Because we have two roots to choose from, we pick the one that avoids possible subtraction loss:

$$x = \begin{cases} 2C/(-B - \sqrt{B^2 - 4AC}) & \text{for } B > 0, \\ 2C/(-B + \sqrt{B^2 - 4AC}) & \text{for } B \leq 0. \end{cases}$$

In that form, inaccuracies in A have little effect on the accuracy of the roots if $B^2 \gg |4AC|$. In addition, for our Taylor-series application, the value C corresponds to the leading term of the series, which is almost always a small, and exactly representable, constant for the functions of interest in this book, in which case, no additional rounding error is propagated to the roots from the numerator $2C$.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_16

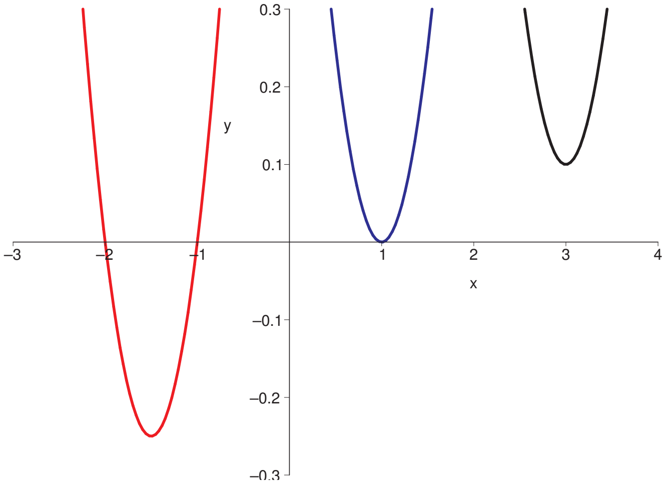


Figure 16.1: The three possible cases for the roots of a quadratic equation

$$y(x) = (x - x_1)(x - x_2) = Ax^2 + Bx + C = 0,$$

with *real* coefficients A, B, and C.
In the red left curve, $y(x) = (x+2)(x+1) = x^2 + 3x + 2$, the roots differ: $x_1 = -2$ and $x_2 = -1$.
In the blue middle curve, $y(x) = (x-1)^2 = x^2 - 2x + 1$, the roots coincide: $x_1 = x_2 = 1$.
In the black right curve, formed by moving $y(x) = (x-3)^2 = x^2 - 6x + 9$ upward by $\frac{1}{10}$, the formerly coincident real roots now differ and are complex: $x_{1,2} = 3 \pm \sqrt{\frac{1}{10}}i$.

If the *discriminant*, $B^2 - 4AC$, is positive and nonzero, the roots are real and different. If the discriminant is zero, the two roots are real and identical. Otherwise, when the discriminant is negative, the roots are complex and different. The three cases are illustrated in **Figure 16.1**.

In approximations to real functions, we expect only real roots, so we could treat a negative discriminant as a numerical artifact, and arbitrarily set it to zero. However, in the following discussion, we allow for complex roots.

A general numerical solver for the roots of a quadratic equation needs to handle cases where one or more of the coefficients are zero, infinity, or NaN. We summarize the possibilities in **Table 16.1** on the next page.

Apart from errors in the coefficients A, B, and C, there are three major sources of computational error in the roots:

- The discriminant suffers subtraction loss, that is, $B^2 \approx 4AC$.
- Either, or both, of the terms B^2 and $4AC$ in the discriminant, or their difference, are too large or too small to represent in the finite precision and range of floating-point arithmetic: they suffer overflow or underflow.
- The divisions underflow or overflow.

The roots of the quadratic equation are invariant under uniform exact scaling of the coefficients, because that just corresponds to multiplying both sides of the equation by a constant. We can therefore largely eliminate the possibility of destructive premature overflow and underflow by rescaling the coefficients by a common power of the base. The C99 and mathcw libraries provide the `ILOGB()` and `SCALBN()` functions that do the job.

The remaining problem is then the accurate computation of $\sqrt{B^2 - 4AC}$. If we have higher precision available such that B^2 and $4AC$ can be computed exactly, then the subtraction incurs one rounding error in the higher precision, and the error in the square root then corresponds to roughly one rounding error in the original (lower) precision.

Most commonly, however, higher precision is not available, so we then ask: *what is the worst-case error?* Clearly, the error is largest when the subtraction loses *all* significant digits, producing a zero result, even though the exact answer may be nonzero. It is easy to exhibit a case where that happens:

- Observe that in any finite-precision arithmetic system, there is always a smallest value, ϵ, such that $\mathrm{fl}(1+\epsilon) \neq 1$, but any smaller value, say $\epsilon/2$, results in $\mathrm{fl}(1+\epsilon/2) = 1$. The value ϵ is called the *machine epsilon*.

Table 16.1: Special cases in the solution of the quadratic equation, $Ax^2 + Bx + C = A(x - x_1)(x - x_2) = 0$. Expansion of the middle part shows that the roots are related by $x_1 x_2 = C/A$, so once one of them has been determined by a numerically stable formula, the other is readily obtained.

Indeterminate roots are best represented as NaNs, when IEEE 754 arithmetic is available.

Cases -6 through -1 must be checked first, in any order, but cases 0 through 7 should be handled in ascending order.

When the coefficients are real, complex numbers can arise only in cases 5 and 7, and then only if the value under the square-root operation is negative. In IEEE 754 arithmetic, a reasonable representation of a complex root in real arithmetic is a NaN, and the `sqrt()` function ensures that result.

Case	A	B	C	Roots
-6	any	any	NaN	indeterminate solution
-5	any	NaN	any	indeterminate solution
-4	NaN	any	any	indeterminate solution
-3	any	any	$\pm\infty$	indeterminate solution
-2	any	$\pm\infty$	any	indeterminate solution
-1	$\pm\infty$	any	any	indeterminate solution
0	0	0	0	indeterminate solution
1	0	0	nonzero	no solution
2	0	nonzero	0	$x_1 = 0$, no x_2
3	0	nonzero	nonzero	$x_1 = -C/B$, no x_2
4	nonzero	0	0	$x_1 = 0$, $x_2 = 0$
5	nonzero	0	nonzero	$x_1 = +\sqrt{-C/A}$, $x_2 = -\sqrt{-C/A}$
6	nonzero	nonzero	0	$x_1 = 0$, $x_2 = -B/A$
7	nonzero	nonzero	nonzero	$z = -\left(B + \operatorname{sign}(B)\sqrt{B^2 - 4AC}\right)/2$, $x_1 = C/z$, $x_2 = z/A$

- Choose coefficients of the quadratic equation as follows:

$$\begin{aligned}
\delta &= \sqrt{\epsilon}/2, \\
A &= 1/4, \\
B &= 1 + \delta, \\
C &= 1 + 2\delta.
\end{aligned}$$

- Compute the exact and floating-point discriminants:

$$\begin{aligned}
\operatorname{exact}(B^2 - 4AC) &= (1+\delta)^2 - 4(1/4)(1+2\delta) \\
&= (1 + 2\delta + \delta^2) - (1 + 2\delta) \\
&= \delta^2, \\
\operatorname{fl}(B^2 - 4AC) &= \operatorname{fl}\left((1+\delta)^2 - 4(1/4)(1+2\delta)\right) \\
&= \operatorname{fl}(1+2\delta) - \operatorname{fl}(1+2\delta) \\
&= 0.
\end{aligned}$$

- Compute the exact and floating-point values of the value z defined in case 7 of Table 16.1:

$$\begin{aligned}
\operatorname{exact}(z) &= \operatorname{exact}\left(-(B + \operatorname{sign}(B)\sqrt{B^2 - 4AC})/2\right) \\
&= -\left((1+\delta) + \sqrt{\delta^2}\right)/2 \\
&= -(1+2\delta)/2, \\
\operatorname{fl}(z) &= \operatorname{fl}\left(-(B + \operatorname{sign}(B)\sqrt{B^2 - 4AC})/2\right) \\
&= -(1 + \delta + 0)/2 \\
&= -(1+\delta)/2.
\end{aligned}$$

- Finally, compute the relative error in the computed z, and simplify with the reciprocal formula $1/(1+x) = 1 - x + x^2 - x^3 + \cdots$:

$$\begin{aligned}
\operatorname{relerr}\bigl(\mathrm{fl}(z), \mathrm{exact}(z)\bigr) &= \bigl(\mathrm{fl}(z) - \mathrm{exact}(z)\bigr) / \mathrm{exact}(z) \\
&= \delta/(1+2\delta) \\
&= \delta(1 - (2\delta) + (2\delta)^2 + \cdots) \\
&\approx \delta \\
&\approx \sqrt{\epsilon}/2.
\end{aligned}$$

The last result shows that, when there is severe subtraction loss in the discriminant, as many as *half of the digits* in the roots can be in error.

As we noted earlier, if we could just compute $B^2 - 4AC$ *exactly*, then we could produce the roots with small total error, no more than that from four rounding errors. To do so, we need a temporary precision that is at least double that of our working precision. Because we often do not have that luxury, we have to simulate the higher precision in working precision, and the traditional way to do that is to split each number into a sum of two terms: a high-order term whose square is exactly representable, plus a low-order term. We showed how to do the split in **Section 13.11** on page 359.

We can then write the splitting and discriminant relations like this:

$$\begin{aligned}
A &= A_{\mathrm{hi}} + A_{\mathrm{lo}}, \\
B &= B_{\mathrm{hi}} + B_{\mathrm{lo}}, \\
C &= C_{\mathrm{hi}} + C_{\mathrm{lo}}, \\
B^2 - 4AC &= (B_{\mathrm{hi}}^2 - 4A_{\mathrm{hi}}C_{\mathrm{hi}}) + \\
&\quad \bigl((2B_{\mathrm{hi}}B_{\mathrm{lo}} - 4A_{\mathrm{hi}}C_{\mathrm{lo}} - 4A_{\mathrm{lo}}C_{\mathrm{hi}}) + \\
&\quad (B_{\mathrm{lo}}^2 - 4A_{\mathrm{lo}}C_{\mathrm{lo}})\bigr).
\end{aligned}$$

The first parenthesized term contributing to the discriminant contains the high-order part, and is computed exactly. It is where most of the subtraction loss happens. The second and third terms contain the middle and low-order parts, providing an essential correction to the discriminant.

The code that implements the computation of the roots of the quadratic equation is lengthy, but follows the fourteen cases in order. It looks like this:

```
int
QERT(fp_t a, fp_t b, fp_t c, fp_cx_t x1, fp_cx_t x2)
{
    int result;

    result = QERT_REAL;

    if ( ISNAN(a) || ISNAN(b) || ISNAN(c) || ISINF(a) || ISINF(b) || ISINF(c) )
    {   /* cases -6, -5, -4, -3, -2, -1 */
        x1[0] = x1[1] = x2[0] = x2[1] = SET_EDOM(QNAN(""));
        result = QERT_INDETERMINATE;
    }
    else
    {
        if (a == ZERO)
        {                               /* cases 0, 1, 2, 3 */
            result = QERT_INDETERMINATE;
            x2[0] = x2[1] = SET_EDOM(QNAN(""));

            if (b == ZERO)              /* cases 0 and 1 */
                x1[0] = x1[1] = x2[0];
            else if (c == ZERO)         /* case 2 */
```

```
                x1[0] = x1[1] = ZERO;
            else                        /* case 3 */
            {   /* NB: can underflow or overflow */
                x1[0] = -c / b;
                x1[1] = ZERO;
            }
        }
        else                            /* a is nonzero */
        {                               /* cases 4, 5, 6, 7 */
            volatile fp_t r;

            if (b == ZERO)              /* cases 4 and 5 */
            {
                if (c == ZERO)          /* case 4 */
                    x1[0] = x1[1] = x2[0] = x2[1] = ZERO;
                else
                {                       /* case 5 */
                    /* NB: can underflow or overflow */
                    r = -c / a;
                    STORE(&r);

                    if (r >= ZERO)      /* real root */
                    {
                        x1[0] = SQRT(r);
                        x1[1] = ZERO;
                    }
                    else                /* imaginary root */
                    {
                        x1[0] = ZERO;
                        x1[1] = SQRT(-r);
                    }
                    x2[0] = -x1[0];
                    x2[1] = -x1[1];
                }
            }
            else                        /* cases 6 and 7 */
            {
                if (c == ZERO)
                {                       /* case 6 */
                    x1[0] = x1[1] = ZERO;
                    x2[0] = -b / a;
                    x2[1] = ZERO;
                }
                else                    /* case 7 */
                {
                    fp_t discr_sqrt;
                    fp_pair_t bb_4ac;
                    int n, na, nb, nc;

                    na = ILOGB(a);
                    nb = ILOGB(b);
                    nc = ILOGB(c);
                    n = IMAX(IMAX(na, nb), nc);
                    a = SCALBN(a, -n);
                    b = SCALBN(b, -n);
                    c = SCALBN(c, -n);
                    PPROSUM(bb_4ac, b, b, -FOUR * a, c);
```

```
#if (B != 2) && (B != 4)
                    {   /* 4*a may not be exact: add error term */
                        fp_t err;

                        err = FMA(FOUR, a, -FOUR * a);
                        bb_4ac[0] = FMA(-err, c, bb_4ac[0]);
                    }
#endif

                    r = PEVAL(bb_4ac);  /* r = b*b - 4*a*c */
                    STORE(&r);

                    if (r >= ZERO)
                    {   /* real discriminant and roots */
                        fp_t z;

                        discr_sqrt = SQRT(r);

                        if (b < ZERO)
                            z = -HALF * (b - discr_sqrt);
                        else
                            z = -HALF * (b + discr_sqrt);

                        x1[0] = c / z;
                        x1[1] = ZERO;
                        x2[0] = z / a;
                        x2[1] = ZERO;
                    }
                    else
                    {   /* imaginary discriminant and
                           complex roots */
                        fp_t cc[2], zz[2];

                        result = QERT_COMPLEX;

                        discr_sqrt = SQRT(-r);

                        if (b < ZERO)
                        {
                            zz[0] = -HALF * b;
                            zz[1] =  HALF * discr_sqrt;
                        }
                        else
                        {
                            zz[0] = -HALF * b;
                            zz[1] = -HALF * discr_sqrt;
                        }

                        cc[0] = c;
                        cc[1] = ZERO;

                        CXDIV(x1, cc, zz);
                        x2[0] = zz[0] / a;
                        x2[1] = zz[1] / a;
                    }
                }
```

```
            }
        }

        if (x1[0] > x2[0])
        {   /* order roots so that Real(x1) <= Real(x2) */
            SWAP(x1[0], x2[0]);
            SWAP(x1[1], x2[1]);
        }
        else if ((x1[0] == x2[0]) && (x1[1] > x2[1]))
            SWAP(x1[1], x2[1]);
    }

    return (result);
}
```

The `QERT()` function returns a status report of -1 if either or both roots are indeterminate, 0 if both roots are real, and $+1$ if the roots are complex or pure imaginary. With IEEE 754 arithmetic, tests for NaN could identify the return of indeterminate roots, and tests for zero imaginary parts could identify the case of real roots. However, with older arithmetic systems, we could only transmit that information through the roots by declaring at least one floating-point value to be of special significance. A status return value is clearly preferable, and it can be discarded when it is not required.

The `volatile` qualifier in one declaration statement forces the compiler to use memory values of the declared variables, instead of using values cached inside the CPU, possibly in higher precision. The `STORE()` macro takes care of systems where `volatile` is not supported, or is handled improperly or unreliably.

The splitting task required for accurate evaluation of the discriminant is hidden inside a separate function, `PPROSUM()`, because similar computations are present elsewhere in the library.

Complex arithmetic is required for at most one root, in the computation $x_1 = C/z$; we use the `CXDIV()` primitive to do the work. The second root, $x_2 = z/A$, has a real divisor, so we simply divide each component by that value, instead of invoking `CXDIV()` again. That could produce a different value than `CXDIV()` would return, in the case of Infinity or NaN components. However, that is unlikely to matter.

The final ordering of the roots is not strictly necessary, but can be convenient for the user.

16.2 Root sensitivity

KEEP CALM, AND DON'T BITE THE DENTAL SURGEON.

— SYMPATHY CARD HUMOR.

To better understand the relation between the coefficients and the roots of quadratic equations, it is useful to examine the effect on the roots of small changes in the coefficients. That is, we want to know the error-magnification factors that we introduced in **Section 4.1** on page 61. With some straightforward calculus, and introduction of the variable s for the square root of the discriminant, we can simplify the results to these equations:

$$\begin{aligned}
x &= (-B \pm \sqrt{B^2 - 4AC})/(2A), \\
s &= \sqrt{B^2 - 4AC}, \\
(\delta x/x) &= (-1 \mp C/(xs))(\delta A/A), \\
&= \mp(B/s)(\delta B/B), \\
&= \mp(C/(xs))(\delta C/C).
\end{aligned}$$

The last three relations show how changes in the coefficients affect the roots, and each of them involves a coefficient divided by s. We conclude that when the discriminant is small compared to the coefficients, or equivalently, when the roots are close together, then small changes in coefficients produce large changes in the roots.

16.3 Testing a quadratic-equation solver

I VENTURE TO GUESS THAT NOT MORE THAN FIVE QUADRATIC SOLVERS EXIST ANYWHERE THAT MEET THE GENERAL LEVEL OF SPECIFICATIONS.

— GEORGE E. FORSYTHE,
What is a Satisfactory Quadratic Equation Solver? (1969).

Our solver `QERT()` is essentially a three-input function with five outputs, so exhaustive testing with all possible floating-point arguments is impossible. However, we know that the schoolbook formula for the roots of a quadratic equation contains two subtractions, and the possibility of intermediate underflow or overflow or digit loss in the products B^2 and $4AC$. The worst cases for computing the discriminant $B^2 - 4AC$ arise when the products are large and the difference is small, but nonzero. Testing should therefore concentrate on finding suitable values of the coefficients that produce those worst cases, and should include samples with real roots, with complex roots, and with pure imaginary roots.

Scaling of the coefficients does not change the roots, so we can easily find test values that cause premature overflow or underflow by using scale factors of the form β^k, where k is a positive or negative integer.

Although we can easily choose small or large random coefficients that cause underflow or overflow, it may be hard to guess values that produce a small discriminant. For example, choose two moderately large random integers M and N, set the roots to $x_k = M \pm iN$, and expand the quadratic $(x - x_1)(x - x_2)$. The coefficients are then $A = 1$, $B = 2M$, and $C = M^2 + N^2$, but we must ensure that both B and C are exactly representable. That is easily done if we pick random integers in the range $[\beta^{\lfloor t/2 \rfloor}, \beta^{\lfloor t/2+1 \rfloor} - 1]$, as long as we are prepared to discard those choices where $C > \beta^t$, the value at the end of the range of exactly representable integers. That approach produces two complex roots, but the discriminant, $-4N^2$, is large.

The random integers can be chosen with the help of an arbitrary random-number generator that produces uniformly distributed values in $[0, 1]$ and a code fragment like this:

```
fp_t M, N, w;

w = SCALBN(FP(1.0), FP_T_MANT_DIG / 2); /* w = beta**(floor(t/2)) */
M = FLOOR(w + URAND() * (w * (BASE - ONE) - ONE));
N = FLOOR(w + URAND() * (w * (BASE - ONE) - ONE));
```

By scaling M or N by a power of the base, we can control the relative magnitudes of the real and imaginary parts, as long as we continue to ensure exact coefficients. Tests for roots with large or small ratios of real and imaginary parts are then easy to construct. To get pure imaginary roots, set $B = M = 0$. To generate another test set of coefficients and roots, swap A and C and scale the exact roots by A/C.

There are a few good ways to find coefficients that produce the smallest discriminant, ± 1.

The first is to choose B to be a large random odd integer, and then set $A = (B - 1)/2$ and $C = (B + 1)/2$. The discriminant is then $+1$, and the roots are real with values -1 and $-(B + 1)/(B - 1)$. If B is not random, but chosen to have the special form $\beta^n + 1$, then with positive integer values of n, the second root is also exactly representable. When $\beta \neq 2$, scale the coefficients by two to ensure exact representability.

The influential numerical analyst George Forsythe quoted in the epigraph for this section considers some hard cases [For69a, For69b] of quadratic equations. He points out the need for computation of the discriminant in higher precision, and for scaling to avoid premature overflow and underflow. However, he does not exhibit a practical algorithm for their solution.

Kahan [Kah04b] discusses the problems of accurate computation of the discriminant, and testing of the solver, but leaves a proof of his discriminant algorithm for future work. That task was completed by Sylvie Boldo [Bol09] as this book was nearing completion, and is astonishingly difficult. In the conclusion to her paper, she remarks:

> The proofs are still "far longer and trickier than the algorithms and programs in question." ... The ratio 1 line of C for 500 lines of formal proof will certainly not convince Kahan.

In his notes, Kahan points out that the famous Fibonacci sequence that we discussed in **Section 2.7** on page 15 provides a nice way to generate test values for quadratic solvers. Each Fibonacci number is the sum of the preceding two, with the simple recurrence

$$F_0 = 0, \qquad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2}, \qquad \text{for } n = 2, 3, 4, \ldots,$$

The numbers grow exponentially, and $F_n \approx \phi F_{n-1}$ for large n. Here, $\phi = (1+\sqrt{5})/2 \approx 1.618$ is the golden ratio. The largest exactly representable F_n values are for $n = 36, 78, 93, 164$, and 343 in the five formats of extended IEEE 754 binary arithmetic.

The relevance of the Fibonacci numbers to testing solvers of quadratic equations is this relation:

$$F_{n-1}^2 - F_n F_{n-2} = (-1)^n.$$

We can therefore choose coefficients

$$A = F_n/2, \qquad B = F_{n-1}, \qquad C = F_{n-2}/2,$$

to get discriminants of the minimum nonzero magnitude, although, in practice, we scale the coefficients by a factor of two to keep them whole numbers. That scaling increases the discriminants to ±4. The roots are given by

$$x_k = \begin{cases} (-F_{n-1} \pm 1)/F_n, & \textit{when } n \textit{ is even,} \\ (-F_{n-1} \pm i)/F_n, & \textit{when } n \textit{ is odd.} \end{cases}$$

For large n, their real parts tend to $1/\phi = \phi - 1 \approx 0.618$, and their imaginary parts, $1/F_n$, are tiny.

We can mix in some randomness by scaling the coefficients with a factor S chosen like this:

$$\begin{aligned} M &= \text{random integer in } [\beta^{t-1}, \beta^t - 1], \\ &= 1/\epsilon + \texttt{URAND()} \times ((\beta - 1)/\epsilon - 1), \\ S &= \text{floor}(M/F_n). \end{aligned}$$

As F_n grows, the number of distinct S values drops, so the number of unique scale factors is limited when n is large, but $\mathcal{O}(\beta^t)$ different scale factors are possible when n is small.

Kahan further observed that a simple generalization of the Fibonacci sequence of the form

$$G_0 = 0, \quad G_1 = 1, \quad G_n = mG_{n-1} + G_{n-2}, \qquad \textit{for } n = 2, 3, 4, \ldots \textit{ and integer } m > 1,$$

quickly produces large numbers that have the same discriminant relation as the Fibonacci sequence, and can be used for test coefficients with F replaced by G. Varying the parameter m and randomization with M and S lets us produce many more random test values.

The file `tqert.c`, and its companions with the usual type suffixes, implement tests using numbers chosen according to those algorithms. The coefficients are used with different scaling factors and sign changes to produce further test values. The programs also check for the solver's correct handling of coefficients that are Infinity or NaN, and include Forsythe's hard cases with additional coefficient scaling over most of the floating-point range.

Extensive tests on several current CPU architectures, and the VAX, suggest that the functions in our `QERT()` family compute roots that agree with exact roots with a relative error no worse than 2.85 ulps with *round-to-nearest* arithmetic, which is acceptably accurate considering that the schoolbook formula for the roots has five or six opportunities for rounding error. Boldo's proof gives an upper bound of two ulps for the error in Kahan's discriminant computation, which differs from ours.

In practice, the results from the `QERT()` family are often correctly rounded, as this worst-case fragment of the test output for IEEE 754 32-bit decimal arithmetic shows:

```
Test 3: scaled Fibonacci-like: a = G[n], b = 2 * G[n-1], c = G[n-2]

Total tests of QERT(): 10179288 (1191744 >= 1/2 ulp)
Worst error        = 1.970 ulps
Average error      = 0.238 ulps
Variance           = 0.051 ulps
Standard deviation = 0.226 ulps
Root mean square   = 0.314 ulps
        a =  2465455.
        b =  2042448.
        c =  423007.0
        x1 = (-0.4142132, -0.001015200)
        e1 = (-0.4142132, -0.001015200)
        x2 = (-0.4142135,  0.001015200)
        e2 = (-0.4142132,  0.001015200)
```

The average error is small, and only about 12% of the results have errors larger than $\frac{1}{2}$ ulp. The last seven lines show the coefficients that produce the largest error, and the computed and exact roots.

16.4 Summary

Despite their schoolbook simplicity, robust solution of quadratic equations on a computer is a challenging problem that requires great care in programming, and access to higher-precision arithmetic. As Forsythe remarked, *School examples* do *factor with a frequency bewildering to anyone who has done mathematics outside of school* [For69a, pp. 144–145]. Real applications are much more demanding, and the roots are rarely simple integers. Extreme differences in coefficient magnitudes, and cases where one or more coefficients are zero, are common in practice. When the coefficients are determined from experimental measurements, or from computer approximations, it is essential to be aware of the error-magnification problem described in **Section 16.2** on page 471 that can produce huge changes in the roots from tiny changes in the coefficients. Such changes may reflect low-level details of the design of the floating-point arithmetic system, such as rounding behavior, or be caused by instruction reordering during compiler optimizations, or by inaccuracies in input and output conversions.

17 Elementary functions in complex arithmetic

> ONE OF THE MOST PROFOUND JOKES OF NATURE IS THE SQUARE ROOT OF MINUS ONE THAT PHYSICIST ERWIN SCHRÖDINGER PUT INTO HIS WAVE EQUATION WHEN HE INVENTED WAVE MECHANICS IN 1926. ... THE SCHRÖDINGER EQUATION DESCRIBES CORRECTLY EVERYTHING WE KNOW ABOUT THE BEHAVIOR OF ATOMS. IT IS THE BASIS OF ALL OF CHEMISTRY AND MOST OF PHYSICS. AND THAT SQUARE ROOT OF MINUS ONE MEANS THAT NATURE WORKS WITH COMPLEX NUMBERS AND NOT WITH REAL NUMBERS.[1]
>
> — FREEMAN DYSON
> *Birds and Frogs (2008 AMS Einstein Lecture).*

Since the invention of complex arithmetic about two centuries ago, mathematicians have learned how to extend functions from the real line to the complex plane. The field of *complex analysis* is now well understood, and mathematics courses in that area are commonly offered in the first years of undergraduate college eduction.

Most of the real functions that we consider in this book have companions in the complex domain, but the computational techniques that are required to handle them may be quite different. Computation of real functions can often be reduced to a smaller interval where a one-dimensional polynomial fit of low order provides an accurate representation of the function. Extending that idea to complex functions would require fits to polynomials in two variables that approximate a *surface* instead of a *line*, and achieving high accuracy with such fits is difficult.

Surface fits with bivariate polynomials are commonly used in computer visualization of three-dimensional objects, and since the 1960s, have seen increasing use in design of automobiles, ships, aircraft, and other products. Since the 1990s, computer animations have become practical for television and film production, and there is now a world-wide video-game industry whose computational needs have driven floating-point hardware development, especially in *graphics processing units* (GPUs). However, in all of those applications, accuracy beyond three to six decimal digits is unlikely to be needed, and the floating-point hardware in current GPUs provides only rough conformance to a *subset* of IEEE 754 arithmetic, and may support only the 32-bit format. Newer GPU designs offer the 64-bit format, but at the time of writing this, none provides longer formats. In this book, our goal is to approach machine precision, requiring up to 70 decimal digits for the longest supported data types.

Fortunately, for some of the complex elementary functions that we cover in this chapter, it is possible to express their real and imaginary parts separately in terms of real elementary functions that can be computed accurately, with only a few additional floating-point operations. Even with perfect rounding of the real elementary functions, the final results for the complex functions inevitably suffer a few rounding errors.

Unfortunately, there is a hidden problem that can cause catastrophic accuracy loss in small regions of the complex plane: *all four* of the basic operations of complex arithmetic are subject to subtraction loss. The tests in **Section 15.16** on page 458 show that bit loss is likely to be common until the quality of implementations of complex arithmetic is improved substantially. The simplest way to reduce that loss is to use higher intermediate precision. We can easily do that for the `float` data type, but some platforms offer nothing longer than the `double` type.

17.1 Research on complex elementary functions

Numerical analysts have paid much less attention to the computation of complex functions than to real functions. For example, the important journal *ACM Transactions on Mathematical Software (TOMS)* contains about 50 articles on

[1] In 1928, the British theoretical physicist Paul Dirac developed the first relativistic wave equation that successfully combined Einstein's Special Relativity with quantum mechanics, albeit for only a single particle. Positive and negative solutions of a square root in his equation led him to predict the existence of *holes*, or equivalently, positively charged electrons. In August 1932, 27-year-old American physicist Carl Anderson found the new particles in experiments, and called them *positrons*. Dirac shared the Nobel Prize in Physics with the Austrian theoretical physicist Erwin Schrödinger in 1933 for their work on the quantum theory of atoms. Anderson shared the Nobel Prize in Physics in 1936 with Austrian–American physicist Victor Hess for their discovery of the positron and cosmic rays.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_17

complex arithmetic among the 1500 published since the first issue in 1975. The more theoretical journal *Mathematics of Computation* has about 90 such articles among the more than 6700 published since 1943. In *BIT* and *Numerische Mathematik*, about one in 250 articles deals with complex arithmetic.

Among the TOMS articles are two on complex division [Ste85, Pri91], one on the complex inverse cosine and sine [HFT97], three on larger collections of complex elementary functions [HFT94, Neh07, Smi98], and just one on testing their accuracy [Cod93a]. Smith's paper discusses their computation with multiple-precision arithmetic, and Neher treats them with 64-bit interval arithmetic. An important recent paper in *Mathematics of Computation* investigates the accuracy of complex multiplication [BPZ07].

There is an extensive discussion of computational and mathematical issues in defining the elementary functions for complex arguments in the book *Common Lisp — The Language* [Ste90, Chapter 12].

17.2 Principal values

In the polar representation of complex numbers, $z = r\exp(\theta i)$, we can augment θ by any integer multiple of 2π without changing the value of z, because that operation corresponds to repeated traversals of a circle of radius $r = |z|$, bringing us back to the same point in the complex plane. Using the trigonometric expansion of the exponential of a pure imaginary value,

$$\exp(\alpha i) = \cos(\alpha) + \sin(\alpha) i, \qquad \text{for real } \alpha,$$

we can write that observation like this:

$$\begin{aligned}\exp\big((\theta + k(2\pi))i\big) &= \exp(\theta i)\exp(2k\pi i), \qquad \text{for } k = 0, \pm1, \pm2, \dots,\\ &= \exp(\theta i)\big(\cos(2k\pi) + \sin(2k\pi)i\big)\\ &= \exp(\theta i)(1 + 0i)\\ &= \exp(\theta i).\end{aligned}$$

Thus, although z is a *unique point* in the complex plane, it has *multiple representations* that are mathematically equivalent. When we compute a function of z, the different representations of z can give different answers. A simple example is the logarithm function, which we evaluate like this:

$$\begin{aligned}\log(z) &= \log(r\exp(\theta i))\\ &= \log(r\exp((\theta + k(2\pi))i))\\ &= \log(r) + (\theta + k(2\pi))i.\end{aligned}$$

The real part is always the same, but the imaginary part depends on our choice of k. The usual convention is to restrict the angle θ to the interval $[0, 2\pi)$ and set $k = 0$. That choice selects a unique value of the function, called the *principal value*. Textbooks and research literature sometimes use capitalized function names to indicate the principal value, but in programming languages that support complex arithmetic, library algorithms for complex functions are simply chosen to produce principal values, and the function names are not altered.

17.3 Branch cuts

Kahan's often-cited article *Branch Cuts for Complex Elementary Functions or Much Ado About Nothing's Sign Bit* [Kah87] appears in a conference-proceedings volume, and is recommended reading for reaching a better understanding of the subject of this chapter. Kahan discusses important issues in complex binary and decimal arithmetic, accurate computation of complex elementary functions, and the need for a signed zero. He also treats a feature of complex functions that we have not previously encountered — *branch cuts*.

We introduce branch cuts by analogies with functions in real arithmetic. Some otherwise well-behaved real functions have discontinuities along the real axis:

- The inverse hyperbolic tangent, shown in **Figure 12.6** on page 350, grows to $-\infty$ as x approaches -1 from the right, and to $+\infty$ as x moves toward $+1$ from the left. It is undefined in real arithmetic for arguments outside the interval $[-1, +1]$.

- The tangent, graphed in **Figure 11.3** on page 302, is a different example: $\tan(x)$ grows with increasing x until the function approaches an asymptote where it rises to $+\infty$. It then continues to grow from $-\infty$ as x passes the position of the asymptote. The discontinuities in $\tan(x)$ lie at odd multiples of $\frac{1}{2}\pi$.

- The *Heaviside step function* defined by

$$H(x) = \begin{cases} 0, & \textit{if } x < 0, \\ \text{arbitrary value in } [0,1], & \textit{if } x = 0, \\ 1, & \textit{if } x > 0, \end{cases}$$

  jumps from zero to one as x moves from left to right across the origin.

All of those kinds of behaviors are seen in complex functions, but they seem to occur more often than in real functions.

- In the complex plane, a point z_0 where the function $f(z)$ grows to infinity is called a *pole*. The coefficient of $1/(z - z_0)$ in a series expansion of $f(z)$ in powers of $(z - z_0)^k$ about that point is called the *residue*. We can illustrate that with a Mathematica session:

```
% math
In[1]:= Series[Tan[z], {z, Pi/2, 3}]

                      -Pi         -Pi     3
                     (--- + z)   (--- + z)
             1         2           2           -Pi     4
Out[1]= -(-------) + --------- + ---------- + O[--- + z]
          -Pi            3           45          2
          --- + z
           2

In[2]:= Residue[Tan[z], {z, Pi/2}]

Out[2]= -1
```

  Although we do not discuss them further in this book, residues are of utmost importance in complex analysis, and figure prominently in the integration of complex functions.

- The analogue of a finite jump on a line is a tear in a surface, and is called a *branch cut*. For example, the complex square root function, $\sqrt{z} = \sqrt{x + yi}$, and in general, the complex power function, z^a, for nonintegral values of a, have a branch cut in the imaginary part along the entire negative real axis as **Figure 17.1** on page 481 clearly shows. We can see that both numerically and symbolically in a short Maple session by fixing x and letting y approach zero from above and from below, and then asking for the mathematical limits of that process:

```
% maple

> Digits := 10:

> evalf(sqrt(-25 + 1.0e-10 * I));
                                     -10
                      0.1000000000 10    + 5.000000000 I

> evalf(sqrt(-25 + 1.0e-100 * I));
                                     -100
                      0.1000000000 10     + 5.000000000 I

> evalf(sqrt(-25 + 1.0e-10000000 * I));
                                  -10000000
                   0.1000000000 10          + 5.000000000 I
```

```
> evalf(sqrt(-25 - 1.0e-10000000 * I));
                                  -10000000
                   0.1000000000 10          - 5.000000000 I

> evalf(sqrt(-25 - 1.0e-100 * I));
                                     -100
                      0.1000000000 10     - 5.000000000 I

> evalf(sqrt(-25 - 1.0e-20 * I));
                                     -20
                      0.1000000000 10    - 5.000000000 I

> limit(sqrt(-25 + y * I), y = 0, left);
                                     -5 I

> limit(sqrt(-25 + y * I), y = 0, right);
                                      5 I
```

Mathematically, the last examples say that $\lim_{y\to\pm 0}\sqrt{-|x|+yi} = \pm\sqrt{|x|}i$. It is clear that the sign of zero is of considerable significance here. If the host floating-point system has only a positive zero, or a platform with IEEE 754 arithmetic has carelessly implemented library routines, or faulty code optimization or instruction generation, that lose the sign of zero, *there are likely to be computational anomalies in the neighborhood of the branch cut*. In the rest of this chapter, we often use explicit signs on zeros.

17.4 Software problems with negative zeros

This author's extensive experience with multiple compilers, operating systems, and CPU architectures shows that one cannot guarantee production of a true negative zero with C code like this:

```
static const double negzero = -0.0;
```

This two-step process does not always succeed either:

```
double negzero;
static const double zero = 0.0;
negzero = -zero;
```

Adding a `volatile` qualifier should fix the problem, but does not if the hardware negation instruction loses the sign:

```
static volatile double zero = 0.0;
negzero = -zero;
```

The safe way to get a negative zero is to use the sign-transfer function:

```
negzero = copysign(0.0, -1.0);
```

In IEEE 754 arithmetic, the expression $-1.0/$`infty()` generates a negative zero, but also sets floating-point exception flags.

Even after the required value has been generated, its sign may be invisible in output reports, because some library output routines may erroneously suppress the signs of zero, and also of NaNs. The mathcw library remedies those defects.

Signed floating-point zeros are uncommon in older architectures, although their bit patterns are usually representable. IBM System/360 treats a negative zero input operand as positive, but hardware generates only the positive form. The DEC PDP-11 and VAX architectures define a negative zero to be a *reserved operand* that causes a run-time instruction fault, even with `move` instructions; see **Appendix H.4** on page 956 for details. On the DEC PDP-10, a negative zero is an unnormalized value equal to -1, but it misbehaves in arithmetic instructions that expect normalized operands. The hardware instructions on those DEC architectures never generate negative zeros; they have to be created in software by compile-time storage initialization, or by run-time bit masking.

Table 17.1: Complex elementary function dependency tree. Notice that complex arithmetic joins trigonometric and hyperbolic functions into groups with common dependencies.

Function	Dependent functions
`cabs()`	`hypot()`
`carg()`	`atan2()`
`ccbrt()`	`cabs(), cbrt()`
`cexp()`	`cos(), exp(), sin()`
`clog()`	`cabs(), carg(), log()`
`csqrt()`	`cabs(), sqrt()`
`cpow()`	`cexp(), clog()`
`ccos()`	`cos(), cosh(), sin(), sinh()`
`csin()`	
`ctan()`	
`ccosh()`	
`csinh()`	
`ctanh()`	
`cacos()`	`clog(), csqrt()`
`casin()`	
`catan()`	
`cacosh()`	
`casinh()`	
`catanh()`	

17.5 Complex elementary function tree

Apart from the basic four arithmetic operations, the key low-level operations in the computation of the complex elementary functions are the absolute value and argument functions that are needed for conversion of complex numbers from Cartesian to polar form. Once those operations are available, we can compute roots and logarithms. The complex exponential needs the real cosine and sine. The complex trigonometric and hyperbolic families need the real trigonometric and hyperbolic cosine and sine. The inverse trigonometric and hyperbolic functions need only the complex logarithm and complex square root. Those relations are summarized in Table 17.1. Of particular interest is that cosines and sines are always needed in pairs, and for the same argument. The mathcw library exploits that by using the `sincos()` and `sinhcosh()` families that avoid unnecessary duplicate computations. For that reason, those combined functions ought to be part of standard libraries for all programming languages that offer complex arithmetic.

The key observation from Table 17.1 is that `cabs()`, `carg()`, `cexp()`, and `clog()` are needed for the computation of other elementary functions of complex arguments. The accuracy attainable for the remaining elementary functions is limited by the accuracy of those four functions, and of the functions that implement complex add, subtract, multiply, and divide.

17.6 Series for complex functions

We show many examples of Taylor-series expansions of real functions in this book. For all of the elementary functions, those expansions also apply for complex arguments, but in some cases, additional restrictions on those arguments are necessary. The general relations given throughout this chapter hold mathematically, but in most cases, they are computationally unsatisfactory when arguments are small, because they either lose digits, or they botch the sign of zero. Thus, for the complex elementary functions, summation of Taylor series is an essential element of the computations.

For the elementary functions of real arguments, we can often sum a few terms, and handle zero arguments separately to guarantee correct handling of signed zeros. For the functions of complex arguments, we always have a pair of numbers to deal with, and both must be treated carefully. For that reason, for most functions, we adopt a uniform approach of summing series of one, two, four, eight, or sixteen terms, because all that we need to compute the associated cutoffs is the real square-root function. The one-term case is included so as to handle the sign of zero properly.

The sixteen-term summation extends the range of z that can be handled, and is longer than we normally use for the real functions. With complex arithmetic, the alternative to series summation is evaluation of transcendental functions, often of complex arguments, so the series approach is faster.

For real functions, we select truncated series by comparing $|x|$ with a cutoff value chosen to ensure that the sum of the discarded terms would not change the last stored digit. For complex functions, we could use $|z|$ in that selection, but that involves the hypotenuse function (see **Section 8.2** on page 222 and **Section 8.3** on page 227). Instead, we use the cheaper *one-norm*, which produces lower and upper bounds on $|z|$:

$$\begin{aligned} z &= x + yi, && \textit{for real } x \textit{ and } y, \\ ||z||_1 &= |x| + |y|, && \textit{one-norm}, \\ ||z||_2 &= |z|, && \textit{complex absolute value}, \\ &= \sqrt{x^2 + y^2}, && \textit{two-norm, or Euclidean norm, or hypotenuse}, \\ \tfrac{1}{2}||z||_1 \le |z| &\le ||z||_1, && \textit{by geometrical observation.} \end{aligned}$$

Equality is only possible in the last relation if $z = 0$, or if z has only one nonzero component. The relation is clearly true from inspection of the sides of a right triangle. Using half the one-norm also prevents the overflow that is possible in the two-norm, but means that we must incorporate an outer factor of a half in the cutoffs computed in the one-time initialization blocks.

If we sum the first n terms $c_0 + c_1 z + \cdots + c_{n-1} z^{n-1}$, then we want the discarded sum $c_n z^n + c_{n+1} z^{n+1} + \cdots$ to add up to less than half the machine epsilon, ϵ, if all terms are positive, or half the little machine epsilon, ϵ/β, if term signs may differ. If we further assume that successive terms of the Taylor series decrease by at least a factor of two, then the discarded sum is no larger than $c_n z^n (1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots) \le 2 c_n z^n$. The cutoff condition then is that $c_n z^n / c_0 < \frac{1}{4}\epsilon/\beta$, or

$$\begin{aligned} u &= \tfrac{1}{4}\epsilon/\beta, \\ z_n &< \big((c_0/c_n)u\big)^{1/n}, \\ \texttt{ZCUT}_n &= \tfrac{1}{2} z_n. \end{aligned}$$

The cutoff value in the last equation is to be compared with $\frac{1}{2}||z||_1$.

17.7 Complex square root

The complex square root is graphed in **Figure 17.1** on the next page, and is the first function that we consider in this chapter with a branch cut. It is conventionally defined by this relation:

$$\sqrt{z} = \mathbf{exp}(\tfrac{1}{2}\,\mathbf{log}(z)), \qquad \textit{definition of complex square root.}$$

In this section, we show how to compute it more accurately, and faster, than via its definition in transcendental functions.

The complex square root and complex logarithm both have a branch cut on the surface of the imaginary part along the negative real axis.

The complex square root has this important symmetry relation under conjugation:

$$\begin{aligned} z &= x + yi, && \textit{for real } x \textit{ and } y, \\ z^\star &= x - yi, && \textit{conjugation operation}, \\ \sqrt{z^\star} &= (\sqrt{z})^\star, && \textit{mathematical notation}, \\ \texttt{csqrt(conj(z))} &= \texttt{conj(csqrt(z))}, && \textit{computer software notation.} \end{aligned}$$

It also satisfies a critical reciprocal symmetry relation:

$$\sqrt{1/z} = 1/\sqrt{z},$$

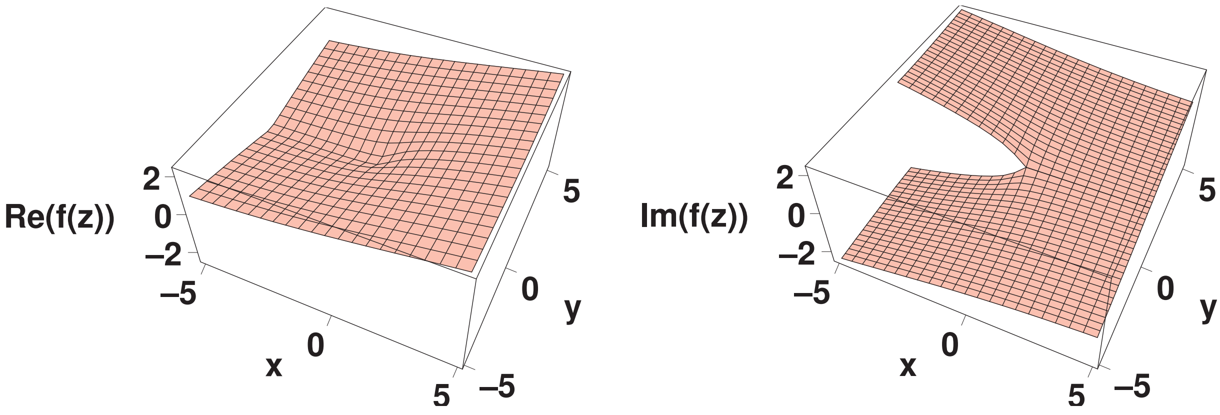


Figure 17.1: Real (left) and imaginary (right) surfaces of `csqrt(z)`, the complex square root function defined by $\sqrt{z} = \sqrt{x+yi} = z^{1/2}$. There is a cusp along the entire negative real axis in the real component, and a *branch cut* along that same axis in the imaginary component.

$$\texttt{csqrt(1.0 / z)} \approx \texttt{1.0 / csqrt(z)}.$$

The reciprocal relation holds only approximately in computer software because of rounding, and possible subtraction loss, in the two division operations.

We can relate the real and imaginary parts of the complex square root to the parts of $z = x + yi$ like this [Str59, Fri67]:

$$\begin{aligned}
\sqrt{z} &= a + bi, && \textit{for real } a \textit{ and } b, \\
|\sqrt{z}|^2 &= |z| \\
&= a^2 + b^2, \\
z &= (a+bi)^2 \\
&= (a^2 - b^2) + 2abi, \\
x &= a^2 - b^2, \\
y &= 2ab.
\end{aligned}$$

We now expand x in two ways and solve for the unknowns:

- Add and subtract b^2, and solve for b and then a:

$$\begin{aligned}
x &= (a^2 + b^2) - 2b^2 \\
&= |z| - 2b^2, \\
\boldsymbol{b} &= \boldsymbol{\pm\sqrt{\tfrac{1}{2}(-x + |z|)}}, \\
\boldsymbol{a} &= \boldsymbol{y/(2b)}.
\end{aligned}$$

- Add and subtract a^2, and solve for a and then b:

$$\begin{aligned}
x &= 2a^2 - a^2 - b^2 \\
&= 2a^2 - |z|, \\
\boldsymbol{a} &= \boldsymbol{\pm\sqrt{\tfrac{1}{2}(x + |z|)}}, \\
\boldsymbol{b} &= \boldsymbol{y/(2a)}.
\end{aligned}$$

Table 17.2: Special cases for the complex square root function, `csqrt(z)`, according to Kahan [Kah87] and the C99 Standard [C99, TC3 §G.6.4.2, page 479]. The variable f is any *finite* value, g is any *positive-signed* finite value, and x is any floating-point value, including Infinity and NaN.

z	$\sqrt{z}$	**Remark**
`conj(z)`	`conj(csqrt(z))`	conjugation symmetry relation
$-g \pm 0i$	$+0 \pm \sqrt{g}i$	
$x \pm \infty i$	$+\infty \pm \infty i$	unusual; Kahan: set *invalid* flag if x is a NaN
$+\infty \pm fi$	$+\infty \pm 0i$	
$+\infty \pm \mathrm{NaN}i$	$+\infty \pm \mathrm{NaN}i$	unusual
$-\infty \pm fi$	$+0 \pm \infty i$	
$-\infty \pm \mathrm{NaN}i$	$\mathrm{NaN} \pm \infty i$	unusual: sign of imaginary part unspecified
$\mathrm{NaN} + fi$	$\mathrm{NaN} + \mathrm{NaN}i$	C99: optionally raise *invalid* exception
$f + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	C99: optionally raise *invalid* exception
$\mathrm{NaN} + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	

To avoid subtraction loss, and get the conventional positive sign of the square root for nonnegative real arguments, we use the formulas like this:

$$\begin{aligned}
a &= +\sqrt{\tfrac{1}{2}(|x| + |z|)}, && \textit{when } x \geq +0,\\
b &= y/(2a);\\
b &= \begin{cases} +\sqrt{\tfrac{1}{2}(|x| + |z|)}, \\ -\sqrt{\tfrac{1}{2}(|x| + |z|)}, \end{cases} && \begin{aligned}&\textit{when } x \leq -0 \textit{ and } y \geq +0,\\ &\textit{when } x \leq -0 \textit{ and } y \leq -0,\end{aligned}\\
a &= y/(2b).
\end{aligned}$$

Notice that for negative x, the sign of b is that of y, guaranteeing correct behavior in the neighborhood of the branch cut. Also, the conjugation symmetry rule clearly holds exactly, because the replacement $y \to -y$ forces the change $b \to -b$.

Provided that the real square root function and complex absolute value functions are accurate, as ours are, then we expect no more than *six* rounding errors in each component of the computed complex square root, and most commonly, only *two* ($\beta = 2$) or *three* ($\beta > 2$) per component.

Although the formulas for a and b look simple, they are subject to premature overflow and underflow, and cannot be used directly without further scaling. We show how to make that adjustment shortly.

Kahan recommends, and the C99 Standard requires, particular handling of special values, as summarized in Table 17.2. Some of them are not obvious, particularly the cases where one argument component is a NaN, yet the output result is not NaN ± NaNi. The C99 rule quoted in Chapter 15 on page 441 requires that a complex number with one infinite component be treated as Infinity, even if the other component is a NaN.

At the origin, our formulas produce

$$\begin{aligned}
\sqrt{+0 \pm 0i} &= +0 \pm (0/0)i,\\
\sqrt{-0 \pm 0i} &= -0 \pm (0/0)i.
\end{aligned}$$

The value of the real part is in conformance with the IEEE 754 requirement that $\sqrt{-0} = -0$, but the imaginary part is a NaN. Unfortunately, the second case conflicts with the requirement of the second line of Table 17.2, which demands that $\sqrt{-0 \pm 0i} = +0 \pm 0i$. We therefore need to check for the special case $z = 0$, and handle it separately.

Kahan's code uses `scalb()` to scale the computation away from the underflow and overflow regions. We take a different and faster approach that accomplishes the same goal: compute $w = s^{-2}z$, where s is a suitable power of the base so that the scaling is *exact*, find the complex square root of w, and then unscale to recover $\sqrt{z} = s\sqrt{w}$.

Our code for complex square root using the complex-as-real representation is longer than the simple formulas for a and b would suggest:

```
void
CXSQRT(fp_cx_t result, const fp_cx_t z)
{   /* complex square root: result = sqrt(z) */
    fp_t x, y;

    x = CXREAL_(z);
    y = CXIMAG_(z);

    if (ISINF(y))
    {
        if (ISNAN(x))
            (void)QNAN("");             /* Kahan requires, C99 does not */

        CXSET_(result, COPYSIGN(y, ONE), y);
    }
    else if (ISINF(x))
    {
        if (x > ZERO)
            CXSET_(result, x, ISNAN(y) ? y : COPYSIGN(ZERO, y));
        else
            CXSET_(result, ISNAN(y) ? y : ZERO, COPYSIGN(x, y));
    }
    else if (ISNAN(x))
        CXSET_(result, x, COPYSIGN(x, y));
    else if (ISNAN(y))
        CXSET_(result, y, y);
    else
    {
        fp_t a, b, s, s_inv_sq, t, u, v;
        fp_cx_t w;
        static const fp_t B_SQ = (fp_t)(B * B);
        static const fp_t B_INV = FP(1.) / (fp_t)B;
        static const fp_t B_INV_SQ = FP(1.) / (fp_t)(B * B);

        if ( (QABS(x) > ONE) || (QABS(y) > ONE) )
        {
            s = (fp_t)B;
            s_inv_sq = B_INV_SQ;
        }
        else
        {
            s = (fp_t)B_INV;
            s_inv_sq = B_SQ;
        }

        CXSCALE_(w, z, s_inv_sq);       /* w = z / s**2 */
        u = CXREAL_(w);
        v = CXIMAG_(w);
        t = SQRT(HALF * (QABS(u) + CXABS(w)));

        if (t == ZERO)                  /* then x = +/-0, y = +/-0 */
            CXSET_(result, ZERO, y);
        else if (u >= ZERO)
        {
            a = t;
            b = v / (a + a);
            CXSET_(result, s * a, s * b);
```

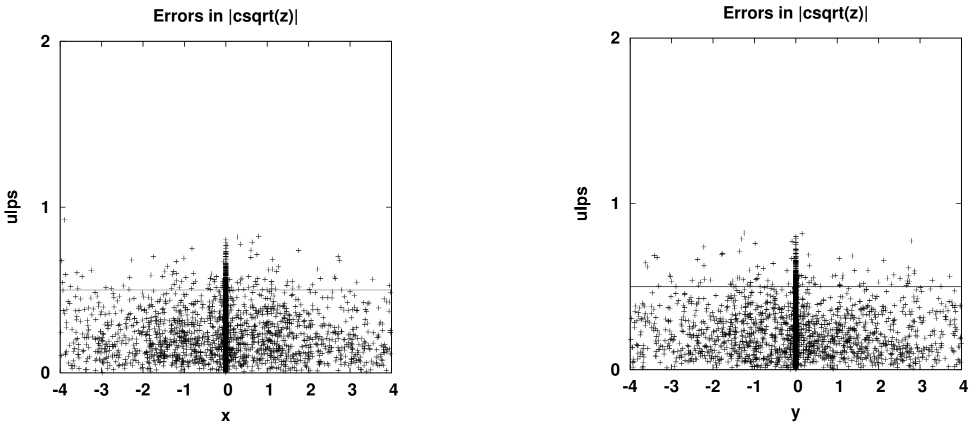


Figure 17.2: Errors in the `csqrt()` function. The error is measured in the function magnitude, rather than in its individual components, because the relative error in the smaller component can be large. Although the horizontal range is limited, plots over a wider range are similar to these. The apparent spike near zero is an artifact caused by using random arguments taken from a logarithmic distribution.

```
            }
            else
            {
                b = COPYSIGN(t, v);
                a = v / (b + b);
                CXSET_(result, s * a, s * b);
            }
        }
    }
```

Because the case $z = 0$ is expected to be rare, we do not test both parts for a zero value: instead, we check the scaled real value $t = \sqrt{(|u| + |w|)/2}$.

The special-case tests require run-time overhead on every call, but are unavoidable, because Infinity and NaN inputs to the normal formulas tend to produce NaN outputs.

The companion function that provides a native complex square root is a simple wrapper that looks like this:

```
fp_c_t
CSQRT(fp_c_t z)
{   /* complex sqrt: return sqrt(z) */
    fp_cx_t zz, result;

    CTOCX_(zz, z);
    CXSQRT(result, zz);

    return (CXTOC_(result));
}
```

Most of the native complex functions in the mathcw library look much like that one, so we omit their code in the remainder of this chapter.

In order to show the errors of the computed function values, we could show the errors in the real and imaginary components of the function as surfaces on the x–y plane. However, point plots of those surfaces are too cluttered to be useful. Also, when the magnitudes of the function-value components differ sharply, the smaller is likely to have a large relative error that cannot be removed without working in higher precision. In this chapter, we therefore display the errors in $|f(z)|$ as two plots, one for the real component of the argument, and the other for the imaginary component. Although the two plots often look alike, that is not always the case. Figure 17.2 shows the measured

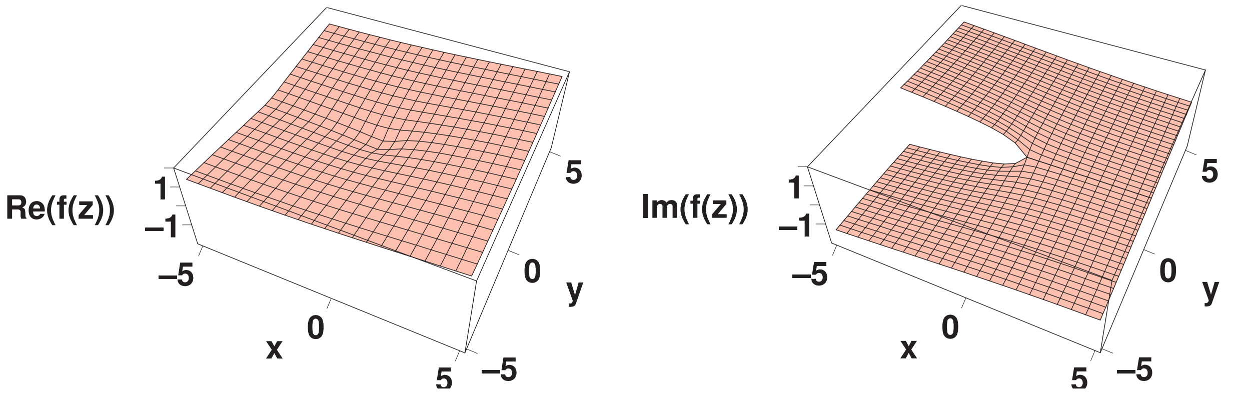


Figure 17.3: Real (left) and imaginary (right) surfaces of $\mathrm{cbrt}(z) = \mathrm{cbrt}(x + yi) = z^{1/3}$, the complex cube root function. There is a cusp along the entire negative real axis in the real component, and a *branch cut* along that same axis in the imaginary component.

errors in `csqrt(z)` for the IEEE 754 64-bit format. The plots provide a test of that function, as well as of the complex-as-real function `cxsqrt(result,z)` that serves as the kernel of the complex function. Similar plots are expected for other data sizes, and also for decimal arithmetic.

17.8 Complex cube root

The complex cube root is plotted in **Figure 17.3**, and its surfaces are similar to those for the square root. Both functions have a branch cut along the negative real axis, and they inherit that feature from their definitions in terms of the exponential and logarithm. For the cube root, we have the definition

$$\sqrt[3]{z} = \exp(\tfrac{1}{3}\log(z))$$

and the symmetry relations

$$\begin{aligned}
\sqrt[3]{z^\star} &= (\sqrt[3]{z})^\star, && \textit{mathematical notation,}\\
\texttt{ccbrt(conj(z))} &= \texttt{conj(ccbrt(z))}, && \textit{computer software notation,}\\
\sqrt[3]{-z} &= -\sqrt[3]{z}, &&\\
1/\sqrt[3]{z} &= \sqrt[3]{1/z}, &&\\
\texttt{ccbrt(1.0 / z)} &\approx \texttt{1.0 / ccbrt(z)}. &&
\end{aligned}$$

The complex cube root, `ccbrt(z)`, is not required by C99, but we provide it because we have its real counterpart, `cbrt(x)`. The expansion of the Cartesian form of the square root in **Section 17.7** on page 481 led to an effective algorithm for its computation, but the technique does not readily work for the cube root. Instead, we derive a straightforward algorithm from the polar form:

$$\begin{aligned}
z &= r\exp(\theta i), && \textit{for real } r \textit{ and } \theta,\\
\sqrt[3]{z} &= \exp\left(\tfrac{1}{3}\log(r\exp(\theta i))\right) &&\\
&= \exp\left(\tfrac{1}{3}(\log(r) + \theta i)\right) &&\\
&= \sqrt[3]{r} \times \left(\cos(\tfrac{1}{3}\theta) + \sin(\tfrac{1}{3}\theta)i\right). &&
\end{aligned}$$

We find the desired handling of special cases summarized in **Table 17.3** on the next page by analogies with those of the square-root function, and from numerical and symbolic experiments in symbolic-algebra systems.

Table 17.3: Special cases for the complex cube root function, `ccbrt(z)`. The variable f is any *finite* value, g is any *positive-signed* finite value, and x is any floating-point value, including Infinity and NaN.

z	**cbrt**(z)	**Remark**
`conj(z)`	`conj(ccbrt(z))`	conjugation symmetry relation
$+0 \pm 0i$	$+0 \pm 0i$	
$-0 \pm 0i$	$+0 \pm 0i$	
$x \pm \infty i$	$+\infty \pm \infty i$	unusual; optionally raise *invalid* exception if x is a NaN
$+\infty \pm fi$	$+\infty \pm 0i$	
$+\infty \pm \mathrm{NaN}i$	$+\infty \pm \mathrm{NaN}i$	unusual
$-\infty \pm \mathrm{NaN}i$	$\mathrm{NaN} \pm \infty i$	unusual: sign of imaginary part unspecified
$\mathrm{NaN} + fi$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$f + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$\mathrm{NaN} + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	

There are three possible cube roots for any argument z, and when z is a negative real value, one of the roots is negative and real, and the other two are complex conjugates. For example,

$$\begin{aligned}\sqrt[3]{-1} &= \{-1, \cos(\tfrac{1}{3}\pi) + \sin(\tfrac{1}{3}\pi)i, \cos(\tfrac{1}{3}\pi) - \sin(\tfrac{1}{3}\pi)i\} \\ &= \{-1, \tfrac{1}{2}\sqrt{3} + \tfrac{1}{2}i, \tfrac{1}{2}\sqrt{3} - \tfrac{1}{2}i\}.\end{aligned}$$

For $\sqrt[3]{-|x| \pm 0i}$, our formula produces the complex root $-\sqrt[3]{|x|}(\frac{1}{2}\sqrt{3} \pm \frac{1}{2}i)$ as the *principal value*, instead of the simpler real root, $-\sqrt[3]{|x|}$.

Because the root functions are multivalued, taking the n-th root of the n-th power does not in general recover the starting value. We can see that in a Maple session:

```
% maple
> z := 1 + 2*I:
> evalf((z**(1/3))**3);
                                     1. + 2. I
> zzz := z**3;
                                    -11 - 2 I
> evalf(zzz**(1/3));
                              1.232050808 - 1.866025404 I
> evalf((zzz**(1/3))**3);
                                    -11. - 2. I
> solve('z'**3 = -11 - 2*I, 'z');
                1/2          1/2              1/2          1/2
    -1/2 - I + 3    - 1/2 I 3   , -1/2 - I - 3    + 1/2 I 3    , 1 + 2 I
> evalf(%);
  1.232050808 - 1.866025404 I, -2.232050808 - 0.1339745960 I, 1. + 2. I
```

Here, the principal-value root returned by the cube-root operation differs from the simpler root $1 + 2i$ that we started with. Thus, *the expression* $(z^n)^{1/n}$ *cannot be simplified to* z *when* z *is complex.*

Our code for the complex cube root is a straightforward transcription of our transcendental formula, with special case handling to satisfy the requirements of Table 17.3, and exact scaling to avoid premature overflow in $|z|$:

```
void
CXCBRT(fp_cx_t result, const fp_cx_t z)
{   /* complex cube root: result = cbrt(z) */
    fp_t x, y;

    x = CXREAL_(z);
    y = CXIMAG_(z);
    if (ISINF(y))
    {
        if (ISNAN(x))                   /* set invalid flag */
            (void)QNAN("");

        CXSET_(result, COPYSIGN(y, ONE), y);
    }
    else if (ISINF(x))
    {
        if (x > ZERO)
            CXSET_(result, x, ISNAN(y) ? y : COPYSIGN(ZERO, y));
        else
            CXSET_(result, ISNAN(y) ? y : ZERO, COPYSIGN(x, y));
    }
    else if (ISNAN(x))
        CXSET_(result, x, COPYSIGN(x, y));
    else if (ISNAN(y))
        CXSET_(result, y, y);
    else
    {
        fp_t c_3, r, r_3, s, s_3, s_inv_cube, t_3;
        fp_cx_t w;
        static const fp_t B_CUBE = (fp_t)(B * B * B);
        static const fp_t B_INV = FP(1.) / (fp_t)B;
        static const fp_t B_INV_CUBE = FP(1.) / (fp_t)(B * B * B);

        if ( (QABS(x) > ONE) || (QABS(y) > ONE) )
        {
            s = (fp_t)B;
            s_inv_cube = B_INV_CUBE;
        }
        else
        {
            s = (fp_t)B_INV;
            s_inv_cube = B_CUBE;
        }

        CXSCALE_(w, z, s_inv_cube);     /* w = z / s**3 */
        r = CXABS(w);
        t_3 = CXARG(w) / THREE;
        SINCOS(t_3, &s_3, &c_3);
        r_3 = s * CBRT(r);
        CXSET_(result, r_3 * c_3, r_3 * s_3);
    }
}
```

The function family `CCBRT(z)` is provided by a short interface to the code for `CXCBRT(z)`, similar to that for `CSQRT(z)`; we therefore omit its code.

The measured errors in the IEEE 754 64-bit function `ccbrt(z)` are graphed in Figure 17.4 on the next page, and the remarks in the caption for the square function in Figure 17.2 on page 484 apply equally well to the cube root.

17.9 Complex exponential

The complex exponential function is graphed in Figure 17.5 on page 489. The function is smooth and defined everywhere on the complex plane: there are neither branch cuts nor poles.

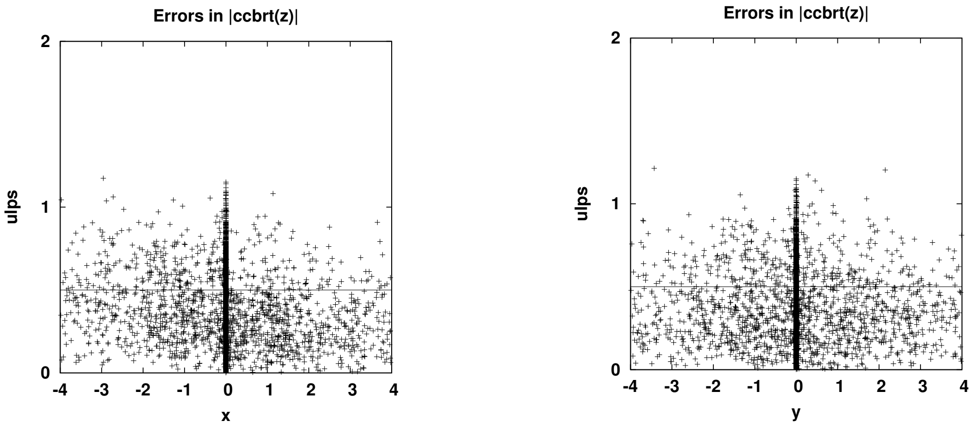


Figure 17.4: Errors in the `ccbrt()` function.

The complex exponential function satisfies an important symmetry relation under conjugation, as well as the usual reciprocation, argument sum and product, and power relations:

$$\begin{aligned}
w &= u + vi, && \textit{for real } u \textit{ and } v, \\
z &= x + yi, && \textit{for real } x \textit{ and } y, \\
z^{\star} &= x - yi, && \textit{conjugation operation}, \\
\exp(z^{\star}) &= \big(\exp(z)\big)^{\star}, && \textit{mathematical notation}, \\
\texttt{cexp(conj(z))} &= \texttt{conj(cexp(z))}, && \textit{computer software notation}, \\
\exp(-z) &= 1/\exp(z), && \\
\exp(w+z) &= \exp(w)\exp(z), && \\
\exp(wz) &= \big(\exp(w)\big)^{z} && \\
&= \big(\exp(z)\big)^{w}, && \\
w^{z} &= \mathbf{exp}\,\big(z\,\mathbf{log}(w)\big), && \textit{definition of complex power.}
\end{aligned}$$

For the complex exponential function, the Taylor series

$$\mathrm{cexp}(z) = 1 + z + z^2/2! + z^3/3! + \cdots + z^k/k! + \cdots$$

converges rapidly, and provides a suitable method for handling small arguments inline. To facilitate getting the code right, two Maple files, `polyfma.map` and `cpolyfma.map` in the `maple` subdirectory, provide functions for converting a truncated series to C code that can be used directly for real and complex-as-real data types.

The exponential function can be decomposed into real and imaginary parts by expanding the exponential of the imaginary part in trigonometric functions:

$$\begin{aligned}
z &= x + yi, && \textit{for real } x \textit{ and } y, \\
\exp(z) &= \exp(x + yi) && \\
&= \exp(x)\exp(yi) && \\
&= \exp(x)\big(\cos(y) + \sin(y)i\big) && \\
&= Ec + Esi, && \\
E &= \exp(x), && \\
c &= \cos(y), && \\
s &= \sin(y). &&
\end{aligned}$$

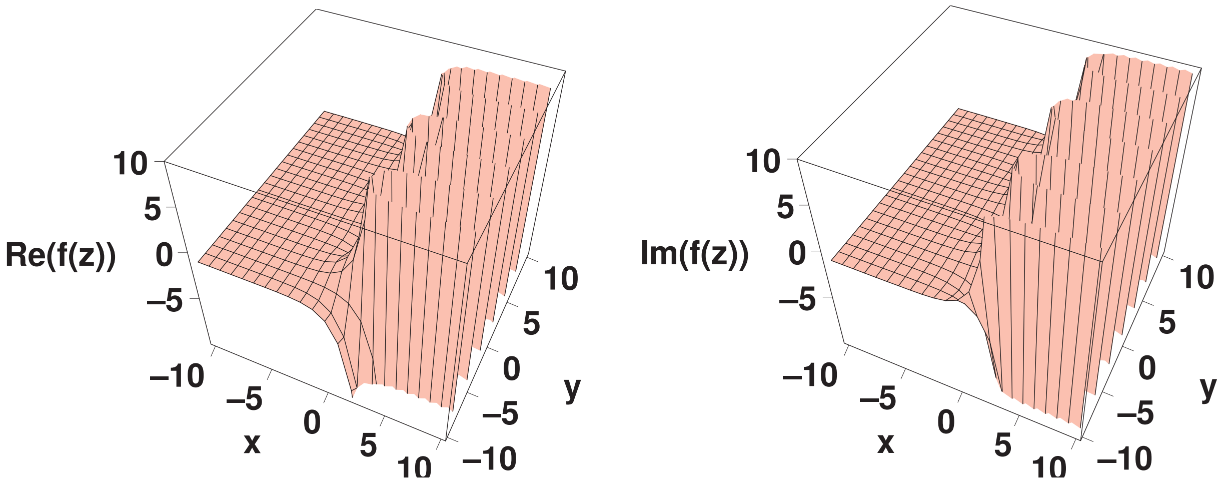


Figure 17.5: Real (left) and imaginary (right) surfaces of `cexp(z)`, the complex exponential function defined by $\exp(z) = e^z = \exp(x + yi)$.

The real exponential occurs in both real and imaginary parts of the result, scaled by the real cosine and sine, each of which lies in $[-1, +1]$. Thus, both components grow with increasing $|z|$, and the trigonometric functions must be accurate over the full range of y. The mathcw library guarantees exact trigonometric argument reduction to ensure that accuracy, but on those many platforms where native libraries are less careful about argument reduction, the complex exponential soon returns nonsensical results. On such systems, moving to higher precision is unlikely to improve the accuracy of argument reduction, so programmers need to be aware that `cexp(z)` may be useless for even moderately sized imaginary parts.

The C99 Standard mandates the handling of the special cases summarized in **Table 17.4** on the following page.

Our code for the complex exponential function is based on the complex-as-real representation. Testing shows that direct programming of the formula involving real functions does not produce the required special values, so we are forced to clutter the code with explicit handling of Infinity and NaN values to obey the Standard, and we also include Taylor-series summation to handle small arguments accurately and quickly.

The final code looks like this:

```
/* NB: r = x*y + z, but z is real */
#define CXFMA(r,x,y,z) { CXMUL(r, x, y); CXREAL_(r) += z; }

void
CXEXP(fp_cx_t result, const fp_cx_t z)
{   /* complex exponential: result = exp(z) */
    fp_t c, e, q, s, x, y;

    x = CXREAL_(z);
    y = CXIMAG_(z);

    if (ISINF(x))
    {
        if (x >= ZERO)
        {
            if (ISNAN(y))
                CXSET_(result, x, y);
            else if (ISINF(y))
                CXSET_(result, x, QNAN(""));
            else if (y == ZERO)
```

Table 17.4: Special cases for the complex exponential function, `cexp(z)`, according to the C99 Standard [C99, TC3 §G.6.3.1, page 478]. The value f is any *finite* value, and h is any *nonzero* finite value.

z	$\exp(z)$	Remark
`conj(z)`	`conj(cexp(z))`	conjugation symmetry relation
$\pm 0 \pm 0i$	$1 \pm 0i$	
$f \pm \infty i$	$\mathrm{NaN} + \mathrm{NaN}i$	raise *invalid* exception
$f \pm \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$+\infty \pm 0i$	$+\infty \pm 0i$	
$-\infty \pm fi$	$+0\cos(f) + 0\sin(f)i$	
$+\infty \pm hi$	$+\infty\cos(h) \pm \infty\sin(h)i$	
$-\infty \pm \infty i$	$\pm 0 \pm 0i$	result signs unspecified
$+\infty \pm \infty i$	$\pm\infty \pm \mathrm{NaN}i$	raise *invalid* exception; result real sign unspecified
$-\infty \pm \mathrm{NaN}i$	$\pm 0 \pm 0i$	result signs unspecified
$+\infty \pm \mathrm{NaN}i$	$\pm\infty \pm \mathrm{NaN}i$	result real sign unspecified
$\mathrm{NaN} \pm 0i$	$\mathrm{NaN} \pm 0i$	
$\mathrm{NaN} \pm hi$	$\mathrm{NaN} \pm \mathrm{NaN}i$	optionally raise *invalid* exception
$\mathrm{NaN} \pm \mathrm{NaN}$	$\mathrm{NaN} \pm \mathrm{NaN}i$	

```
                CXSET_(result, x, y);
            else
                CXSET_(result, x, COPYSIGN(x,y));
        }
        else
            CXSET_(result, ZERO, COPYSIGN(ZERO, y));
    }
    else if (ISINF(y))
    {
        q = QNAN("");                   /* must raise invalid f.p. exception */
        CXSET_(result, COPYSIGN(q, ONE), COPYSIGN(q, y));
    }
    else if (ISNAN(x))
    {
        if (y == ZERO)
            CXSET_(result, x, y);
        else
            CXSET_(result, x, COPYSIGN(x, y));
    }
    else if (ISNAN(y))
    {
        q = QNAN("");                   /* must raise invalid f.p. exception */
        CXSET_(result, COPYSIGN(q, ONE), COPYSIGN(q, y));
    }
    else    /* exp(a + b*i) = exp(a) * cos(b) + exp(a) * sin(b) * i */
    {
        fp_cx_t sum;
        fp_t half_z1;
        static fp_t ZCUT_1, ZCUT_2, ZCUT_4, ZCUT_8, ZCUT_16;
        static int do_init = 1;

        if (do_init)
        {
            fp_t u;
            u = FOURTH * FP_T_EPSILON / (fp_t)B;

            /* ZCUT_n = (1/2) * ((n + 1)! * u)**(1/n) */

            ZCUT_1  = u;
            ZCUT_2  = HALF * SQRT(FP(6.0) * u);
            ZCUT_4  = HALF * SQRT(SQRT(FP(120.0) * u));
            ZCUT_8  = HALF * SQRT(SQRT(SQRT(FP(362880.0) * u)));
            ZCUT_16 = HALF * SQRT(SQRT(SQRT(SQRT(FP(355687428096000.0) * u))));
            do_init = 0;
        }

        half_z1 = CXHALF1NORM_(z);

        if (half_z1 < ZCUT_1)
            CXSET_(result, ONE + x, y);
        else if (half_z1 < ZCUT_2)
        {   /* 2-term Taylor series: exp(z) - 1 = z + z**2 / 2! */
            CXSCALE_(sum, z, HALF);
            CXMUL(sum, sum, z);
            CXADD_(sum, z, sum);
            CXADDR_(result, sum, ONE);
        }
        else if (half_z1 < ZCUT_4)
        {   /* 4-term Taylor series */
            CXSCALE_(sum, z, FP(1.0) / FP(24.0));
            CXADDR_(sum, sum, FP(1.0) / FP(6.0));
            CXFMA(sum, sum, z, FP(1.0) / FP(2.0));
            CXMUL(sum, sum, z);
            CXMUL(sum, sum, z);
            CXADD_(sum, z, sum);
            CXADDR_(result, sum, ONE);
        }
        else if (half_z1 < ZCUT_8)
        {   /* 8-term Taylor series */
            /* ... code omitted ... */
        }
        else if (half_z1 < ZCUT_16)
        {   /* 16-term Taylor series */
            /* ... code omitted ... */
        }
        else
        {
            SINCOS(y, &s, &c);
            e = EXP(x);
            CXSET_(result, e * c, e * s);
        }
    }
}
```

The CXFMA() macro looks like a fused multiply-add operation, but it is implemented with separate operations, and adds the final *real* argument inline.

We cannot avoid computing the trigonometric functions when the real exponential underflow or overflows, because their signs are needed to determine the signs of the components of the final result: they may be ± 0 or $\pm\infty$.

In older floating-point systems without Infinity or NaN, the result of the real exponential should be capped at the largest representable number when the true result would overflow. Both components of the final result are then large but inaccurate. To better support such systems, we could supply this simpler version of the code (excluding the lengthy Taylor-series blocks):

```
void
CXEXP(fp_cx_t result, const fp_cx_t z)  /* pre-IEEE-754 ONLY! */
{   /* complex exponential: result = exp(z) */
    fp_t c, E, s;

    SINCOS(CXIMAG_(z), &s, &c);
    E = EXP(CXREAL_(z));

    if (E == FP_T_MAX)
        CXSET_(result, COPYSIGN(FP_T_MAX, c), COPYSIGN(FP_T_MAX, s));
    else
        CXSET_(result, E * c, E * s);
}
```

We have *not* done that in the mathcw library, because complex arithmetic is a new feature in C99, and unlikely to be supported on old platforms, even those that now run only in simulators.

The function family `CEXP(z)` is the usual simple wrapper around a call to `CXEXP(z)`, and thus, not shown here.

The measured errors in the IEEE 754 64-bit function `cexp(z)` are plotted in **Figure 17.6** on the next page. The bottom pair of graphs shows the behavior on a logarithmic argument scale, from which it is evident that Taylor-series summation provides correctly rounded function values.

17.10 Complex exponential near zero

The C99 Standard does not specify a complex version of `expm1(x)` for computing the difference $\exp(x) - 1$ accurately when $|x|$ is small. We remedy that unfortunate omission by providing our own implementation of the function family `CEXPM1(z)`.

Outside the Taylor-series region, we use the relation of that function to the hyperbolic tangent (see **Section 10.2** on page 273):

$$\operatorname{cexpm1}(z) = 2\tanh(z/2)/\big(1 - \tanh(z/2)\big).$$

The argument cutoff for the longest truncated series ensures that the hyperbolic tangent is never called for the IEEE 32-bit and 64-bit formats.

The code for the complex exponential, minus one, is similar to that shown for `CEXP()` in the preceding section, so we show only significant fragments here:

```
void
CXEXPM1(fp_cx_t result, const fp_cx_t z)
{   /* complex exponential minus one: result = cexp(x) - 1 */
    fp_t x, y;

    x = CXREAL_(z);
    y = CXIMAG_(z);

    if (ISINF(x) && ISNAN(y))
        CXSET_(result, COPYSIGN(x, 1.0), y);
    else if (ISNAN(x) && ISINF(y))
        CXSET_(result, COPYSIGN(y, 1.0), x);
    else
    {
        fp_cx_t sum;
        fp_t half_z1;
        static fp_t ZCUT_1, ZCUT_2, ZCUT_4, ZCUT_8, ZCUT_16;
        static int do_init = 1;
```

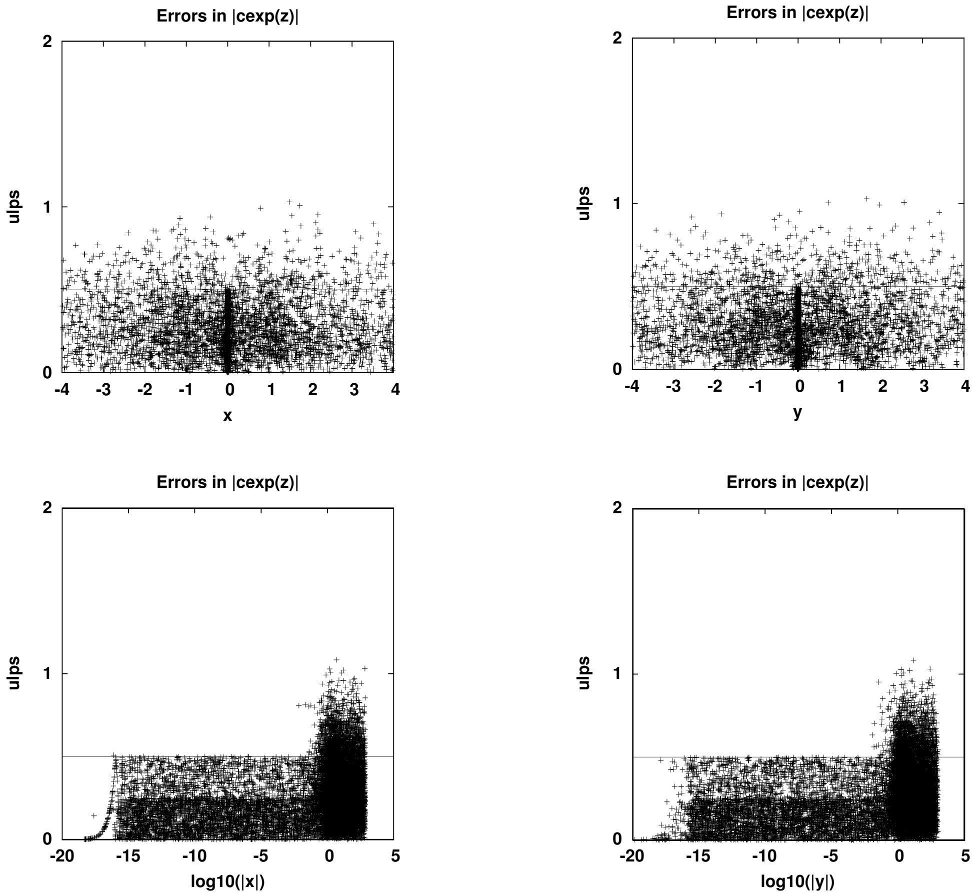


Figure 17.6: Errors in the cexp() function.

```
/*
** Cutoff initialization and Taylor series block omitted
*/

else if (half_z1 < HALF)
{   /* exp(z) - 1 = 2 * tanh(z/2) / (1 - tanh(z/2)) */
    fp_cx_t d, half_z, q, t;

    CXSET_(half_z, HALF * x, HALF * y);
    CXTANH(t, half_z);
    CXSET_(d, ONE - CXREAL_(t), -CXIMAG_(t));
    CXDIV(q, t, d);
    CXADD_(result, q, q);
}
else if (QABS(x) < LN_2)
{   /* handle subtraction loss in real(exp(z) - 1) */
    fp_t c, em1, s, u, v;

    em1 = EXPM1(x);
    SINCOS(y, &s, &c);
    u = c - ONE;
```

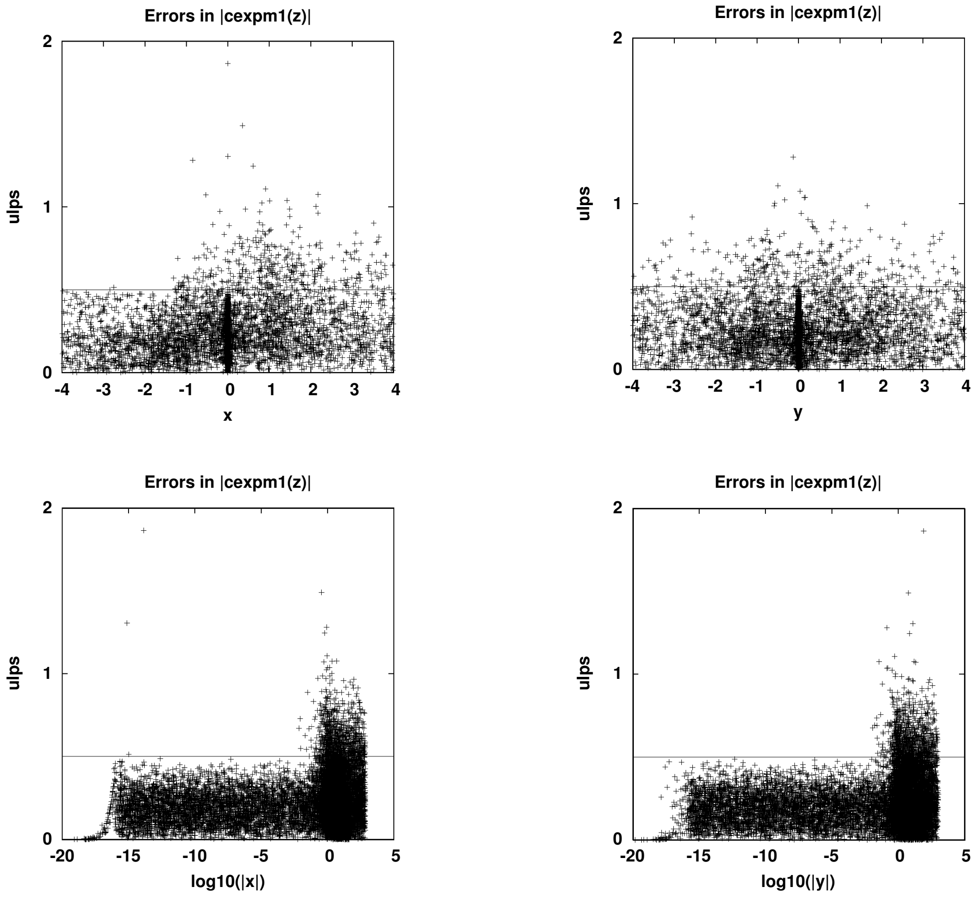


Figure 17.7: Errors in the cexpm1() function.

```
            u += c * em1;
            v = s * em1;
            v += s;
            CXSET_(result, u, v);
        }
        else
        {
            fp_cx_t e;

            CXEXP(e, z);
            CXSUBR_(result, e, ONE);
        }
    }
}
```

The code for the complex-arithmetic function family CEXPM1(z) is a simple wrapper around that for CXEXPM1(z), so we omit it.

The measured errors in the IEEE 754 64-bit function cexpm1(z) are shown in Figure 17.7. The plot pair with arguments on a logarithmic scale shows the superiority of the Taylor-series summation over other routes to the function. Comparison with Figure 17.6 on the preceding page shows that the errors are larger in our cexpm1(z) function, but it nevertheless is the preferred way to compute exponentials of small arguments.

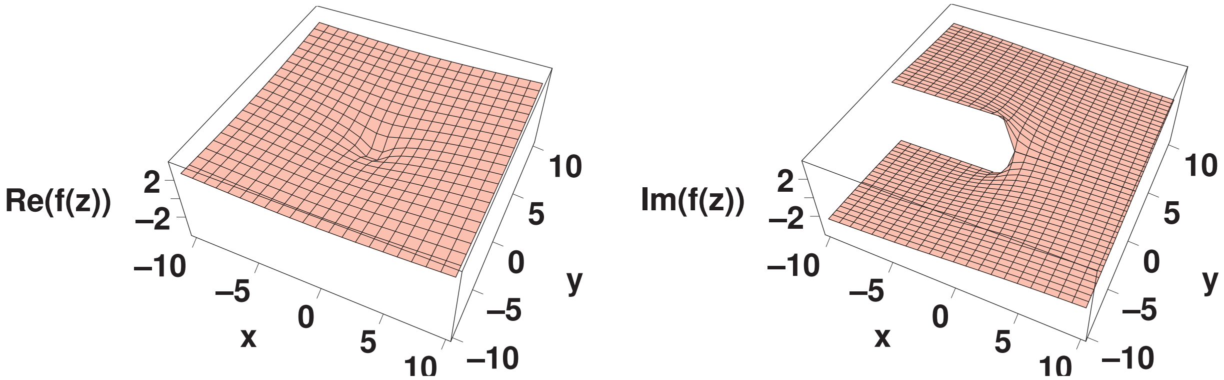


Figure 17.8: Real (left) and imaginary (right) surfaces of `clog(z)`, the complex logarithm function defined by $\log z = \log(x + yi)$. There is a pole at the origin in the real surface where the function goes to $-\infty$ from all directions as $z \to 0$, and there is a *branch cut* along the entire negative real axis in the surface of the imaginary component.

17.11 Complex logarithm

The complex logarithm is graphed in **Figure 17.8**. Like the complex root functions, it has a branch cut in the surface of the imaginary part along the negative real axis. Unlike the root functions, the surface of its real part has a negative pole at the origin.

The complex logarithm has an important conjugation symmetry relation:

$$\begin{aligned} z &= x + yi, && \textit{for real } x \textit{ and } y, \\ z^\star &= x - yi, && \textit{conjugation operation}, \\ \log(z^\star) &= (\log(z))^\star, && \textit{mathematical notation}, \\ \texttt{clog(conj(z))} &= \texttt{conj(clog(z))}, && \textit{computer software notation}. \end{aligned}$$

It satisfies the usual product-to-sum and quotient-to-difference relations expected of logarithms:

$$\begin{aligned} \log(wz) &= \log(w) + \log(z), \\ \log(w/z) &= \log(w) - \log(z), \\ \log(1/z) &= -\log(z). \end{aligned}$$

The C99 Standard specifies the special cases summarized in **Table 17.5** on the following page.

The complex logarithm is particularly easy to compute once we convert its argument from Cartesian form to polar form:

$$\begin{aligned} z &= x + yi, && \textit{for real } x \textit{ and } y, \\ &= r\exp(\theta i), && \textit{for real } r \textit{ and } \theta, \\ r &= |z| = \texttt{cabs(z)} = \texttt{hypot(x, y)}, \\ \theta &= \texttt{carg(z)} = \operatorname{atan2}(y, x), \\ \log(z) &= \log(r\exp(\theta i)) = \log(r) + \theta i. \end{aligned}$$

As long as the complex argument function is properly implemented to guarantee the conjugation symmetry relation `carg(conj(z))` $= -$`carg(z)`, then our complex logarithm function ensures that its own symmetry relation is always obeyed.

Our code for the complex-as-real representation includes a minimal number of tests to handle the required special cases for which the simple code does not work properly. It also includes exact downward scaling of large arguments to avoid premature overflow in $|z|$. Scaling is unnecessary for arguments of small or tiny magnitude, so we normally avoid it. Our code looks like this:

Table 17.5: Special cases for the complex logarithm function, `clog(z)`, according to the C99 Standard [C99, TC3 §G.6.3.2, pages 478–479]. The value f is any *finite* value, and g is any *positive-signed* finite value.

z	$\log(z)$	**Remark**
`conj(z)`	`conj(clog(z))`	conjugation symmetry relation
$-0+0i$	$-\infty+\pi i$	raise *divbyzero* exception
$+0+0i$	$-\infty+0i$	raise *divbyzero* exception
$f\pm\infty i$	$+\infty\pm\frac{1}{2}\pi i$	
$f\pm\text{NaN}i$	$\text{NaN}+\text{NaN}i$	optionally raise *invalid* exception
$-\infty\pm gi$	$+\infty\pm\pi i$	
$+\infty\pm gi$	$+\infty\pm 0i$	
$-\infty\pm\infty i$	$+\infty\pm\frac{3}{4}\pi i$	
$+\infty\pm\infty i$	$+\infty\pm\frac{1}{4}\pi i$	
$\pm\infty\pm\text{NaN}i$	$+\infty\pm\text{NaN}i$	unusual
$\text{NaN}\pm fi$	$\text{NaN}\pm\text{NaN}i$	optionally raise *invalid* exception
$\text{NaN}\pm\infty i$	$+\infty\pm\text{NaN}i$	unusual
$\text{NaN}\pm\text{NaN}i$	$\text{NaN}\pm\text{NaN}i$	
$+\infty\pm fi$	$+\infty\pm 0i$	
$+\infty\pm\text{NaN}i$	$+\infty+\text{NaN}i$	unusual
$-\infty\pm fi$	$+0\pm\infty i$	
$-\infty\pm\text{NaN}i$	$\text{NaN}\pm\infty i$	unusual
$\text{NaN}+fi$	$\text{NaN}+\text{NaN}i$	
$f+\text{NaN}i$	$\text{NaN}+\text{NaN}i$	
$\text{NaN}+\text{NaN}i$	$\text{NaN}+\text{NaN}i$	

```
void
CXLOG(fp_cx_t result, const fp_cx_t z)
{   /* complex logarithm: result = log(z) */
    fp_t r, t, x, y;
    static const fp_t X_CUT = FP(0.5) * FP_T_MAX;

    x = CXREAL_(z);
    y = CXIMAG_(z);

    if (ISINF(x) && ISNAN(y))
        CXSET_(result, COPYSIGN(x, 1.0), y);
    else if (ISNAN(x) && ISINF(y))
        CXSET_(result, COPYSIGN(y, 1.0), x);
    else if ((QABS(x) > X_CUT) || (QABS(y) > X_CUT))
    {   /* possible overflow of |z| */
        static const fp_t B_INV = FP(1.) / (fp_t)B;
        fp_cx_t w;
        volatile fp_t s;

        /* log(w) = log(z/B), so log(z) = log(w) + log(B) */

        CXSET_(w, x * B_INV, y * B_INV);
        r = CXABS(w);
        t = CXARG(w);
        s = LOG(r);
        s += LOG_B_HI;
        STORE(&s);
        s += LOG_B_LO;
```

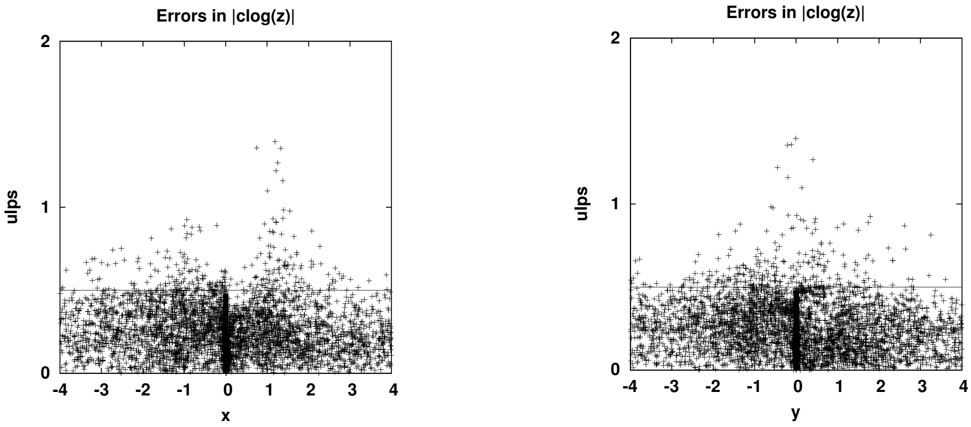


Figure 17.9: Errors in the `clog()` function.

```
            STORE(&s);
            CXSET_(result, s, t);
        }
        else if ( (HALF < x) && (x < TWO) )
        {
            fp_cx_t w;

            CXSUBR_(w, z, ONE);
            CXLOG1P(result, w);
        }
        else
        {
            r = CXABS(z);
            t = CXARG(z);
            CXSET_(result, LOG(r), t);
        }
    }
```

The second last code block handles arguments whose real component lies near one by diverting computation to a function described in the next section. It was added after testing of an earlier version of the code showed errors up to 12 ulps in that region. The change pushed the maximum error below 1.61 ulps.

The code for the complex-arithmetic function family `CLOG(z)` is a simple wrapper around that for `CXLOG(z)`, so we omit it.

The measured errors in the IEEE 754 64-bit function `clog(z)` are displayed in **Figure 17.9**. Outside the plotted range, errors are almost always below the target half-ulp line.

17.12 Complex logarithm near one

In **Section 10.4** on page 290, we discussed the need for accurate computation of the real function $\log(1+x)$. Although the C99 Standard does not specify such a function for complex arguments, one is definitely needed. The mathcw library provides the function family `CLOG1P(z)` for computing $\log(1+z)$. Numerical experiments show that a different treatment of that function is needed in the complex case. In particular, the relation between the logarithm of an exact argument and that of a slightly perturbed, but exactly representable, argument does not provide sufficient accuracy. The code in file `cxl1px.h` implements a computation of $\log(1+z)$ with one of seven algorithms:

- For small arguments, sum the Taylor series

$$\log(1+z) = z - z^2/2 + z^3/3 - \cdots - z^{2k}/(2k) + z^{2k+1}/(2k+1) - \cdots$$

  of orders one, two, four, eight, or sixteen.

- For magnitudes $|z| < \frac{1}{2}$, sum an alternate series [AS64, §4.1.27, page 68] [OLBC10, §4.6.6] given by

$$w = z/(2+z),$$
$$\log(1+z) = 2(w + w^3/3 + w^5/5 + \cdots + w^{2k-1}/(2k-1) + \cdots).$$

  That series converges more rapidly than the normal Taylor series, and requires 8, 18, 21, 37, and 76 terms for the five extended IEEE 754 binary formats for the worst case of $z = -\frac{1}{2}$ and $w = -\frac{1}{3}$. It has the disadvantage that the first term is no longer exact, but instead involves two rounding errors, and the outer factor of two introduces another rounding error for nonbinary bases. Also, although the powers of w are all odd, their signs may nevertheless differ because they are complex numbers: subtraction loss is possible, though less likely because of the restriction on $|z|$.

- Otherwise, compute $\log(1+z)$ directly.

The term counts in the series for the intermediate case are larger than we would like, but the normal Taylor series needs about three times as many terms for the same accuracy, and direct computation of $\log(1+z)$ suffers digit loss. The reciprocals of integers in the series are computed at compile time to machine precision, so no run-time divisions are required.

Here is what our code looks like:

```
void
CXLOG1P(fp_cx_t result, const fp_cx_t z)
{   /* complex logarithm near one: result = log(1 + z) */
    fp_t r, t, x, y;

    x = CXREAL_(z);
    y = CXIMAG_(z);

    if (ISINF(x) && ISNAN(y))
        CXSET_(result, COPYSIGN(x, 1.0), y);
    else if (ISNAN(x) && ISINF(y))
        CXSET_(result, COPYSIGN(y, 1.0), x);
    else
    {
        fp_cx_t sum;
        fp_t half_z1;
        static fp_t ZCUT_1, ZCUT_2, ZCUT_4, ZCUT_8, ZCUT_16;
        static int do_init = 1;

        if (do_init)
        {
            fp_t u;

            u = FOURTH * FP_T_EPSILON / (fp_t)B;
            ZCUT_1  = u;
            ZCUT_2  = HALF * SQRT(THREE * u);
            ZCUT_4  = HALF * SQRT(SQRT(FP(5.0) * u));
            ZCUT_8  = HALF * SQRT(SQRT(SQRT(FP(9.0) * u)));
            ZCUT_16 = HALF * SQRT(SQRT(SQRT(SQRT(FP(17.0) * u))));
            do_init = 0;
        }
```

```
/* cheap lower bound to |z| without overflow */
half_z1 = CXHALF1NORM_(z);

if (half_z1 < ZCUT_1)           /* 1-term Taylor series */
    CXCOPY_(result, z);
else if (half_z1 < ZCUT_2)      /* use 2-term Taylor series */
{  /* Taylor series: log(1 + z) = z - z**2/2 + z**3/3 - ... */
    CXMUL(sum, z, z);
    CXSCALE_(sum, sum, -HALF);
    CXADD_(result, z, sum);
}
else if (half_z1 < ZCUT_4)      /* use 4-term Taylor series */
{
    CXSCALE_(sum, z,   FP(-1.0) / FP(4.0));
    CXADDR_(sum, sum,  FP( 1.0) / FP(3.0));
    CXFMA(sum, sum, z, FP(-1.0) / FP(2.0));
    CXMUL(sum, sum, z);
    CXMUL(sum, sum, z);
    CXADD_(result, z, sum);
}
else if (half_z1 < ZCUT_8)      /* use 8-term Taylor series */
{
    CXSCALE_(sum, z,   FP(-1.0) / FP(8.0));
    CXADDR_(sum, sum,  FP( 1.0) / FP(7.0));
    CXFMA(sum, sum, z, FP(-1.0) / FP(6.0));
    CXFMA(sum, sum, z, FP( 1.0) / FP(5.0));
    CXFMA(sum, sum, z, FP(-1.0) / FP(4.0));
    CXFMA(sum, sum, z, FP( 1.0) / FP(3.0));
    CXFMA(sum, sum, z, FP(-1.0) / FP(2.0));
    CXMUL(sum, sum, z);
    CXMUL(sum, sum, z);
    CXADD_(result, sum, z);
}
else if (half_z1 < ZCUT_16)     /* use 16-term Taylor series */
{
    /* ... code omitted ... */
}
else if (half_z1 < FOURTH)
{
    fp_cx_t d, two_w, w, ww, sum;
    int k;

    CXSET_(d, TWO + x, y);      /* d = 2 + z */
    CXDIV(w, z, d);             /* w = z / (2 + z) */
    CXMUL(ww, w, w);
    CXADD_(two_w, w, w);

    k = N_R_2KM1 - 2;
    CXSET_(sum, r_2km1[k + 1], ZERO);

    for ( ; k > 1; --k)         /* r_2km1[k] = 1 / (2*k - 1) */
        CXFMA(sum, sum, ww, r_2km1[k]); /* Horner form */

    CXMUL(sum, sum, ww);
    CXMUL(sum, two_w, sum);
    CXADD_(result, two_w, sum);
}
```

```
        else
        {
            fp_cx_t zp1;

            CXSET_(zp1, ONE + x, y);
            r = CXABS(zp1);
            t = CXARG(zp1);
            CXSET_(result, LOG(r), t);
        }
    }
}
```

The Taylor-series code accounts for more than half of the function body, but it executes quickly, and guarantees high accuracy for small arguments. The cutoff value ZCUT_16 decreases from 0.370 in the IEEE 754 32-bit format, to 0.110 in the 64-bit format, and to 0.008 in the 128-bit format.

The only delicate point in the code is the summation of the series for intermediate arguments. The loop body is kept short by omitting convergence tests, at the expense of often-unnecessary additional iterations. The loop accumulates the value $\texttt{sum} = 1/3 + w^2/5 + w^4/7 + \cdots$ in nested Horner form, and the final result is obtained from $2w + 2w(w^2 \times \texttt{sum})$ to reduce the rounding error that would be incurred if the loop ran one more iteration, adding the final (lowest-order) term last.

With real arithmetic, our normal programming procedure accumulates the sum in order of decreasing term magnitudes, starting with the second term, and exits the loop as soon as the sum is not changed by the addition of the current term. There is code in cxl1px.h to do that for complex arithmetic, using the one-norm, $|\operatorname{real}(z)| + |\operatorname{imag}(z)|$, in the convergence test. Timing tests show no significant difference between the two methods, so only the shorter Horner form is shown in the code display.

The code for the complex-arithmetic function family CLOG1P(z) is a simple wrapper around that for CXLOG1P(z), so we omit it.

The measured errors in the IEEE 754 64-bit function clog1p(z) are plotted in **Figure 17.10** on the next page. The bottom pair of plots shows the behavior on a logarithmic argument scale, confirming the predicted accuracy of Taylor-series summation. Outside the range shown in the graphs, the function is almost always correctly rounded, with errors below the half-ulp line.

17.13 Complex power

The complex power function is defined in terms of the exponential and logarithm functions like this:

$$z^a = \mathbf{exp}(a \,\mathbf{log}(z)), \qquad \textit{definition of complex power.}$$

We showed in **Chapter 14** on page 411 that the problem of error magnification in the exponential function when the argument is large is a serious impediment to straightforward application of the definition. The real power function, pow(x,y), is by far the most difficult of the standard elementary functions to compute accurately. Moving to the complex domain makes the problem even harder. Here, we take the simplest implementation approach: when higher precision is available, we use it to hide most of the accuracy problems, and otherwise, we use working precision with a declaration that our complex exponential function family is much in need of improvement.

The power function has two arguments, so there are potentially many more special cases of arguments than in the lengthy lists that we tabulate in earlier sections for single-argument functions. The C99 Standard does not attempt to enumerate them. Instead, it notes that the power function cpow(z,a) may be computed as cexp(a * clog(z)), but permits implementations to provide more careful handling of special cases.

The code is straightforward, but must be regarded as preliminary:

```
void
CXPOW(fp_cx_t result, const fp_cx_t z, const fp_cx_t a)
{   /* complex power: result = z**a */
    /* use hp_t and hp_cx_t internally to reduce accuracy loss */
    hp_cx_t aa, aa_log_zz, hp_result, log_zz, zz;
    fp_t u, v, x, y;
```

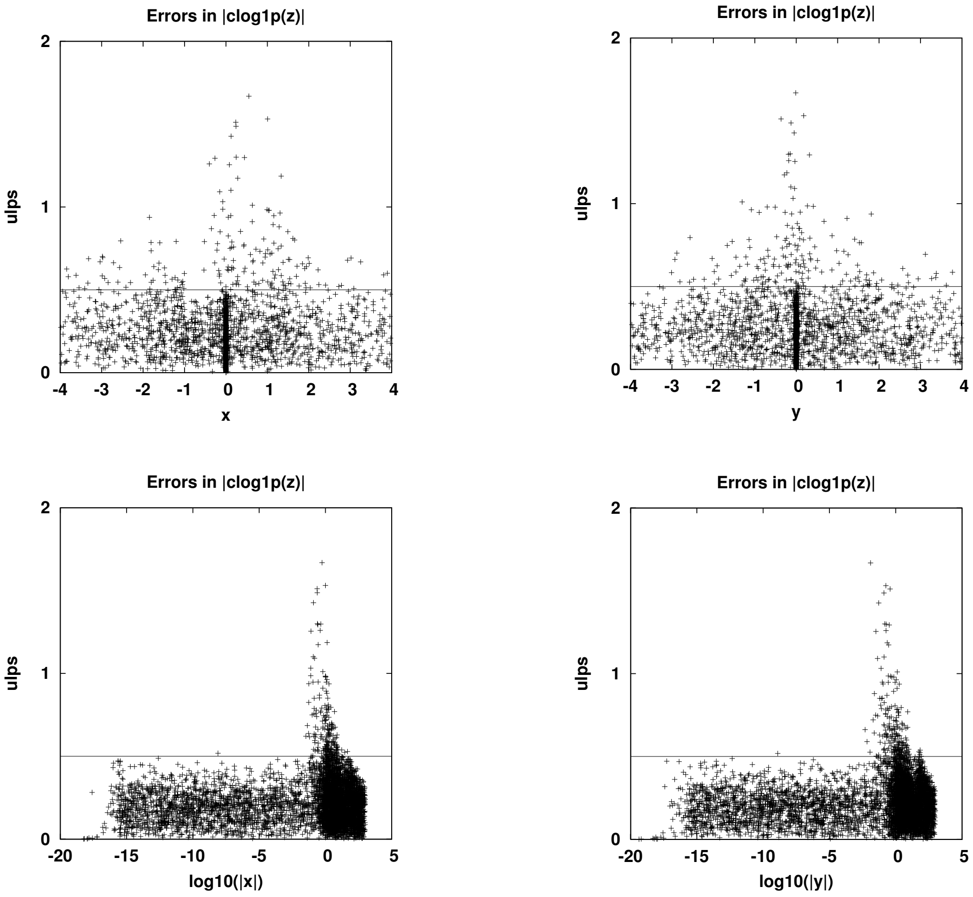


Figure 17.10: Errors in the `clog1p()` function.

```
    u = CXREAL_(a);
    v = CXIMAG_(a);
    x = CXREAL_(z);
    y = CXIMAG_(z);
    HP_CXSET_(zz, (hp_t)x, (hp_t)y);

    if ( (v == ZERO) && (y == ZERO) )           /* real case */
        HP_CXSET_(hp_result, (hp_t)POW(x, u), y + v);
    else if ( (v == ZERO) && (TRUNC(u) == u) && (QABS(u) <= (fp_t)INT_MAX) )    /* integer power */
        HP_CXIPOW(hp_result, zz, (int)u);
    else                                        /* general case: z**a = cexp(a * clog(z)) */
    {
        HP_CXSET_(aa, (hp_t)u, (hp_t)v);
        HP_CXLOG(log_zz, zz);
        HP_CXMUL(aa_log_zz, aa, log_zz);
        HP_CXEXP(hp_result, aa_log_zz);
    }

    CXSET_(result, (fp_t)HP_CXREAL_(hp_result), (fp_t)HP_CXIMAG_(hp_result));
}
```

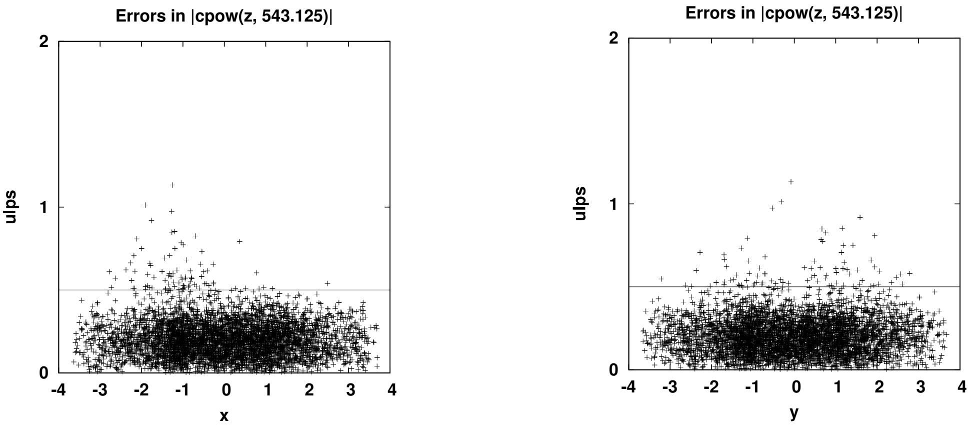


Figure 17.11: Errors in the `cpow()` function.

The only concessions that we make to efficiency are to divert the special case of integer powers of complex numbers to the `CXIPOW()` family, and the case of real z and a to the real `POW()` routine. We do not show the code here for `CXIPOW()`, because it is just a simplified version of the real function family, `IPOW()`, that uses bitwise exponent reduction and squaring to compute integer powers with an almost minimal number of multiplications (see **Section 14.3** on page 414).

The native complex power function, `CPOW(z)`, is the usual simple wrapper around a call to `CXPOW(z)`, and we omit it.

The measured errors in the complex power function for a single large nonintegral exponent are shown in **Figure 17.11**. The internal use of higher precision usually produces correctly rounded results if the exponent is not too large. The exponent chosen for the plots pushes the maximum errors up to about 1.14 ulps, but most results lie below the desirable half-ulp line.

Despite that graphical observation, errors in the complex power function can sometimes be larger, because we noted in the retrospective in **Section 14.15** on page 440 that the worst cases for the real power function may require triple the working precision. The known issues with complex-arithmetic primitives suggest that guaranteeing always-correct rounding for the `CPOW()` family may need *six times* working precision.

17.14 Complex trigonometric functions

The cosine, sine, and tangent for complex arguments are shown in **Figure 17.12** through **Figure 17.14** on the facing page. They each depend on the real trigonometric and hyperbolic functions through these defining relations:

$$\begin{aligned}
z &= x + yi, && \textit{for real } x \textit{ and } y,\\
\cos(z) &= \cos(x)\cosh(y) - \sin(x)\sinh(y)i,\\
\sin(z) &= \sin(x)\cosh(y) + \cos(x)\sinh(y)i,\\
\tan(z) &= \sin(z)/\cos(z)\\
&= \frac{\sin(2x) + \sinh(2y)i}{\cos(2x) + \cosh(2y)}.
\end{aligned}$$

The C99 Standard does not tabulate special cases of those functions for Infinity, NaN, and signed zero arguments, because they can be inferred from entries in **Table 17.7** on page 513 through **Table 17.9** on page 514 and these relations between the trigonometric and hyperbolic functions of complex arguments:

$$\begin{aligned}
\cos(zi) &= \cosh(z), & \cos(z) &= \cosh(zi),\\
\sin(zi) &= \sinh(z)i, & \sin(z) &= -\sinh(zi)i,
\end{aligned}$$

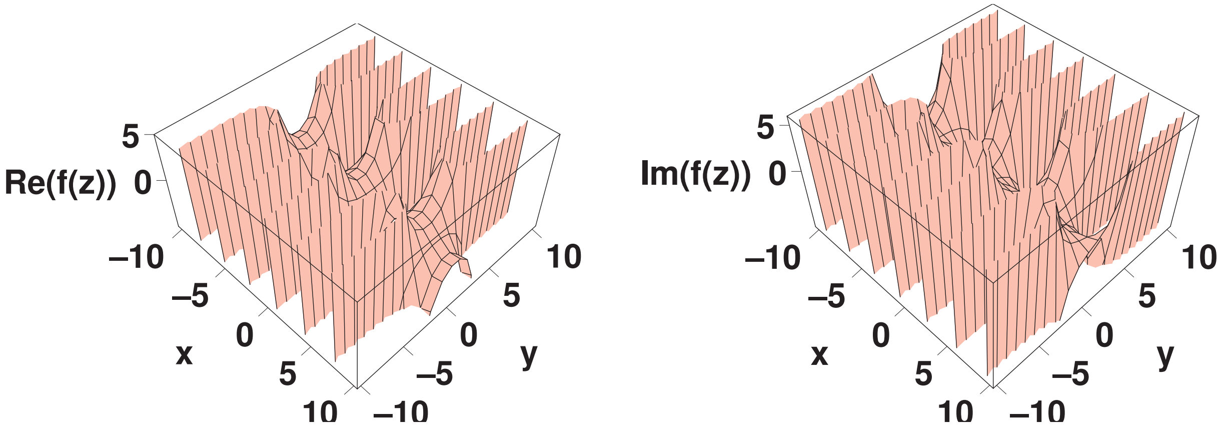


Figure 17.12: Real (left) and imaginary (right) surfaces of `ccos(z)`, the complex trigonometric cosine function. There are neither branch cuts nor poles.

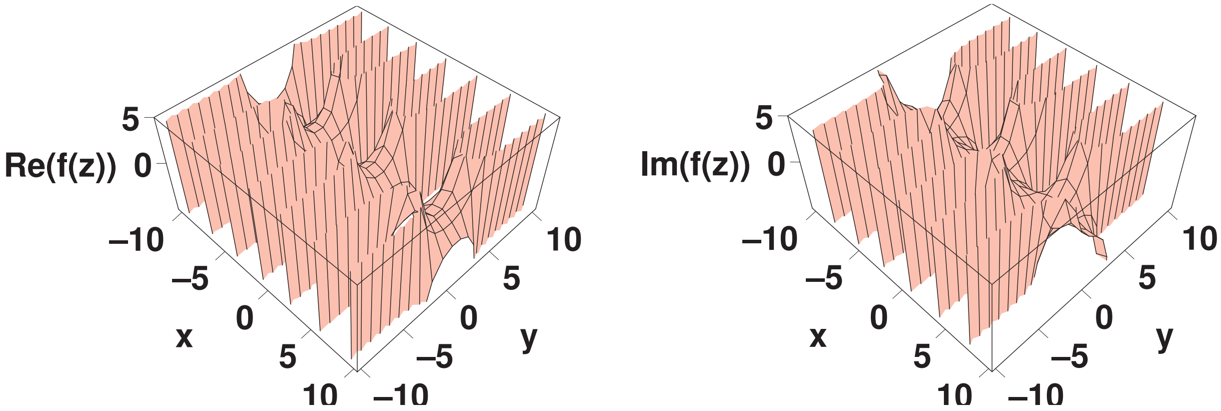


Figure 17.13: Real (left) and imaginary (right) surfaces of `csin(z)`, the complex trigonometric sine function. There are neither branch cuts nor poles.

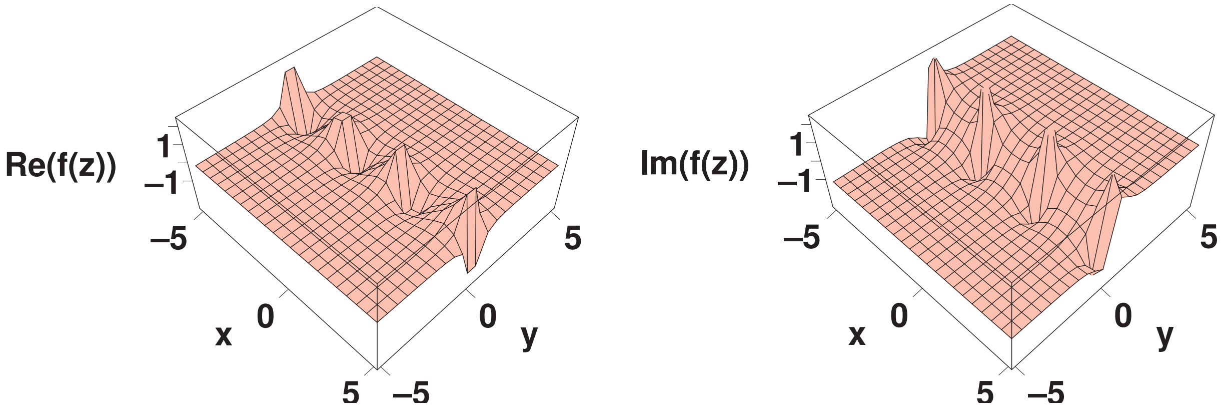


Figure 17.14: Real (left) and imaginary (right) surfaces of `ctan(z)`, the complex trigonometric tangent function. There are no branch cuts, but there are poles to $\pm\infty$ along the real axis in both the imaginary and real components.

$$\tan(zi) = \tanh(z)i, \qquad \tan(z) = -\tanh(zi)i.$$

All three functions are subject to subtraction loss for certain combinations of x and y, and the tangent suffers argument error in forming $2x$ in nonbinary bases. The tangent formula with the sum in the denominator should not be used when $\cos(2x) < 0$ and $\cosh(2y) < 2$, for which $|y| < 0.659$. The easiest way to reduce the losses is to perform the computations in the next higher precision, when that is feasible.

Our code for the three functions has a similar structure, so we show only that for the complex cosine using our complex-as-real type:

```
void
CXCOS(fp_cx_t result, const fp_cx_t z)
{   /* complex cosine: result = cos(z) */
    hp_t c, ch, s, sh, x, y;

    x = (hp_t)CXREAL_(z);
    y = (hp_t)CXIMAG_(z);
    HP_SINCOS(x, &s, &c);
    HP_SINHCOSH(y, &sh, &ch);
    CXSET_(result, (fp_t)(c * ch), (fp_t)(-s * sh));
}
```

As we observed in **Section 17.5** on page 479, it is helpful here to have the `SINCOS()` and `SINHCOSH()` families available for computing two function values while sharing parts of their code, and avoiding needless duplication of the difficult job of accurate argument reduction.

Existing implementations of those complex-valued functions are likely to produce nonsensical answers for even moderate z values because of inaccurate argument reduction and subtraction loss.

The function families `CCOS(z)`, `CSIN(z)`, and `CTAN(z)` are simple wrappers around calls to their complex-as-real companions.

The measured errors in the IEEE 754 `complex double` versions of the trigonometric functions are shown in **Figure 17.15** through **Figure 17.17** on the facing page. The internal use of higher precision produces results that are almost always correctly rounded.

17.15 Complex inverse trigonometric functions

The inverse trigonometric functions for complex arguments are graphed in **Figure 17.18** through **Figure 17.20** on page 506. They are related to the complex logarithm and square root, and are defined by these relations:

$$\operatorname{acos}(z) = \begin{cases} -\log(z + \sqrt{1-z^2} \times i)i, \\ +\log(z - \sqrt{1-z^2} \times i)i, \end{cases}$$
$$\operatorname{asin}(z) = \begin{cases} -\log(\sqrt{1-z^2} + z \times i)i, \\ +\log(\sqrt{1-z^2} - z \times i)i, \end{cases}$$
$$\operatorname{atan}(z) = \tfrac{1}{2}\log\left(\frac{i+z}{i-z}\right) i.$$

Where two formulas are given, the one that minimizes subtraction loss should be chosen, but notice that such losses can happen in either, or both, of the real and imaginary parts of the arguments.

The inverse trigonometric functions satisfy these important symmetry relations:

$$\operatorname{acos}(-z) = \pi - \operatorname{acos}(z), \qquad \operatorname{asin}(-z) = -\operatorname{asin}(z), \qquad \operatorname{atan}(-z) = -\operatorname{atan}(z).$$

Two of them are related to their inverse hyperbolic companions with arguments multiplied by the imaginary unit, like this:

$$\operatorname{asin}(z) = -\operatorname{asinh}(zi)i, \qquad \operatorname{atan}(z) = -\operatorname{atanh}(zi)i.$$

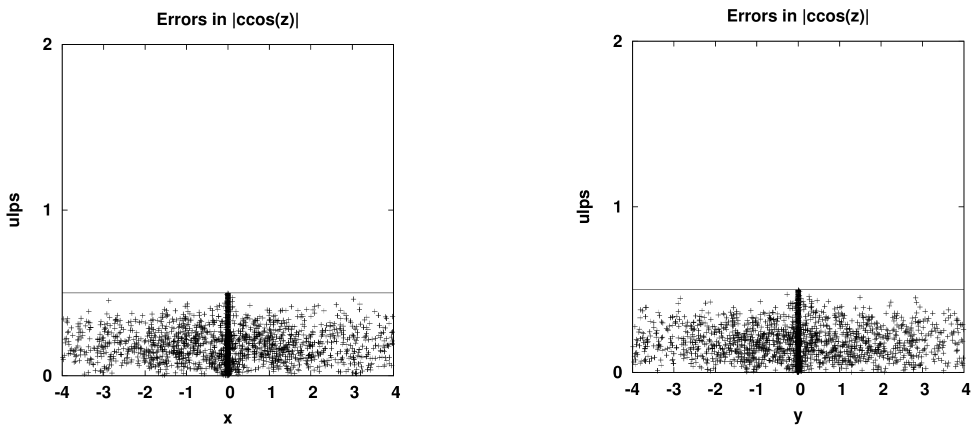


Figure 17.15: Errors in the `ccos()` function.

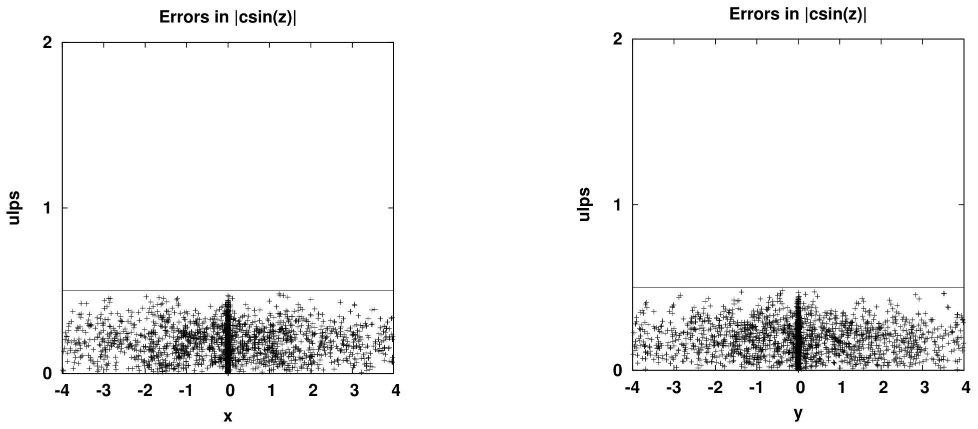


Figure 17.16: Errors in the `csin()` function.

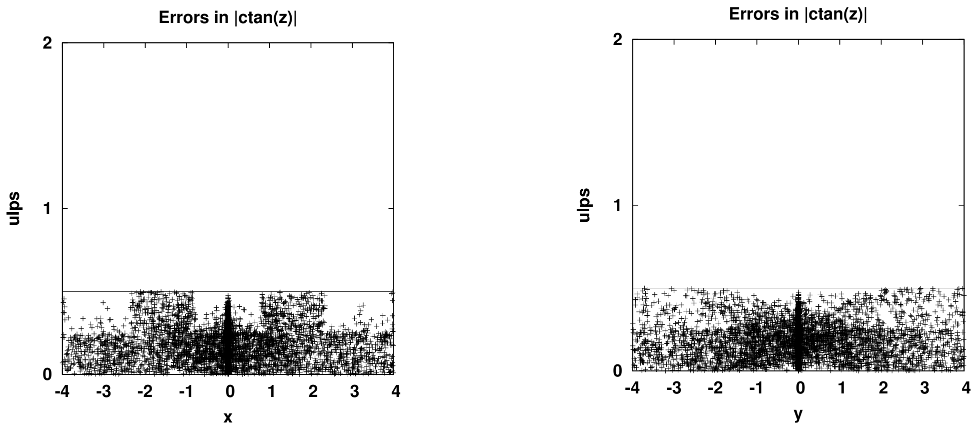


Figure 17.17: Errors in the `ctan()` function.

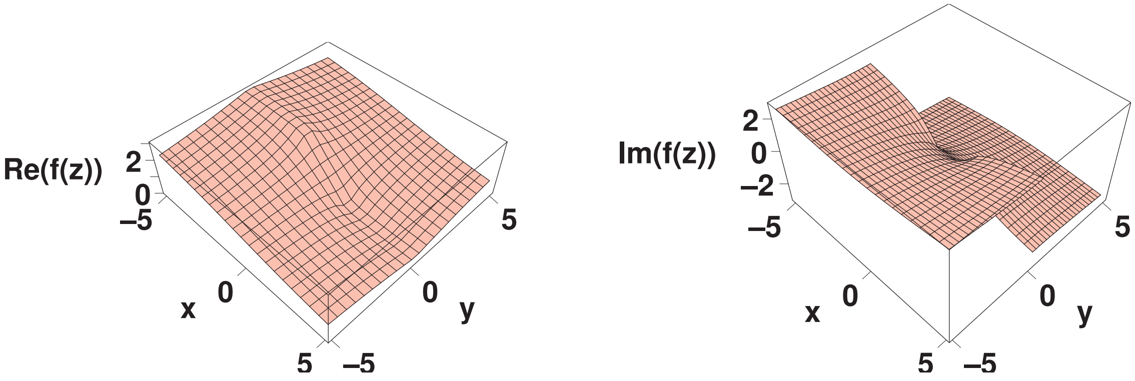


Figure 17.18: Real (left) and imaginary (right) surfaces of `cacos(z)`, the complex inverse trigonometric cosine function. There are cusps on the real axis on $(-\infty, -1)$ and $(+1, +\infty)$ on the real surface, and branch cuts on the same intervals on the imaginary surface. Seen from an opposite viewpoint, the branch cuts resemble those shown in Figure 17.19.

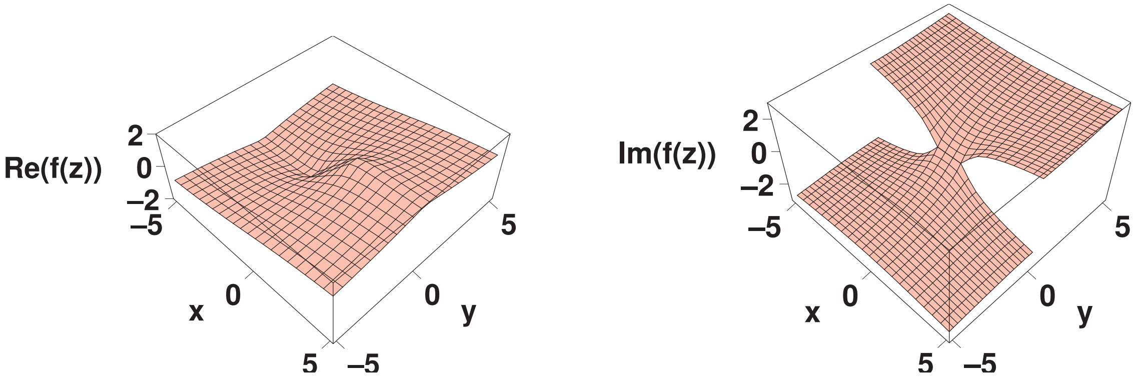


Figure 17.19: Real (left) and imaginary (right) surfaces of `casin(z)`, the complex inverse trigonometric sine function. There are cusps on the real axis on $(-\infty, -1)$ and $(+1, +\infty)$ on the real surface, and branch cuts on the same intervals on the imaginary surface.

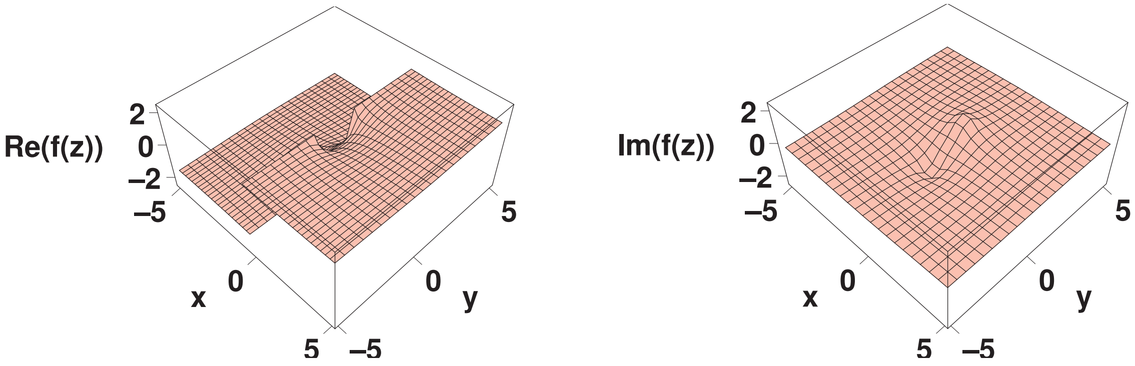


Figure 17.20: Real (left) and imaginary (right) surfaces of `catan(z)`, the complex inverse trigonometric tangent function. There are branch cuts on the imaginary axis on $(-\infty, -1]$ and $[+1, +\infty)$ on the real surface, and poles at $z = \pm i$ on the imaginary surface.

Table 17.6: Special cases for the complex inverse trigonometric cosine function, `cacos(z)`, according to the C99 Standard [C99, TC3 §G.6.1.1, page 474]. The value f is any *finite* value, g is any *positive-signed* finite value, and h is any *nonzero* finite value.

z	**acos**(z)	**Remark**
`conj(z)`	`conj(cacos(z))`	conjugation symmetry relation
$\pm 0 + 0i$	$\frac{1}{2}\pi - 0i$	
$\pm 0 + \mathrm{NaN}i$	$\frac{1}{2}\pi + \mathrm{NaN}i$	
$f + \infty i$	$\frac{1}{2}\pi - \infty i$	
$h + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$-\infty + gi$	$\pi - \infty i$	
$+\infty + gi$	$+0 - \infty i$	
$-\infty + \infty i$	$\frac{3}{4}\pi - \infty i$	
$+\infty + \infty i$	$\frac{1}{4}\pi - \infty i$	
$\pm\infty + \mathrm{NaN}i$	$\mathrm{NaN} \pm \infty i$	result imaginary sign unspecified
$\mathrm{NaN} + fi$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$\mathrm{NaN} + \infty i$	$\mathrm{NaN} - \infty i$	
$\mathrm{NaN} + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	

The C99 Standard tabulates the special cases for the inverse cosine shown in Table 17.6. Special cases for the inverse sine and tangent are inferred from the symmetry relations to their inverse hyperbolic companions, and from the entries in Table 17.11 on page 518 and Table 17.12 on page 518.

The inverse trigonometric functions do not reduce nicely into separate real and imaginary parts: we have to use complex arithmetic to compute the arguments of the logarithms and square roots.

Where complex values are multiplied by the imaginary unit, i, in the defining formulas, that must be done by component swap and sign change, *not* by calling a function for complex multiplication: compute $zi = (x + yi)i = -y + xi$. The inline macro `CXMULBYI_()` in the header file `cxcw.h` hides the details of that operation.

The arguments of the square roots are subject to serious subtraction loss, but that can be reduced by computing $1 - z^2$ as $(1 + z)(1 - z)$. Nevertheless, that still leaves the problem of accurate computation of the square root when $|z|$ is small, for which we have this Taylor series:

$$\sqrt{1 - z^2} = 1 - z^2/2 - z^4/8 - z^6/16 - 5z^8/128 + \cdots .$$

We also recall from Section 17.12 on page 497 that we should take care to rewrite expressions of the form $\log(1 + z)$ as `clog1p(z)`.

The conclusion from those remarks is clear: if we compute the inverse trigonometric functions of complex arguments only in working precision, there are many opportunities for errors from subtraction loss and inadequate argument precision. Higher internal precision, when available, provides a reasonable way to hide most of the loss without doing extensive, and complicated, numerical analysis.

We show only the code for the inverse cosine of complex-as-real arguments here, because that for the inverse sine and inverse tangent has a similar structure:

```
void
CXACOS(fp_cx_t result, const fp_cx_t z)
{   /* complex inverse cosine: result = acos(z) */
    /* symmetry: acos(-z) = PI - acos(z) */

    fp_t half_z1, x, y;
    static fp_t ZCUT_1;
    static int do_init = 1;

    if (do_init)
    {
        ZCUT_1  = FOURTH * FP_T_EPSILON / (fp_t)B;
        do_init = 0;
    }

    x = CXREAL_(z);
    y = CXIMAG_(z);
    half_z1 = CXHALF1NORM_(z);

    if (ISINF(x))
    {
        if (ISINF(y))
        {
            if (x > ZERO)
                CXSET_(result, PI_QUARTER, -y);
            else
                CXSET_(result, THREE_PI_QUARTER, -y);
        }
        else if (ISNAN(y))
            CXSET_(result, COPYSIGN(y, ONE), x);
        else if (SIGNBIT(x) == 0)
            CXSET_(result, ZERO, -COPYSIGN(x, y));
        else
            CXSET_(result, PI, -COPYSIGN(x, y));
    }
    else if (ISNAN(x))
    {
        if (ISINF(y))
            CXSET_(result, COPYSIGN(x, ONE), -y);
        else if (ISNAN(y))
            CXSET_(result, COPYSIGN(x, ONE), y);
        else
        {
            (void)QNAN("");     /* raise invalid f.p. exception */
            CXSET_(result, COPYSIGN(x, ONE), COPYSIGN(x, y));
        }
    }
    else if (ISINF(y))
        CXSET_(result, PI_HALF, -y);
    else if (ISNAN(y))
    {
        if (x == ZERO)
            CXSET_(result, PI_HALF, y);
        else
        {
            (void)QNAN("");     /* optionally raise invalid exception */
            CXSET_(result, COPYSIGN(y, ONE), y);
        }
    }
    else if (y == ZERO)
        CXSET_(result, ACOS(x), COPYSIGN(ZERO, -COPYSIGN(ONE, y)));
    else if (half_z1 < ZCUT_1)
        CXSET_(result, PI_HALF - x, -y);
    else
    {
        hp_cx_t p, r, s, t, u, v, w, zz;
        hp_t xx, yy;
        xx = (hp_t)x;
        yy = (hp_t)y;

        if (HP_SIGNBIT(xx))
            HP_CXSET_(zz, -xx, -yy);            /* zz = -z */
        else
            HP_CXSET_(zz, xx, yy);              /* zz = +z */

        HP_CXSET_(r, HP(1.0) - HP_CXREAL_(zz), -HP_CXIMAG_(zz));
        HP_CXSET_(s, HP(1.0) + HP_CXREAL_(zz), HP_CXIMAG_(zz));
        HP_CXMUL(t, r, s);
        HP_CXSQRT(r, t);
        HP_CXSET_(u, -HP_CXIMAG_(r), HP_CXREAL_(r));

        if (HP_SIGNBIT(HP_CXREAL_(zz)) == HP_SIGNBIT(HP_CXREAL_(u)))
        {                                       /* use -i * log(zz + i * sqrt(1 - zz**2)) */
            HP_CXADD_(v, zz, u);
            HP_CXLOG(w, v);                     /* log(zz + i * sqrt(1 - zz**2)) */
            HP_CXSET_(p, HP_CXIMAG_(w), -HP_CXREAL_(w));
        }
        else
        {                                       /* use +i * log(zz - i * sqrt(1 - zz**2)) */
            HP_CXSUB_(v, zz, u);
            HP_CXLOG(w, v);                     /* log(zz - i * sqrt(1 - zz**2)) */
            HP_CXSET_(p, -HP_CXIMAG_(w), HP_CXREAL_(w));
        }

        if (HP_SIGNBIT(xx))                     /* acos(-z) = PI - acos(z) */
        {
            volatile hp_t q;

            q = (hp_t)PI_HI - HP_CXREAL_(p);
            HP_STORE(&q);
            q += (hp_t)PI_LO;
            CXSET_(result, (fp_t)q, (fp_t)(-HP_CXIMAG_(p)));
        }
        else
            CXSET_(result, (fp_t)HP_CXREAL_(p), (fp_t)HP_CXIMAG_(p));
    }
}
```

The code is not optimal. We use only the one-term form of the Taylor series. We base the choice of logarithm formula only on the sign of the real part, rather than the larger of the real and imaginary parts. We do not use the CLOG1P() family for arguments near one. However, we preserve the important symmetry relation for argument negation.

The function families CACOS(z), CASIN(z), and CATAN(z) for the native complex types are simple wrappers around the functions for the complex-as-real types, and thus, are not shown here.

The measured errors in the IEEE 754 complex double versions of the inverse trigonometric functions are shown in Figure 17.21 through Figure 17.23 on the following page. The internal use of higher precision ensures that results are almost always correctly rounded.

17.16 Complex hyperbolic functions

Graphs of the hyperbolic functions of complex argument are shown in Figure 17.24 through Figure 17.26 on page 511. They are defined in terms of trigonometric and hyperbolic functions of real arguments like this [Smi98]:

$$z = x + yi, \qquad \text{for real } x \text{ and } y,$$

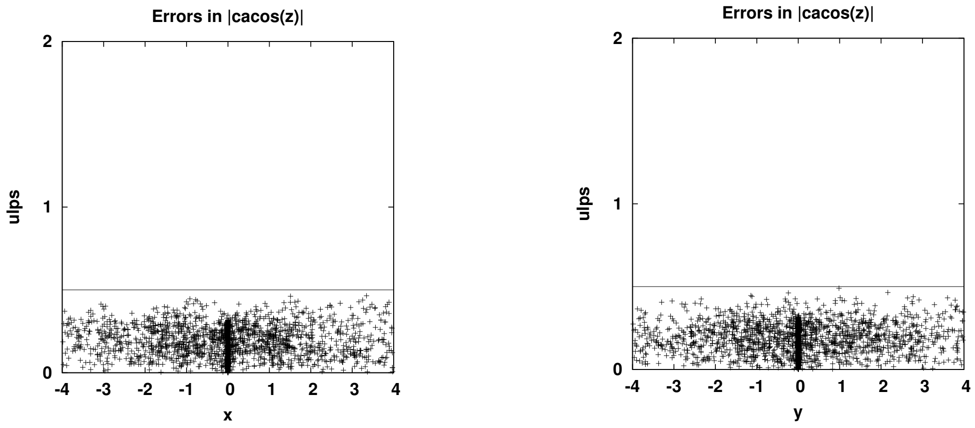


Figure 17.21: Errors in the `cacos()` function.

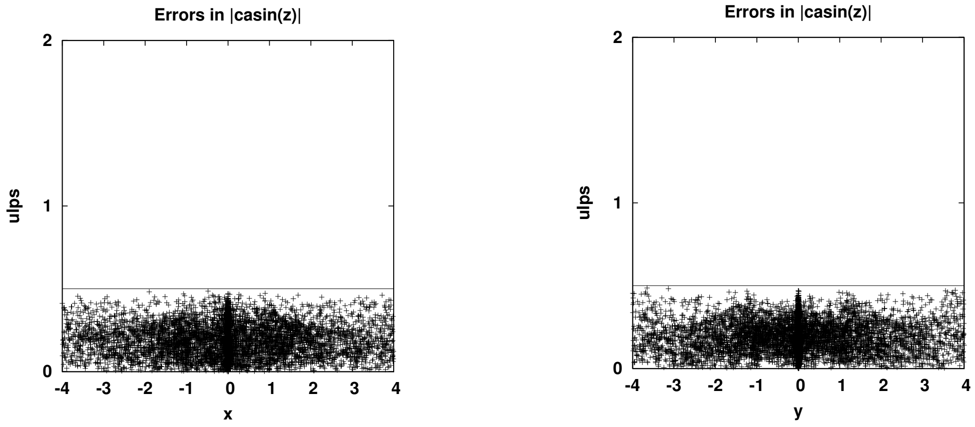


Figure 17.22: Errors in the `casin()` function.

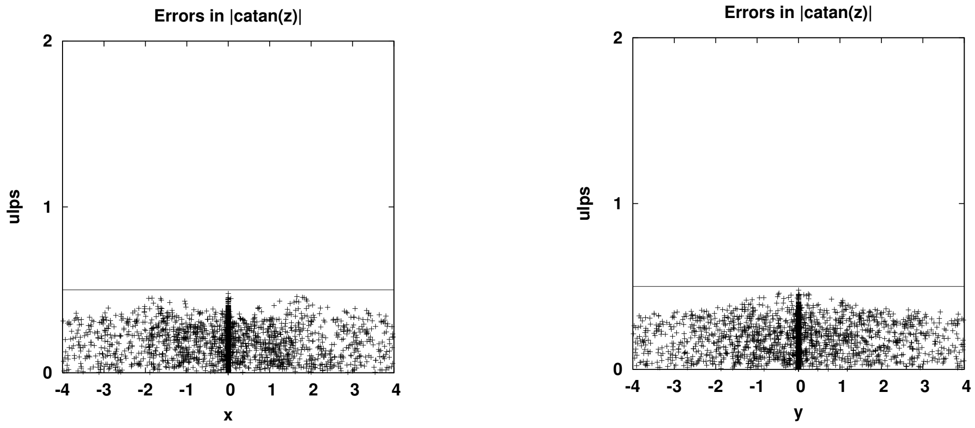


Figure 17.23: Errors in the `catan()` function.

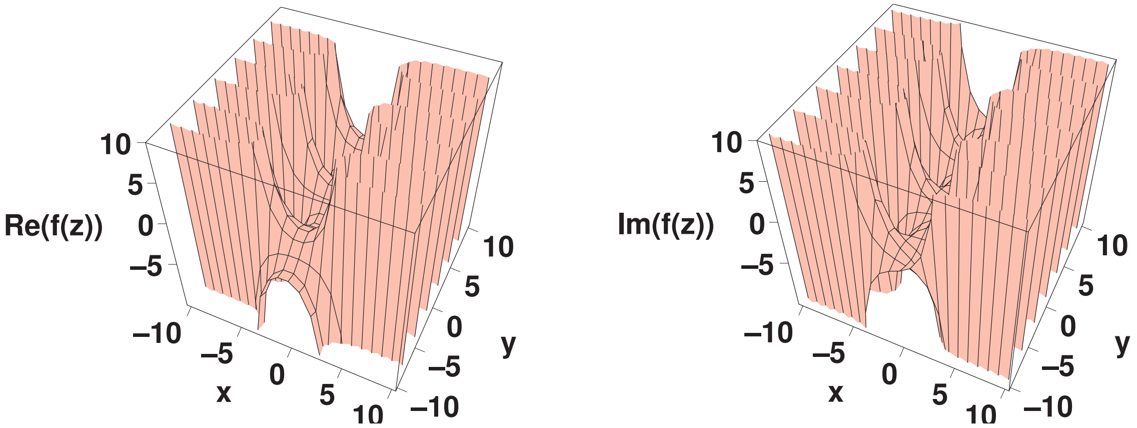


Figure 17.24: Real (left) and imaginary (right) surfaces of `ccosh(z)`, the complex hyperbolic cosine function. There are no branch cuts or poles, but there is sometimes exponential growth in both surfaces.

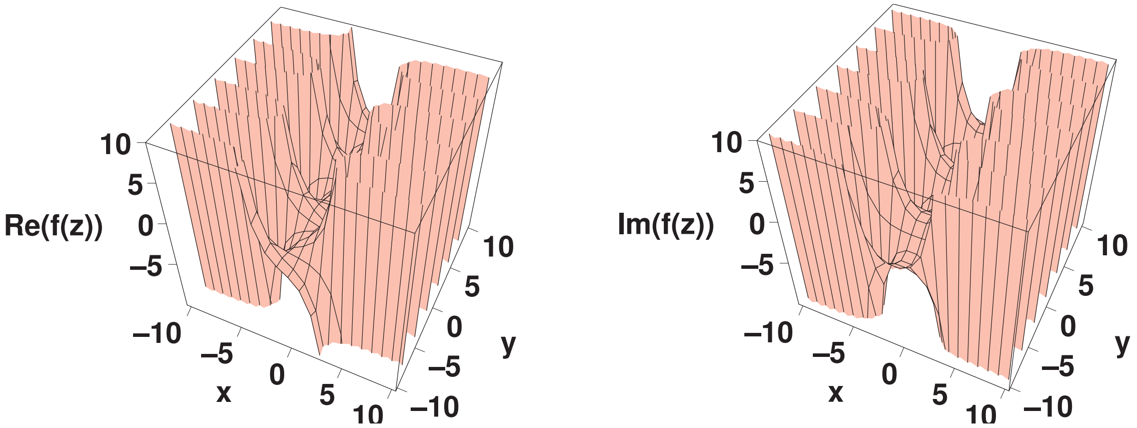


Figure 17.25: Real (left) and imaginary (right) surfaces of `csinh(z)`, the complex hyperbolic sine function. There are no branch cuts or poles, but there is sometimes exponential growth in both surfaces.

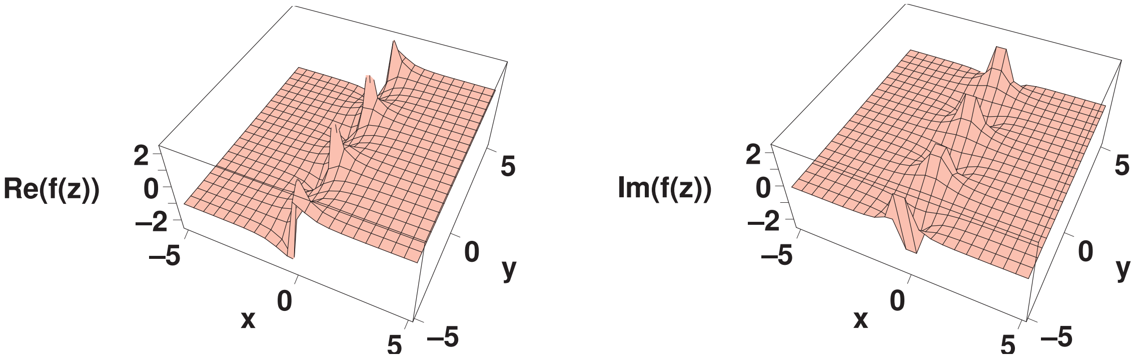


Figure 17.26: Real (left) and imaginary (right) surfaces of `ctanh(z)`, the complex hyperbolic tangent function. There are asymptotic poles to $\pm\infty$ on the imaginary axis for y an odd multiple of $\pi/2$ on the surface of the imaginary component.

$$\begin{aligned}\cosh(z) &= \cosh(x)\cos(y) + \sinh(x)\sin(y)i,\\ \sinh(z) &= \sinh(x)\cos(y) + \cosh(x)\sin(y)i,\\ \tanh(z) &= \sinh(z)/\cosh(z),\\ &= \frac{\sinh(2x) + \sin(2y)i}{\cosh(2x) + \cos(2y)}.\end{aligned}$$

The hyperbolic functions have these important symmetry relations under argument negation:

$$\cosh(-z) = \cosh(z), \qquad \sinh(-z) = -\sinh(z), \qquad \tanh(-z) = -\tanh(z).$$

As we noted in **Section 17.14** on page 502, the hyperbolic and trigonometric functions of complex arguments are related:

$$\begin{aligned}\cosh(z) &= \cos(zi), & \cosh(zi) &= \cos(z),\\ \sinh(z) &= -\sin(zi)i, & \sinh(zi) &= \sin(z)i,\\ \tanh(z) &= -\tan(zi)i, & \tanh(zi) &= \tan(z)i.\end{aligned}$$

The C99 Standard specifies the special cases that are summarized in **Table 17.7** on the next page through **Table 17.9** on page 514.

The real and imaginary parts require only function calls and products, so subtraction loss is not a problem for the hyperbolic cosine and sine. However, the doubled arguments in the definition of the hyperbolic tangent introduce argument errors in nonbinary bases that we can hide by computing in higher intermediate precision. Also, the denominator may lose leading digits when $\cos(2y) < 0$ and $\cosh(2x) < 2$, for which $|x| < 0.659$. In that case, it is better to compute the hyperbolic tangent from the ratio $\sinh(z)/\cosh(z)$. Accurate results for arguments of large magnitude can be expected only if argument reduction in the real hyperbolic and trigonometric functions is exact, as it is in the mathcw library, but not in most native libraries. Here is the code for the hyperbolic tangent with complex-as-real argument:

```
void
CXTANH(fp_cx_t result, const fp_cx_t z)
{   /* complex hyperbolic tangent: result = tanh(z) */
    /* tanh(z) = sinh(z) / cosh(z) */
    /* tanh(x+y*i) = (sinh(2*x)+ sin(2*y)*i)/(cosh(2*x) + cos(2*y)) */

#if FP_T_DIG == HP_T_DIG        /* no higher precision available */
    fp_t x, y;
    fp_cx_t ch, sh;

    x = CXREAL_(z);
    y = CXIMAG_(z);

    if (QABS(x) < FP(0.659))    /* cosh(2 * 0.658...) == 2 */
    {
        CXCOSH(ch, z);
        CXSINH(sh, z);
        CXDIV(result, sh, ch);  /* tanh(z) = sinh(z) / cosh(z) */
    }
    else
    {
        fp_t c, ch, d, s, sh;

        SINHCOSH(x + x, &sh, &ch);
        SINCOS(y + y, &s, &c);
        d = ch + c;             /* no leading digit loss when c < 0 */
        CXSET_(result, sh / d, s / d);
    }
```

Table 17.7: Special cases for the complex hyperbolic cosine function, `ccosh(z)`, according to the C99 Standard [C99, TC3 §G.6.2.4, page 476]. The value h is any *nonzero* finite value.

z	**cosh**(z)	**Remark**
$-z$	`ccosh(z)`	function is even
`conj(z)`	`conj(ccosh(z))`	conjugation symmetry relation
$+0 \pm 0i$	$1 \pm 0i$	
$-0 \pm 0i$	$1 \mp 0i$	
$+0 + \infty i$	$\mathrm{NaN} \pm 0i$	raise *invalid* exception; result imag. sign unspecified
$+0 + \mathrm{NaN}i$	$\mathrm{NaN} \pm 0i$	result imaginary sign unspecified
$h + \infty i$	$\mathrm{NaN} + \mathrm{NaN}i$	raise *invalid* exception
$h + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$+\infty \pm 0i$	$\infty \pm 0i$	
$+\infty \pm hi$	$\infty \cos(h) \pm \infty \sin(h)i$	
$+\infty + \infty i$	$\pm\infty + \mathrm{NaN}i$	raise *invalid* exception; result real sign unspecified
$+\infty - \infty i$	$\pm\infty - \mathrm{NaN}i$	raise *invalid* exception; result real sign unspecified
$+\infty + \mathrm{NaN}i$	$+\infty + \mathrm{NaN}i$	
$\mathrm{NaN} + 0i$	$\mathrm{NaN} \pm 0i$	result imaginary sign unspecified
$\mathrm{NaN} - 0i$	$\mathrm{NaN} \mp 0i$	result imaginary sign unspecified
$\mathrm{NaN} + hi$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$\mathrm{NaN} + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	

Table 17.8: Special cases for the complex hyperbolic sine function, `csinh(z)`, according to the C99 Standard [C99, TC3 §G.6.2.5, pages 476–477]. The value g is any *positive* finite value, and h is any *nonzero* finite value.

z	**sinh**(z)	**Remark**
$-z$	$-$`csinh(z)`	function is odd
`conj(z)`	`conj(csinh(z))`	conjugation symmetry relation
$+0 \pm 0i$	$+0 \pm 0i$	
$-0 \pm 0i$	$-0 \pm 0i$	
$+0 + \infty i$	$\pm 0 + \mathrm{NaN}i$	raise *invalid* exception; result real sign unspecified
$+0 + \mathrm{NaN}i$	$\pm 0 + \mathrm{NaN}i$	result real sign unspecified
$g + \infty i$	$\mathrm{NaN} + \mathrm{NaN}i$	raise *invalid* exception
$g + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$+\infty \pm 0i$	$\infty \pm 0i$	
$+\infty \pm gi$	$\infty \cos(g) \pm \infty \sin(g)i$	
$+\infty + \infty i$	$\pm\infty + \mathrm{NaN}i$	raise *invalid* exception; result real sign unspecified
$+\infty - \infty i$	$\pm\infty - \mathrm{NaN}i$	raise *invalid* exception; result real sign unspecified
$+\infty + \mathrm{NaN}i$	$\pm\infty + \mathrm{NaN}i$	result real sign unspecified
$\mathrm{NaN} \pm 0i$	$\mathrm{NaN} \pm 0i$	
$\mathrm{NaN} + hi$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$\mathrm{NaN} + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	

```
#else /* higher precision hides subtraction loss in denominator */
    hp_t xx, yy;

    xx = HP_CXREAL_(z);
    yy = HP_CXIMAG_(z);

    if (QABS(xx) < HP(0.659))   /* cosh(2 * 0.658...) == 2 */
    {   /* tanh(z) = sinh(z) / cosh(z) */
```

Table 17.9: Special cases for the complex hyperbolic tangent function, `ctanh(z)`, according to the C99 Standard [C99, TC3 §G.6.2.6, page 477]. The value f is any *finite* value, g is any *positive* finite value, and h is any *nonzero* finite value.

z	**tanh**(z)	**Remark**
$-z$	$-$`ctanh(z)`	function is odd
`conj(z)`	`conj(ctanh(z))`	conjugation symmetry relation
$+0 \pm 0i$	$+0 \pm 0i$	
$-0 \pm 0i$	$-0 \pm 0i$	
$f + \infty i$	$\mathrm{NaN} + \mathrm{NaN}i$	raise *invalid* exception
$f + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$+\infty + gi$	$1 + 0\sin(2y)i$	
$+\infty + \infty i$	$1 \pm 0i$	result imaginary sign unspecified
$+\infty + \mathrm{NaN}i$	$1 \pm 0i$	result imaginary sign unspecified
$\mathrm{NaN} \pm 0i$	$\mathrm{NaN} \pm 0i$	
$\mathrm{NaN} + hi$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$\mathrm{NaN} + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	

```
            hp_cx_t ch, sh, zz;
            hp_cx_t hp_result;

            HP_CXSET_(zz, xx, yy);
            HP_CXCOSH(ch, zz);
            HP_CXSINH(sh, zz);
            HP_CXDIV(hp_result, sh, ch);
            CXSET_(result, (fp_t)CXREAL_(hp_result), (fp_t)CXIMAG_(hp_result));
        }
        else                        /* no subtraction loss in denominator */
        {   /* tanh(z) = (sin(2x) + sin(2y)*i) / (cosh(2x) + cos(2y)) */
            hp_t c, ch, d, s, sh;

            HP_SINHCOSH(xx + xx, &sh, &ch);
            HP_SINCOS(yy + yy, &s, &c);
            d = ch + c;             /* no leading digit loss when c < 0 */
            CXSET_(result, (fp_t)(sh / d), (fp_t)(s / d));
        }
    #endif /* FP_T_DIG == HP_T_DIG */

    }
```

We use the ratio of complex functions if subtraction loss is possible, and otherwise, we do the computation in real arithmetic.

We omit the straightforward code for the hyperbolic cosine and sine, and that for the simpler wrappers that provide the function families `CCOSH(z)`, `CSINH(z)`, and `CTANH(z)` for the native complex types.

The measured errors in the IEEE 754 `complex double` versions of the hyperbolic functions are shown in Figure 17.27 through Figure 17.29 on the next page. The errors in the hyperbolic cosine and sine rise to about 1.2 ulps because those functions are computed in normal working precision. The use of higher internal precision for the hyperbolic tangent produces results that are almost always correctly rounded.

17.17 Complex inverse hyperbolic functions

The inverse hyperbolic functions of complex arguments are plotted in Figure 17.30 through Figure 17.32 on page 516. They are defined in terms of the logarithms and square roots of complex arguments like this [Kah87]:

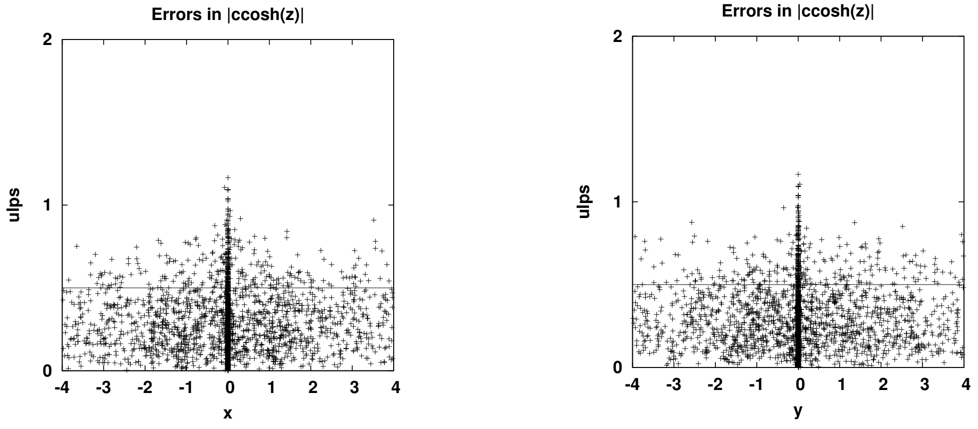


Figure 17.27: Errors in the `ccosh()` function.

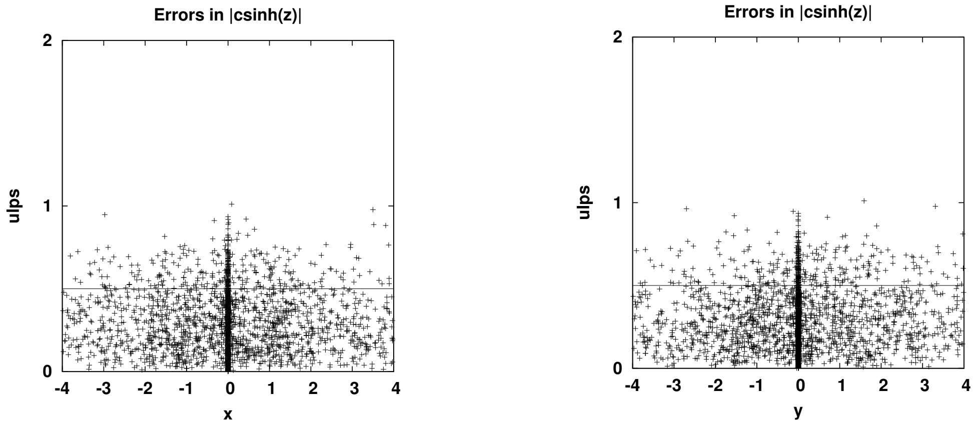


Figure 17.28: Errors in the `csinh()` function.

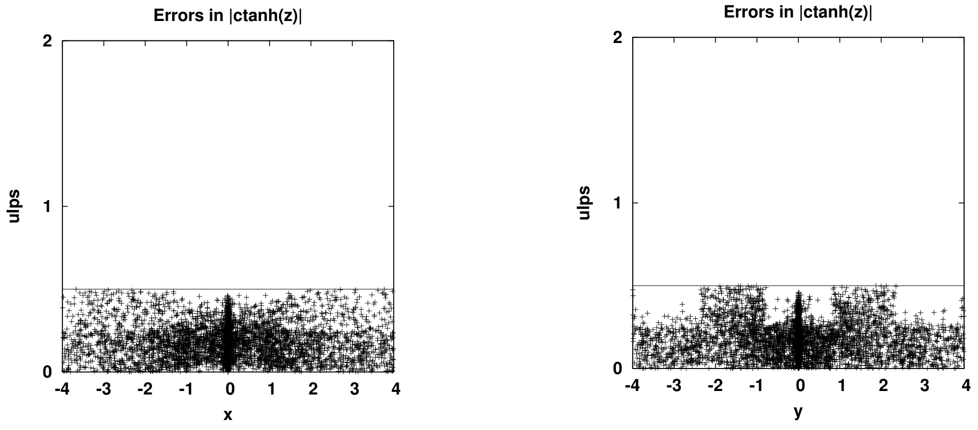


Figure 17.29: Errors in the `ctanh()` function.

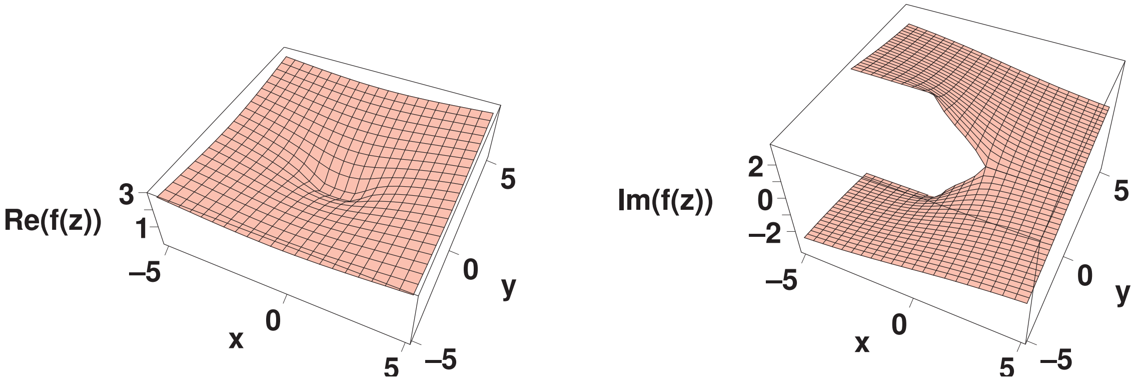


Figure 17.30: Real (left) and imaginary (right) surfaces of `cacosh(z)`, the complex inverse hyperbolic cosine function. There is a branch cut on the surface of the imaginary component for x in $(-\infty, +1)$.

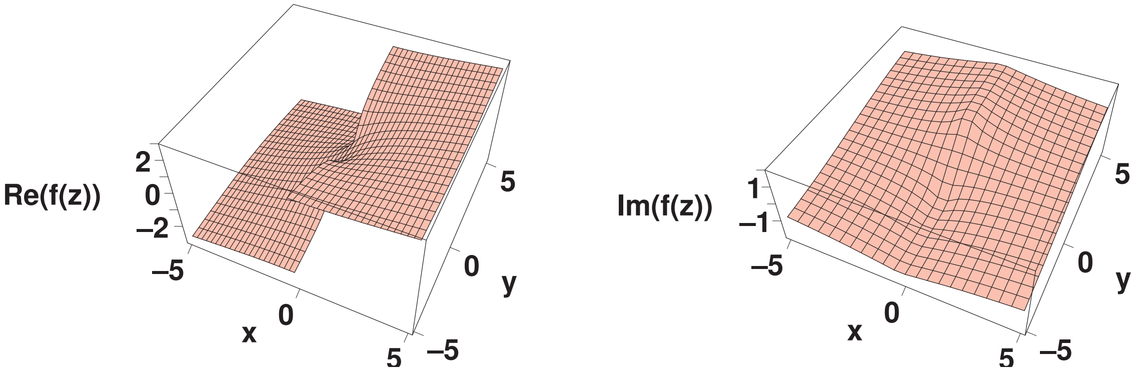


Figure 17.31: Real (left) and imaginary (right) surfaces of `casinh(z)`, the complex inverse hyperbolic sine function. There are branch cuts on the surface of the real component on the negative imaginary axis for y in $(-\infty, -1)$ and $(+1, +\infty)$.

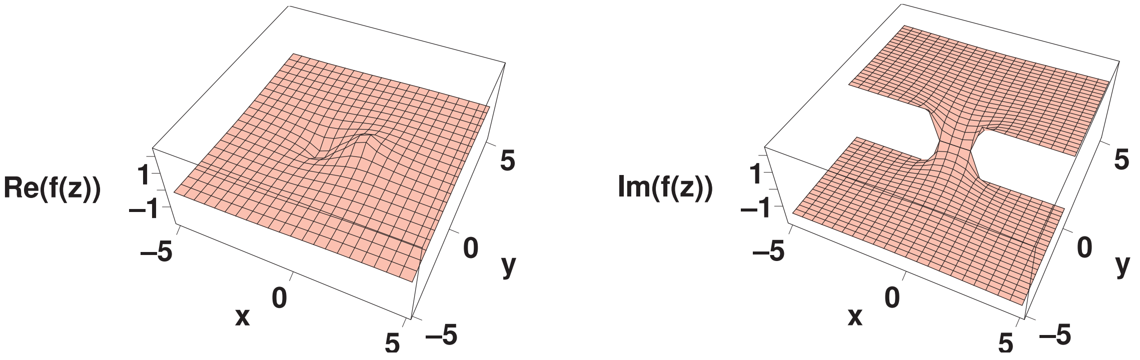


Figure 17.32: Real (left) and imaginary (right) surfaces of `catanh(z)`, the complex inverse hyperbolic tangent function. There are branch cuts on the surface of the imaginary component for x on $(-\infty, -1]$ and $[+1, +\infty)$.

$$\begin{aligned}\operatorname{acosh}(z) &= 2\log(\sqrt{(z-1)/2}+\sqrt{(z+1)/2}),\\ \operatorname{asinh}(z) &= \log(z+\sqrt{z^2+1}),\\ \operatorname{atanh}(z) &= \tfrac{1}{2}\log((1+z)/(1-z)).\end{aligned}$$

The inverse hyperbolic functions have these important symmetry relations:

$$\operatorname{acosh}(-z) = \pi i - \operatorname{acosh}(z), \qquad \operatorname{asinh}(-z) = -\operatorname{asinh}(z), \qquad \operatorname{atanh}(-z) = -\operatorname{atanh}(z).$$

Two of the functions are related to their inverse trigonometric companions with arguments multiplied by the imaginary unit, like this:

$$\operatorname{asinh}(z) = -\operatorname{asin}(zi)i, \qquad \operatorname{atanh}(z) = -\operatorname{atan}(zi)i.$$

The C99 Standard requires the special cases for the inverse hyperbolic functions tabulated in **Table 17.10** through **Table 17.12** on the next page.

The thorny computational issues for the complex logarithm and complex square root that we discuss in **Section 17.15** on page 507 suggest that we use the symmetry relations to compute two of the inverse hyperbolic functions from the inverse trigonometric functions. Our code for the inverse hyperbolic sine looks like this:

```
void
CXASINH(fp_cx_t result, const fp_cx_t z)
{   /* complex inverse hyperbolic sine: asinh(z) = -asin(z i) i */
    fp_cx_t zz;

    CXMULBYI_(zz, z);            /* zz = z * i */
    CXASIN(result, zz);
    CXSET_(result, CXIMAG_(result), -CXREAL_(result));
}
```

That for the inverse hyperbolic tangent is equally short.

For the remaining inverse hyperbolic function, we use a straightforward implementation of the definition:

```
void
CXACOSH(fp_cx_t result, const fp_cx_t z)
{   /* complex inverse hyperbolic cosine:
       acosh(z) = 2 * log(sqrt((z - 1)/2) + sqrt((z + 1)/2)) */
    fp_cx_t r, s, t, u, v, w;

    CXSET_(r, HALF * CXREAL_(z) - HALF, HALF * CXIMAG_(z));
    CXSET_(s, HALF * CXREAL_(z) + HALF, HALF * CXIMAG_(z));
    CXSQRT(t, r);                /* t = sqrt((z - 1)/2) */
    CXSQRT(u, s);                /* u = sqrt((z + 1)/2) */
    CXADD(v, t, u);              /* v = t + u */
    CXLOG(w, v);                 /* w = log(v) */
    CXADD(result, w, w);         /* result = 2 * w */
}
```

We do not need to use higher internal precision in that function to reduce argument errors in the square root and logarithm, because both functions are relatively insensitive to such errors. From **Table 4.1** on page 62, the error magnification of the square root is $\frac{1}{2}$, and that of the logarithm is $1/\log(z)$.

The wrapper functions `CACOSH(z)`, `CASINH(z)`, and `CATANH(z)` for the native complex types are simple, so we omit them.

The measured errors in the IEEE 754 `complex double` versions of the inverse hyperbolic functions are shown in **Figure 17.33** through **Figure 17.35** on page 519. The inverse hyperbolic cosine, `cacosh(z)`, is computed in working precision, and shows errors up to about 2 ulps, and in our tests, about one in a thousand random arguments produces errors between 2 and 10 ulps. The other two inverse hyperbolic functions use their inverse trigonometric companions, and because those functions use higher precision internally, the results are almost always correctly rounded.

Table 17.10: Special cases for the complex inverse hyperbolic cosine function, `cacosh(z)`, according to the C99 Standard [C99, TC3 §G.6.2.1, pages 474–475]. The value f is any *finite* value, g is any *positive-signed* finite value.

z	**acosh**(z)	**Remark**
`conj(z)`	`conj(cacosh(z))`	conjugation symmetry relation
$\pm 0 + 0i$	$+0 + \frac{1}{2}\pi i$	
$\pm 0 - 0i$	$+0 - \frac{1}{2}\pi i$	
$f + \infty i$	$+\infty + \frac{1}{2}\pi i$	
$f + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$-\infty + gi$	$+\infty + \pi i$	
$+\infty + gi$	$+\infty + 0i$	
$-\infty + \infty i$	$+\infty + \frac{3}{4}\pi i$	
$+\infty + \infty i$	$+\infty + \frac{1}{4}\pi i$	
$\pm\infty + \mathrm{NaN}i$	$+\infty + \mathrm{NaN}i$	
$\mathrm{NaN} + fi$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$\mathrm{NaN} + \infty i$	$+\infty + \mathrm{NaN}i$	
$\mathrm{NaN} + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	

Table 17.11: Special cases for the complex inverse hyperbolic sine function, `casinh(z)`, according to the C99 Standard [C99, TC3 §G.6.2.2, page 475]. The value f is any *finite* value, g is any *positive-signed* finite value, and h is any *nonzero* finite value.

z	**asinh**(z)	**Remark**
$-z$	$-$`casinh(z)`	function is odd
`conj(z)`	`conj(casinh(z))`	conjugation symmetry relation
$\pm 0 + 0i$	$\pm 0 + 0i$	
$\pm 0 - 0i$	$\pm 0 - 0i$	
$g + \infty i$	$+\infty + \frac{1}{2}\pi i$	
$f + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$+\infty + gi$	$+\infty + 0i$	
$+\infty + \infty i$	$+\infty + \frac{1}{4}\pi i$	
$+\infty + \mathrm{NaN}i$	$+\infty + \mathrm{NaN}i$	
$\mathrm{NaN} \pm 0i$	$\mathrm{NaN} \pm 0i$	
$\mathrm{NaN} + hi$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$\mathrm{NaN} + \infty i$	$\pm\infty + \mathrm{NaN}i$	result real sign unspecified
$\mathrm{NaN} + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	

Table 17.12: Special cases for the complex inverse hyperbolic tangent function, `catanh(z)`, according to the C99 Standard [C99, TC3 §G.6.2.3, pages 475–476]. The value f is any *finite* value, g is any *positive-signed* finite value, and h is any *nonzero* finite value.

z	**atanh**(z)	**Remark**
$-z$	$-$`catanh(z)`	function is odd
`conj(z)`	`conj(catanh(z))`	conjugation symmetry relation
$+0 \pm 0i$	$+0 \pm 0i$	
$+0 \pm \mathrm{NaN}i$	$+0 \pm \mathrm{NaN}i$	
$+1 \pm 0i$	$\infty \pm 0i$	raise *divbyzero* exception
$g + \infty i$	$+0 + \frac{1}{2}\pi i$	
$h + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$+\infty + gi$	$+0 + \frac{1}{2}\pi i$	
$+\infty + \infty i$	$+0 + \frac{1}{2}\pi i$	
$+\infty + \mathrm{NaN}i$	$+0 + \mathrm{NaN}i$	
$\mathrm{NaN} + fi$	$\mathrm{NaN} + \mathrm{NaN}i$	optionally raise *invalid* exception
$\mathrm{NaN} + \infty i$	$\pm 0 + \frac{1}{2}\pi i$	result real sign unspecified
$\mathrm{NaN} + \mathrm{NaN}i$	$\mathrm{NaN} + \mathrm{NaN}i$	

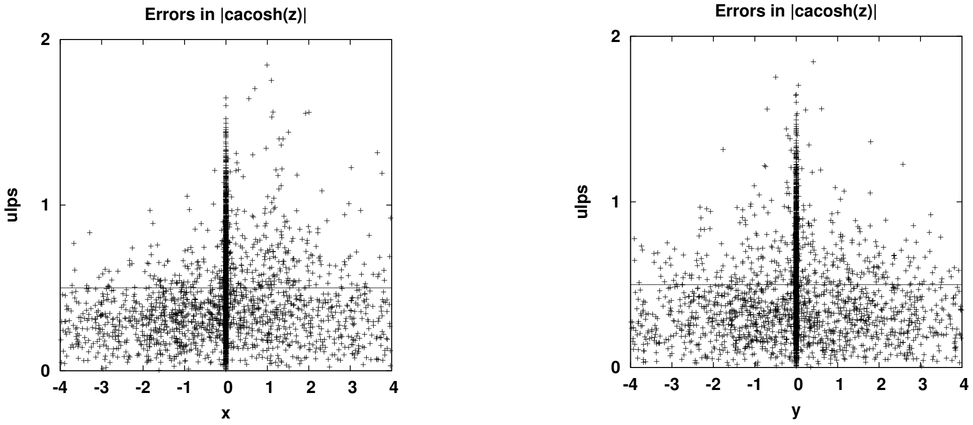


Figure 17.33: Errors in the `cacosh()` function.

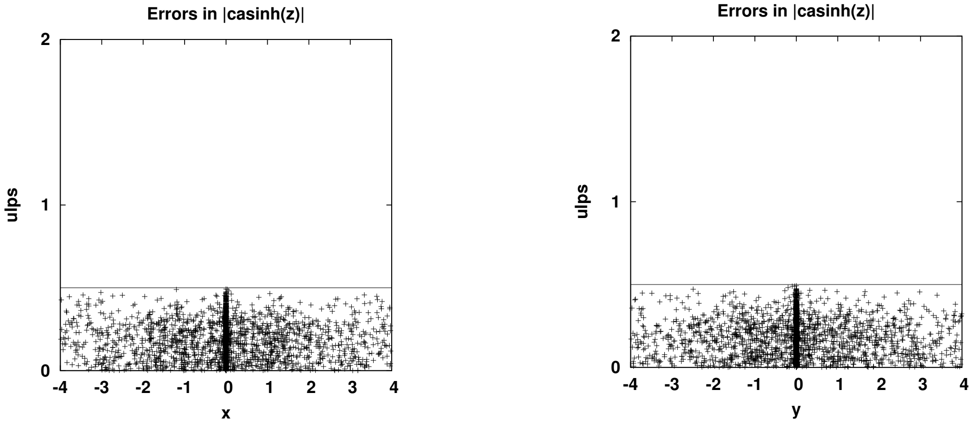


Figure 17.34: Errors in the `casinh()` function.

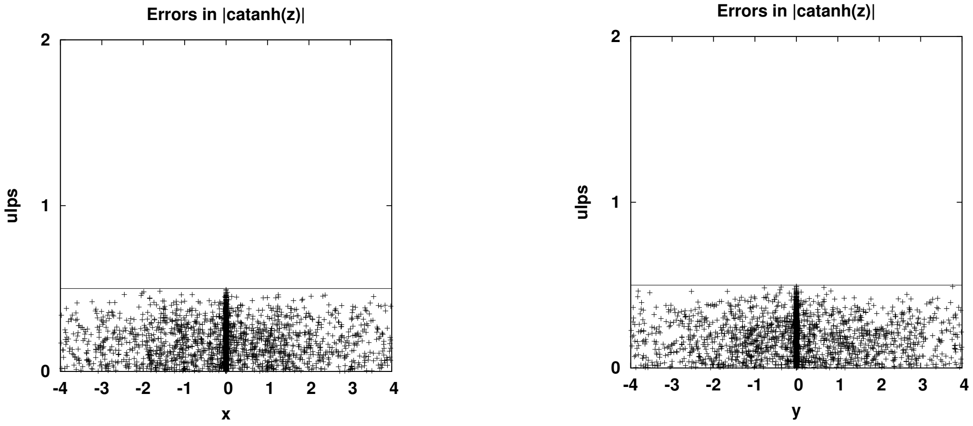


Figure 17.35: Errors in the `catanh()` function.

17.18 Summary

We treat the elementary functions of real arguments in several earlier chapters of this book, but we squeeze all of their companions for complex arguments into this single long chapter. There is clearly more to be said, and more to do, to improve the quality of the complex elementary functions. In many cases, detailed numerical analysis remains to be done, and new computational routes to some of the functions need to be found.

We commented in **Section 17.1** on page 475 about the scarcity of research publications in the area of complex arithmetic and numerical functions with complex arguments. The only textbook treatment of the complex elementary functions [Tho97] known to this author uses simple transcriptions of mathematical formulas into Fortran code with no concern whatever for accuracy, which disappears all too easily in subtraction loss in the formulas themselves, as well as in the complex arithmetic primitives on many systems.

In this chapter, we use the simplest algorithms when they are numerically satisfactory, but we also point out regions where subtraction loss, and poor-quality argument reduction in the real functions, can produce nonsensical results. To avoid the subtraction loss, or at least cover it up almost everywhere, we resort to summation of series expansions, or computation in higher precision.

Nevertheless, the excursions to higher precision are ineffective on those deficient systems from several vendors that fail to provide any numeric data type longer than the 64-bit format, even when the underlying hardware makes an 80-bit or 128-bit format available.

When higher precision is not available for intermediate computations, our algorithms may not produce the accuracy expected of a high-quality numerical function library. Although Fortran has supplied a complex data type since that extension was first introduced in the early 1960s (CDC 3600 48-bit computers in 1962, and the IBM 7090 36-bit family in 1963), no widely available publication, to this author's knowledge, has surveyed the quality of complex arithmetic primitives and elementary functions in a broad range of programming languages and platforms. The evidence of **Chapter 15** and this chapter is that extreme caution is advised in using complex arithmetic in any programming language — compiler code generation and library routines for complex arithmetic are likely to have received much less use and testing than for real arithmetic, and simplistic algorithms are likely to completely destroy accuracy in some regions of the complex plane, even if those regions happen to be important in *your* computations.

Besides those implementation issues of accuracy, there are also problems on platforms that lack signed zeros, or have them, but through compiler and library carelessness, make their signs uncertain.

Many of the complex elementary functions have branch cuts, yet textbooks, mathematical tables, and research literature may differ in their choices of where those branch cuts lie. Because branch cuts are a common feature of many complex functions, it is important to make sure that the mathematical sources and the software agree on branch-cut locations.

The decision in the C99 Standard to treat complex numbers as infinite as long as at least one of their components is Infinity complicates software design and behavior. The many tables in this chapter that are needed to describe the Standard-mandated behavior of the complex elementary functions for arguments that are signed zero, Infinity, or NaN suggest that implementations in practice are likely to differ in at least some of those cases. The differing behavior of complex division with zero denominators or infinite components that we discussed in **Section 15.9** on page 449 is also likely to cause grief when software using complex arithmetic is ported from one system to another.

Our assessment of the accuracy of our implementations of the complex elementary functions uses the error in the absolute value of the function value. When separate computational routes with real arithmetic exist for each of the components of the function result, both components should be computable with high accuracy. However, when the function result is computed with complex arithmetic, it should be expected that the component of smaller magnitude may have high relative error. That issue is unlikely to be considered by numerical programmers who are new to complex arithmetic, yet it may be of significant concern for applications of the computed complex numbers. It may also be a source of numerical discrepancies between runs of the same program with complex arithmetic on different platforms, or with different compilers and libraries on the same system.

The experience of writing the software for the two chapters on complex arithmetic in this book suggests that programmers who intend to write new software in complex arithmetic should use the *highest precision available*, even if it seems excessive for their problem. There are simply too many places in complex arithmetic where significant digits can silently vanish, even when similar computations in real arithmetic are known to be numerically stable and accurate.

18 The Greek functions: gamma, psi, and zeta

> GAMMA, *n.*: The third letter of the Greek alphabet, γ, Γ, represented historically by *c*, phonetically by *g*, in the Roman and English alphabet.
>
> DIGAMMA, *n.*: A name first found in the grammarians of the first century (so called because its form, F, resembles two gammas, Γ, set one above the other). A letter corresponding in derivation and alphabetic place to the Latin and modern European *F*, once belonging to the Greek alphabet.
>
> PSI (psē or sī), *n.*: A Greek letter, ψ, Ψ. It belongs to the Ionic alphabet, and stands for *ps* or *phs*. The character may be a modification of ϕ, Φ [Greek letter *phi*].
>
> ZETA, *n*: The sixth letter of the Greek alphabet, ζ, Z, corresponding to the English Z.
>
> — *New Century Dictionary* (1914).

Previous chapters of this book cover the *elementary functions* that are taught in secondary-school mathematics, and included in many programming languages. The gamma, psi, and Riemann zeta functions that we treat in this chapter are among the most important of the class known in mathematics and physics as *special functions*. As the class name suggests, they are more difficult, and less-commonly encountered, than the elementary functions. The gamma function, in particular, appears in many areas of mathematics, physics, and engineering. It arose in an area of mathematics that is today called *number theory*: the study of the properties of integers beyond the simple operations of schoolbook arithmetic.

The primary comprehensive summaries of mathematical relations for the gamma and psi functions are available in [AS64, Chapter 6], [SO87, Chapters 43 and 44], and [OLBC10, Chapter 5]. Other useful resources include [Bry08, §5.1 and §6.2], [Rai60, Chapter 2], [Luk69a, Chapter 2], [Luk69b, Chapter 14], [Olv74, Chapter 2], [WG89, Chapter 3], [Tem96, Chapter 3], and [AAR99, Chapter 1].

Discussions of the computation of the gamma and psi functions are available in [HCL$^+$68, Chapter 6], [Mos89, Chapter 5], [Bak92, Chapter 6], [ZJ96, Chapter 3], [Tho97, Chapter 6], as well as in several research papers [Ami62, Lan64, CH67, TS69, CST73, Amo83, Cod88b, Cod91, CS91, Cod93b, BB00].

The Riemann zeta function's properties are summarized in [AS64, Chapter 23] and [OLBC10, Chapter 25], but there is little published about its computation, apart from the description of its implementation in the Cephes library [Mos89, §7.10.2]. However, it figures in several popular descriptions of mathematical problems [Dev02, Der03, Haw05, Lap08, O'S07, Roc06, Sab03, dS03]. The verified solution of one of them, the famous *Riemann Hypothesis*, may win mathematical fame and fortune (a one million dollar prize) for some clever person [CJW06, Clay09].

18.1 Gamma and log-gamma functions

In 1729, Euler invented the gamma function, $\Gamma(x)$, as an extension of the factorial function to the entire real axis. The gamma function and the logarithm of its absolute value, $\log|\Gamma(x)|$, are available in C99, and implemented in the mathcw library, as the functions `tgamma()` and `lgamma()`. The functions are sketched in **Figure 18.1** on the next page.

The function name `tgamma()` stands for *true gamma*, to distinguish it from the function `gamma()` which was confusingly, and regrettably, used in some historical C implementations for $\log|\Gamma(x)|$. Those implementations return the sign of $\Gamma(x)$ in a global integer variable named `signgam`. C99's `lgamma()` does not provide that variable, but the mathcw implementation does.

Global variables are unusable in the presence of threads unless locking mechanisms are used at each access to the variables to restrict reading and writing to a single thread. Thus, in a threaded program, a test of `signgam` after a call

© Springer International Publishing AG 2017
N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_18

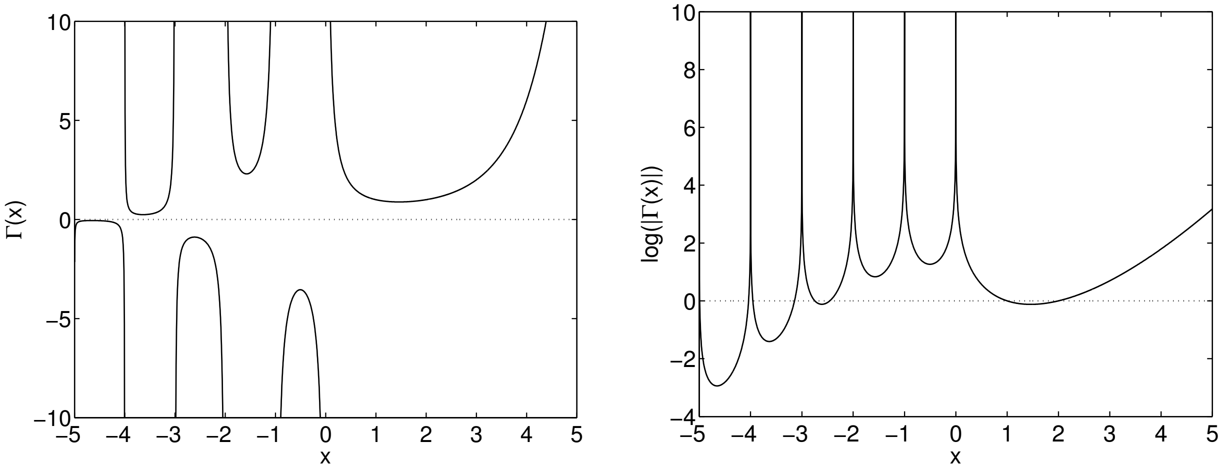


Figure 18.1: The gamma function and the logarithm of its absolute value. There are poles at zero and negative integers, and for negative arguments, $\Gamma(x)$ soon resembles a series of open rectangles alternating above and below the axis.

to `lgamma()` could get a value set by a call to that function in another thread. Some vendor implementations solve that problem by providing extensions to the C99 library:

```
float lgammaf_r       (float x,       int *psign);
double lgamma_r       (double x,      int *psign);
long double lgammal_r (long double x, int *psign);
```

Notice their irregular naming: the suffix `_r`, which stands for *reentrant*, follows any precision suffix on the name `lgamma`. Those functions return $\log|\Gamma(x)|$ as a function value, and store the sign of $\Gamma(x)$, $+1$ for positive and -1 for negative, in a thread-private local variable whose address is supplied as the second argument. The mathcw library provides those three functions, and their companions for other supported floating-point types.

Both `tgamma()` and `lgamma()` are challenging functions to compute accurately, because $\Gamma(x)$ grows rapidly for increasing positive x, and has poles at zero and the negative integers.

The gamma function increases so rapidly on the positive axis that the overflow limit is quickly reached, just before $x = 36$ (IEEE 754 32-bit binary arithmetic), $x = 72$ (64-bit), $x = 1756$ (80-bit and 128-bit), and $x = 20\,368$ (256-bit). However, the log of the absolute value of the gamma function is representable over most of the floating-point range.

Mathematically, the gamma function is never zero, but between negative integers, it gets so close to zero that floating-point underflow to subnormals or zero soon occurs. That happens for arguments below $x = -42$, -184, -1766, -1770, and $-20\,382$ for the extended IEEE 754 32-bit, 64-bit, 80-bit, 128-bit, and 256-bit binary formats, respectively. The corresponding limits for the four decimal formats are $x = -74$, -216, -2143, and $-310\,966$.

The behavior of $\Gamma(x)$ near negative integers is worth investigating numerically. **Table 18.1** on the facing page shows how sensitive the function is to tiny argument changes near the poles, and how quickly it approaches the x axis.

The gamma function has these special values and other relations:

$$
\begin{aligned}
\Gamma(\tfrac{1}{4}) &\approx 3.625\,609\,908\,221\,908\,\dots, \\
\Gamma(\tfrac{1}{3}) &\approx 2.678\,938\,534\,707\,747\,\dots, \\
\Gamma(\tfrac{1}{2}) &= \sqrt{\pi} \approx 1.772\,453\,850\,905\,516\,\dots, \\
\Gamma(\tfrac{2}{3}) &\approx 1.354\,117\,939\,426\,400\,\dots, \\
\Gamma(\tfrac{3}{4}) &\approx 1.225\,416\,702\,465\,177\,\dots, \\
\Gamma(1) &= 1,
\end{aligned}
$$

Table 18.1: Behavior of $\Gamma(x)$ near poles. The value $\delta = 10^{-70}$ is the negative machine epsilon in the extended 256-bit decimal format, and the tabulated values are for $\Gamma(x + n\delta)$. A unit step in the 70-th digit of the argument usually changes all digits of the function value, and the function values drop precipitously toward the x axis as x becomes more negative. The data are from a 100-digit Maple computation.

	x						
n	-1	-10	-20	-50	-100	-200	-1000
0	$-\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
1	-1.000e+70	2.756e+63	4.110e+51	3.288e+05	1.072e-88	1.268e-305	2.485e-2498
2	-5.000e+69	1.378e+63	2.055e+51	1.644e+05	5.358e-89	6.340e-306	1.243e-2498
3	-3.333e+69	9.186e+62	1.370e+51	1.096e+05	3.572e-89	4.227e-306	8.284e-2499
4	-2.500e+69	6.889e+62	1.028e+51	8.220e+04	2.679e-89	3.170e-306	6.213e-2499
5	-2.000e+69	5.511e+62	8.221e+50	6.576e+04	2.143e-89	2.536e-306	4.970e-2499

$$\begin{aligned}
\Gamma(1.461\,632\,144\,968\,\ldots) &= 0.885\,603\,194\,410\,\ldots, && \textit{only minimum for } x > 0,\\
\Gamma(\tfrac{3}{2}) &= \tfrac{1}{2}\sqrt{\pi} \approx 0.886\,226\,925\,452\,758\,\ldots,\\
\Gamma(2) &= 1,\\
\Gamma(\tfrac{5}{2}) &= \tfrac{3}{4}\sqrt{\pi} \approx 1.329\,340\,388\,179\,137\,\ldots,\\
\Gamma(n+\tfrac{1}{4}) &= (4n-3)!!!!\,\Gamma(\tfrac{1}{4})/4^n, && \textit{for integer } n \geq 0,\\
\Gamma(n+\tfrac{1}{3}) &= (3n-2)!!!\,\Gamma(\tfrac{1}{3})/3^n,\\
\Gamma(n+\tfrac{1}{2}) &= (2n-1)!!\,\Gamma(\tfrac{1}{2})/2^n,\\
\Gamma(n+\tfrac{2}{3}) &= (3n-1)!!!\,\Gamma(\tfrac{2}{3})/3^n,\\
\Gamma(n+\tfrac{3}{4}) &= (4n-1)!!!!\,\Gamma(\tfrac{3}{4})/4^n,\\
\Gamma(n+1) &= n!,\\
\Gamma(x+1) &= x\Gamma(x),\\
\Gamma(x+n) &= \Gamma(x)\prod_{k=0}^{n-1}(x+k), && \textit{for integer } n = 1, 2, 3, \ldots,\\
\Gamma(x)\Gamma(1-x) &= -x\Gamma(-x)\Gamma(x)\\
&= \pi/\sin(\pi x),\\
\Gamma(-|x|) &= -(\pi/\sin(\pi|x|))/\Gamma(|x|+1), && \textit{reflection formula,}\\
\Gamma(1-x) &= -x\Gamma(-x).
\end{aligned}$$

Here, the double-factorial notation, $n!!$, means the downward product of alternate integers, with obvious extensions for the triple factorial and quadruple factorial:

$$\begin{aligned}
n!! &= n \times (n-2) \times (n-4) \times \cdots \times (2 \text{ or } 1)\\
&= n(n-2)!!,\\
1!! &= 0!! = (-1)!! = 1,\\
n!!! &= n \times (n-3) \times (n-6) \times \cdots \times (3 \text{ or } 2 \text{ or } 1)\\
&= n(n-3)!!!,\\
2!!! &= 1!!! = 0!!! = (-1)!!! = (-2)!!! = 1,\\
n!!!! &= n \times (n-4) \times (n-8) \times \cdots \times (4 \text{ or } 3 \text{ or } 2 \text{ or } 1)\\
&= n(n-4)!!!!,\\
3!!!! &= 2!!!! = 1!!!! = 0!!!! = (-1)!!!! = (-2)!!!! = (-3)!!!! = 1.
\end{aligned}$$

Although the special formulas for $\Gamma(n + p/q)$ are unlikely to be exploited in the software implementation of the gamma function, they can nevertheless be useful for testing such code.

There are Taylor-series expansions that are essential for accurate computation, and for understanding the behavior of the function near its poles:

$$\begin{aligned}
\gamma &= 0.577\,215\,664\,901\,532\,860\,\ldots. && \textit{Euler–Mascheroni constant.}\\
\Gamma(-n+x) &= 1/(n!\,x)+\cdots, && \textit{for positive integer } n,\\
\Gamma(-1+x) &= (1/x)\big(-1+(\gamma-1)x+(\gamma-1-(\gamma^2/2+\pi^2/12))x^2+\cdots\big),\\
\Gamma(x) &= (1/x)\big(1-\gamma x+(\gamma^2/2+\pi^2/12)x^2-\cdots\big),\\
\Gamma(1+x) &= 1-\gamma x+(\gamma^2/2+\pi^2/12)x^2-\cdots,\\
\Gamma(2+x) &= 1+(1-\gamma)x+(-\gamma+\gamma^2/2+\pi^2/12)x^2+\cdots,\\
\log(\Gamma(x)) &= -\log(x)-\gamma x+(\pi^2/12)x^2+\cdots.
\end{aligned}$$

Because the Taylor-series coefficients grow rapidly in complexity, and contain transcendental constants instead of simple rational numbers, they are not amenable to run-time computation. Instead, we precompute them numerically to high precision in a symbolic-algebra system, and store them in a constant table as hexadecimal floating-point values for use with nondecimal arithmetic and C99 compilers, and also as decimal values. To give a flavor, here is the most important expansion, truncated to just three decimal digits:

$$\begin{gathered}
\Gamma(1+x) \approx 1-0.577\,x+0.989\,x^2-0.907\,x^3+0.981\,x^4-0.982\,x^5+\\
0.993\,x^6-0.996\,x^7+0.998\,x^8-0.999\,x^9+\cdots.
\end{gathered}$$

The coefficient magnitudes do not change much, so in practice, we use the more rapidly convergent series for the reciprocal of that function, and then invert the expansion, again showing only three decimal digits:

$$\begin{aligned}
\Gamma(1+x) &= 1/(1+\gamma x+(\pi^2/12-\gamma^2/2)x^2+\cdots)\\
&\approx 1/(1+0.577\,x-0.655\,x^2-0.0420\,x^3+0.166\,x^4-0.0421\,x^5-\\
&\qquad 0.00962\,x^6+0.00721\,x^7-0.000116\,x^8-0.000215\,x^9+\cdots).
\end{aligned}$$

We show in **Section 18.1.1.3** on page 529 how to remove most of the error in the final division.

For integer multiples of arguments, there are relations that can also be helpful for software testing:

$$\begin{aligned}
\Gamma(2x) &= (2^{2x}/(2\sqrt{\pi}))\Gamma(x)\Gamma(x+\tfrac{1}{2}),\\
\Gamma(3x) &= (3^{3x}/(2\pi\sqrt{3}))\Gamma(x)\Gamma(x+\tfrac{1}{3})\Gamma(x+\tfrac{2}{3}),\\
\Gamma(nx) &= \sqrt{\frac{2\pi}{n}}\,\frac{n^{nx}}{(2\pi)^{n/2}}\prod_{k=0}^{n}\Gamma(x+k/n), && n = 2, 3, 4, \ldots.
\end{aligned}$$

However, the powers n^{nx} and the products of n gamma functions of smaller arguments reduce the accuracy of the right-hand sides, so they need to be computed in higher precision.

The first relation involving $\Gamma(1-x)$ shows how to generate values of $\Gamma(x)$ for x in $(-1,0)$ from function values for x in $(0,1)$. The recurrence $\Gamma(x-1)=\Gamma(x)/(x-1)$ then provides a path to more negative arguments, at a cost of one multiply or divide for each unit step towards more negative x values. In practice, we start the recurrence from the interval $(1,2)$, where the gamma function varies least, and where we have an accurate rational polynomial fit that provides a small correction to an exact value. For even more negative values, the second relation for $\Gamma(x)\Gamma(1-x)$ provides a suitable path, although the argument reduction in $\sin(\pi x)$ requires careful handling to avoid accuracy loss. We treat that problem in detail in **Section 18.1.3** on page 531.

The gamma function has an asymptotic expansion for large positive arguments:[1]

$$\begin{gathered}
\Gamma(x) \asymp \sqrt{2\pi/x}\,x^x\exp(-x)\times\Bigg(1+\frac{1}{12x}+\frac{1}{288x^2}-\\
\frac{139}{51\,840x^3}-\frac{571}{2\,488\,320x^4}+\frac{163\,879}{209\,018\,880x^5}+\cdots\Bigg).
\end{gathered}$$

[1] See sequences *A001163*, *A001164*, *A046968* and *A046969* at `http://oeis.org/`, or [SP95, sequence M4878], for more terms, and sequences *A005146* and *A005147* for a continued-fraction expansion of $\Gamma(x)$.

That expansion is called *Stirling's series*, and it dates from 1730. By combining the first term of that series with the relation $\Gamma(n+1) = n\Gamma(n) = n!$, one can easily obtain *Stirling's approximation* for the factorial of large numbers:

$$n! \asymp \sqrt{2\pi n}\,(n/e)^n, \qquad \text{for } n \to \infty.$$

The approximation is sometimes written in logarithmic form with largest terms first:

$$\log(n!) \asymp (n + \tfrac{1}{2})\log(n) - n + \tfrac{1}{2}\log(2\pi), \qquad \text{for } n \to \infty.$$

The logarithm of the gamma function also has an asymptotic expansion:[1]

$$\begin{aligned}
\log(\Gamma(x)) &\asymp (x - \tfrac{1}{2})\log(x) - x + \log(2\pi)/2 + \sum_{k=1}^{\infty} \frac{B_{2k}}{2k(2k-1)x^{2k-1}} \\
&\asymp (x - \tfrac{1}{2})\log(x) - x + \log(2\pi)/2 + \frac{1}{12x} - \frac{1}{360x^3} + \frac{1}{1260x^5} - \\
&\quad \frac{1}{1680x^7} + \frac{1}{1188x^9} - \frac{691}{360\,360x^{11}} + \frac{1}{156x^{13}} - \frac{3617}{122\,400x^{15}} + \\
&\quad \frac{43\,867}{244\,188x^{17}} - \frac{174\,611}{125\,400x^{19}} + \frac{77\,683}{5796x^{21}} - \frac{236\,364\,091}{1\,506\,960x^{23}} + \cdots.
\end{aligned}$$

Here, the coefficients B_{2k} are the *Bernoulli numbers* that we met in the series for the trigonometric and hyperbolic tangents. We revisit them in **Section 18.5** on page 568.

Coefficient growth from the increasing Bernoulli numbers first sets in at the term with x^{19} in the denominator. Neither of the asymptotic expansions is particularly convenient for general precision-independent programming, because the coefficients do not have simple forms that can be generated on-the-fly. Ideally, they should be tabulated as rational numbers that are evaluated in floating-point arithmetic to machine precision at compilation time, but already at $k = 30$ in the expansion for $\log|\Gamma(x)|$, the numerator overflows in several floating-point systems, even though the ratio is representable:

$$\begin{aligned}
B_{60}/(60 \times 59) &= -1\,215\,233\,140\,483\,755\,572\,040\,304\,994\,079\,820\,246\,041\,491/ \\
&\qquad 201\,025\,024\,200 \\
&\approx -6.045 \times 10^{30}.
\end{aligned}$$

In practice, therefore, the coefficients must be tabulated as floating-point numbers that may be subject to base-conversion errors. In the mathcw library code, they are stored as hexadecimal and decimal values.

18.1.1 Outline of the algorithm for `tgamma()`

Implementing code for the gamma function that achieves the high accuracy of the elementary functions in the mathcw library is a challenging problem. The first draft of the gamma-function code for that library was based on historical work in Fortran [Cod91, Cod93b, BB00]. It was not satisfactory, because accuracy measurements showed regions with errors above 10 ulps, and error growth with increasing argument magnitude. The solution proved to be a radical restructuring of the code to compute the gamma function in several regions as the sum of an exact value and a small correction. The new code also uses error compensation and fused multiply-add operations to compute sums and products without error growth. We first outline the argument regions treated, and then describe some of the techniques in more detail. However, the code in `tgammx.h` is much too long to show in its entirety here — over 800 lines, excluding its extensive comments and coefficient tables.

The C99 Standard has this to say about the handling of errors in the computation of `tgamma(x)` [C99, §7.12.8.3, TC3]:

> *The `tgamma()` functions compute the gamma function of `x`. A domain error or range error may occur if `x` is a negative integer or zero. A range error may occur if the magnitude of `x` is too large or too small.*

We handle the computation of $\Gamma(x)$ separately in each of several regions, which we describe from left to right on the real axis, after treating the IEEE 754 special values:

x is a NaN : Set a domain error and set the result to that NaN.

x is $\pm\infty$: Set a range error and set the result to that Infinity. The behavior of the gamma function on the negative axis makes its sign indeterminate for arguments of large magnitude. Because those arguments are integer values where the function has poles to $\pm\infty$, one could reasonably argue that $\Gamma(-\infty)$ should evaluate to a NaN. However, it is likely to be more useful in practice to return an Infinity of the same sign as the argument.

$-\infty \le x \le -$XMAX : The value of XMAX is β^t for base β and t-digit precision. The argument magnitude is so large that it has no fractional part, so we are on one of the asymptotes for which $\Gamma(x) = \pm\infty$. The sign is not determinable, but we follow the handling of infinite arguments, set a range error, and set the result to $-\infty$, or if the host arithmetic does not support infinities, the most negative finite floating-point value. The mathcw library INFTY() function family hides the platform dependence.

x is a negative integer : Set a range error and set the result to $-\infty$, or if that is unsupported, the most negative floating-point value.

$-$XMAX $< x \le$ XCUT_REFLECT : Use the reflection formula for $\Gamma(-|x|)$ to move the computation onto the positive x axis. Because the formula involves $1/\Gamma(|x|+1)$, premature overflow in the denominator produces a result that has erroneously underflowed to zero. We eliminate that problem by extending the overflow cutoff, and computing an exactly scaled denominator. The final result is then carefully reconstructed to allow graceful underflow into the subnormal region. We defer the accurate computation of the trigonometric term in the reflection formula until **Section 18.1.3** on page 531.

The cutoff XCUT_REFLECT is set to -216, a value below which $\Gamma(x)$ underflows to zero in the IEEE 754 32-bit and 64-bit binary and decimal formats. That means that the reflection formula is used for those data types only when the result would be zero. The discussion of the error plots in **Section 18.1.2** on page 531 justifies that choice.

$-\frac{1}{2} < x < -0$: In this region, downward recurrence produces errors that are larger than desired, so we use separate Chebyshev polynomial fits like this:

$$\Gamma(x) = \begin{cases} -2\sqrt{\pi} + (x+\frac{1}{2})\mathcal{P}_1(x), & \text{for } x \text{ in } [-\frac{1}{2}, -\frac{1}{3}], \\ 1/x - 125/128 + \mathcal{P}_2(x), & \text{for } x \text{ in } [-\frac{1}{3}, -\frac{1}{9}], \\ 1/x - \gamma + \mathcal{P}_3(x), & \text{for } x \text{ in } [-\frac{1}{9}, 0]. \end{cases}$$

The polynomials contribute only a few percent of the total function value, and the division and sums are computed with two-part transcendental constants and error recovery.

XCUT_REFLECT $< x \le -$Taylor-series cutoff : Use downward recurrence with error compensation in the product to recover the function value from that of a reduced argument in $[1,2]$. We discuss the product computation in **Section 18.1.1.2** on page 528.

$-$Taylor-series cutoff $< x < 0$: Compute $\Gamma(x)$ from the fast-converging reciprocal Taylor series of $\Gamma(x+1)$ with compile-time constant coefficients, and one downward recurrence. We use series of orders 2, 4, 8, 16, and 32, depending on the argument size. Those series are longer than we normally employ, but the alternative algorithms are even more demanding.

The cutoffs are computed in the usual way as $\sqrt[n]{(\frac{1}{2}\epsilon/\beta)/c_n}$, where c_n is the final series coefficient, using nested SQRT() calls. However, because sign changes are possible, the cutoffs must be further restricted to prevent subtraction loss. Maple finds such loss for $|x| > \frac{1}{2}c_3/c_4 \approx 0.126$.

The code evaluates the Horner form in order of increasing term magnitudes in a way that allows compilers to exploit hardware fused multiply-add operations:

```
sum = c0[n];

for (k = n - 1; k >= 1; --k)
    sum = QFMA(sum, x, c0[k]);
xsum = x * sum;
err = ERRMUL(xsum, x, sum);
t = xsum * xsum / (ONE + xsum) - err;
result = t - xsum;
```

The computed value is $\Gamma(1+x) - 1$, which is why the loop ends at 1, instead of at 0. The final result is a small correction, no larger than -0.115, to an exact value, and there is never leading-digit loss in that subtraction. The macro `QFMA(x,y,z)` expands to `FMA(x,y,z)` when that operation is fast, and otherwise to `x * y + z`. The `ERRMUL()` operation (described later in **Section 18.1.1.2** on page 529) is essential for reducing the error in reciprocation. The final result requires a division by x, making it subject to overflow when x is tiny. **Section 18.1.1.4** on page 530 describes how that problem is handled.

$\boldsymbol{x = \pm 0}$: Set a range error, and set the result to $\pm\infty$, matching the sign of the argument. When signed zero is not supported, use $+\infty$ if available, or else the largest positive floating-point number.

$\boldsymbol{0 < x <}$ **Taylor-series cutoff** : Use the same Taylor-series code as for tiny negative arguments.

Taylor-series cutoff $\boldsymbol{\le x < 1}$: If x is sufficiently close to 1, set the result to $1 + (\Gamma(1+x) - 1)$, where the parenthesized expression is computed from a Taylor-series sum. Otherwise, use the recurrence relation $\Gamma(x) = \Gamma(1+x)/x$, and the polynomial approximation for the interval $[1,2]$, $\Gamma(1+x) \approx 1 + x\mathcal{P}(x)/\mathcal{Q}(x)$, to recover $\Gamma(x) \approx 1/x + \mathcal{P}(x)/\mathcal{Q}(x)$, and compute that expression as described later in **Section 18.1.1.3** on page 529.

$\boldsymbol{x = 1}$: Set the result to 1. This special case avoids the computation in the next case, and generates an exact result that in turn ensures accurate computation of $\Gamma(n+1) = n!$, where $n = 0, 1, 2, \ldots$. We saw in **Section 5.7** on page 112 that direct multiplication for computing factorials is exact in IEEE 754 64-bit binary arithmetic up to 22!, instead of the expected 18!, because of trailing zero bits.

$\boldsymbol{1 < x < 2}$: Compute $\Gamma(x) \approx \Gamma(z) + (x-z)^2\mathcal{P}(x)/\mathcal{Q}(x)$, where z is the location of the only minimum on the positive axis. Fitting about the minimum makes monotonicity more likely.

A Chebyshev fit of $\Gamma(x) - 121/128$ provides a fallback at higher precisions where Maple's minimax fits fail.

The code in this region computes $\Gamma(x) - 1$, rather than $\Gamma(x)$, to provide additional effective precision that is useful elsewhere.

$\boldsymbol{x = 2}$: Set the result to 1.

$\boldsymbol{x}$ **is an integer in** $[\mathbf{3}, \texttt{NMAX} + \mathbf{1}]$: Find $\Gamma(x)$ by lookup in a table of factorials. This case is optional, but is advisable because gamma-function software should provide fast, and correctly rounded, factorials. `NMAX` is the largest integer value for which $\Gamma(\texttt{NMAX}+1) = \texttt{NMAX}!$ is finite, although it may be reduced to economize on storage.

The factorial table can be a compile-time constant, or else can be generated at run time in an initialization block that is executed on the first call to the function, and does the computation in the highest available precision. Unless higher precision is used, table entries may not be correctly rounded, so we provide a constant table in both hexadecimal and decimal formats.

A reasonably portable choice of `NMAX` is 33, because its factorial is representable in IEEE 754 32-bit arithmetic, as well as in most historical single-precision designs. The `mathcw` code in `tgamm.h` supplies up to 301 finite compile-time entries in the factorial table, but terminates table entries as soon as the overflow limit is reached, using preprocessor conditionals with the `FP_T_MAX_10_EXP` constants derived from the standard header file `<float.h>`. That provides coverage for $n!$ values up to 10^{614}, so that factorials in the `float` and `double` types, and their counterparts in decimal arithmetic, are always correctly rounded.

$\boldsymbol{2 \le x \le}$ **`CUTOFF`** : The value of `CUTOFF` is the precision-dependent limit above which we can use the asymptotic expansion of $\Gamma(x)$. Let $n = \operatorname{floor}(x-1)$. Then $z = x - n$ lies in the interval $[1,2]$ where $\Gamma(z)$ is set by direct assignment, or can be found from the rational approximation. Then use upward recurrence from $\Gamma(z)$ to $\Gamma(z+n)$ to obtain $\Gamma(x)$ with just n extra multiplies. The product error compensation described in **Section 18.1.1.2** on the next page sharply reduces the error in that process.

CUTOFF $< x \leq$ **XBIG** : The value XBIG is the precision- and range-dependent limit at which $\Gamma(x)$ overflows. The asymptotic expansion for $\Gamma(x)$ is unsatisfactory for reasons discussed in Section 18.1.1.1. Therefore, find $\log(\Gamma(x))$ from its asymptotic expansion, and then recover $\Gamma(x)$ from $\exp(\log(\Gamma(x)))$. We examine the advisability of that traditional approach in Section 18.1.1.1 as well.

XBIG $< x \leq +\infty$: Set the result to $+\infty$ if available, or else the largest positive floating-point number.

In each region whose computation is required in another region, we compute a pair of exact high and approximate low parts whose implicit sum represents the result to somewhat more than working precision. That is inferior to using the next higher precision throughout, and complicates programming, but it reduces errors substantially, including those in the gamma functions of the highest-supported precision.

18.1.1.1 Asymptotic expansions

Most existing implementations of the gamma function use the asymptotic expansion for $\log(\Gamma(x))$, rather than that for $\Gamma(x)$. The former requires `EXP()` and `LOG()`, whereas the latter requires calls to `EXP()` and `POW()`. We saw in Chapter 14 that the power function is hard to make accurate for large arguments, so additional error can be introduced into $\Gamma(x)$. However, there is another problem: x^x can overflow even though the product $x^x \exp(-x)$ is representable. Error magnification in the power function and premature overflow make the asymptotic expansion for $\Gamma(x)$ unsuitable for floating-point computation.

There is yet another source of error that traditional algorithms overlook: error magnification in the exponential function for large arguments. Numerical and graphical experiments with changes in the value of CUTOFF show that it is better to avoid the asymptotic series altogether, and instead, pay the price of upward recurrence. Because the gamma function reaches the overflow limit so quickly, a huge number of multiplies is never needed. Only the future 256-bit format, where the overflow limit is reached at $x \approx 20\,368$, is likely to require a fallback to the asymptotic series.

The mathcw library code in tgammx.h accordingly sets the value of CUTOFF to XBIG, preventing execution of the private internal function that sums the asymptotic expansion for $\log(\Gamma(x))$. That improves the accuracy of the 64-bit function by about two decimal digits, and that of the 128-bit function by about four digits. When 256-bit arithmetic becomes available, numerical experiments can determine a suitable value for CUTOFF for that precision.

Instead of summing the asymptotic series, many previous implementations of the gamma function replace the series by a polynomial approximation in the variable $t = 1/x^2$, and use it for $x > 8$ (see further comments in Section 18.1.5). That lowers the error in the products needed for the recurrence relations, but cannot achieve good accuracy unless the code simulates higher internal precision for intermediate computations, as well as for the exponential, logarithm, and sine functions, as the excellent Sun Microsystems SOLARIS version does. Our delayed use of the asymptotic series ensures that it converges quickly, without the need for a separate polynomial approximation.

18.1.1.2 Recurrence-relation accuracy

When the recurrence relations are used, we require products of possibly many terms. Because the error in any partial product is magnified by the next multiplication, error accumulation is a serious problem. The fused multiply-add operation provides a way to reduce the product error to an almost-negligible size. Such products are needed in at least three places in tgammx.h, so a private internal function encapsulates the computation:

```
static fp_t
Gamma_product(fp_t r, fp_t e, fp_t x, int n, fp_t *p_err,
              fp_t *p_scale)
{   /* compute result = (r + e) * x * (x + 1) * ... * (x + n - 1)
       accurately; the true unscaled value is (result + *p_err) / (*p_scale) */
    fp_t err, result, scale, t, y;
    int i;

    err = e;
    result = r;
    scale = *p_scale;
    y = x;
    for (i = 1; i <= n; ++i)
    {
        fp_t d, prod;

        if (result > RESCALE_CUTOFF)
        {                                /* rescale to avoid premature overflow */
            err    *= FP_T_EPSILON;
            result *= FP_T_EPSILON;
            scale  *= FP_T_EPSILON;
        }

        prod = result * y;               /* approximate product */
        d = ERRMUL(prod, result, y);     /* error term */
        err = err * y + d;               /* optionally could be FMA(err,y,d) */
        result = prod;
        y += ONE;
    }

    t = result + err;
    err = ERRSUM(t, result, err);
    result = t;

    if (p_err != (fp_t *)NULL)
        *p_err = err;

    if (p_scale != (fp_t *)NULL)
        *p_scale = scale;

    return (result);
}
```

The value of RESCALE_CUTOFF is set in the one-time initialization block to the exact product FP_T_MAX * FP_T_EPSILON, which is sufficiently far from the overflow limit that multiplications are safe. The error in each partial product in the loop is accumulated, and then added to the final result. The effect of that correction is to make the error in the function result largely independent of the argument magnitude. Without it, the error is proportional to the magnitude of the largest factor.

The errors of elementary floating-point operations are recovered with the help of four convenient macros supplied by the private mathcw library header file prec.h, and used in more than a dozen places in tgammx.h:

```
/* a/b = quotient + ERRDIV(quotient,a,b) */
#define ERRDIV(quotient,a,b) (FMA(-(b), (quotient), (a)) / (b))

/* a*b = product + ERRMUL(product,a,b) */
#define ERRMUL(product,a,b) (FMA((a), (b), -(product)))

/* sqrt(a) = root + ERRSQRT(root,a) */
#define ERRSQRT(root,a)     (FMA(-(root), (root), (a)) / (root + root))

/* a + b = sum + ERRSUM(sum,a,b), where |a| >= |b| (unchecked!) */
#define ERRSUM(sum,a,b)     (((a) - (sum)) + (b))
```

They hide details whose inline exposure would make the code harder to program correctly, and read.

For more on the computation of accurate products, and their applications, see [Gra09].

18.1.1.3 Sums of rational numbers

In parts of the gamma-function code, we need to compute sums of the form $a/b + c/d$, where the first term has exactly representable parts, and dominates the inexact correction of the second part. We can reduce the errors of division by rewriting the expression as follows:

$$\begin{aligned}
z &= a/b + c/d, && \textit{desired function value,}\\
r &= \mathrm{fl}(a/b), && \textit{rounded quotient,}\\
a/b &= r + \delta, && \textit{exact quotient,}\\
\delta &= a/b - r \\
&= (a - rb)/b \\
&= \mathrm{fma}(-r, b, a)/b, && \textit{error correction for quotient,}\\
z &= r + (\delta + c/d) \\
&= r + (d\delta + c)/d \\
&= r + \mathrm{fma}(d, \delta, c)/d, && \textit{accurate function value.}
\end{aligned}$$

The first term in the final sum, r, can be taken as exact, and the second term provides a small correction, so the sum is often correctly rounded.

18.1.1.4 Avoiding catastrophic overflow

On pre-IEEE-754 systems, floating-point overflow may be a fatal error that terminates the job. Gamma-function computations are full of opportunities for overflow that IEEE 754 infinities handle gracefully. In `tgammx.h`, critical divisions that are subject to overflow are relegated to a function that avoids the division if it would cause possibly fatal overflow:

```
static fp_t
quotient(fp_t x, fp_t y)
{   /* return x / y without job-terminating overflow on non-IEEE-754 systems */
    fp_t result;

#if defined(HAVE_IEEE_754)

    result = x / y;

    if (ISINF(result))
        (void)SET_ERANGE(result);       /* optional in C99 */

#else

    if (is_fdiv_safe(x, y))
        result = x / y;
    else                                /* x / y overflows */
        result = SET_ERANGE((SIGNBIT(x) == SIGNBIT(y)) ? INFTY() : -INFTY());

#endif

    return (result);
}
```

The check for safe division is handled by this private function:

```
static int
is_fdiv_safe(fp_t x, fp_t y)
{   /* return 1 if x / y does not overflow, where, unchecked,
       x and y are normal and finite, else return 0 */
    int result;

    if (y == ZERO)
        result = 0;
    else if ( (x == ZERO) && (y != ZERO) )
        result = 1;
    else
    {
        fp_t xabs, yabs;

        xabs = QABS(x);
        yabs = QABS(y);

        if (yabs >= xabs)
            result = 1;
        else if (yabs >= ONE)
            result = 1;
        else
            result = (xabs <= (FP_T_MAX * yabs));
    }

    return (result);
}
```

18.1.2 Gamma function accuracy

Figure 18.2 graphs the errors over a small linear argument range near the region where rational polynomial fits are used, and over a wide logarithmic argument range to show the behavior near some of the underflow and overflow limits. Figure 18.3 on page 533 shows the measured errors in our implementation of the gamma function over a linear argument range. The plots demonstrate that the functions are accurate to the overflow limit as $x \to \pm 0$.

The larger errors in the bottom plots for the higher precisions are due to the use of the reflection formula below `XCUT_REFLECT` $= -216$. That is why we avoid that computational route for the more commonly used single- and double-precision gamma functions.

18.1.3 Computation of $\pi/\sin(\pi x)$

The reflection formula for $\Gamma(-|x|)$ allows us to move the computation from the difficult region of negative arguments to the better-behaved region of positive arguments, but it requires accurate evaluation of $\pi/\sin(\pi x)$. From Table 4.1 on page 62, the error-magnification factor for $\sin(x)$ is $x/\tan(x)$, so $\sin(\pi x)$ loses accuracy when $|x|$ is large, and also when $\tan(\pi x) \approx 0$, which happens when x is close to a whole number. Direct computation of $\sin(\pi x)$ as `sin(PI * x)` is therefore likely to be inaccurate, and its subsequent use in the reflection formula can produce wildly incorrect values of the gamma function of negative arguments.

In Section 11.7 on page 315, we discuss trigonometric functions of arguments in units of π, and show how to compute those functions accurately. The code in `tgammx.h` contains three separate approaches to the problem:

- Borrow code from the file `sinpix.h` (see Section 11.7 on page 315 and Section 11.7.1 on page 316).
- Call the `sinpi(x)` function directly if it is available, as it is in the mathcw library.
- Recognize that, for the gamma function, we later need the inverse of $\sin(\pi x)/\pi$, and that expression itself can be evaluated by a Taylor-series expansion for small x, and otherwise by a polynomial fit for x on the interval $[0, \frac{1}{2}]$. The symmetry relation $\sin(\pi(1-x)) = \sin(\pi x)$ that is evident from the graph of the sine function, and easily derived from its angle-sum formula, handles x values on the interval $[\frac{1}{2}, 1]$. Larger values of x are not required as arguments in the sine function, because we can use the reduction $\sin(\pi(n+r)) = (-1)^n \sin(\pi r)$. The computation of $\sin(\pi x)/\pi$ includes an error estimate so that the inverse can be made more accurate.

18.1.4 Why `lgamma(x)` is hard to compute accurately

Because the error-magnification factor for the logarithm is $1/\log(x)$ (see Table 4.1 on page 62), the `lgamma()` family is sensitive to argument errors when the logarithm is near zero. There are only two such instances on the positive

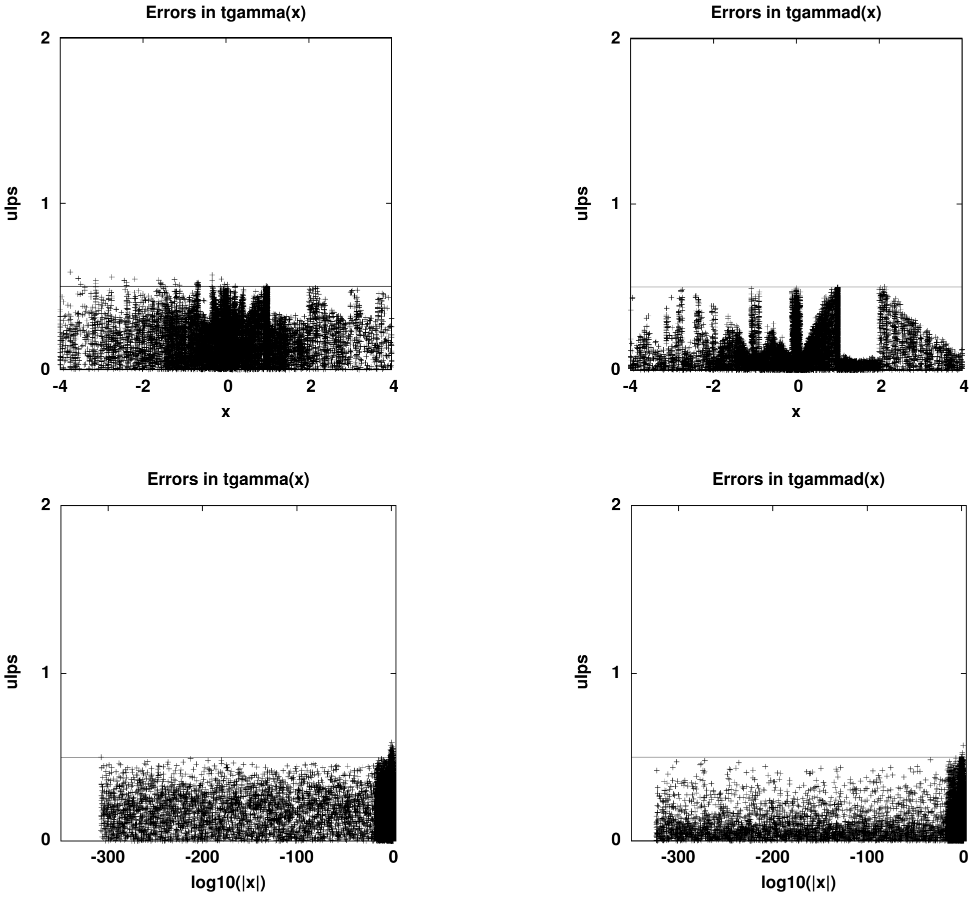


Figure 18.2: Errors in the `TGAMMA()` functions, measured against high-precision values from Maple. The top pair show the errors for small arguments, and the bottom pair graph the errors over a logarithmic argument range.

axis, but there are two zeros between every pair of consecutive integers below -2 on the negative axis, as graphed in **Figure 18.1** on page 522, and shown numerically in **Table 18.2** on page 534. We therefore expect larger errors in `lgamma(x)` for x in the region of those zeros, as well as for arguments near negative integers, where we already know the errors are large. Thus, an implementation of `lgamma(x)` that merely computes `log(tgamma(x))` in working precision is doomed in those regions, and will suffer premature overflow and underflow over much of the floating-point range where $\log(|\Gamma(x)|)$ is of reasonable magnitude.

Previous work on the gamma function and its logarithm cited at the beginning of this chapter mostly concentrates on the behavior of the functions for positive arguments, and many implementations are computationally poor on the negative axis. For the ordinary gamma function, we saw that either downward recurrence or the reflection formula provides a reasonable way to find $\Gamma(x)$ for negative arguments. For the logarithm of the absolute value of the gamma function, that is no longer the case, at least if we wish to achieve low *relative*, rather than *absolute*, error.

The problem on the negative axis is that, when $\Gamma(x) \approx \pm 1$, the function is steep, so tiny changes in its argument have a large effect on the function value. Even when the gamma-function value is determined accurately, its logarithm is roughly proportional to the difference of that value from one, because we know from the Taylor series that $\log(1+t) \approx t - \frac{1}{2}t^2 + \frac{1}{3}t^3 - \cdots$. Higher internal precision, a few digits more than twice working precision, is the only general solution to that problem. Because the zeros always occur singly, near any particular zero z, we can compute $\log(|\Gamma(x)|)$ accurately from $(x-z)\mathcal{P}(x)$, where $\mathcal{P}(x)$ is a suitable polynomial approximation. We use that technique to produce a low relative error for the first six zeros on the negative axis.

As x becomes more negative, the zeros of `lgamma(x)` get closer to the poles, and we see from **Table 18.2** that

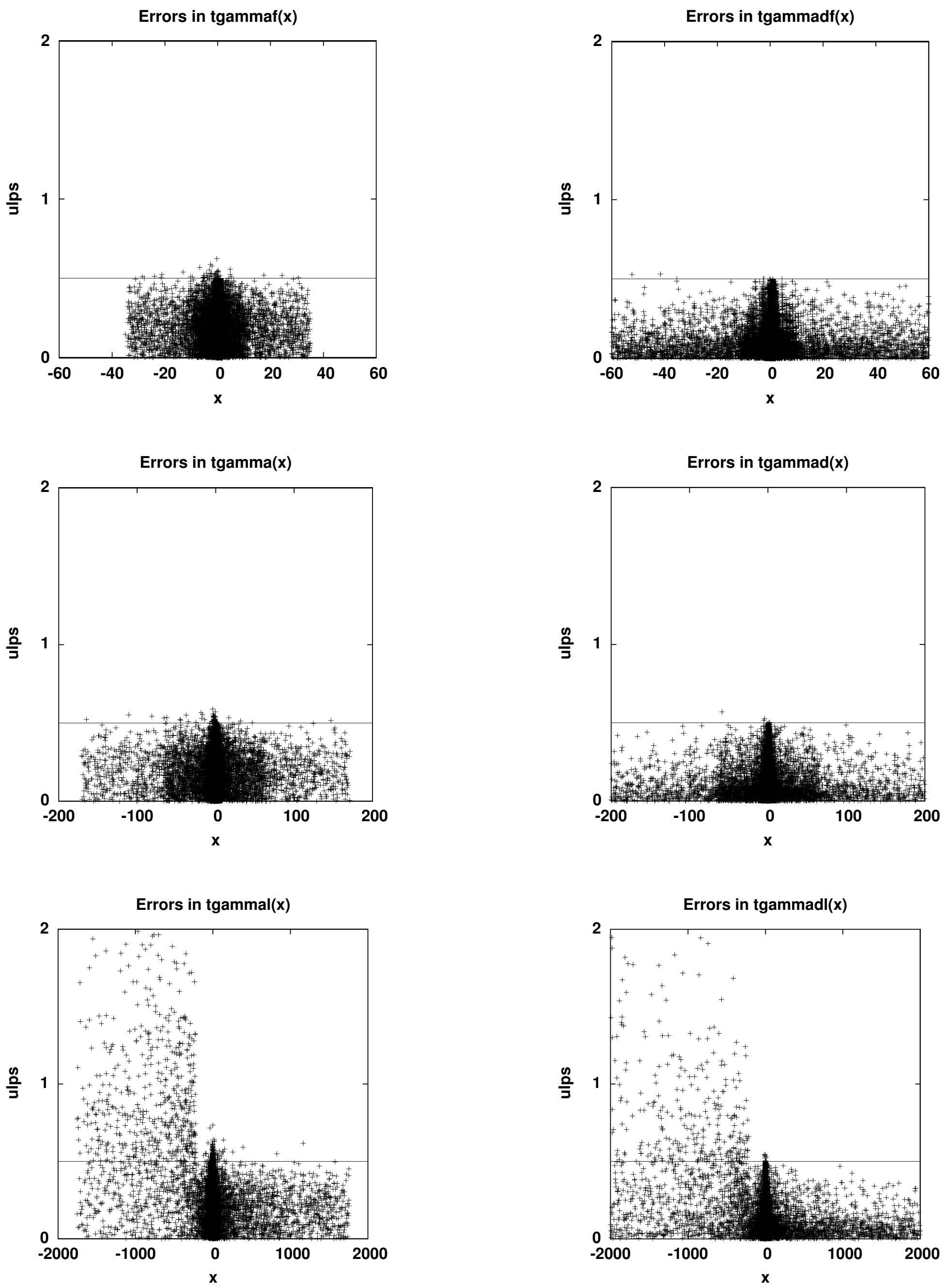


Figure 18.3: Errors in the `TGAMMA()` functions, measured against high-precision values from Maple over the representable range of the functions.

already near $x = -10$, the zeros hug the asymptotes closer than one single-precision machine epsilon, effectively making the zeros computationally inaccessible. Consequently, the largest measured errors from either recurrence or reflection are found for x in $[-5, -2]$. By contrast, near the two exact zeros on the positive axis, we can use Taylor-series expansions and other techniques to evaluate `lgamma(x)` with negligible accuracy loss.

Table 18.2: Truncated locations of the first 28 zeros of `lgamma(x)`, counting down from the largest argument. Except at the first two arguments, which are exact positive integer values, the remaining zeros all fall on the negative axis, and rapidly approach the asymptotes at negative integers.

$+2.000\,000\,000\,000\,000$	$-4.991\,544\,640\,560\,047$	$-8.000\,024\,800\,270\,681$	$-11.999\,999\,997\,912\,324$
$+1.000\,000\,000\,000\,000$	$-5.008\,218\,168\,322\,593$	$-8.999\,997\,244\,250\,977$	$-12.000\,000\,002\,087\,675$
$-2.457\,024\,738\,220\,800$	$-5.998\,607\,480\,080\,875$	$-9.000\,002\,755\,714\,822$	$-12.999\,999\,999\,839\,409$
$-2.747\,682\,646\,727\,412$	$-6.001\,385\,294\,453\,155$	$-9.999\,999\,724\,426\,629$	$-13.000\,000\,000\,160\,590$
$-3.143\,580\,888\,349\,980$	$-6.999\,801\,507\,890\,637$	$-10.000\,000\,275\,573\,013$	$-13.999\,999\,999\,988\,529$
$-3.955\,294\,284\,858\,597$	$-7.000\,198\,333\,407\,324$	$-10.999\,999\,974\,947\,890$	$-14.000\,000\,000\,011\,470$
$-4.039\,361\,839\,740\,536$	$-7.999\,975\,197\,095\,820$	$-11.000\,000\,025\,052\,106$	$-14.999\,999\,999\,999\,235$

18.1.5 Outline of the algorithm for `lgamma()`

The computation of $\log|\Gamma(x)|$ in `lgammx.h` calls `TGAMMA(x)` directly when possible, and borrows several of the private internal functions from `tgammx.h`. We require the sign of $\Gamma(x)$ for the global variable `signgam`, or the second argument of one of the `lgamma_r()` functions. That sign is easily determined: except for certain special cases noted in the algorithm description, $\Gamma(x)$ is negative if $x < 0$ and $\operatorname{floor}(x)$ is odd. Otherwise, it is positive. The `floor()` call is not needed when x is known to lie in a particular unit interval.

The C99 Standard has this description of error handling in the computation of `lgamma(x)` [C99, §7.12.8.4, TC3]:

> *The `lgamma()` functions compute the natural logarithm of the absolute value of gamma of `x`. A range error occurs if `x` is too large. A range error may occur if `x` is a negative integer or zero.*

Unlike the Standard's specification of the true gamma function, `tgamma(x)` (see Section 18.1.1 on page 525), there is no provision for a domain error at the poles. For consistency, we set a range error at the poles in both `lgamma(x)` and `tgamma(x)`, and reserve the domain error for a NaN argument.

The code in `lgammx.h` is too long to display here, but it follows these steps:

x is a NaN : Set `signgam` to $+1$, set a domain error, and set the result to that NaN.

$-\infty \leq x \leq$ **−`XMAX`** : The value of `XMAX` is β^t for base β and t-digit precision, and x is consequently a negative whole number corresponding to one of the asymptotes. Set `signgam` to -1 to follow the convention adopted for `tgamma(x)`, set a range error, and set the result to $+\infty$.

x is a negative whole number : Set `signgam` to -1, set a range error, and set the result to $+\infty$.

$x <$ **`XCUT_REFLECT`** : Use the reflection formula with careful computation of the sine factor, as described in Section 18.1.3 on page 531. Set the result to

$$\log(|\Gamma(-|x|)|) = \log(\pi)_{\text{hi}} + \big(\log(\pi)_{\text{lo}} - \big(\log(|\sin(\pi x)|) + \log(|\Gamma(|x| + 1)|)\big)\big)$$

where the logarithm of π is represented as a sum of exact high and approximate low parts, and the last logarithm is evaluated by the private internal function that computes the asymptotic series if $|x| + 1$ is above the series cutoff, and otherwise, by a recursive call to `LGAMMA_R()`.

The cutoff is adjustable, but we set it to -216, the same value used in `tgamma(x)`, unless upward adjustment is needed to prevent premature overflow. That modification is required for at least the `float` format because of its limited exponent range on all systems. The initialization code computes `XCUT_OVERFLOW` as the largest whole number whose factorial is below the overflow limit, and `XCUT_REFLECT` is set to the larger of -216 and −`XCUT_OVERFLOW`.

In this region, the zeros of $\log|\Gamma(x)|$ are too close to the asymptotes to be representable, so accuracy loss near the zeros is not a concern.

x is in $[$−`XBIG`, $-10]$: Compute `LOG(TGAMMA(x))`, but preferably in the next higher precision. Here, `XBIG` is the argument above which the computed $\Gamma(x)$ would overflow.

x is near one of the six zeros in $[-5, -2]$: Evaluate a Chebyshev approximation, $(x - z)\sum_{k=0}^{n} T_k(t)$, where $z = z_{\mathrm{hi}} + z_{\mathrm{lo}}$ represents the location of the zero to twice working precision, and the high part is subtracted first. The variable t lies on $[-1, +1]$, and is computed accurately from a simple expression of the form $ax + b$, where a and b are exactly representable small, and usually whole, numbers.

x is in $[-10, -\frac{1}{2}]$: Use downward recurrence to find $\Gamma(x) = \Gamma(x+n)/(x \times (x+1) \times \cdots \times (x+n-1))$, where n is chosen so that $x + n$ is in $[1, 2]$. The numerator is evaluated in the form $1 + (\Gamma(x+n) - 1)$, with the parenthesized expression computed to working precision from the rational polynomials used in `tgammx.h`, or from Taylor-series summation near the end points. Compute the denominator product with error correction. Then obtain $\log(|\Gamma(x)|)$ from $\mathrm{log1p}(\Gamma(x+n) - 1) - \log(|\mathrm{product}|)$.

$x < -$Taylor-series cutoff : Use the next higher precision, if possible, to compute the result as $\log(\Gamma(x))$. That effectively handles both the accuracy loss near the asymptotes, and the cumulative errors in the recurrence and reflection formulas. Otherwise, we have to accept large relative errors near the zeros below those treated by the Chebyshev approximations.

$x = \pm 0$: Set `signgam` to the sign of x, set a range error, and set the result to $+\infty$.

$|x| <$ Taylor-series cutoff : Sum the Taylor series $\log|\Gamma(x)| \approx -\log(x) - \gamma x + \cdots$, where series of orders 2, 4, 8, 16, and 32 are chosen according to the argument magnitude, and the coefficients are taken from a compile-time constant table. Replace the leading coefficient by a sum of high and low parts for better accuracy.

As with the series for `tgamma(x)` (see **Section 18.1.1** on page 526), after determining the cutoffs in the usual way, they must be further restricted to prevent subtraction loss. Maple finds such loss for $|x| > \frac{1}{2}c_0/c_1 \approx 0.351$.

Taylor-series cutoff $\leq x$ and $x < \frac{1}{2}$: Compute $\mathrm{log1p}(\Gamma(x) - 1)$, where the argument difference is computed directly from a rational polynomial approximation.

x is in $(\frac{1}{2}, \frac{3}{4})$: Compute a minimax rational polynomial approximation, $\mathcal{R}(t)$, for t in $[0, 1]$, such that

$$\log(\Gamma(\tfrac{1}{2} + \tfrac{1}{4}t)) = (1 - t)\log(\Gamma(\tfrac{1}{2})) + t\log(\Gamma(\tfrac{3}{4})) + t\mathcal{R}(t).$$

That is a linear interpolation over the interval, plus a small correction that contributes less than 8% to the function value. Represent the gamma-function constants as sums of exact high and approximate low parts. Use fused multiply-add operations to compute the terms containing products with the high parts of the two constants.

x is near $+1$ or $+2$: Compute the result from a Taylor-series expansion for $\log(\Gamma(x))$ about that endpoint, with a pair representation of the low-order coefficient. That produces accurate results near the two zeros on the positive axis.

x in $[\frac{3}{4}, 1]$: Compute $\mathrm{log1p}(\Gamma(x) - 1)$, where the argument difference is computed directly from a rational polynomial approximation.

x in $[1, 4]$: Compute one of several polynomial approximations:

$$\begin{aligned}
\log(\Gamma(z)) + (x - z)^2\mathcal{R}_1(x - z), &\qquad \textit{for } x \textit{ in } [1.35, 1.65], \\
\log(\Gamma(\tfrac{5}{4})) + (x - \tfrac{5}{4})\mathcal{R}_2(x - \tfrac{5}{4}), &\qquad \textit{for } x \textit{ in } [1, \tfrac{3}{2}], \\
\log(\Gamma(\tfrac{7}{4})) + (x - \tfrac{7}{4})\mathcal{R}_3(x - \tfrac{7}{4}), &\qquad \textit{for } x \textit{ in } [\tfrac{3}{2}, 2], \\
(x - 2)\log(\Gamma(\tfrac{5}{2})) + (x - \tfrac{5}{2})\mathcal{R}_4(x - \tfrac{5}{2}), &\qquad \textit{for } x \textit{ in } [2, 3], \\
(4 - x)\log(\Gamma(3)) + (x - 3)\log(\Gamma(4)) + & \\
(x - 3)\mathcal{R}_5(x - 3), &\qquad \textit{for } x \textit{ in } [3, 4].
\end{aligned}$$

In the first, $z \approx 1.461$ is the position of the only minimum of the gamma function on the positive axis. Expansion about that point encourages the fit to be monotonic on either side.

In each case, represent the transcendental constants as pair sums, and use error correction in the evaluations. In the region gaps, use $\mathrm{log1p}(\Gamma(x) - 1)$ or $\log(\Gamma(x))$, whichever is more accurate.

$x < \max(\texttt{XBIG}, \texttt{CUTOFF})$: Compute the result as `LOG(TGAMMA(x))`.

$\texttt{CUTOFF} \le x < +\infty$: If $\log(x) \ge \texttt{FP\_T\_MAX}/x$, set a range error and set the result to $+\infty$.

Otherwise, sum the asymptotic expansion of $\log(\Gamma(x))$, omitting those parts of the expansion that contribute negligibly to the result. The sum of reciprocal powers is not needed at all when $x > \sqrt{1/\epsilon}$.

When possible, compute the logarithm in the next higher precision, and then split that value into a sum of high and low parts in working precision. It is then possible to achieve almost perfect rounding of `LGAMMA(x)` in the asymptotic region.

Many implementations of `lgamma(x)` replace the asymptotic expansion by a polynomial fit in the variable $1/x^2$, or use less-well-known, but faster-converging, sums due to Lanczos [Lan64]. The Lanczos sums are valid also for the gamma function of complex arguments, a function that we do not treat in this book. One of his sums, truncated to two terms, takes the form

$$\Gamma(z) = (z+1)^{z-\frac{1}{2}} \exp(-(z+1))\sqrt{2\pi}\left(0.999\,779 + \frac{1.084\,635}{z}\right).$$

It has a relative error below 2.4×10^{-4} for *all* complex z with positive real part. For `lgamma(x)`, in order to avoid premature overflow, replace the right-hand side with the sum of the logarithms of each of the factors, and use the `LOG1P()` family when the logarithm argument is near one.

Because our code needs to cater to many different floating-point precisions and both binary and decimal bases, we find it simplest to use the asymptotic expansion directly. Unlike most asymptotic expansions, this one can be used to reach full accuracy for all of the precisions that we support, as long as the cutoff is chosen appropriately. We never need more than 30 terms, and usually, just a few terms suffice.

$x = +\infty$: Set a range error and set the result to $+\infty$.

18.1.6 Log-gamma function accuracy

An early draft of the code for `lgamma(x)` used fewer polynomial approximations, but showed unacceptably high errors. Those errors were gradually reduced by introducing more polynomial fits, and sometimes, several alternative fitting functions had to be tried and their accuracy compared. The final code is, alas, horridly complicated by the many different approximations needed to drive the errors down. The gamma function, and its logarithm, are good examples of how higher precision in intermediate computations can eliminate many computational difficulties, and allow use of much simpler code.

Figure 18.4 on the next page shows the measured errors in our implementations of the log-gamma function over a small linear interval where the problems near the zeros on the negative axis are most visible, and over a logarithmic interval. Figure 18.5 on page 538 shows the errors in the binary and decimal functions over a wider linear interval.

Without the special handling of the zeros in $[-5, -2]$ with polynomial fits, and use of higher precision on the negative axis, the error plots show huge spikes near the zeros where the errors grow arbitrarily large. The random arguments used in the plots are not dense enough to sample close to the zeros. The lower accuracy of the reflection formula is evident in Figure 18.5 where the initially low errors on the negative axis rise sharply when the reflection cutoff is reached.

18.2 The `psi()` and `psiln()` functions

The function `psi(x)` computes the derivative of the logarithm of the gamma function:

$$\begin{aligned}\psi(x) &= d(\log\Gamma(x))/dx,\\ &= (d\Gamma(x)/dx)/\Gamma(x),\\ &= \Gamma'(x)/\Gamma(x).\end{aligned}$$

From that, we see that the derivative of the gamma function, $\Gamma'(x)$, is the product $\psi(x)\Gamma(x)$. The gamma function can be negative for negative arguments, but its logarithm is then not real-valued. Consequently, the second formula is regarded as the main one in real arithmetic, and $\psi(x)$ is then defined for both negative and positive arguments.

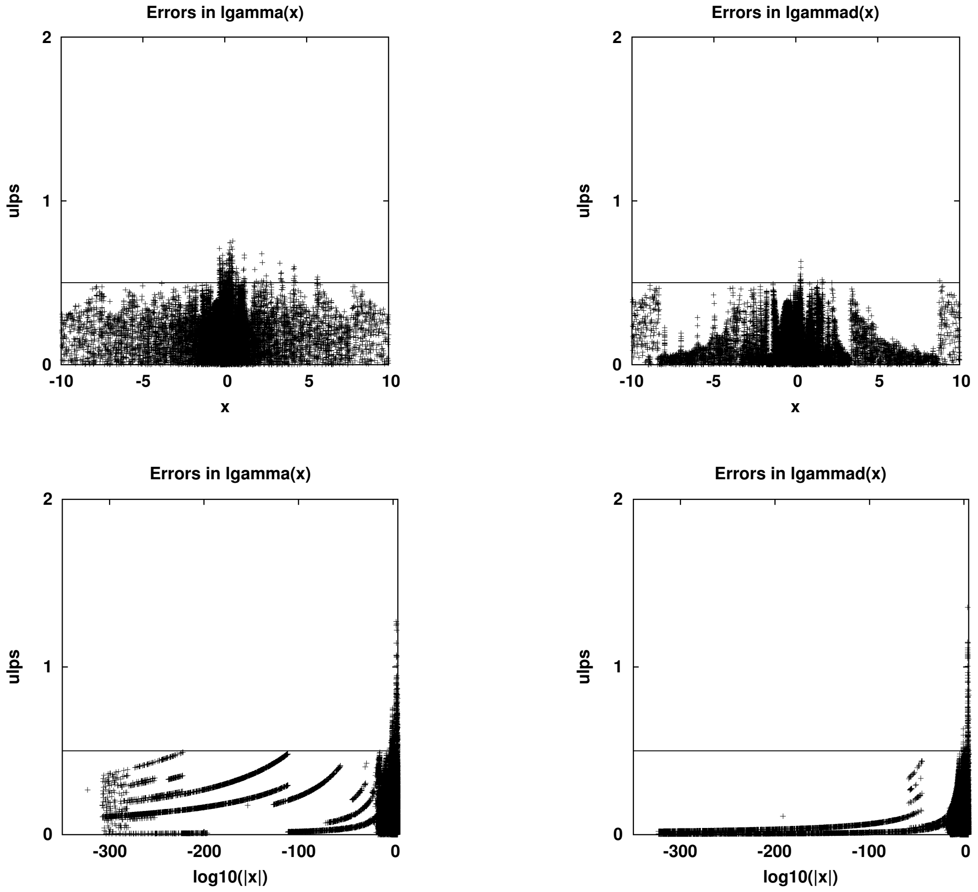


Figure 18.4: Errors in the `LGAMMA()` functions, measured against high-precision values from Maple. The top pair show the errors for small arguments, and the bottom pair graph the errors over a logarithmic argument range.

The mathematical history of the psi function is cloudy, but seems to be traceable back to 1809 in writings of Legendre.

In older books, the psi function is called the *digamma function*, after an obsolete letter in ancient Greek. Higher derivatives of $\log(\Gamma(x))$ produce the *polygamma functions* that we treat later in **Section 18.3** on page 547. The name *polygamma* is a mathematical invention; there is no such Greek letter. It may have been first used in 1919 as an extension of *digamma function* and *trigamma function* [Pai19].

The Maple symbolic-algebra system provides the psi function as `Psi(z)`. Mathematica supplies `PolyGamma[z]`. MATLAB has only the numerical function `psi(x)`, and restricted to nonnegative arguments. Maxima has the numerical (big-float) function `bfpsi0(z,d)`, where d is the precision in decimal digits, as well as the symbolic function `psi(x)`. MuPAD, PARI/GP, and REDUCE have `psi(z)`.

As the argument z suggests, those functions can be defined for complex arguments, but we do not provide that extension in the mathcw library. Although some of those languages use capitalized function names, modern notation uses lowercase $\psi(z)$, rather than uppercase $\Psi(z)$.

As x increases, $\psi(x)$ approaches $\log(x)$ reasonably quickly, so a companion function, `psiln(x)`, computes the difference $\psi(x) - \log(x)$ accurately. Such a function is needed, because direct computation of the difference suffers bit loss starting at $x \approx 2.13$; ten bits are lost at $x \approx 109$, and twenty are lost at $x \approx 48\,600$. The `psi()` and `psiln()` functions are illustrated in **Figure 18.6** on page 539.

Because of its approach to the logarithm on the positive axis, $\psi(x)$ never gets very big as x increases: at the argument overflow limits in the five extended IEEE 754 binary formats, $\psi(x)$ is about 89, 710, 11 357, 11 357, and

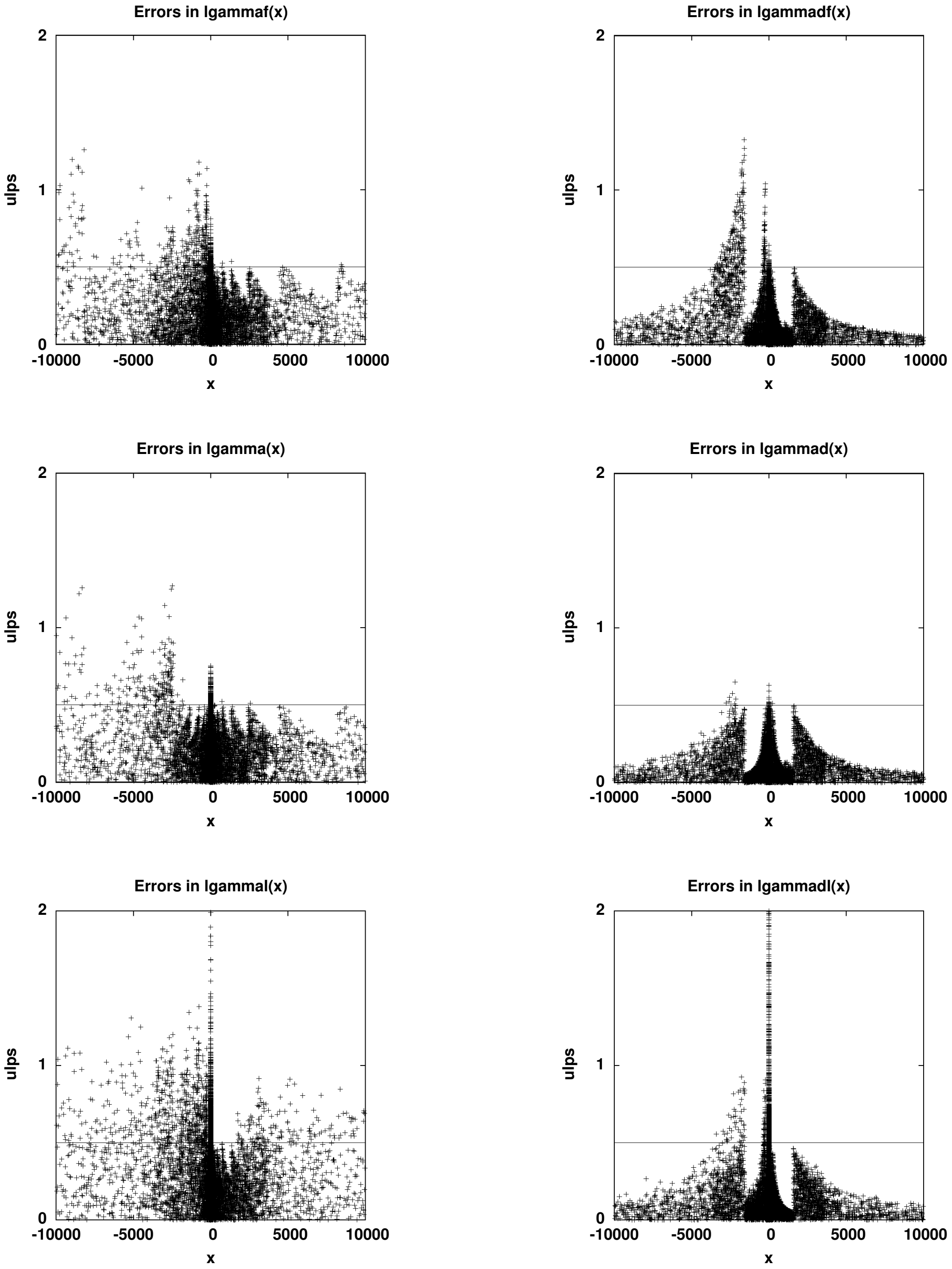


Figure 18.5: Errors in the LGAMMA() functions, measured against high-precision values from Maple over a wide argument range.

181 704, respectively.

18.2.1 Psi function poles and zeros

There are poles of $\psi(x)$ at zero and at negative integral values of x. There is only a single zero on the positive axis, but on the negative axis, there is one zero between each pair of integral values of x, and those zeros move closer to

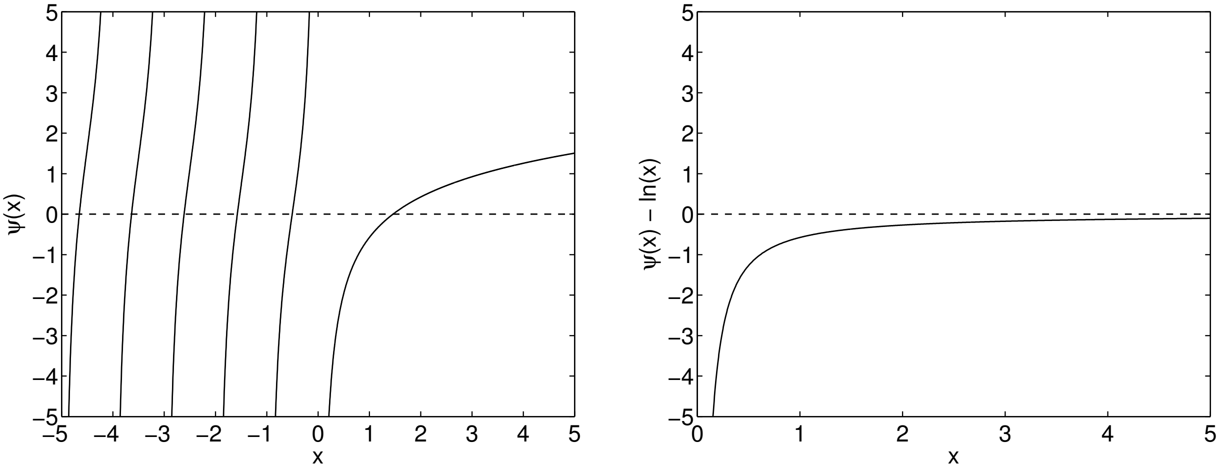


Figure 18.6: The `psi()` and `psiln()` functions, computed from a wrapper function that extends MATLAB's deficient `psi(x)` onto the negative axis. The `psiln()` function is undefined for $x < 0$, and our implementation returns a NaN in that case for IEEE 754 arithmetic.

the left asymptote as x becomes more negative. **Table 18.3** on the next page gives an idea of how they behave.

Counting left from the only zero on the positive axis, the zeros occur at these approximate argument values:

$$x_k \approx (1/\pi)\,\mathrm{atan}(\pi/\log(k)) - k, \qquad k = 0, 1, 2, \ldots .$$

That simple formula predicts the zeros to an accuracy of at least two decimal places in the argument for $k > 4$.

In our software, we require accurate knowledge of only the first root, which we denote by

$$z = 1.461\,632\,144\,968\,362\,341\,262\,659\,542\,325\,721\,328\,468\,196\ \ldots .$$

Because the derivative of $\psi(x)$ is everywhere positive and nonzero, the error-magnification formula $x\psi'(x)/\psi(x)$ (see **Section 4.1** on page 61) tells us that the relative error in `psi(x)` increases without bound near zeros of the function.

18.2.2 Recurrence relations for psi functions

Recurrence relations provide one important way to compute the psi function from knowledge of its value on a single unit interval, possibly represented there by a rational polynomial approximation, or a truncated series:

$$\begin{aligned}
\psi(x+1) &= \psi(x) + 1/x, \\
\psi(x-1) &= \psi(x) - 1/(x-1), \\
\psi(x+n) &= \psi(x) + \sum_{k=0}^{n-1} \frac{1}{x+k}, && \text{for } n = 1, 2, 3, \ldots, \\
&= \psi(x) + \frac{1}{x} + \frac{1}{x+1} + \cdots + \frac{1}{x+n-1}, && \text{upward recurrence,} \\
\psi(x-n) &= \psi(x) - \sum_{k=0}^{n-1} \frac{1}{x-n+k}, && \text{for } n = 1, 2, 3, \ldots, \\
&= \psi(x) - \frac{1}{x-n} - \frac{1}{x-n+1} - \cdots - \frac{1}{x-1}, && \text{downward recurrence.}
\end{aligned}$$

The recurrence relations for the gamma function are products, but for the psi function, they are sums, and we therefore have to consider the possibility of significance loss when two similar numbers of opposite sign are added. On the positive axis, we have $\psi(x) > 0$ only for $x > z$.

Table 18.3: Truncated locations of the first 32 zeros of `psi(x)`, and some later ones, counting down from the largest argument. All but the first lie on the negative axis. Compared to the zeros of `lgamma(x)`, these zeros are slow to approach the asymptotes at negative integers, and they consequently remain a serious impediment to achieving high accuracy in `psi(x)` for negative arguments.

1.461 632 144 968 362	−7.687 788 325 031 626	−15.730 988 906 332 882	−23.752 362 937 385 182
−0.504 083 008 264 455	−8.695 764 163 816 401	−16.734 356 723 955 736	−24.754 370 257 822 971
−1.573 498 473 162 390	−9.702 672 540 001 864	−17.737 475 159 977 589	−25.756 275 080 771 035
−2.610 720 868 444 145	−10.708 740 838 254 145	−18.740 374 944 780 100	−26.758 086 286 661 366
−3.635 293 366 436 901	−11.714 133 061 228 954	−19.743 081 672 590 218	−27.759 811 695 826 706
−4.653 237 761 743 142	−12.718 971 025 749 207	−20.745 616 863 607 526	−28.761 458 227 264 866
−5.667 162 441 556 886	−13.723 347 457 363 827	−21.747 998 768 201 130	−29.763 032 029 127 463
−6.678 418 213 073 427	−14.727 334 416 018 529	−22.750 242 984 306 060	−30.764 538 586 718 172
	−100.809 855 037 646 773	−1 000 000.928 827 867 117 256	
	…	…	
	−1000.864 158 855 781 428	−10 000 000.938 726 159 976 686	
	…	…	
	−10 000.895 366 998 627 445	−100 000 000.946 230 525 005 199	
	…	…	
	−100 000.915 205 593 725 096	−1 000 000 000.952 109 703 719 312	

The upward recurrence has the form $c = a + b$, where $c > a$, and it suffers bit loss if b/a lies in $[-2, -\frac{1}{2}]$. Eliminate b to find $b/a = (c - a)/a$, and bound the loss interval by solving two equations:

$$(c - a)/a = -2, \qquad\qquad (c - a)/a = -\tfrac{1}{2},$$
$$c = -a, \qquad\qquad 2c = a.$$

The first is a feasible solution, but the second is not because of the constraint $c > a$, and further numerical experiments show that the first provides upper bounds. We obtain a few bit-loss intervals by numerical solution of the first, and reformat the output into three columns:

```
% maple
> for n from 1 to 15 do
>   printf("%2d  [0, %.4f]\n",
>          n, fsolve(Psi(x + n) = -Psi(x), x = 0 .. 2))
> end do:
 1  [0, 1.0703]          6  [0, 0.5354]         11  [0, 0.4246]
 2  [0, 0.8474]          7  [0, 0.5033]         12  [0, 0.4118]
 3  [0, 0.7171]          8  [0, 0.4778]         13  [0, 0.4005]
 4  [0, 0.6342]          9  [0, 0.4570]         14  [0, 0.3906]
 5  [0, 0.5772]         10  [0, 0.4395]         15  [0, 0.3817]
```

Any convenient unit interval can be chosen for approximating $\psi(x)$ on the positive axis, as long as it excludes the interval $[0, 1.071]$. The interval $[2, 3]$ is therefore a reasonable start for upward recurrence.

A similar analysis for downward recurrence produces these bit-loss intervals, after reformatting the output list:

```
> for n from 1 to 30 do
>     printf("%2d  [%7.4f, %7.4f]\n",
>            n, fsolve(  Psi(x - n) = -Psi(x), x = 0 .. 25),
>               fsolve(2*Psi(x - n) =  Psi(x), x = 0 .. 25))
> end do:
 1  [ 2.0703,  3.0891]  11  [10.2860, 15.3422]  21  [12.1878, 12.4047]
 2  [ 2.8474,  4.4763]  12  [12.4118, 16.4885]  22  [12.1847, 12.3950]
 3  [ 3.7171,  5.7823]  13  [12.2743, 12.6282]  23  [ 6.1910, 12.3865]
 4  [ 4.6342,  2.4617]  14  [12.2429, 12.5572]  24  [ 6.1892, 12.3788]
 5  [ 5.5772,  8.2745]  15  [12.2261, 12.5147]  25  [11.1777, 12.3719]
 6  [ 6.5354,  9.4844]  16  [12.2151, 12.4847]  26  [ 6.1860, 12.3656]
```

```
  7  [ 6.3229, 10.6780]  17  [12.2071, 12.4619]  27  [11.1741, 12.3598]
  8  [ 0.4921, 11.8585]  18  [12.2008, 12.4437]  28  [11.1724, 12.3545]
  9  [ 3.2653, 13.0283]  19  [12.1957, 12.4285]  29  [10.1721, 12.3496]
 10  [ 6.2471, 14.1891]  20  [12.1915, 12.4158]  30  [11.1695, 12.3450]
```

Some of those descend into negative arguments, $x - n$, but we delay that case until the next section. The cases $n = 1$ and $n = 2$ show that the psi function on the interval $[1, 2]$ cannot be reliably determined from values on $[2, 3]$, and that we can safely recur downward to the interval $[0, 1]$ from $[1, 2]$, but not from $[2, 3]$. We return to that problem later in **Section 18.2.7** on page 543.

The numerical investigations of subtraction loss suggest that accurate computation of the psi function by recurrence relations must be based on separate treatment of the intervals $[1, 2]$ and $[2, 3]$. The first brackets the root at $x = z \approx 1.461$, so further special handling is necessary near that root to ensure high accuracy.

18.2.3 Psi functions with negative arguments

As with the gamma function, there is a relation between psi functions of positive and negative arguments:

$$\psi(-x) = \psi(x+1) + \pi/\tan(\pi x), \qquad \textit{reflection formula.}$$

Here again, instead of the gamma function's product formula, we have a sum, and numerical difficulties are readily evident. There is exactly one root of $\psi(x)$ between every pair of adjacent negative integers. Thus, if $\psi(-x)$ is near a root on the negative axis, its value is obtained from the sum of two right-hand side terms, at least one of which is never small for $x > 2$, so there must have been catastrophic cancellation.

Even in the absence of subtraction loss, the trigonometric term requires careful handling to avoid serious accuracy loss. We discuss it further in **Section 18.2.9** on page 545.

The reflection formula therefore does not provide a satisfactory way to compute `psi(x)` on the negative axis for arguments near a root. Most previous software implementations of the psi function ignore that serious problem, and as a result, can be wildly inaccurate for negative arguments. Unlike the log-gamma function, the roots do not hug the poles (compare **Table 18.2** on page 534 and **Table 18.3** on the facing page), so the root regions remain a computational problem over the entire negative axis.

18.2.4 Psi functions with argument multiples

Like the gamma function, psi functions of argument multiples are related, but with sums instead of products:

$$\begin{aligned}
\psi(2x) &= \log(2) + (\psi(x) + \psi(x + \tfrac{1}{2}))/2, \\
\psi(nx) &= \log(n) + \frac{1}{n}\sum_{k=0}^{n-1} \psi(x + k/n) && \textit{for } n = 2, 3, 4, \ldots, \\
\psi(n) &= -\gamma + \sum_{k=1}^{n-1} \frac{1}{k}, \\
\gamma &= 0.577\,215\,664\,901\,532\,860\,\ldots && \textit{Euler–Mascheroni constant.}
\end{aligned}$$

Those relations can be useful identities for software testing, but we do not exploit them for computing `psi(x)`.

The Euler–Mascheroni constant, γ, turns up repeatedly in series expansions of the gamma, log-gamma, and psi functions. We represent that constant in our software as a sum of an exact high part and an approximate low part, $\gamma = \gamma_{\rm hi} + \gamma_{\rm lo}$, effectively defining it to twice working precision.

The psi function for integer arguments, $\psi(n)$, turns up later in the chapter on Bessel functions (see **Section 21.3** on page 703). The formula for $\psi(n)$ involves a partial sum of the first $n - 1$ terms of the *harmonic series*. The psi function provides one of the best ways to sum the harmonic series for large n.

The medieval French theologian/scientist Nicole Oresme observed about 1350 that the harmonic-series sum can be grouped and bounded like this [Sti02, page 172] [Dun91, Chapter 8]:

$$\begin{aligned}
s &= 1 + (\tfrac{1}{2}) + (\tfrac{1}{3} + \tfrac{1}{4}) + (\tfrac{1}{5} + \tfrac{1}{6} + \tfrac{1}{7} + \tfrac{1}{8}) + \cdots \\
&> 1 + (\tfrac{1}{2}) + (\tfrac{1}{4} + \tfrac{1}{4}) + (\tfrac{1}{8} + \tfrac{1}{8} + \tfrac{1}{8} + \tfrac{1}{8}) + \cdots \\
&> 1 + (\tfrac{1}{2}) + (\tfrac{1}{2}) + (\tfrac{1}{2}) + \cdots.
\end{aligned}$$

Because there is an infinite number of terms, the sum of halves must diverge, showing that the psi function grows without bound as its argument increases. However, as we noted earlier, the limited floating-point range of arguments sharply restricts the function growth. The divergence of the harmonic series is extremely slow: after a googol (10^{100}) terms, its sum still lies below 231.

18.2.5 Taylor-series expansions of psi functions

The psi function has series expansions that we can use to compute function values near some important small arguments:

$$
\begin{aligned}
\psi(x) &= -(1/x)(1 + \gamma x - \tfrac{\pi^2}{6}x^2 + \zeta(3)x^3 \cdots) && \textit{Taylor series,} \\
\psi(1+x) &= -\gamma + \tfrac{\pi^2}{6}x - \zeta(3)x^2 \cdots, \\
\psi(z+x) &= \sum_{k=1}^{\infty} (1/k!)\psi^{(k)}(z)x^k \\
&\approx 0.967\,672\,x - 0.442\,763\,x^2 + 0.258\,500\,x^3 - \cdots, \\
\psi(2+x) &= (1-\gamma) + (\tfrac{\pi^2}{6} - 1)x + (1 - \zeta(3))x^2 + \cdots, \\
\psi(3+x) &= (\tfrac{3}{2} - \gamma) + (\tfrac{\pi^2}{6}x - \tfrac{5}{4})x + (\tfrac{9}{8} - \zeta(3))x^2 + \cdots, \\
\zeta(3) &\approx 1.202\,056\,903\,159\,594\,285\,\ldots && \textit{Riemann zeta function value.}
\end{aligned}
$$

The expansion coefficients involve either powers of π, or zeta functions of integer arguments, or for the expansion near z, polygamma functions at z. They can all be generated explicitly to any order by symbolic-algebra systems. The coefficients are best handled by precomputing them to high precision, and then storing them in a compile-time constant table. See **Section 11.2** on page 303 for a definition of, and brief comment on, the Riemann zeta function, $\zeta(z)$, and **Section 18.7** on page 579 for details of its properties and computation.

Near the positive zero of the psi function, as long as the difference between the argument and the transcendental value z is computed accurately, the function result can never be zero. Here is a sample calculation in 32-bit decimal arithmetic:

```
% hocd32
hocd32> z = 1.461_632; psi(nextafter(z,0)); psi(z); psi(nextafter(z,2))
-1.107_954e-06
-1.402_819e-07
 8.273_900e-07
```

18.2.6 Asymptotic expansion of the psi function

The psi function has an asymptotic expansion that handles large arguments:

$$
\begin{aligned}
\psi(x) &\asymp \log x - 1/(2x) - \sum_{k=1}^{\infty} B_{2k}/(2kx^{2k}) && \textit{asymptotic expansion,} \\
&\asymp \log x - 1/(2x) - \frac{1}{12x^2} + \frac{1}{120x^4} - \frac{1}{252x^6} + \frac{1}{240x^8} - \frac{1}{132x^{10}} \\
&\quad + \frac{691}{32\,760x^{12}} - \frac{1}{12x^{14}} + \frac{3617}{8160x^{16}} - \frac{43\,867}{14\,364x^{18}} + \frac{174\,611}{6600x^{20}} - \cdots.
\end{aligned}
$$

The coefficients in the asymptotic expansion involve the *Bernoulli numbers*, B_{2k}, that we encountered in series expansions of the trigonometric and hyperbolic tangents and the gamma function; they are discussed further in **Section 18.5** on page 568. Coefficient growth in the asymptotic expansion first sets in at the term with x^{18} in the denominator. Convergence is rapid enough that if we sum the series to machine precision, we can achieve full accuracy in IEEE 754 binary arithmetic for x larger than 3 (32-bit), 6 (64-bit), 8 (80-bit), 13 (128-bit), and 26 (256-bit). The corresponding term counts are 7, 15, 14, 29, and 72. The psi function is one of the few in the mathcw library where an asymptotic expansion is actually useful for computation.

As with the asymptotic series for gamma and log-gamma, that for the psi function can be represented by a polynomial approximation in powers of $1/x^2$. Our software does not do so, because it needs to handle a wide variety of bases and precisions, and it uses that series only when convergence is rapid.

At least one existing psi-function implementation uses the asymptotic expansion as the starting point, followed by downward recurrence to obtain $\psi(x)$. However, accuracy with that approach is unacceptably poor, especially near the poles and zeros of the psi function.

18.2.7 Psi function on $[0, 1]$

We noted near the end of Section 18.2.2 on page 539 that downward recurrence to $[0, 1]$ is stable from $[1, 2]$, but not from $[2, 3]$.

Near the pole at $x = 0$, we should use the Taylor series to determine the function accurately where it changes most rapidly, and the question is then whether we could use it over the interval $[0, 1]$.

Unfortunately, numerical evaluation of the Taylor-series expansions of $\psi(x)$ and $\psi(1 + x)$ shows that the coefficients quickly approach values near ± 1, which means that the series convergence depends entirely on decreasing values of the powers x^k. If $x \approx 1$, there is no useful convergence, and there is also massive cancellation in the sums because of the alternating signs. Even if $x \approx \frac{1}{2}$, there is still cancellation, and about as many terms are needed as there are bits of precision.

One solution would be to split the interval $[0, 1]$ into multiple regions, each with its own expansion. That could lead to acceptably fast convergence, but needs substantial table storage. Alternatively, one could use rational polynomial or Chebyshev fits of $x\psi(x)$ or some other auxiliary function, but those fits are poor unless the function is nearly linear.

Fortunately, in 1981 Peter McCullagh discovered a new series expansion [McC81] of the psi function, and that series is eminently suitable for computation. It takes the form

$$\psi(1 + z) = -\gamma - \sum_{k=1}^{\infty} (-z)^k \big(c_k + a_k/(z + k)\big), \qquad \textit{valid for all finite complex } z.$$

The two coefficient arrays are defined by

$$a_k = k^{-k}, \qquad c_k = \sum_{n=k+1}^{\infty} n^{-(k+1)}.$$

and their values are all positive. Accurate evaluation of the coefficients requires high-precision arithmetic, so they are best computed in a symbolic-algebra system, and then stored in psi-function software as compile-time constant tables. The c_k values, and all but the first two a_k values, fall by at least *ten* for successive indexes k. Each term in the sum requires a division, but we only use the series for one unit interval, and convergence is rapid when $|z| \leq 1$.

As usual with such sums where the number of terms is not known in advance, we leave the first term to last, and add the terms for $k = 2, 3, 4, \ldots$ until their floating-point sum no longer changes, at which point we terminate the loop. In the first term, we have $a_1 = 1$ and $c_1 \approx 0.644\,934$, so we rearrange it as

$$\begin{aligned}
\big(c_1 + 1/(x + 1)\big)x &= \big(c_1 + (x + 1 - x)/(x + 1)\big)x \\
&= \big((c_1 + 1) - x/(x + 1)\big)x \\
&= \big(((c_1)_{\mathrm{hi}} + 1) + ((c_1)_{\mathrm{lo}} - x/(x + 1))\big)x,
\end{aligned}$$

We then add, in order, $-\gamma_{\mathrm{lo}}$, the term with $(c_1)_{\mathrm{lo}}$, the term with $(c_1)_{\mathrm{hi}}$, and finally, $-\gamma_{\mathrm{hi}}$, to obtain an accurate function result.

18.2.8 Outline of the algorithm for `psi()`

The complete algorithm in the file `psix.h` for computing $\psi(x)$ considers several argument regions, from left to right on the real axis, after handling a NaN:

***x* is a NaN** : Set a domain error, and set the result to that NaN.

$-\infty \le x \le$ **−XMAX** : The argument magnitude is so large that it has no fractional part, so we are on one of the asymptotes for which $\psi(x) = \pm\infty$. The sign is not determinable, so we could set the result to `QNAN("")`. However, we follow our practice in `tgamma(x)`, and arbitrarily set the result to $+\infty$, to distinguish it from the $-\infty$ that is approached as $x \to +0$. We also set a range error. A suitable value of `XMAX` is β^t for base β and t-digit precision.

***x* is a negative whole number** : Set the result to $+\infty$ and set a range error.

−XMAX $< x \le$ **XCUT_REFLECT** : Use the reflection formula and restart the computation, now for positive x. We consider the trigonometric part in **Section 18.2.9** on the facing page.

−XCUT_REFLECT $< x \le$ **−XSMALL** : Use downward recurrence from the interval $[1, 2]$. `XSMALL` is set to $\sqrt{3(\frac{1}{2}\epsilon/\beta)}/\pi$, a value such that $\pi/\tan(\pi x)$ can be computed with correct rounding from the first two terms of its Taylor series (see **Section 2.6** on page 10).

−XSMALL $< x \le$ **−XCUT_0_1** : Use the reflection formula, but replace the term $\pi/\tan(\pi x)$ by its two-term Taylor series $1/x - \frac{1}{3}\pi^2 x$, and use the McCullagh series implemented in the private function `psi_mccullagh(x)` for computing $\psi(1 + x)$ without evaluating $1 + x$, because that would lose trailing digits.

−XCUT_0_1 $< x <$ **+XCUT_0_1** : Use the Taylor series about the origin (see **Section 18.2.5** on page 542), with a check that $1/x$ does not overflow. If it would overflow, avoid the division, set the result to −`copysign(∞,x)`, and set a range error.

XCUT_0_1 $\le x < \frac{3}{4}$: Use downward recurrence from $[1, 2]$ to compute $\psi(x) = \psi(x + 1) - 1/x$. Use the McCullagh series for $\psi(1 + x)$.

$\frac{3}{4} \le x < 1$: Sum the McCullagh series for the exact argument $1 - x$.

$x = 1$: The result is $-\gamma$, correctly rounded.

$1 < x < 2$: If x is near z, sum a Taylor-series expansion about z with orders 2, 4, 8, 16, or 32, depending on the size of $|x - z|$, and represent the low-order coefficient as a two-part sum.

If $x < \frac{5}{4}$, use the McCullagh series for the exact argument $x - 1$.

If $\frac{3}{2} < x$, compute a minimax rational polynomial approximation, $\mathcal{R}_1(t)$, for t in $[0, 1]$, such that

$$\psi(\tfrac{3}{2} + \tfrac{1}{2}t) = (1 - t)\psi(\tfrac{3}{2}) + t\psi(2) + t(1 - t)\mathcal{R}_1(t).$$

That is linear interpolation with a positive polynomial correction that contributes less than 10% of the result, and that is zero at both endpoints. Represent the two constants $\psi(\frac{3}{2})$ and $\psi(2)$ as two-part sums, and sum the terms with error recovery.

Otherwise, use a polynomial approximation $\psi(x) \approx (x - z)\mathcal{R}_2(x)$. Compute the factor $(x - z)$ from $(x - z_{\mathrm{hi}}) - z_{\mathrm{lo}}$, where z_{hi} is exactly representable. That factor encourages correct behavior, including monotonicity, for $x \approx z$.

$x = 2$: The result is $1 - \gamma$, correctly rounded.

$2 < x < 3$: Compute a minimax rational polynomial approximation, $\mathcal{R}_3(t)$, for t in $[0, 1]$, such that

$$\psi(2 + t) = (1 - t)\psi(2) + t\psi(3) + t(1 - t)\mathcal{R}_3(t).$$

That is linear interpolation with a positive polynomial correction that contributes less than 5% of the result, and that is zero at both endpoints. Represent the two constants $\psi(2)$ and $\psi(3)$ as two-part sums, and sum the terms with error recovery.

$x = 3$: The result is $\frac{3}{2} - \gamma$, correctly rounded.

$3 < x <$ XCUT_ASYMPTOTIC : Let $n = \mathrm{floor}(x-1)$. Then use upward recurrence to compute $\psi(x) = \psi(x-n) + \sum_{k=0}^{n-1}(1/(x-n+k))$, where $x-n$ is in the region $[2,3]$ of the rational approximation. Use error recovery to ensure an accurate sum of reciprocals.

The upper limit is small enough that only a few steps are needed: 6 in the smaller formats, about 15 in most `double` formats, and up to 60 in the largest formats.

XCUT_ASYMPTOTIC $\leq x <$ XLARGE : Sum `N_ASYMPTOTIC` terms of the asymptotic series.

The constant `XCUT_ASYMPTOTIC` is the precision-dependent limit above which we can use the asymptotic expansion. Typical values of that limit are given in **Section 18.2.6** on page 542.

XLARGE $\leq x < +\infty$: Set the result to $\log(x)$, the first term of the asymptotic series.

The limit `XLARGE` is chosen to be the precision-dependent argument at which the asymptotic expansion reduces to its first term.

Otherwise, $x = +\infty$: Set the result to x, and set a range error.

Here are sample values of the decision constants for the IEEE 754 64-bit binary format, rounded to three digits:

`N_ASYMPTOTIC = 6`	*number of terms in asymptotic series*
`XCUT_0_1 = 9.62e-17`	*cutoff for 2-term Taylor series about 0*
`XCUT_ASYMPTOTIC = 15.0`	*cutoff for switch to asymptotic series*
`XCUT_MIN_32 = 0.465`	*cutoff for 32-term Taylor series about z*
`XCUT_REFLECT = -10.0`	*use reflection formula below this value*
`XCUT_TAN_2 = 4.11e-09`	*$\pi/\tan(\pi x)$ simplifies below this value*
`XLARGE = 2.71e+14`	*$\psi(x) = \log(x)$ above this value*
`XMAX = 9.01e+15`	*x is a whole number above this value*

The troublesome areas in that algorithm are the cases of negative arguments, and arguments near the poles and zeros of $\psi(x)$. **Figure 18.7** on the following page shows the measured errors in our implementations of the `psi(x)` function.

18.2.9 Computing $\pi/\tan(\pi x)$

When we use argument reflection to move the computation of $\psi(x)$ from negative x to positive x, we require the term $\pi/\tan(\pi x)$. The error-magnification factor for $\tan(x)$ in **Table 4.1** on page 62 is proportional to $x/\tan(x)$ when $\tan(x)$ is small, and to $x\tan(x)$ when $\tan(x)$ is large. Because the range of $\tan(x)$ is $(-\infty, \infty)$ for any interval of width π, even tiny errors in computing the argument πx can make a large difference in the function value. Direct computation with `tan(PI * x)` is therefore inadvisable.

As we saw in **Section 18.1.3** on page 531, there are three reasonable solutions:

- Borrow code from the file `tanpix.h` (see **Section 11.7** on page 315 and **Section 11.7.2** on page 318).
- Call `tanpi(x)` directly when it is available.
- Compute $\tan(\pi x)/\pi$ from a Taylor-series expansion when x is small, and otherwise from a polynomial fit for x on the interval $[0, \frac{1}{4}]$. Use these symmetry relations for the other needed intervals:

$$\begin{aligned}
1/\tan(\pi(\tfrac{1}{2} - x)) &= \tan(\pi x), && \textit{for } x \textit{ on } [\tfrac{1}{4}, \tfrac{1}{2}],\\
1/\tan(\pi(x - \tfrac{1}{2})) &= -\tan(\pi x), && \textit{for } x \textit{ on } [\tfrac{1}{2}, \tfrac{3}{4}],\\
\tan(\pi(1 - x)) &= -\tan(\pi x), && \textit{for } x \textit{ on } [\tfrac{3}{4}, 1],\\
\tan(\pi(n + r)) &= \tan(\pi r), && \textit{for } x \textit{ outside } [0, 1].
\end{aligned}$$

In each case, the argument reduction is exact, and it is possible to compute an error correction to improve the accuracy of the final inversion to recover $\pi/\tan(\pi x)$.

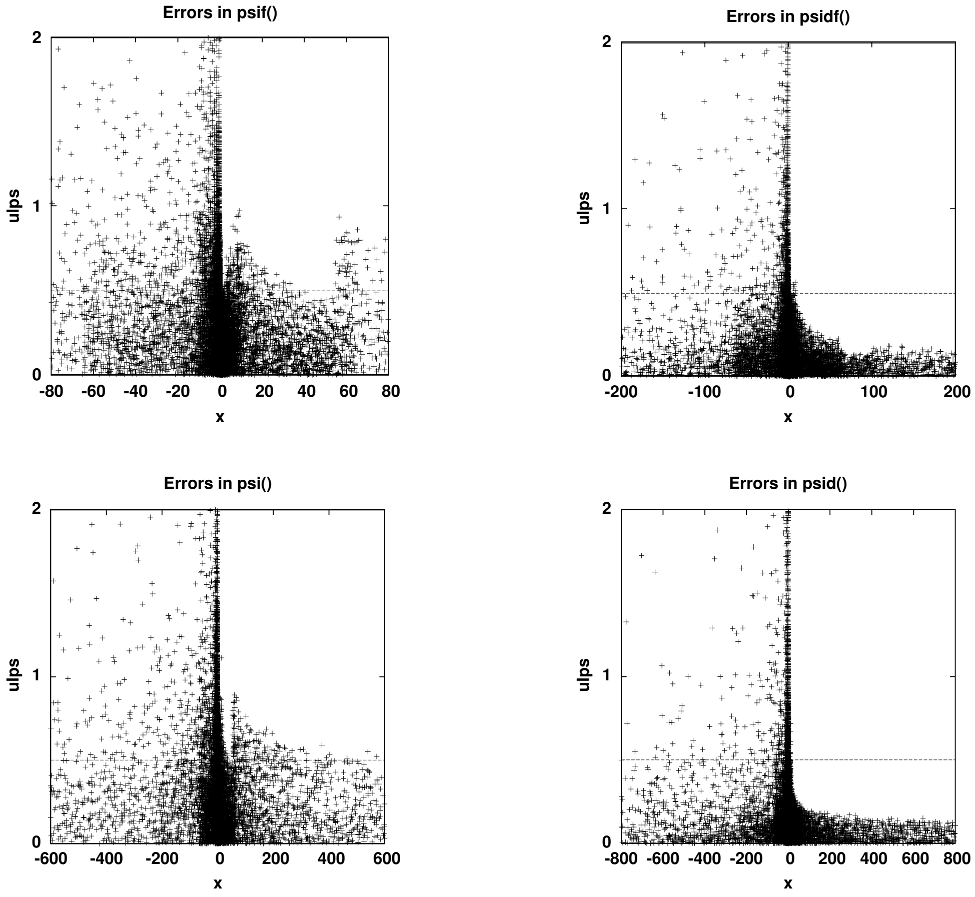


Figure 18.7: Errors in the `PSI()` functions.

18.2.10 Outline of the algorithm for `psiln()`

The algorithm for computing `psiln(x)` $= \psi(x) - \log(x)$ is a simplification of that for $\psi(x)$, with these argument ranges, and the same limits defined by uppercase macro names:

x is a NaN : Set a domain error, and set the result to that NaN.

$x < 0$: Set a domain error, and set the result to `QNAN("")`.

$x = 0$: Set a range error, and set the result to $-\infty$.

$0 < x <$ `X1` : Choose `X1` to be the point at which bit loss begins for $\psi(x) - \log(x)$. The value of `X1` is the solution of $\psi(x)/\log(x) = \frac{1}{2}$, for which Maple finds this accurate value:

```
Digits := 100:
X1 := fsolve(Psi(x)/log(x) = 1/2, x = 0 ..  10);
```

$1.819\,537\,948\,238\,785\,644\,418\,866\,760\,843\,345\,720\,043\,932\,706\ldots$

Compute `PSILN(x)` from `PSI(x) - LOG(x)`.

`X1` $\le x <$ `CUTOFF` : The computation can use the recurrences for $\psi(x \pm n)$, but numerical experiments recommend

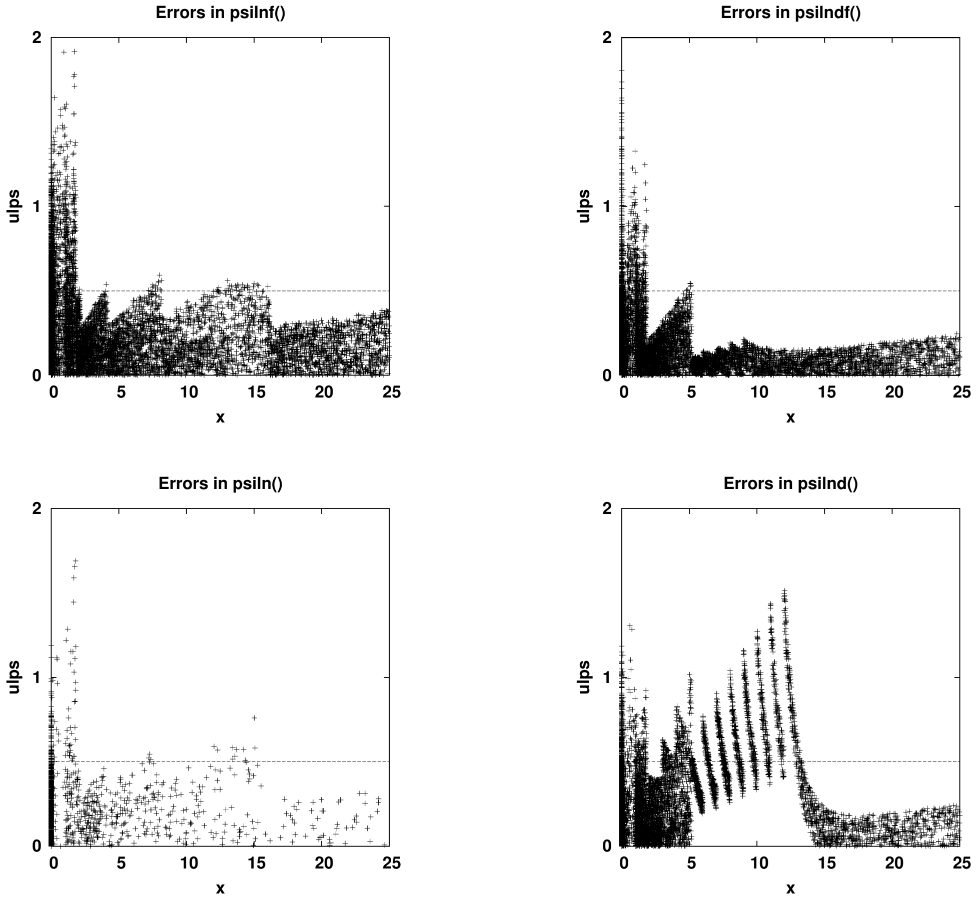


Figure 18.8: Errors in the `PSILN()` functions.

the downward direction. Choose $n = \mathrm{ceil}(\texttt{CUTOFF} - x)$, and compute the result from

$$\texttt{psiln(x)} = \texttt{psiln(x + n)} - \log(x/(x+n)) - \sum_{k=1}^{n}(1/(x+n-k)).$$

Compute the logarithm term as $\mathrm{log1p}(-n/(x+n))$ when $x/(x+n)$ lies in $[1-1/\beta, 1+1/\beta]$, because that function is then more accurate.

`CUTOFF` $\le x <$ **`XLARGE`** : Sum the asymptotic expansion, omitting the initial $\log(x)$ term.

`XLARGE` $\le x \le +\infty$: Set the result to zero.

Figure 18.8 show the measured errors in our implementation of the function `psiln(x)`.

18.3 Polygamma functions

MATHEMATICAL FACT MAY BE COMPUTATIONAL FICTION.

— ANONYMOUS

The polygamma functions are higher derivatives of the psi function, and are usually defined with this notation:

$$\begin{aligned}
\psi^{(0)}(z) &= \psi(z), && \textit{normal psi function,} \\
\psi^{(n)}(z) &= \frac{d^n \psi(z)}{dz^n} = \frac{d^{n+1}\Gamma(z)}{dz^{n+1}} \frac{1}{\Gamma(z)}, && \text{for } n = 1, 2, 3, \ldots .
\end{aligned}$$

The parenthesized superscript n is called the *order* of the function.

The polygamma functions can also be defined as integrals:

$$\psi^{(n)}(z) = (-1)^{n+1} \int_0^\infty \frac{t^n \exp(-zt)}{1 - \exp(-t)}\, dt, \qquad \text{for } n = 1, 2, 3, \ldots \text{ and } \operatorname{real}(z) > 0.$$

We discuss the suitability of that formula for numerical computation later in **Section 18.3.3** on page 558.

The Maple symbolic-algebra system provides those functions as `Psi(z)` and `Psi(n,z)`, with `Psi(0,z) = Psi(z)`. Mathematica supplies them with the names `PolyGamma[z]` and `PolyGamma[n,z]`. MATLAB has only the numerical function `psi(x)`, and restricted to nonnegative arguments. Maxima has the numerical (big-float) functions `bfpsi0 (z,d)` and `bfpsi(n,z,d)`, where d is the precision in decimal digits, as well as the symbolic functions `psi(x)` and `psi[n](x)`. MuPAD calls them `psi(z)` and `psi(z,n)`. PARI/GP has `psi(z)`, but no polygamma function. REDUCE provides `psi(z)`, `psi(n,z)`, and `polygamma(n,z)`, where the last two are equivalent.

As the argument z suggests, those functions can be defined for complex arguments, but we do not provide that extension in the `mathcw` library. Although some of those languages use capitalized function names, modern mathematical notation uses lowercase $\psi^{(n)}(z)$, rather than uppercase $\Psi^{(n)}(z)$.

Because $\psi(x)$ approaches $\log(x)$ for large x, and the logarithm is a slowly increasing function, higher derivatives are small, and the integral form shows that the integrand decreases exponentially. We therefore have this limiting behavior:

$$\lim_{x \to +\infty} \psi^{(n)}(x) \to (-1)^{n+1} \times 0, \qquad \text{for } n = 1, 2, 3, \ldots .$$

Like the psi function, the polygamma functions have poles at arguments that are zero or negative integers:

$$\psi^{(n)}(-m) = \pm\infty, \qquad \text{for } m = 0, 1, 2, \ldots .$$

Elsewhere, the polygamma functions are single-valued, and defined over the entire complex plane.

Figure 18.9 on the next page shows the first four polygamma functions of real arguments. Functions of odd order have similar appearance, but the minima on the negative axis increase with n. Functions of even order resemble each other, and their steepness grows with n.

There is a reflection formula that allows us to find the values of polygamma functions of negative arguments from function values at positive arguments:

$$\psi^{(n)}(1 - z) = (-1)^n \left(\psi^{(n)}(z) + \pi \frac{d^n \cot(\pi z)}{dz^n} \right).$$

The reflection rule can also be applied for z in $[0, \frac{1}{2}]$ to move the computation to $[\frac{1}{2}, 1]$, away from the pole at the origin.

The higher-order derivatives of the cotangent are an unpleasant complication for a software implementation of the reflection rule. On initial expansion by a symbolic-algebra system, the derivatives are horribly messy, but they can be simplified with the introduction of two abbreviations for common subexpressions. Here are the first ten derivatives to show how they behave:

$$\begin{aligned}
c &= \cot(\pi z), && \textit{convenient shorthand,} \\
d &= 1 + c^2, && \textit{another shorthand,} \\
\frac{d \cot(\pi z)}{dz} &= (d)(-\pi), \\
\frac{d^2 \cot(\pi z)}{dz^2} &= (2cd)(-\pi)^2,
\end{aligned}$$

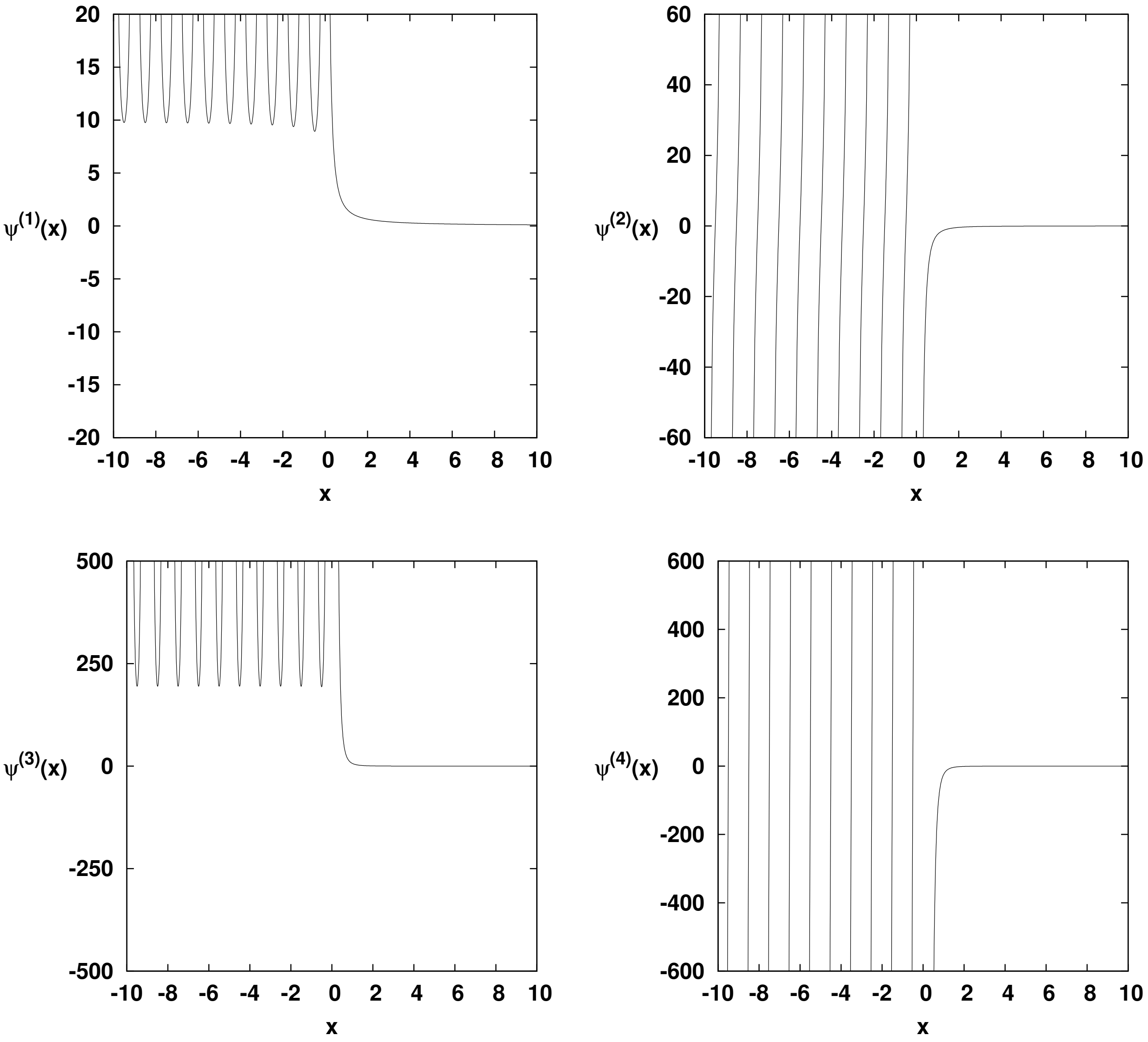


Figure 18.9: Polygamma functions of order 1 through 4. Functions of even order have zeros between negative integers, whereas functions of odd order have no real zeros. On the positive axis, all tend to zero for large n or large x. There are poles at the origin and at negative integer arguments.

$$
\begin{aligned}
\frac{d^3\cot(\pi z)}{dz^3} &= (4c^2d + 2d^2)(-\pi)^3, \\
\frac{d^4\cot(\pi z)}{dz^4} &= (8c^3d + 16cd^2)(-\pi)^4, \\
\frac{d^5\cot(\pi z)}{dz^5} &= (16c^4d + 88c^2d^2 + 16d^3)(-\pi)^5, \\
\frac{d^6\cot(\pi z)}{dz^6} &= (32c^5d + 416c^3d^2 + 272cd^3)(-\pi)^6, \\
\frac{d^7\cot(\pi z)}{dz^7} &= (64c^6d + 1824c^4d^2 + 2880c^2d^3 + 272d^4)(-\pi)^7,
\end{aligned}
$$

$$\begin{aligned}
\frac{d^8\cot(\pi z)}{dz^8} &= (128c^7d + 7680c^5d^2 + 24576c^3d^3 + 7936cd^4)(-\pi)^8, \\
\frac{d^9\cot(\pi z)}{dz^9} &= (256c^8d + 31\,616c^6d^2 + 185\,856c^4d^3 + 137\,216c^2d^4 + \\
&\qquad 7936d^5)(-\pi)^9, \\
\frac{d^{10}\cot(\pi z)}{dz^{10}} &= (512c^9d + 128\,512c^7d^2 + 1\,304\,832c^5d^3 + 1\,841\,152c^3d^4 + \\
&\qquad 353\,792cd^5)(-\pi)^{10}.
\end{aligned}$$

Evidently, the derivatives have the general form

$$\frac{d^n\cot(\pi z)}{dz^n} = (-\pi)^n \sum_{k=1}^{\lceil n/2\rceil} a_k c^{n-2k+1} d^k,$$

We recognize that the leading coefficient is $a_1 = 2^{n-1}$, and that relation suffices to start the upward progression. The other coefficients form sequences that are found in Sloane's on-line encyclopedia,[2] but do not appear to have simple closed formulas.

Even-order derivatives are positive, and odd-order ones are negative. The terms within each parenthesized sum have the same signs, so in the absence of overflow, the computation of the derivatives is perfectly stable. The coefficients are exact integers, but already at $n = 13$, some are too big to represent as 32-bit signed integers, and at $n = 21$, coefficients overflow the 64-bit signed integer format. Because integer overflow wraps positive values to negative, it would be disastrous to use the overflowed coefficients. Instead, we represent them in the widest available floating-point type, even though that may have lower precision than the largest integer type.

Differentiating the general formula for the n-th derivative of $\cot(\pi x)$ produces a new set of coefficients, b_k, corresponding to an increase of n by 1, and once generated, they replace the old a_k values. We can therefore produce the coefficients for a given n with code that looks like this:

```
hp_t a[(NMAX + 1) / 2], b[(NMAX + 1) / 2];
int k, kmax, m, n;

n = /* to be supplied */;
a[1] = HP(1.);

for (m = 1; m < n; ++m)
{
    kmax = (m + 1) / 2;

    for (k = 1; k <= kmax; ++k)
        b[k] = (hp_t)(2 * k) * a[k];

    kmax++;
    b[kmax] = HP(0.);

    for (k = 2; k <= kmax; ++k)
        b[k] += (hp_t)((m + 1) - 2 * k + 2) * a[k-1];

    for (k = 1; k <= kmax; ++k)
        a[k] = b[k];
}
```

The coefficients must be stored in arrays, so we are forced to set a limit, NMAX, on the largest n that we can support in the region of negative arguments, because the mathcw library design guidelines forbid use of dynamic memory allocation. Numerical tests show that, starting from random values of x, the derivatives soon grow large. A small Maple function reports where the overflow limits are not yet reached, and some experimentation produces this session log:

[2] See http://oeis.org/.

```
> Digits := 40:

> test := proc(nmax, ktest, dmax)
>             local d, x, z:
>             x := evalf(rand() / 10**12):
>             d := evalf(subs(z = x, diff(cot(Pi*z), z$nmax))):
>             if (evalb(abs(d) < dmax)) then
>                 printf("x = %10.7f\t%5d\t%5d\t% 15.7e\n", x, ktest, nmax, d)
>             end if
>         end proc:

> # 32-bit IEEE 754 binary overflow limit
> for k from 1 to 100 do test(28, k, 3.402823e38) end:
x =  0.4906575     38      28     5.9577625e+37
...
x =  0.5067042     85      28    -4.1659363e+37

> for k from 1 to 1000 do test(29, k, 3.402823e38) end:
[no output]

> # 64-bit IEEE 754 binary overflow limit
> for k from 1 to 100 do test(150, k, 1.797693e+308) end:
x =  0.4971346    150   1.0173715e+308
x =  0.4993021    150   2.2047452e+307
x =  0.4996253    150   1.1775761e+307

> for k from 1 to 1000 do test(160, k, 1.797693e+308) end:
[no output]

> # 80-bit and 128-bit IEEE 754 binary overflow limit
> for k from 1 to 100 do test(1600, k, 1.189731e+4932) end:
x =  0.4972015     36    1600    1.1922704e+4919
x =  0.5041801     93    1600   -1.0259594e+4921
...

> for k from 1 to 100 do test(1700, k, 1.189731e+4932) end:
[no output]
```

Evidently, values of NMAX $= 30, 160$, and 1700 are likely to suffice for the three main IEEE 754 binary formats. Somewhat larger values are needed for the wider range of the decimal formats, and NMAX $= 2000$ is enough for the 128-bit size. However, because some of the coefficients are inexact for $n > 20$, we conclude that accuracy of the derivatives must deteriorate beyond that limit. Computed values of the polygamma functions on the negative axis for large n are therefore unreliable, and numerical tests suggest that $n = 40$ is a more realistic maximum practical order in that region.

For positive integer arguments, the polygamma functions are intimately related to the Riemann zeta numbers, $\zeta(n)$, that we treat later in **Section 18.7** on page 579:

$$\psi^{(n)}(1) = (-1)^{n+1} n!\, \zeta(n+1), \qquad \text{for } n = 1, 2, 3, \ldots,$$

$$\psi^{(n)}(m) = (-1)^{n} n! \left(-\zeta(n+1) + 1 + \frac{1}{2^{n+1}} + \frac{1}{3^{n+1}} + \cdots + \frac{1}{(m-1)^{n+1}} \right).$$

We describe later a function family, ZETNUM(n), that computes the zeta numbers, $\zeta(n)$. For $n > 1$, their values are roughly $1.645, 1.202, 1.082, 1.037, \ldots$, and they tend smoothly to 1 from above. There is clearly subtraction loss in the first two terms of the expansion for $\psi^{(n)}(m)$. A second function family, ZETNM1(n), computes the difference $\zeta(n) - 1$ accurately, but the sum of the first two terms remains negative. Numerical tests show that there is almost always severe subtraction loss when $m > 1$, making use of the relation impractical in fixed-precision arithmetic.

Half-integral arguments of polygamma functions also have special forms:

$$\psi^{(n)}(\tfrac{1}{2}) = (-1)^{n+1} n!\,(2^{n+1}-1)\zeta(n+1), \qquad \text{for } n = 1,2,3,\ldots,$$
$$\psi^{(1)}(m+\tfrac{1}{2}) = \tfrac{1}{2}\pi^2 - 4\sum_{k=1}^{m}(2k-1)^{-2}, \qquad \text{for } m = 1,2,3,\ldots.$$

Numerical tests show that the subtraction in the second relation loses one decimal digit for small m, and about three digits for $m = 100$.

For fixed order, the polygamma functions satisfy these recurrence relations:

$$\psi^{(n)}(z+1) = \psi^{(n)}(z) + (-1)^n n!\, z^{-n-1},$$
$$\psi^{(n)}(z+m) = \psi^{(n)}(z) + (-1)^n n! \sum_{k=0}^{m-1}(z+k)^{-(n+1)}, \qquad \text{for } m = 1,2,3,\ldots,$$
$$\psi^{(n)}(z) = \psi^{(n)}(z-m) + (-1)^n n! \sum_{k=1}^{m}(z-k)^{-(n+1)}, \qquad \text{for } m = 1,2,3,\ldots.$$

For fixed n and positive x, the polygamma functions have no zeros, and have the sign $(-1)^{n+1}$, so they are negative for even n, and positive for odd n. The right-hand side always has a subtraction that may cause catastrophic loss of leading digits if the recurrence is used in the upward direction. However, in the downward direction, the recurrence is stable. An early published algorithm [TS69] for that function family consequently used downward recurrence from the asymptotic region described later.

A numerical experiment in Maple easily finds the intervals in which the recurrence formula suffers subtraction loss (output has been reformatted into three columns):

```
> for n in {
>               seq(i, i = 1 .. 9, 1),
>               seq(i, i = 10 .. 90, 10),
>               seq(i, i = 100 .. 900, 100)
>          }
> do
>     printf("%3d  [0, %8.3f]\n", n, fsolve(Psi(n,x) / ((-1)**n * n! * x**(-n - 1)) = -2, x = 0 .. 2*n))
> end do:
1  [0,    1.390]      10  [0,   14.370]     100  [0,  144.212]
2  [0,    2.830]      20  [0,   28.797]     200  [0,  288.482]
3  [0,    4.272]      30  [0,   43.224]     300  [0,  432.751]
4  [0,    5.714]      40  [0,   57.651]     400  [0,  577.021]
5  [0,    7.157]      50  [0,   72.078]     500  [0,  721.290]
6  [0,    8.599]      60  [0,   86.504]     600  [0,  865.560]
7  [0,   10.042]      70  [0,  100.931]     700  [0, 1009.829]
8  [0,   11.485]      80  [0,  115.358]     800  [0, 1154.099]
9  [0,   12.927]      90  [0,  129.785]     900  [0, 1298.368]
```

We conclude that subtraction loss is likely if $x < 1.45n$. Software may need to tabulate the problem intervals so that alternate algorithms, or higher precision, can be used in the loss regions. To illustrate the severity of the problem, consider this example with a nine-digit loss:

$$\begin{aligned}
\psi^{(10)}(9.5-8) &= -42\,108.858\,768\,975\,491\,796\,771\ \ldots, \\
10! \sum_{k=1}^{8}(9.5-k)^{-(10+1)} &= +42\,108.858\,670\,481\,478\,569\,400\ \ldots, \\
\psi^{(10)}(9.5) &= \qquad -0.000\,098\,494\,013\,227\,370\ \ldots.
\end{aligned}$$

The psi and polygamma functions satisfy argument-multiplication formulas that can be useful for software testing:

$$\psi(mz) = \log(m) + m^{-1} \sum_{k=0}^{m-1} \psi(z + \frac{k}{m}), \qquad \text{for } m = 1, 2, 3, \ldots,$$

$$\psi^{(n)}(mz) = m^{-n-1} \sum_{k=0}^{m-1} \psi^{(n)}(z + \frac{k}{m}), \qquad \text{for } m, n = 1, 2, 3, \ldots.$$

The second formula is numerically stable on the positive axis, but the first may be subject to subtraction loss for $z < 1.461\ldots$, below the only location where $\psi(x) < 0$ for $x > 0$. Once again, a numerical experiment identifies the problem argument ranges, and output has been reformatted into three columns:

```
> for x from 0.05 to 1.50 by 0.05 do
>     m_min := infinity:
>     m_max := -m_min:
>     for m from 1 to 50 do
>         s := log(m):
>         t := sum(Psi(x + k/m), k = 0 .. m - 1):
>         r := evalf(s / t):
>         if (evalb( (-2 <= r) and (r <= -1/2)) ) then
>             m_min := min(m_min, m):
>             m_max := max(m_max, m):
>         end if
>     end do:
>     if (evalf(m_min <= m_max)) then
>         printf("%4.2f  %2d  %2d\n", x, m_min, m_max)
>     end if
> end do:
0.65   3   7           0.80   2  26           0.95   2  50
0.70   3  11           0.85   2  42           1.00   2   2
0.75   2  17           0.90   2  50           1.05   2   2
```

There are series expansions that can be computationally useful in those regions where the sums converge quickly, and where subtraction loss is not a problem:

$$\psi^{(n)}(1 + z) = (-1)^{n+1} \sum_{k=0}^{\infty} \frac{(-1)^k (n + k)!\, \zeta(n + k + 1) z^k}{k!}, \qquad \text{for } |z| < 1,$$

$$\psi^{(n)}(z) = (-1)^{n+1} n! \sum_{k=0}^{\infty} (z + k)^{-n-1}, \qquad \text{if } z \neq 0, -1, -2, \ldots.$$

The first sum converges quickly for tiny $|z|$, but the second is only practical when $n \gg |z|$.

The alternating signs in the first sum can cause subtraction loss. Numerical experiments in Maple suggest that the sum should be used only when $|z| < 1/(16(n + 1))$.

The second sum shows polygamma-function behavior near the origin:

$$\lim_{z \to 0} \psi^{(n)}(z) \to (-1)^{n+1} n!\, z^{-n-1}.$$

We can use that one-term formula computationally whenever $|z^{n+1}| < \frac{1}{2}\epsilon/\beta$. Because the power decreases as n grows, we apply that formula for *all* n whenever $|z| < \sqrt{\frac{1}{2}\epsilon/\beta}$. More generally, for arbitrary $n > 0$, and t base-β digits of precision, we can use the formula whenever $(n + 1) \times (\operatorname{logb}(|z|) + 1) < -(t + 1)$.

The analysis of the preceding paragraph is based on our usual practice of setting a loop termination condition with the assumption of rapidly decreasing terms. However, in this case that assumption is incorrect, and we can improve the computed result by adding a small correction. To find that correction, we examine the omitted terms of the sum like this:

```
% maple
> f := proc(n, x)
>           local k:
>           return sum((x + k)**(-n - 1), k = 1 .. infinity)
>       end proc:

> for n from 1 to 12 do
>      t1 := convert(series(f(n,x), x = 0, 2), polynom):
>      c0 := op(1,t1):
>      c1 := evalf(coeff(t1,x)):
>      printf("n = %3d  c0 = %19a  err = % 10.6f * x\n", n, c0, c1)
> end do:
n =   1  c0 =            1/6*Pi^2  err =  -2.404114 * x
n =   2  c0 =             Zeta(3)  err =  -3.246970 * x
n =   3  c0 =           1/90*Pi^4  err =  -4.147711 * x
n =   4  c0 =             Zeta(5)  err =  -5.086715 * x
n =   5  c0 =          1/945*Pi^6  err =  -6.050096 * x
n =   6  c0 =             Zeta(7)  err =  -7.028541 * x
n =   7  c0 =         1/9450*Pi^8  err =  -8.016067 * x
n =   8  c0 =             Zeta(9)  err =  -9.008951 * x
n =   9  c0 =       1/93555*Pi^10  err = -10.004942 * x
n =  10  c0 =            Zeta(11)  err = -11.002707 * x
n =  11  c0 = 691/638512875*Pi^12  err = -12.001473 * x
n =  12  c0 =            Zeta(13)  err = -13.000796 * x
```

Evidently, the coefficient of x in the truncated correction $c_0^{(n)} + c_1^{(n)}x$ soon approaches $-(n+1)$, so with a short table of coefficients for small n, we can easily compute a tiny negative adjustment to the first term, which is just $\zeta(n+1)$. That value is computed rapidly by the mathcw library function `ZETNUM(n+1)` (see **Section 18.7** on page 579). Thus, for tiny x, we compute

$$\psi^{(n)}(x) = (-1)^{n+1} n! \left(x^{-(n+1)} + (\zeta(n+1) + c_1^{(n)} x) \right).$$

The correction from the inner parenthesized sum is small, but its inclusion makes correctly rounded results more likely near the pole at the origin.

Finally, there is an asymptotic expansion that can be used for sufficiently large arguments:

$$\psi^{(n)}(z) \asymp (-1)^{n+1} z^{-n} \left((n-1)! + \tfrac{1}{2} n!\, z^{-1} + \sum_{k=1}^{\infty} B_{2k} \frac{(2k+n-1)!}{(2k)!} z^{-2k} \right).$$

Each of the three terms in the outer parenthesized sum can be the dominant one, depending on the values of n and x. The factorials and powers are subject to floating-point overflow and underflow, so for computational purposes, we rewrite the expansion like this:

$$\psi^{(n)}(z) \asymp (-1)^{n+1} z^{-n} n! \left(\frac{1}{n} + \frac{1}{2z} + \frac{1}{n!} \sum_{k=1}^{\infty} B_{2k} \frac{(2k+n-1)!}{(2k)!} z^{-2k} \right).$$

In that form, the terms in the parenthesized sum are small, and the underflow and overflow problems reside entirely in the outer multiplier $z^{-n} n!$. We discuss later in **Section 18.3.2** on page 557 how that factor is handled.

The terms in the asymptotic expansion require the rapidly growing Bernoulli numbers, B_{2k} (see **Section 18.5** on page 568). As always with asymptotic series, the sum can be accumulated only while the term magnitudes decrease. With the help of the limiting value for the ratio $B_{2k}/(2k)!$ that we derive later, we can estimate the terms like this:

$$t_k = B_{2k} \frac{(2k+n-1)!}{(2k)!} z^{-2k} \approx \frac{(-1)^{k+1} 2(2k+n-1)!}{(2\pi z)^{2k}}.$$

The critical term ratio is then

$$\left| \frac{t_{k+1}}{t_k} \right| \approx \frac{(2k+n+1)(2k+n)}{(2\pi z)^2} \approx \left(\frac{k}{\pi z} \right)^2, \qquad \text{provided } k \gg n.$$

Table 18.4: Accuracy of the asymptotic series for polygamma functions, $\psi^{(n)}(x)$. The numbers tabulated are the counts of correct decimal digits for various orders (columns) and arguments (rows) when term growth begins.

							n							
x	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**20**	**30**	**40**	**50**
10	25	23	22	21	19	18	17	16	16	15	8	4	2	1
20	52	50	48	47	45	44	43	41	40	39	30	23	17	13
30	80	77	75	74	72	70	69	67	66	65	54	45	37	31
40	107	104	102	100	99	97	95	94	92	91	78	68	60	52
50	134	131	129	127	125	124	122	120	119	117	104	93	83	75

We therefore expect that we can sum terms until $k \approx 3|z|$.

A numerical experiment for various orders and real arguments produces the data shown in Table 18.4 that predict the maximum accuracy attainable with the asymptotic series. We conclude that, for the 16-digit formats typical of the `double` data type, we can handle $n \leq 8$ for $x = 10$, and $n \leq 40$ for $x = 20$. For larger x values, the asymptotic series is sufficiently accurate, and converges quickly, but premature overflow in the terms because of the limited exponent range of hardware floating-point arithmetic remains a nuisance.

The authors of the previously cited paper [TS69] suggest that the asymptotic series be used whenever $x > \max(\frac{1}{2}(n+d), n)$, where d is the precision in decimal digits. A rough fit to our tabulated data shows that the coefficient $\frac{1}{2}$ in that relation should be replaced by 0.921 when $n = 0$, by 0.846 when $n = 1, \ldots,$ and by 0.384 when $n = 19$. That last coefficient can be reused for larger n values.

18.3.1 Applications of polygamma functions

One of the important uses of psi and polygamma functions is summation of finite and infinite series of reciprocal linear functions. For constants b and c and integer $n > 1$, we have these relations:

$$\begin{aligned}
\sum_{k=0}^{K-1} \frac{1}{kb+c} &= \frac{\psi((Kb+c)/b) - \psi(c/b)}{b}, \\
\sum_{k=0}^{K-1} \frac{(-1)^k}{kb+c} &= \frac{G((b+c)/b) - (-1)^K G((Kb+c)/b)}{2b}, \\
G(x) &= \psi(\tfrac{1}{2}(x+1)) - \psi(\tfrac{1}{2}x), && \textit{Bateman G function,} \\
G(1+x) &= \frac{2}{x} - G(x), && \textit{upward recurrence,} \\
G(1-x) &= \frac{2\pi}{\sin(\pi x)} - G(x), && \textit{reflection rule,} \\
G(x) &= \sum_{k=0}^{\infty} \frac{2}{(2k+x)(2k+x+1)}, \\
\sum_{k=0}^{\infty} \frac{1}{(kb+c)^n} &= \frac{\psi^{(n-1)}(c/b)}{(-b)^n (n-1)!}, \\
\sum_{k=0}^{\infty} \frac{(-1)^k}{(kb+c)^n} &= \frac{\psi^{(n-1)}(c/(2b)) - \psi^{(n-1)}((b+c)/(2b))}{(-2b)^n (n-1)!}.
\end{aligned}$$

The Bateman G function cannot be reliably computed from the difference of two psi functions, because they have nearly identical values for large x. On the positive axis, $G(x)$ is a slowly decaying positive function, with $G(0) = +\infty$ and $G(\infty) = 0$. On the negative axis, $G(x)$ has poles at negative integers, and otherwise, is negative in unit intervals with odd left endpoints, and positive in intervals with even left endpoints. We do not implement the Bateman function in the mathcw library, but if the reader wishes to do so, a good starting point could be a modification of the code for $\psi(x)$ in `psix.h`.

Brychkov's book of tables [Bry08, §5.1 and §6.2] contains a substantial collection of formulas involving sums and integrals of the psi function, and a few for the polygamma functions.

18.3.2 Computing the polygamma functions

Spanier and Oldham [SO87, Chapter 44] discuss the psi and polygamma functions, and sketch simple iterative schemes for their computation. Those algorithms have been implemented in the file `pgamma.hoc`, and comments there document numerical experiments that show that the code is capable of reasonable accuracy for small arguments and orders. However, the algorithms are unsuitable for a general library implementation that must handle the entire floating-point range.

Baker [Bak92, pages 109–114] and Thompson [Tho97, §6.2] provide low-accuracy code for the psi function, and the polygamma functions of orders $n = 1$ and $n = 2$. Moshier [Mos89, §7.3] has a more careful and accurate treatment, but only for the psi function. Zhang and Jin [ZJ96, Chapter 3] cover only the psi function, and accuracy is low.

The complex, and lengthy, presentation of the properties of the polygamma functions in **Section 18.3** on page 547 shows that they are mathematically difficult functions. We augmented that description with numerical investigations of the practicality of software implementations of mathematical relations for the polygamma functions, and found repeatedly that the standard recurrences and series are subject to severe digit loss from subtractions, and that premature overflow or underflow is likely in terms that involve factorials, or powers, or, on the negative axis, cotangents whose values are periodic with values in $[-\infty, +\infty]$.

Symbolic-algebra systems often hide the problem of accuracy loss in numerical computation by redoing the work in successively higher internal precision until an answer has been obtained that is stable to at least the user's requested precision. Those systems reduce the problem of premature overflow by having extremely wide exponent ranges. For example, Maple handles numbers with more than 25 million decimal digits, and represents exponents of 10 as 32-bit signed integers, allowing numbers as large as $10^{2\,147\,483\,646}$. Mathematica uses binary floating-point arithmetic, with a largest value near $10^{323\,228\,458}$. The ranges of other algebra systems differ, but are still much larger than those of hardware floating-point arithmetic.

In conventional programming languages, we do not have the luxury of adjustable precision or extreme exponent ranges, and we also want to make computations fast. The only reasonable choice if we are to achieve acceptable accuracy for library versions of the polygamma functions is to use a longer floating-point type whenever there is danger of overflow or subtraction loss. The IEEE 754 design with representations of overflowed values as signed infinities allows nonstop computing, but in older floating-point designs where overflow is often fatal, and in some, where exponent ranges may be fixed for all precisions, it is difficult to program polygamma function computations to avoid job-terminating overflow.

Our algorithm for the polygamma functions is based on that in Thompson's book [Tho97, §6.2], where only $\psi(x)$, $\psi^{(1)}(x)$, and $\psi^{(2)}(x)$ for nonnegative x are treated in about four pages of Fortran code with a design goal of ten-decimal-digit accuracy. However, our code permits computation of the polygamma functions for arbitrary n, subject to the overflow limits of the chosen floating-point type when $x < 0$. For positive arguments, x, our code produces results that are accurate to working precision when a higher intermediate precision is available.

Because we have no widely applicable range reduction formulas for the polygamma functions, and because we have two parameters n and x, we cannot reduce the computation to a small interval where a single polynomial approximation suffices, as we do for many of the elementary functions. Instead, we are often forced to sum general series until terms become small enough to be ignored, or in the case of the asymptotic series, until terms grow.

We name our function family `PGAMMA(n,x)`, and the lengthy code in the file `pgammx.h` follows these steps in order:

x is a NaN : Set a domain error, and set the result to that NaN.

$n < 0$: Set a domain error, and set the result to a quiet NaN.

$n = 0$: Set the result to `PSI(x)`, so that henceforth, we have $n > 0$.

x is $-\infty$: Set a range error, and set the result to $+\infty$, even though the sign is indeterminate, and the function range is $[-\infty, +\infty]$. A NaN could also be a suitable return value for this case, but there is no ISO Standard to guide us.

x is $+\infty$: Set the result to -0 if n is even, and $+0$ if n is odd.

$|x| < \frac{1}{2}$ **and** $(n+1)(\mathrm{logb}(x)+1) < -(t+1)$: Use the leading term of the sum with terms $(x-k)^{-n-1}$, plus the zeta-number correction described on page 554.

$x < 0$: Use the reflection formula to move the computation to the positive axis. Do the computation in the next higher precision because of the steep behavior of the polygamma functions of negative arguments. Use the `COTANPI()` family for accurate computation of the trigonometric term. Apply the cotangent argument-sum formula to replace $\cot(\pi(1-x))$ by $\cot(-\pi x) = -\cot(\pi x)$, guaranteeing an exact argument. Handle the derivatives of the cotangent for n in $[1,25]$ with inline code generated by a symbolic-algebra system, and for arbitrary order n, with a symbolic-derivative routine based on the code shown in Section 18.3 on page 550.

x **in** $[0, \frac{1}{2})$: Use the reflection formula to move the computation to the interval $[\frac{1}{2}, 1]$ (described two regions later).

x **in** $(1 - \texttt{XCUT\_1}, 1 + \texttt{XCUT\_1})$: Starting from the second term of the series expansion of $\psi^{(n)}(1+x)$, sum terms to convergence, then add the first term. Call the `ZETNUM()` function for values of the factors $\zeta(n+k+1)$ in the coefficients. The factorials in the coefficients can be eliminated by using a straightforward recurrence for the terms. The rounding errors from repeated multiplications in the recurrence are tolerable because the term magnitudes decrease. The value of `XCUT_1` is $1/(16(n+1))$.

x **in** $[\frac{1}{2}, \frac{3}{2})$: Because we have no usable series expansion in this region for small n, use one of fifteen separate Chebyshev expansions for n in $[1,15]$. For the 256-bit formats, fits are needed for n in $[1,30]$. For larger orders, sum the series containing terms $(x+k)^{-n-1}$. In both cases, compute the result in the next higher precision, and return an error estimate along with the function value in working precision.

The large number of polynomial fits is regrettably necessary in order to avoid excessive term counts in the series summation. The UNIX `size` command applied to the compiled polygamma object files reports storage sizes of 32 KB to 148 KB, values that are exceeded in the `mathcw` library only by the power functions. Those sizes may be excessive for historical architectures, but are unlikely to be of concern on modern systems.

x **in** $[\frac{3}{2}, \frac{5}{2})$: Thompson's algorithm in this region uses upward recurrence from the interval $[\frac{1}{2}, \frac{3}{2})$. However, as n increases, subtraction loss soon destroys accuracy. Consequently, we use the same algorithm as on the interval $[\frac{1}{2}, \frac{3}{2})$, but with new polynomial fits.

x **in** $[\frac{5}{2}, \infty)$: Use the m-step downward recurrence formula from the region where the asymptotic series is usable, doing all calculations in the next higher precision. If term growth is encountered, recompute the polygamma function $\psi^{(n)}(x+m)$ from its series with terms $(x+m-k)^{-n-1}$. Compute the argument $x+m$ and the function in the next higher precision to eliminate argument error.

In the asymptotic expansion, we promised earlier to discuss the computation of the outer factor $x^{-n}n!$. We use the asymptotic expansion only when $x > 1$, so x^{-n} can only underflow. With the `float` data type on many systems, $n!$ overflows already for $n > 34$, and with the `double` type in IEEE 754 arithmetic, $n!$ is representable only for $n < 171$. Mathematically, we can write the factor like this:

$$\begin{aligned}
x^{-n}n! &= \exp(\log(x^{-n}n!)),\\
&= \exp(-n\log(x) + \log(n!)),\\
&= \exp(-n\log(x) + \log(\Gamma(n+1))),\\
&\approx \texttt{EXP(-n * LOG(x) + LGAMMA(n + 1))}.
\end{aligned}$$

The last line relates the factor to standard library functions. The logarithms solve the premature-overflow problem. If the expression underflows, it does so with a large negative argument of the exponential. If it overflows, then the argument must be large and positive, and the exponential function is programmed to handle that case sensibly.

Unfortunately, that apparently simple solution is unsatisfactory, because of the problem of error magnification in the exponential function: its error is proportional to the magnitude of its argument (see Section 4.1 on page 61).

The first version of the code in `pgammx.h` computed x^{-n} from `IPOW(x,-n)` and $n!$ from `FACT(n)`. If the power underflowed to a subnormal or zero, or the factorial overflowed, the code switched to the exponential form. Testing showed large errors when either n or x is large, and that problem was traced to the inherent error magnification of the exponential function.

The solution is to exploit the fact that floating-point arithmetic represents numbers in exponential form, just as we did in our implementations of the exponential, logarithm, and power functions that are described in earlier chapters of this book. That is, we rewrite the expressions like this:

$$\begin{aligned}
n! &= \beta^{\log_\beta(n!)} \\
&= \beta^{j+g}, && \textit{for integer } j\textit{, and fractional } g \textit{ in } [-\tfrac{1}{2}, \tfrac{1}{2}], \\
x &= f \times \beta^k, && \textit{for integer } k\textit{, and fractional } f \textit{ in } [1/\beta, 1), \\
k &= \texttt{LOGB(x)} + 1, && \textit{exact computation of } k, \\
f &= \texttt{SCALBN(x, -k)}, && \textit{exact computation of } f, \\
x^{-n} &= f^{-n} \times \beta^{-kn}, \\
x^{-n}n! &= f^{-n} \times \beta^g \times \beta^{j-kn}, \\
&= \texttt{SCALBN(IPOW(f,-n) * EXPBASE(g), j - k * n)}
\end{aligned}$$

Here, `EXPBASE()` means `EXP2()`, `EXP8()`, `EXP10()`, or `EXP16()`, depending on the floating-point base. The argument of that function is small, so error magnification is eliminated, and the function value lies in the small interval $[1/\sqrt{\beta}, \sqrt{\beta}]$. Because of the restrictions on f and n, we have `IPOW(f,-n)` > 1. Premature overflow in that power, or its product with the exponential, is still possible, but much less likely than the underflow of x^{-n}. Integer overflow is also possible in the expression $j - kn$, but we can handle that with our safe-integer-arithmetic test functions, `is_add_safe()` and `is_mul_safe()`.

We could find $\log_\beta(n!)$ from $\log(n!)\log(\beta)$, and compute that value from $\log(\Gamma(n+1))\log(\beta) =$ `LGAMMA(n + 1) * LOG_BETA`, where the last factor is a compile-time constant. However, the log-gamma function is expensive, and the required range of n is often modest, so we added a new function family, `LOGBFACT(n,&j)`, to the `mathcw` library. The code in `lgbfax.h` uses table lookup to find the required value, and returns a value g in $[-\frac{1}{2}, \frac{1}{2}]$ such that $\log_\beta n! = j + g$, where the integer j is returned via the second pointer argument. However, if j is too large to represent as an integer, the entire function result is contained in g, which is then unrestricted. To reduce table size, the code only stores high-precision values of $\log(n!)$ at steps of a size chosen so that we can find $\log(n!) = \log(n(n-1)\cdots(m+1)m!) = \log(n(n-1)\cdots(m+1)) + \log(m!)$, where the product in the first logarithm is exact, and the second logarithm is found in the reduced table. For example, storage of the logarithms of 0!, 5!, 10!, ... allows the code to find intermediate values with at most three multiplications and a logarithm. With the IEEE 754 binary 53-bit significand, the products are exact for up to 13-bit factors, so we can handle $n < 16\,384$ that way. In practice, we limit the maximum tabulated value to n such that $(\frac{1}{2}n)!$ is just below the overflow limit. The code in `lgbfax.h` handles larger n values by calling the log-gamma function.

The extra precision provided by the representation $\log_\beta(n!) = j + g$ allows errors in similar expressions involving $n!$ to be reduced or eliminated, justifying the addition of `LOGBFACT(n,&j)` to the library.

Because most of our algorithm works in the next higher precision, tests of the `float` and `double` functions in binary and decimal arithmetic show relative errors below $\frac{1}{2}$ ulp, even for orders up to 5000. For negative arguments, the steepness of the polygamma functions soon reduces accuracy, and the results are not trustworthy for $n > 40$. Much higher precision appears to be needed to solve that problem. When a higher precision data type is not available, accuracy suffers, as shown in **Figure 18.10** on the facing page.

18.3.3 Retrospective on the polygamma functions

Our algorithm for accurate computation of the polygamma functions is complicated, and the code in `pgammx.h` is intricate and long. It is therefore reasonable in retrospect to ask whether simpler approaches are possible. Certainly none of the software in the previously cited works [TS69, SO87, Mos89, Bak92, Tho97], except possibly that of Moshier for the psi function, provides the accuracy required for the `mathcw` library.

At the start of our description of the polygamma functions, we gave an integral representation, and integrals can often be evaluated to reasonable accuracy by numerical quadrature. There are several methods that follow early work of Gauss, differing in range $[a, b]$, weight functions $w(x)$, nodes x_k, and weights w_k:

$$\int_a^b f(x)w(x)\,dx \approx \sum_{k=1}^{N} w_k f(x_k).$$

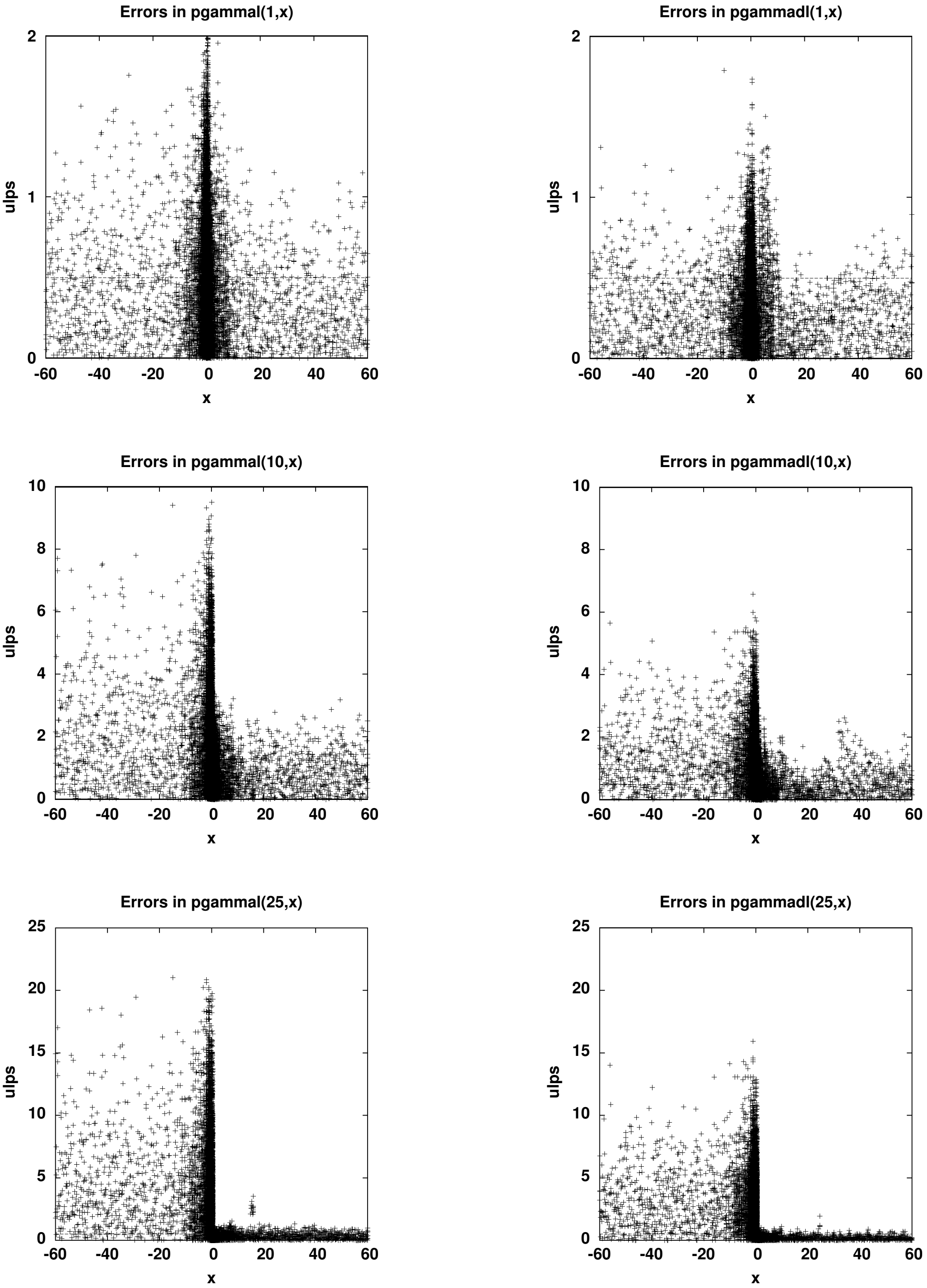


Figure 18.10: Errors in the 80-bit binary `pgammal(n,x)` and 128-bit decimal `pgammadl(n,x)` functions for $n = 1, 10$, and 25, where no higher intermediate precision is yet available. The errors are computed relative to high-precision values from Maple. Notice that the vertical ranges differ, and that errors for our implementation of the polygamma functions of order n are generally below n ulps.

Those methods have the property that the weights are all positive, and sum to one. The error in the quadrature is proportional to a scaled high-order derivative, $f^{(2N)}(x)/(2N)!$, for some x in the interval of integration. The quadrature is therefore exact when $f(x)$ is a polynomial of degree $2N-1$ or less.

For the polygamma functions, the most suitable method is N-point *Gauss–Laguerre quadrature*, characterized by an infinite range, and a decaying exponential weight function:

$$\int_0^\infty f(x)\exp(-x)\,dx \approx \sum_{k=1}^{N} w_k f(x_k).$$

If we replace t by t/z in our original integral, we can find the function needed for the Gauss–Laguerre form:

$$\psi^{(n)}(z) = (-1)^{n+1}\int_0^\infty \frac{(t/z)^n \exp(-t)}{z(1-\exp(-t/z))}\,dt,$$

$$f(z) = \frac{(t/z)^n}{z(1-\exp(-t/z))} \approx \frac{-\texttt{IPOW(t / z, n)}}{\texttt{z * EXPM1(-t / z)}}.$$

Some early research papers and later books [SZ49, AS64, SS66, DR84, Zwi92, OLBC10] tabulate the nodes x_k and weights w_k for members of the Gauss quadrature family, but the orders and accuracy are too low for our needs.

Fortunately, modern symbolic-algebra systems make it possible to compute the nodes and weights for any order and precision. The Maple file `glquad.map` provides functions for automatic generation of Gauss–Laguerre quadrature functions in C and hoc. Some of those functions are incorporated in a test program, `glquad.c`, that compares the accuracy of the quadrature approach with that provided by our `pgammal(n,x)` function.

Tests with `glquad.c` for a range of orders and arguments show that 100-point quadrature is reasonably effective for $n < 100$ and $x > 1.5$, with errors below 25 ulps in the 128-bit IEEE 754 format. However, for smaller x values, accuracy deteriorates rapidly, and only a few leading digits are correct. The problem is that, although the integrand is positive, for small x, its maximum occurs well past the last node point, which, for an order-N Gauss–Laguerre quadrature, lies below $4N$. The nodes therefore fail to sample the integrand where it is large. Although the Gauss–Laguerre weights are easily obtained from the nodes, no explicit formula for the nodes is known. For large N, their accurate determination requires high precision, and long computing times, in a symbolic-algebra system.

We conclude that numerical quadrature *could* be a component of general software for computing polygamma functions, but it fails to provide a complete solution. The small-argument region is the difficult one where convergence of standard mathematical formulas is poor, but downward recurrence solves that problem.

18.4 Incomplete gamma functions

The gamma function can be represented as an integral of the form

$$\Gamma(a) = \int_0^\infty t^{a-1}\exp(-t)\,dt, \qquad a > 0.$$

That integral can be split into lower and upper parts, called the *ordinary* and *complementary* incomplete gamma functions, defined as follows for $x \geq 0$:

$$\begin{aligned} \gamma(a,x) &= \int_0^x t^{a-1}\exp(-t)\,dt, && \textit{ordinary,} \\ \Gamma(a,x) &= \int_x^\infty t^{a-1}\exp(-t)\,dt, && \textit{complementary,} \\ \gamma(a,x)+\Gamma(a,x) &= \Gamma(a), && \textit{by definition.} \end{aligned}$$

It is helpful to remember that the *lowercase* incomplete gamma refers to the *lower* region, and the *uppercase* incomplete gamma handles the *upper* region.

Those two are among the most important of the special functions, and they are more difficult to compute than the single-argument elementary functions. Our notation is common in mathematical treatments of those functions, and the complete and incomplete functions with the uppercase Greek letter are distinguished by their argument counts.

Those functions are not part of the Standard C or POSIX libraries, but we cover them in this book, and include them in the mathcw library, for reasons to be revealed shortly.

The Maple symbolic-algebra system provides the complementary function as `GAMMA(a,x)`, and Mathematica calls that function `Gamma[a,x]`. Maxima calls it `gamma_incomplete(a,x)`, MuPAD names it `igamma(a,x)`, and PARI/GP calls it `incgam(a,x)`. Those algebra systems expect the ordinary incomplete gamma function to be obtained by symbolic subtraction from the complete gamma function. That omission is unfortunate, because numerical evaluation of $\gamma(a,x)$ then requires a precision high enough to hide the subtraction loss.

From the definitions of the incomplete gamma functions, we have these limiting cases:

$$\begin{aligned} \gamma(a,0) &= 0, & \Gamma(a,0) &= \Gamma(a), \\ \gamma(a,\infty) &= \Gamma(a), & \Gamma(a,\infty) &= 0. \end{aligned}$$

Although the incomplete gamma functions are defined only for $x \geq 0$, they can be extended so that their first argument, a, can range over $(-\infty, +\infty)$. The complicated behavior of the ordinary gamma function at negative arguments, with poles at zero and negative integer arguments, carries over to $\gamma(a,x)$, which for all $x \geq 0$, has poles when a is a negative integer. For $x > 0$, the complementary function, $\Gamma(a,x)$, is free of poles.

Like the complete gamma function, the incomplete gamma functions readily exhibit floating-point underflow and overflow when computed numerically. For that reason, it is conventional to work instead with scaled functions that have better numerical behavior, and have definitions that depend on the sign of the argument a:

$$\begin{aligned} g(a,x) &= \begin{cases} \gamma(a,x)/\Gamma(a), & \textit{when } 0 < a, \\ 1 - \Gamma(a,x)/\Gamma(a), & \textit{when } a \leq 0, \end{cases} \\ G(a,x) &= \begin{cases} \Gamma(a,x)/\Gamma(a), & \textit{when } 0 < a, \\ x^{-a} \exp(x)\Gamma(a,x)/\Gamma(a), & \textit{when } a \leq 0. \end{cases} \end{aligned}$$

We assume henceforth that $x \geq 0$ without explicitly stating that restriction. Both functions are nonnegative, and satisfy the relation

$$g(a,x) + G(a,x) = 1, \qquad \textit{when } 0 < a.$$

Under the stated condition, both functions lie in $[0,1]$.

The incomplete gamma functions are indeterminate when both arguments are zero, and it is conventional to use these limiting values of the scaled functions:

$$\begin{aligned} g(a,0) &= 0, & G(a,0) &= 1, & &\textit{when } 0 < a, \\ g(a,0) &= 1, & G(a,0) &= \infty, & &\textit{when } a = 0, \\ g(a,0) &= \infty, & G(a,0) &= 1/|a|, & &\textit{when } a < 0. \end{aligned}$$

The scaled functions satisfy these recurrence relations:

$$\begin{aligned} G(a+1,x) &= G(a,x) + x^a \exp(-x)/\Gamma(a+1), \\ g(a+1,x) &= g(a,x) - x^a \exp(-x)/\Gamma(a+1). \end{aligned}$$

The first of those relations is computationally stable only in the upward direction, and the second only in the downward direction.

The incomplete gamma functions arise in various areas of chemistry, engineering, mathematics, physics, probability, and statistics. For our purposes, they are of particular interest because of their relations to other important functions:

$$\begin{aligned} \operatorname{chisq}(\nu,x) &= \gamma(\nu/2, x/2)/\Gamma(\nu/2), & &\nu = 1,2,3,\ldots, \\ &= g(\nu/2, x/2), \\ \operatorname{erf}(x) &= g(\tfrac{1}{2}, x^2), \\ \operatorname{erfc}(x) &= G(\tfrac{1}{2}, x^2), \\ \exp(-x) &= \Gamma(1,x), & &x \geq 0, \end{aligned}$$

$$\begin{aligned} &= G(1,x), \\ E_1(x) &= \Gamma(0,x), \\ E_\nu(x) &= x^{\nu-1}\Gamma(1-\nu,x), \\ g(n+1,x) &= 1 - \exp(-x)\sum_{k=0}^{n} x^k/k!, && n = 0,1,2,\ldots. \end{aligned}$$

The chi-square (Greek letter *chi*, χ) probability distribution function, and its inverse function, is useful for the analysis of models of experimental data. Its arguments are the number of degrees of freedom, ν (Greek letter *nu*), and the chi-square measure, x; see **Section 7.15.1** on page 197 for details. In the statistics literature, two common notations for the chi-square probability functions are related to our functions like this:

$$\begin{aligned} P(\chi^2|\nu) &= \mathrm{chisq}(\nu,\chi^2), & Q(\chi^2|\nu) &= 1 - P(\chi^2|\nu), && \nu = 1,2,3,\ldots, \\ &= g(\nu/2,\chi^2/2), & &= G(\nu/2,\chi^2/2), \\ &= \mathrm{gami}(\nu/2,\chi^2/2), & &= \mathrm{gamic}(\nu/2,\chi^2/2). \end{aligned}$$

Here, χ^2 is the chi-square measure and we introduce the functions gami() and gamic() shortly. The *Handbook of Mathematical Functions* summarizes their properties and gives extensive tables of their values; see [AS64, Chapter 26] and [OLBC10, Chapter 8].

The relation of the incomplete gamma functions to the two error functions provides a useful check for computer code, although our algorithms for the error functions described in **Section 19.1.2** on page 595 are faster, and achieve better accuracy, than is likely to be possible for the more difficult incomplete gamma functions.

Although we do not discuss it further in this book, the exponential integral $E_\nu(x)$ is important for some computational areas, such as transport problems and molecular scattering theory. When ν is a negative integer, the functions $E_\nu(x)$ are related to certain integrals that appear in the quantum mechanics of molecules.

The algorithm that we use for the incomplete gamma functions is that of Gautschi [Gau79a, Gau79b], although our code is developed independently of his. Subsequent articles on those functions in the journal of publication, and in statistical journals, do not offer the accuracy that we require for the mathcw library.

The presence of two arguments in the incomplete gamma functions makes a significant difference in how they are computed. Rational polynomial fits in a single variable are no longer applicable, so we have to evaluate the functions from series summation, continued fractions, and recurrence relations, some needing hundreds of terms, and the computations often require other library functions. Thus, we do not expect to achieve either the high accuracy, or the speed, that we reach for the elementary functions.

For $a > 0$, we can compute the smaller of $g(a,x)$ and $G(a,x)$, and obtain the larger by a single subtraction. However, which of them is smaller depends on the arguments. Thus, our code computes them together, and wrapper functions then make them available separately. Our function interface looks like this:

```
void   gamib (double result[/* 2 */], double a, double x);
double gami  (double a, double x);
double gamic (double a, double x);
```

Companions for other floating-point types take the usual suffixes. The first of them, `gamib()`, returns $g(a,x)$ in `result[0]` and $G(a,x)$ in `result[1]`.

Gautschi's treatment of those functions is thorough, and requires a 15-page article. Here, we only sketch the steps, leaving out the many details that are necessary in the computer program. The need for several separate computational recipes is a major impediment to implementing the code, and Gautschi's monolithic Fortran implementation is difficult to follow. Our code splits the job into 13 separate functions, of which `gamib()` is the only public one. All of the logic control is handled by a dozen `if` statements in `gamib()`, and all of the computation is relegated to the 12 private functions, each of which does a single task.

Which of the $g(a,x)$ and $G(a,x)$ functions to compute first is controlled by the values of two critical cutoffs, both positive:

$$\begin{aligned} a_c &= \begin{cases} \log(\frac{1}{2})/\log(x), & \text{when } 0 < x < \frac{1}{4}, \\ x + \frac{1}{4}, & \text{when } x \geq \frac{1}{4}, \end{cases} \\ x_c &= \tfrac{3}{2}. \end{aligned}$$

With those cutoffs in hand, here are the steps of Gautschi's algorithm for the incomplete gamma functions:

- If either argument is a NaN, the results are set to that NaN, and `errno` is set to `EDOM`.
- If $x < 0$, the results are quiet NaNs, and `errno` is set to `EDOM`.
- The special cases listed earlier for zero and infinite arguments are handled quickly by direct assignment. Henceforth, we have finite a, and $x > 0$.
- If $x \leq x_c$ and $a < -\frac{1}{2}$, compute $G(a, x)$ by upward recurrence, and compute $g(a, x)$ from its second definition in terms of $G(a, x)$ and $\Gamma(a)$. The number of terms in the recurrence is essentially $\lfloor -a \rfloor$, so to prevent unbounded execution time, our code sets a loop limit of 2135, based on the observation that $\Gamma(-2135.5) \approx 5.28 \times 10^{-6185}$ is below the smallest subnormals in 128-bit binary and decimal arithmetic. For $a < -2135$, the code returns a quiet NaN, even though $G(-2135.5, 1) \approx 0.000\,468$.

  If a better algorithm can be found for handling negative a of large magnitude, that protective infelicity should be eliminated. In practice, it is unlikely to be met in most applications of the incomplete gamma functions.

  Gautschi's code does not make such a check, so it can be made to run arbitrarily long by passing it a negative a of large magnitude.
- If $x \leq x_c$ and a is in $[-\frac{1}{2}, a_c]$, compute $G(a, x)$ by the Taylor series for $\Gamma(a, x)$, noting the dependence of the definition of $G(a, x)$ on the sign of a. Set $g(a, x) = 1 - G(a, x)$ when $0 < a$, and otherwise compute it from its second definition.
- If $x \leq x_c$ and $a_c < a$, compute $g(a, x)$ by a power series, and set $G(a, x) = 1 - g(a, x)$.
- If $x_c < x$ and $a \leq a_c$, compute $G(a, x)$ from a continued fraction. If $0 < a$, set $g(a, x) = 1 - G(a, x)$. Otherwise, compute $g(a, x)$ from its second definition.
- Otherwise, we have $x_c < x$ and $a_c < a$. Compute $g(a, x)$ by a power series, and set $G(a, x) = 1 - g(a, x)$.

Most of those steps require at least one invocation of the exponential, logarithm, and complete gamma functions. That means that, despite the identity $\Gamma(a, 0) = \Gamma(a)$, a software implementation of the incomplete gamma function is unlikely to provide an independent route to the complete gamma function. The power series for $\gamma(a, x)$ is subject to serious subtraction loss, and requires careful rearrangement of terms to reduce that loss. The continued fraction is evaluated in the forward direction, using a transformation to a sum that reduces the term-growth problem that we discuss in **Section 2.7** on page 12, and still permits early loop exit when the next term added no longer affects the sum.

Our outline of the algorithm is clearly too limited to allow programming of the computational parts, but it does show how the computation may require either $g(a, x)$, or $G(a, x)$, or both, making it practical to produce both of them in one function. For some argument combinations, one of the functions is almost free.

Graphical displays of the errors in results from our initial code, as well as from that of several other implementations of those functions, revealed a serious problem: accuracy deteriorates for arguments of comparable magnitudes larger than about 30. Tracing the algorithm's execution revealed the culprit: the computation of the scale factor $x^a \exp(-x)/\Gamma(a)$. Although the exponential term is nicely restricted to $[0, 1]$, it is subject to underflow for large $|x|$. The other two terms can both overflow and underflow, depending on the values of a and x. Even though the scale factor may be representable in floating-point arithmetic if computed in exact arithmetic, underflows and overflows often prevent its computation as written.

Gautschi's code, and most others, handles that problem by rewriting the scale factor as $\exp(a \log(x) - x - \mathrm{lgamma}(|a|))$, and then accounting for the sign of $\Gamma(a)$ when $a < 0$.

When the computed scale factor is close to one, the argument of the exponential must be nearly zero, and thus, must have suffered massive subtraction loss, and has seriously lowered accuracy. For small arguments z, from the Taylor series, we know that $\exp(z) \approx 1 + z$, so the error in z is masked by the addition. Nevertheless, the measured errors reach more than 600 ulps for a in $[-100, 100]$ and x in $[0, 100]$.

One reasonable solution is to compute the scale factor in the next higher precision, if it is available. However, it seems better to retain the original form as long as its factors, and their products and quotients, neither overflow nor underflow. The range of arguments that must be supported makes the computation difficult, but the payoff is substantial, reducing the worst-case errors to about 5 ulps when $a > 0$.

The programming is simplified if we introduce floating-point companions, `is_fdiv_safe(x,y)` and `is_fmul_safe(x,y)`, to the functions for safe arithmetic with integers described in **Section 4.10.1** on page 74. They are private functions, each of about 20 lines of code, that check whether their arguments can be divided or multiplied without overflow, and without causing overflow themselves. They assume that their arguments are finite, and that underflows are harmless.

The scale factor can then be computed with this function:

```
static fp_t
pow_exp_gam (fp_t a, fp_t x)
{   /* compute x**a * exp(-x) / tgamma(a) avoiding premature underflow and overflow */
    fp_t e;                     /* EXP(-x) */
    fp_t g;                     /* TGAMMA(a) */
    fp_t p;                     /* POW(x,a) */
    fp_t r;                     /* FABS(result) */
    fp_t result;
    int use_exp_log, use_lgamma;
    static int do_init = 1;
    static fp_t EXP_ARG_UFL = FP(0.0);
    static fp_t TGAMMA_ARG_OFL = FP(0.0);

    if (do_init)
    {

#if defined(HAVE_FP_T_DECIMAL_LONG_LONG_DOUBLE)
        TGAMMA_ARG_OFL = FP(310942.551258823560);
#elif defined(HAVE_FP_T_DECIMAL_LONG_DOUBLE)
        TGAMMA_ARG_OFL = FP(22124.54995666246323632807135355445);
#elif defined(HAVE_FP_T_DECIMAL_DOUBLE)
        TGAMMA_ARG_OFL = FP(205.37966293287085);
#elif defined(HAVE_FP_T_DECIMAL_SINGLE)
        TGAMMA_ARG_OFL = FP(69.32968);
#elif defined(HAVE_FP_T_OCTUPLE) && defined(HAVE_IEEE_754)
        TGAMMA_ARG_OFL = FP(71422.483408627342);
#elif defined(HAVE_FP_T_QUADRUPLE) && defined(HAVE_IEEE_754)
        TGAMMA_ARG_OFL = FP(1755.548342904462917);
#elif defined(HAVE_FP_T_DOUBLE) && defined(HAVE_IEEE_754)
        TGAMMA_ARG_OFL = FP(171.6243765);
#elif defined(HAVE_FP_T_SINGLE) && defined(HAVE_IEEE_754)
        TGAMMA_ARG_OFL = FP(35.040);
#else
        fp_t t;                 /* TGAMMA(z) */
        fp_t z;

        z = FP(12.0);
        t = FP(39916800.0);     /* exactly representable TGAMMA(z) */

        for (;;)                /* advance by TGAMMA(z + 1) = z * TGAMMA(z) */
        {
            if (t >= (FP_T_MAX / z))    /* z * t would overflow */
                break;

            t *= z;             /* TGAMMA(z + 1) is finite */
            z++;                /* exact */
        }

        TGAMMA_ARG_OFL = z;

#endif /* defined(HAVE_FP_T_DECIMAL_LONG_LONG_DOUBLE) */

        EXP_ARG_UFL = LOG(FP_T_MIN);
        do_init = 0;
    }

    use_exp_log = 1;
    use_lgamma = 1;
    g = ZERO;

    do                          /* one-trip loop for breaks */
    {
        if (x < EXP_ARG_UFL)    /* EXP(-x) likely underflows */
            break;

        if (a > TGAMMA_ARG_OFL) /* TGAMMA(a) likely overflows */
            break;

        e = EXP(-x);

        if (e <= FP_T_MIN)      /* exp(-x) unexpectedly underflows */
            break;

        p = POW(x, a);

        if (p <= FP_T_MIN)      /* x**a underflows */
            break;
        else if (FP_T_MAX <= p) /* x**a overflows */
            break;

        g = TGAMMA(a);

        if (g <= FP_T_MIN)      /* TGAMMA(a) underflows */
            break;
        else if (FP_T_MAX <= g) /* TGAMMA(a) probably overflows */
            break;

        use_lgamma = 0;         /* TGAMMA(a) is finite and nonzero */

        /* Compute result = (p * e / g) without overflow */

        if (is_fmul_safe(p, e))
        {
            result = p * e;

            if (is_fdiv_safe(result, g))
                result /= g;
            else
                break;
        }
        else if (is_fdiv_safe(p, g))
        {
            result = p / g;
            result *= e;        /* possible (harmless) underflow */
        }
        else if (is_fdiv_safe(e, g))
        {
            result = e / g;
```

```
                if (is_fmul_safe(result, p))
                    result *= p;
                else
                    break;
            }
            else
                break;

            r = (result < ZERO) ? -result : result;

            if (r <= FP_T_MIN)      /* probable underflow */
                break;
            else if (r >= FP_T_MAX) /* probable overflow */
                break;

            use_exp_log = 0;

        } while (0);                /* end one-trip loop */

        if (use_exp_log)
        {
            hp_t d;
            int sign_tgamma;

            sign_tgamma = use_lgamma ? 1 : ((g < ZERO) ? -1 : 1);
            d = (hp_t)x + (use_lgamma ? HP_LGAMMA_R((hp_t)a, &sign_tgamma)
                                      : HP_LOG((hp_t)(g < ZERO ? -g : g)));
            result = (fp_t)HP_EXP((hp_t)a * HP_LOG((hp_t)x) - d);

            if (sign_tgamma < 0)
                result = -result;
        }

        return (result);
    }
```

A simpler version of that function, pow_exp(), computes the factor $x^a \exp(-x)$ that is needed for the computation of $G(a,x)$.

In the one-time initialization block, the argument at which the exponential underflows is easy to determine from the logarithm, assuming that both functions are accurate, as ours are. The smallest nonnegative argument for which the complete gamma function overflows is set by assignment for the common cases, and otherwise, is quickly estimated to the nearest smaller whole number by upward recurrence from a known value that is representable in all floating-point architectures, without needing to call the gamma function.

We assume that the exponential and power functions return the largest floating-point number, or Infinity, in the event that their results are too large to represent. When $|\Gamma(a)|$ is representable and already computed, we take its logarithm; otherwise, we require the extension of the log-gamma function that also returns the needed sign.

The safety checks prevent overflows that would be fatal on many older architectures, but we assume that underflows are silently flushed either to zero, or else to the smallest representable magnitude of the appropriate sign. The presence of a dozen break statements shows the considerable logical complexity of the computation.

Further testing showed that a significant source of error is subtraction loss in the computation of $\Gamma(a,x)$ for a in $[-\frac{1}{2}, a_c]$ and x in $[0, x_c]$. The easiest way to reduce that loss is to do that portion of the computation in the next higher precision, which is what our final version of the code does.

Because there are two arguments, the error plots require three-dimensional views, and are shown in **Figure 18.11** for binary and decimal single-precision functions, and in **Figure 18.12** for the corresponding double-precision functions. For positive arguments less than about 30, most of the errors remain well below 2 ulps.

The test programs gamierf*.c compare computed values of $g(\frac{1}{2}, x^2)$ and $G(\frac{1}{2}, x^2)$ with results from the highest-available precision of the error functions for logarithmically distributed x values chosen from several consecutive

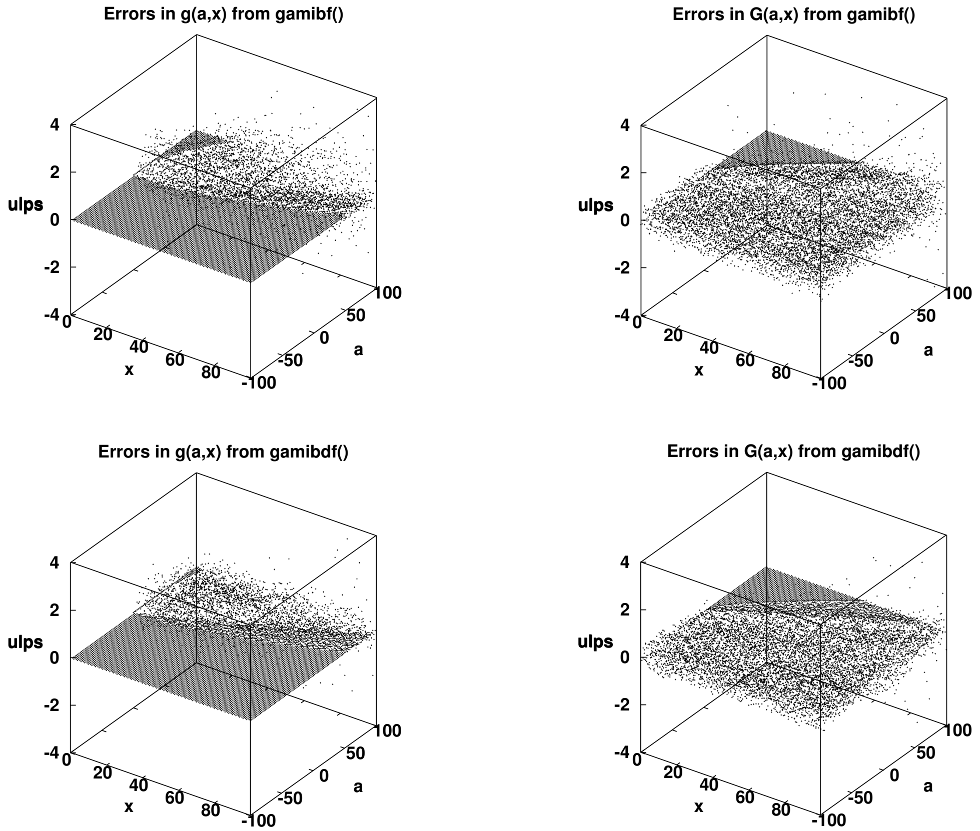


Figure 18.11: Errors in the `gamibf()` and `gamibdf()` functions. For large $|a|$ and large x, errors in the binary functions can reach about 15 ulps over the argument range shown here. The worst errors in the decimal functions are about 4 ulps.

subintervals of $[0, \infty)$, and purified so that x^2 is exact. For single- and double-precision versions of the incomplete gamma functions in both binary and decimal arithmetic, the largest errors are usually below 2.5 ulps, except for `gamibd()`, where errors reach 3.5 ulps.

Similar checks of the relation $G(1, x) = \exp(-x)$ with the test programs `gamiexp*.c` find no errors above 2.3 ulps.

The test programs `ixsq*.c` use the inverse of the chi-square function, which ultimately depends on the incomplete gamma functions, to reproduce several pages of tables in well-known mathematical handbooks and texts. The data in the published tables are given only to five or six digits, so reproducing those tables is only a rough test.

Other tests output triples of a, x, and $g(a, x)$ or $G(a, x)$ as Maple expressions that, when evaluated, report the error in ulps compared to high-precision values.

Treatments of the incomplete gamma function in other books (see [Bak92, Chapter 6], [GDT$^+$05, Chapter 7], [Mos89, Chapter 5], [PTVF07, Chapter 6], [Tho97, Chapter 6], and [ZJ96, Chapter 3]) usually have shorter code than ours, but they do not handle the case of $a \leq 0$, and the last also excludes $x = 0$. Except for Moshier's code, they support only the `double` type. None of them offers the more accurate computation of the scale factors that our `pow_exp()` and `pow_exp_gam()` functions provide.

Just before this book went to press, new work [GRAST16] was published that addresses some of the difficulties of computing the incomplete gamma function for negative arguments.

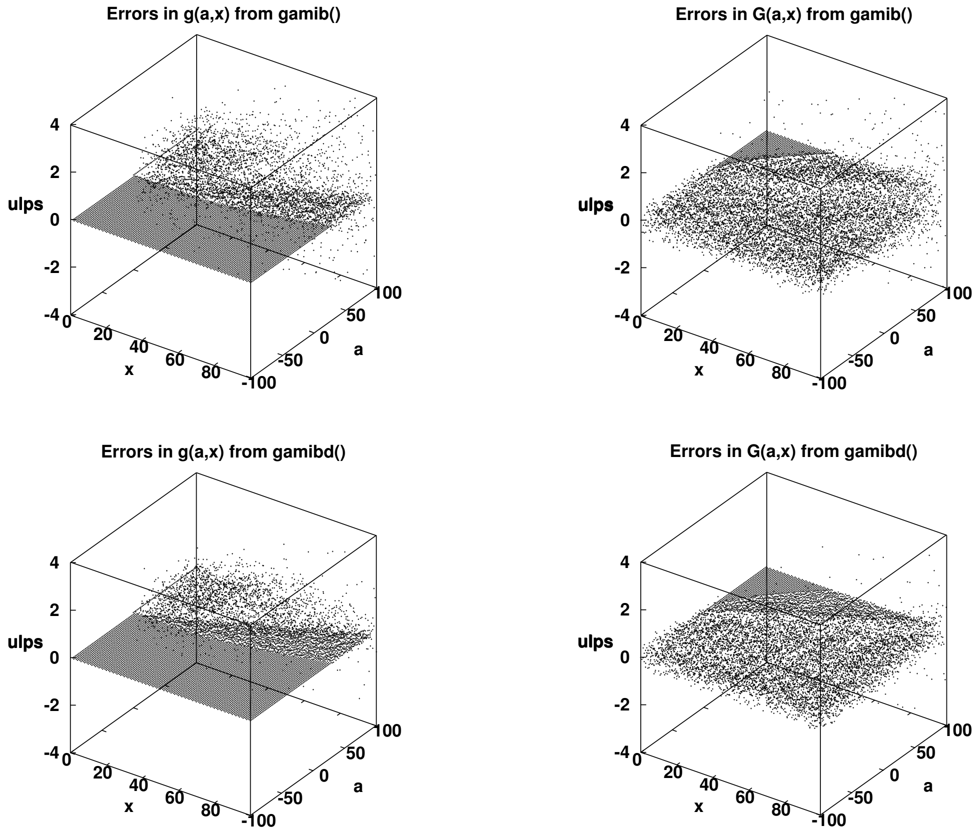


Figure 18.12: Errors in the `gamib()` and `gamibd()` functions. For large $|a|$ and large x, errors in the binary functions can reach about 15 ulps over the argument range shown here. The worst errors in the decimal functions are almost 6 ulps.

18.5 A Swiss diversion: Bernoulli and Euler

In this, and earlier, chapters, we encountered the famous *Bernoulli numbers*[3] in Taylor expansions of various functions, and listed the first two dozen in **Table 11.1** on page 304 to show their rapid growth.

The Bernoulli numbers were discovered, sometime after 1677, by Jacob (also called Jacques or James) Bernoulli, one of at least eight famous mathematicians and scientists in six generations of a family from Basel, Switzerland.[4] It was learned much later that, about the same time, the numbers were also discovered independently in Japan by Takakazu Seki Kowa. The work of both was published after their deaths, so the discovery dates are uncertain. Bernoulli's work was more widely known, so history gave him the credit.

The Bernoulli numbers arise often enough in the elementary and special functions that it is advisable in a mathematical software library to provide easy access to them, especially because they are not easily generated in conventional programming languages with arithmetic of fixed, and limited, precision.

[3] See entries *A000367* and *A002445* in Neal Sloane's *On-Line Encyclopedia of Integer Sequences* at `http://oeis.org/`, or in the printed book [SP95, sequences M4039 and M4189].

[4] See, for example, various online encyclopedia entries, including the *Dictionnaire historique de la Suisse* (`http://www.hls-dhs-dss.ch/textes/f/F23988.php`).

The world's first significant computer program may have been that written in 1842 by Lady Augusta Ada Lovelace for the computation of Bernoulli numbers [HH80, KT99]. See also `http://www.fourmilab.ch/babbage/sketch.html`.

Mathematically, the Bernoulli numbers, B_n, are defined as the coefficients in this expansion:[5]

$$\frac{x}{\exp(x) - 1} = \sum_{n=0}^{\infty} B_n \frac{x^n}{n!}.$$

Here is what the series expansion looks like in a Maple session, with a verification of the relation of the expansion coefficients to the Bernoulli numbers (and output reformatted into three columns):

```
> taylor(x/(exp(x) - 1), x = 0, 12);
                 2          4           6             8
1 - 1/2 x + 1/12 x  - 1/720 x  + 1/30240 x  - 1/1209600 x  +

               10      11
    1/47900160 x   + O(x  )

> for n from 0 to 10 do printf("%2d  %a\n", n, bernoulli(n)/n!) end do:
0  1                 4  -1/720              8  -1/1209600
1  -1/2              5  0                   9  0
2  1/12              6  1/30240            10  1/47900160
3  0                 7  0
```

Notice that the $B_n/n!$ values appear to form a decreasing sequence. Bounds given later show that to be true for *all* Bernoulli numbers. Because that ratio is common in series expansions that contain Bernoulli numbers, it is helpful to remember that fact in order to be able to estimate relative term magnitudes.

Mathematica makes those numbers available as `BernoulliB[n]`, Maxima as `bern(n)`, MuPAD as `bernoulli(n)`, PARI/GP as `bernfrac(n)`, and REDUCE as `Bernoulli(n)`.

In the expansion, notice that there is only one nonzero *odd-order* Bernoulli number, $B_1 = -\frac{1}{2}$; all others are zero. Expansions that involve Bernoulli numbers often require only the even ones, with coefficients B_{2n}.

The Bernoulli numbers turn up in the series expansions of several other functions, including these:

$$\begin{aligned}
\cot(z) &= \sum_{n=0}^{\infty} \frac{(-1)^n 2^{2n} B_{2n}}{(2n)!} z^{2n-1}, && \textit{for } |z| < \pi,\\
\coth(z) &= \frac{1}{z} + \sum_{n=1}^{\infty} (-1)^{n-1} \frac{2^{2n} B_{2n}}{(2n)!} z^{2n-1}, && \textit{for } |z| < \pi,\\
\csc(z) &= \frac{1}{z} + \sum_{n=1}^{\infty} \frac{(-1)^{n-1} 2(2^{2n-1} - 1) B_{2n}}{(2n)!} z^{n}, && \textit{for } |z| < \pi,\\
\operatorname{csch}(z) &= \frac{1}{z} + \sum_{n=1}^{\infty} \frac{2(2^{2n-1} - 1) B_{2n}}{(2n)!} z^{2n-1}, && \textit{for } |z| < \pi,\\
\tan(z) &= \sum_{n=1}^{\infty} (-1)^{n-1} \frac{B_{2n}(-4)^n (1 - 4^n)}{(2n)!} z^{2n-1}, && \textit{for } |z| < \tfrac{1}{2}\pi,\\
\tanh(z) &= \sum_{n=1}^{\infty} (-1)^{n-1} \frac{4^n (4^n - 1) B_{2n}}{(2n)!} z^{2n-1}, && \textit{for } |z| < \tfrac{1}{2}\pi,\\
\log(\cos(z)) &= \sum_{n=1}^{\infty} \frac{(-1)^n 2^{2n-1} (2^{2n} - 1) B_{2n}}{n(2n)!} z^{2n}, && \textit{for } |z| < \tfrac{1}{2}\pi,\\
\log(\sin(z)) &= \sum_{n=1}^{\infty} \frac{(-1)^n 2^{2n-1} B_{2n}}{n(2n)!} z^{2n}, && \textit{for } |z| < \pi,\\
\log(\tan(z)) &= \sum_{n=1}^{\infty} \frac{(-1)^{n-1} 2^{2n} (2^{2n-1} - 1) B_{2n}}{n(2n)!} z^{2n}, && \textit{for } |z| < \tfrac{1}{2}\pi,\\
\log(\Gamma(x)) &\asymp (x - \tfrac{1}{2}) \log(x) - x + \log(2\pi)/2 + \sum_{n=1}^{\infty} \frac{B_{2n}}{2n(2n-1)x^{2n-1}},
\end{aligned}$$

[5] See the *Bernoulli Number* article in Eric Weisstein's outstanding online resource, *MathWorld*, at `http://mathworld.wolfram.com/`, or in the print edition [Wei09].

$$\psi(z) \asymp \log(z) - 1/(2z) - \sum_{n=1}^{\infty} \frac{B_{2n}}{2nz^{2n}}, \qquad \text{for } |\arg(z)| < \pi,$$

$$\psi^{(n)}(z) \asymp (-1)^{n+1} z^{-n} n! \left(\frac{1}{n} + \frac{1}{2z} + \frac{1}{n!} \sum_{k=1}^{\infty} B_{2k} \frac{(2k+n-1)!}{(2k)!} z^{-2k} \right).$$

Another important example where Bernoulli numbers appear is the *Euler–Maclaurin summation formula* that relates an integral of a function over an interval $[a, b]$ to a sum of function values at successive integers with a correction:

$$\int_a^b f(t)\,dt = \sum_{k=a}^{b} f(k) - \tfrac{1}{2}[f(a) + f(b)] \qquad \text{trapezoid rule,}$$

$$- \sum_{k=1}^{\infty} \frac{B_{2k}[f^{(2k-1)}(b) - f^{(2k-1)}(a)]}{(2k)!}, \qquad \text{correction.}$$

The quantity following the equals sign on the first line is the famous *trapezoid rule* for numerical quadrature. The sum on the second line contains the rapidly decreasing coefficients $B_{2k}/(2k)!$, so as long as the higher-order derivatives remain small, that sum provides a small correction. Alternatively, it may sometimes be easy to do the integral, and hard to do the first sum, in which case, that sum is the unknown to be solved for.

The Bernoulli numbers can be calculated directly from this double sum:

$$B_n = \sum_{k=0}^{n} \frac{1}{k+1} \left(\sum_{m=0}^{k} (-1)^m \binom{k}{m} m^n \right).$$

They can be estimated with this strict inequality:

$$\frac{2(2n)!}{(2\pi)^{2n}} < (-1)^{n+1} B_{2n} < \frac{2(2n)!}{(2\pi)^{2n}} \frac{1}{1 - 2^{1-2n}}, \qquad \text{for } n = 1, 2, 3, \ldots .$$

A numerical computation in Maple shows how tight that inequality is by reporting the relative error of the bounds with respect to the Bernoulli number:

```
> for n in {seq(i, i = 1..9), seq(i, i = 10..90, 10)} do
>     u := (2 * (2*n)!) / ((2 * Pi)**(2*n)):
>     v := (-1)**(n + 1) * bernoulli(2*n):
>     w := u / (1 - 2**(1 - 2*n)):
>     printf("%3d  %10.3g  %13.3g\n", n, (u - v) / v, (w - v) / v)
> end do:
  1      -0.392          0.216          10   -9.54e-07       9.53e-07
  2     -0.0761         0.0559          20   -9.09e-13       9.09e-13
  3      -0.017         0.0147          30   -8.67e-19       8.67e-19
  4    -0.00406        0.00378          40   -8.27e-25       8.27e-25
  5   -0.000994       0.000961          50   -7.89e-31       7.89e-31
  6   -0.000246       0.000242          60   -7.52e-37       7.52e-37
  7   -6.12e-05       6.08e-05          70   -7.17e-43       7.17e-43
  8   -1.53e-05       1.52e-05          80   -6.84e-49       6.84e-49
  9   -3.82e-06       3.81e-06          90   -6.53e-55       6.53e-55
```

The output has been reformatted into two columns. The bounds are clearly excellent estimates, because they deliver about six correct digits already at $n = 10$.

From those tight bounds, we can now answer the question raised earlier about the behavior of the commonly occurring quotient of a Bernoulli number and a factorial:

$$\frac{B_{2n}}{(2n)!} \approx (-1)^{n+1} \frac{2}{(2\pi)^{2n}}, \qquad \text{Bernoulli/factorial quotient estimate.}$$

Here is a numerical demonstration of that estimate in hoc, including a report of the relative error:

```
hoc64> u_last = 1
hoc64> for (n = 1; n < 18; ++n)                                        \
hoc64> {
hoc64>     u = bernum(2*n) / factorial(2*n);
hoc64>     v = (-1)**(n + 1) * 2 / (2*PI)**(2*n)
hoc64>     printf("%2d  % 6.2f  % 17.9..3e  % 17.9..3e  % 10.3e\n", 2*n, u_last / u, u, v, (u - v) / u)
hoc64>     u_last = u
hoc64> }
 2   12.00   8.333_333_333e-02   5.066_059_182e-02   3.921e-01
 4  -60.00  -1.388_888_889e-03  -1.283_247_782e-03   7.606e-02
 6  -42.00   3.306_878_307e-05   3.250_504_604e-05   1.705e-02
 8  -40.00  -8.267_195_767e-07  -8.233_624_348e-07   4.061e-03
10  -39.60   2.087_675_699e-08   2.085_601_411e-08   9.936e-04
12  -39.51  -5.284_190_139e-10  -5.282_890_090e-10   2.460e-04
14  -39.49   1.338_253_653e-11   1.338_171_693e-11   6.124e-05
16  -39.48  -3.389_680_296e-13  -3.389_628_495e-13   1.528e-05
18  -39.48   8.586_062_056e-15   8.586_029_281e-15   3.817e-06
20  -39.48  -2.174_868_699e-16  -2.174_866_624e-16   9.540e-07
22  -39.48   5.509_002_828e-18   5.509_001_515e-18   2.385e-07
24  -39.48  -1.395_446_469e-19  -1.395_446_385e-19   5.961e-08
26  -39.48   3.534_707_040e-21   3.534_706_987e-21   1.490e-08
28  -39.48  -8.953_517_427e-23  -8.953_517_394e-23   3.725e-09
30  -39.48   2.267_952_452e-24   2.267_952_450e-24   9.313e-10
32  -39.48  -5.744_790_669e-26  -5.744_790_668e-26   2.328e-10
34  -39.48   1.455_172_476e-27   1.455_172_476e-27   5.821e-11
```

We see that the Bernoulli-number estimate provides at least 10 decimal digits above B_{34}, and that the magnitude of the quotient $B_{2n}/(2n)!$ falls by $(2\pi)^2 \approx 40$ for each unit increase in n.

The quotient of successive Bernoulli numbers is sometimes of interest, and we can use our estimates to make this prediction:

$$\frac{B_{2n}}{B_{2n-2}} \approx -\frac{2n(2n-1)}{(2\pi)^2}.$$

A numerical experiment similar to the earlier ones shows that our estimate is correct to three figures already for B_{10}/B_8, and to six figures for B_{20}/B_{18}.

The Bernoulli numbers have this asymptotic relation:

$$\lim_{n\to\infty} B_{2n} \asymp 4\sqrt{\pi n}\left(\frac{n}{\pi e}\right)^{2n}.$$

It predicts only two correct decimal digits at $n = 10$, three at $n = 100$, and four at $n = 1000$.

For large n, there is an easier way to compute B_{2n} by summing a fast-converging series that we discuss later in **Section 18.7** on page 582.

Bernoulli numbers have a complicated recurrence relation that requires all previous B_k values:

$$B_n = \frac{1}{2(1-2^n)}\sum_{k=0}^{n-1}\binom{n}{k}2^k B_k.$$

That relation can also be written like this:

$$B_0 = 1, \qquad B_1 = -\tfrac{1}{2}, \qquad \sum_{k=0}^{n-1}\binom{n}{k}B_k = 0, \qquad \text{for } n = 2,3,4,\ldots.$$

Because the Bernoulli numbers grow, and alternate in sign, there can sometimes be serious leading digit cancellation in the evaluation of those sums. The Bernoulli numbers therefore cannot be generated accurately from that formula without access to high-precision rational integer arithmetic, but their quick approach to the floating-point overflow limit in most hardware arithmetic designs means that it is feasible to precompute them in a symbolic-algebra system,

Table 18.5: Euler numbers of even order. Those of odd order are all zero. All Euler numbers are integers, and their signs are determined by the relations $E_{4n} > 0$ and $E_{4n+2} < 0$. Growth is rapid, leaving no room here for our usual digit grouping. The last number shown, E_{54}, has 61 digits. E_{100} has 139 digits, E_{200} has 336 digits, and E_{500} has 1037 digits.

2n	E_{2n}	**2n**	E_{2n}
0	1	28	1252259641403629865468285
2	−1	30	−441543893249023104553682821
4	5	32	177519391579539289436664789665
6	−61	34	−80723299235887898062168247453281
8	1385	36	41222060339517702122347079671259045
10	−50521	38	−23489580527043108252017828576198947741
12	2702765	40	14851150718114980017877156781405826684425
14	−199360981	42	−10364622733519612119397957304745185976310201
16	19391512145	44	7947579422597592703608040510088070619519273805
18	−2404879675441	46	−6667537516685544977435028474773748197524107684661
20	370371188237525	48	6096278645568542158691685742876843153976539044435185
22	−69348874393137901	50	−6053285248188621896314383785111649088103498225146815121
24	15514534163557086905	52	6506162486684608847715870634080822983483644236765385576565
26	−4087072509293123892361	54	−7546659939008739098061432565889736744212240024711699858645581

and then record their values as floating-point numbers in a compile-time constant table of modest size. For IEEE 754 binary arithmetic, the table requires only 33 entries in the 32-bit format, 130 in the 64-bit format, and 1157 in the 80-bit and 128-bit formats. The wider exponent range of the IEEE 754 decimal formats requires a few more: 58, 153, and 1389, respectively.

With a bit of extra software logic, there is no need to store the odd-indexed zero numbers, so in the mathcw library, we hide the table access in a function family, `BERNUM(n)`. The functions return a NaN for negative indexes, a table entry for representable values, and otherwise, $+\infty$. As usual, the numbers are stored in both hexadecimal and decimal formats to make correctly rounded values more likely.

The files `bernumf.c` and `bernum.c` in the subdirectory `exp` support our claim of numerical instability. They implement Bernoulli number evaluation with a compile-time choice of the recurrence relation or the direct sum in the `float` and `double` data types:

- Tests of the recurrence show errors up to several hundred ulps already at B_{10}, and complete loss of all significant digits at B_{22} and B_{50} in the two formats.
- Tests show that the direct sum is much better. Nevertheless, losses up to a thousand ulps are reported just before subtraction of Infinity produces NaN values.

There is a related set of numbers bearing the name of another Swiss mathematician from Basel, Leonhard Euler, who is viewed by many as the greatest mathematician of all time.

The *Euler numbers*,[6] E_n, were first published by Euler in 1755. They are defined by the expansion of the hyperbolic secant function:

$$\operatorname{sech}(z) = \frac{1}{\cosh(z)} = \frac{2}{\exp(z) + \exp(-z)} = \sum_{n=0}^{\infty} E_n \frac{z^n}{n!}, \qquad \text{for } |z| < \tfrac{1}{2}\pi.$$

The Euler numbers can also be defined by the expansion of the trigonometric secant function:

$$\sec(z) = \frac{1}{\cos(z)} = \sum_{n=0}^{\infty} (-1)^n E_{2n} \frac{z^{2n}}{(2n)!}, \qquad \text{for } |z| < \tfrac{1}{2}\pi.$$

The first 28 nonzero Euler numbers are shown in **Table 18.5**.

As with the Bernoulli numbers, we can illustrate the series with Maple, and verify the coefficient relations to the Euler numbers (with output reformatted into three columns):

[6] See entry *A000364* at `http://oeis.org/` or [SP95, sequence M1492], and the *MathWorld* entry at `http://mathworld.wolfram.com/EulerNumber.html` or in the print edition [Wei09].

```
> taylor(sech(x), x = 0, 12);
         2         4   61   6   277   8    50521   10      12
1 - 1/2 x  + 5/24 x  - --- x  + ---- x  - ------- x   + O(x  )
                       720      8064      3628800

> for n from 0 to 10 do printf("%2d  %a\n", n, euler(n)/n!) end do:
0  1                    4  5/24               8  277/8064
1  0                    5  0                  9  0
2  -1/2                 6  -61/720           10  -50521/3628800
3  0                    7  0
```

Mathematica supplies Euler numbers with `EulerE[n]`, and Maxima and REDUCE with `euler(n)`.

The `mathcw` library provides the Euler numbers with the function family `EULNUM(x)`, and the code in the file `eulnx.h` is much like that in `bernx.h`, with checks for an out-of-range index, producing either a NaN or $+\infty$, or else returning the result of a fast table lookup.

Like the Bernoulli numbers, the Euler numbers have a complicated recurrence relation that requires all earlier numbers, and exact integer arithmetic, to evaluate reliably:

$$E_{2n} = -\left[1 + \binom{2n}{2} E_2 + \binom{2n}{4} E_4 + \cdots + \binom{2n}{2n-2} E_{2n-2}\right].$$

The recurrence relation can be written more compactly like this:

$$\sum_{k=0}^{n} \binom{2n}{2k} E_{2k} = 0, \qquad E_{2n+1} = 0, \qquad \text{for } n = 0, 1, 2, \ldots.$$

The files `eulnumf.c` and `eulnum.c` in the subdirectory `exp` implement computation of the Euler numbers by the recurrence relation. Their tests demonstrate losses up to seven ulps before subtraction of Infinity produces NaN values. That is *much* better than the companion tests of the Bernoulli numbers, and the reason for the improvement is explained shortly when we look more at the growth of the Euler numbers. The conclusion is that run-time generation is marginally acceptable for Euler numbers, but hopeless for Bernoulli numbers, at least without access to higher-precision arithmetic, or the better algorithm given shortly in **Section 18.5.1**.

Although we have no computational use for them in this section, there are complicated relations between the Bernoulli and Euler numbers:

$$\begin{aligned}
E_{2n} &= 1 - \sum_{k=1}^{n} \binom{2n}{2k-1} \frac{2^{2k}(2^{2k}-1)}{2k} B_{2k}. \\
B_n &= \sum_{k=0}^{n-1} \frac{n}{4^n - 2^n} E_k, && \text{for } n = 2, 4, 6, \ldots, \\
E_n &= \sum_{k=1}^{n} \frac{2^k - 4^k}{k} B_k, && \text{for } n = 2, 4, 6, \ldots, \\
\pi &\asymp 2(2^{2n} - 4^{2n}) \frac{B_{2n}}{E_{2n}}, && \text{for large } n.
\end{aligned}$$

The last relation shows that the Euler numbers have larger magnitudes than those of the Bernoulli numbers of the same index. Numerical experiments show that the asymptotic relation holds for $n > 1$, and the two sides agree to eight decimal digits already at $n = 8$.

The relations can be written compactly another way:

$$\begin{aligned}
E_{n-1} &= \frac{(4B-1)^n - (4B-3)^n}{2n}, \\
E_{2n} &= \frac{4^{2n+1}}{2n+1} (B - \tfrac{1}{4})^{2n+1}.
\end{aligned}$$

The symbol B here is interpreted to mean that each power B^k in the expansion is replaced by Bernoulli number B_k.

Finally, if we combine the estimate of B_{2n} with the relationship between the Bernoulli and Euler numbers, and discard smaller terms, we predict that the common quotient of an Euler number and a factorial can be estimated by this relation:

$$\frac{E_{2n}}{(2n)!} \approx (-1)^n 2 \left(\frac{2}{\pi}\right)^{2n+1}, \qquad \textit{Euler/factorial quotient estimate.}$$

Here is a test of how good our estimate is, and how fast the quotients drop:

```
hoc64> u_last = 1
hoc64> for (n = 1; n < 18; ++n)                                      \
hoc64> {
hoc64>     u = eulnum(2*n) / factorial(2*n);
hoc64>     v = (-1)**n * 2 * (2 / PI) ** (2*n + 1)
hoc64>     printf("%2d  % 6.2f  % 17.9..3e  % 17.9..3e  % 10.3e\n", 2*n, u_last / u, u, v, (u - v) / u)
hoc64>     u_last = u
hoc64> }
 2   -2.00  -5.000_000_000e-01  -5.160_245_509e-01  -3.205e-02
 4   -2.40   2.083_333_333e-01   2.091_368_732e-01  -3.857e-03
 6   -2.46  -8.472_222_222e-02  -8.475_998_213e-02  -4.457e-04
 8   -2.47   3.435_019_841e-02   3.435_192_686e-02  -5.032e-05
10   -2.47  -1.392_223_325e-02  -1.392_231_156e-02  -5.625e-06
12   -2.47   5.642_496_810e-03   5.642_500_345e-03  -6.264e-07
14   -2.47  -2.286_819_095e-03  -2.286_819_254e-03  -6.966e-08
16   -2.47   9.268_129_274e-04   9.268_129_346e-04  -7.742e-09
18   -2.47  -3.756_231_339e-04  -3.756_231_342e-04  -8.603e-10
20   -2.47   1.522_343_222e-04   1.522_343_222e-04  -9.560e-11
22   -2.47  -6.169_824_688e-05  -6.169_824_688e-05  -1.062e-11
24   -2.47   2.500_535_761e-05   2.500_535_761e-05  -1.182e-12
26   -2.47  -1.013_428_972e-05  -1.013_428_972e-05  -1.329e-13
28   -2.47   4.107_272_920e-06   4.107_272_920e-06  -1.609e-14
30   -2.47  -1.664_615_015e-06  -1.664_615_015e-06  -3.689e-15
32   -2.47   6.746_430_546e-07   6.746_430_546e-07  -2.197e-15
34   -2.47  -2.734_225_313e-07  -2.734_225_313e-07  -1.936e-15
```

The estimate is correct to five decimal digits for $E_{10}/10!$, and successive quotients fall off by about a factor of 2.5, much slower than the Bernoulli–factorial quotients.

An estimate of the quotient of successive Euler numbers is now readily obtained:

$$\frac{E_{2n}}{E_{2n-2}} \approx -2n(2n-1)\left(\frac{2}{\pi}\right)^2 = -16\left(\frac{2n(2n-1)}{(2\pi)^2}\right).$$

The quotient on the right has the same form as that in the Bernoulli-number quotient estimate, but there is an additional factor of 16. That factor gives additional growth to the Euler-number quotient, and explains the better numerical stability of the calculation of Euler numbers by their recurrence relation.

Armed with the mathematical relations and numerical experiments of this section, when we meet a sum containing Bernoulli or Euler numbers, we can now roughly estimate its coefficients, and thus predict how rapidly the sum converges, without needing to use those big numbers directly.

18.5.1 Bernoulli numbers revisited

An alternative approach to computation of the Bernoulli numbers exploits their relation to the coefficients of the expansion of the tangent function [KB67], [OLBC10, §24.15], [BZ11, pp. 156–157]:

$$\tan(z) = \sum_{n=0}^{\infty} T_n\, z^n/n!, \qquad \textit{Taylor-series expansion for } |z| < \tfrac{1}{2}\pi,$$

$$T_{2n} = 0, \qquad \textit{even-order coefficients are zero,}$$

$$B_{2n} = (-1)^{n-1}\, 2n\, T_{2n-1}/(4^{2n} - 2^{2n}), \qquad \textit{for } n = 1, 2, 3, \ldots.$$

Table 18.6: Tangent numbers of odd order, T_{2n+1}. Those of even order are all zero. All tangent numbers are positive integers, and grow rapidly, so we suppress digit grouping. The last number shown, T_{51}, has 57 digits. T_{101} has 141 digits, T_{201} has 338 digits, and T_{501} has 1039 digits. Growth is similar to that of the Euler numbers, and soon exceeds the range of hardware floating-point arithmetic.

$2n+1$	T_{2n+1}	$2n+1$	T_{2n+1}
1	1	27	70251601603943959887872
3	2	29	23119184187809597841473536
5	16	31	8713962757125169296170811392
7	272	33	3729407703720529571097509625856
9	7936	35	1798651693450888780071750349094912
11	353792	37	970982810785059112379399707952152576
13	22368256	39	583203324917310043943191641625494290432
15	1903757312	41	387635983772083031828014624002175135645696
17	209865342976	43	283727921907431909304183316295787837183229952
19	29088885112832	45	227681379129930886488600284336316164603920777216
21	4951498053124096	47	199500252157859031027160499643195658166340757225472
23	1015423886506852352	49	190169564657928428175235445073924928592047775873499136
25	246921480190207983616	51	196535694915671808914892880726989984967498805398829268992

The odd-order *tangent numbers*, T_{2n+1}, grow rapidly, as shown in **Table 18.6**.

The essential point is that the tangent numbers can be generated from a stable recurrence where all terms are *positive*:

$$T_{2j-1} = (j-1)!, \qquad \textit{initialization for } j = 1, 2, \ldots, n,$$

$$T_{2j-1} = \sum_{j=k}^{n} \big((j-k+2)T_{2j-1} + (j-k)T_{2j-3} \big), \qquad \textit{for } k = 2, 3, \ldots, n \textit{ and } j = k, \ldots, n.$$

Recovery of a Bernoulli number B_{2n} from T_{2n-1} is then a simple scaling. Test programs `bernum2*.c` show that the average error in Bernoulli numbers computed from tangent numbers is below 1.7 ulps, with worst-case errors below 6.9 ulps. Because the relation involves a division of two large numbers, premature overflow is a problem, but it can be minimized with exact scaling (by the smallest normal number) of the computed tangent numbers. That scaling roughly doubles the largest subscript for which Bernoulli numbers can be computed from tangent numbers. A second set of test programs, `bernum3*.c` implements that scaling.

18.6 An Italian excursion: Fibonacci numbers

In 1202, Leonardo Pisano (Leonardo of Pisa), also called Leonardo Fibonacci (from Filius Bonaccii, son of Bonaccio), published an important mathematics book, *Liber Abaci* (Book of Calculation) that introduced the Hindu numerals 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 to Europe, and the Latinized Arabic word, *zephirum*, which became *zefiro* in Italian, and in the Venetian dialect, *zero*, the name by which we know it in English, and several other European languages, today.

Fibonacci's book also brought to Europe the notion of an *algorithm*, a name that arises from that of the Ninth Century Persian scholar Muhammed al-Khwarizmi, and the subject of *algebra*, a name that comes from the title of al-Khwarizmi's book, *Hisab Al-Jabr wal Mugabalah* (Book of Calculations, Restoration and Reduction). Fibonacci's book was written in Latin, and long remained unavailable in English, until the 800th anniversary publication of the late Laurence Sigler's *Fibonacci's Liber Abaci: A Translation into Modern English of Leonardo Pisano's Book of Calculation* [Sig02].

Liber Abaci is a large collection of arithmetic problems, expressed in words, rather than the symbols common in modern notation. One of them poses, and solves, this problem:

> *How many pairs of rabbits can be produced in a year from a single pair if each pair produces a new pair every month, each new pair reproduces starting at the age of one month, and rabbits never die?*

Figure 18.13: Pascal's Triangle and Fibonacci numbers. Each number in the centered triangular array is either one (at the edges), or the sum of the two adjacent numbers in the previous row. The numbers in each row are the coefficients of the binomial expansion of that row order. For example, $(a+b)^4 = a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4$.
To reveal the hidden Fibonacci numbers, realign the normal centered triangular array to a right-triangular display, and then sum the upward diagonals to obtain the sequence $1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$, as indicated by the bold bracketed numbers.

								1								
							1		1							
						1		2		1						
					1		3		3		1					
				1		4		6		4		1				
			1		5		10		10		5		1			
		1		6		15		20		15		6		1		
	1		7		21		35		35		21		7		1	
1		8		28		56		70		56		27		7		1

$\longrightarrow$

	[1]							
1	**[1]**	**[2]**						
1	1	**[3]**	**[5]**					
1	2	1	**[8]**	**[13]**				
1	3	3	1	**[21]**	**[34]**			
1	4	6	4	1				
1	5	10	10	5	1			
1	6	15	20	15	6	1		
1	7	21	35	35	21	7	1	
1	8	28	56	70	56	28	8	1

The solution is a sequence with terms $1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$, and it can be written compactly as this simple recurrence:

$$F_0 = 0, \qquad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2}, \qquad \text{for } n = 2, 3, 4, \ldots.$$

The total number of rabbit pairs after the births at the start of month n is F_{n+1}, so the answer to Fibonacci's problem is the count a year later, at the start of month thirteen: $F_{14} = 377$.

That famous sequence is known as the *Fibonacci sequence*,[7] and despite its simple origin in an unrealistic problem (because rabbits eventually die), in the intervening centuries, it has been encountered in an astonishing variety of fields of study, from growth patterns of galaxies, marine shells, and plants, to random-number generation, to computer-network optimization. As shown in **Figure 18.13**, the Fibonacci numbers are also the diagonal sums of the numbers in *Pascal's Triangle*, which displays the coefficients of terms in the expansion of $(a+b)^n$ in rows indexed by n.

So much has been learned about the Fibonacci recurrence, and new material continues to be discovered, that a mathematical journal, *The Fibonacci Quarterly*,[8] founded in 1963, is entirely devoted to the study of Fibonacci numbers and sequences. A search of the *MathSciNet* and *zbMATH* databases turns up more than 2000 and 5300 research articles with Fibonacci in their titles. Web searches find more than a million Web pages with mention of Fibonacci numbers.

The Fibonacci recurrence can be run backward as well, so that the indexes $k = -1, -2, \ldots, -10$ produce the sequence $1, -1, 2, -3, 5, -8, 13, -21, 34, -55$. Apart from signs, those are the same numbers as in the forward direction, so we have an important symmetry relation for negative indexes:

$$F_{-k} = (-1)^{k+1} F_k.$$

The Fibonacci numbers appear as the coefficients in this series expansion [SP95, page 10]:

$$\frac{1}{1 - x - x^2} = \sum_{k=0}^{\infty} F_k x^k.$$

There are numerous relations among the Fibonacci numbers, including these:

$$F_{n+2} = 1 + \sum_{k=1}^{n} F_k, \qquad F_n F_{n+1} = \sum_{k=1}^{n} F_k^2,$$

[7] See the *MathWorld* article at `http://mathworld.wolfram.com/FibonacciNumber.html` or in the print edition [Wei09].

[8] See the journal Web site at `http://www.fq.math.ca/` and its bibliography at `http://www.math.utah.edu/pub/tex/bib/index-table-f.html#fibquart`.

Table 18.7: The first 100 Fibonacci numbers. Each member of the sequence is the sum of the two preceding members.

n	F_n	n	F_n	n	F_n	n	F_n
0	0	25	75 025	50	12 586 269 025	75	2 111 485 077 978 050
1	1	26	121 393	51	20 365 011 074	76	3 416 454 622 906 707
2	1	27	196 418	52	32 951 280 099	77	5 527 939 700 884 757
3	2	28	317 811	53	53 316 291 173	78	8 944 394 323 791 464
4	3	29	514 229	54	86 267 571 272	79	14 472 334 024 676 221
5	5	30	832 040	55	139 583 862 445	80	23 416 728 348 467 685
6	8	31	1 346 269	56	225 851 433 717	81	37 889 062 373 143 906
7	13	32	2 178 309	57	365 435 296 162	82	61 305 790 721 611 591
8	21	33	3 524 578	58	591 286 729 879	83	99 194 853 094 755 497
9	34	34	5 702 887	59	956 722 026 041	84	160 500 643 816 367 088
10	55	35	9 227 465	60	1 548 008 755 920	85	259 695 496 911 122 585
11	89	36	14 930 352	61	2 504 730 781 961	86	420 196 140 727 489 673
12	144	37	24 157 817	62	4 052 739 537 881	87	679 891 637 638 612 258
13	233	38	39 088 169	63	6 557 470 319 842	88	1 100 087 778 366 101 931
14	377	39	63 245 986	64	10 610 209 857 723	89	1 779 979 416 004 714 189
15	610	40	102 334 155	65	17 167 680 177 565	90	2 880 067 194 370 816 120
16	987	41	165 580 141	66	27 777 890 035 288	91	4 660 046 610 375 530 309
17	1 597	42	267 914 296	67	44 945 570 212 853	92	7 540 113 804 746 346 429
18	2 584	43	433 494 437	68	72 723 460 248 141	93	12 200 160 415 121 876 738
19	4 181	44	701 408 733	69	117 669 030 460 994	94	19 740 274 219 868 223 167
20	6 765	45	1 134 903 170	70	190 392 490 709 135	95	31 940 434 634 990 099 905
21	10 946	46	1 836 311 903	71	308 061 521 170 129	96	51 680 708 854 858 323 072
22	17 711	47	2 971 215 073	72	498 454 011 879 264	97	83 621 143 489 848 422 977
23	28 657	48	4 807 526 976	73	806 515 533 049 393	98	135 301 852 344 706 746 049
24	46 368	49	7 778 742 049	74	1 304 969 544 928 657	99	218 922 995 834 555 169 026

$$\begin{aligned}
F_{2n} &= F_{n+1}^2 - F_{n-1}^2, & F_{3n} &= F_{n+1}^3 + F_n^3 + F_{n-1}^3, \\
F_{2n} &= \sum_{k=0}^{n} \binom{n}{k} F_k, & F_{3n} &= \sum_{k=0}^{n} \binom{n}{k} 2^k F_k, \\
F_{2n+1} &= 1 + \sum_{k=1}^{n} F_{2k}, & (-1)^n &= F_{n-1}F_{n+1} - F_n^2, \\
F_{n+1}^2 &= 4F_nF_{n-1} + F_{n-2}^2, & F_{n+m} &= F_{n-1}F_m + F_nF_{m+1}.
\end{aligned}$$

The last of those relations can be used recursively to reduce a large index to smaller ones for which F_k values are known. We show two other ways to find Fibonacci numbers of large index later in this section.

Like the Bernoulli and Euler numbers, the Fibonacci numbers grow quickly, although not as rapidly as B_{2n} and E_{2n}. The first hundred of them are shown in Table 18.7.

Fibonacci number growth is exponential, with the curious result, discovered by the Scottish mathematician Robert Simson in 1753, that

$$\lim_{n\to\infty} F_n = \phi F_{n-1}.$$

Here, the value ϕ is a famous mathematical constant known since antiquity, the *golden ratio*:

$$\begin{aligned}
\phi &= \tfrac{1}{2}(\sqrt{5}+1) \\
&\approx 1.618\,033\,988\,749\,894\,848\,204\,586\,834\,365\,638\,117\,720\,309\,179\,805\,\ldots.
\end{aligned}$$

The golden ratio is the last of the *big five* mathematical constants — e, i, π, γ, and ϕ — and its story is told in two recent books [HF98, Liv02].

There is a formula that relates successive Fibonacci numbers, and is valid for all of them:

$$F_{n+1} = \lfloor \phi F_n + \tfrac{1}{2} \rfloor, \qquad \text{for } n = 0, 1, 2, \ldots.$$

If the Fibonacci sequence is calculated in signed integer arithmetic, then the overflow limit is reached at F_{46} with 32-bit integers. After F_{78}, 53-bit integers overflow. After F_{92} (fewer than 8 years in Fibonacci's rabbit problem), 64-bit integers fail. After F_{184}, 128-bit integers are insufficient.

Floating-point arithmetic of fixed precision lacks sufficient digits to handle large Fibonacci numbers exactly, but it does handle the growth somewhat better. Nevertheless, the overflow limit of the 128-bit IEEE 754 binary format is reached after 23 597 terms, for which the Fibonacci number requires 4932 decimal digits, of which only the first 36 are representable in that format.

The simple recurrence relation of the Fibonacci numbers means that there is no need to have a large compile-time constant table. Instead, we can compute them as needed, and then store them in a private internal table for fast access on later calls. If we restrict table entries to exactly representable integer values, then the table size is modest.

The mathcw function family `FIBNUM(n)` is implemented in file `fibnx.h`. The computation is simple, and this author has a Web site[9] that does the job, with specified additional requirements, in about 50 programming languages as an amusing introductory programming problem.

Once we pass the exact whole-number overflow limit, we have to deal with rounding errors. The computation is numerically stable because all terms are positive, but there is no need to keep adding number pairs, because we can estimate large Fibonacci numbers to near machine accuracy from the relation

$$F_n \approx \phi^{n-\mathrm{last}} F_{\mathrm{last}},$$

by scaling the last-stored *exact* value at the cost of one exponentiation. Because ϕ is an irrational value, it cannot be represented exactly. The error-magnification factor of the power function is its exponent argument (see **Table 4.1** on page 62), so we compute the power in the next higher precision to reduce error growth. That is likely to be acceptable, because many applications of Fibonacci numbers require only a few members of the sequence, and they can be obtained by exact table lookup.

There is another way to compute Fibonacci numbers of large index directly:

$$F_n = \mathrm{round}\left(\frac{\phi^n}{\sqrt{5}}\right), \qquad \text{for } n = 0, 1, 2, \dots .$$

That relation is *exact* if the right-hand side can be computed with enough fractional digits to make the rounding decision certain.

A variant of that explicit relation looks like this:

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right) \qquad \text{for } n = 0, 1, 2, \dots .$$

We can make a simple test of the recurrence relation in floating-point arithmetic past the exactly representable number limit:

```
hoc64> load("libmcw")

hoc64> for (k = 2; k < 200; ++k)                        \
hoc64>    if ((fibnum(k-2) + fibnum(k-1)) != fibnum(k))       \
hoc64>        printf("%d ", k);
82 83 85 89 95 98 101 105 108 109 111 117 118 120 121 124 135 141 144
151 156 158 161 171 175 176 187 199
```

With the IEEE 754 64-bit `double` type, only Fibonacci numbers up to F_{78} are exactly representable in the 53-bit significand. Four elements later, the recurrence relation fails because of rounding errors.

To further investigate the size of the rounding errors, we check the computed Fibonacci numbers against values produced in the next higher-precision format, rounded to working precision, and reformatted into three columns:

```
hoc64> load("libmcw-hp")
```

[9] See *Fun with Fibonacci* at `http://www.math.utah.edu/~beebe/software/java/fibonacci/`. That site includes the complete Latin text of the original Fibonacci rabbit problem, with an English translation, and more information about Sigler's book.

```
hoc64> for (k = 2; k <= 1000; ++k) \
hoc64> {
hoc64>     u = ulps(hp_fibnum(k), fibnum(k))
hoc64>     if (u > 1) printf("%4d %.3f ulps\n", k, u)
hoc64> }
  ...                  ...                  ...
  101 1.030 ulps       500 3.084 ulps       998 6.188 ulps
  102 1.273 ulps       501 2.542 ulps       999 6.119 ulps
  103 1.574 ulps       502 3.142 ulps      1000 3.782 ulps
  ...                  ...
```

Evidently, the rounding errors are initially small, but grow to a few ulps at $F_{1000} \approx 4.346 \times 10^{208}$.

18.7 A German gem: the Riemann zeta function

With only a few dozen letters available in the Latin and Greek alphabets, mathematicians often reuse the same letter for different purposes. The zeta function discussed in this section is that of the Nineteenth Century German mathematician Bernhard Riemann [Der03, Lau08, Sab03], so his name is attached to it to distinguish it from other people's zeta functions. For brevity, in the rest of this section, we usually omit the qualifier.

The zeta function has a simple definition:

$$\zeta(z) = \sum_{k=1}^{\infty} k^{-z}, \qquad \textit{provided } \operatorname{real}(z) > 1.$$

The special case $z = 1$ reduces to the divergent sum of the harmonic series, but elsewhere on the complex plane, the function is finite, provided that the reflection rule given shortly is used for real arguments $x < 1$.

Euler began his studies of infinite sums of reciprocal integer powers, $\zeta(n)$, about 1730 [Ayo74]. In 1737, he extended the function to real arguments, but it was Riemann's generalization to complex arguments in 1859 that revealed the mathematical treasures hidden in the zeta function. Today, we consider the zeta function to be defined in the entire complex plane, minus the point $z = 1$. In most of this section, we restrict ourselves to arguments on the real axis, and henceforth use real x instead of complex z. Before we do so, however, we make this observation:

> *Riemann conjectured that the nontrivial zeros (those off the real axis) of the complex zeta function are all of the form $z = \frac{1}{2} + yi$, that is, they lie on the line $x = \frac{1}{2}$ in the complex plane [Der03, Lap08, Lau08, O'S07, Pat88, Roc06, Sab03].*
>
> *The Riemann Hypothesis is the eighth of a famous set of 23 problems [Gra00] announced in 1900 by the influential German mathematician David Hilbert.*[10] *A century later, it heads the list of famous unsolved problems in The Clay Mathematics Institute Millennium Prize challenge [CJW06, Clay09], despite more than 150 years of study by many of the world's leading mathematicians.*
>
> *The famous English mathematician G. H. Hardy proved in 1914 that there are an infinite number of zeros on that line, others later proved that there are an infinite number of zeros with x in $(0, 1)$, and more recently, it has been proved that at least two-fifths of the zeros lie on that line. However, none of those results compels* all *of the zeros to have $x = \frac{1}{2}$.*
> *By 2004, numerical studies have computed more than 10^{13} zeros without finding deviations from Riemann's hypothesis.*[11]
> *Many mathematicians believe the hypothesis is a theorem (that is, must be a fact, rather than a conjecture), and a lot of recent mathematical research assumes the truth of the Riemann Hypothesis to make further progress in other areas.*

Real arguments of the zeta function on $(-\infty, 1)$ are mapped to the positive axis with either of these formulas found by Euler in 1747:

$$\zeta(z) = 2^z \pi^{z-1} \sin(\tfrac{1}{2}\pi z)\Gamma(1-z)\zeta(1-z), \qquad \textit{reflection rule,}$$

$$\zeta(1-z) = 2(2\pi)^{-z} \cos(\tfrac{1}{2}\pi z)\Gamma(z)\zeta(z), \qquad \textit{reflection rule.}$$

The computation for negative arguments is expensive because of the need for at least four transcendental function evaluations. In addition, the argument reduction in the trigonometric functions needs careful handling to avoid serious accuracy loss, as we saw in **Section 18.1.3** on page 531.

[10] See also `http://en.wikipedia.org/wiki/Hilbert's_problems`.

[11] See Eric Weisstein's discussion and table of historical records at `http://mathworld.wolfram.com/RiemannZetaFunctionZeros.html` or in the print edition [Wei09], and the online article at `http://en.wikipedia.org/wiki/Riemann_hypothesis`. Web searches easily find graphical animations of the behavior of the zeta function along the critical line; see, for example, `http://web.viu.ca/pughg/RiemannZeta/RiemannZetaLong.html`.

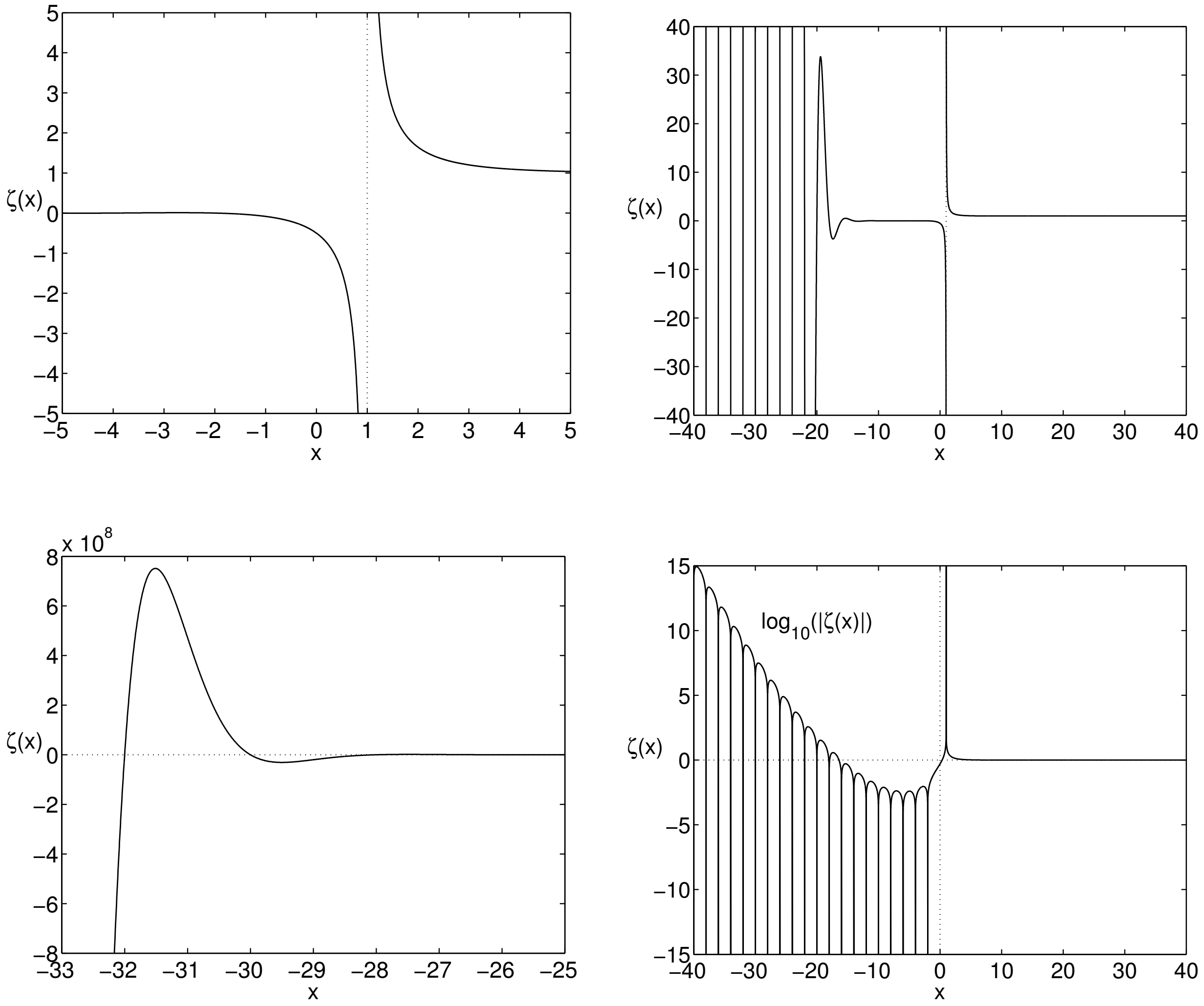


Figure 18.14: The real Riemann zeta function on four different scales. There is a single pole at $x = 1$, indicated by a vertical dotted line. The extremes on the negative axis in the upper-right graph are not poles, but rather, increasingly sharp oscillations around negative even integers. The bottom-left plot shows detail in that region, emphasizing the rapid growth as the argument becomes more negative. Notice the extended vertical range in the bottom graphs, where the horizontal dotted axis line has been added to make the rise and fall more visible. To compress the peaks on the negative axis, the lower-right plot shows a logarithm of the absolute value of the zeta function.

The Riemann zeta function is graphed in **Figure 18.14**. It is instructive to investigate its behavior on the negative axis by examining function values near even integers with a spacing equal to the machine epsilon of the IEEE 754 32-bit decimal format:

```
% maple
> Digits := 75:
> epsilon := 1.0e-6:
> for x from -20 to -50 by -2 do
>     printf("  %3d  %+28.6f  %d  %+28.6f\n", x, Zeta(x - epsilon), Zeta(x), Zeta(x + epsilon))
> end do:
  -20                     -0.000132  0                      +0.000132
```

```
-22                     +0.001548  0                     -0.001548
-24                     -0.021645  0                     +0.021645
-26                     +0.356380  0                     -0.356379
-28                     -6.824568  0                     +6.824547
-30                   +150.395456  0                   -150.394981
-32                  -3779.085152  0                  +3779.072731
-34                +107403.843683  0                -107403.477841
-36               -3427919.832550  0               +3427907.769952
-38             +122083301.056668  0             -122082858.429988
-40            -4824153825.043452  0            +4824135845.935097
-42           +210423664149.341611  0           -210422859638.052708
-44         -10084537705316.655384  0         +10084498221743.078686
-46        +528769774333454.877181  0        -528767657572332.113664
-48      -30216628059523083.809344  0      +30216504552061415.757252
-50   +1875220594685297466.063292  0   -1875212778359822930.953882
```

The function is steep near negative even integers, but there are no poles on the negative axis.

There are important special cases of the zeta function:

$$\zeta(0) = -\tfrac{1}{2}, \qquad \lim_{\delta \to 0} \zeta(1-\delta) = -\infty, \qquad \lim_{\delta \to 0} \zeta(1+\delta) = +\infty.$$

The Taylor-series expansions near 0 and 1 require some auxiliary constants, γ_k, defined like this:

$$\gamma_k = \lim_{m \to \infty} \Big(\sum_{k=1}^{m} \frac{\log(k)^n}{k} - \frac{\log(m)^{n+1}}{n+1} \Big).$$

The series expansions then look symbolically and numerically like this:

$$\begin{aligned}
\zeta(x) &= -\tfrac{1}{2} + \big(-\tfrac{1}{2}\log(2\pi)\big)\,x \\
&\qquad + \big(-\tfrac{1}{4}\log(2\pi)^2 - \tfrac{1}{48}\pi^2 + \tfrac{1}{4}\gamma^2 + \tfrac{1}{2}\gamma_1\big)\,x^2 + \cdots, \\
&\approx -\tfrac{1}{2} - 0.918\,939\,x - 1.003\,178\,x^2 - 1.000\,785\,x^3 - \\
&\quad 0.999\,879\,x^4 - 1.000\,002\,x^5 + \cdots, \\
\zeta(1+t) &= \frac{1}{t}(1 + \gamma t - \gamma_1 t^2 + \tfrac{1}{2}\gamma_2 t^3 - \tfrac{1}{6}\gamma_3 t^4 + \tfrac{1}{24}\gamma_4 t^5 - \cdots), \\
&\approx \frac{1}{t}(1 + 0.577\,216\,t + 0.072\,815\,8\,t^2 - 0.004\,845\,18\,t^3 - \\
&\qquad 0.000\,342\,306\,t^4 + 0.000\,096\,890\,5\,t^5 - \cdots).
\end{aligned}$$

Because of their computational complexity, the γ_k constants, and the general expansion coefficients, are best computed in high-precision arithmetic with a symbolic-algebra system, and then tabulated in other software as compile-time constants.

The behavior near the pole at $x = 1$ is driven mainly by the $1/t$ factor, so we expect to be able to compute the zeta function accurately near that pole.

For x in $[0, 1)$, the reflection formulas move points near the pole at $x = 1$ to the nicer region near $x = 0$ where the convergent series expansion for $\zeta(x)$ can be applied. For small negative x, the reflection formulas move the computation onto the region near $1 + |x|$ where we can use the $\zeta(1+t)$ expansion with $t = |x|$. For more negative x values, the original Riemann sum can be used to find the value of $\zeta(1 + |x|)$ in the reflection rule.

At positive nonzero even integer arguments, zeta-function values are powers of π scaled by the Bernoulli numbers discussed in **Section 18.5** on page 568:

$$\zeta(2) = \pi^2/6, \qquad \zeta(4) = \pi^2/90, \qquad \zeta(6) = \pi^6/945,$$

$$\zeta(8) = \pi^8/9\,450, \qquad \zeta(10) = \pi^{10}/93\,555, \qquad \zeta(2n) = (-1)^{n-1}\frac{(2\pi)^{2n}}{2(2n)!}B_{2n}.$$

The general relation for *even* integer arguments, $\zeta(2n)$ (for $n = 1, 2, 3, \dots$), was discovered by Euler in 1740. For large n, direct summation of the zeta-function definition provides a fast way to find the Bernoulli numbers B_{2n}.

At positive *odd* integer arguments, the zeta-function values do not have simple closed forms.

Despite the rapidly growing numerator and denominator in $\zeta(2n)$, their increase must be similar, because the original definition of the zeta function tells us that its limiting behavior for large x *must* look like this:

$$\lim_{x\to\infty} \zeta(x) \to 1 + 2^{-x} \to 1.$$

For that reason, it is useful to have a separate function that computes the difference $\zeta(x) - 1$ directly, as two of the libraries mentioned in **Section 18.7.1** on the next page do, so that $\zeta(x)$ can be determined to machine accuracy by adding the tiny value $\zeta(x) - 1$ to 1.

On the negative axis, there are simple values classed by even and odd indexes, with just one out-of-class value at the origin. Here are the first dozen:

$$\begin{aligned} \zeta(0) &= -1/2, & \zeta(-1) &= -1/12, \\ \zeta(-2) &= 0, & \zeta(-3) &= 1/120, \\ \zeta(-4) &= 0, & \zeta(-5) &= -1/252, \\ \zeta(-6) &= 0, & \zeta(-7) &= 1/240, \\ \zeta(-8) &= 0, & \zeta(-9) &= -1/132, \\ \zeta(-10) &= 0, & \zeta(-11) &= 691/32\,760. \end{aligned}$$

The general form of the zeta function at negative integers is either zero, or a simple scaling of a Bernoulli number:

$$\zeta(-2n) = 0, \qquad \zeta(-2n+1) = -\frac{B_{2n}}{2n}.$$

Although that looks simpler than the corresponding relation for B_{2n} on the positive axis, it is computationally worse for finding B_{2n} because of the greater difficulty of computing the zeta function of negative arguments.

Those results, and the known alternating signs of the Bernoulli numbers, explain the oscillatory behavior seen in the rightmost graphs in **Figure 18.14** on page 580. On the negative axis, the function values soon approach the overflow limits, but there are no poles there. In IEEE 754 32-bit binary arithmetic, $\zeta(x)$ overflows for $x \approx -66.053$. In the 64-bit format, overflow happens for $x \approx -260.17$, and in the 80-bit and 128-bit formats, for $x \approx -1754.55$. That last limit represents premature overflow, and would be somewhat more negative if the computation could be done with a wider exponent range.

In some applications, zeta functions appear only with integer arguments, and it is worthwhile to consider treating them separately, because we can handle them more quickly, and more accurately, than the zeta function of general real arguments.

Here is an example of the use of the zeta function for speeding the convergence of an infinite sum [AW05, page 385], [AWH13, page 17]. Consider the computation of the slowly converging sum:

$$\begin{aligned} s &= \sum_{n=1}^{\infty} \frac{1}{1+n^2} \\ &\approx 0.5000 + 0.2000 + 0.1000 + 0.0588 + 0.0385 + 0.0270 + \\ &\quad\; 0.0200 + 0.0154 + 0.0122 + 0.0099 + 0.0082 + 0.0069 + \cdots. \end{aligned}$$

Expand one term of the sum as a series in powers of $1/n^2$:

$$\begin{aligned} \frac{1}{1+n^2} &= \frac{1}{n^2}\left(\frac{1}{1+n^{-2}}\right) \\ &= \frac{1}{n^2}(1 - n^{-2} + n^{-4} - n^{-6} + n^{-8} - \cdots) \\ &= n^{-2} - n^{-4} + n^{-6} - n^{-8} + n^{-10} - \cdots \\ &= n^{-2} - n^{-4} + n^{-6} - n^{-8} + n^{-10} - \frac{1}{n^{12} + n^{10}}. \end{aligned}$$

Substituting that expansion into the original sum produces zeta functions of integer arguments, and a much-faster-converging sum:

$$\begin{aligned} s &= \zeta(2) - \zeta(4) + \zeta(6) - \zeta(8) + \zeta(10) - \sum_{n=1}^{\infty} \frac{1}{n^{12} + n^{10}} \\ &\approx (\zeta(2) - \zeta(4) + \zeta(6) - \zeta(8) + \zeta(10)) - \\ &\quad (0.500\,000\,000 + 0.000\,195\,312 + 0.000\,001\,694 + 0.000\,000\,056 + \cdots) \\ &\approx (1.576\,871\,117) - (0.500\,000\,000 + 0.000\,195\,312 + 0.000\,001\,694 + \cdots) \\ &\approx 1.076\,674\,047\,. \end{aligned}$$

With that new expansion, only *six terms* of the infinite sum are needed to produce ten correct digits, an accuracy that is confirmed by results from Maple's `sum()` function:

```
> for Digits from 10 to 40 by 10 do
>     printf("%3d  %.24f\n", Digits, sum(1 / (1 + n**2), n = 1 .. infinity))
> end do:
 10  1.076674048000000000000000
 20  1.076674047468581174200000
 30  1.076674047468581174134051
 40  1.076674047468581174134051
```

By contrast, adding a million terms of the original sum produces only five correct digits.

18.7.1 Computing the Riemann zeta function

The Riemann zeta function is absent from the mathematical libraries of ISO Standard programming languages. However, Maple supplies it as `Zeta(z)`, and Mathematica as `Zeta[z]`. MATLAB, Maxima, MuPAD, and PARI/GP agree on calling it `zeta(z)`. Despite the capitalized function name in two of those languages, the mathematical symbol for the function is always the lowercase Greek letter: $\zeta(z)$.

Only a few mathematical libraries include the zeta function:

- The Cephes library [Mos89, §7.10.2] provides `zetac(x)` for $\zeta(x) - 1$, and `zeta(x,a)` for computing the Hurwitz zeta function, $\sum_{k=1}^{\infty}(k + a)^{-x}$, a generalization of the Riemann zeta function.
- The *GNU Scientific Library* [GDT+05] has four zeta functions:

`gsl_sf_zeta_int(n)`	returns $\zeta(n)$,
`gsl_sf_zeta(x)`	returns $\zeta(x)$,
`gsl_sf_zetam1_int(n)`	returns $\zeta(n) - 1$,
`gsl_sf_zetam1(x)`	returns $\zeta(x) - 1$.

- Thompson includes a short Fortran implementation in his book [Tho97], but only for $\zeta(x)$ with $x > 1$. He claims about 10-digit accuracy.
- Baker's implementation in C [Bak92, page 519] handles arbitrary real arguments.

Surprisingly, no implementation of the Riemann zeta function can be found in the *ACM Collected Algorithms*, or the journals *ACM Transactions on Mathematical Software (TOMS)* and *Applied Statistics*, or Netlib, or the commercial IMSL and NAG libraries; they are otherwise rich sources of high-quality mathematical software.

The code in the `mathcw` library files `zetax.h` and `zetm1x.h` is inspired by Moshier's computational algorithm. However, we include Taylor-series evaluation, and use completely different polynomial fits.

***x* is a NaN** : Set a domain error, and set the result to that NaN.

***x* is $-\infty$** : Set a range error, and set the result to $+\infty$, even though the sign is indeterminate, and the function range is $[-\infty, +\infty]$. A NaN could also be a suitable return value for this case, but there is no ISO Standard to guide us.

x is a whole number : Use the fast special-case code in `ZETNUM(n)`. Its algorithm is described shortly.

$-\infty \le x \le -$`XMAX` : The argument magnitude is so large that it has no fractional part, and is even (because all practical floating-point bases are even). Set the result to 0, because $\zeta(-2n) = 0$. A suitable value for `XMAX` in base β and t-digit precision is β^t.

$-$`XMAX` $< x < -$Taylor-series cutoff : Apply the reflection formula to remap the computation to the positive axis. However, because of the large error magnification of the functions in the reflection formula, do the trigonometric argument reduction carefully (preferably with the `COSPI()` and `SINPI()` families), and use the next higher precision, when available.

$|x| <$ Taylor-series cutoff : Sum the Taylor series for $\zeta(x)$ to produce a small correction to the value $\zeta(0) = -\frac{1}{2}$. We use series of orders 2, 4, 8, 16, and 32 to improve accuracy in part of the troublesome interval $(0, 1)$.

Taylor-series cutoff $\le x < 1$: Recover the function value from $(1 - x)(\zeta(x) - 1) \approx \mathcal{R}_1(x)$, where $\mathcal{R}_1(x)$ is a minimax rational polynomial fit. Use error recovery to reduce the error in forming $1/(1 - x)$, and form the function result like this:

$$\zeta(x) = \begin{cases} 1 + \left(1 + \dfrac{x}{1-x}\right)\mathcal{R}_1(x), & \text{for } 0 < x < \frac{1}{2}, \\ 1 + \dfrac{1}{1-x}\mathcal{R}_1(x), & \text{for } \frac{1}{2} \le x < 1. \end{cases}$$

The first case preserves accuracy as $x \to +0$, and the second has an exact denominator.

$0 < |x - 1| <$ Taylor-series cutoff : Sum the Taylor series for $\zeta(1 + t)$, where $t = x - 1$. We use series up to order 32.

$x = 1$: Set a range error, and return $+\infty$.

$1 < x < 10$: Sum Taylor series near $x = 2, 3$, and 4 for improved accuracy. Otherwise, use $\zeta(x) \approx 1 + 2^{-x}(x/(x - 1))\mathcal{R}_2(1/x)$, where $\mathcal{R}_2(1/x)$ is a rational minimax polynomial fit. Notice that the variable in that polynomial is a reciprocal; that choice makes the polynomial almost linear in this region. Compute 2^{-x} with `EXP2(-x)` in preference to `POW(TWO,-x)`.

$10 \le x < 26$:

The function value is $\zeta(x) \approx 1 + (2^{-x} + 2^{-x}\exp(\mathcal{R}_3(x)))$, where the fitting function $\mathcal{R}_3(x) = \log((\zeta(x) - 1)2^x - 1)$ is nearly linear in this region. The polynomial term contributes less than 0.00002 of the final function value in this region.

Moshier has a minimax fit in this case for a somewhat different function, but Maple is unable to compute minimax fits of $\mathcal{R}_3(x)$ beyond the lowest required precisions, so our code falls back to a Chebyshev expansion for $\mathcal{R}_3(x)$ at higher precisions.

The choice of endpoints ensures exact conversion of x to the Chebyshev variable on $[-1, +1]$.

$26 \le x < 90$: Use a fit $\mathcal{R}_4(x)$ of the same form as the previous region. The polynomial term contributes less than 4×10^{-13} of the function value in this region. Using two separate fits on $[10, 90]$ reduces the length of the Chebyshev expansions.

For decimal arithmetic, we use a similar fit, but for the interval $[10, 110]$, to get an accurate conversion of x to the Chebyshev variable.

$90 \le x <$ `XHIGH` : Sum the zeta-function definition directly, exiting the loop when convergence has been reached. Only a few terms are needed, and we can avoid computing some of the higher powers by factoring them into products of saved lower powers. For example, $4^{-x} = 2^{-x} \times 2^{-x}$, and $6^{-x} = 2^{-x} \times 3^{-x}$.

The cutoff `XHIGH` is determined by finding the smallest x value for which a two-term sum guarantees correct rounding: we then have $2^{-x} = \frac{1}{2}\epsilon/\beta$, or `XHIGH` $= -\log_2(\frac{1}{2}\epsilon/\beta)$. The required cutoffs range from 24.25 in the 32-bit formats up to about 238 in the 256-bit formats. In the binary formats, the cutoff is just $t + 1$ for t-bit precision.

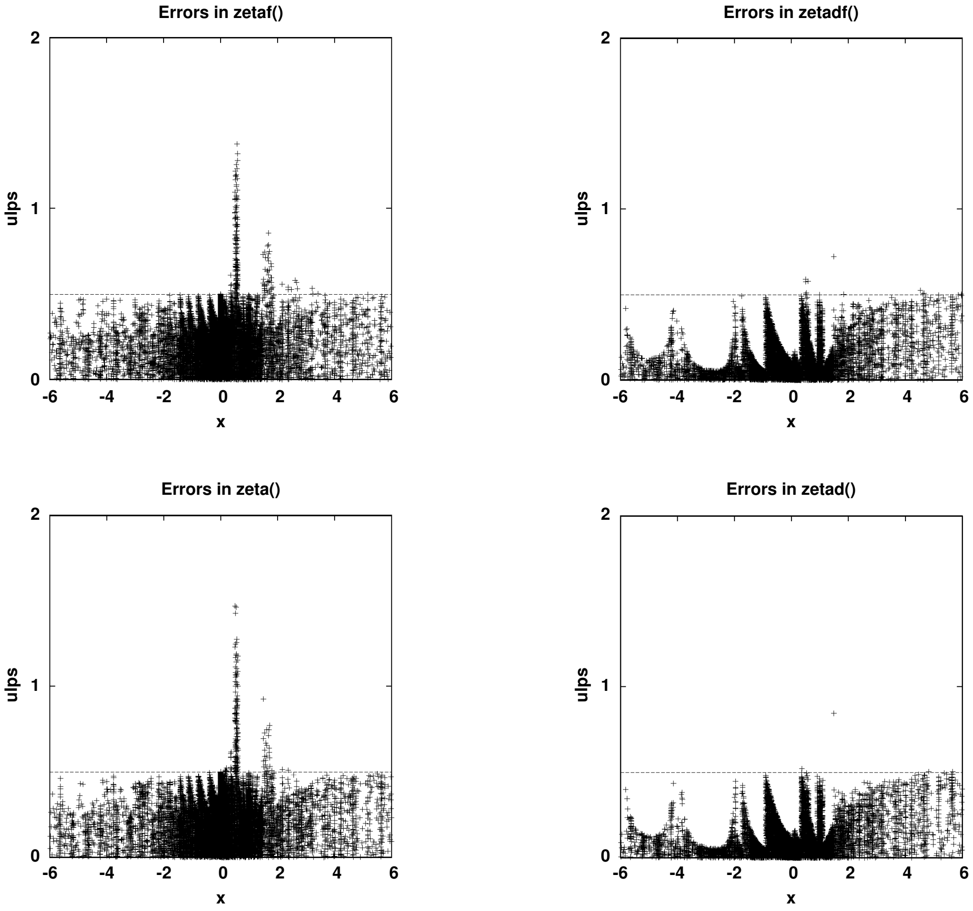


Figure 18.15: Errors in the `ZETA()` functions.

`XHIGH` $< x \leq +\infty$: The function value is 1 to machine precision. Adding a tiny value to that number ensures the setting of the IEEE 754 *inexact* flag.

Moshier was careful to express his fitting functions in terms of $\zeta(x) - 1$. so minimal changes are needed to convert the algorithm to that needed for the variant function `zetam1(x)`. In particular, the changes increase the `XHIGH` cutoffs. For that reason, and to avoid an additional rounding error, it is not advisable to implement `ZETA(x)` simply as `ZETAM1(x)` + `ONE`.

The measured errors in our implementation of $\zeta(x)$ are graphed in Figure 18.15, and those for $\zeta(x) - 1$ are shown in Figure 18.16 on the following page. Outside the argument range shown in those plots, the function results are almost always correctly rounded, although that is only true for negative arguments when a higher-precision data type is available.

For applications that require zeta-function values only for integer arguments, we supply two more function families, `ZETNUM(n)` and `ZETNM1(n)`, that return $\zeta(n)$ and $\zeta(n) - 1$, respectively. They handle the computation as follows:

$n < 0$: Return zero if n is even, and $-B_{2-2\lceil n/2 \rceil}/(2 - 2\lceil n/2 \rceil)$ if n is odd. The required Bernoulli number is obtained from `BERNUM()`, and its code uses fast table lookup.

For `ZETNM1(n)`, use a separate small table of values of $\zeta(n) - 1$ to avoid subtraction loss for n in $[-17, -1]$.

The one minor flaw is that for n large enough that the Bernoulli number is just above the overflow limit, the exact quotient might still be representable, but our algorithm returns an Infinity. To avoid that, we use the next

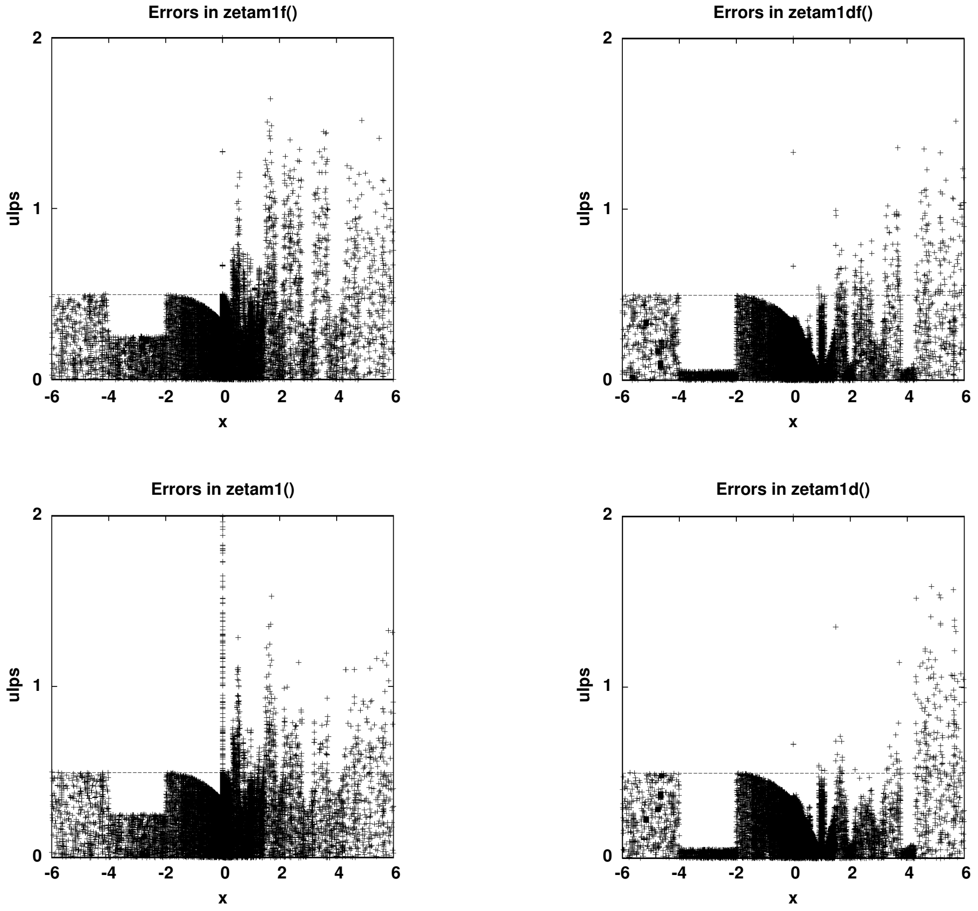


Figure 18.16: Errors in the `ZETAM1()` functions.

higher precision, so that at least for `float`, and often for `double`, function types, we eliminate the premature overflow, except in deficient older floating-point designs with the same exponent range for all precisions.

$\boldsymbol{n = 1}$: Set a range error and return $+\infty$.

$\boldsymbol{n = 0}$ **or** $\boldsymbol{n > 1}$: For n up to a few hundred (depending on the floating-point range), return a correctly rounded value from a compile-time private constant table. Otherwise, n is large enough that the Riemann sum can be truncated to two terms, $1 + 2^{-n}$. Compute the power as `EXP2(-n)`, because that is faster than the `POW()` family, and may be more accurate as well; in particular, it is *exact* in binary arithmetic.

Our implementations of code for the zeta function makes the common case of small positive integer arguments both fast and correctly rounded. That is important for code that needs zeta functions in time-critical inner loops. When arguments are whole numbers, our code also guarantees identical results from the `ZETA()` and `ZETNUM()`, and the `ZETAM1()` and `ZETNM1()`, families. Application programmers are advised to consider revising computations to exploit the higher effective accuracy provided by the `ZETAM1()` and `ZETNM1()` routines.

18.7.2 Greek relatives of the Riemann zeta function

There are three relatives of the zeta function that introduce alternating signs, or include only odd integers [AS64, Chapter 23]:

$$\beta(x) = \sum_{k=1}^{\infty}(-1)^{k-1}(2k-1)^{-x} = \sum_{k=0}^{\infty}(-1)^{k}(2k+1)^{-x}, \qquad \textit{beta function,}$$

$$\eta(x) = \sum_{k=1}^{\infty}(-1)^{k-1}k^{-x} = (1-2^{1-x})\zeta(x), \qquad \textit{eta function,}$$

$$\lambda(x) = \sum_{k=1}^{\infty}(2k-1)^{-x} = (1-2^{-x})\zeta(x), \qquad \textit{lambda function.}$$

Because of their simple relation to the zeta function, there is no strong reason to implement special code for the eta and lambda functions: the required scale factor is easily computed to machine precision with the `EXP2M1()` family.

The beta function is often prefixed by the names Catalan or Dirichlet, Nineteenth Century Belgian and German mathematicians who contributed to the study of that function. It is defined for negative arguments through this relation, valid for all complex z:

$$\beta(1-z) = \left(\frac{2}{\pi}\right)^{z} \sin(\tfrac{1}{2}\pi z)\Gamma(z)\beta(z).$$

The beta function for real arguments is graphed in **Figure 18.17** on the next page.

For negative integer arguments, and odd positive integer arguments, the beta function can be readily obtained from the Euler numbers returned by our `EULNUM()` family through the relations

$$\beta(-2n) = \tfrac{1}{2}E_{2n}, \qquad \beta(-2n-1) = 0, \qquad \beta(2n+1) = \frac{(\frac{1}{2}\pi)^{2n+1}}{2(2n)!}|E_{2n}|.$$

No simple closed form is known for the beta function of positive even integer arguments.

Here are some special values of those functions:

$$\begin{aligned}
&\eta(1) = \log(2), && \eta(2) = \frac{\pi^2}{12}, && \eta(4) = \frac{7\pi^4}{720},\\
&\lambda(1) = \infty, && \lambda(2) = \frac{\pi^2}{8}, && \lambda(4) = \frac{\pi^2}{96},\\
&\beta(0) = \tfrac{1}{2}, && && \\
&\beta(1) = \frac{\pi}{4}, && \beta(2) \approx 0.915\,965\,594\,177\,219\,015\,054\,603\,\ldots, && \beta(3) = \frac{\pi^2}{32}.
\end{aligned}$$

The constant $\beta(2)$ is known as *Catalan's constant*. Maple and Mathematica provide it as the built-in value `Catalan` and can evaluate it numerically to arbitrary precision. It is related to the Riemann zeta function like this:[12]

$$\beta(2) = 1 - \sum_{k=1}^{\infty}\frac{k\,\zeta(2k+1)}{16^k}.$$

The sum converges reasonably quickly, and generates about as many decimal digits as terms summed: 64 terms produce 75 correct digits.

Most of the symbolic-algebra systems available to this author are unable to produce a Taylor series for the beta function, but Maple succeeds:

$$\beta(x) = \left(\sum_{k=1}^{\infty}(-1)^{k-1}\right) + \left(\sum_{k=1}^{\infty}(-1)^{k-1}\log(2k-1)\right)x +$$

[12] See the *MathWorld* article at `http://mathworld.wolfram.com/CatalansConstant.html` or in the print edition [Wei09].

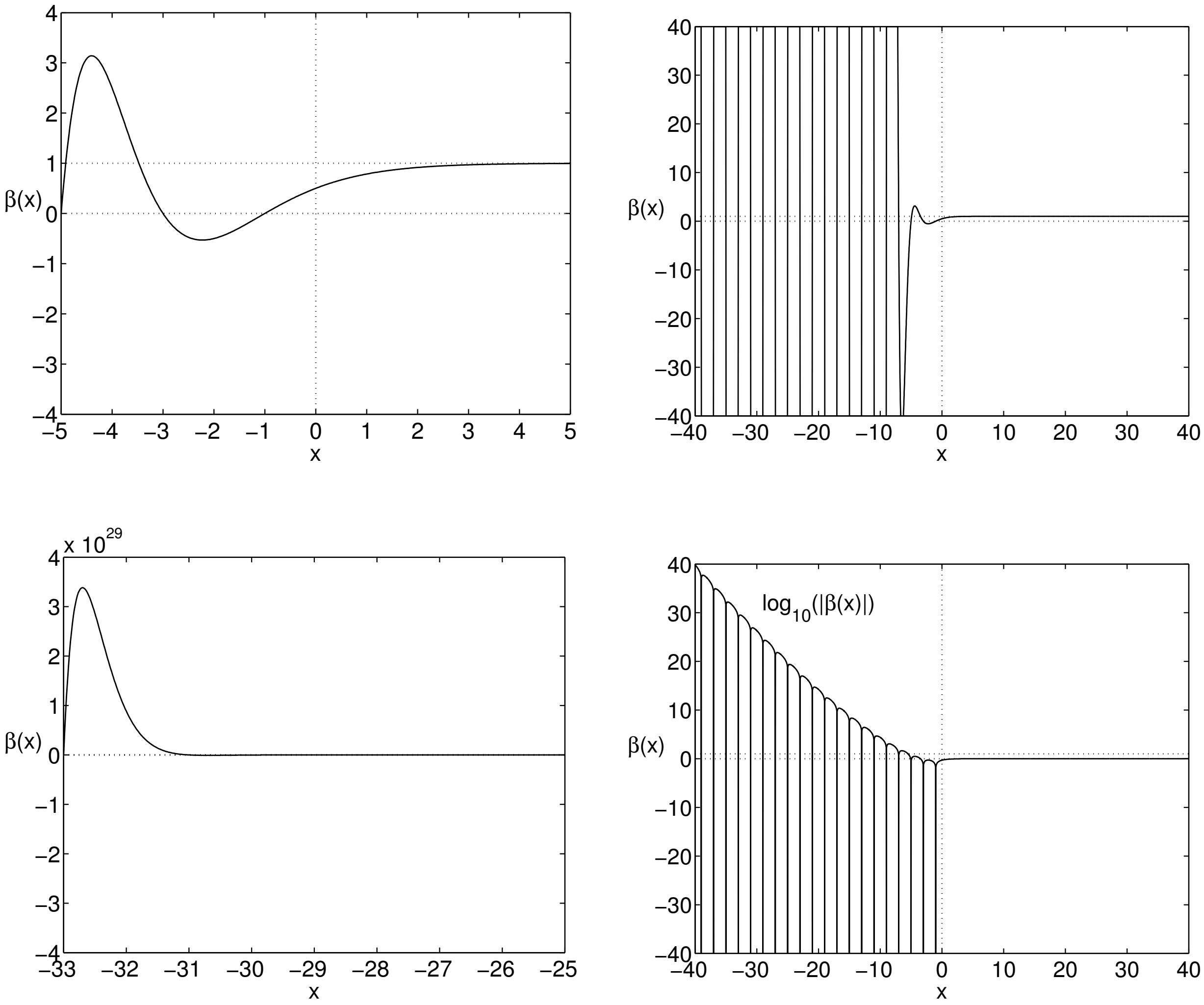


Figure 18.17: The real Catalan/Dirichlet beta function on four different scales. The dotted horizontal and vertical lines mark the axes and the limiting value for large positive arguments.
On the positive axis, the function is well-behaved and free of poles and zeros. On the negative axis, the initial good behavior seen in the upper-left graph soon becomes wild oscillations evident in the upper-right plot.
Function growth is so rapid on the negative axis that an attempt to see the oscillations on a linear scale fails, as in the lower-left graph. However, the growth becomes manageable when we take a logarithm of the absolute value, as shown in the lower-right plot.

$$\left(\sum_{k=1}^{\infty} \tfrac{1}{2}(-1)^{k-1}(\log(2k-1))^2\right) x^2 +$$
$$\left(\sum_{k=1}^{\infty} -\tfrac{1}{6}(-1)^{k-1}(\log(2k-1))^3\right) x^3 + \cdots .$$

That is daunting, and also puzzling, because it appears that each of the infinite sums is either uncertain, or likely to diverge. In mathematics, infinite sums often hold surprises, and Maple is fortunately able to numerically evaluate the expansion taken to terms of higher order:

$$\begin{aligned}\beta(x) = {} & \tfrac{1}{2} + 0.391\,594\,392\,706\,837\,x - 0.116\,416\,441\,749\,488\,x^2 + \\ & 0.004\,513\,226\,707\,904\,77\,x^3 + 0.008\,472\,226\,111\,133\,55\,x^4 - \\ & 0.003\,363\,554\,823\,024\,05\,x^5 + 0.000\,655\,340\,450\,069\,919\,x^6 - \\ & 0.000\,049\,935\,341\,465\,263\,3\,x^7 - 0.000\,011\,067\,375\,016\,246\,8\,x^8 + \\ & 0.000\,004\,831\,936\,836\,840\,8\,x^9 + \mathcal{O}(x^{10})\end{aligned}$$

It is therefore feasible to tabulate numerical expansion coefficients in problem regions, such as near the function zeros at odd negative integer arguments.

Baker [Bak92, page 526] provides a brief implementation of the beta function for positive arguments, but tests show that his algorithm is only practical for $x > 4.92$, because for smaller arguments, the internal summation does not converge to working precision.

The author of this book has been unable to find other published algorithms specifically for the beta function, but there is a useful class of techniques for summing alternating series, such as those for the beta and eta functions, that provides an effective solution through *convergence acceleration*.[13] Two of the earliest-known approaches are due to Euler (17??) and Kummer (1837) [AS64, §3.6.27, §3.6.26] [OLBC10, §3.9], but better methods have since been discovered. There are book-length treatments of the subject [BR91, BGM96, Sid03], and shorter descriptions [HCL$^+$68, §2.8] [PTVF07, §5.3]. One of the simplest recent methods for finding the sum of an alternating series with decreasing terms [CVZ00] requires no array storage, and is implemented in this hoc function:

```
func altsum(x, n)                    \
{   # return sum(k = 0:infinity) (-1)**k * A(x,k) using first n terms
    e = (3 + sqrt(8))**n
    d = (e + 1/e) / 2
    b = -1
    c = -d
    sum = 0

    for (k = 0; k <= n; ++k)          \
    {
        c = b - c
        sum += c * A(x, k)
        b = (k + n) * (k - n) * b / ((k + 1/2) * (k + 1))
    }

    return (sum / d)
}
```

For the beta and eta functions, and the zeta function through its relation to the latter, the algorithm is surprisingly effective. Here are some results for brute-force and accelerated computation of the beta function:

$$\begin{aligned}\beta(x,n) &= \sum_{k=0}^{n}(-1)^k(2k+1)^{-x}, && \textit{n-term approximation to } \beta(x), \\ \beta(1, 1000) &\approx 0.785\,6, \\ \beta(1, 1\,000\,000) &\approx 0.785\,398\,4, \\ \beta(1, 1\,000\,000\,0000) &\approx 0.785\,398\,163\,1, \\ \beta(1) &\approx 0.785\,398\,163\,397\,448\,309\,615\,660\,845\,819\,875\,721\,049\,\ldots, \\ \texttt{altsum(1, 10)} &\approx 0.785\,398\,163\,4, \\ \texttt{altsum(1, 20)} &\approx 0.785\,398\,163\,397\,448\,309\,9, \\ \texttt{altsum(1, 30)} &\approx 0.785\,398\,163\,397\,448\,309\,615\,660\,848,\end{aligned}$$

[13] See `http://mathworld.wolfram.com/ConvergenceImprovement.html` and `http://en.wikipedia.org/wiki/Series_acceleration`.

$$\texttt{altsum(1, 40)} \approx 0.785\,398\,163\,397\,448\,309\,615\,660\,845\,819\,877.$$

The numerical values are truncated after the first incorrect digit. The results for $\beta(1,n)$ are from a high-precision symbolic-algebra computation, and those for our hoc function use 34-digit decimal arithmetic. The last computed value is within 2.14 ulps of the exact $\beta(1)$.

As usual with sequences of decreasing terms, it is advisable to delay summing the leading term until last, to minimize accumulation of rounding errors. The changes to `altsum()` are simple, but we omit them here.

In `altsum()`, the fact that d requires a square root and a power, determines the starting value of c, and appears as a divisor in the final result, means that the final accuracy is limited by the quality of the power function. That is another reason why it is advisable to sum one or more of the leading terms separately.

To apply a function like `altsum()` in practical code, we could compute the sums at run time with increasing values of n, terminating when the results are sufficiently converged. However, it is better to make experiments to determine the minimal values of n that produce converged results for fixed computational precision, and the required ranges of x. A short table of (x,n) pairs then allows a suitable single value of n to be found quickly for a given argument x.

The mathcw library provides four function families for support of the beta function:

```
double beta   (double x);       /* beta(x)     */
double betam1 (double x);       /* beta(x) - 1 */
double betnum (int n);          /* beta(n)     */
double betnm1 (int n);          /* beta(n) - 1 */
```

They have the usual companions for other floating-point types. For increasing positive x, $\beta(x)$ approaches 1 from below, so the functions `betam1(x)` and `betnm1(n)` provide an accurate value of the difference from that limiting value.

The functions of integer arguments use fast table lookup for n in $[0,100]$ to retrieve correctly rounded values of the beta function. Below that interval, they use the `EULNUM()` family for the required Euler numbers. Above that interval, they sum the rapidly decreasing series of terms $(-1)^k(2k+1)^{-n}$ to find the function value.

Our algorithms for the `BETA()` and `BETAM1()` families are patterned after those used for the zeta function and described in Section 18.7.1 on page 583. However, the code is simplified by avoiding Taylor-series expansions near zeros on the negative axis, and instead using excursions to the next higher precision, so that our beta-function results are almost always correctly rounded, and the usual error plots are therefore omitted. When a higher precision is not available, measurements show errors up to 2.5 ulps for x in $[0,5]$, and up to 1 ulp in $[5,15]$.

The Catalan/Dirichlet beta function is absent from most symbolic-algebra systems, but some of those systems can evaluate infinite sums numerically. From code in the files `beta*.map`, Maple produced the private table of values of $\beta(n)-1$ in `betnm.h` that `BETNM1(n)` accesses. Here is a suitable function definition for Maple:

```
betam1 := proc(x)
              local k:
              return Re(evalf(sum((-1)**(k - 1)*(2*k - 1)^(-x), k = 2 .. infinity)))
          end proc:
```

A Mathematica implementation looks like this:

```
betam1 = Function[x, Sum[(-1)^(k - 1)*(2*k - 1)^(-x), {k,2,Infinity}]]
```

Despite the infinite sums, the algebra systems are able to evaluate $\beta(x)-1$ numerically reasonably quickly, although the Maple version is slow for large arguments.

18.8 Further reading

There is a huge mathematical literature on zeta functions. We cited several books [CJW06, Clay09, Der03, Gra00, Lau08, Sab03], and one historical survey article [Ayo74] to guide the reader to some of the more interesting features of zeta functions, without exposing the deep mathematics that underlies the zeta function, and that has been developed since the work of Euler and Riemann.

We also remarked on the connection of the zeta function to prime numbers, a subject that has occupied mathematicians for millennia. See Ribenboim's books [Rib91, Rib96, Rib04] for recent surveys of that topic.

Prime numbers are no longer just of interest in pure mathematics. They lie at the heart of some of the most effective algorithms in modern cryptography, and are thus intimately related to the security of modern communications systems. Much of the introductory material on prime numbers can be understood by anyone with only high-school mathematics, without the need for calculus or computers. There are many interesting books on the subject, and this author has several personal favorites [FS03, FF01, Sch96, Sch96, Sch00, Sch03, Sin99]. Some of them are likely to available in local community libraries, making them readily accessible to readers without access to a large academic library. Many other books and research articles are recorded in several extensive online bibliographies; look for the string *crypto* in the archive index at `http://www.math.utah.edu/pub/tex/bib/index-table.html`.

Eric Weisstein's *MathWorld* Web site is a rich source of well-written, and reliable, short articles on almost any area of modern mathematics. Importantly, it contains many links to other online resources, as well as to published literature. Access to the *MathSciNet* and *zbMATH* databases may be available through a nearby academic library; those databases provide an excellent way to track topics in mathematics back to the original research articles.

The extraordinary contributions of Leonhard Euler to mathematics are documented in several books and articles, some of which were published to mark the 300-year anniversary of Euler's birth [BMY07, BDS07, BS07, Dun91, Dun99, Dun07, Fel07, Fin97, Gau08, HHPM07, Nah06, San07a, San07b]. Euler and the Bernoulli family are also covered in two other books [Dun91, Ten09], and there is even a book devoted to the history of the Euler–Mascheroni constant [Hav03]. Republications of Euler's original works, *Leonhardi Euleri opera omnia*, continue to be produced, and fill more than *fifty* volumes in Latin [Eul92].[14]

18.9 Summary

The gamma and log-gamma functions are among the most important of all of the special functions in mathematics. Their logarithmic derivatives, the psi function and the higher polygamma functions, are less commonly encountered. Nevertheless, they all deserve accurate treatment, because they often occur in critical kernels of other computations. The Fortran 2008 Standard [FTN10] introduces functions with the names `gamma(x)` and `log_gamma(x)`, but some vendors have supplied them for decades, possibly under different names. The C library does not supply a factorial function, yet that function is commonly needed in combinatorics, probability, and statistics. Our gamma function therefore handles the case $\Gamma(n+1) = n!$ by fast table lookup, ensuring correctly rounded factorials over the entire range where they are representable in single- and double-precision formats.

Like the gamma function, the incomplete gamma functions turn up in a surprising number of areas, and one of their most common applications is for the computation of probability functions, such as the chi-square measure. For that reason alone, the incomplete gamma functions, and the chi-square function and its inverse, deserve to be part of standard mathematical libraries in all major programming languages.

The psi function is rare, but it is important to remember that it is a useful tool for evaluating the partial sums of the harmonic series that show up in many applications, especially in number theory and the analysis of computer algorithms. We meet the psi function again in this book in the chapter on Bessel functions (see **Section 21.3** on page 703).

The related polygamma functions are needed for evaluation of certain reciprocal sums, and cannot be computed accurately with simple code. Our implementations of those functions handle arbitrary order and argument range, and on many systems, the single and double precision versions are accurate to working precision.

Factorials, binomial coefficients, and the special numbers associated with the beta and zeta functions, and the names of Bernoulli, Euler, and Fibonacci, are commonly needed in series expansions, so it makes sense to provide accurate implementations of them in mathematical software libraries. The `mathcw` library supplies the `FACT()` and `DFACT()` families for single and double factorials, the `LOGBFACT()` family for base-β logarithms of factorials, and the `BINOM()` family for binomial coefficients. The latter are programmed to avoid premature overflow, and all of the functions that supply those special numbers use fast table lookup to handle most common cases. Outside the range of tabulated values, they take care to reduce unnecessary rounding error in order to achieve high accuracy. The factorial functions also remember the last computed value so that it can be returned quickly if they are called again with the same argument. The remembered value can also sometimes serve as an accurate starting value for further computation on subsequent calls.

The zeta-function family is rarely provided in mathematical software libraries, but deserves to be. Our implementation achieves satisfactory accuracy for positive real arguments, but can do so for negative arguments only when

[14] See also the online Euler Archive at `http://www.math.dartmouth.edu/~euler/`.

higher precision is available. The zeta-function definition for negative arguments requires the reflection rule (see **Section 18.7** on page 579), and that in turn requires exponentials, powers, trigonometric functions, and zeta functions of positive arguments. The rapidly changing, and steep, function values in the negative-argument region strongly suggest that higher precision for all of those auxiliary functions is the only practical way of reliably computing the zeta functions there.

Beta, eta, lambda, and zeta functions of integer arguments have special properties that allow them to be derived from Euler and Bernoulli numbers, exponentials, or fast table lookup. Because some applications that need zeta functions require them only for integer arguments, it is worthwhile to handle them in separate library functions that compute them more quickly, and almost-always correctly rounded as well.

The approach to limiting values of the beta and zeta functions recommends the provision of auxiliary functions that compute accurate differences from those limits: the `mathcw` library accordingly supplies the `BETAM1()`, `BETNM1()`, `ZETAM1()`, and `ZETNM1()` families.

The techniques shown in **Section 18.7** on page 582 and **Section 18.7.2** on page 589 for dramatic acceleration of the convergence of sums are worthy of further study, because sums are ubiquitous in applied mathematics.

19 Error and probability functions

> **erf**, *n.*: In Cape Colony, some parts of the State of New York, and other regions originally settled by the Dutch, a small inherited house-and-garden lot in a village or settlement.
>
> **phi**, *n.*: The Greek letter ϕ, φ, Φ, corresponding to the English *ph* (f).
>
> — *New Century Dictionary* (1914).

Like the gamma and psi functions, the functions treated in this chapter are among the most important of the *special functions*.

The mathematical properties of the functions in this chapter are summarized in [AS64, OLBC10, Chapter 7] and [SO87, Chapter 40]. Computational algorithms for them are given in [HCL+68, Chapter 6], [Mos89, Chapter 5], [Bak92, Chapter 7], [Tho97, Chapter 10], and [ZJ96, Chapter 16], as well as in some research articles [Cod69, Cod88a, Cod90, CS91, Cod93b].

19.1 Error functions

The error function, `erf(x)`, is defined as the area under the scaled normal curve between the origin and x:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)\, dx.$$

The normalizing factor $2/\sqrt{\pi}$ ensures that $\operatorname{erf}(\infty) = 1$.

The complementary error function, `erfc(x)`, is the area under the scaled normal curve between x and infinity:

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty \exp(-t^2)\, dx.$$

The two functions, and the scaled normal curve, are shown in **Figure 19.1** on the next page.

The normal and complementary error functions, and their close relatives described in **Section 19.4** on page 610, figure prominently in applications in probability and statistics, and after the original Fortran elementary functions, are probably the most important functions in mathematical and statistical software. It is therefore regrettable that they were left out of common programming languages for fifty years, until their inclusion in C99. The Fortran 2008 Standard [FTN10] introduces them to that language with the same names, and also provides `erfc_scaled(x)` for the value of $\exp(x^2) \times \operatorname{erfc}(x)$.

19.1.1 Properties of the error functions

The normal and complementary error functions satisfy the relations

$$\begin{aligned}
\operatorname{erf}(x) + \operatorname{erfc}(x) &= 1, && \text{for } x \text{ in } (-\infty, +\infty),\\
\operatorname{erf}(-x) &= -\operatorname{erf}(x),\\
\operatorname{erfc}(-x) &= 2 - \operatorname{erfc}(x),
\end{aligned}$$

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_19

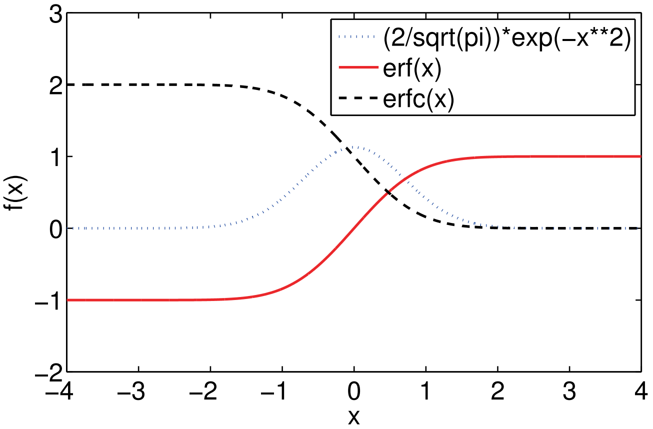


Figure 19.1: Error functions and normal curve.

and have these special values:

$$
\begin{aligned}
\operatorname{erf}(-\infty) &= -1, & \operatorname{erfc}(-\infty) &= 2,\\
\operatorname{erf}(0) &= 0, & \operatorname{erfc}(0) &= 1,\\
\operatorname{erf}(\pm 0.476\,936\ldots) &= \pm\tfrac{1}{2}, & \operatorname{erfc}(0.476\,936\ldots) &= \tfrac{1}{2},\\
\operatorname{erf}(+\infty) &= 1, & \operatorname{erfc}(+\infty) &= 0.
\end{aligned}
$$

For small arguments, the error functions have these Taylor series expansions:

$$
\begin{aligned}
\operatorname{erf}(x) &= (2/\sqrt{\pi})\left(x - x^3/3 + x^5/10 - x^7/42 + x^9/216 - x^{11}/1320 + \cdots\right)\\
&= (2/\sqrt{\pi})\sum_{k=0}^{\infty}\frac{(-1)^k}{(2k+1)k!}x^{2k+1},\\
\operatorname{erf}(x) &= (2/\sqrt{\pi})\exp(-x^2)(x + 2x^3/3 + 4x^5/15 + \cdots)\\
&= \exp(-x^2)\sum_{k=0}^{\infty} x^{2k+1}/\Gamma(k+3/2).
\end{aligned}
$$

Although the gamma function occurs formally in the expansion of $\operatorname{erf}(x)$, in practice, it is not required to sum the series, because the coefficients can be obtained from the starting value $\Gamma(3/2) = \sqrt{\pi}/2$ and the recurrence relation $\Gamma(z+1) = z\Gamma(z)$. The outer factor $2/\sqrt{\pi}$ is, of course, applied last.

The error functions have simple continued-fraction expansions for $x > 0$:

$$
\begin{aligned}
\operatorname{erf}(x) &= (2/\sqrt{\pi})\exp(-x^2)\left(\frac{x}{1-}\,\frac{2x^2}{3+}\,\frac{4x^2}{5-}\,\frac{6x^2}{7+}\,\frac{8x^2}{9-}\,\frac{10x^2}{11+}\,\frac{12x^2}{13-}\cdots\right),\\
\operatorname{erfc}(x) &= (1/\sqrt{\pi})\exp(-x^2)\left(\frac{1}{x+}\,\frac{1/2}{x+}\,\frac{2/2}{x+}\,\frac{3/2}{x+}\,\frac{4/2}{x+}\,\frac{5/2}{x+}\,\frac{6/2}{x+}\cdots\right).
\end{aligned}
$$

Numerical tests of those expansions with their implementations in the files `erf-cf.hoc` and `erfc-cf.hoc` show reasonable convergence for $\operatorname{erf}(x)$ when $0 < x < 1$, and for $\operatorname{erfc}(x)$ when $x > 4$.

For large arguments, the complementary error function has an asymptotic expansion:

$$\begin{aligned}
\operatorname{erfc}(x) &\asymp (1/(x\sqrt{\pi}))\exp(-x^2)\left(1 - \frac{1}{2x^2} + \frac{3}{4x^4} - \frac{15}{8x^6} + \frac{105}{16x^8}\right. \\
&\qquad \left. - \frac{945}{32x^{10}} + \frac{10\,395}{64x^{12}} - \frac{135\,135}{128x^{14}} + \frac{2\,027\,025}{256x^{16}} - \frac{34\,459\,425}{512x^{18}} + \cdots\right) \\
&\asymp (1/(x\sqrt{\pi}))\exp(-x^2)\sum_{k=0}^{\infty}\frac{(-1)^k(2k)!}{4^k\,k!\,x^{2k}}.
\end{aligned}$$

Figure 2.1 on page 20 gives a flavor of the applicability of the asymptotic expansion for practical computation.

Because the exponential $\exp(-t^2)$ in the integrand falls off so rapidly, $\operatorname{erf}(x)$ approaches its limit value quickly: in 32-bit IEEE 754 arithmetic, $\operatorname{fl}(\operatorname{erf}(4)) = 1$ to machine precision. In 64-bit IEEE 754 arithmetic, $\operatorname{fl}(\operatorname{erf}(6)) = 1$ to machine precision, and in 128-bit IEEE 754 arithmetic, $\operatorname{fl}(\operatorname{erf}(9)) = 1$ to machine precision.

That rapid approach to the limit has these ramifications:

- By subtraction from one, it is possible to compute one error function from the other only in a sharply limited range. The two are equal at $x \approx 0.476\,936$, and there is significance loss in forming $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$ in binary arithmetic for x in the approximate interval $[0.48, \infty)$.
- It is always safe to compute the smaller of the error functions, and then obtain the larger by subtraction.
- The complementary error function underflows to zero at $x \approx 11, 28$, and 107 for the three C-language precisions in IEEE 754 arithmetic.

The first derivatives of the error functions are given by:

$$\begin{aligned}
\operatorname{erf}'(x) &= +\sqrt{4/\pi}\exp(-x^2), \\
\operatorname{erfc}'(x) &= -\sqrt{4/\pi}\exp(-x^2) \\
&= -\operatorname{erf}'(x).
\end{aligned}$$

We can use them to compute the error-magnification factors discussed in **Section 4.1** on page 61 and tabulated in **Table 4.1** on page 62, and graph them with MATLAB's implementations of the error functions, as shown in **Figure 19.2** on the following page.

The second derivatives have a particularly simple form that we exploit later in iterative schemes for the computation of function inverses:

$$\begin{aligned}
\operatorname{erf}''(x) &= -2x\operatorname{erf}'(x), \\
\operatorname{erfc}''(x) &= -2x\operatorname{erfc}'(x).
\end{aligned}$$

Mathematically, the error function can be computed from the regularized incomplete gamma function:

$$\operatorname{erf}(x) = \begin{cases} +P(\tfrac{1}{2}, x^2), & \textit{for } x \geq 0, \\ -P(\tfrac{1}{2}, x^2), & \textit{for } x < 0. \end{cases}$$

That approach is used in Thompson's book [Tho97], but that is rarely suitable, because accurate and portable implementations of the incomplete gamma function are hard to find, the right-hand side suffers from premature overflow and underflow in forming x^2, and the relation gives no way to determine the complementary error function accurately.

19.1.2 Computing the error functions

The Sun Microsystems fdlibm package (discussed later in **Chapter 25.3** on page 824) provides an algorithm for the two error functions that can be generalized to handle arithmetics and precisions other than IEEE 754 64-bit arithmetic. The fdlibm recipe provides the basis for the mathcw library code, and also illustrates some interesting programming practices that are worth documenting here. After dealing with the special cases of NaN, zero, and infinite

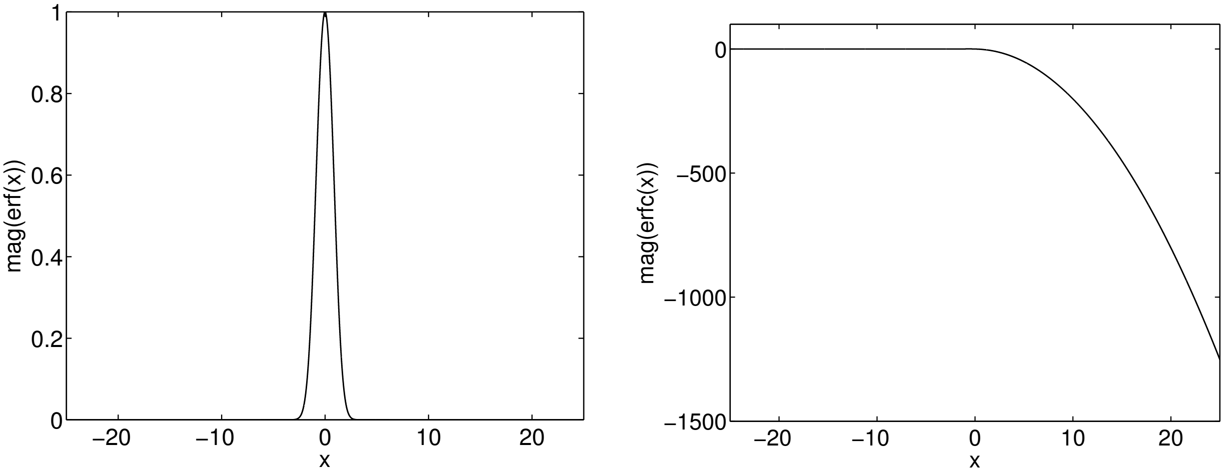


Figure 19.2: Error-magnification factors for the error functions as implemented in MATLAB. Notice the different vertical scales. Although $\operatorname{erf}(x)$ is largely insensitive to argument errors, $\operatorname{erfc}(x)$ has large sensitivity when x is large, or equivalently, the function value is small.

arguments, the algorithm handles the computation in five regions, each with different rational polynomial approximations $\mathcal{R}_r(x) = \mathcal{P}_r(x)/\mathcal{Q}_r(x)$. In the mathcw code that implements that computation, most multiply-add opportunities are wrapped in `FMA()` macros, and expressions of the form $1 + x$ are computed as $(\frac{1}{2} + \frac{1}{2}x) + (\frac{1}{2} + \frac{1}{2}x)$ to avoid bit loss under wobbling precision. For the ordinary error function, our code also adds a sixth region for small arguments, where we sum the Taylor series.

[R0] $|x|$ small:

Sum a six-term Taylor series. The sum is scaled by the transcendental constant $2/\sqrt{\pi}$, a value that introduces additional rounding error, and has leading zero bits in hexadecimal arithmetic. Those leading zeros are eliminated if we instead use $1/\sqrt{\pi}$. For improved accuracy, we represent that constant as a sum of exact high and approximate low parts, and then double the scaled sum.

For $|x|$ near the underflow limit, the intermediate expressions can become subnormal or underflow to zero, so we move away from the underflow region by scaling the computation by the exactly representable inverse cube of the machine epsilon, $1/\epsilon^3$, and then rescale the result by ϵ^3.

[R1] $|x|$ in $[\mathbf{small}, \mathbf{27/32})$:

$$\operatorname{erf}(x) \approx x + x\mathcal{R}_1(x^2),$$

$$\operatorname{erfc}(x) \approx \begin{cases} 1 - \operatorname{erf}(x), & \textit{for } x \textit{ in } [-27/32, +1/4), \\ \frac{1}{2} + \left((\frac{1}{2} - x) - x\mathcal{R}_1(x^2)\right), & \textit{for } x \textit{ in } [+1/4, 27/32). \end{cases}$$

The error function must *not* be computed as $\operatorname{erf}(x) = x(1 + \mathcal{R}_1(x^2))$, because that loses three leading bits on base-16 systems.

In this region, the magnitude of the term $x\mathcal{R}_1(x^2)$ is never more than $0.13|x|$, so there is no possibility of subtraction loss in binary arithmetic, and the result is determined from a small correction to an exact value.

[R2] $|x|$ in $[\mathbf{27/32}, \mathbf{5/4})$:

$$s = |x| - 1,$$

$$\operatorname{erf}(x) \approx \operatorname{sign}(x)(C_2 + \mathcal{R}_2(s)),$$

$$\operatorname{erfc}(x) \approx \begin{cases} (1 - C_2) - \mathcal{R}_2(s), & \textit{for } x > 0, \\ 1 + (C_2 + \mathcal{R}_2(s)), & \textit{for } x < 0. \end{cases}$$

The constant C_2 is chosen to lie roughly halfway between $\operatorname{erf}(27/32)$ and $\operatorname{erf}(5/4)$, so that the rational polynomial is a minimal correction to that constant.

The fdlibm choice is the halfway point, truncated to single precision, but that value is then tied to IEEE 754 arithmetic.

A more portable choice used in the mathcw library is the nearby value $C_2 = 27/32$, which is exactly representable in both binary and decimal on all current and historical floating-point architectures. That choice makes the rational polynomials usable on all systems.

The range of the rational polynomials is then roughly $[-0.076, +0.079]$, a small correction to C_2 that effectively gains one decimal digit of precision.

[R3] $|x|$ in $[\mathbf{5/4}, \mathbf{1/0.35})$:

$$\begin{aligned} z &= 1/x^2, \\ \operatorname{erfc}(x) &\approx (1/x)\exp(-x^2 - 9/16 + \mathcal{R}_3(z)), \\ \operatorname{erf}(x) &= 1 - \operatorname{erfc}(x). \end{aligned}$$

In this region, $\operatorname{erfc}(x)$ is less than 0.078, so $\operatorname{erf}(x)$ is determined by a small correction to an exact constant. However, our approximation for $\operatorname{erfc}(x)$ requires a function evaluation and a division, introducing at least two rounding errors into the result. The exponential decrease of the complementary error function makes it difficult to represent as the sum of an exact term and a small correction.

The range of $-x^2 - 9/16$ is $[-0.000\,175, -0.119]$ in this region, and $\mathcal{R}_3(z)$ changes sign. Thus, there is the possibility of subtraction loss in forming the argument of the exponential. However, there is another problem: x^2 requires twice the number of bits available, so its truncation introduces an argument error that is magnified by x^2 in the function value (see **Table 4.1** on page 62). The solution is to represent x^2 as the sum of an exact value and a small correction, then factor the two parts into two calls to the exponential:

$$\begin{aligned} s &= (\text{float})x, \\ x^2 &= s^2 - (s^2 - x^2) \\ &= s^2 - (s - x)(s + x), \\ \exp(-x^2 - 9/16 + \mathcal{R}_3(z)) &= \exp(-s^2 - 9/16) \times \\ &\quad\ \exp((s - x)(s + x) + \mathcal{R}_3(z)). \end{aligned}$$

Numerical experiments show that the relative error in the two ways to compute the exponential grows with x, and for $x \approx 25.568$, can be more than 500 ulps in the IEEE 754 80-bit format, corresponding to a loss of about 9 bits.

In current and most historical floating-point systems, except for the Harris /6 and /7 (see **Table H.1** on page 948), there are at least twice as many significand bits in a `double` as in a `float`, so coercing x to a `float` produces a value whose square is exactly representable in `double` or higher precision.

In the first exponential, the argument is exactly representable in bases that are a power of two. In the second exponential, the term with the factor $s - x$ is of the order of the single-precision machine epsilon, so it is a small correction to the rational polynomial.

For single-precision computation, however, that does not work, because $s = x$ exactly. What we want is a value for s with half as many bits as x.

Because x is inside the range $(1, 3)$, it requires two bits for the integer part. If there are t bits in the significand, then if we add and subtract $2 \times 2^{\lceil t/2 \rceil} = 2^{\lceil (t+2)/2 \rceil}$, we can produce s with the desired number of bits.

For example, for IEEE 754 32-bit arithmetic, $t = 24$, so the addend is $2^{13} = 8192$. We can compute $s = x + 8192$ followed by $s = s - 8192$ to get s representable in 12 or fewer bits, provided that higher intermediate precision is avoided.

[R4] $|x|$ in $[\mathbf{1/0.35}, X_4)$:

$$z = 1/x^2,$$

$$\operatorname{erfc}(x) = \begin{cases} (1/x)\exp(-x^2 - 9/16 + \mathcal{R}_4(z)), & \textit{for } x > 0, \\ 2 - (1/x)\exp(-x^2 - 9/16 + \mathcal{R}_4(z)), & \textit{for } x \textit{ in } (-6, 0), \\ 2 - \texttt{TINY}, & \textit{for } x \textit{ in } (-\infty, -6], \end{cases}$$

$$\operatorname{erf}(x) = \begin{cases} \operatorname{sign}(x)(1 - \operatorname{erfc}(x)), & \textit{for } x \textit{ in } [1/0.35, C_4), \\ \operatorname{sign}(x)(1 - \texttt{TINY}), & \textit{for } x \textit{ in } [C_4, X_4). \end{cases}$$

The exponentials are each computed as the product of two exponentials, as in region **R3**, and the bit-trimming addend must be matched to the value of $|x|$.

The value of C_4 is chosen such that $\operatorname{fl}(\operatorname{erf}(C_4)) = 1$ to machine precision. As we noted earlier, that happens for values between 4 and 9, depending on the precision. To be independent of precision, we choose $C_4 = 9$ on all systems. It will only require adjustment when new arithmetics extend the precision range beyond that of the IEEE 754 128-bit format.

The value `TINY` must be small enough that $\operatorname{fl}(1 - \texttt{TINY}) = 1$ to machine precision in default rounding, and also that $\operatorname{fl}(\texttt{TINY} \times \texttt{TINY})$ underflows to zero. A suitable choice is the smallest normalized floating-point number, which is standardly available in the C language as the constants `FLT_MIN`, `DBL_MIN`, and `LDBL_MIN` in the standard header file `<float.h>`.

The code computes $1 - \texttt{TINY}$ and $2 - \texttt{TINY}$ rather than just returning the values 1 or 2, so that the IEEE 754 *inexact* exception flag is set, and nondefault rounding modes are properly accommodated. The variable `TINY` is declared with the `volatile` modifier, and a call to `STORE(&TINY)` is executed at function entry, to prevent compile-time evaluation of expressions involving `TINY`.

The upper limit X_4 could be chosen flexibly so that $\operatorname{fl}(\operatorname{erf}(X_4)) = 1$ for each machine precision. However, that would mean not only separate limits for each host precision, but also separate polynomials. The fdlibm choice is $X_4 = 28$, which is suitable for IEEE 754 64-bit arithmetic. For the `mathcw` library, we choose a value $X_4 = 107$ near the limit for which $\operatorname{erfc}(x)$ underflows in IEEE 754 128-bit arithmetic. Smaller values of X_4 are undesirable, because they would result in premature underflow of $\operatorname{erfc}(x)$.

[R5] $|x|$ in $[X_4, \infty)$:

$$\operatorname{erf}(x) = \operatorname{sign}(x)(1 - \texttt{TINY}),$$

$$\operatorname{erfc}(x) = \begin{cases} \texttt{TINY}^2, & \textit{for } x > 0, \\ 2 - \texttt{TINY}, & \textit{for } x < 0. \end{cases}$$

As in region 4, computing $\operatorname{fl}(2 - \texttt{TINY})$ with default rounding results in 2 with the *inexact* exception flag set. Computing $\operatorname{fl}(\texttt{TINY} \times \texttt{TINY})$ results in 0 with the IEEE 754 *inexact* and *underflow* flags set. Other rounding modes are also handled properly.

Whenever the computation of a function is split into multiple regions, there is always the risk that important properties, such as symmetry and monotonicity, are lost at the region boundaries. Tests of our algorithms for $\operatorname{erf}(x)$ and $\operatorname{erfc}(x)$ examined the function values at argument intervals of one ulp for 128 ulps on either side of each region boundary, and found no loss of monotonicity.

The errors in our implementations of the error functions are shown in **Figure 19.3** on the facing page and **Figure 19.4** on page 600. They show that the ordinary error function is almost always correctly rounded. The approximations used for the complementary error functions in regions 3 and 4 suffer from two or more rounding errors, and it appears difficult to reduce those errors without higher intermediate precision.

19.2 Scaled complementary error function

The `ERFCS(x)` family provides $\operatorname{erfcs}(x) = \exp(x^2) \times \operatorname{erfc}(x)$, the scaled complementary error function needed by Fortran 2008. The algorithm in the file `erfcsx.h` is similar to that in `erfcx.h`, with these changes:

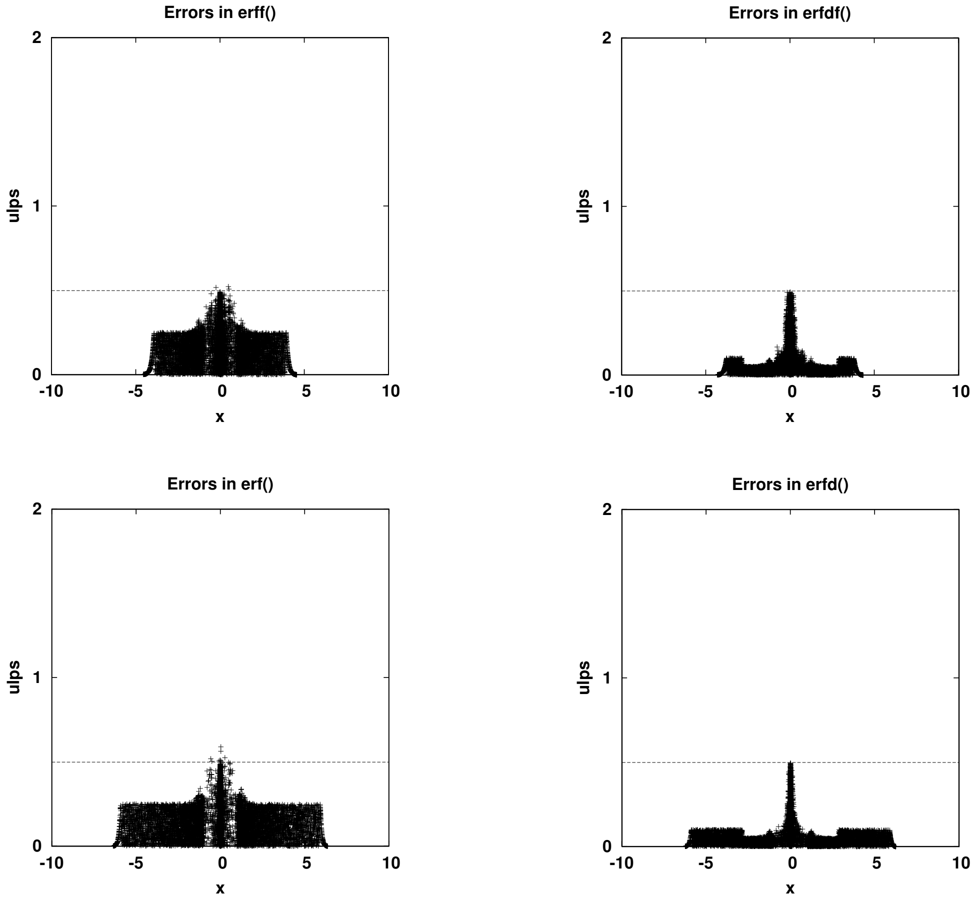


Figure 19.3: Errors in the ordinary error functions.

- For positive arguments, the complications of accurate computation of the factor $\exp(x^2)$ in regions **R3** and **R4** are eliminated.

- The cutoff X_4 is reduced to 10, shortening region **R4**.

- Region **R5** is reduced to $[X_4, X_5)$ with $X_5 = \beta^{\lceil t/2 \rceil}$, and its algorithm is replaced by a Lentz-style continued fraction. Numerical experiments show that at most 75 terms are needed to recover 70 decimal digits in the function value, and usually, many fewer suffice.

- A new region **R6** for x in $[X_5, \infty)$ is handled by a two-term continued fraction, $\epsilon(\sqrt{1/\pi}/(\epsilon x + \frac{1}{2}\epsilon/x))$. The exact scaling by the machine epsilon moves the parenthesized expression away from the IEEE 754 subnormal region, preventing accuracy loss, and premature underflow when subnormals are not supported.

- In regions **R5** and **R6**, the factor $\sqrt{1/\pi}$ is represented by a two-part split, and the final product is computed with a fused multiply-add operation.

- For improved accuracy in regions **R5** and **R6**, the continued fraction is computed as $1/(x + y) = u + v$, with $u = \mathrm{fl}(1/(x + y))$, so that the product with the split $\sqrt{1/\pi}$ is effectively computed in higher precision. That technique reduces the relative errors by 1 to 2 ulps.

The scaled complementary error function is representable for negative arguments only until $2\exp(x^2)$ overflows, but it is computable for all positive floating-point arguments. The computation is fast for large arguments, and rarely

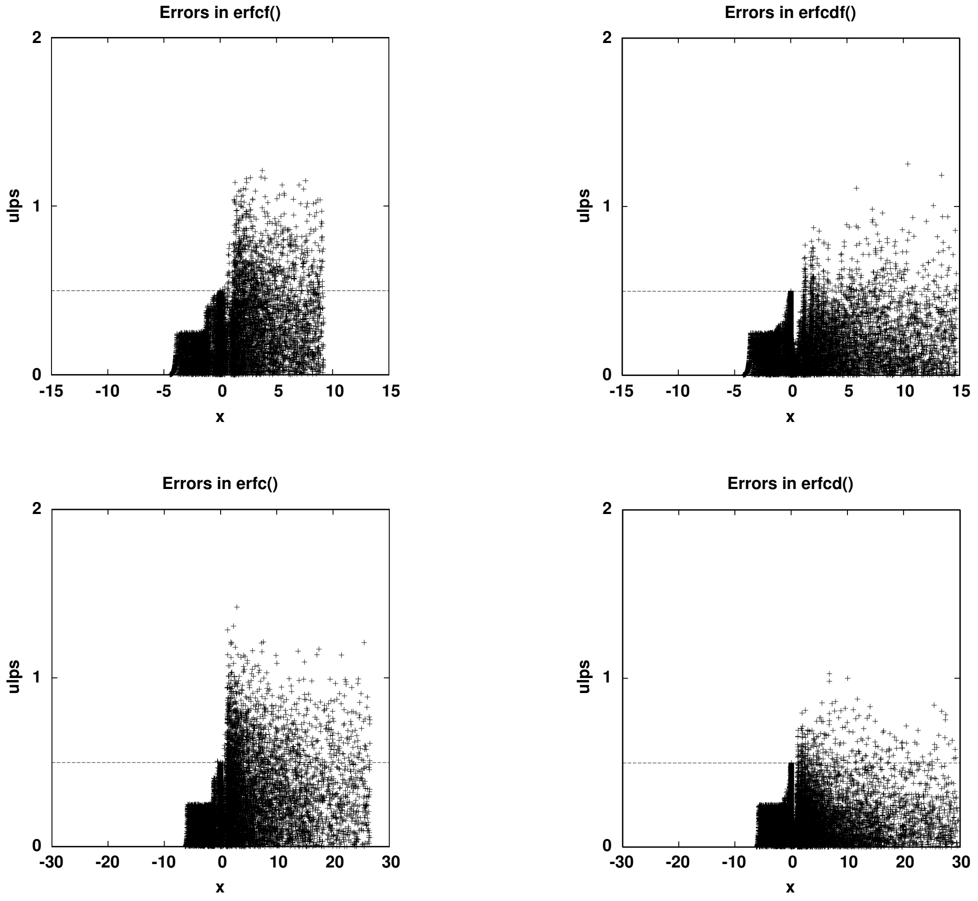


Figure 19.4: Errors in the complementary error functions.

suffers more than one or two rounding errors. Figure 19.5 on the facing page shows the errors in the 64-bit binary and decimal functions.

19.3 Inverse error functions

The inverse error functions are defined by

$$\operatorname{ierf}(\operatorname{erf}(x)) = x, \qquad \operatorname{ierfc}(\operatorname{erfc}(x)) = x.$$

A literature search found only a few references to their computation [Has55, Phi60, Car63, Str68, Fet74, BEJ76, Sch78, Wic88, Pop00, VS04], plus an interesting recent one on the general problem of the computation of inverses of functions [Dom03]. There is also an algorithm given in Moshier's book [Mos89, §5.14.4].

The inverse error functions are not often found in mathematical libraries or in symbolic-algebra systems, despite their applications in probability and statistics. We look at their use in those subjects in Section 19.4 on page 610.

The inverse error functions are mentioned only briefly in the *Handbook of Mathematical Functions* [AS64, Eqn. 26.2.22, 26.2.23, and 26.2.49] and not, as expected, in the chapter on error functions, but instead in the far-away chapter on probability functions. They appear there primarily as low-accuracy rational approximations to solutions of equations involving opaquely named functions $P(x)$ and $Q(x)$. They also receive brief treatment in the later edition [OLBC10, §7.17].

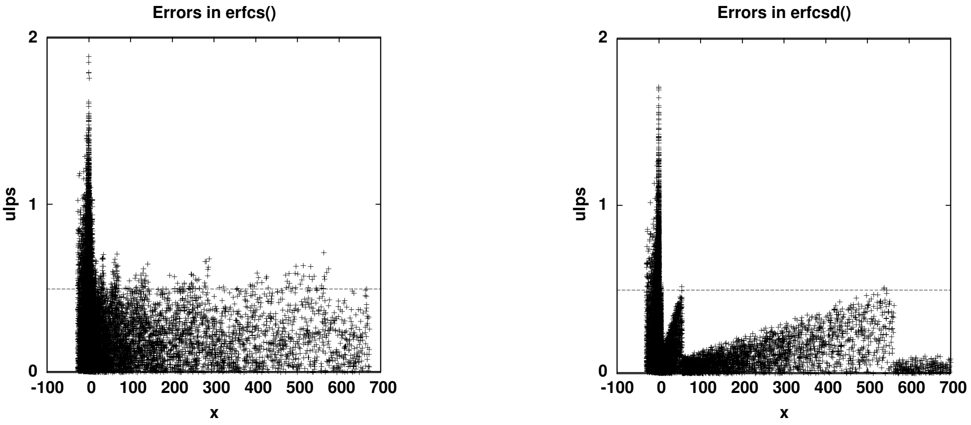


Figure 19.5: Errors in the scaled complementary error functions.

The S-Plus and R statistics programming languages have neither of the inverse error functions, although they have related functions that we treat in **Section 19.4** on page 616. MATLAB supplies both of the inverse error functions, with the names `erfinv()` and `erfcinv()`. Mathematica also has both, but with the names `InverseErf[]` and `InverseErfc[]`.

19.3.1 Properties of the inverse error functions

Mathematica can compute the series expansion of the inverse error function, but not the complementary one:

```
% math
In[1]:= Series[InverseErf[x], {x, 0, 7}]

                       3/2  3       5/2  5         7/2  7
        Sqrt[Pi] x   Pi    x    7 Pi    x    127 Pi    x        8
Out[1]= ---------- + -------- + ---------- + ------------ + O[x]
            2           24         960          80640

In[2]:= Series[InverseErfc[x], {x, 1, 2}]

Out[2]= InverseErf[Infinity, -x]
```

However, recall from **Section 2.10** on page 20 that Mathematica knows how to invert series, so we can find the expansion of the complementary inverse error function another way:

```
In[3]:= InverseSeries[Series[Erfc[x], {x, 0, 10}]]

                                 3/2         3       5/2         5
        -(Sqrt[Pi] (-1 + x))   Pi    (-1 + x)    7 Pi    (-1 + x)
Out[3]= -------------------- - --------------- - ----------------- -
                 2                   24                 960

           7/2         7          9/2         9
     127 Pi    (-1 + x)    4369 Pi    (-1 + x)              11
     ------------------- - -------------------- + O[-1 + x]
           80640                11612160
```

The functions as implemented in MATLAB are graphed in **Figure 19.6** on the following page, from which we can

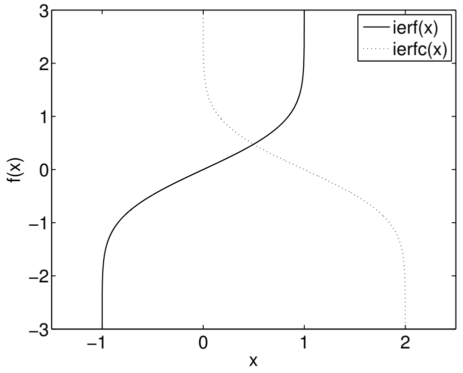


Figure 19.6: Inverse error functions. Both functions go to ±∞ at the endpoints of their intervals, but even in 128-bit IEEE 754 binary arithmetic, ierf(x) only reaches about ±8.65 at arguments just one machine epsilon away from those endpoints. Its companion, ierfc(x), is about 106.54 at the minimum normal number, and about 106.90 at the minimum subnormal number.

identify some more useful relations:

$$\begin{aligned}
\operatorname{ierf}(x) &= -\operatorname{ierf}(-x), && \textit{for } x \textit{ in } (-1, +1),\\
\operatorname{ierfc}(1+x) &= -\operatorname{ierfc}(1-x), && \textit{for } x \textit{ in } (-1, +1),\\
\operatorname{ierfc}(x) &= -\operatorname{ierfc}(2-x), && \textit{for } x \textit{ in } (0, +2],\\
\operatorname{ierf}(x) &= -\operatorname{ierfc}(1+x), && \textit{for } x \textit{ in } (-1, +1),\\
\operatorname{ierfc}(x) &= \operatorname{ierf}(1-x), && \textit{for } x \textit{ in } (0, +2).
\end{aligned}$$

Although those identities might suggest that only one of the functions need be computed, that is not the case. When x is small, fl($1 \pm x$) loses trailing bits, and for even smaller x, all bits of x are lost. It is then better to compute ierfc($1 + x$) from − ierf(x), and ierf($1 - x$) from ierfc(x).

Maple has neither of the inverse error functions, and it is not even immediately obvious how to define them computationally. Fortunately, a Web search found several exchanges on the Maple mailing list from the summer of 2000 that included an unexpected, and simple, way to define them symbolically:

```
% maple
...
## Maple arrow (map) one-line function definitions:
> ierf  := x -> solve(erf(y)  = x, y):
> ierfc := x -> solve(erfc(y) = x, y):
## Alternate Maple syntax:
> ierf2  := proc(x) local y: return solve(erf(y)  = x, y) end proc:
> ierfc2 := proc(x) local y: return solve(erfc(y) = x, y) end proc:
```

Despite the definition of the inverse error functions as the computationally difficult solutions of nonlinear equations, Maple is able to report their Taylor series expansions:

```
>  taylor(ierf(x), x = 0, 7);
  1/2       3/2          5/2
Pi        Pi     3   7 Pi     5      7
----- x + ----- x  + ------- x  + O(x )
  2        24         960



> taylor(ierfc(x), x = 1, 7);
```

```
      1/2             3/2                5/2
    Pi              Pi           3   7 Pi           5            7
  - ----- (x - 1) - ----- (x - 1)  - ------- (x - 1)  + O((x - 1) )
      2              24                960
```

Notice that the ierfc(x) is expanded about $x = 1$, *not* $x = 0$, where it has a pole.

Maple can also find derivatives of the error functions and their inverses:

```
  > diff(erf(x), x);
                              2
                    2 exp(-x )
                    ----------
                        1/2
                      Pi

  > diff(erfc(x), x);
                               2
                     2 exp(-x )
                   - ----------
                         1/2
                       Pi

  > diff(ierf(x), x);
                           1/2
                         Pi
          1/2 ---------------------------
                                         2
              exp(-RootOf(-erf(_Z) + x) )

  > diff(ierfc(x), x);
                           1/2
                         Pi
         -1/2 ----------------------------
                                          2
              exp(-RootOf(-erfc(_Z) + x) )
```

The `RootOf()` function in the output needs some explanation. When Maple produces `RootOf(f(_Z))`, it stands for a root _Z of the equation $f(\_Z) = 0$. Here, we have $-\operatorname{erfc}(\_Z) + x = 0$, which has the solution $\_Z = \operatorname{ierfc}(x)$. Thus, the derivatives of the inverse error functions are

$$\begin{aligned}\operatorname{ierf}'(x) &= +\sqrt{\pi/4}\exp\left((\operatorname{ierf}(x))^2\right),\\ \operatorname{ierfc}'(x) &= -\sqrt{\pi/4}\exp\left((\operatorname{ierfc}(x))^2\right).\end{aligned}$$

With the derivatives, we can compute the error-magnification factors discussed in **Section 4.1** on page 61 and tabulated in **Table 4.1** on page 62, and graph them with MATLAB's implementations of the inverse error functions, as shown in **Figure 19.7** on the next page. As **Figure 19.6** on the facing page suggests, ierf(x) is sensitive to argument errors near the poles.

For ierfc(x), the factor x in the error magnification eliminates the argument-error sensitivity near the left pole at $x = 0$, but there is large sensitivity near $x = 1$ and $x = 2$.

19.3.2 Historical algorithms for the inverse error functions

Strecok's work [Str68] on the inverse error functions predated most symbolic-algebra systems. To obtain the series expansions of the inverse error functions, he first derived an integro-differential equation relating the inverse error function to its derivative:

$$-1/\operatorname{ierf}'(x) = 2\int_0^x \operatorname{ierf}(t)\,dt - 2/\sqrt{\pi}.$$

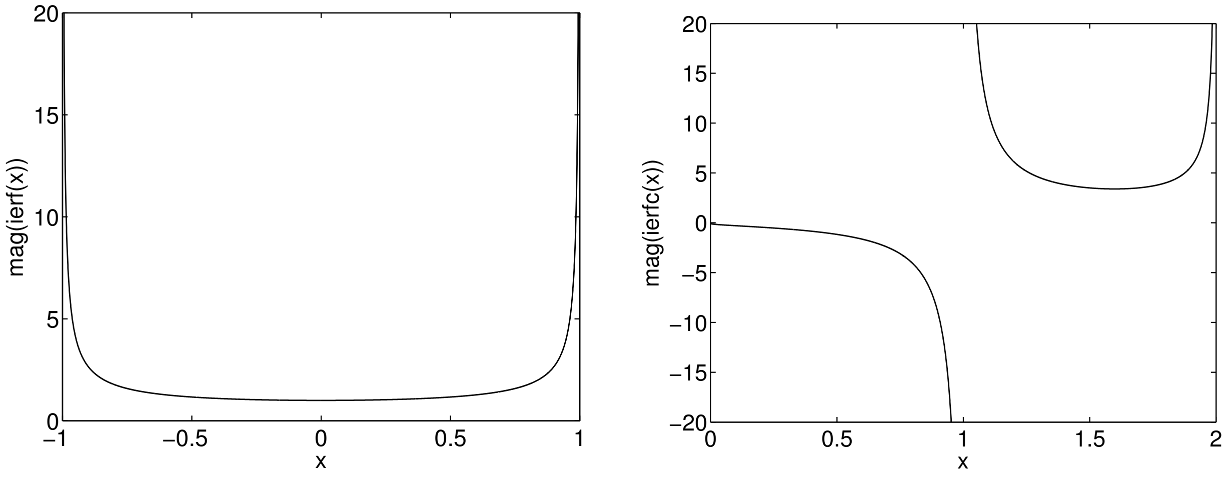


Figure 19.7: Error-magnification factors for the inverse error functions as implemented in MATLAB.

He then expanded ierf(x) in an infinite series with unknown coefficients, substituted the series into the integro-differential equation, and solved for the coefficients. In his article, he tabulated the coefficients of the first 200 terms to 25 decimal digits, along with coefficients for a 38-term Chebyshev polynomial economization of comparable accuracy.

Now, thanks to Maple, we can easily reproduce the numbers in his Table 2 with just two statements:

```
> Digits := 25:
> evalf(convert(convert(series(ierf(x), x = 0, 400), polynom),
>               horner));
(0.8862269254527580136490835 + (0.2320136665346544935535339 + (

0.1275561753055979582539996 + (0.08655212924154753372964164 + (

0.06495961774538541338201449 + (0.05173128198461637411263173 + (
...
0.001023171163866410634078506 + (0.001017612474788160882928580

 + (0.001012112075399955109701667 + 0.001006669063824750191511020

  2   2   2
 x ) x ) x )
...
```

The final coefficients differ from Strecok's in the last digit. To resolve that discrepancy, a second run with `Digits` set to 35 showed that the last three or four digits in Maple's output are untrustworthy,[1] and that Strecok's coefficients can be off by one in the last digit. However, we do not use that expansion, except to demonstrate agreement with prior work.

Later, Popov [Pop00] used three different rational polynomial approximations of degree $\langle 5/2 \rangle$, plus an improved logarithmic form for $x \approx 1$, as starting values for iterations based on higher-order Newton–Raphson-like schemes, obtaining eight correct decimal digits after just one iteration.

MATLAB computes the inverse error functions from rational polynomials that depend on the argument range, producing initial estimates that are correct to about nine decimal digits. It then applies one iteration of Halley's

[1] Maple's multiple-precision arithmetic system was replaced at version 9 with a faster implementation, but alas, accuracy of the last few digits suffered. In version 8 and earlier, the internal precision was slightly higher than the `Digits` value, so that output values were normally correct to the last displayed digit.

method (see **Section 2.4** on page 9) to produce results that are accurate to IEEE 754 double precision.

19.3.3 Computing the inverse error functions

We could prepare an analogue of the algorithm developed for the error function in **Section 19.1** on page 593, adapting it for computation of the inverse error function, with separate polynomial approximations in each of several regions. However, we first investigate an alternative approach: use Newton–Raphson iteration to find $\mathrm{ierf}(x)$ from $\mathrm{erf}(x)$ and $\mathrm{erf}'(x)$, and similarly for $\mathrm{ierfc}(x)$, just as we did for the square root in **Section 8.1** on page 215. That is, we seek a solution y of

$$f(y) = \mathrm{erf}(y) - x = 0, \qquad \text{for constant } x.$$

The solution y is, by definition of the inverse, $\mathrm{ierf}(x)$. Of course, the almost-vertical behavior near the poles (see **Figure 19.6** on page 602) means that the inverse function cannot be computed accurately there from a numerical solution of that equation without higher-precision values of the error function.

The Newton–Raphson formula says that starting from a suitable approximate value, y_0, the iteration

$$\begin{aligned} y_{n+1} &= y_n - f(y_n)/f'(y_n) \\ &= y_n - (\mathrm{erf}(y_n) - x)/\,\mathrm{erf}'(y_n) \end{aligned}$$

produces the desired solution y, and it does so with quadratic convergence. It is also *self correcting*: errors do not accumulate from one iteration to the next.

To use the Newton–Raphson iteration, we need good starting estimates of the inverse error functions, so that we can converge rapidly to a solution. One possibility is to use a polynomial approximation to $\mathrm{ierf}(x)$, but we can see from **Figure 19.6** on page 602 that the poles prevent use of a single polynomial over the needed range of x. Maple easily finds an 8-bit $\langle 1/1 \rangle$ approximation to $\mathrm{ierf}(x)$ for x in $[0, 0.6]$, and a 7-bit $\langle 1/2 \rangle$ approximation to $1/\,\mathrm{ierf}(x)$ for x in $[0.6, 0.99]$, but the region $[0.99, 1]$ is not easy to represent by a low-order rational polynomial. Another possibility is to use linear interpolation in a table of precomputed function values.

Fortunately, we can avoid the complication of multiple polynomial fits and table lookup, because Strecok found a simple and useful approximation to $\mathrm{ierf}(x)$:

$$\mathrm{ierf}(x) \approx \sqrt{-\log\big((1-x)(1+x)\big)}, \qquad \text{for } x \text{ in } [0,1).$$

Strecok then used that approximation to define a correcting polynomial approximation, $\mathcal{R}(x)$, such that

$$\mathrm{ierf}(x) \approx \mathcal{R}(x)\sqrt{-\log\big((1-x)(1+x)\big)}.$$

provides a highly accurate procedure for computing $\mathrm{ierf}(x)$. He used three different polynomials, $\mathcal{R}(x)$, of degree as high as 37, depending on the argument region, with coefficients given to 25 decimal digits. That gave a maximum error in $\mathrm{ierf}(\mathrm{erf}(x))/x - 1$ of about 10^{-22}, and in $(1 - \mathrm{erf}(\mathrm{ierf}(1-x)))/x - 1$, a maximum error of about 10^{-19}. However, for the mathcw library, we want to extend the accuracy of all functions to about 75 decimal figures, so approximations of much higher degree could be required, even though using rational polynomials would reduce the maximum degree needed.

Plots of the inverse error function and its approximation suggest that introduction of a simple scale factor would improve the fit, as shown in **Figure 19.8** on the following page. The scale factor 0.9 in the figure was chosen from a few numerical experiments.

An optimal fit would minimize the absolute area between the function and its approximation, or alternatively, the square of the difference between the two. That should be straightforward in Maple, taking care to avoid infinities at the poles:

```
> app_ierf := x -> sqrt(-log((1 - x)*(1 + x))):
> h := proc(s)
    local x;
    return evalf(Int((evalf(ierf(x)) - s * evalf(app_ierf(x)))^2, x = 0.01 .. 0.99))
  end proc:
> minimize(h(s), s = 0.8 .. 1, location = true);
```

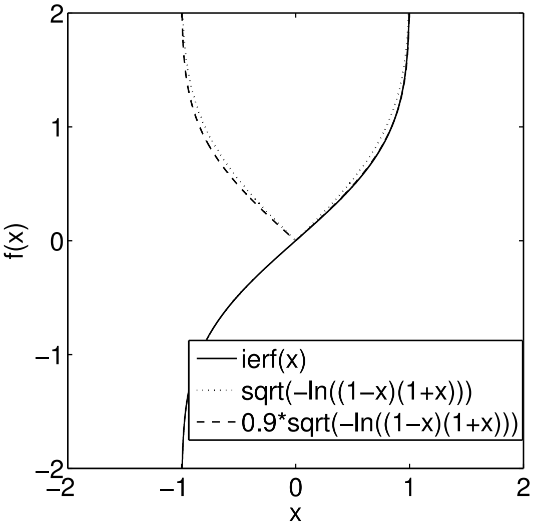


Figure 19.8: Inverse error function and two approximations to it. The curves on the negative axis are close if we negate the approximating functions in that region.

Unfortunately, that failed because Maple returned a symbolic value for the integral involving the `RootOf()` function that we described earlier.

The next attempt used Simpson's rule quadrature for numerical integration:

```
> with(student,simpson):
> h := proc(s)
           local x;
           return evalf(simpson((ierf(x) - s * app_ierf(x))^2, x = 0.01 .. 0.99))
       end proc:
> minimize(h(s), s = 0.8 .. 1.0, location = true);
    0.0001215565363, {[{s = 0.9064778675}, 0.0001215565363]}
```

That gives a better idea of what the scale factor, s, should be. Further plots showed that the deviation between the two functions increases sharply as $x \to +1$, suggesting that there is no point in working hard to minimize a difference that is unavoidably large anyway. We therefore settle on fitting a smaller interval, and improve the quadrature accuracy by requesting many more rectangle subdivisions than the default of four:

```
> Digits := 20:
> h := proc(s)
           local x;
           return evalf(simpson((ierf(x) - s * app_ierf(x))^2, x = 0 .. 0.9, 1024))
       end proc:
> minimize(h(s), s = 0.8 .. 1.0, location = true);
                                  -5
  0.66872028701973500778 10  ,


                                                                -5
  {[{s = 0.89470647317266353771}, 0.66872028701973500778 10  ]}
```

Based on the Maple optimization, we select $s = 0.8947$ to get an approximation to $\operatorname{ierf}(x)$ that is within about 0.2% of the expected result for x in the range $[0, 0.9]$. The relative error in that approximation is graphed in **Figure 19.9**.

More recently, Van Eetvelt and Shepherd [VS04] developed four-digit approximations to $\operatorname{erf}(x)$ and $\operatorname{ierf}(x)$. Although their results are not of sufficient accuracy to be of interest here, those authors observed that the graphs of $\operatorname{erf}(x)$ and $\operatorname{atanh}(x)$ are similar, so a scaled approximation of the form $\operatorname{erf}(x) \approx s \operatorname{atanh}(x)$ could be a useful starting point. In Maple, we find

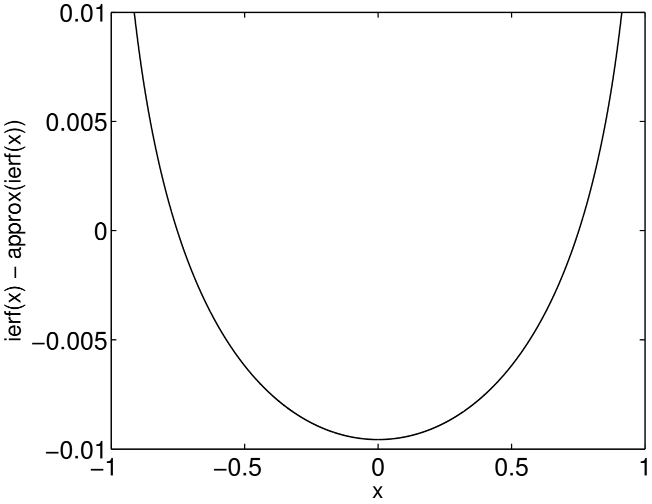


Figure 19.9: Relative error in approximation of inverse error function by $0.8947\sqrt{-\log((1-x)*(1+x))}\,\mathrm{sign}(x)$.

```
> h2 := proc(s)
    local x;
    return evalf(simpson((erf(x) - s * tanh(x))^2, x = 0 .. 3, 1024))
  end proc:
> minimize(h2(s2), s2 = 0.9 .. 1.2, location = true);
        0.002943920301, {[{s2 = 1.047318936}, 0.002943920301]}
```

The inverse functions of each should also be similar, and we find a scaled approximation from Maple:

```
> h3 := proc(s)
    local x;
    return evalf(simpson((ierf(x) - s * arctanh(x))^2, x = 0 .. 0.999, 1024))
 end proc:
> minimize(h3(s3), s3 = 0.5 .. 1.5, location = true);
        0.003332120317, {[{s3 = 0.7810559327}, 0.003332120317]}
```

The computed minima in those results are not as small as we found for the scaled logarithmic approximation, so we do not pursue them further.

In Strecok's initial estimate of $\mathrm{ierf}(x)$, we observe that the argument of the approximating function loses trailing digits from the factors $(1 \pm x)$ as $x \to 0$. That is easily remedied by expanding the argument: $\log\big((1-x)(1+x)\big) = \log(1-x^2) = \mathrm{log1p}(-x^2)$. The latter function retains full accuracy.

We recall from **Figure 19.1** on page 594 that $\mathrm{erf}(x)$ flattens out quickly for $|x| > 2$, so we must ask whether its derivative, which appears in the denominator in the Newton–Raphson iteration, can be computationally zero. As the caption in **Figure 19.6** on page 602 notes, the argument range where the inverse error function is finite is sharply limited: we do not require $|y| > 8.65$ even in 128-bit IEEE 754 arithmetic. The smallest possible value of the derivative of the inverse error function is therefore about $\sqrt{4/\pi}\exp(-8.65^2) \approx 3.2 \times 10^{-33}$, which is larger than the underflow limit of current and historical floating-point systems. Thus, we need not check for a zero denominator in the iterations for $\mathrm{ierf}(x)$, provided that the special arguments $x = \pm 0, \pm 1, \pm\infty$, and NaN are handled separately.

For the inverse complementary error function, $\mathrm{ierfc}(x)$, arguments can be much smaller as the pole at $x = 0$ is approached. Although in IEEE 754 arithmetic, $\exp(-(\mathrm{ierfc}(\text{smallest normal number}))^2)$ is still a normal number, with subnormal arguments, the result of the exponential can be subnormal. On older floating-point architectures, or even modern ones with a deficient exponential function, the exponential could underflow to zero. Thus, for $\mathrm{ierfc}(x)$, a check for a zero divisor is required, even if it only rarely succeeds.

The numerator of the Newton–Raphson formula, $x - \mathrm{erf}(y_n)$, loses leading bits as we approach convergence. However, that is expected, because successive iterates y_n should not change much. There are, however, two concerns when we decide how to terminate the iterations. First, it is possible that the y_n values may sometimes oscillate by a few ulps, instead of converging to a single value. Second, when y_n is large, the denominator is tiny, so the correction term $(x - \operatorname{erf}(y_n)) / \operatorname{erf}'(y_n)$ could be quite large.

We resolve those issues by terminating the iteration if the absolute value of the difference of two successive iterates fails to decrease, and using a starting approximation that ensures that a small denominator is accompanied by a small numerator, so that the correction to y_n is of modest size.

Numerical experiments reveal another problem, however: when $x \to 0$, the computed solution is less accurate than is desirable, and it also fails to obey the reflection rule, $\operatorname{ierf}(x) = -\operatorname{ierf}(-x)$. That problem is solved by avoiding the Newton–Raphson iterations for small $|x|$, and using the Taylor series instead, with a cutoff chosen so that the last term summed is less than $\epsilon/4$, for correct rounding. From Maple output with a few more terms than shown on page 602, we can simplify the results by removing common factors, producing this Taylor series:

$$\begin{aligned}\operatorname{ierf}(x) = x\sqrt{\pi}\big(&1/2 + (\pi/24)x^2 + (7\pi^2/960)x^4 + (127\pi^3/80\,640)x^6 + \\ &(4369\pi^4/11\,612\,160)x^8 + \mathcal{O}(x^{10})\big).\end{aligned}$$

As with other functions in the mathcw library, we accommodate hexadecimal floating-point arithmetic by normalizing the series so that its first term has no leading zero bits. That choice is harmless, and without additional cost, for other number bases.

Notice that all terms in the series are positive, and the terms fall off rapidly as the square of x. Thus, apart from requiring fractional coefficients, the series can be summed accurately.

However, the leading scale factor of $\sqrt{\pi}$ loses three bits in hexadecimal normalization, so on such a system, it should be computed as $2\sqrt{\pi/4}$ to avoid unnecessary bit loss, and the value $\sqrt{\pi/4}$ should be split into the sum of an exact high and approximate low part.

A final point to consider is *symmetry*: the inverse error function is antisymmetric about $x = 0$, and the inverse complementary error function is antisymmetric about $x = 1$. For $\operatorname{ierf}(x)$, it is a simple matter to compute the value only for $|x|$, and then invert the sign if x is negative. However, the relation $\operatorname{ierfc}(x) = -\operatorname{ierfc}(2 - x)$ requires more care to satisfy. That is because x is exactly representable, but $2 - x$ need not be exact, and the equality holds only when both are exact.

Numerical experiments shows that for random arguments logarithmically distributed in the range $[\epsilon, 1]$, only about 4.3% in IEEE 754 32-bit format, 1.9% in the 64-bit and 80-bit formats, and 0.9% in the 128-bit format, provide exact representation of both x and $2 - x$. On the 36-bit DEC PDP-10, about 4.0% of single-precision numbers have that property. The overhead of guaranteeing the expected symmetry is relatively small, and worth doing because preservation of mathematical properties is important.

MATLAB's implementation of $\operatorname{ierfc}(x)$ does *not* take account of the symmetry relations, and we can see a dramatic effect on accuracy by comparison with high-precision results from Maple. We need to be careful to choose exactly representable arguments, so expressions involving multiples of the machine epsilon are a good choice. MATLAB uses IEEE 754 double-precision arithmetic, so we know that its machine epsilon is 2^{-52}. Here are experiments in Maple, using numerical values copied from a MATLAB session to compute the error in ulps:

```
> eps := 2**(-52):

> ulp := proc (approx,exact)
>   return round(evalf(abs((approx - exact) / exact) / eps))
> end proc:

> ulp(5.80501868319345, ierfc(eps));
            3

> ulp(-5.80501867863269, ierfc(2 - eps));
    3538292

> ulp(1.967819077581127e-16, ierfc(1-eps));
    5081431

> ulp(-1.967819077581127e-16, ierfc(1+eps));
    5081431
```

```
> ulp(6.182080735938092e-11, ierfc(1 - 314159*eps));
    5081430

> ulp(-6.182080735938092e-11, ierfc(1 + 314159*eps));
    5081430

> ulp(4.61116923088750, ierfc(314159*eps));
        1

# compute ierfc(2 - x) as -ierfc(x)
> ulp(-4.61116922790890, -ierfc(314159*eps));
    2909116
```

Maple had trouble with the last one, incorrectly returning a complex value for $\operatorname{ierfc}(2 - 314\,159\epsilon)$. Increasing the number of digits to 500 did not repair the difficulty, so we used the mathematically equivalent $-\operatorname{ierfc}(314\,159\epsilon)$.

By contrast, our implementation has no such accuracy problems, and the same tests all report rounded errors of zero ulps.

Here is a summary of our algorithms for computing the inverses of the normal and complementary error functions, omitting the essential initial checks for special arguments:

- If $|x|$ is small, use a five-term Taylor series for both functions, and for slightly larger $|x|$, sum a nine-term series for $\operatorname{ierfc}(x)$.
- For $\operatorname{ierf}(x)$, if x is in $(-1, -\frac{1}{2})$, compute the result from $-\operatorname{ierfc}(1 + x)$. If x is in $(\frac{1}{2}, 1)$, find the result as $\operatorname{ierfc}(1 - x)$. In both cases, the argument expression is *exact*. The switch to the complementary function for $|x| \to 1$ improves accuracy, compared to the algorithms of the remaining steps.
- For x in $[-\frac{1}{2}, +\frac{1}{2}]$, use a rational polynomial fit suggested by the form of the Taylor series:

  $$\operatorname{ierf}(x) \approx 2x\sqrt{\pi/4}\left(\tfrac{1}{2} + x^2\mathcal{R}(x^2)\right)$$

  In this interval, the term $x^2\mathcal{R}(x^2)$ is never larger than about 0.0382, effectively adding about one decimal digit of precision.

  For $1 - x$ in $[-\frac{1}{2}, +\frac{1}{2}]$ (that is, x in $[\frac{1}{2}, \frac{3}{2}]$), compute $\operatorname{ierfc}(x)$ as $\operatorname{ierf}(1 - x)$ using the same polynomial fit.
- For x in $(0, \frac{1}{2})$ or $(\frac{3}{2}, 2)$, use a scaled version of Strecok's logarithmic formula to compute a starting value of $y \approx \operatorname{ierf}(|x|)$ or $y \approx \operatorname{ierfc}(|x|)$:

  ```
  y = FP(0.8947) * SQRT(-LOG1P(-x*x));
  ```
- Iterate the Newton–Raphson procedure to improve the estimate of the inverse so as to find a root of $f(y) = \operatorname{erf}(y) - |x| = 0$ or $f(y) = \operatorname{erfc}(y) - x = 0$. The core computation for $\operatorname{ierf}(x)$ is

  ```
  ynew = y - (ERF(y) - xabs)/ERF1(y);
  ```

  For $\operatorname{ierfc}(x)$, a check for a zero denominator is advisable:

  ```
  d = ERFC1(y);

  if (d == ZERO)
      break;

  ynew = y - (ERFC(y) - x) / d;
  ```

  Terminate the loop if `ynew` is the same as `y`, or the magnitude of their difference increases compared to the previous iteration, or a limit on the iteration count (we use 8) is reached.

- Alternatively, make use of the simple form of the second derivatives (see **Section 19.1.1** on page 595) to rewrite the cubicly convergent Halley formula (see **Section 2.4** on page 9) for ierf(x) like this:

  ```
  f = ERF(y) - xabs;
  ynew = y - f / (ERF1(y) + xabs * f);
  ```

  For ierfc(x), the essential computation looks like this:

  ```
  f = ERFC(y) - x;
  d = ERFC1(y) + x * f;

  if (d == ZERO)
      break;

  ynew = y - f / d;
  ```

  The additional cost of two floating-point operations is negligible, and convergence is improved.

- For ierf(x), if x is negative, invert the sign of the computed y. Then return the final value of y.

Our choice of the region in which the polynomial fit is used was made after finding larger-than-desired errors in an initial implementation that relied on the iterative schemes. The particular interval chosen eliminates the iterations for ierf(x), but not for ierfc(x). The measured errors in our implementations of the inverse error functions are shown in **Figure 19.10** and **Figure 19.11** on the next page.

19.4 Normal distribution functions and inverses

Statisticians often work with the *cumulative distribution function*, $\Phi(x)$, of the *standard normal distribution*, where Φ is the uppercase Greek letter *phi*. The normal curve and its cumulative distribution function are so important in the field of statistics that textbooks may devote a chapter or more to them and their applications; see, for example, Devore's popular book [Dev08b, Chapter 4], which also features a table of numerical values of $\Phi(x)$ on the inside covers, similar to our **Table 19.1** on page 612 and **Table 19.2** on page 613.

$\Phi(x)$ is defined as the area that lies to the *left* of x under the curve of the standard normal distribution:

$$\begin{aligned}\Phi(x) &= (1/\sqrt{2\pi}) \int_{-\infty}^{x} \exp(-t^2/2)\, dt \\ &= (1/\sqrt{\pi}) \int_{-\infty}^{x/\sqrt{2}} \exp(-s^2)\, ds \\ &= \tfrac{1}{2}(2/\sqrt{\pi}) \int_{-x/\sqrt{2}}^{\infty} \exp(-s^2)\, ds \\ &= \tfrac{1}{2}\operatorname{erfc}(-x/\sqrt{2}).\end{aligned}$$

That is sometimes called the *lower-tail area*. From the equation, we solve for x, and then use it to find the inverse of $\Phi(x)$:

$$\begin{aligned}x &= -\sqrt{2}\operatorname{ierfc}(2\Phi(x)) \\ &= \Phi^{-1}(\Phi(x)), && \textit{by definition of the inverse}, \\ \Phi^{-1}(p) &= -\sqrt{2}\operatorname{ierfc}(2p), && \textit{by substituting } p = \Phi(x).\end{aligned}$$

Using p for the argument of the inverse function emphasizes that p is often a probability value in $[0, 1]$. The inverse function is sometimes called the *normal quantile function* or the *probit* function.

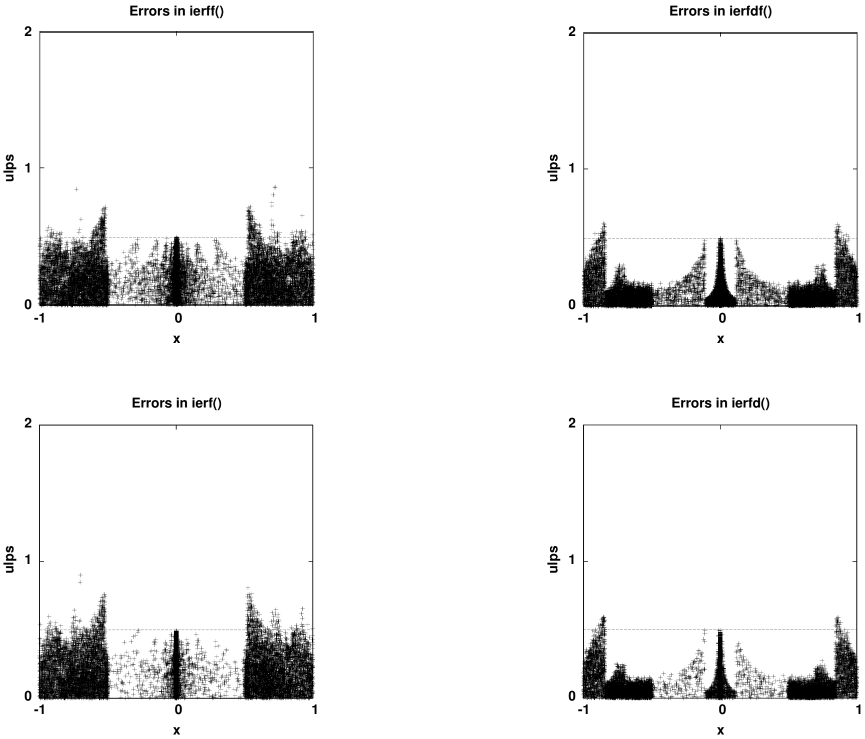


Figure 19.10: Errors in inverse error functions.

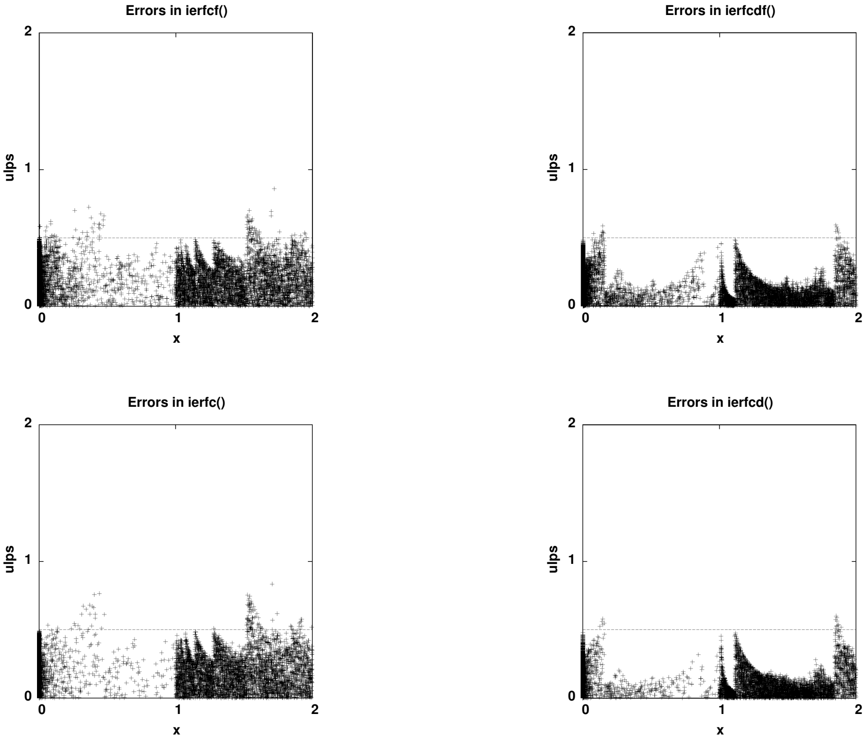


Figure 19.11: Errors in complementary inverse error functions.

Table 19.1: Cumulative distribution function of the standard normal distribution for $x \leq 0$. To find $\Phi(-2.25)$, for example, locate the row corresponding to the first fractional digit, and the column corresponding to the second fractional digit: the value at that row and column is **0.0122**. Linear interpolation between values in adjacent columns provides a reasonable approximation to $\Phi(x)$ for intermediate x values. The inverse function can be estimated as well: the argument of $\Phi^{-1}(0.3) \approx 0.5244$ lies between the shaded boxes in the thirtieth row.

	$\mathbf{\Phi(x)}$ **and** $\mathbf{\Phi_c(-x)}$									
x	**0.00**	**0.01**	**0.02**	**0.03**	**0.04**	**0.05**	**0.06**	**0.07**	**0.08**	**0.09**
−3.4	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0002
−3.3	0.0005	0.0005	0.0005	0.0004	0.0004	0.0004	0.0004	0.0004	0.0004	0.0003
−3.2	0.0007	0.0007	0.0006	0.0006	0.0006	0.0006	0.0006	0.0005	0.0005	0.0005
−3.1	0.0010	0.0009	0.0009	0.0009	0.0008	0.0008	0.0008	0.0008	0.0007	0.0007
−3.0	0.0013	0.0013	0.0013	0.0012	0.0012	0.0011	0.0011	0.0011	0.0010	0.0010
−2.9	0.0019	0.0018	0.0018	0.0017	0.0016	0.0016	0.0015	0.0015	0.0014	0.0014
−2.8	0.0026	0.0025	0.0024	0.0023	0.0023	0.0022	0.0021	0.0021	0.0020	0.0019
−2.7	0.0035	0.0034	0.0033	0.0032	0.0031	0.0030	0.0029	0.0028	0.0027	0.0026
−2.6	0.0047	0.0045	0.0044	0.0043	0.0041	0.0040	0.0039	0.0038	0.0037	0.0036
−2.5	0.0062	0.0060	0.0059	0.0057	0.0055	0.0054	0.0052	0.0051	0.0049	0.0048
−2.4	0.0082	0.0080	0.0078	0.0075	0.0073	0.0071	0.0069	0.0068	0.0066	0.0064
−2.3	0.0107	0.0104	0.0102	0.0099	0.0096	0.0094	0.0091	0.0089	0.0087	0.0084
−2.2	0.0139	0.0136	0.0132	0.0129	0.0125	**0.0122**	0.0119	0.0116	0.0113	0.0110
−2.1	0.0179	0.0174	0.0170	0.0166	0.0162	0.0158	0.0154	0.0150	0.0146	0.0143
−2.0	0.0228	0.0222	0.0217	0.0212	0.0207	0.0202	0.0197	0.0192	0.0188	0.0183
−1.9	0.0287	0.0281	0.0274	0.0268	0.0262	0.0256	0.0250	0.0244	0.0239	0.0233
−1.8	0.0359	0.0351	0.0344	0.0336	0.0329	0.0322	0.0314	0.0307	0.0301	0.0294
−1.7	0.0446	0.0436	0.0427	0.0418	0.0409	0.0401	0.0392	0.0384	0.0375	0.0367
−1.6	0.0548	0.0537	0.0526	0.0516	0.0505	0.0495	0.0485	0.0475	0.0465	0.0455
−1.5	0.0668	0.0655	0.0643	0.0630	0.0618	0.0606	0.0594	0.0582	0.0571	0.0559
−1.4	0.0808	0.0793	0.0778	0.0764	0.0749	0.0735	0.0721	0.0708	0.0694	0.0681
−1.3	0.0968	0.0951	0.0934	0.0918	0.0901	0.0885	0.0869	0.0853	0.0838	0.0823
−1.2	0.1151	0.1131	0.1112	0.1093	0.1075	0.1056	0.1038	0.1020	0.1003	0.0985
−1.1	0.1357	0.1335	0.1314	0.1292	0.1271	0.1251	0.1230	0.1210	0.1190	0.1170
−1.0	0.1587	0.1562	0.1539	0.1515	0.1492	0.1469	0.1446	0.1423	0.1401	0.1379
−0.9	0.1841	0.1814	0.1788	0.1762	0.1736	0.1711	0.1685	0.1660	0.1635	0.1611
−0.8	0.2119	0.2090	0.2061	0.2033	0.2005	0.1977	0.1949	0.1922	0.1894	0.1867
−0.7	0.2420	0.2389	0.2358	0.2327	0.2296	0.2266	0.2236	0.2206	0.2177	0.2148
−0.6	0.2743	0.2709	0.2676	0.2643	0.2611	0.2578	0.2546	0.2514	0.2483	0.2451
−0.5	0.3085	0.3050	0.3015	0.2981	0.2946	0.2912	0.2877	0.2843	0.2810	0.2776
−0.4	0.3446	0.3409	0.3372	0.3336	0.3300	0.3264	0.3228	0.3192	0.3156	0.3121
−0.3	0.3821	0.3783	0.3745	0.3707	0.3669	0.3632	0.3594	0.3557	0.3520	0.3483
−0.2	0.4207	0.4168	0.4129	0.4090	0.4052	0.4013	0.3974	0.3936	0.3897	0.3859
−0.1	0.4602	0.4562	0.4522	0.4483	0.4443	0.4404	0.4364	0.4325	0.4286	0.4247
−0.0	0.5000	0.4960	0.4920	0.4880	0.4840	0.4801	0.4761	0.4721	0.4681	0.4641

Table 19.2: Cumulative distribution function of the standard normal distribution for $x \geq 0$. Locate a value of the complementary function, $\Phi_c(-0.43) \approx 0.6664$, at the table entry for $\Phi(0.43)$.

	$\Phi(x)$ and $\Phi_c(-x)$									
x	**0.00**	**0.01**	**0.02**	**0.03**	**0.04**	**0.05**	**0.06**	**0.07**	**0.08**	**0.09**
0.0	0.5000	0.5040	0.5080	0.5120	0.5160	0.5199	0.5239	0.5279	0.5319	0.5359
0.1	0.5398	0.5438	0.5478	0.5517	0.5557	0.5596	0.5636	0.5675	0.5714	0.5753
0.2	0.5793	0.5832	0.5871	0.5910	0.5948	0.5987	0.6026	0.6064	0.6103	0.6141
0.3	0.6179	0.6217	0.6255	0.6293	0.6331	0.6368	0.6406	0.6443	0.6480	0.6517
0.4	0.6554	0.6591	0.6628	**0.6664**	0.6700	0.6736	0.6772	0.6808	0.6844	0.6879
0.5	0.6915	0.6950	0.6985	0.7019	0.7054	0.7088	0.7123	0.7157	0.7190	0.7224
0.6	0.7257	0.7291	0.7324	0.7357	0.7389	0.7422	0.7454	0.7486	0.7517	0.7549
0.7	0.7580	0.7611	0.7642	0.7673	0.7704	0.7734	0.7764	0.7794	0.7823	0.7852
0.8	0.7881	0.7910	0.7939	0.7967	0.7995	0.8023	0.8051	0.8078	0.8106	0.8133
0.9	0.8159	0.8186	0.8212	0.8238	0.8264	0.8289	0.8315	0.8340	0.8365	0.8389
1.0	0.8413	0.8438	0.8461	0.8485	0.8508	0.8531	0.8554	0.8577	0.8599	0.8621
1.1	0.8643	0.8665	0.8686	0.8708	0.8729	0.8749	0.8770	0.8790	0.8810	0.8830
1.2	0.8849	0.8869	0.8888	0.8907	0.8925	0.8944	0.8962	0.8980	0.8997	0.9015
1.3	0.9032	0.9049	0.9066	0.9082	0.9099	0.9115	0.9131	0.9147	0.9162	0.9177
1.4	0.9192	0.9207	0.9222	0.9236	0.9251	0.9265	0.9279	0.9292	0.9306	0.9319
1.5	0.9332	0.9345	0.9357	0.9370	0.9382	0.9394	0.9406	0.9418	0.9429	0.9441
1.6	0.9452	0.9463	0.9474	0.9484	0.9495	0.9505	0.9515	0.9525	0.9535	0.9545
1.7	0.9554	0.9564	0.9573	0.9582	0.9591	0.9599	0.9608	0.9616	0.9625	0.9633
1.8	0.9641	0.9649	0.9656	0.9664	0.9671	0.9678	0.9686	0.9693	0.9699	0.9706
1.9	0.9713	0.9719	0.9726	0.9732	0.9738	0.9744	0.9750	0.9756	0.9761	0.9767
2.0	0.9772	0.9778	0.9783	0.9788	0.9793	0.9798	0.9803	0.9808	0.9812	0.9817
2.1	0.9821	0.9826	0.9830	0.9834	0.9838	0.9842	0.9846	0.9850	0.9854	0.9857
2.2	0.9861	0.9864	0.9868	0.9871	0.9875	0.9878	0.9881	0.9884	0.9887	0.9890
2.3	0.9893	0.9896	0.9898	0.9901	0.9904	0.9906	0.9909	0.9911	0.9913	0.9916
2.4	0.9918	0.9920	0.9922	0.9925	0.9927	0.9929	0.9931	0.9932	0.9934	0.9936
2.5	0.9938	0.9940	0.9941	0.9943	0.9945	0.9946	0.9948	0.9949	0.9951	0.9952
2.6	0.9953	0.9955	0.9956	0.9957	0.9959	0.9960	0.9961	0.9962	0.9963	0.9964
2.7	0.9965	0.9966	0.9967	0.9968	0.9969	0.9970	0.9971	0.9972	0.9973	0.9974
2.8	0.9974	0.9975	0.9976	0.9977	0.9977	0.9978	0.9979	0.9979	0.9980	0.9981
2.9	0.9981	0.9982	0.9982	0.9983	0.9984	0.9984	0.9985	0.9985	0.9986	0.9986
3.0	0.9987	0.9987	0.9987	0.9988	0.9988	0.9989	0.9989	0.9989	0.9990	0.9990
3.1	0.9990	0.9991	0.9991	0.9991	0.9992	0.9992	0.9992	0.9992	0.9993	0.9993
3.2	0.9993	0.9993	0.9994	0.9994	0.9994	0.9994	0.9994	0.9995	0.9995	0.9995
3.3	0.9995	0.9995	0.9995	0.9996	0.9996	0.9996	0.9996	0.9996	0.9996	0.9997
3.4	0.9997	0.9997	0.9997	0.9997	0.9997	0.9997	0.9997	0.9997	0.9997	0.9998

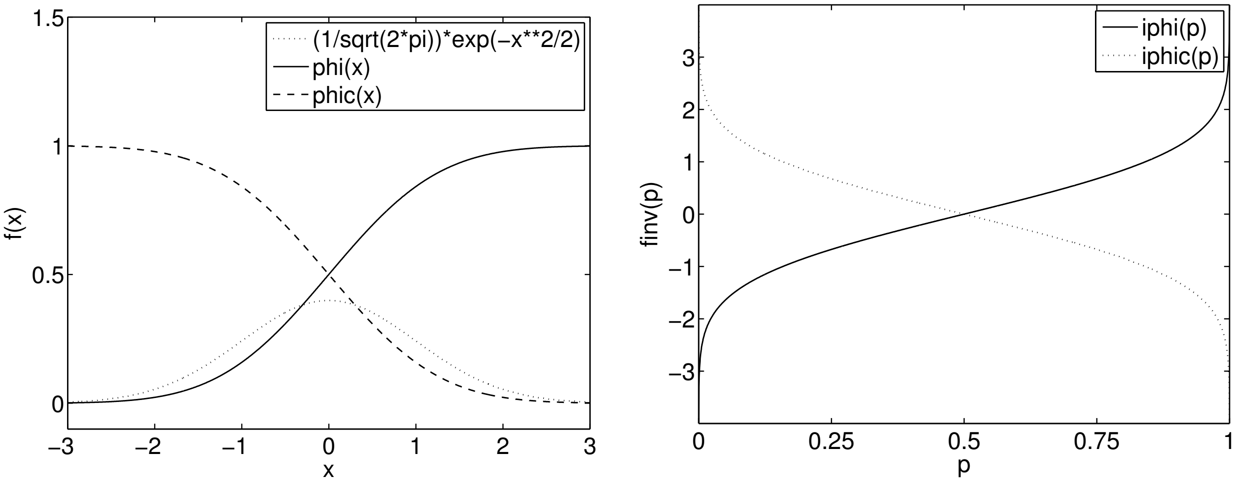


Figure 19.12: The cumulative distribution functions of the standard normal distribution, their inverses, and the famous bell-shaped curve of the standard normal distribution, as drawn by MATLAB.

The *complementary cumulative distribution function*, $\Phi_c(x)$, of the standard normal distribution is the area that lies to the *right* of x under the normal distribution curve:

$$\begin{aligned}\Phi_c(x) &= (1/\sqrt{2\pi}) \int_x^{\infty} \exp(-t^2/2)\,dt \\ &= (1/\sqrt{\pi}) \int_{x/\sqrt{2}}^{\infty} \exp(-s^2)\,ds \\ &= \tfrac{1}{2}\operatorname{erfc}(x/\sqrt{2}).\end{aligned}$$

That can be denoted the *upper-tail area*.

The inverse of $\Phi_c(x)$ has a simple relation to the inverse complementary error function:

$$\begin{aligned}\Phi_c^{-1}(\Phi_c(x)) &= x, && \textit{by definition of the inverse,} \\ &= \sqrt{2}\operatorname{ierfc}(2\Phi_c(x)), \\ \Phi_c^{-1}(p) &= \sqrt{2}\operatorname{ierfc}(2p), && \textit{by substituting } p = \Phi_c(x).\end{aligned}$$

The mathcw library provides those functions as `phi(x)`, `phic(x)`, `iphi(p)`, and `iphic(p)` for arguments of type `double`, with companions for other data types identified by the usual suffixes. As with the gamma function, $\Gamma(x) =$ `tgamma(x)`, their names are spelled in lowercase to conform to the conventions of the Standard C library. **Figure 19.12** shows graphs of $\Phi(x)$, $\Phi_c(x)$, and the standard normal curve.

The ordinary and inverse cumulative distribution functions of the standard normal distribution satisfy these important relations:

$$\begin{aligned}\Phi(x) + \Phi_c(x) &= 1, \\ \Phi(x) + \Phi(-x) &= 1, \\ \Phi_c(x) + \Phi_c(-x) &= 1, \\ \Phi(x) &= \Phi_c(-x), \\ \Phi^{-1}(p) + \Phi_c^{-1}(p) &= 0, \\ \Phi^{-1}(p) &= -\Phi_c^{-1}(p) \\ &= \Phi_c^{-1}(1-p) \\ &= -\Phi^{-1}(1-p),\end{aligned}$$

$$\begin{aligned}
\Phi^{-1}(\tfrac{1}{2} - d) &= -\Phi^{-1}(\tfrac{1}{2} + d), \\
\Phi_c^{-1}(\tfrac{1}{2} - d) &= -\Phi_c^{-1}(\tfrac{1}{2} + d), \\
\Phi_c^{-1}(\tfrac{1}{2} - d) &= \Phi^{-1}(\tfrac{1}{2} + d).
\end{aligned}$$

The ranges of the cumulative distribution functions are $[0, 1]$ for arguments in $(-\infty, +\infty)$. Arguments of the inverses of the cumulative distribution functions are restricted to $[0, 1]$, and the function ranges are $(-\infty, +\infty)$. The symmetries that are evident in **Figure 19.12** are helpful: if we can compute accurate values of $\Phi(x)$ for x in $(-\infty, 0]$ and $\Phi^{-1}(p)$ for p in $[0, \tfrac{1}{2}]$, then we can accurately determine the remaining values by sign inversion, or by subtraction from one.

The functions and their inverses have these Taylor series:

$$\begin{aligned}
\Phi(x) &= \tfrac{1}{2} + \frac{1}{\sqrt{2\pi}}(x - \frac{1}{6}x^3 + \frac{1}{40}x^5 - \frac{1}{336}x^7 + \frac{1}{3456}x^9 - \cdots), \\
\Phi_c(x) &= \tfrac{1}{2} - \frac{1}{\sqrt{2\pi}}(x - \frac{1}{6}x^3 + \frac{1}{40}x^5 - \frac{1}{336}x^7 + \frac{1}{3456}x^9 - \cdots), \\
\Phi^{-1}(\tfrac{1}{2} + d) &= +\sqrt{2\pi}(d + \frac{1}{3}\pi d^3 + \frac{7}{30}\pi^2 d^5 + \frac{127}{630}\pi^3 d^7 + \frac{4369}{22\,680}\pi^4 d^9 + \cdots), \\
\Phi_c^{-1}(\tfrac{1}{2} + d) &= -\sqrt{2\pi}(d + \frac{1}{3}\pi d^3 + \frac{7}{30}\pi^2 d^5 + \frac{127}{630}\pi^3 d^7 + \frac{4369}{22\,680}\pi^4 d^9 + \cdots).
\end{aligned}$$

For large positive arguments, the cumulative distribution functions have these asymptotic series, easily generated with the help of Maple's `asympt()` function:

$$\begin{aligned}
\Phi(x) &\asymp 1 - \sqrt{\frac{\exp(-x^2)}{2\pi}}(\frac{1}{x} - \frac{1}{x^3} + \frac{3}{x^5} - \frac{15}{x^7} + \frac{105}{x^9} - \frac{945}{x^{11}} + \frac{10\,395}{x^{13}} - \cdots), \\
\Phi_c(x) &\asymp \sqrt{\frac{\exp(-x^2)}{2\pi}}(\frac{1}{x} - \frac{1}{x^3} + \frac{3}{x^5} - \frac{15}{x^7} + \frac{105}{x^9} - \frac{945}{x^{11}} + \frac{10\,395}{x^{13}} - \cdots).
\end{aligned}$$

Factoring the exponential out of the square root shows that $\Phi_c(x)$ decays like $\exp(-x^2/2)$. In IEEE 754 32-bit arithmetic, that factor underflows to subnormals for $x \approx 13.2$, and the series must be summed up to the term containing x^{-9}. The corresponding limits for the 64-bit format are $x \approx 37.6$ and x^{-15}, and for the 80-bit and 128-bit formats, $x \approx 150.7$ and x^{-21}. Thus, even though the asymptotic series diverges (see **Section 2.9** on page 19), we can sum it to machine precision for most floating-point systems.

Statisticians use the notation z_p for the function $\Phi_c^{-1}(p)$. It is the $100(1-p)$-th percentile of the standard normal distribution. For example, $z_{0.05} = \Phi_c^{-1}(0.05) \approx 1.645$ is the value of x at the 95-th percentile: 95% of the area under the curve of the standard normal distribution lies to the left of $x \approx 1.645$, and 5% to the right.

If a random variable x has a normal distribution with mean μ and standard deviation σ, then the translated and scaled variable $r = (x - \mu)/\sigma$ has a *standard* normal distribution. That simple relation allows all normal distributions to be transformed to the standard one. The probability that x lies in the interval $[a, b]$ is then given by

$$P(a \le x \le b) = \Phi((b - \mu)/\sigma) - \Phi((a - \mu)/\sigma).$$

Similarly, for the standard normal distribution, the probability that r lies in the interval $[c, d]$ is

$$P(c \le r \le d) = \Phi(d) - \Phi(c).$$

From that, it follows that the probability that a normally distributed random value x exceeds the mean by at least n standard deviations is given by

$$\begin{aligned}
P((\mu + n\sigma) \le x \le \infty) &= \Phi(\infty) - \Phi\big(((\mu + n\sigma) - \mu)/\sigma\big) \\
&= 1 - \Phi\big(((\mu + n\sigma) - \mu)/\sigma\big) \\
&= \Phi_c\big(((\mu + n\sigma) - \mu)/\sigma\big) \\
&= \Phi_c(n).
\end{aligned}$$

Table 19.3: Probability in the normal distribution of exceeding the mean by n standard deviations.

n	0	1	2	3	4	5	6
$\Phi_c(n)$	0.5	0.159	0.0228	0.00135	3.17×10^{-05}	2.87×10^{-07}	9.87×10^{-10}

That probability is *independent of the mean and standard deviation*, and drops off rapidly, as Table 19.3 illustrates. The data in that table lead to the handy rules of thumb that only about *one in a thousand* normally distributed random values lies more than *three* standard deviations above the mean, and fewer than *one in a million* lie more than *five* standard deviations above.[2]

The S-Plus and R statistics programming languages supply $\Phi(x)$ as the function `pnorm(x)`. Their quantile function, `qnorm(p)`, computes $\Phi^{-1}(p)$. In the R language, $\Phi_c(x)$ can be obtained from `pnorm(x, lower.tail = FALSE)`, but there seems to be no standard function in S-Plus for that value.

Because of their simple relationship to the error function family, the cumulative distribution functions of the standard normal distribution can be computed directly from them, although there are pitfalls in doing so. Argument accuracy is slightly reduced in two of the four functions by the need to scale by $1/\sqrt{2}$, an irrational number, and thus, not exactly representable.

The identity $\Phi(x) = \frac{1}{2}\,\mathrm{erfc}(-x/\sqrt{2})$ and the graphs of the error magnification in Figure 19.2 on page 596 show that $\Phi(x)$ is sensitive to even that small scaling error for negative x of large magnitude, and $\Phi_c(x)$ has similar sensitivity when x is large and positive.

The graphs of the error magnification in Figure 19.7 on page 604 show that both $\Phi^{-1}(p)$ and $\Phi_c^{-1}(p)$ are sensitive to argument errors for p near $\frac{1}{2}$ and 1.

Tests of prototype implementations of the cumulative distribution functions and their inverses using their simple relations to the error-function family showed errors up to a few thousand ulps in regions of extreme argument sensitivity. However, such large errors are unacceptable in the mathcw library, so the current implementation does the computation in the next higher precision, and then coerces the final result to working precision, as illustrated by the code for computing $\Phi(x)$:

```
fp_t
PHI(fp_t x)
{
    fp_t result;

    if (ISNAN(x))
        result = SET_EDOM(QNAN(""));
    else if (ISINF(x))
        result = (x < ZERO) ? ZERO : ONE;
    else if (x < -ASYMPTOTIC_SERIES_CUTOFF)
        result = PHIC(-x);
    else
    {
        hp_t y;

        y = -HP_FMA(SQRT_HALF_HI, (hp_t)x, SQRT_HALF_LO * (hp_t)x);
        result = (fp_t)((hp_t)HALF * HP_ERFC(y));
    }

    return (result);
}
```

The function diverts to `PHIC(x)` when the asymptotic series can be effectively summed, avoiding code duplication, and ensuring accurate treatment of the small tail region.

[2] The phrase *six sigma* with 1.5σ drift has been used in advertising in the computer industry to indicate a product-quality goal of about three defects per million units manufactured ($\Phi_c(6 - 1.5) \approx 3.40 \times 10^{-6}$). Unfortunately, the advertisements often drop the 1.5σ bias, suggesting an unattainable defect rate of about one in a thousand million.

The inverse function, $\Phi^{-1}(p)$, requires more argument checking, and separate handling of the case of a hexadecimal base to reduce the impact of wobbling precision, because $1/\sqrt{2} \approx 0.707$ has no leading zero bits, whereas $\sqrt{2} \approx 1.414$ has three leading zero bits:

```
fp_t
IPHI(fp_t p)
{
    fp_t result;

    if (ISNAN(p))
        result = SET_EDOM(QNAN(""));
    else if (p < ZERO)
        result = SET_EDOM(QNAN(""));
    else if (p == ZERO)
        result = -INFTY();
    else if (p == ONE)
        result = INFTY();
    else if (p > ONE)
        result = SET_EDOM(QNAN(""));
    else
    {
        hp_t t;

        t = -HP_IERFC((hp_t)p + (hp_t)p);

#if B == 16
        t = HP_FMA(SQRT_HALF_HI, t, SQRT_HALF_LO * t);
        result = (fp_t)(t + t);
#else
        result = (fp_t)HP_FMA(SQRT_TWO_HI, t, SQRT_TWO_LO * t);
#endif

    }

    return (result);
}
```

The code for the complementary functions, `phic(x)` and `iphic(p)`, is similar to those, apart from an additional block in the former for summing the asymptotic series, so we omit it here.

Tests of the current implementations of the cumulative distribution function family for the standard normal distribution show that the results in single and double precision are, as expected, correctly rounded, so we omit the usual figure with plots of measured errors. However, the large errors remain in the functions for the highest available precision. To eliminate that problem, a more complex algorithm modeled on that for `erf()` and `erfc()` is needed, but we leave that for future work.

19.5 Summary

The ordinary and complementary error functions, their companions for the normal distribution, and their inverses, have broad applications, and deserve to be part of the mathematical libraries of most programming languages. Our implementations provide reasonable first drafts of those functions, but more work needs to be done to remove the need for higher precision in the code for $\Phi(x)$, $\Phi_c(x)$, $\Phi^{-1}(p)$, and $\Phi_c^{-1}(p)$, and to improve the accuracy of the inverse functions when their values are near the underflow limit.

Moshier's algorithm for $\Phi^{-1}(p)$ [Mos89, §5.14.4] is promising, but Maple is unable to produce fits to his approximating function, because it erroneously returns complex values for $\Phi^{-1}(p)$ when p is tiny. A proper solution may have to await fast and accurate implementations of the inverse error functions in Maple.

Although there are several publications in the statistics literature about the computation of the cumulative distribution functions and their inverses [Pól49, Bak61, Cyv64, Mac65, HJ67a, HJ67b, Ber68, Mac68, Ada69, OE74, BS77, Pag77, Ham78, Sch79a, Bai81, Sho82, Lin89, Lin90], they are of little help here, because they are of low accuracy, and because they completely ignore the small tail regions. However, one paper deserving of more study is that of George Marsaglia and co-workers [MZM94]; it claims single-precision accuracy, so perhaps its methods can be extended to higher precision.

20 Elliptic integral functions

> ELLIPTIC INTEGRAL, N.: AN INTEGRAL EXPRESSING
> THE LENGTH OF THE ARC OF AN ELLIPSE.
>
> — *New Century Dictionary* (1914).

The functions named in the title of this chapter have attracted the interest of several famous mathematicians, among them Abel, Euler, Gauss, Hermite, Jacobi, Kronecker, Lagrange, Legendre, Ramanujan, Riemann, and Weierstrass. Their properties are well-chronicled in several books, including [AS64, Chapter 17], [Law89], [OLBC10, Chapter 19], [Wal96], [Wei99], [GRJZ07, §8.1], and [JD08, Chapter 12], with extensive tables of properties and integrals of related functions in [BF71].

None of the elliptic integrals in this chapter has a simple closed form. For the mathcw library, in the first part of this chapter we implement only the easier single-argument complete functions, which can be computed by a method discovered by Gauss [AAR99, page 132] that involves the *arithmetic-geometric mean*, the subject of the first section of this chapter. We then show how iterative algorithms can calculate the more difficult multiple-argument incomplete elliptic integral functions. We finish the chapter with a discussion of the properties and computation of several other important functions in the large elliptic-integral family.

20.1 The arithmetic-geometric mean

The *arithmetic mean* of two numbers a and b is just their average, $\frac{1}{2}(a+b)$. Their *geometric mean* is the square root of their product, $\sqrt{ab}$, and to avoid complex numbers, we assume that a and b have the same sign. The two means usually differ, but they can be made to converge to a common value with an iteration of the following form, where we introduce a third value, c:

$$\begin{aligned} a_0 &= a, & b_0 &= b, & c_0 &= c, \\ a_j &= \tfrac{1}{2}(a_{j-1}+b_{j-1}), & b_j &= \sqrt{a_{j-1}b_{j-1}}, & c_j &= \tfrac{1}{2}(a_{j-1}-b_{j-1}), \qquad \text{for } j = 1, 2, 3, \ldots. \end{aligned}$$

The iteration terminates at some step $j = n$ when $\mathrm{fl}(c_n) = 0$ to machine precision, and the value a_n is then called the *arithmetic-geometric mean* (AGM).

If $a > b > 0$, then we have from their definitions that $a_j < a_{j-1}$ and $b_j > b_{j-1}$, so the a_j shrink and the b_j grow. Thus, the geometric mean is never bigger than the arithmetic mean, and the AGM lies between those two means. Computationally, that means that as long as a and b are representable, all members of the AGM sequence are as well, and lie in the range $[b, a]$. However, the product $a_j b_j$ in the square root is subject to premature overflow and underflow, so the geometric mean needs to be computed as $\sqrt{a_j}\sqrt{b_j}$ unless we know in advance that $a_j b_j$ is always representable. Similarly, the sum that forms a_j can overflow when half that sum is still representable. Rewriting it as $\frac{1}{2}a_{j-1} + \frac{1}{2}b_{j-1}$ solves the overflow problem, but introduces a possible premature underflow.

Unlike Newton–Raphson iterations, AGM iterations are *not* self-correcting: errors accumulate in each step. Fortunately, only a few iterations are needed in practice.

It can be proved that, in exact arithmetic, the c_j always decrease by at least a factor of two, adding one bit to the converging values of a_j and b_j. It is easy to exhibit a case where that happens: choose $a > 0$ and $b = 0$. The b_j terms from the geometric mean are then always zero, and a_j and c_j are halved on each step, eventually underflowing to zero in any finite-precision floating-point system.

In practice, however, we usually have $a \approx b$, and when $a > b > 0$, the convergence is *quadratic*, doubling the number of converged digits in a_j at each step. It is useful to prove that convergence claim, because there is a bonus in doing so. We start by factoring the difference of squares, and then solving for c_{j+1}:

$$a_j^2 - b_j^2 = (a_j + b_j)(a_j - b_j) = (2a_{j+1})(2c_{j+1}), \qquad c_{j+1} = (a_j^2 - b_j^2)/(4a_{j+1}).$$

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_20

Next, we expand a_j^2 and c_j^2 according to their definitions in the AGM, take their difference, and solve for c_j^2:

$$\begin{aligned} a_j^2 &= \tfrac{1}{4}(a_{j-1}+b_{j-1})^2, & c_j^2 &= \tfrac{1}{4}(a_{j-1}-b_{j-1})^2, \\ &= \tfrac{1}{4}(a_{j-1}^2+2a_{j-1}b_{j-1}+b_{j-1}^2), & &= \tfrac{1}{4}(a_{j-1}^2-2a_{j-1}b_{j-1}+b_{j-1}^2), \\ &= \tfrac{1}{4}(a_{j-1}^2+2b_j^2+b_{j-1}^2), & &= \tfrac{1}{4}(a_{j-1}^2-2b_j^2+b_{j-1}^2), \\ a_j^2-c_j^2 &= b_j^2, & c_j^2 &= a_j^2-b_j^2. \end{aligned}$$

From the two formulas for the c coefficients, we can conclude that

$$c_1 = \tfrac{1}{2}(a_0-b_0), \qquad c_{j+1} = \tfrac{1}{4}c_j^2/a_{j+1}, \qquad \text{for } j = 1,2,3,\ldots.$$

For numerical work, to avoid premature underflow and overflow, evaluate the right-hand side as $(\frac{1}{2}c_j)((\frac{1}{2}c_j)/a_{j+1})$, and obey the parentheses.

That important result demonstrates that the difference $a_j - b_j$ is *squared* in the next iteration, producing quadratic convergence. The bonus is that it allows computation of c_{j+1} without having to subtract two almost-equal numbers, at the extra cost of one division, and provided that we obey the restriction on the argument ordering. Of course, if a and b are close to begin with, it is likely that their floating-point exponents are the same, in which case the subtraction $a_j - b_j$ is exact. However, when that is not the case, and we need the sequence of c_j values, it may be more accurate to compute them by multiplication and division than by subtraction.

Some applications of the arithmetic-geometric mean need only a and b and the final a_n, whereas others require a, b, c, and some or all of the intermediate values a_j, b_j, and c_j. In the mathcw library, we therefore provide both scalar and vector functions, with these generic prototypes:

```
fp_t AGM(fp_t a, fp_t b);

fp_t VAGM(fp_t a, fp_t b, fp_t c, int nabc,
          fp_t aj[/*nabc*/], fp_t bj[/*nabc*/], fp_t cj[/*nabc*/],
          int * pneed);
```

The functions in both families return the final arithmetic-geometric mean, a_n. Maple provides the scalar function as `GaussAGM(a,b)`, Mathematica supplies it as `ArithmeticGeometricMean[a,b]`, PARI/GP offers `agm(a,b)`, and REDUCE has `AGM_function(a,b,c)`.

The vector version records the sequence members in the argument arrays, as long as there is sufficient space to do so. On return, `need` is the number of elements that needed to be set: if it is larger than the vector size `nabc`, the arrays are too small, and the remaining sequence values after entry `[nabc - 1]` have not been recorded. However, the function result still provides the converged final value a_n. The code in `VAGM()` avoids storage into the location of `need` if it is a `NULL` pointer.

Although the AGM iteration looks trivial, there are computational difficulties due to the nature of finite precision and range of floating-point arithmetic that must be dealt with in a general and robust library routine:

- NaN arguments, arguments of $(\infty, 0)$ or $(0, \infty)$, and arguments of opposite sign, must produce NaN results.
- Infinite arguments must produce a result of Infinity.
- It can happen that, near convergence, the a_j and b_j oscillate and differ by one or a few ulps, never managing to reach the equality that produces $c_j = 0$ and terminates the iteration.
- For starting values a and b near the floating-point limits, the sum $a_{j-1} + b_{j-1}$ can overflow, or halving it can underflow.
- For about *half* the floating-point range, the product $a_{j-1}b_{j-1}$ underflows or overflows.

Three obvious properties of the arithmetic-geometric mean provide a solution to those problems:

$$\operatorname{agm}(a,b) = \operatorname{agm}(b,a), \qquad \operatorname{agm}(a,0) = 0, \qquad \operatorname{agm}(sa,sb) = s\operatorname{agm}(a,b).$$

The first two allow us to avoid the slow one-bit-at-a-time iteration if either argument is zero. The third permits scaling to prevent premature overflow or underflow. By choosing $s = \beta^m$, where β is the base and m is an integer, the scaling of the arguments and the final a_n are exact operations.

The vector function has more detail, but no important additional computational features, so we show only the scalar function here. The code looks like this:

```
fp_t
AGM(fp_t a, fp_t b)                     /* arithmetic-geometric mean of a and b */
{
    fp_t result;

    if (ISNAN(a))
        result = SET_EDOM(a);
    else if (ISNAN(b))
        result = SET_EDOM(b);
    else if (ISINF(a))
        result = (b == ZERO) ? SET_EDOM(QNAN("")) : SET_ERANGE(a);
    else if (ISINF(b))
        result = (a == ZERO) ? SET_EDOM(QNAN("")) : SET_ERANGE(b);
    else if (a == ZERO)
        result = a;                     /* ensure exact result, and avoid a slow loop */
    else if (b == ZERO)
        result = b;                     /* ensure exact result, and avoid a slow loop */
    else if (a == b)
        result = a;
    else if (SIGNBIT(a) != SIGNBIT(b))
        result = SET_EDOM(QNAN(""));
    else
    {
        int n, negative_a;

        negative_a = (a < ZERO);

        if (negative_a)
        {
            a = -a;
            b = -b;
        }

        for (n = 0; n < MAXAGM; ++n)
        {
            fp_t a_n, b_n;

            a_n = (MAX(a, b) > ONE) ? (HALF * a + HALF * b) : HALF * (a + b);
            b_n = SQRT(a) * SQRT(b);    /* two square roots avoid premature overflow/underflow */
            a = a_n;
            b = b_n;

            if (FABS(a - b) < (FP_T_EPSILON * a))
                break;                  /* early exit: convergence is quadratic */
        }

        result = negative_a ? -a : a;
    }

    return (result);
}
```

Table 20.1: Computing π from the quadratically convergent algorithm for the arithmetic-geometric mean. Figures after the first incorrect digit have been omitted. The last row has been truncated to fit the page, but matches π to the last displayed digit.

A ninth-order (nonic) algorithm reaches more than 1.4 million digits in six iterations, and could produce more than 1000 million digits in ten iterations.

j	$p_j(\to \pi)$
1	3.18
2	3.1416
3	3.14159 26538
4	3.14159 26535 89793 23846 6
5	3.14159 26535 89793 23846 26433 83279 50288 41971 699
6	3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510 58209 74944 59230 78164

The usual checks on entry handle the special cases of NaN and Infinity, and the checks for zero arguments avoid a slow loop, and ensure correct handling of signed zeros. We handle the problem of oscillation at convergence by two safety measures: limiting the number of iterations to MAXAGM, and leaving the loop early if a_n and b_n are within a relative machine epsilon of each other. The bound could be loosened to a few ulps if further computational experience on some historical platforms shows it to be necessary.

The geometric mean must be computed with *two* square roots, rather than one, to avoid premature overflow and underflow. It is not sufficient to just do an initial scaling of the arguments, because even with that scaling, intermediate sequence members can be out of range. To see how that happens, consider the case with a near the maximum normal number, and b in $(0, 1)$. The arithmetic mean reduces a by a factor of two in each of the first few iterations, whereas the geometric mean grows rapidly. Because a_j is still close to the overflow limit, $a_j b_j$ soon overflows, even though $\sqrt{a_j b_j}$ is representable.

The loop limit of MAXAGM is chosen based on numerical experiments that show that even with arguments of wildly differing exponents in 128-bit arithmetic, fewer than 12 iterations invariably suffice, so the larger limit of 20 should handle future 256-bit arithmetic.

For random arguments in $(0, 1)$, 64-bit IEEE 754 arithmetic rarely needs more than five iterations, as demonstrated by this test in hoc with a version of the AGM instrumented with a global iteration count:

```
hoc64> load ("agm")
hoc64> for (k = 0; k < 70; ++k) {agm(rand(), rand()); print K_agm, ""}
5 6 6 4 4 4 4 4 5 4 4 5 6 5 5 4 5 5 4 5 4 4 5 4 6 5 4 4 4 5 7 5 4 4 4
5 4 5 4 4 5 5 5 7 4 5 4 5 4 4 4 5 5 5 3 4 4 5 6 4 5 5 5 3 4 6 5 4 5 4
```

The test can easily be extended to search for higher iteration counts:

```
hoc64> for (k = 0; k < 1_000_000; ++k) \
hoc64>     { agm(rand(), rand()); if (K_agm > 7) print K_agm, "" }
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 ...
```

There are 190 counts of 8 in the million tests, and just one count of 9 in a variant with 10^8 tests. In the 32-bit format, no counts exceed 7 in that number of tests, and in the 80-bit and 128-bit formats, none is above 9.

Apart from its application to the computation of the functions in this chapter, here are some examples of the AGM for practical work:

- With $a_0 = 1$, $b_0 = 2$, and $s_0 = \frac{1}{2}$, form a_j and b_j as in the AGM, but set $c_j = a_j^2 - b_j^2$ and $s_j = s_{j-1} - 2^j c_j$. Then $p_j = 2a_j^2/s_j$ converges to π quadratically. The working precision should be higher than the target precision. To avoid loss of accuracy, compute c_j as $(a_j + b_j)(a_j - b_j)$. The subtraction in the second term is then exact, because the terms have the same exponent. Table 20.1 shows the output of a short Maple program that implements the iteration.

  Variants of that algorithm provide cubic, quartic, quintic, … convergence, and have been used to compute π to more than 10^{11} digits. For details and motivation, see [BBBP97] and [BB04, Chapter 3].

The relation of π to the AGM is sufficiently rich that there is an entire book devoted to it [BB87b]. A more recent book also covers both subjects [AH01], and there are several general books about the mathematical fascination of π [Bar92, Bec93, Bar96, Bla97, BBB00, EL04b, Nah06].

Gauss apparently did not connect the AGM to the computation of π, and the earliest record of the AGM iteration in his manuscripts is from 1809 [AH01, page 101], a decade after he independently discovered it in 1799. Lagrange found it first, and before 1785, but Gauss got the credit. More than 150 years elapsed before the first application, in 1975 [Bre76b, Sal76], of the AGM to the π problem, a period in which several people spent years of their lives computing π from slower-converging series; see the tables in [BBBP97, pages 9–10] and [AH01, pages 205–207]. A thousand correct digits were only reached in 1949, a million in 1973, 1.6×10^7 in 1983 [BB83], 10^9 in 1989, 10^{12} in 2002, and 2.7×10^{12} in 2010. With the quartic algorithm, fewer than 100 full-precision operations of multiply, divide, and square root suffice to produce 45 million digits of π [BB87b, page 341].

- The inverse tangent can be computed from the AGM-like steps

$$\begin{aligned} r &= \sqrt{1+x^2}, & a_0 &= 1/r, & b_0 &= 1, \\ a_j &= \tfrac{1}{2}(a_{j-1}+b_{j-1}), & b_j &= \sqrt{a_j b_{j-1}}, & &\text{for } j = 1,2,3,\ldots,n, \\ \operatorname{atan}(x) &= x/(ra_n). \end{aligned}$$

Notice that the computation of b_j differs slightly from that of the standard AGM algorithm: it uses the new a_j instead of the old a_{j-1}. To avoid overflow, when x is sufficiently large, replace the first step by $r = x$. Compute the square root as the product of two roots.

Unfortunately, convergence of that algorithm is not quadratic: each iteration supplies just *two* more bits in the result. It is tedious to show that by hand, but a short Maple program makes it easier:

```
> NMAX := 6:

> a := array(0 .. NMAX):
> b := array(0 .. NMAX):
> c := array(0 .. NMAX):

> assume(B > 0, A >= B):

> a[0] := A:
> b[0] := B:
> c[0] := (a[0] - b[0])/2:

> for j from 1 to NMAX do
>     a[j] := simplify( (a[j-1] + b[j-1]) / 2 ):
>     b[j] := simplify( sqrt(a[j] * b[j-1]) ):
>     c[j] := simplify( (a[j-1] - b[j-1]) / 2 ):
> end do:

> for j from 1 to NMAX - 1 do
>     printf("c[%d] / c[%d] = %a\n", j + 1, j, simplify(limit(c[j + 1] / c[j], A = B)))
> end do:

c[2] / c[1] = 1/4
c[3] / c[2] = 1/4
c[4] / c[3] = 1/4
c[5] / c[4] = 1/4
c[6] / c[5] = 1/4
```

- To compute the inverse trigonometric and hyperbolic cosines, run the iteration found by Gauss in 1800 [Car71, page 498]:

$$a_j = \tfrac{1}{2}(a_{j-1}+b_{j-1}), \qquad \text{for } j = 1,2,3,\ldots,n,$$

$$\begin{aligned}
b_j &= \sqrt{a_j b_{j-1}}, \\
\operatorname{acos}(a_0/b_0) &= \sqrt{b_0^2 - a_0^2}/a_n, && \text{for } 0 \le a_0 < b_0, \\
\operatorname{acosh}(a_0/b_0) &= \sqrt{a_0^2 - b_0^2}/a_n, && \text{for } 0 < b_0 < a_0.
\end{aligned}$$

The convergence is the same as that for the inverse-tangent algorithm, gaining only two bits per step.

- With t-bit precision, and m chosen such that $s = 2^m x > 2^{t/2}$, then the steps

$$\log(2) = \frac{\frac{1}{2}\pi}{m \operatorname{agm}(1, 4/2^m)} \qquad \log(x) = \frac{\frac{1}{2}\pi}{\operatorname{agm}(1, 4/s)} - m\log(2)$$

provide a fast route to a high-precision value of the logarithm of x. That value can then be used with Newton–Raphson iteration to find a high-precision value of $\exp(x)$ [BB04, page 228].

Because of the subtraction in forming $\log(x)$, there is the possibility of loss of significant digits, but graphs of the relative error in numerical experiments with that algorithm show a more serious problem: errors of a few thousand ulps for arguments $x \approx 1$. In that region, the computation should therefore switch to an algorithm for computing $\operatorname{log1p}(x-1)$, as discussed in **Section 10.4** on page 290.

20.2 Elliptic integral functions of the first kind

The *incomplete* elliptic integral function of the *first kind* is defined by these relations in three different notations [AS64, Chapter 17] [OLBC10, §19.1]:

$$\begin{aligned}
F(\phi\backslash\alpha) &= \int_0^{\phi} [1 - (\sin(\alpha))^2(\sin(\theta))^2]^{-\frac{1}{2}}\, d\theta, \\
F(\phi|m) &= \int_0^{\sin(\phi)} [(1-t^2)(1-mt^2)]^{-\frac{1}{2}}\, dt \\
F(\phi,k) &= \int_0^{\sin(\phi)} [(1-t^2)(1-(kt)^2)]^{-\frac{1}{2}}\, dt \\
k &= \sin(\alpha), && \textit{so } k \textit{ is in } [-1,1], \\
m &= k^2, && \textit{and } m \textit{ is in } [0,1].
\end{aligned}$$

Because only even powers of the integration variable, t, appear in the integrand, it is unchanged under sign inversion of t. Thus, we have an important symmetry relation:

$$F(-\phi|m) = -F(\phi|m).$$

When $\phi = \frac{1}{2}\pi$, and thus, $\sin(\phi) = 1$, the bivariate function reduces to a simpler one called the *complete* elliptic integral function of the *first kind*:

$$\begin{aligned}
K(\alpha) &= F(\tfrac{1}{2}\pi\backslash\alpha) \\
&= \int_0^{\pi/2} [1 - (\sin(\alpha)\sin(\theta))^2]^{-\frac{1}{2}}\, d\theta, \\
K(k) &= F(\tfrac{1}{2}\pi, k) \\
&= \int_0^1 [1 - (kt)^2]^{-\frac{1}{2}}\, dt, \\
K(m) &= F(\tfrac{1}{2}\pi|m) \\
&= \int_0^{\pi/2} [1 - m(\sin(\theta))^2]^{-\frac{1}{2}}\, d\theta \\
&= \int_0^1 [(1-t^2)(1-mt^2)]^{-\frac{1}{2}}\, dt,
\end{aligned}$$

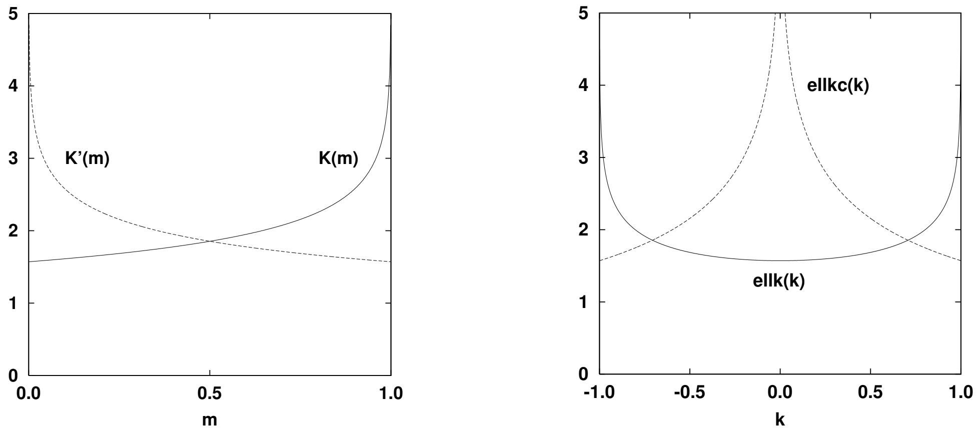


Figure 20.1: Complete elliptic integral functions of the first kind. $K(m)$ has a pole at $m = 1$, and $K'(m)$ a pole at $m = 0$. The mathcw library function $\mathrm{ellk}(k)$ has poles at $k = \pm 1$, and $\mathrm{ellkc}(k)$ a pole at $k = 0$. The finite limits are $K(0) = K'(1) = \mathrm{ellk}(0) = \mathrm{ellkc}(\pm 1) = \frac{1}{2}\pi$. The function pairs are equal at $m = \frac{1}{2}$ and $k = \pm\sqrt{\frac{1}{2}}$.

$$\begin{aligned}
\mathrm{ellk}(k) &= K(k), && \textit{mathcw library notation,}\\
\mathrm{EllipticK}(k) &= K(k), && \textit{Maple notation,}\\
\mathrm{EllipticK}[m] &= K(m), && \textit{Mathematica notation,}\\
\mathrm{elliptic\_kc}(m) &= K(m), && \textit{Maxima notation,}\\
\mathrm{EllipticK}(m) &= K(m), && \textit{REDUCE notation.}
\end{aligned}$$

In the literature on elliptic integral functions, k is called the *modulus*, and m is called the *parameter*. The presence of three argument conventions, α, k, and m, due to historical development, is a nuisance, particularly when some of the symbolic-algebra systems use the same function name but different arguments, so be sure to check the argument conventions carefully before using software implementation of elliptic functions.

For numerical work, to handle accurately the case of $m \approx 1$, it proves useful to have an additional function, called the *complementary* complete elliptic integral function of the first kind. It is defined by

$$\begin{aligned}
K'(k) &= K(\sqrt{1-k^2}),\\
K'(m) &= K(1-m),\\
\mathrm{ellkc}(k) &= K'(k), && \textit{mathcw library notation,}\\
\mathrm{EllipticCK}(k) &= K'(k), && \textit{Maple notation,}\\
\mathrm{EllipticK}[1-m] &= K'(m), && \textit{Mathematica notation.}
\end{aligned}$$

Here, the prime on $K'(m)$ reflects historical notation, and does *not* mean the first derivative of $K(m)$. The notation $m' = 1 - m$ is common in publications that treat elliptic integral functions, although we henceforth avoid it.

Our naming convention follows that for the ordinary and complementary error functions, and their inverses: the base name `ellk`, then the letter `c` in the complementary case, followed by any of the precision suffixes used in the C library. In Maple, the letter `C` for the complementary function *precedes* the letter `K` in the name `EllipticCK`.

Mathematica does not provide a separate function for the complementary case. That does not matter for applications in symbolic algebra, but it does for numerical evaluation.

The complete elliptic integral functions of the first kind are graphed in **Figure 20.1**. The presence of poles alerts us to the problem of error magnification that we discussed in **Section 4.1** on page 61, so we show plots of the magnification in the left half of **Figure 20.3** on page 629. From the plots, we can see that the values of $K(m)$ and $\mathrm{ellk}(k)$ are particularly sensitive to their arguments for $m = k^2 \approx 1$, but that problem can be eliminated by switching to the complementary functions.

The behavior near the poles is described by these relations:

$$\lim_{m\to 1} K(m) = \tfrac{1}{2}\ln(16/(1-m)), \qquad \lim_{m\to 0} K'(m) = \tfrac{1}{2}\ln(16/m).$$

The logarithm term slows the approach to infinity: when m is the smallest normal number in 64-bit IEEE 754 arithmetic, $K'(m) \approx 710$.

High-precision numerical experiments in Maple allow us to determine the cutoffs where we can switch to the limit formulas, guaranteeing correct behavior near the poles. The final version of that code reports the errors in ulps for each of the extended IEEE 754 precisions:

```
> Digits := 140:
> LK := proc(m) return ln(16/(1 - m)) / 2 end proc:
> re := proc(k) return (LK(k^2) - ellk(k))/ellk(k) end proc:

> for t in {7, 16, 34, 70} do
>     eps := 10**(-t+1):    x := 1 - eps:
>     printf("%2d  %.3g  ", t, re(x)/ eps)
> end do:
 7  -0.437  16  -0.473  34  -0.487  70  -0.494

> for t in {24, 53, 64, 113, 237} do
>     eps := 2**(-t+1):    x := 1 - eps:
>     printf("%3d  %.3g  ", t, re(x)/ eps)
> end do:
 24  -0.445   53  -0.474   64  -0.478  113  -0.487  237  -0.591

> LKC := proc(m) return ln(16/m) / 2 end proc:
> rec := proc(k) return (LKC(k^2) - ellkc(k))/ellkc(k) end proc:

> for t in {7, 16, 34, 70} do
>     eps := 10**(-t+1):    x := sqrt(eps):
>     printf("%2d  %.3g ", t, rec(x)/ eps)
> end do:
 7  -0.220  16  -0.237  34  -0.244  70  -0.247

> for t in {24, 53, 64, 113, 237} do
>     eps := 2**(-t+1):    x := sqrt(eps):
>     printf("%3d  %.3g  ", t, rec(x)/ eps)
> end do:
 24  -0.223   53  -0.237   64  -0.239  113  -0.244  237  -0.247
```

The cutoffs all produce relative errors below a half ulp, and the negative signs on the errors tell us that the limiting form approaches the true value from below. We can therefore use these formulas near the endpoints:

$$\begin{aligned}
\operatorname{ellk}(k) &= \log(16/(1-k^2))/2, && \textit{when } (1-|k|) \le \epsilon, \\
\operatorname{ellkc}(k) &= \log(16/(k^2))/2, && \textit{when } |k| \le \sqrt{\epsilon}, \\
&= \log(4/|k|) \\
&= \log(4) - \log(|k|), && \textit{overflow-free computational form.}
\end{aligned}$$

The first is not useful, because there are only two representable values in binary arithmetic, and ten in decimal arithmetic, to which it applies. However, the second is eminently practical.

The complete elliptic integral functions of the first kind have these Taylor series, providing a fast way to compute the functions near their smallest values:

$$\begin{aligned}
K(m) &= \tfrac{1}{2}\pi(1 + (1/4)m + (9/64)m^2 + (25/256)m^3 + \\
&\qquad (1225/16\,384)m^4 + (3969/65\,536)m^5 + \cdots), \\
K'(m) &= \tfrac{1}{2}\pi(1 + (1/4)(1-m) + (9/64)(1-m)^2 + (25/256)(1-m)^3 + \\
&\qquad (1225/16\,384)(1-m)^4 + (3969/65\,536)(1-m)^5 + \cdots).
\end{aligned}$$

Notice that the complementary function is expanded in powers of $(1-m)$.

The coefficients in the expansions have a surprisingly intimate relation to the binary number system. The general term in the expansion of $K(m)$ is

$$\frac{2^{-2w(n)}\binom{2n}{n}^2}{\operatorname{denom}(W(n))}\,m^n,$$

where $w(n)$ is the number of 1-bits in the binary representation of n (see [Slo07, A038534]), and $\operatorname{denom}(W(n))$ is the denominator, after removing common integer factors, of this product:

$$W(n) = \prod_{j=1}^{n} \frac{(2j)^2-1}{(2j)^2}.$$

One example of the use of $K(m)$ is computation of the period of an oscillating pendulum (see [Law89, Chapter 5], [Tem96, page 316], [Ueb97, page 12], [AW05, §5.8], [AWH13, §18.8.1], or for more depth, [AE06, Chapter 1]). If the pendulum has frequency ω (lowercase Greek letter *omega*), and initial angular displacement ϕ, then with $k = \sin(\phi/2)$, the period is $T = 4K(k)/\omega$.

The function $K(m)$ also appears in the problem of finding the electrostatic potential above a conducting disk, the potential from a current loop, and the magnetization of a rod perpendicular to a plane: see [Law89, Chapter 5], [Jac75, pages 131, 178, and 208], and [OLBC10, §19.33]. $F(\phi|m)$ arises in the determination of the stress on a vertical column under load. The two elliptic functions are also required in several areas of astronomy and astrophysics [Fuk09a, §1.2], [Fuk09b, §1].

20.3 Elliptic integral functions of the second kind

The incomplete elliptic integral functions of the *second kind* have definitions similar to those of the first kind, but with the square root, instead of inverse square root, in the integrand:

$$\begin{aligned}
E(\phi\backslash\alpha) &= \int_0^{\phi} [1-(\sin(\alpha))^2(\sin(\theta))^2]^{1/2}\,d\theta,\\
E(\phi|m) &= \int_0^{\sin(\phi)} (1-t^2)^{-1/2}(1-mt^2)^{1/2}\,dt,\\
E(\phi,k) &= \int_0^{\sin(\phi)} (1-t^2)^{-1/2}(1-(kt)^2)^{1/2}\,dt,\\
k &= \sin(\alpha), && \textit{modulus in } [-1,+1],\\
m &= k^2, && \textit{parameter in } [0,1].
\end{aligned}$$

Setting $\phi = \frac{1}{2}\pi$ produces the *complete* elliptic integral function of the *second kind*, defined by:

$$\begin{aligned}
E(\alpha) &= E(\tfrac{1}{2}\pi\backslash\alpha)\\
&= \int_0^{\pi/2} [1-(\sin(\alpha)\sin(\theta))^2]^{1/2}\,d\theta,\\
E(k) &= E(\tfrac{1}{2}\pi,k)\\
&= \int_0^{\pi/2} [1-(k\sin(\theta))^2]^{1/2}\,d\theta\\
&= \int_0^{1} (1-t^2)^{-1/2}(1-(kt)^2)^{1/2}\,dt,\\
E(m) &= E(\tfrac{1}{2}\pi|m)\\
&= \int_0^{\pi/2} [1-m(\sin(\theta))^2]^{1/2}\,d\theta\\
&= \int_0^{1} (1-t^2)^{-1/2}(1-mt^2)^{1/2}\,dt
\end{aligned}$$

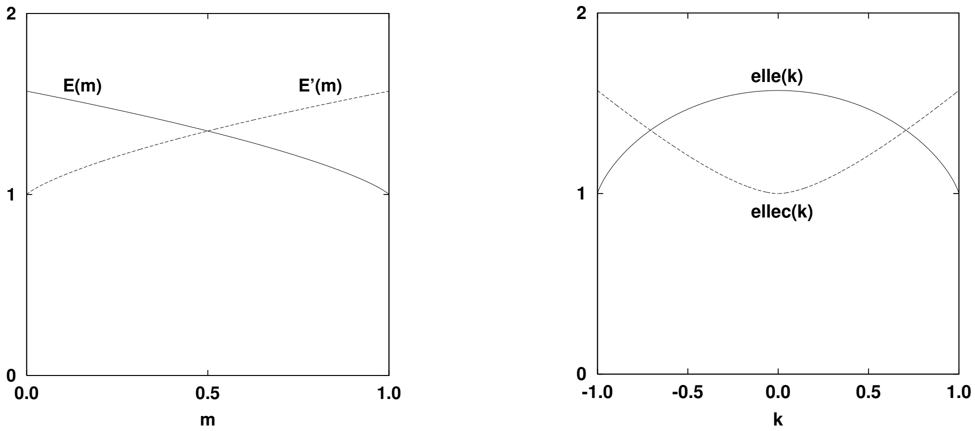


Figure 20.2: Complete elliptic integral functions of the second kind. The functions lie in $[1, \frac{1}{2}\pi]$ for arguments m in $[0, 1]$ and k in $[-1, 1]$. The limits are $E(0) = E'(1) = \frac{1}{2}\pi$, $E(1) = E'(0) = 1$, $\operatorname{elle}(\pm 1) = \operatorname{ellec}(0) = 1$, and $\operatorname{elle}(0) = \operatorname{ellec}(\pm 1) = \frac{1}{2}\pi$. The function pairs are equal at $m = \frac{1}{2}$ and $k = \pm\sqrt{\frac{1}{2}}$.

$$\begin{aligned}
&= \int_0^1 (1 - t^2)^{-1/2} (1 - (kt)^2)^{1/2}\, dt, \\
\operatorname{elle}(k) &= E(k), && \textit{mathcw library notation,} \\
\operatorname{EllipticE}(k) &= E(k), && \textit{Maple notation,} \\
\operatorname{EllipticE}[m] &= E(m), && \textit{Mathematica notation,} \\
\operatorname{elliptic\_ec}(m) &= E(m), && \textit{Maxima notation,} \\
\operatorname{EllipticE}(m) &= E(m), && \textit{REDUCE notation.}
\end{aligned}$$

Just as with the function of the first kind, we have an important reflection symmetry relation:

$$E(-\phi|m) = -E(\phi|m).$$

The *complementary* complete elliptic integral function of the *second kind* is given by:

$$\begin{aligned}
E'(k) &= E(\sqrt{1 - k^2}), \\
E'(m) &= E(1 - m), \\
\operatorname{ellec}(k) &= E'(k), && \textit{mathcw library notation,} \\
\operatorname{EllipticCE}(k) &= E'(k), && \textit{Maple notation,} \\
\operatorname{EllipticE}[1 - m] &= E'(m), && \textit{Mathematica notation.}
\end{aligned}$$

The functions are shown in **Figure 20.2**, and their error magnifications are graphed in the right half of **Figure 20.3** on the next page. As with the functions of the first kind, the complementary functions of the second kind are less sensitive to argument errors.

Mathematica does not offer a separate function for the complementary case, but we can easily provide definitions for the first and second kinds with code like this:

```
EllipticCK := Function[m, EllipticK[1 - m]]
EllipticCE := Function[m, EllipticE[1 - m]]
```

The first derivatives of the complete elliptic integral functions require those functions of both the first and second kind:

$$\frac{dK(m)}{dm} = \frac{E(m) - (1 - m)K(m)}{2m(1 - m)}, \qquad \frac{d\,\operatorname{ellk}(k)}{dk} = \frac{\operatorname{elle}(k)}{k(1 - k^2)} - \frac{\operatorname{ellk}(k)}{k},$$

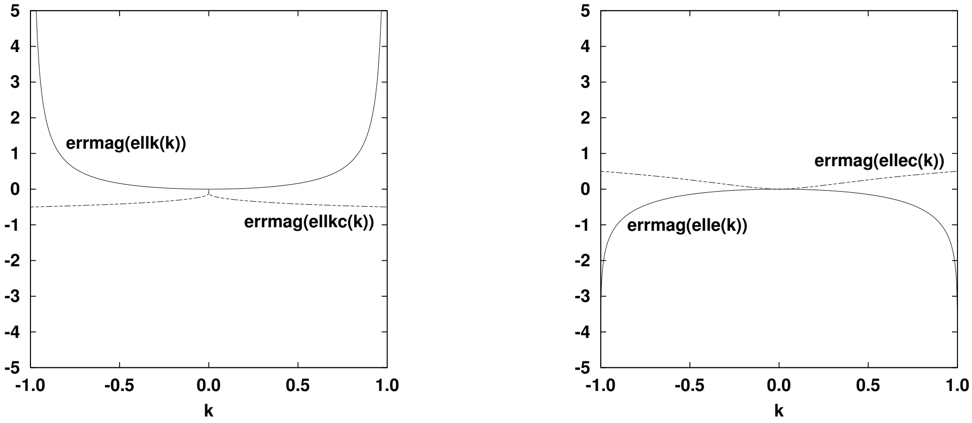


Figure 20.3: Error magnification in the complete elliptic integral functions of the first (left graph) and second (right graph) kinds. The curves for the ordinary functions approach ±∞ for $k \approx \pm 1$, whereas those for the complementary functions remain within $[-\frac{1}{2}, +\frac{1}{2}]$.

$$
\begin{aligned}
\frac{dK'(m)}{dm} &= \frac{mK'(m) - E'(m)}{2m(1-m)}, & \frac{d\,\mathrm{ellkc}(k)}{dk} &= \frac{k\,\mathrm{ellkc}(k)}{(1-k^2)} - \frac{\mathrm{ellec}(k)}{k(1-k^2)}, \\
\frac{dE(m)}{dm} &= \frac{E(m) - K(m)}{2m}, & \frac{d\,\mathrm{elle}(k)}{dk} &= \frac{\mathrm{elle}(k) - \mathrm{ellk}(k)}{k}, \\
\frac{dE'(m)}{dm} &= \frac{K'(m) - E'(m)}{2(1-m)}, & \frac{d\,\mathrm{ellec}(k)}{dk} &= \frac{k}{(1-k^2)}(\mathrm{ellkc}(k) - \mathrm{ellec}(k)).
\end{aligned}
$$

The complete elliptic integral functions of the second kind have these Taylor series for $m \approx 0$:

$$
\begin{aligned}
E(m) &= (\tfrac{1}{2}\pi)(1 - (1/4)m - (3/64)m^2 - (5/256)m^3 - (175/16\,384)m^4 - \\
&\qquad (441/65\,536)m^5 + \cdots), \\
E'(m) &= (\tfrac{1}{2}\pi)(1 - (1/4)(1-m) - (3/64)(1-m)^2 - (5/256)(1-m)^3 - \\
&\qquad (175/16\,384)(1-m)^4 - (441/65\,536)(1-m)^5 + \cdots).
\end{aligned}
$$

The general term in the series for $E(m)$ is similar to that for $K(m)$, but fractionally smaller (see [Slo07, A038535]):

$$
\frac{(-1)^{2n}}{1-2n}\,\frac{2^{-2w(n)}\binom{2n}{n}^2}{\mathrm{denom}(W(n))}\,m^n.
$$

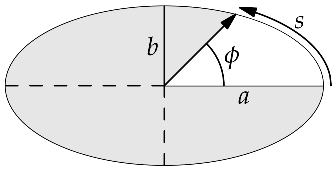
At the other endpoint, the two functions have these series:

$$
\begin{aligned}
E(m) &= 1 + (\ln(2) - \tfrac{1}{4}\ln(1-m) - \tfrac{1}{4})(1-m) + \\
&\qquad \big((3/8)\ln(2) - (3/32)\ln(1-m) - 13/64\big)(1-m)^2 + \\
&\qquad \big((15/64)\ln(2) - (15/256)\ln(1-m) - 9/64\big)(1-m)^3 + \cdots, \\
E'(m) &= 1 + (\ln(2) - \tfrac{1}{4}\ln(m) - \tfrac{1}{4})m + \\
&\qquad \big((3/8)\ln(2) - (3/32)\ln(m) - 13/64\big)m^2 + \\
&\qquad \big((15/64)\ln(2) - (15/256)\ln(m) - 9/64\big)m^3 + \cdots.
\end{aligned}
$$

Notice that only a single logarithm is required to evaluate all of the coefficients, and that, for the terms shown, the rational numbers are exactly representable in binary, decimal, and hexadecimal arithmetic, provided the decimal system offers at least seven digits, as it does for extended IEEE 754 arithmetic.

For an ellipse aligned with the coordinate axes, with semimajor axis of length a along x, and semiminor axis of length b along y, measure an angle θ counterclockwise from the x axis. The arc length s from $\theta = 0$ to $\theta = \phi$ is

$$\begin{aligned} s &= a \int_0^{\phi} \sqrt{1 - k^2 (\sin(\theta))^2}\, d\theta \\ &= aE(\phi \backslash \operatorname{asin} k), \\ k &= \sqrt{1 - (b/a)^2}. \end{aligned}$$



Here, the value k is called the *eccentricity*[1] of the ellipse, and is often instead denoted by e (then *not* the base of the natural logarithm).

Setting $\phi = \frac{1}{2}\pi$ shows that the quarter arc in the first quadrant has length $s = aE(k)$, so the perimeter of an ellipse is $4aE(k)$. For a circle of unit radius, we have $a = b = 1$, for which $k = 0$, so we can conclude that $E(0) = \frac{1}{2}\pi$.

The function $E(m)$ also appears in the electrostatic problems cited earlier on page 627 for $K(m)$.

If we temporarily omit the argument (m) to simplify the display, the four complete elliptic integral functions satisfy the famous *Legendre relation*:

$$\boldsymbol{KE' + K'E - KK' = \tfrac{1}{2}\pi,} \qquad \text{for } m \text{ in } (0, 1).$$

Decorated with arguments, and with the complementary functions eliminated, the relation looks like this:

$$\boldsymbol{K(m)E(1-m) + K(1-m)E(m) - K(m)K(1-m) = \tfrac{1}{2}\pi,} \qquad \text{for } m \text{ in } (0, 1).$$

For a proof, see [AAR99, page 137].

Here is a hoc session in 128-bit decimal arithmetic to check how closely the Legendre relation is satisfied for its implementations of those functions, with reports of the error in ulps for a few random arguments:

```
hocd128> load("ell")
hocd128> func K(m)  return ellk (sqrt(m))
hocd128> func KC(m) return ellkc(sqrt(m))
hocd128> func E(m)  return elle (sqrt(m))
hocd128> func EC(m) return ellec(sqrt(m))
hocd128> func LR(m) return K(m)*EC(m) + KC(m)*E(m) - K(m)*KC(m)
hocd128> for (k = 0; k < 14; ++k) printf("%.2f ",ulp(LR(rand()),PI/2))
1.27 0.00 1.27 1.91 2.55 1.27 1.27 1.27 0.64 1.27 1.27 1.27 1.91 2.55
```

At that precision, the computed value `PI/2` differs from $\frac{1}{2}\pi$ by one unit in the 34th digit, so our four functions seem to perform quite well.

20.4 Elliptic integral functions of the third kind

The *incomplete* elliptic integral function of the *third kind* is named with an uppercase Greek letter *pi* and has three arguments [BF71, pages 223–239]. It is defined by an integral in three different notations that are distinguished by punctuation and argument-naming conventions:

$$\begin{aligned} \Pi(n; \phi \backslash \alpha) &= \int_0^{\phi} (1 - n(\sin(\theta))^2)^{-1} (1 - (\sin(\alpha)\sin(\theta))^2)^{-1/2}\, d\theta, \\ \Pi(n, \phi, k) &= \int_0^{\phi} (1 - n(\sin(\theta))^2)^{-1} (1 - (k\sin(\theta))^2)^{-1/2}\, d\theta \\ &= \int_0^{\sin\phi} (1 - nt^2)^{-1} ((1 - t^2)(1 - (kt)^2))^{-1/2}\, dt, \\ \Pi(n, \phi | m) &= \int_0^{\phi} (1 - n(\sin(\theta))^2)^{-1} (1 - m(\sin(\theta))^2)^{-1/2}\, d\theta, \\ k &= \sin(\alpha), \qquad \textit{modulus in } [-1, +1], \end{aligned}$$

[1] Planetary orbits in our solar system are elliptical, with eccentricities k of 0.21 (Mercury), 0.01 (Venus), 0.02 (Earth), 0.09 (Mars), 0.05 (Jupiter), 0.06 (Saturn), 0.05 (Uranus), 0.01 (Neptune), and 0.25 (Pluto). However, orbits of other planetary bodies differ, with eccentricities of 0.97 (Halley's Comet), 0.83 (Icarus asteroid), and 0.9999 (Comet Kohoutek). The latter has $a/b \approx 70$.

$$\begin{aligned} m &= k^2, && \textit{parameter in } [0,1], \\ n &\geq 0. \end{aligned}$$

Be warned that some publications, and some symbolic-algebra systems, interchange the first two arguments, and sometimes, an argument ϕ is replaced by $\sin(\phi)$.

When $\phi = \frac{1}{2}\pi$, we have the two-argument *complete* elliptic integral function of the third kind, again with three different argument conventions:

$$\Pi(n\backslash\alpha) = \Pi(n,k) = \Pi(n|m) = \Pi(n;\tfrac{1}{2}\pi\backslash\alpha) = \Pi(n,\tfrac{1}{2}\pi,k) = \Pi(n,\tfrac{1}{2}\pi|m).$$

The function of the third kind has these special cases:

$$\begin{aligned}
\Pi(0,\phi,k) &= F(\phi,k), && \textit{elliptic integral function of first kind,} \\
\Pi(0,\phi,0) &= \phi, \\
\Pi(n,\phi,0) &= \begin{cases} (1-n)^{-1/2}\operatorname{atan}((1-n)^{1/2}\tan\phi), & \textit{for } n<1, \\ \tan(\phi), & \textit{for } n=1, \\ (n-1)^{-1/2}\operatorname{atanh}((n-1)^{1/2}\tan\phi), & \textit{for } n>1, \end{cases} \\
\Pi(n,\phi,1) &= (1-n)^{-1}\times \\
&\quad \left(\log(\tan(\phi)+\sec(\phi)) - \tfrac{1}{2}\sqrt{n}\log\left(\frac{1+\sqrt{n}\sin(\phi)}{1-\sqrt{n}\sin(\phi)}\right)\right), \\
&\qquad \textit{provided } n \neq 1, \\
\Pi(n,\tfrac{1}{2}\pi,1) &= +\infty, && \textit{for all } n \geq 0.
\end{aligned}$$

For small ϕ, and n of limited size, the elliptic integral function of the third kind has this Taylor-series expansion:

$$\begin{aligned}
\Pi(n,\phi,k) = \phi &+ \tfrac{1}{3!}(2n+k^2)\phi^3 + \\
&\tfrac{1}{5!}(-8n-4k^2+12k^2n+24n^2+9k^4)\phi^5 + \\
&\tfrac{1}{7!}(16k^2-180k^4+225k^6+32n-240k^2n + \\
&\quad 270k^4n-480n^2+360k^2n^2+720n^3)\phi^7+\cdots.
\end{aligned}$$

Series exist for the case of small n and small k, but they are too complicated to be of computational use.

Maple provides the incomplete elliptic integral function of the third kind as `EllipticPi(sin(`ϕ`), n, k)`; note the unusual argument conventions and order. Mathematica has `EllipticPi[n, `ϕ`, m]`, where $m = k^2$. Maxima supplies `elliptic_pi(n, `ϕ`, m)`. The `mathcw` library has the function family `ELLPI(n,phi,k)`, but we delay a discussion of its computational algorithm until **Section 20.10** on page 650.

Because the function of the third kind is less-commonly encountered, we do not provide a separate function family in the `mathcw` library for the complete function: just use $\phi = \frac{1}{2}\pi$ in a call to the `ELLPI()` family.

20.5 Computing $K(m)$ and $K'(m)$

The discovery of the relation of the arithmetic-geometric mean to the elliptic integral functions is an early example of success in the emerging area of *experimental mathematics* [BBG03, BB04, BBC$^+$07, BD09], whereby results of a high-precision numerical computation suggest mathematical formulas that are later derived rigorously.

In 1799, Gauss computed

$$\operatorname{agm}(1,\sqrt{2}) = 1.198\,140\,234\,735\,592\,207\,439\,922\,492\,280\,323\,878\ \ldots$$

to 11 figures, and found that it agreed with the reciprocal of the value of an integral tabulated by Stirling:

$$\frac{2}{\pi}\int_0^1 \frac{dt}{\sqrt{1-t^2}}.$$

Gauss conjectured, and later proved, that the two numbers are indeed identical, and predicted that his observation would open a new field of analysis. Today, that field is elliptic integrals and modular functions. For more on that story, with pictures of Gauss's original manuscripts, see [AH01, Chapter 7] and [BB04, page 12].

The arithmetic-geometric mean leads to the complete elliptic integral function of the first kind like this [AS64, page 598], [MH72], [OLBC10, §19.8]:

$$\begin{aligned} k &= \sin(\alpha), & a_0 &= 1, \\ m &= k^2 = (\sin(\alpha))^2, & b_0 &= \cos(\alpha) = \sqrt{1-(\sin(\alpha))^2}, \\ K(m) &= \frac{\frac{1}{2}\pi}{\operatorname{agm}(a_0, b_0)} = \frac{\frac{1}{2}\pi}{\operatorname{agm}(1, \sqrt{1-k^2})}. \end{aligned}$$

By definition of the AGM, at convergence we have $a_n = b_n$, so we can do one more step to find

$$\operatorname{agm}(a_n, b_n) = \operatorname{agm}(a_{n+1}, b_{n+1}) = \operatorname{agm}((a_n + b_n)/2, \sqrt{a_n b_n}).$$

That result allows us to conclude that

$$\begin{aligned} \operatorname{agm}(1+k, 1-k) &= \operatorname{agm}(\tfrac{1}{2}(1+k+1-k), \sqrt{(1+k)(1-k)}) \\ &= \operatorname{agm}(1, \sqrt{1-k^2}). \end{aligned}$$

Thus, we can eliminate the trigonometric functions and square roots to arrive at optimal formulas for computation of the complete elliptic integral function of the first kind:

$$\begin{aligned} \operatorname{ellk}(k) &= K(k) \\ &= \frac{\frac{1}{2}\pi}{\operatorname{agm}(1+k, 1-k)}, && \textit{use for binary and decimal arithmetic,} \\ &= \frac{\frac{1}{4}\pi}{\operatorname{agm}(\frac{1}{2}+\frac{1}{2}k, \frac{1}{2}-\frac{1}{2}k)}, && \textit{use for hexadecimal arithmetic.} \end{aligned}$$

Apply the symmetry relation $\operatorname{ellk}(-k) = \operatorname{ellk}(k)$ to reduce the argument range to $[0, 1]$. In binary arithmetic, the subtraction $1 - k$ is *exact* for k in $[\frac{1}{2}, 1]$, and, at least in the IEEE 754 system, correctly rounded otherwise.

A separate formula for $\operatorname{ellk}(k)$ is needed for hexadecimal arithmetic, because the values $\frac{1}{4}\pi$ and $\frac{1}{2} + \frac{1}{2}k$ then have no leading zero significand bits, reducing the effect of wobbling precision.

Here is the mathcw library code for the complete elliptic integral function of the first kind:

```
fp_t
ELLK(fp_t k)
{   /* complete elliptic integral of the FIRST kind */
    fp_t result;
    static fp_t cut_taylor = FP(0.);
    static fp_t last_k = FP(-2.);       /* out-of-range k */
    static fp_t last_result = FP(0.);
    static int do_init = 1;

    if (do_init)
    {
        cut_taylor = SQRT(FP_T_EPSILON + FP_T_EPSILON);
        do_init = 0;
    }

    if (ISNAN(k))
        result = SET_EDOM(k);
    else if ( (k < -ONE) || (ONE < k) )
        result = SET_EDOM(QNAN(""));
    else if ( (k == ONE) || (k == -ONE) )
        result = SET_ERANGE(INFTY());
    else if (k == last_k)     /* use history for common K(k) calls */
        result = last_result;
    else if (k == ZERO)
        result = PI_HALF;
    else
    {
        if (k < ZERO)   /* use symmetry to ensure positive argument */
            k = -k;

        if (k < cut_taylor)
        {             /* two-term Taylor series for correct rounding */

#if defined(HAVE_WOBBLING_PRECISION)
            result = PI * (HALF + EIGHTH * k * k);
#else
            result = PI_HALF * (ONE + FOURTH * k * k);
#endif /* defined(HAVE_WOBBLING_PRECISION) */

        }
        else          /* use AGM algorithm */
        {

#if defined(HAVE_WOBBLING_PRECISION)

            {
                fp_t half_k;

                half_k = HALF * k;
                result = PI_QUARTER / AGM(HALF + half_k, HALF - half_k);
            }
#else
            result = PI_HALF / AGM(ONE + k, ONE - k);
#endif /* defined(HAVE_WOBBLING_PRECISION) */

        }
    }

    last_k = k;
    last_result = result;

    return (result);
}
```

The function maintains a history of its most-recent argument and result, and checks whether it can return that value without further computation. Although that optimization is possible in almost any function in the mathcw library, we rarely use it. However, in the mathematics of elliptic functions, the expression $K(k)$ is so common that programmers are likely to call `ELLK(k)` repeatedly with the same argument.

Figure 20.4 on the next page shows the measured errors for that function family.

To find a route to the complementary function, we start with

$$\begin{aligned}\operatorname{ellkc}(k) &= K'(k) = K(\sqrt{1-k^2}) = \operatorname{ellk}(\sqrt{1-k^2})\\ &= \frac{\frac{1}{2}\pi}{\operatorname{agm}(1+\sqrt{1-k^2}, 1-\sqrt{1-k^2})}.\end{aligned}$$

That is unsuitable for numeric computation when k is small, because the second argument of agm() suffers catas-

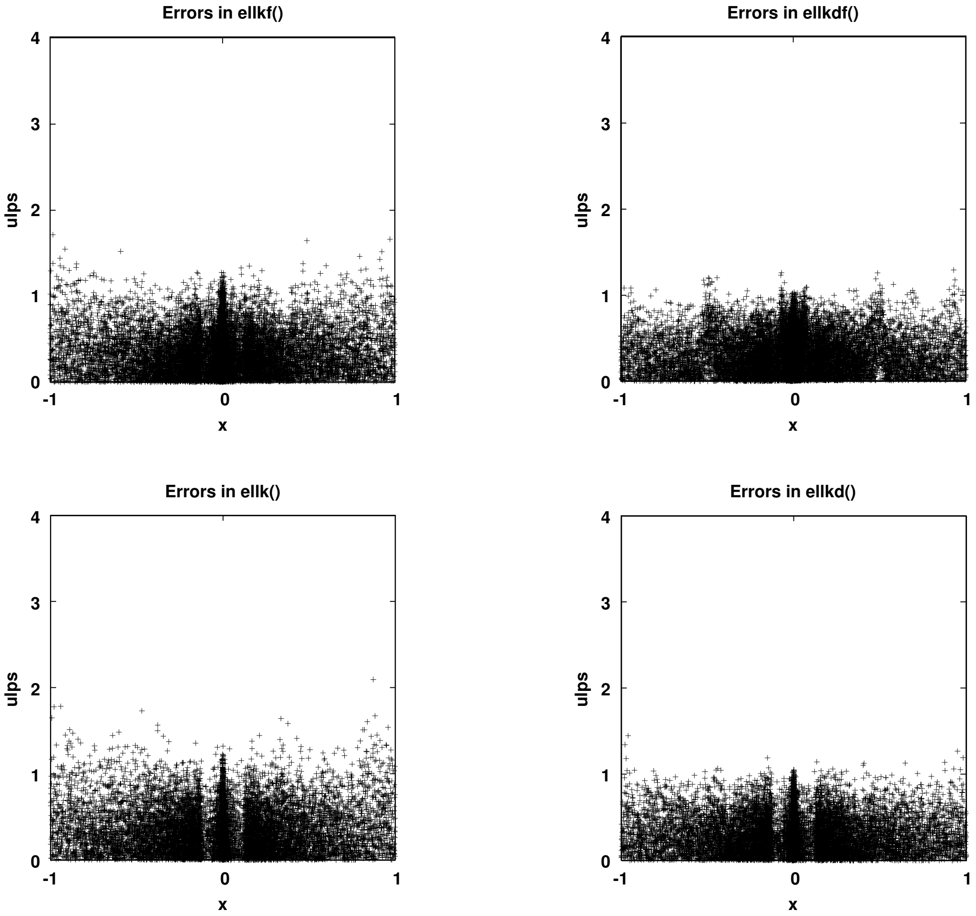


Figure 20.4: Errors in the complete elliptic integral functions of the first kind. Notice that the vertical scale is larger than in other chapters of this book.

trophic accuracy loss. To solve that problem, rewrite the arguments as follows:

$$\begin{aligned}
a &= 1 + \sqrt{1-k^2}, && \textit{first argument of } \texttt{agm()}, \\
&= 1 + \sqrt{(1-k)(1+k)}, && \textit{computation is } \text{always } \textit{stable,} \\
b &= 1 - \sqrt{1-k^2}, && \textit{second argument of } \texttt{agm()}, \\
&= \frac{1+\sqrt{1-k^2}}{1+\sqrt{1-k^2}}(1-\sqrt{1-k^2}) \\
&= \frac{1-(1-k^2)}{1+\sqrt{1-k^2}} \\
&= k^2/a, && \textit{stable computational form.}
\end{aligned}$$

Now we can determine b stably without subtraction loss, but because k is restricted to $[-1, 1]$, k^2 is subject to premature underflow. We remove that problem by using the scaling relation for the AGM with $s = 1/k$:

$$\begin{aligned}
\operatorname{ellkc}(k) &= \frac{\frac{1}{2}\pi}{\operatorname{agm}(a,b)} = \frac{\frac{1}{2}\pi}{k\operatorname{agm}(a/k,b/k)} = \frac{\frac{1}{2}\pi}{k\operatorname{agm}(a/k,k/a)} = \frac{\frac{1}{2}\pi}{k\operatorname{agm}(1/c,c)}, \\
c &= k/a.
\end{aligned}$$

For hexadecimal arithmetic, we work with arguments a' and b' of half size, so that a' lies in $[\frac{1}{2}, 1]$, avoiding leading zero bits:

$$\begin{aligned} a &= 2a', & b &= 2b', \\ a' &= \tfrac{1}{2} + \tfrac{1}{2}\sqrt{1-k^2}, & b' &= \tfrac{1}{2}k^2/a = \tfrac{1}{4}k^2/a', \\ d' &= b'/k = k/(4a'), \end{aligned}$$

$$\operatorname{ellkc}(k) = \frac{\frac{1}{4}\pi}{\operatorname{agm}(a', b')} = \frac{\frac{1}{4}\pi}{k \operatorname{agm}(a'/k, b'/k)} = \frac{\frac{1}{4}\pi}{k \operatorname{agm}(\frac{1}{4}(1/d'), d')}.$$

Our final computational formulas for $\operatorname{ellkc}(k)$ are stable even for tiny k, as long as c and d' do not underflow, and their reciprocals do not overflow.

When k is tiny, we have $\operatorname{fl}(a) = 2$ and $\operatorname{fl}(a') = 1$, so c underflows when k is smaller than twice the smallest representable number, and d' underflows when k is below four times the smallest number. Thus, intermediate underflow is not an issue until k is within four ulps of the underflow limit.

The design of the extended IEEE 754 arithmetic system ensures that the reciprocal of the smallest normal number is representable, so when c and d' are normal, their reciprocals cannot overflow. However, once they enter the subnormal region, overflow is possible. In all binary formats of the IEEE 754 system, `FP_T_MIN / (1 / FP_T_MAX)` is almost 4, and in all of the decimal formats, it is close to 100. In both cases, the expression is almost β^2, and the reciprocal of the largest finite number is subnormal.

We therefore conclude that, for IEEE 754 arithmetic, $\operatorname{ellkc}(k)$ can be computed stably for k above `FP_T_MIN/4`, but for k values below that, overflow soon occurs in one of the AGM arguments, which in turn produces a return value of Infinity, making the denominator $k \operatorname{agm}(1/c, c)$ evaluate to Infinity as well, producing $\operatorname{fl}(\operatorname{ellkc}(k)) = 0$ for tiny k. That result is completely wrong, because $\operatorname{ellkc}(k)$ is infinite at $k = 0$. The problem is that $\operatorname{ellkc}(k)$ should grow like $\log(1/k)$, so $k \operatorname{agm}(1/c, c)$ should approach zero. The solution, then, is to check for a return value from $\operatorname{agm}(1/c, c)$ that is above, say, half the overflow limit, and in that case, return a value of Infinity without further computation.

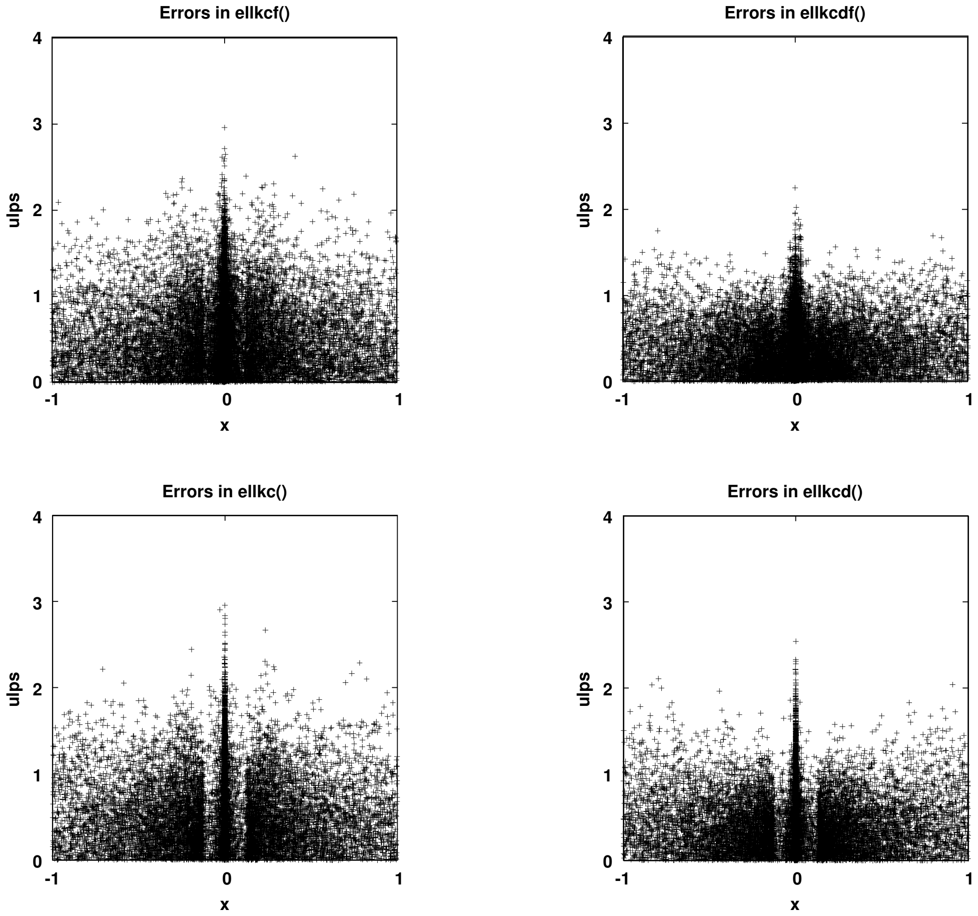
The issues of premature overflow in the arguments disappear if we use the logarithmic limiting form of $\operatorname{ellkc}(k)$ given on page 626, because we then never invoke the AGM algorithm when k is tiny. Otherwise, we should return Infinity, without calculating the AGM, if $|k| < \min(\texttt{FP\_T\_MIN}, 1/\texttt{FP\_T\_MAX})$.

Our code for the scalar version of the complementary complete elliptic integral function of the first kind looks like this:

```
fp_t
ELLKC(fp_t k)
{   /* complementary complete elliptic integral of the FIRST kind */
    volatile fp_t result;
    static fp_t cut_limit = FP(0.);
    static fp_t cut_taylor = FP(0.);
    static fp_t last_k = FP(-2.);       /* out-of-range k */
    static fp_t last_result = FP(0.);
    static int do_init = 1;

    if (do_init)
    {
        cut_limit = SQRT(FP_T_EPSILON);
        cut_taylor = SQRT_TWO * cut_limit;
        do_init = 0;
    }

    if (ISNAN(k))
        result = SET_EDOM(k);
    else if ( (k < -ONE) || (ONE < k) )
        result = SET_EDOM(QNAN(""));
    else if (k == last_k)     /* use history for common K'(k) calls */
        result = last_result;
    else if ( (k == ONE) || (k == -ONE) )
```

```
            result = PI_HALF;
        else if (k == ZERO)
            result = SET_ERANGE(INFTY());
        else
        {
            fp_t a, b, onemxx, t;

            if (k < ZERO)   /* use symmetry to ensure positive argument */
                k = -k;

            if (k <= cut_limit)
            {
                result = LN_4_LO - LOG(k);
                STORE(&result);
                result += LN_4_HI;
            }
            else if ((onemxx = (ONE + k)*(ONE - k), onemxx <= cut_taylor))
            {

#if defined(HAVE_WOBBLING_PRECISION)
                result = PI * (HALF + EIGHTH * onemxx);
#else
                result = PI_HALF * (ONE + FOURTH * onemxx);
#endif /* defined(HAVE_WOBBLING_PRECISION) */

            }
            else
            {

#if defined(HAVE_WOBBLING_PRECISION)
                a = HALF + HALF * SQRT(onemxx);
                b = FOURTH * k / a;
                t = AGM(FOURTH / b, b);
                result = (t > HALF * FP_T_MAX) ? INFTY() :
                         PI_QUARTER / (k * t);
#else
                a = ONE + SQRT(onemxx);
                b = k / a;
                t = AGM(ONE / b, b);
                result = (t > HALF * FP_T_MAX) ? INFTY() :
                         PI_HALF / (k * t);
#endif /* defined(HAVE_WOBBLING_PRECISION) */

            }
        }

        last_k = k;
        last_result = result;

        return (result);
}
```

Figure 20.5 on the facing page shows the measured errors for the complementary function family.


Errors in ellkcf()

Errors in ellkcdf()

Errors in ellkc()

Errors in ellkcd()

Figure 20.5: Errors in the complementary complete elliptic integral functions of the first kind.

20.6 Computing $E(m)$ and $E'(m)$

The terms in the Taylor series for $E(m)$ and $E'(m)$ given on page 629 fall in magnitude, despite the presence of the logarithms of m and $1 - m$, because $\lim_{m \to +0} m \ln(m) = 0$. Their relative magnitudes are not obvious because of the logarithms, but a short investigation in Maple reports relative errors in ulps for the extended IEEE 754 binary and decimal formats:

```
> Digits := 140:
> T1  := proc(m) return 1 end proc:
> T2  := proc(m) return evalf((ln(2) - (1/4)*ln(m) - 1/4)*m) end proc:
> T3  := proc(m) return evalf(((3/8)*ln(2) - (3/32)*ln(m) - 13/64) *
>                              m**2) end proc:
> R21 := proc(m) return evalf(T2(m) / T1(m)) end proc:
> R32 := proc(m) return evalf(T3(m) / T2(m)) end proc:
> for t in {24, 53, 64, 113, 237} do
>     eps := 2**(-t+1):    m := eps:
>     printf("%3d  %6.3f  %6.3f\n", t, R21(m) / eps, R32(m) / eps)
> end do:
 24   4.429   0.350
 53   9.454   0.363
 64  11.360   0.365
```

```
113  19.851   0.369
237  41.339   0.372

> for t in {7, 16, 35, 70} do
>     eps := 10**(-t+1):    m := eps:
>     printf("%3d  %6.3f  %6.3f\n", t, R21(m) / eps, R32(m) / eps)
> end do:
  7   3.897   0.347
 16   9.078   0.363
 35  20.015   0.370
 70  40.163   0.372
```

Thus, a three-term sum should produce correctly rounded results for $m < \epsilon$, or equivalently, $k < \sqrt{\epsilon}$.

A slight modification of the loops shows that smaller cutoffs of $\epsilon/2^7$ and $\epsilon/10^2$ reduce the second term to less than 0.42ϵ, allowing a two-term sum. However, that is a severe reduction of the range where the truncated Taylor series applies.

A further small change to the loops shows that four terms suffice with a cutoff of $m < \frac{1}{4}\sqrt{\epsilon}$, where the first three terms produce an error below 0.48ϵ, and the fourth term is needed for correct rounding.

Outside the Taylor-series region, we compute the complete elliptic integral of the second kind from the relation [AS64, page 599]

$$\frac{K(m) - E(m)}{K(m)} = \tfrac{1}{2}(c_0^2 + 2c_1^2 + 4c_2^2 + \cdots + 2^j c_j^2 + \cdots) = \sigma(m),$$

where the coefficients in the summation are

$$c_0 = \sin(\alpha) = k, \qquad c_j = \tfrac{1}{2}(a_{j-1} - b_{j-1}), \qquad \text{for } j = 1, 2, 3, \ldots.$$

We then have the final result

$$\boldsymbol{E(m) = K(m) - K(m)\sigma(m).}$$

Both $E(m)$ and $K(m)$ are positive, but $E(m)$ remains small, whereas $K(m)$ has a pole at $m = 1$. Thus, near that pole, $E(m)$ is determined by the difference of two large numbers. In binary arithmetic, the subtraction causes significance loss for $\sigma(m) > \frac{1}{2}$. Maple readily finds the m and k values for which that happens:

```
> Digits := 40:
> K := proc(m) return EllipticK(sqrt(m)) end proc:
> E := proc(m) return EllipticE(sqrt(m)) end proc:
> fsolve((K(m) - E(m))/K(m) = 1/2, m = 0 .. 1);
> sqrt(%);
```

The output shows these results:

$$m = 0.826\,114\,765\,984\,970\,336\,177\,373\,238\,600\,756\,406\,034\,\ldots,$$
$$k = 0.908\,908\,557\,548\,541\,478\,236\,118\,908\,744\,793\,504\,901\,\ldots.$$

Thus, there is significance loss in computing elle(k) for $|k|$ in the approximate range $[0.9, 1]$, and the loss increases as $k \to 1$. A partial solution is a relation that allows shifting the argument, at the expense of one extra elliptic function evaluation:

$$z = \sqrt{1 - m}, \qquad E(m) = (1 + z)E\left(((1 - z)/(1 + z))^2\right) - zK(m).$$

That remaps m in the range $[0.82, 1]$ nonuniformly onto $[0.16, 1]$, reducing, but not entirely eliminating, the region of significance loss.

We require the vector version of the AGM to get the c_j coefficients, but our replacement of agm$(1, \sqrt{1-k^2})$ with agm$(1 + k, 1 - k)$ produces different c_j values. However, a numerical investigation shows that c_{j+1} from the second iteration matches the original coefficient c_j, and a symbolic computation in Maple confirms that.

The code for the complete elliptic integral function of the second kind is then a direct application of our formulas, with extra care to sum the $\sigma(m)$ series in reverse order, and shift the coefficient subscripts upward:

```
fp_t
ELLE(fp_t k)
{   /* complete elliptic integral of the SECOND kind */
    fp_t result;
    static fp_t cut_remap = FP(0.);
    static fp_t cut_taylor = FP(0.);
    static fp_t last_k = FP(1.);
    static fp_t last_result = FP(1.);   /* E(1) = 1 */
    static int do_init = 1;

    if (do_init)
    {
        cut_remap = FP(0.908);
        cut_taylor = HALF * SQRT(SQRT(FP_T_EPSILON));
        do_init = 0;
    }

    if (ISNAN(k))
        result = SET_EDOM(k);
    else if ( (k < -ONE) || (ONE < k) )
        result = SET_EDOM(QNAN(""));
    else if (k == last_k)     /* use history for common E(k) calls */
        result = last_result;
    else if ( (k == ONE) || (k == -ONE) )
        result = ONE;
    else if (k == ZERO)
        result = PI_HALF;
    else
    {
        fp_t a[MAXAGM], b[MAXAGM], c[MAXAGM], K_of_m, sigma, two_to_j;
        int j, need;

#if defined(HAVE_WOBBLING_PRECISION)
        fp_t half_k;
#endif /* defined(HAVE_WOBBLING_PRECISION) */

        if (k < ZERO)   /* use symmetry to ensure positive argument */
            k = -k;

        if ((ONE - k) < cut_taylor)
        {   /* sum four-term Taylor series in Horner form */
            fp_t ln_z, sum, z;

            z = (ONE + k) * (ONE - k);  /* z == 1 - k**2 == 1 - m */
            ln_z = LOG1P(-k * k);
            sum = (FP(60.) * LN_2 - FP(15.) * ln_z - FP(36.)) * FP(0.001953125);
            sum = sum * z + (FP(24.) * LN_2 - FP(6.) * ln_z - FP(13.)) * FP(0.0078125);
            sum = sum * z + (LN_2 - FOURTH * ln_z - FOURTH) * HALF;
            sum = sum * z + HALF;
            result = sum + sum;
        }
        else if (k > cut_remap)
        {
            fp_t y, z;

            z = SQRT((ONE - k) * (ONE + k));
            y = (ONE - z) / (ONE + z);
```

```
                result = (ONE + z) * ELLE(y) - z * ELLK(k);
            }
            else
            {
                c[0] = ZERO;
                c[1] = ZERO;
                c[2] = ZERO;        /* need in case need < 3 on return */

#if defined(HAVE_WOBBLING_PRECISION)
                half_k = HALF * k;
                K_of_m = PI_QUARTER / VAGM(HALF + half_k, HALF - half_k,
                                           half_k, (int)elementsof(a),
                                           a, b, c, &need);
#else
                K_of_m = PI_HALF / VAGM(ONE + k, ONE - k, k,
                                        (int)elementsof(a),
                                        a, b, c, &need);
#endif /* defined(HAVE_WOBBLING_PRECISION) */

                sigma = ZERO;
                two_to_j = TWO;

                for (j = 2; j < (need - 1); ++j)
                {
                    two_to_j += two_to_j;
                    sigma += two_to_j * c[j + 1] * c[j + 1];
                }

                sigma += TWO * c[1 + 1] * c[1 + 1];
                sigma += c[0 + 1] * c[0 + 1];

#if defined(HAVE_WOBBLING_PRECISION)
                sigma += sigma;
#else
                sigma *= HALF;
#endif /* defined(HAVE_WOBBLING_PRECISION) */

                result = K_of_m - sigma * K_of_m;
            }
        }

        last_k = k;
        last_result = result;

        return (result);
}
```

We rewrote the Taylor series sum in a form that is optimal for hexadecimal arithmetic so as not to require base-dependent code. However, in the hexadecimal case, the iteration has half-size initial values. The output coefficients c_j are then also halved, and the $\sigma(m)$ sum is consequently a quarter of the correct value. Thus, instead of multiplying the sum by a half, we have to double it.

Figure 20.6 on the next page shows the measured errors for the `ELLE()` function family. Although the peak errors are above our library goal of two ulps, most are well below that value.

The computation of $E'(m)$ differs from that of $E(m)$ only in the starting values of the AGM iteration: b_0 and c_0 are swapped. That means that we can use the original $\operatorname{agm}(1, k)$ with $c_0 = \sqrt{1-k^2}$, so the coefficient index shift is not needed. There may be accuracy loss in forming c_0, but its value is not used, except for assignment, inside the vector AGM routine, and we only require c_0^2 for the $\sigma(m)$ sum. That square can be computed accurately from $(1+k)(1-k)$.

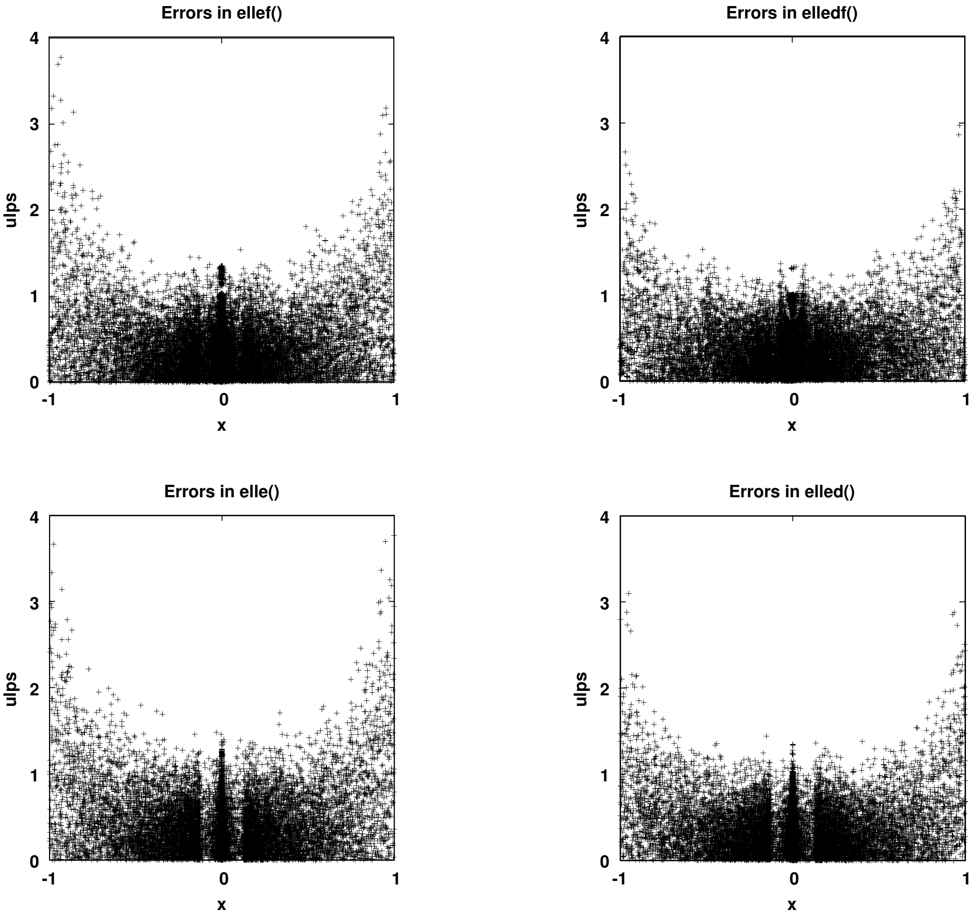


Figure 20.6: Errors in the complete elliptic integral functions of the second kind.

As with $E(m)$, there is significance loss in forming $K'(m) - \sigma'(m)K'(m)$, but now as $m \to 0$, and a partial solution is supplied by another argument-shift relation:

$$z = 2\sqrt{k}/(1+k), \qquad E'(m) = (1+k)E'(z^2) - kK'(m).$$

Here is the code for the complementary complete elliptic integral of the second kind, incorporating the Taylor series, argument shift, and the AGM algorithm:

```
fp_t
ELLEC(fp_t k)
{   /* complementary complete elliptic integral of the SECOND kind */
    fp_t result;
    static fp_t cut_remap = FP(0.);
    static fp_t cut_taylor = FP(0.);
    static fp_t last_k = FP(-2.);       /* out-of-range k */
    static fp_t last_result = FP(0.);
    static int do_init = 1;

    if (do_init)
    {
        cut_remap = FP(0.25);
        cut_taylor = HALF * SQRT(SQRT(FP_T_EPSILON));
        do_init = 0;
    }

    if (ISNAN(k))
        result = SET_EDOM(k);
    else if ( (k < -ONE) || (ONE < k) )
        result = SET_EDOM(QNAN(""));
    else if (k == last_k)     /* use history for common E(k) calls */
        result = last_result;
    else if ( (k == ONE) || (k == -ONE) )
        result = PI_HALF;
    else if (k == ZERO)
        result = ONE;
    else
    {
        fp_t a[MAXAGM], b[MAXAGM], c[MAXAGM], KC_of_m, sigma, two_to_j;
        int j, need;

#if defined(HAVE_WOBBLING_PRECISION)
        fp_t half_k;
#endif /* defined(HAVE_WOBBLING_PRECISION) */

        if (k < ZERO)   /* use symmetry to ensure positive argument */
            k = -k;

        if (k < cut_taylor)
        {   /* sum four-term Taylor series in Horner form */
            fp_t ln_m, m, sum;

            m = k * k;
            ln_m = TWO * LOG(k);
            sum = (FP(60.) * LN_2 - FP(15.) * ln_m - FP(36.))
                  * FP(0.001953125);
            sum = sum * m + (FP(24.) * LN_2 - FP(6.) * ln_m - FP(13.))
                  * FP(0.0078125);
            sum = sum * m + (LN_2 - FOURTH * ln_m - FOURTH) * HALF;
            sum = sum * m + HALF;
            result = sum + sum;
        }
        else if (k < cut_remap)
        {
            fp_t z, onepx;

            onepx = ONE + k;
            z = TWO * SQRT(k) / onepx;
            result = onepx * ELLEC(z) - k * ELLKC(k);
        }
        else
        {
#if defined(HAVE_WOBBLING_PRECISION)
            half_k = HALF * k;
            KC_of_m = PI_QUARTER / VAGM(HALF, half_k, ZERO,
                                        (int)elementsof(a),
                                        a, b, c, &need);
#else
            KC_of_m = PI_HALF / VAGM(ONE, k, ZERO, (int)elementsof(a),
                                     a, b, c, &need);
#endif /* defined(HAVE_WOBBLING_PRECISION) */

            sigma = ZERO;
            two_to_j = TWO;

            for (j = 2; j < need; ++j)
            {
                two_to_j += two_to_j;
                sigma += two_to_j * c[j] * c[j];
            }

            sigma += TWO * c[1] * c[1];

#if defined(HAVE_WOBBLING_PRECISION)
            /* replace c[0]**2 by accurate (1/2 - (k/2)**2) =
                (1/2 - k/2) * (1/2 + k/2) */
            sigma += (HALF - half_k) * (HALF + half_k);
            sigma += sigma;
#else
            /* replace c[0]**2 by accurate (1 - k**2) =
               (1 - k) * (1 + k) */
            sigma += (ONE - k) * (ONE + k);
            sigma *= HALF;
#endif /* defined(HAVE_WOBBLING_PRECISION) */

            result = KC_of_m - sigma * KC_of_m;
        }
    }

    last_k = k;
    last_result = result;

    return (result);
}
```

In the call to VAGM(), we supply c_0 as zero. There is no need to compute its true value, because we evaluate c_0^2 accurately later. As in the code for ELLE(), we provide special handling for hexadecimal arithmetic.

Figure 20.7 on the following page shows the measured errors in the ELLEC() function family. The peak errors are above the mathcw library goal of two ulps, and worse than those for the ELLK(), ELLKC(), and ELLE() families.

20.7 Historical algorithms for elliptic integrals

There are many published articles and book sections with algorithms for computing the elliptic integral functions. Here is a sampler:

- In a book published before high-level programming languages were invented [Has55, pages 170–175], Hastings uses low-order polynomial approximations of degrees two to four of the form

$$K(m) \approx \mathcal{P}_K(\eta) + \mathcal{Q}_K(\eta)\log(\eta), \qquad E(m) \approx \mathcal{P}_E(\eta) + \mathcal{Q}_E(\eta)\log(\eta),$$
$$m = k^2, \qquad \eta = 1 - k^2, \qquad \text{for } \eta \text{ in } (0, 1],$$

  (η is the lowercase Greek letter *eta*) to obtain four to eight decimal digits of accuracy. Given the argument range of the logarithm, it is now preferable to replace $\log(\eta)$ by the more accurate $\operatorname{log1p}(-m)$, a function that did not enter mathematical libraries until more than three decades later. For better accuracy near $m \approx 1$, switch to the complementary function form, using an accurate argument η directly, rather than computing it from $1 - m$. The *Handbook of Mathematical Functions* uses the Hastings polynomials [AS64, pages 591–592].

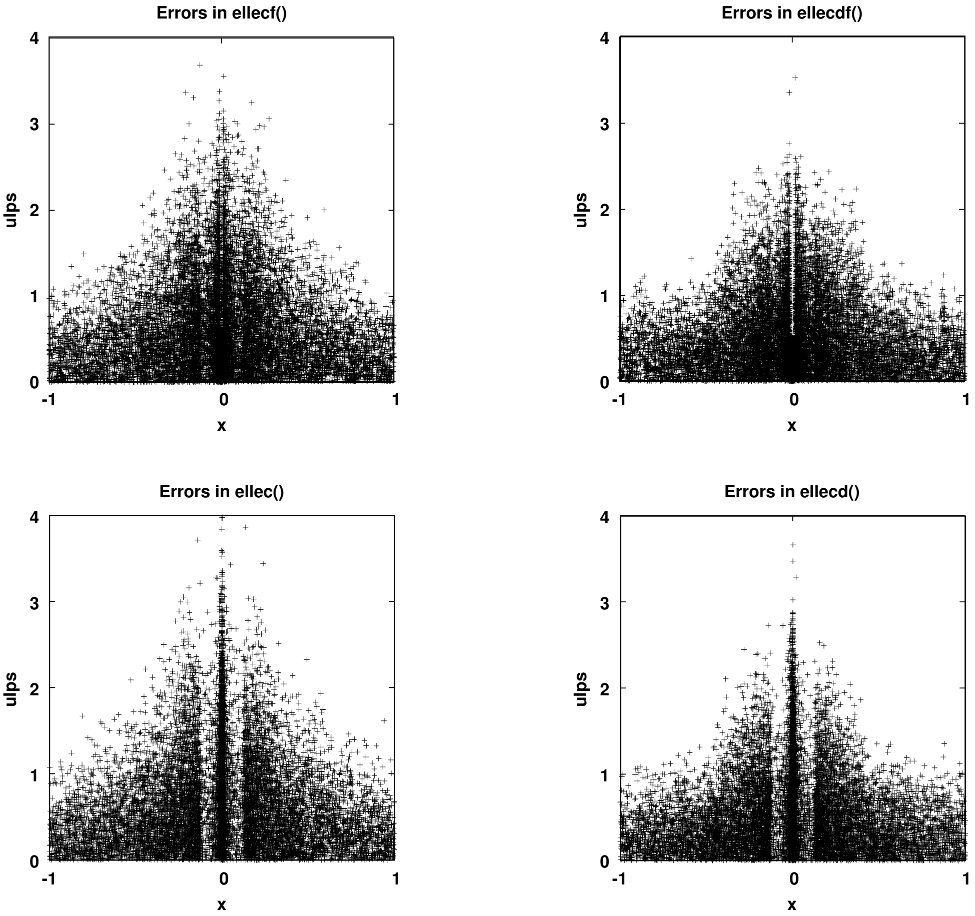


Figure 20.7: Errors in the complementary complete elliptic integral functions of the second kind.

- Some of Cody's earliest works deal with the complete elliptic integral functions, extending Hastings' formula to polynomial degrees of two to ten, obtaining nearly 18 decimal digits of accuracy [FC64, Cod65a, Cod65b].

  Given that two polynomials and a logarithm must be evaluated, and the polynomials depend on the host precision, it would appear that Cody's approach has no significant advantages over the simple AGM algorithm, which needs only elementary operations and square roots.

- The later book by Hart and others [HCL$^+$68, page 154] gives essentially the same polynomials as those of Cody, but with a few more decimal places.

- For small arguments, Moshier's Cephes library [Mos89, pages 387–395] uses a two-term Taylor series, and otherwise, uses the Hastings-style formula with polynomials of degree 10, but their coefficients differ from those of Cody. Cephes also includes code for the incomplete elliptic integral functions of the first and second kinds.

- Two other books [Tho97, ZJ96] that we have mentioned elsewhere use the AGM algorithm, but without consideration of the extra issues that our code handles.

- Gil, Segura, and Temme [GST07, pages 344–347] have a brief discussion of elliptic integrals.

- Carlson has published many articles, and a book, on special functions, with a particular emphasis on elliptic integrals [Car77, Chapter 9]. We examine his techniques in the next section.

- Fukushima recently published partial code [Fuk10] for a fast algorithm for computing the incomplete elliptic integral function of the first kind, $F(\phi|m)$, as a Fortran function `elf(`ϕ, ϕ_c, m_c`)`, with $\phi_c = \frac{1}{2}\pi - \phi$, $m_c = 1 - m$, and $m = k^2$. He also supplies short implementations of two of the inverse Jacobian elliptic functions that we discuss later in **Section 20.16** on page 664. This author completed the missing code and supplied an interface to the mathcw library routine `ellk(k)` to provide the $K(k)$ values needed by Fukushima's algorithm. Limited testing shows agreement to within one or two ulps between the two approaches, and that Fukushima's code can be several times faster than ours. However, it is specialized to just a single elliptic function, whereas ours are built on Carlson's general framework for handling the elliptic functions of all three kinds. One of Fukushima's inverse Jacobian elliptic functions fails when ϕ_c is tiny, and the other when its arguments are both near one, but that can be handled with additional code that checks for extreme values.

  Earlier papers by Fukushima discuss fast computation of the complete elliptic integral functions and the Jacobian elliptic functions, and compare his methods with previous work, but do not supply source code [FI94, Fuk09a, Fuk09b].

20.8 Auxiliary functions for elliptic integrals

The function defined by

$$R(a;\, b_1, b_2, \ldots, b_n;\, z_1, z_2, \ldots, z_n) = \frac{\Gamma(a+c)}{\Gamma(a)\Gamma(c)} \int_0^\infty t^{c-1}(t+z_1)^{-b_1}(t+z_2)^{-b_2}\cdots(t+z_n)^{-b_n}\,dt,$$
$$c = (b_1 + b_2 + \cdots + b_n) - a.$$

is called a *hypergeometric R-function*. It is an elliptic integral when exactly *four* of the parameters a, c, b_1, b_2, ..., b_n are half integers, and all of the remaining parameters are integers. The arguments z_1, z_2, ..., z_n are arbitrary complex numbers, but when they have real values, they are required to be nonnegative, because otherwise, the integral diverges or is undefined. When any of the b_k parameters are equal, the function is *symmetric* in the corresponding z_k arguments. The elliptic integral is called *complete* when both a and c are half-odd integers, or when one of the z_k arguments is zero.

Legendre showed that exactly *four* elliptic integrals are linearly independent, and all others can be reduced by argument transformations to linear combinations of the basic four, with polynomial coefficients. There is unlimited freedom in the choice of those four functions, and in the early work on elliptic integrals, there was not yet enough knowledge about them to make an optimal choice. Legendre's selection of those four as $D(\phi, k)$, $E(\phi, k)$, $F(\phi, k)$, and $\Pi(n, \phi, k)$ is now, in hindsight, regrettable, because they make analysis unnecessarily complicated.

In several decades of work, Carlson and his students have shown that a better choice of the four basic elliptic integrals is one that maximizes argument symmetry. In his book [Car77, Chapter 9], Carlson recommends this basic set:

$$\begin{aligned}
(xyz)^{-1/2} &= R(+\tfrac{3}{2};\, \tfrac{1}{2}, \tfrac{1}{2}, \tfrac{1}{2};\, x, y, z), \\
R_F(x, y, z) &= R(+\tfrac{1}{2};\, \tfrac{1}{2}, \tfrac{1}{2}, \tfrac{1}{2};\, x, y, z), && \textit{elliptic integral of first kind,} \\
&= \tfrac{2}{3} R_H(x, y, z, 0), \\
R_G(x, y, z) &= R(-\tfrac{1}{2};\, \tfrac{1}{2}, \tfrac{1}{2}, \tfrac{1}{2};\, x, y, z), && \textit{elliptic integral of second kind} \\
R_H(x, y, z, \rho) &= R(+\tfrac{1}{2};\, \tfrac{1}{2}, \tfrac{1}{2}, \tfrac{1}{2}, 1;\, x, y, z, \rho), && \textit{elliptic integral of third kind.}
\end{aligned}$$

In the last function, ρ is the lowercase Greek letter *rho*. It is chosen from a different alphabet because it does not participate in any symmetry relations. The b_k values are all equal in Carlson's selection, so the functions are symmetric in the arguments x, y, and z. In addition, the parameter choice reduces the ratio of gamma functions to simple rational numbers, possibly involving an additional factor of π.

Carlson's book [Car77, page 97] introduces a second notation for the R-functions, moving the leading argument of the original form to a *negated subscript*:

$$R_{-a}(b_1, b_2, \ldots, b_n;\, z_1, z_2, \ldots, z_n) = R(a;\, b_1, b_2, \ldots, b_n;\, z_1, z_2, \ldots, z_n).$$

In that alternate form, *add* the subscript to the parameters b_1 through b_n, then subtract 1 to find the power of t in the integrand. The out-of-place sign-flipped function argument is confusing, so in this book, we keep the original notation.

Zill and Carlson [ZC70] propose a symmetric function defined by

$$R_J(x,y,z,\rho) = R(+\tfrac{3}{2};\ \tfrac{1}{2}, \tfrac{1}{2}, \tfrac{1}{2}, 1;\ x,y,z,\rho)$$

that can sometimes be a useful alternative to $R_H(x,y,z,\rho)$. They also define a few other functions for mathematical convenience:

$$\begin{aligned}
R_C(x,y) &= R(+\tfrac{1}{2}; \tfrac{1}{2}, \tfrac{1}{2}; x, y), \\
&= R_F(x,y,y), && \text{not } \textit{symmetric in } x \textit{ and } y, \\
R_D(x,y,z) &= R(+\tfrac{3}{2}; \tfrac{1}{2}, \tfrac{1}{2}, \tfrac{3}{2}; x,y,z), \\
&= R_J(x,y,z,z), && \textit{symmetric} \text{ only } \textit{in } x \textit{ and } y, \\
R_E(x,y) &= R(-\tfrac{1}{2};\ \tfrac{1}{2}, \tfrac{1}{2};\ x, y) \\
&= \tfrac{4}{\pi} R_G(x,y,0), \\
R_K(x,y) &= R(+\tfrac{1}{2};\ \tfrac{1}{2}, \tfrac{1}{2};\ x, y) \\
&= \tfrac{2}{\pi} R_F(x,y,0) \\
&= \frac{2}{\pi\sqrt{y}} K\left(\sqrt{\left|\frac{y-x}{y}\right|}\right) \\
&= \tfrac{1}{2} R_L(x,y,0) \\
&= \tfrac{4}{3\pi} R_H(x,y,0,0), \\
R_L(x,y,\rho) &= R(+\tfrac{1}{2};\ \tfrac{1}{2}, \tfrac{1}{2}, 1;\ x,y,\rho) \\
&= \tfrac{8}{3\pi} R_H(x,y,0,\rho), \\
R_M(x,y,\rho) &= R(+\tfrac{3}{2};\ \tfrac{1}{2}, \tfrac{1}{2}, 1;\ x,y,\rho) \\
&= R_L(x,y,xy/\rho)/\rho \\
&= \tfrac{4\pi}{3} R_J(x,y,0,\rho).
\end{aligned}$$

The last four of those functions are symmetric in x and y.

If we expand the general R-functions in those definitions, we can write most of the auxiliary functions in integral form, and relate them to others, like this [Car70, ZC70, Car77, Car79, CN81, Car95]:

$$\begin{aligned}
R_C(x,y) &= \tfrac{1}{2} \int_0^\infty (t+x)^{-1/2} (t+y)^{-1}\, dt, \\
R_D(x,y,z) &= \tfrac{3}{2} \int_0^\infty (t+x)^{-1/2} (t+y)^{-1/2} (t+z)^{-3/2}\, dt, \\
R_F(x,y,z) &= \tfrac{1}{2} \int_0^\infty (t+x)^{-1/2} (t+y)^{-1/2} (t+z)^{-1/2}\, dt \\
&= \frac{1}{\sqrt{z-x}} F\left(\sin(\operatorname{acos}(\sqrt{x/z})), \sqrt{\frac{z-y}{z-x}}\right), \\
R_H(x,y,z,\rho) &= \tfrac{3}{4} \int_0^\infty t(t+x)^{-1/2} (t+y)^{-1/2} (t+z)^{-1/2} (t+\rho)^{-1}\, dt \\
&= \tfrac{3}{2} R_F(x,y,z) - \tfrac{1}{2} \rho R_J(x,y,z,\rho), \\
R_J(x,y,z,\rho) &= \tfrac{3}{2} \int_0^\infty (t+x)^{-1/2} (t+y)^{-1/2} (t+z)^{-1/2} (t+\rho)^{-1}\, dt \\
&= \frac{1}{(x-y)(x-z)} \left(3x R_F(x,y,z) - 6 R_G(x,y,z) + 3\sqrt{\frac{yz}{x}}\right), \\
R_K(x,y) &= \tfrac{1}{\pi} \int_0^\infty t^{-1/2} (t+x)^{-1/2} (t+y)^{-1/2}\, dt,
\end{aligned}$$

$$\begin{aligned}
R_L(x,y,\rho) &= \tfrac{2}{\pi}\int_0^\infty t^{+1/2}(t+x)^{-1/2}(t+y)^{-1/2}(t+\rho)^{-1}\,dt,\\
R_M(x,y,\rho) &= \tfrac{2}{\pi}\int_0^\infty t^{-1/2}(t+x)^{-1/2}(t+y)^{-1/2}(t+\rho)^{-1}\,dt.
\end{aligned}$$

The integrals from the R-function definitions of $R_E(x,y)$ and $R_G(x,y,z)$ diverge, so those two functions must be computed from this alternate integral, which *does* converge:

$$R_G(x,y,z) = \tfrac{1}{4}\int_0^\infty t(t+x)^{-1/2}(t+y)^{-1/2}(t+z)^{-1/2}\left[\frac{x}{t+x}+\frac{y}{t+y}+\frac{z}{t+z}\right]dt.$$

The function $R_G(x,y,z)$ can be written in terms of two other auxiliary functions like this:

$$R_G(x,y,z) = \tfrac{1}{2}\left(zR_F(x,y,z) - \tfrac{1}{3}(x-z)(y-z)R_D(x,y,z) + \sqrt{\frac{xy}{z}}\right).$$

When $z = 0$, use symmetry to exchange arguments of $R_G(x,y,z)$ in order to avoid a zero denominator in the final square root.

The function $R_G(x,y,z)$ can also be represented by a double integral that is simplified by the introduction of three intermediate variables:

$$\begin{aligned}
&\ell = \sin(\theta)\cos(\phi), \qquad m = \sin(\theta)\sin(\phi), \qquad n = \cos(\theta),\\
&R_G(x,y,z) = \tfrac{2}{\pi}\int_0^{\pi/2}\int_0^{\pi/2}\sqrt{\ell^2x+m^2y+n^2z}\,\sin(\theta)\,d\theta\,d\phi.
\end{aligned}$$

The function $R_E(x,y)$ also has two alternate forms:

$$\begin{aligned}
R_E(x,y) &= \tfrac{2}{\pi}\int_0^{\pi/2}\sqrt{x(\sin(\theta))^2+y(\cos(\theta))^2}\,d\theta,\\
&= \frac{2\sqrt{y}}{\pi}E\left(\sqrt{\left|\frac{y-x}{y}\right|}\right).
\end{aligned}$$

The auxiliary functions satisfy normalization rules that are computationally useful for checking their software implementations, and play a key role in the computation of four of them:

$$\begin{aligned}
R_C(x,x) &= x^{-1/2}, & R_H(x,x,x,x) &= x^{-1/2},\\
R_D(x,x,x) &= x^{-3/2}, & R_J(x,x,x,x) &= x^{-3/2},\\
R_E(x,x) &= x^{+1/2}, & R_K(x,x) &= x^{-1/2},\\
R_F(x,x,x) &= x^{-1/2}, & R_L(x,x,x) &= x^{-1/2},\\
R_G(x,x,x) &= x^{+1/2}, & R_M(x,x,x) &= x^{-3/2}.
\end{aligned}$$

More generally, the R-function satisfies this normalization rule:

$$R(a;\, b_1,b_2,\ldots,b_n;\, z,z,\ldots,z) = z^{-a}.$$

Argument-scaling relations allow software to avoid premature underflow and overflow:

$$\begin{aligned}
R_C(sx,sy) &= s^{-1/2}R_C(x,y), & R_H(sx,sy,sz,s\rho) &= s^{-1/2}R_H(x,y,z,\rho),\\
R_D(sx,sy,sz) &= s^{-3/2}R_D(x,y,z), & R_J(sx,sy,sz,s\rho) &= s^{-3/2}R_J(x,y,z,\rho),\\
R_E(sx,sy) &= s^{+1/2}R_E(x,y), & R_K(sx,sy) &= s^{-1/2}R_K(x,y),\\
R_F(sx,sy,sz) &= s^{-1/2}R_F(x,y,z), & R_L(sx,sy,s\rho) &= s^{-1/2}R_L(x,y,\rho),\\
R_G(sx,sy,sz) &= s^{+1/2}R_G(x,y,z), & R_M(sx,sy,s\rho) &= s^{-3/2}R_M(x,y,\rho).
\end{aligned}$$

The general R-function scaling relation follows from its normalization rule:

$$R(a;\, b_1, b_2, \ldots, b_n;\, sz, sz, \ldots, sz) = s^{-a} R(a;\, b_1, b_2, \ldots, b_n;\, z, z, \ldots, z).$$

Argument addition rules, where $xy = ab$, provide further checks for software, and their symmetries are emphasized by vertical alignment of the sums on the left-hand sides:

$$\begin{aligned}
R_C(a, x+a) &+ \\
R_C(b, x+b) &= R_C(0, x), \\
R_D(a, x+a, y+a) &+ \\
R_D(b, x+b, y+b) &= R_D(0, x, y) - \frac{3}{y\sqrt{x+y+a+b}}, \\
R_F(x+a, y+a, a) &+ \\
R_F(x+b, y+b, b) &= R_F(x, y, 0), \\
R_J(x+a, y+a, a, \rho+a) &+ \\
R_J(x+b, y+b, b, \rho+b) &= R_J(x, y, 0, \rho) - 3R_C(a, b).
\end{aligned}$$

Another rule relates functions of permuted arguments:

$$R_D(x, y, z) + R_D(y, z, x) + R_D(z, x, y) = \frac{3}{\sqrt{xyz}}.$$

In the special case of equal arguments, $x = y = z$, that reduces to the normalization relation $R_D(x, x, x) = x^{-3/2}$.

Upper and lower bounds on function values [CN81, page 398] are relevant for floating-point calculation:

$$\begin{aligned}
M &= \max(x, y, z), & m &= \min(x, y, z), & M^{-1/2} &\le R_F(x, y, z) \le 2m^{-1/2}, \\
M_1 &= \max(M, \rho), & m_1 &= \min(m, \rho), & M_1^{-3/2} &\le R_J(x, y, z, \rho) \le 5m_1^{-3/2}.
\end{aligned}$$

If we recall the relations between function pairs, then the first of them shows that $R_C(x, y)$ and $R_F(x, y, z)$ are defined over the entire floating-point range. The second shows that $R_D(x, y, z)$ and $R_J(x, y, z, \rho)$ are subject to overflow for large arguments, and underflow for small arguments.

All of the relations that we have shown so far also apply when the arguments are complex, but there are additional considerations, and issues of numerical stability [Car95]. In this book, we consider only real versions of the ten elliptic auxiliary functions.

20.9 Computing the elliptic auxiliary functions

In the previous section, we displayed lots of mathematical relations, but did not show how they can be turned into an effective computational algorithm for the elliptic auxiliary functions. To do so, we need just one more relation, known as the *duplication rule*:

$$\begin{aligned}
\lambda &= \sqrt{xy} + \sqrt{yz} + \sqrt{xz}, && \textit{sum of geometric means,} \\
R_F(x, y, z) &= 2R_F(x+\lambda, y+\lambda, z+\lambda), && \textit{then apply scaling rule,} \\
&= R_F(\tfrac{1}{4}(x+\lambda), \tfrac{1}{4}(y+\lambda), \tfrac{1}{4}(z+\lambda)), && \textit{duplication rule.}
\end{aligned}$$

Carlson's key insight is the understanding of the implications of that rule. On the right-hand side, we can consider the three arguments to be updated values of the left-hand side arguments, and iterate the relation:

$$x \leftarrow \tfrac{1}{4}(x+\lambda), \qquad y \leftarrow \tfrac{1}{4}(y+\lambda), \qquad z \leftarrow \tfrac{1}{4}(z+\lambda).$$

The iteration makes the arguments tend to a common value. If they were exactly equal, we could then apply the normalization rule to find the function value with just one more square root.

Carlson shows that the relative difference between any argument x and the mean of all three arguments, $\mu = \frac{1}{3}(x+y+z)$, falls like $4^{-n} = 2^{-2n}$ after n steps. Convergence is therefore linear, gaining just *two bits* per iteration.

To speed it up, he proposes using only a few iterations, until the arguments have similar magnitudes, and then switching to a Taylor-series expansion of the integral.

We leave the messy details of the derivation of the Taylor expansion to his article [Car79, §5], and simply state his critical result: apart from an outer scale factor, a fifth-order expansion has a leading term of 1, and a smallest term whose magnitude is about 2^{-12n}. Thus, after ten iterations, that term is smaller than the machine epsilon of the 128-bit IEEE 754 formats (34 decimal digits), so only a few iterations are needed in practice. Each iteration requires just six multiplies, three adds, and three square roots. When the latter are done in hardware, each costs about as much one divide, or three to ten multiplies. Like the AGM algorithm discussed in **Section 20.1** on page 619, the duplication-rule algorithm is not self correcting: errors accumulate, so it is important to keep the iteration counts small.

The duplication-rule code is relatively fast and compact. It requires only small exactly representable constants, needs neither higher intermediate precision nor polynomial approximations, and uses only the low-level operations of add, multiply, and square root. It is thus ideal for a library that supports many different floating-point architectures and precisions.

Carlson shows that similar algorithms can be derived for the other three auxiliary functions. We implement them in the mathcw library as the function families ELLRC(), ELLRD(), ELLRF(), and ELLRJ(), following Carlson's most recent work that recommends the set $R_C(x,y)$, $R_D(x,y,z)$, $R_F(x,y,z)$, and $R_J(x,y,z,\rho)$ as the basic four elliptic functions. Because of their prior use, we also implement the remaining auxiliary functions with the families ELLRE(), ELLRG(), ELLRH(), ELLRK(), ELLRL(), and ELLRM().

The products under the square roots in the formula for λ are subject to premature overflow or underflow, but we eliminate that problem by using the argument-scaling relations. To give a flavor of how little code is actually needed, here is the main block of the function for computing the simplest of them, $R_C(x,y)$, taken from the file ellrcx.h:

```
fp_t arg_max, mu, s;
int n, n_scale;
static const int NMAX = 35;

arg_max = MAX(x, y);
n_scale = (int)LOGB(arg_max) + 1;

if (IS_ODD(n_scale))            /* ensure that n_scale is even */
    n_scale++;

x = SCALBN(x, -n_scale);        /* now x is in [1/B**2, 1), or is 0 */
y = SCALBN(y, -n_scale);        /* now y is in [1/B**2, 1) */

for (n = 0; n < NMAX; ++n)
{
    fp_t lambda;

    mu = (x + y + y) / THREE;
    s = y - mu;                 /* s/mu is relative error for loop exit test */

    if (QABS(s) < (mu * TOL))   /* early exit at convergence */
        break;

    lambda = SQRT(x * y);
    lambda += lambda + y;
    x = (x + lambda) * FOURTH;
    y = (y + lambda) * FOURTH;
}

s = (y - x) / (THREE * mu);

if (QABS(s) < HALF)             /* Taylor series block */
{
    fp_t r, s_2, s_3, s_4, s_5, s_6, sum, t;
    s_2 = s * s;
    s_3 = s_2 * s;
    s_4 = s_2 * s_2;
    s_5 = s_2 * s_3;
    s_6 = s_3 * s_3;
    r = (FP(159.) / FP(208.)) * s_6;
    sum = r;
    sum += (FP(9.) / FP(22.)) * s_5;
    sum += (FP(3.) / FP( 8.)) * s_4;
    sum += (FP(1.) / FP( 7.)) * s_3;
    sum += (FP(3.) / FP(10.)) * s_2;
    t = RSQRT(mu);

    result = SCALBN(t + t * sum, -n_scale / 2);
}
else                            /* should NEVER happen */
    result = QNAN("");
```

The corresponding code for the other three functions is somewhat longer, but even for the most difficult function, $R_J(x, y, z, \rho)$, the code needs fewer than 75 lines. The remaining auxiliary functions are simply wrappers around calls to others of the set of ten, with initial checks for arguments that are NaN, Infinity, or negative.

Although we predicted the maximum number of iterations needed, the early convergence behavior depends on the relative sizes of the arguments, so the iteration count is data dependent. The published Fortran code [CN81] therefore computes the largest relative difference between the arguments and their mean, and compares that against a user-provided tolerance to permit an early exit from the loop.

Our implementation improves on the Fortran code by setting the tolerance in a one-time initialization block according to the floating-point precision, and using a multiply instead of a divide in the loop-exit test. Our code also includes exact argument scaling so that neither premature underflow nor premature overflow is possible. That reduces the number of square roots, because there is then no need to split square roots of products into products of square roots. The argument scaling also eliminates an underflow problem in the Taylor series that Carlson and Notis report in [CN81].

Testing against symbolic implementations of all of the elliptic integral auxiliary functions in Maple shows that additional care is needed in handling cases of zero arguments, because Carlson's writing is sometimes unclear about whether such arguments are allowed. For example, $R_G(0, 0, z)$ reduces to $\frac{1}{2}\sqrt{z}$, whereas direct application of the formula involving $R_D(x, y, z)$ and $R_F(x, y, z)$ produces $\infty - \infty$, and that evaluates to a NaN in IEEE 754 arithmetic.

20.10 Historical elliptic functions

Here is a summary of the main relations of Legendre's elliptic integrals to Carlson's auxiliary functions, simplified by the introduction of intermediate variables c, d, e, and s:

$$\begin{aligned}
k &= \sin(\alpha), && \textit{so } k \textit{ is in } [-1, +1],\\
c &= \cos(\phi), \qquad s = \sin(\phi), && c \textit{ and } s \textit{ are in } [-1, +1],\\
d &= 1 - (ks)^2, \qquad e = 1 - k^2, \qquad m = k^2, && d, e, \textit{ and } m \textit{ are in } [0, 1],\\
F(\phi\backslash\alpha) &= F(\phi|m) = F(\phi, k), && \textit{alternate notations,}\\
&= sR_F(c^2, d, 1),\\
E(\phi\backslash\alpha) &= E(\phi|m) = E(\phi, k), && \textit{alternate notations,}\\
&= sR_F(c^2, d, 1) - \tfrac{1}{3}k^2 s^3 R_D(c^2, d, 1),\\
\Pi(n; \phi\backslash\alpha) &= \Pi(n, \phi|m) = \Pi(n, \phi, k), && \textit{alternate notations,}\\
&= \textstyle\int_0^\phi (1 + n(\sin(t))^2)^{-1}(1 - (k\sin(t))^2)^{-1/2}\, dt\\
&= sR_F(c^2, d, 1) + \tfrac{1}{3}ns^3 R_J(c^2, d, 1, 1 - ns^2),
\end{aligned}$$

$$\begin{aligned} D(\phi\backslash\alpha) &= D(\phi|m) = D(\phi,k), && \textit{alternate notations,} \\ &= \textstyle\int_0^{\phi}(\sin(t))^2(1-(k\sin(t))^2)^{-1/2}\,dt \\ &= \tfrac{1}{3}s^3 R_D(c^2,d,1), \\ K(k) &= R_F(0,e,1), \\ E(k) &= R_F(0,e,1) - \tfrac{1}{3}k^2 R_D(0,e,1). \end{aligned}$$

Notice that the arguments of the auxiliary functions are mostly the same.

The relation shown for $\Pi(n,\phi,k)$ is taken from [ZC70, equation 2.5]; there are two sign errors in the same relation given in the later paper [Car79, equation 4.3].

For numerical stability, differences of squares in the arguments should be rewritten in product form: $d = (1-ks)(1+ks)$ and $e = (1-k)(1+k)$. For added accuracy, the product factors in d should be computed with fused multiply-add operations.

The mathcw library functions for computing the incomplete Legendre elliptic integrals use the auxiliary functions internally:

$$\begin{aligned} \mathrm{elldi}(\phi,k) &= D(\phi,k), & \mathrm{ellfi}(\phi,k) &= F(\phi,k), \\ \mathrm{ellei}(\phi,k) &= E(\phi,k), & \mathrm{ellpi}(n,\phi,k) &= \Pi(n,\phi,k). \end{aligned}$$

If we take account of the differing argument convention conventions, we can easily reproduce tables in the *Handbook of Mathematical Functions*. For example, part of the table of values of the incomplete elliptic integral of the first kind, $F(\phi\backslash\alpha)$ [AS64, Table 17.5, pages 613–615] looks like the output of this short hoc program:

```
hocd64> for (a = 0; a <= 90; a += 2)          \
hocd64> {
hocd64>     printf("%2d  ", a)
hocd64>     for (p = 0; p <= 20; p += 5)      \
hocd64>         printf("%13.8..5f", ellfi((p/180)*PI, sindeg(a)))
hocd64>     printf("\n")
hocd64>     if ((a % 10) == 8)                \
hocd64>         printf("\n")
hocd64> }
 0    0.00000_000  0.08726_646  0.17453_293  0.26179_939  0.34906_585
 2    0.00000_000  0.08726_660  0.17453_400  0.26180_298  0.34907_428
 4    0.00000_000  0.08726_700  0.17453_721  0.26181_374  0.34909_952
 6    0.00000_000  0.08726_767  0.17454_255  0.26183_163  0.34914_148
 8    0.00000_000  0.08726_860  0.17454_999  0.26185_656  0.34919_998

10    0.00000_000  0.08726_980  0.17455_949  0.26188_842  0.34927_479
12    0.00000_000  0.08727_124  0.17457_102  0.26192_707  0.34936_558
14    0.00000_000  0.08727_294  0.17458_451  0.26197_234  0.34947_200
16    0.00000_000  0.08727_487  0.17459_991  0.26202_402  0.34959_358
18    0.00000_000  0.08727_703  0.17461_714  0.26208_189  0.34972_983
...
80    0.00000_000  0.08737_408  0.17539_854  0.26474_766  0.35614_560
82    0.00000_000  0.08737_528  0.17540_830  0.26478_147  0.35622_880
84    0.00000_000  0.08737_622  0.17541_594  0.26480_795  0.35629_402
86    0.00000_000  0.08737_689  0.17542_142  0.26482_697  0.35634_086
88    0.00000_000  0.08737_730  0.17542_473  0.26483_842  0.35636_908

90    0.00000_000  0.08737_744  0.17542_583  0.26484_225  0.35637_850
```

A second run with the function `ellfi()` replaced by `ellei()` reproduces part of a following table for the elliptic integral of the second kind [AS64, Table 17.6, page 616–618].

A slightly more complex program reproduces a table of values of the incomplete elliptic integral of the third kind [AS64, Table 17.9, pages 625–626]:

```
hocd64> for (n = 0; n <= 1; n += 0.1)         \
hocd64> {
hocd64>     for (a = 0; a <= 90; a += 15)     \
hocd64>     {
hocd64>         printf("%3.1f  %2d", n, a)
hocd64>         for (p = 0; p <= 90; p += 15) \
hocd64>             printf("%9.5f", ellpi(n, (p/180)*PI, sindeg(a)))
hocd64>         printf("\n")
hocd64>     }
hocd64>     printf("\n")
hocd64> }
0.0   0  0.00000  0.26180  0.52360  0.78540  1.04720  1.30900  1.57080
0.0  15  0.00000  0.26200  0.52513  0.79025  1.05774  1.32733  1.59814
0.0  30  0.00000  0.26254  0.52943  0.80437  1.08955  1.38457  1.68575
0.0  45  0.00000  0.26330  0.53562  0.82602  1.14243  1.48788  1.85407
0.0  60  0.00000  0.26406  0.54223  0.85122  1.21260  1.64918  2.15652
0.0  75  0.00000  0.26463  0.54736  0.87270  1.28371  1.87145  2.76806
0.0  90  0.00000  0.26484  0.54931  0.88137  1.31696  2.02759 77.95518

0.1   0  0.00000  0.26239  0.52820  0.80013  1.07949  1.36560  1.65576
0.1  15  0.00000  0.26259  0.52975  0.80514  1.09058  1.38520  1.68536
0.1  30  0.00000  0.26314  0.53412  0.81972  1.12405  1.44650  1.78030
0.1  45  0.00000  0.26390  0.54041  0.84210  1.17980  1.55739  1.96326
0.1  60  0.00000  0.26467  0.54712  0.86817  1.25393  1.73121  2.29355
0.1  75  0.00000  0.26524  0.55234  0.89040  1.32926  1.97204  2.96601
0.1  90  0.00000  0.26545  0.55431  0.89939  1.36454  2.14201 86.50181

...

0.9   0  0.00000  0.26731  0.57106  0.96853  1.58460  2.74440  4.96729
0.9  15  0.00000  0.26752  0.57284  0.97547  1.60516  2.79990  5.09958
0.9  30  0.00000  0.26808  0.57785  0.99569  1.66788  2.97710  5.53551
0.9  45  0.00000  0.26887  0.58508  1.02695  1.77453  3.31211  6.42557
0.9  60  0.00000  0.26966  0.59281  1.06372  1.92081  3.87661  8.20087
0.9  75  0.00000  0.27025  0.59882  1.09535  2.07488  4.74433 12.46409
0.9  90  0.00000  0.27047  0.60110  1.10821  2.14900  5.42126 762.30046

1.0   0  0.00000  0.26795  0.57735  1.00000  1.73205  3.73205      inf
1.0  15  0.00000  0.26816  0.57916  1.00731  1.75565  3.81655      inf
1.0  30  0.00000  0.26872  0.58428  1.02866  1.82781  4.08864      inf
1.0  45  0.00000  0.26951  0.59165  1.06170  1.95114  4.61280      inf
1.0  60  0.00000  0.27031  0.59953  1.10060  2.12160  5.52554      inf
1.0  75  0.00000  0.27090  0.60566  1.13414  2.30276  7.00372      inf
1.0  90  0.00000  0.27112  0.60799  1.14779  2.39053  8.22356      inf
```

Although $\Pi(n; \frac{1}{2}\pi\backslash\frac{1}{2}\pi) = \infty$, the table entries in the last column usually do not show an infinite result, because the computation necessarily uses a truncated value of $\frac{1}{2}\pi$, and growth to the poles is slow.

20.11 Elliptic functions in software

The differing argument conventions for elliptic integrals in symbolic-algebra systems and mathematical texts are a nuisance. We therefore record these function relations for Legendre's incomplete elliptic integrals of the first (F), second (E), and third (Π) kinds to help reduce the confusion:

NBS *Handbook of Mathematical Functions* :

$$F(\phi\backslash\alpha) = \int_0^\phi (1 - (\sin(\alpha)\sin(\theta))^2)^{+1/2}\, d\theta, \qquad \textit{eq. 17.2.7,}$$

$$E(\phi\backslash\alpha) = \int_0^\phi (1 - (\sin(\alpha)\sin(\theta))^2)^{-1/2}\, d\theta, \qquad \textit{eq. 17.2.8,}$$

$$\begin{aligned}\Pi(n;\phi\backslash\alpha) &= \int_0^\phi (1 + n(\sin(\theta))^2)^{-1} \times \\ &\qquad (1 - (\sin(\alpha)\sin(\theta))^2)^{-1/2}\, d\theta, \qquad \textit{eq. 17.2.14 and 17.7.1,}\end{aligned}$$

hoc and mathcw library :

```
func NBSEllF (   phi_, alpha) return (ellfi(   phi_, sin(alpha)))
func NBSEllE (   phi_, alpha) return (ellei(   phi_, sin(alpha)))
func NBSEllPi(n, phi_, alpha) return (ellpi(n, phi_, sin(alpha)))
```

Maple :

```
NBSEllF  := (   phi, alpha) -> EllipticF (sin(phi),    sin(alpha)):
NBSEllE  := (   phi, alpha) -> EllipticE (sin(phi),    sin(alpha)):
NBSEllPi := (n, phi, alpha) -> EllipticPi(sin(phi), n, sin(alpha)):
NBSEllPi := (n, phi, alpha) ->
                               # Alternate form as explicit integral
                               int((1 - n * (sin(theta))**2)**(-1) *
                                   (1 - (sin(alpha) * sin(theta))**2)**(-1/2),
                                   theta = 0 .. phi):
```

Mathematica :

```
NBSEllF [    phi_, alpha_] = EllipticF [   phi, (Sin[alpha])^2]
NBSEllE [    phi_, alpha_] = EllipticE [   phi, (Sin[alpha])^2]
NBSEllPi[n_, phi_, alpha_] = EllipticPi[n, phi, (Sin[alpha])^2]
```

20.12 Applications of elliptic auxiliary functions

Here are some problems whose solutions involve elliptic auxiliary functions [Car77, page 271] [OLBC10, §19.30–19.35]:

- An ellipse aligned with the coordinate axes is a closed curve defined by the equation $(x/a)^2 + (y/b)^2 = 1$. The arc length of an ellipse of semiaxes a, b along x, y measured from the horizontal axis through an angle ϕ (see the sketch on page 630) is given by

  $$\begin{aligned}s(\phi) &= a\int_0^\phi \sqrt{1 - k^2(\sin(\theta))^2}\, d\theta = aE(\phi\backslash\operatorname{asin}(k)) = \operatorname{ellei}(\phi, k), \\ k &= \sqrt{1 - (b/a)^2}.\end{aligned}$$

  The perimeter of the ellipse is

  $$\begin{aligned}P &= 4\int_0^{\pi/2} \sqrt{(a\sin(\theta))^2 + (b\cos(\theta))^2}\, d\theta \\ &= 2\pi R_E(a^2, b^2) \\ &= 2\pi \operatorname{ellre}(a^2, b^2).\end{aligned}$$

  When $a = b = r$, we have $R_E(r^2, r^2) = r$ from the normalization rule, so we recover the schoolbook formula for the circumference of a circle, $C = 2\pi r$.

  The area of the ellipse is just $A = \pi ab$, and when $a = b = r$, that reduces to the familiar $A = \pi r^2$ for a circle.

- An ellipsoid is a volume enclosed by a surface defined by the formula $(x/a)^2 + (y/b)^2 + (z/c)^2 = 1$. The surface area of an ellipsoid with semiaxes a, b, c aligned with the coordinate axes x, y, z is defined with the help of three intermediate variables ℓ, m, n like this:

$$\begin{aligned}
&\ell = \sin(\theta)\cos(\phi), \qquad m = \sin(\theta)\sin(\phi), \qquad n = \cos(\theta),\\
A &= 8\int_0^{\pi/2}\int_0^{\pi/2}\sqrt{(bc\ell)^2 + (acm)^2 + (abn)^2}\,\sin(\theta)\,d\theta\,d\phi\\
&= 4\pi abc R_G(a^{-2}, b^{-2}, c^{-2})\\
&= 4\pi abc\,\mathrm{ellrg}(a^{-2}, b^{-2}, c^{-2}).
\end{aligned}$$

  When $a = b = c = r$, we have $R_G(r^{-2}, r^{-2}, r^{-2}) = 1/r$ from the normalization rule, and $A = 4\pi r^2$, the schoolbook formula for the area of a sphere of radius r.

  Calculating the volume of an ellipsoid does not require elliptic functions: it is just $V = \frac{4}{3}\pi abc$. When the semiaxes are equal, that reduces to $V = \frac{4}{3}\pi r^3$, the standard formula for the volume of a sphere.

- A pendulum oscillating through a small angle ϕ with frequency ω does so with a period given by

$$\begin{aligned}
T &= \tfrac{2\pi}{\omega} R_K((\cos(\phi/2))^2, 1),\\
&= \tfrac{2\pi}{\omega}\,\mathrm{ellrk}((\cos(\phi/2))^2, 1).
\end{aligned}$$

Although we do not reproduce them here, Carlson's book gives formulas for the charge and capacitance of a charged conducting ellipsoid, and for the period of an anharmonic oscillator. Lawden's book also treats those problems, and covers various geometrical problems, vibrations of a spring, and planetary orbits [Law89, Chapters 4 and 5].

20.13 Elementary functions from elliptic auxiliary functions

The auxiliary function $R_C(x, y)$ can also be used to produce several elementary functions, assuming positive x within the normal argument domains:

$$\begin{aligned}
\mathrm{acos}(x) &= \sqrt{1 - x^2}\,R_C(x^2, 1), & \mathrm{acosh}(x) &= \sqrt{x^2 - 1}\,R_C(x^2, 1),\\
\mathrm{acot}(x) &= R_C(x^2, x^2 + 1), & \mathrm{acoth}(x) &= R_C(x^2, x^2 - 1),\\
\mathrm{asin}(x) &= xR_C(1 - x^2, 1), & \mathrm{asinh}(x) &= xR_C(1 + x^2, 1),\\
\mathrm{atan}(x) &= xR_C(1, 1 + x^2), & \mathrm{atanh}(x) &= xR_C(1, 1 - x^2),\\
\log(x) &= (x - 1)R_C(\tfrac{1}{4}(1 + x)^2, x), & \mathrm{log1p}(x) &= xR_C(\tfrac{1}{4}(2 + x)^2, 1 + x).
\end{aligned}$$

As we noted earlier, arguments that are differences of squares must be computed in product form to avoid accuracy loss.

The functions can also be written with rational arguments, and that generality is sometimes useful, because it avoids introducing additional error when the arguments are rational expressions:

$$\begin{aligned}
\mathrm{acos}(x/y) &= \sqrt{y^2 - x^2}\,R_C(x^2, y^2), && \textit{for } 0 \le x \le y,\\
\mathrm{acosh}(x/y) &= \sqrt{x^2 - y^2}\,R_C(x^2, y^2), && \textit{for } y \le x,\\
\mathrm{acot}(x/y) &= yR_C(x^2, x^2 + y^2), && \textit{for all finite } x, y,\\
\mathrm{acoth}(x/y) &= yR_C(x^2, x^2 - y^2), && \textit{for } -y < x < +y,\\
\mathrm{asin}(x/y) &= xR_C(y^2 - x^2, y^2), && \textit{for } -y \le x \le +y,\\
\mathrm{asinh}(x/y) &= xR_C(1 + x^2, y^2), && \textit{for all finite } x, y,\\
\mathrm{atan}(x/y) &= xR_C(y^2, y^2 + x^2), && \textit{for all finite } x, y,\\
\mathrm{atanh}(x/y) &= xR_C(y^2, y^2 - x^2), && \textit{for } -y < x < +y,
\end{aligned}$$

$$\begin{aligned} \log(x/y) &= (x-y)R_C(\tfrac{1}{4}(x+y)^2, xy), && \text{for } 0 < x \text{ and } 0 < y, \\ \operatorname{log1p}(x/y) &= xR_C(\tfrac{1}{4}(x+2y)^2, xy+y^2), && \text{for } -y < x. \end{aligned}$$

In applications where storage space is at a premium, such as in embedded devices, and calculators, code for the $R_C(x,y)$ function could be reused for at least those ten elementary functions. Size restrictions prevent overflow in the products in arguments of some of the inverse functions. The remaining squares halve the usable floating-point range, but that difficulty can be removed by argument reduction, or by using an internal floating-point format with an extended exponent range, as some architectures supply.

20.14 Computing elementary functions via $R_C(x,y)$

To allow testing of the accuracy of software implementations of the relations of the elementary functions to the auxiliary function $R_C(x,y)$, the file `ellfun.c`, and its companions for other precisions, implements the ten, and extends the argument ranges to their normal values. The code looks like this:

```
#define RC(x, y)        ellrc((x), (y))

double
acos(double x)
{
    double y, z;

    y = (ONE - x) * (ONE + x);
    z = SQRT(y) * RC(x * x, ONE);

    return ((x < 0) ? ((PI_HI - z) + PI_LO) : z);
}

double
acosh(double x)
{
    double y;

    y = (x - ONE) * (x + ONE);

    return (SQRT(y) * RC(x * x, ONE));
}

double
acot(double x)
{
    return ((QABS(x) > XLARGE)
            ? (ONE / x)
            : (COPYSIGN(RC(x * x, x * x + ONE), x)));
}

double
acoth(double x)
{
    return ((QABS(x) > XLARGE)
            ? (ONE / x)
            : (COPYSIGN(RC(x * x, (x + ONE) * (x - ONE)), x)));
}

double
asin(double x)
```

```
{
    return (x * RC((ONE - x) * (ONE + x), ONE));
}

double
asinh(double x)
{
    return (x * RC(ONE + x * x, ONE));
}

double
atan(double x)
{
    return (x * RC(ONE, ONE + x * x));
}

double
atanh(double x)
{
    return (x * RC(ONE, (ONE - x) * (ONE + x)));
}

double
log1p(double x)
{
    double result;

    if (x > XBIG)
        result = SQRT(x) * RC(FOURTH * x, ONE);
    else
        result = x * RC(FOURTH * (TWO + x) * (TWO + x), ONE + x);

    return (result);
}

double
log(double x)
{   /* log(x) = log(f * beta**n) = log(f) + n * log(beta) */
    int n;
    double f, result, w, y, z;

    if (QABS(x - ONE) < HALF)
        result = log1p(x - ONE);
    else
    {
        f = FREXP(x, &n);
        y = ONE + f;
        z = RC(FOURTH * y * y, f);
        w = FMA(f, z, -z);
        result = w + (double)n * LOG_BETA;
    }

    return (result);
}
```

Five of the functions are straightforward transcriptions of the mathematical relations. The other five require a bit more care.

The inverse cosine function relation is extended to negative arguments with the identity $\operatorname{acos}(-x) = \pi - \operatorname{acos}(x)$, and the constant π is represented as a two-part sum.

The inverse trigonometric and hyperbolic cotangents need a sign transfer for negative arguments. They avoid premature overflow in squares by switching to the limiting form $1/x$ as soon as the argument magnitude exceeds XLARGE, a compile-time constant with the value $(\frac{1}{2}\epsilon/\beta)^{-1/2}$.

The `log1p(x)` computation prevents premature overflow for arguments that are large enough that $\mathrm{fl}(2+x) = \mathrm{fl}(x)$, because we can then use the argument-scaling relation to reduce the result to

$$xR_C(\tfrac{1}{4}x^2, x) = xx^{-1/2}R_C(\tfrac{1}{4}x, 1) = \sqrt{x}R_C(\tfrac{1}{4}x, 1).$$

The `log(x)` computation requires more code, because testing shows that direct use of the auxiliary function for arguments $x \approx 1$ causes severe accuracy loss. Consequently, the revised code switches to `log1p(x - 1)` in that region. Otherwise, it avoids premature overflow by reducing the argument to the product of a fraction and a power of the base, and then reconstructing the result as $\log(f) + n\log(\beta)$.

Tests of all of those functions, and their `float` companions, against higher-precision functions in the mathcw library show that most errors are well below 1.5 ulps, with peak errors of 2.3 ulps. Although the errors are larger than those of our library functions, considering the simplicity of the entire code, they are remarkably small.

20.15 Jacobian elliptic functions

> THE USE OF MODERN CALCULATING MACHINES HAS GREATLY EXTENDED
> THE SCOPE OF DIRECT NUMERICAL CALCULATION OF ELLIPTIC INTEGRALS . . .
>
> — LOUIS V. KING (1921).

The *Handbook of Mathematical Functions* [AS64, Chapter 16] [OLBC10, Chapter 22] and *An Atlas of Functions* [SO87, Chapter 63] each devote a chapter to a large family of two-argument functions called the Jacobian elliptic functions. There are also treatments of their computation and properties in other books [Mos89, page 396], [Bak92, Chapter 16], [ZJ96, Chapter 18], [Tho97, Chapter 17], [PTVF07, §6.12]. The functions have two significant properties:

- The amplitude function, $\mathrm{am}(u, k)$, recovers the angle ϕ in $u = F(\phi, k)$, and is thus an *inverse function* for Legendre's incomplete elliptic function of the first kind.
- The functions are a generalization of the trigonometric and hyperbolic functions, reducing to those functions when $k = 0$ and $k = 1$:

$$\begin{aligned} \mathrm{sn}(u,0) &= \sin(u), & \mathrm{sn}(u,1) &= \tanh(u), \\ \mathrm{cn}(u,0) &= \cos(u), & \mathrm{cn}(u,1) &= \mathrm{sech}(u) = 1/\cosh(u), \\ \mathrm{sc}(u,0) &= \tan(u), & \mathrm{sc}(u,1) &= \sinh(u). \end{aligned}$$

  The names of the Jacobian elliptic functions are acronyms, so they are usually pronounced as their letter sequences.

Books and journal articles on Jacobian elliptic functions frequently omit the second argument, because it is often constant in a particular mathematical formula. Nevertheless, when you see $\mathrm{sn}\,u$ in print, it is important to remember that it really means $\mathrm{sn}(u, k)$. Similar, the abbreviations $\boldsymbol{E} = E(k)$, $\boldsymbol{K} = K(k)$, and $\theta_i(u) = \theta_i(u, q)$ are commonly used. We avoid that confusing shorthand in the rest of this chapter.

Let us call u the value of the elliptic function of the first kind:

$$\begin{aligned} u &= F(\phi\backslash\alpha) = F(\phi|m) = F(\phi, k), && \textit{alternate notations,} \\ &= \int_0^{\phi} \left(1 - (k\sin(\theta))^2\right)^{-1/2} d\theta, \\ k &= \sin(\alpha), && \textit{elliptic modulus,} \\ m &= k^2, && \textit{elliptic parameter.} \end{aligned}$$

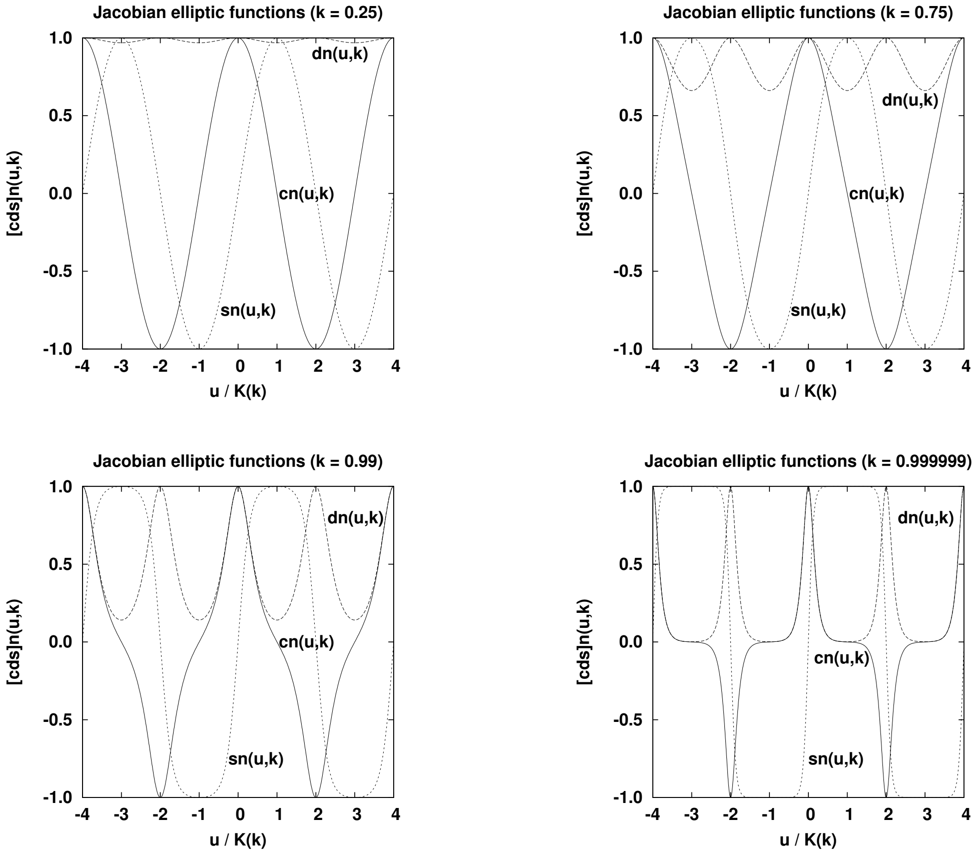


Figure 20.8: Jacobian elliptic functions. The $\mathrm{cn}(u,k)$ and $\mathrm{sn}(u,k)$ functions have period $4K(k)$, and the $\mathrm{dn}(u,k)$ function has period $2K(k)$. Here, $K(k)$ is the complete elliptic integral of the first kind.
In the plots we have $K(0.25) \approx 1.596$, $K(0.75) \approx 1.911$, $K(0.99) \approx 3.357$, and $K(0.999999) \approx 7.947$, and the horizontal axis is scaled to multiples of $K(k)$.
For small k, $\mathrm{dn}(u,k) \approx 1$, but as k increases, $\mathrm{dn}(u,k) \approx \mathrm{cn}(u,k)$ in half of the region.

Clearly, u depends on both ϕ and k, and only two of those three variables can be chosen independently. As in the complete elliptic integrals, we prefer k over the alternate parameterization with α or m to avoid introducing additional argument error from a sine function or a square root. There are then four basic Jacobian elliptic functions:

$$\mathrm{am}(u,k) = \phi, \qquad \mathrm{dn}(u,k) = \left(1 - (k\sin(\phi))^2\right)^{+1/2},$$
$$\mathrm{cn}(u,k) = \cos(\phi), \qquad \mathrm{sn}(u,k) = \sin(\phi).$$

In older texts, $\mathrm{dn}(u,k)$ is often called $\Delta(u,k)$, where Δ is the uppercase Greek letter *delta*. However, the modern name has advantages, as we see shortly.

Once $\mathrm{am}(u,k)$ is known, the other three functions are in principle easily calculated, although finding $\mathrm{dn}(u,k)$ requires extra care to avoid accuracy loss. Those three functions are plotted in **Figure 20.8** for various k values. The functions $\mathrm{cn}(u,k)$ and $\mathrm{sn}(u,k)$ always lie in $[-1,+1]$, but $\mathrm{dn}(u,k)$ is restricted to $[\sqrt{1-k^2}, 1]$. It is always positive, and it can only reach zero when $k = 1$.

Nine other Jacobian elliptic functions are easily generated from three of the basic four, with names based on the functions in the ratios:

$$\mathrm{cd}(u,k) = \mathrm{cn}(u,k)/\,\mathrm{dn}(u,k), \qquad \mathrm{ds}(u,k) = \mathrm{dn}(u,k)/\,\mathrm{sn}(u,k), \qquad \mathrm{nc}(u,k) = 1/\,\mathrm{cn}(u,k),$$
$$\mathrm{cs}(u,k) = \mathrm{cn}(u,k)/\,\mathrm{sn}(u,k), \qquad \mathrm{sc}(u,k) = \mathrm{sn}(u,k)/\,\mathrm{cn}(u,k), \qquad \mathrm{nd}(u,k) = 1/\,\mathrm{dn}(u,k),$$

$$\operatorname{dc}(u,k) = \operatorname{dn}(u,k)/\operatorname{cn}(u,k), \qquad \operatorname{sd}(u,k) = \operatorname{sn}(u,k)/\operatorname{dn}(u,k), \qquad \operatorname{ns}(u,k) = 1/\operatorname{sn}(u,k).$$

We provide all of them in the mathcw library, but with a common prefix, `ELJ`, to indicate their family membership: `ELJAM(u,k)`, `ELJCD(u,k)`, ..., `ELJSN(u,k)`. The function family `ELJAM(u,k)` reduces computed values outside the range $[-\pi, +\pi]$ with the exact argument reduction provided by the `R2P()` family so that the computed amplitude is guaranteed to lie in $[-\pi, +\pi]$.

Some books use $\operatorname{tn}(u,k)$ for $\operatorname{sc}(u,k)$ because of the resemblance to the trigonometric-function relation, $\tan(z) = \sin(z)/\cos(z)$.

As **Figure 20.8** on the facing page shows, the basic three functions are periodic, smooth, well behaved, and nicely bounded, but the other nine have poles at the zeros of $\operatorname{cn}(u,k)$, $\operatorname{dn}(u,k)$, and $\operatorname{sn}(u,k)$.

Maple supplies the Jacobian elliptic functions with names `JacobiAM(z,k)`, `JacobiCD(z,k)` ..., `JacobiSN(z,k)`. Mathematica provides `JacobiAmplitude[u,m]`, `JacobiCD[u,m]`, ..., `JacobiSN[u,m]`, where $m = k^2$. REDUCE uses the same names and arguments as Mathematica. Maxima has `jacobi_am(u,m)`, `jacobi_cd(u,m)`, ..., `jacobi_sn (u,m)`. In all of the algebra systems, the arguments may be complex numbers. MATLAB provides only a single function for real arguments, `ellipj(u,m)`; it returns a three-element vector with sn, cn, and dn values. If the elliptic package is installed, the R statistics programming language provides all of the Jacobian elliptic functions under their conventional mathematical names, but with k replaced by $m = k^2$: $\operatorname{cn}(u,m)$, $\operatorname{dn}(u,m)$, and so on. However, the package does not supply the amplitude function.

We show how to compute the Jacobian elliptic functions in **Section 20.15.2** on page 661, but it is worth noting here that the relations for $\operatorname{cn}(u,k)$ and $\operatorname{dn}(u,k)$ conceal a problem that is evident in **Figure 20.9** on the following page. As $|k| \to 1$, there are arguments $|u| > 4$ for which the amplitude is flat, with values near $\pm\frac{1}{2}\pi$. Because $\operatorname{sn}(u,k) = \sin(\operatorname{am}(u,k))$, the sine is then close to ± 1, and $\operatorname{dn}(u,k)$ requires square roots of differences of almost-equal quantities, resulting in catastrophic subtraction loss. Similarly, $\operatorname{cn}(u,k)$ depends on the value of the cosine near a root, and argument sensitivity is highest there. Regrettably, most published algorithms for computation of the Jacobian elliptic functions ignore that problem.

20.15.1 Properties of Jacobian elliptic functions

The Jacobian elliptic functions are so well-studied that their known relations fill a large handbook [BF71]. In this section, we therefore present only a few of the most important of them.

The functions satisfy these equations:

$$\begin{aligned}
(\operatorname{dn}(u,k))^2 + (k\operatorname{sn}(u,k))^2 &= 1, \qquad (\operatorname{cn}(u,k))^2 + (\operatorname{sn}(u,k))^2 = 1,\\
(\operatorname{dn}(u,k))^2 - (k\operatorname{cn}(u,k))^2 &= 1 - k^2,\\
(\operatorname{cn}(u,k))^2 + (1-k^2)(\operatorname{sn}(u,k))^2 &= (\operatorname{dn}(u,k))^2.
\end{aligned}$$

Those relations are useful for software testing, because they are not used in our algorithms for computing the functions.

The Jacobian elliptic functions obey these argument-symmetry relations:

$$\begin{aligned}
\operatorname{am}(-u,k) &= -\operatorname{am}(u,k), & \operatorname{am}(u,-k) &= \operatorname{am}(u,k),\\
\operatorname{cn}(-u,k) &= +\operatorname{cn}(u,k), & \operatorname{cn}(u,-k) &= \operatorname{cn}(u,k),\\
\operatorname{dn}(-u,k) &= +\operatorname{dn}(u,k), & \operatorname{dn}(u,-k) &= \operatorname{dn}(u,k),\\
\operatorname{sn}(-u,k) &= -\operatorname{sn}(u,k), & \operatorname{sn}(u,-k) &= \operatorname{sn}(u,k).
\end{aligned}$$

There are also argument addition and halving relations:

$$\begin{aligned}
\operatorname{am}(a \pm b,k) &= (\operatorname{atan}(\operatorname{sc}(a,k)\operatorname{dn}(b,k)) \pm \operatorname{atan}(\operatorname{sc}(b,k)\operatorname{dn}(a,k))) \bmod \pi,\\
\operatorname{cn}(a \pm b,k) &= \frac{\operatorname{cn}(a,k)\operatorname{cn}(b,k) \mp \operatorname{sn}(a,k)\operatorname{sn}(b,k)\operatorname{dn}(a,k)\operatorname{dn}(b,k)}{1 - (k\operatorname{sn}(a,k)\operatorname{sn}(b,k))^2},\\
\operatorname{dn}(a \pm b,k) &= \frac{\operatorname{dn}(a,k)\operatorname{dn}(b,k) \mp k^2\operatorname{sn}(a,k)\operatorname{sn}(b,k)\operatorname{cn}(a,k)\operatorname{cn}(b,k)}{1 - (k\operatorname{sn}(a,k)\operatorname{sn}(b,k))^2},\\
\operatorname{sn}(a \pm b,k) &= \frac{\operatorname{sn}(a,k)\operatorname{cn}(b,k)\operatorname{dn}(b,k) \pm \operatorname{sn}(b,k)\operatorname{cn}(a,k)\operatorname{dn}(a,k)}{1 - (k\operatorname{sn}(a,k)\operatorname{sn}(b,k))^2},
\end{aligned}$$

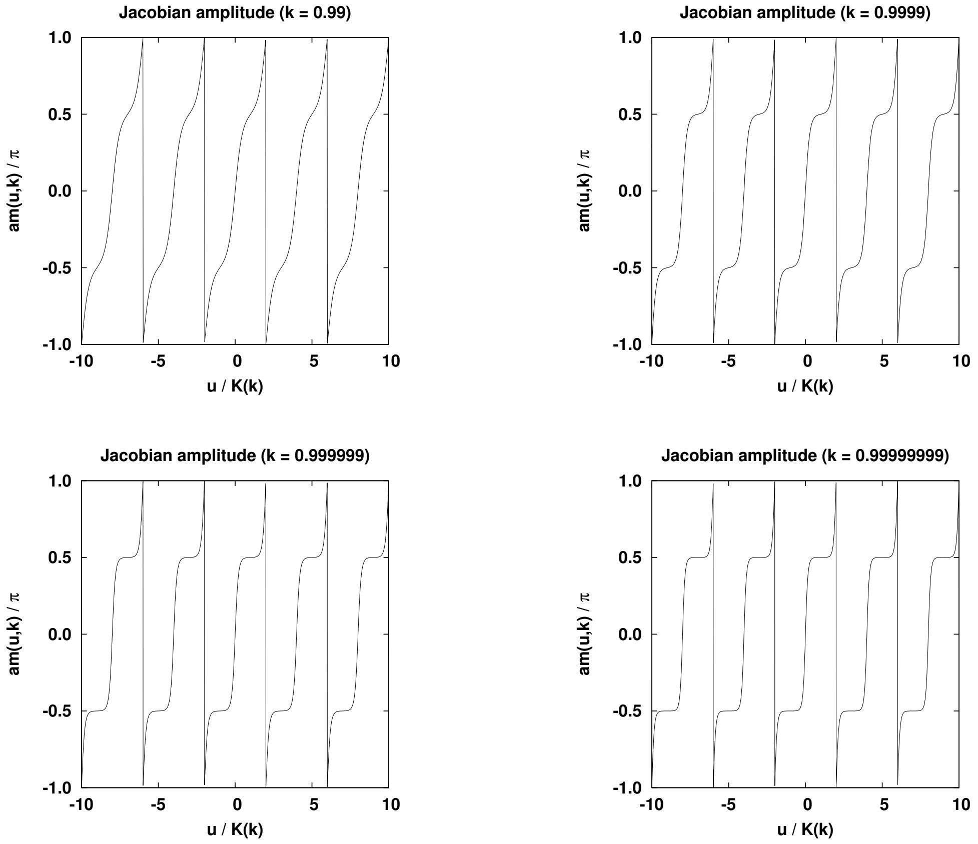


Figure 20.9: Jacobian elliptic function amplitudes for $|k| \to 1$. The flat regions where $\operatorname{am}(u,k) \approx \pm\frac{1}{2}\pi$ are computationally troublesome.

$$\begin{aligned}
\operatorname{cn}(\tfrac{1}{2}u,k) &= \sqrt{\frac{\operatorname{cn}(u,k)+\operatorname{dn}(u,k)}{1+\operatorname{dn}(u,k)}},\\
\operatorname{dn}(\tfrac{1}{2}u,k) &= \sqrt{\frac{\operatorname{cn}(u,k)+\operatorname{dn}(u,k)}{1+\operatorname{cn}(u,k)}},\\
\operatorname{sn}(\tfrac{1}{2}u,k) &= \sqrt{\frac{1-\operatorname{cn}(u,k)}{1+\operatorname{dn}(u,k)}}.
\end{aligned}$$

Those relations are useful for checking the consistency of software implementations of the Jacobian elliptic functions, although the presence of subtractions requires either higher precision, or avoiding tests where there would be subtraction loss.

For small u, symbolic-algebra systems readily find these Taylor-series expansions:

$$\begin{aligned}
\operatorname{am}(u,k) &= u - \left(\frac{k^2}{3!}\right)u^3 + \left(\frac{4k^2+k^4}{5!}\right)u^5 - \left(\frac{16k^2+44k^4+k^6}{7!}\right)u^7 + \cdots,\\
\operatorname{cn}(u,k) &= 1 - \left(\frac{1}{2!}\right)u^2 + \left(\frac{1+4k^2}{4!}\right)u^4 - \left(\frac{1+44k^2+16k^4}{6!}\right)u^6 + \cdots,\\
\operatorname{dn}(u,k) &= 1 - \left(\frac{k^2}{2!}\right)u^2 + \left(\frac{4k^2+k^4}{4!}\right)u^4 - \left(\frac{16k^2+44k^4+k^6}{6!}\right)u^6 + \cdots,
\end{aligned}$$

$$\operatorname{sn}(u,k) = u - \left(\frac{1+k^2}{3!}\right)u^3 + \left(\frac{1+14\,k^2+k^4}{5!}\right)u^5 - \left(\frac{1+135\,k^2+135\,k^4+k^6}{7!}\right)u^7 + \cdots.$$

For small k and small u, we have these Taylor-series expansions:

$$\begin{aligned}
\operatorname{am}(u,k) &= u + \left(\frac{\sin(u)\cos(u) - u}{4}\right)k^2 + \mathcal{O}(k^4),\\
\operatorname{cn}(u,k) &= \cos(u) + \left(\frac{u\sin(u) - (\sin(u))^2\cos(u)}{4}\right)k^2 + \mathcal{O}(k^4),\\
\operatorname{dn}(u,k) &= 1 - \left(\frac{(\sin(u))^2}{2}\right)k^2 + \mathcal{O}(k^4),\\
\operatorname{sn}(u,k) &= \sin(u) + \left(\frac{(\cos(u))^2\sin(u) - u\cos(u)}{4}\right)k^2 + \mathcal{O}(k^4).
\end{aligned}$$

For $|k| \approx 1$ and small u, these Taylor-series expansions hold:

$$\begin{aligned}
\operatorname{am}(u,k) &= \left(-\tfrac{1}{2}\pi + 2\operatorname{atan}(\exp(u))\right) + \left(\frac{\sinh(u) - u\operatorname{sech}(u)}{2}\right)(1-|k|) + \mathcal{O}((1-|k|)^2),\\
\operatorname{cn}(u,k) &= \operatorname{sech}(u) + \left(\frac{u\operatorname{sech}(u)\tanh(u) - \sinh(u)\tanh(u)}{2}\right)(1-|k|) + \mathcal{O}((1-|k|)^2),\\
\operatorname{dn}(u,k) &= \operatorname{sech}(u) + \left(\frac{u\operatorname{sech}(u)\tanh(u) + \sinh(u)\tanh(u)}{2}\right)(1-|k|) + \mathcal{O}((1-|k|)^2),\\
\operatorname{sn}(u,k) &= \tanh(u) + \left(\frac{\tanh(u) - u(\operatorname{sech}(u))^2}{2}\right)(1-|k|) + \mathcal{O}((1-|k|)^2).
\end{aligned}$$

Higher-order coefficients in the series with powers of k or $1-k$ have many trigonometric or hyperbolic functions with arguments that are multiples of powers of u. The coefficients are expensive to compute, and their dependence on u means that they cannot be precomputed and stored as compile-time constants. Nevertheless, the series are essential for accurate function evaluation in the narrow regions where they apply.

Although the Jacobian elliptic functions are periodic, the periods are $2K(k)$ or $4K(k)$. We cannot do the exact argument reduction that we described in **Chapter 9** on page 243, because we would then need to evaluate the complete elliptic function of the first kind accurately to thousands of digits.

The restrictions on k can be relaxed by introducing *reciprocal modulus transformations* when $|k| > 1$:

$$\begin{aligned}
\operatorname{cn}(u,k) &= \operatorname{dn}(ku,1/k), & \operatorname{dn}(u,k) &= \operatorname{cn}(ku,1/k), & \operatorname{sn}(u,k) &= \operatorname{sn}(ku,1/k)/k,\\
\operatorname{nc}(u,k) &= \operatorname{nd}(ku,1/k), & \operatorname{nd}(u,k) &= \operatorname{nc}(ku,1/k), & \operatorname{ns}(u,k) &= k\operatorname{ns}(ku,1/k),\\
\operatorname{cd}(u,k) &= \operatorname{dc}(ku,1/k), & \operatorname{cs}(u,k) &= k\operatorname{ds}(ku,1/k), & &\\
\operatorname{dc}(u,k) &= \operatorname{cd}(ku,1/k), & \operatorname{ds}(u,k) &= k\operatorname{cs}(ku,1/k), & &\\
\operatorname{sc}(u,k) &= \operatorname{sd}(ku,1/k)/k, & \operatorname{sd}(u,k) &= \operatorname{sc}(ku,1/k)/k. & &
\end{aligned}$$

The right-hand side expressions are then in the standard ranges where the elliptical integrals are real when the arguments are real.

We can use one of those relations to find a reciprocal modulus transformation for $\operatorname{am}(u,k)$, the amplitude function, like this:

$$\operatorname{am}(u,k) = \operatorname{asin}(\operatorname{sn}(u,k)) = \operatorname{asin}(\operatorname{sn}(ku,1/k)/k).$$

20.15.2 Computing Jacobian elliptic functions

If k lies outside the range $[-1,+1]$, apply the reciprocal modulus transformations given at the end of the preceding section.

Outside the Taylor-series regions, efficient computation of the basic four functions is best done with the vector AGM algorithm (see **Section 20.1** on page 619, [AS64, §16.4], and [OLBC10, §22.20]), with starting values

$$a_0 = 1, \qquad b_0 = \sqrt{1-k^2}, \qquad c_0 = k, \qquad \text{provided } k \neq 0.$$

Factor the quantity under the square root into a product for accurate computation. After the AGM converges, iterate downward to generate an array of angles, ϕ_j:

$$\begin{aligned}\phi_n &= 2^n a_n u,\\ \sin(2\phi_{j-1} - \phi_j) &= (c_j/a_j)\sin(\phi_j), && \text{for } j = n, n-1, \ldots, 1,\\ \phi_{j-1} &= \tfrac{1}{2}(\phi_j + \operatorname{asin}((c_j/a_j)\sin(\phi_j)).\end{aligned}$$

The amplitude is then $\operatorname{am}(u,k) = \phi = \phi_0$, from which $\operatorname{cn}(u,k)$ and $\operatorname{sn}(u,k)$ are found from their definitions. The remaining function may be found either from $\operatorname{dn}(u,k) = \cos(\phi_0)/\cos(\phi_1 - \phi_0)$, or from the square root given earlier. The cosine form is subject to leading digit loss from the subtraction in the denominator, so it should be avoided.

The VAGM() family in the mathcw library produces the arrays of a_j, b_j, and c_j, and because we need the ϕ_j values in multiple functions, we provide a support function to compute those values along with the AGM coefficients. Its code in the file eljagx.h looks like this:

```
fp_t
ELJAG(fp_t u, fp_t k, int nabc, fp_t aj[/* nabc */], fp_t bj[/* nabc */], fp_t cj[/* nabc */],
      fp_t phij[/* nabc */], int *pneed)
{   /* Jacobian elliptic integral vector AGM iteration */
    fp_t result;
    int need;

    need = 0;

    if (ISNAN(u))
        result = u;
    else if (ISNAN(k))
        result = k;
    else if ( (k <= -ONE) || (ONE <= k) )
        result = SET_EDOM(QNAN(""));    /* avoid |k| = 1 (AGM b = 0) */
    else
    {   /* NBS Handbook of Mathematical Functions, section 16.4 */
        fp_t a, b, c;
        int j, n;

        a = ONE;
        b = SQRT((ONE - k) * (ONE + k));
        c = k;
        (void)VAGM(a, b, c, nabc, aj, bj, cj, &need);
        n = MIN(nabc - 1, need - 1);
        phij[n] = EXP2((fp_t)n) * aj[n] * u;

        for (j = n; j > 0; --j)
        {
            fp_t r;

            r = (cj[j] / aj[j]) * SIN(phij[j]);
            r = MAX(-ONE, MIN(r, ONE));
            phij[j - 1] = HALF * (phij[j] + ASIN(r));
        }

        result = phij[0];
    }

    if (pneed != (int *)NULL)
        *pneed = need;

    return (result);
}
```

The code ensures that the argument of the inverse sine function lies in $[-1,+1]$ to prevent generation of a NaN. For caller convenience, the angle ϕ_0 is returned as a function value, as well as in the zeroth element of the array `phij[]`.

The application of the vector AGM algorithm to the computation of the Jacobian elliptic integral functions dates back to work with the AGM by Lagrange (1784), Legendre, Gauss, and Jacobi. The modern form was discovered by Canadian physicist Louis V. King at McGill University in 1913, but first published in an article in 1921 [Kin21], and a small monograph in 1924 (reprinted in 2007) [Kin24, Kin07].

Although six of the Jacobian elliptic integral functions are defined as ratios of two others, it is more efficient to compute the ϕ_j values once, and then use them to determine both the numerator and denominator. For example, here is the code from file `eljscx.h` for computing $\mathrm{sc}(u,k)$:

```
fp_t
ELJSC(fp_t u, fp_t k)
{   /* Jacobian elliptic integral function sc(u,k) = sn(u,k) / cn(u,k) */
    fp_t result;
    static fp_t last_k = FP(0.);
    static fp_t last_result = FP(0.);
    static fp_t last_u = FP(0.);

    if (ISNAN(u))
        result = u;
    else if (ISNAN(k))
        result = k;
    else if ( (u == last_u) && (k == last_k) )
        result = last_result;
    else if ( (k < -ONE) || (ONE < k) )
        result = ELJSN(u, k) / ELJCN(u, k);
    else if (k == ZERO)
        result = TAN(u);
    else if (QABS(k) == ONE)
        result = SINH(u);
    else if ( (QABS(u) < FP(0.025)) || (QABS(k) < FP(0.0002)) || ((ONE - QABS(k)) < FP(0.0002)) )
        result = ELJSN(u, k) / ELJCN(u, k);
    else
    {   /* NBS Handbook of Mathematical Functions, sections 16.3 and 16.4 */
        fp_t aj[NMAX], bj[NMAX], cj[NMAX], phij[NMAX];
        int need;

        (void)ELJAG(u, k, NMAX, aj, bj, cj, phij, &need);

        if (need > NMAX)
            result = SET_EDOM(QNAN(""));
        else
            result = TAN(phij[0]);
    }

    last_k = k;
    last_result = result;
    last_u = u;

    return (result);
}
```

If $|k| > 1$, we apply the reciprocal modulus transformation. If $k = 0$ or $k = 1$, we use special-case reductions to standard elementary functions. If u is small, or k is near 0 or 1, the special-case handling of $\mathrm{cn}(u,k)$ and $\mathrm{sn}(u,k)$ applies, so we simply compute the ratio of the two. Otherwise, we use `ELJAG()` to find the array of ϕ_j values, from which the result is found from $\sin(\phi_0)/\cos(\phi_0) = \tan(\phi_0)$.

In **Section 20.15** on page 659, we briefly discussed the troublesome case of $|k| \to 1$, where the algorithms for $\mathrm{cn}(u,k)$ and $\mathrm{dn}(u,k)$ lose accuracy. There do not appear to be any standard mathematical formulas for computing those two functions in a way that avoids that loss.

To solve the problem, we need a higher-precision value of the amplitude. We can then obtain its cosine and sine with sufficient additional precision to hide the loss for most values of u. Decompose the amplitude into a sum of exact high and accurate low parts, recall that $\cos(\frac{1}{2}\pi) = 0$ and $\sin(\frac{1}{2}\pi) = 1$, and use the trigonometric angle-sum rules to reduce the sine and cosine of the amplitude like this:

$$\begin{aligned}
\phi &= \texttt{HP\_ELJAM(u, k)}, && \textit{work in next higher precision,}\\
&= \tfrac{1}{2}\pi + \delta, && \textit{where } \delta \textit{ is small,}\\
\sin(\phi) &= \sin(\tfrac{1}{2}\pi)\cos(\delta) + \cos(\tfrac{1}{2}\pi)\sin(\delta)\\
&= \cos(\delta),\\
c &= 1 - \sin(\phi)\\
&= 1 - \cos(\delta)\\
&= -\sum_{n=1}^{\infty}(-1)^n\delta^{2n}/(2n)!, && \textit{fast Taylor-series expansion,}\\
\cos(\phi) &= \cos(\tfrac{1}{2}\pi)\cos(\delta) - \sin(\tfrac{1}{2}\pi)\sin(\delta)\\
&= -\sin(\delta).
\end{aligned}$$

From the reduced formulas, we easily obtain accurate computational formulas for the two functions:

$$\begin{aligned}
\operatorname{cn}(u,k) &= -\sin(\delta),\\
\operatorname{dn}(u,k) &= \sqrt{1-(k\operatorname{sn}(u,k))^2} = \sqrt{1-(k\sin(\phi))^2} = \sqrt{1-(k(1-c))^2} = \sqrt{((1-k)+kc)(1+(k-kc))}.
\end{aligned}$$

The code in `eljcnx.h` and `eljdnx.h` uses those alternative algorithms only when $|k| > 0.99$ and $|u| > 4$, so the extra precision is often not required: indeed, we show in **Section 20.17** on page 668 that such large k values are unlikely to be encountered in practice. When the higher precision is not at least twice that of working precision, such as the common case of an excursion from the IEEE 754 64-bit format to the 80-bit format, the errors in the function values are reduced somewhat, but may still be huge, with more than two-thirds of the low-order bits incorrect. If that is of concern in a particular application, then the only practical solution may be to reimplement the amplitude function in multiple-precision arithmetic in order to obtain a correctly rounded value of δ.

Applications that require $\operatorname{cn}(u,k)$, $\operatorname{dn}(u,k)$, and $\operatorname{sn}(u,k)$ sometimes need only small values of u, and our functions then provide results that have errors of at most a few ulps. However, u is mathematically unrestricted, and we can see that the starting value $\phi_n = 2^n a_n u$ in the AGM algorithm may need a precision much higher than is available if its sine is to be determined accurately. Thus, our code unavoidably loses accuracy for large u, and in nonbinary bases, also for large n. We observed earlier that argument reduction to bring u into a region where the computation is accurate is not practical in programming languages with conventional floating-point data types, because the reduction requires values of $K(k)$ correct to at least as many digits as the exponent range of the floating-point format.

Error plots for the Jacobian elliptic functions are surfaces in the (u,k) plane, but their roughness makes them hard to see. The plots shown in **Figure 20.10** and **Figure 20.11** on page 666 take a simpler approach: for a fixed u, the errors from random values of k taken from a logarithmic distribution in [`FP_T_MIN`, 1], with random sign, are plotted along a vertical line. Similarly, for a fixed k, errors are shown for randomly chosen u values, but u is restricted to the range $[-\pi, +\pi]$.

20.16 Inverses of Jacobian elliptic functions

The Jacobian elliptic functions are single-valued and periodic, so with suitable restrictions on arguments and a fixed modulus k, we can define unique inverses for each of them. Like the trigonometric and hyperbolic functions, we prefix each function name with the letter *a* to indicate the inverse function. With a little algebra, or resort to a handbook [BF71, pages 29–32], we can find these formulas for the original 13 Jacobian elliptic functions and their inverses:

$$k_c = \sqrt{1-k^2}, \qquad \textit{complementary modulus,}$$

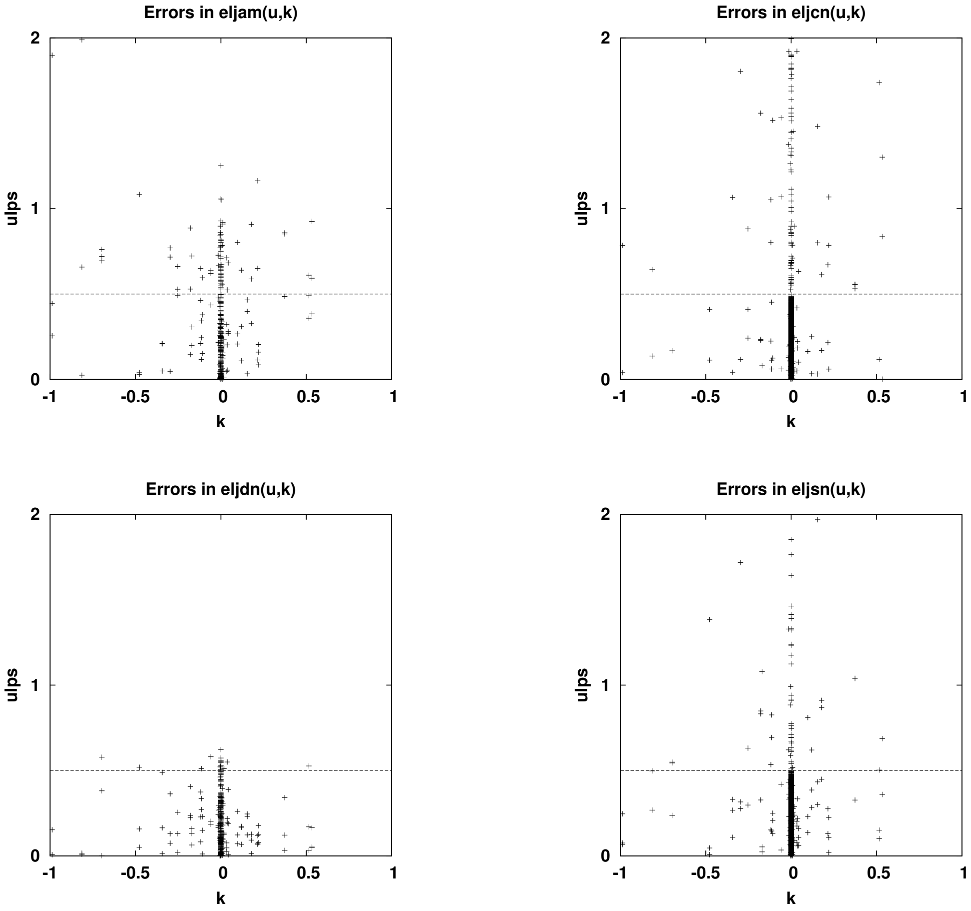


Figure 20.10: Errors in Jacobian elliptic functions along k.

$$\begin{array}{lll}
v = \operatorname{am}(u,k), & u = \operatorname{aam}(v,k) = F(v,k), & v \textit{ in } [0,+\pi/2]; \\
v = \operatorname{cd}(u,k), & u = \operatorname{acd}(v,k) = F(\operatorname{asin}(\sqrt{(1-v^2)/(1-(kv)^2)}),k), & \\
 & \qquad v \textit{ in } [0,+1]; & \\
v = \operatorname{cn}(u,k), & u = \operatorname{acn}(v,k) = F(\operatorname{acos}(v),k), & v \textit{ in } [0,+1]; \\
v = \operatorname{cs}(u,k), & u = \operatorname{acs}(v,k) = F(\operatorname{asin}(1/\sqrt{1+v^2}),k), & v \textit{ in } [0,+\infty); \\
v = \operatorname{dc}(u,k), & u = \operatorname{adc}(v,k) = F(\operatorname{asin}(\sqrt{(v^2-1)/(v^2-k^2)}),k), & v \textit{ in } [+1,\infty); \\
v = \operatorname{dn}(u,k), & u = \operatorname{adn}(v,k) = F(\operatorname{asin}(\sqrt{1-v^2}/k),k), & v \textit{ in } [k_c,+1]; \\
v = \operatorname{ds}(u,k), & u = \operatorname{ads}(v,k) = F(\operatorname{asin}(1/\sqrt{k^2+v^2}),k), & v \textit{ in } [k_c,+\infty); \\
v = \operatorname{nc}(u,k), & u = \operatorname{anc}(v,k) = F(\operatorname{acos}(1/v),k), & v \textit{ in } [+1,+\infty); \\
 & \quad = F(\operatorname{asin}(\sqrt{1-(1/v)^2}),k), & \\
v = \operatorname{nd}(u,k), & u = \operatorname{and}(v,k) = F(\operatorname{asin}(\sqrt{v^2-1}/(kv)),k), & v \textit{ in } [+1,1/k_c); \\
v = \operatorname{ns}(u,k), & u = \operatorname{ans}(v,k) = F(\operatorname{asin}(1/v),k), & v \textit{ in } [+1,+\infty);
\end{array}$$

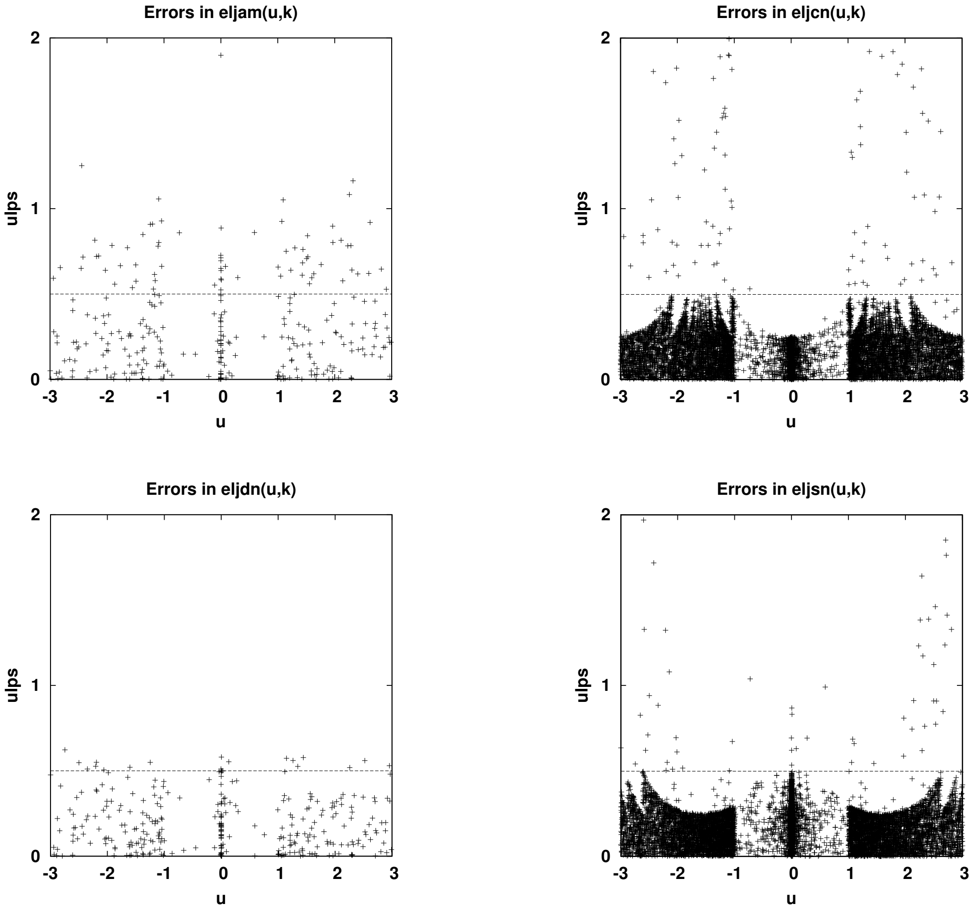


Figure 20.11: Errors in Jacobian elliptic functions along u.

$$v = \mathrm{sc}(u,k), \qquad u = \mathrm{asc}(v,k) = F(\mathrm{asin}(1/\sqrt{1+(1/v)^2}),k), \qquad v \textit{ in } (0,+\infty);$$

$$v = \mathrm{sd}(u,k), \qquad u = \mathrm{asd}(v,k) = F(\mathrm{asin}(v/\sqrt{1+(kv)^2}),k), \qquad v \textit{ in } [0,1/k_c];$$

$$v = \mathrm{sn}(u,k), \qquad u = \mathrm{asn}(v,k) = F(\mathrm{asin}(v),k), \qquad v \textit{ in } [0,+1].$$

Each of the inverse functions corresponds to a particular elliptic integral (see [BF71, pages 29–32], [AS64, equations 17.4.41–17.4.52], or [OLBC10, §22.15]), but here we show only the integrals for the inverses of the basic four Jacobian elliptic functions, and their relations to Legendre's elliptic integral of the first kind:

$$\begin{aligned}
\mathrm{aam}(\phi,k) &= \int_0^\phi \frac{1}{\sqrt{1-(k\sin(\theta))^2}}\,d\theta, && \textit{for } \phi \textit{ in } [0,+\pi/2],\\
&= F(\phi,k),\\
\mathrm{acn}(v,k) &= \int_v^1 \frac{1}{\sqrt{(1-t^2)(k_c^2+k^2t^2)}}\,dt, && \textit{for } v \textit{ in } [0,+1],\\
&= F(\mathrm{asin}(\sqrt{1-v^2}),k),\\
&= F(\mathrm{acos}(v),k),\\
\mathrm{adn}(v,k) &= \int_v^1 \frac{1}{\sqrt{(1-t^2)(t^2-k_c^2)}}\,dt, && \textit{for } v \textit{ in } [0,+1],
\end{aligned}$$

$$\begin{aligned} &= F(\operatorname{asin}(\sqrt{(1-v^2)/k^2}), k), \\ \operatorname{asn}(v,k) &= \int_0^v \frac{1}{\sqrt{(1-t^2)(1-k^2t^2)}}\,dt, && \text{for } v \text{ in } [0,+1], \\ &= F(\operatorname{asin}(v), k). \end{aligned}$$

The 13 formulas for inverse functions in the *Handbook of Elliptic Integrals* are all expressions of the form $F(\operatorname{asin}(\cdot), k)$, although we sometimes use equivalent forms with $\operatorname{acos}(\cdot)$ when that reduces the error in the computed arguments. When the argument is small, it is preferable to use the form with the inverse sine, because that can be computed more accurately than the one with the inverse cosine.

Care is required in computing the arguments of acos() and asin(). Differences of squares must be factored into products for improved accuracy, and premature overflow inside square roots must be avoided by suitable rewriting. For example, when $v > 1$, replace $v/\sqrt{1+v^2}$ by $1/\sqrt{(1/v)^2+1}$. In addition, reciprocals of square roots should be evaluated with the `RSQRT()` family to eliminate the error in division.

Maple has `InverseJacobiAM(v,k)` through `InverseJacobiSN(v,k)`. Except for the inverse amplitude function, Mathematica uses the same names, but the second argument is $m = k^2$. For the inverse amplitude, use `EllipticF [v,k]`. Maxima has `inverse_jacobi_cd(v,m)` through `inverse_jacobi_sn(v,m)`, where again $m = k^2$. The mathcw library functions are `eljaam(v,k)` through `eljasn(v,k)`.

The integrands in the inverse functions always involve even powers of t, so the integrands are symmetric about the origin. However, only *two* of the inverse functions have integration limits that start at zero; thus, they are the only two that exhibit symmetry about the origin:

$$\operatorname{aam}(-v,k) = -\operatorname{aam}(v,k), \qquad \operatorname{asn}(-v,k) = -\operatorname{asn}(v,k).$$

If we extend v beyond its stated limits, then frequently, the argument of the square root becomes negative, making the integral values complex, rather than real. The algebra systems permit arbitrary complex values of v and k, but our code requires real arguments, and enforces limits on them; out-of-range arguments produce NaN results.

The help system in Maple provides limited information about the handling of extended arguments, but does show these connections:

$$\begin{aligned} \operatorname{acd}(z,k) &= K(k) - \operatorname{aam}(\operatorname{asin}(z),k), & \operatorname{anc}(z,k) &= \operatorname{aam}(\operatorname{acos}(1/z),k), \\ \operatorname{acn}(z,k) &= \operatorname{aam}(\operatorname{acos}(z),k), & \operatorname{and}(z,k) &= \operatorname{aam}(\operatorname{acos}(1/z),1/k)/k, \\ \operatorname{acs}(z,k) &= -\operatorname{aam}(\operatorname{asinh}(1/z)i,k_c)i, & \operatorname{ans}(z,k) &= \operatorname{aam}(\operatorname{asin}(1/z),k), \\ \operatorname{adc}(z,k) &= K(k) - \operatorname{aam}(\operatorname{asin}(1/z),k), & \operatorname{asc}(z,k) &= -\operatorname{aam}(\operatorname{asinh}(z)i,k_c)i, \\ \operatorname{adn}(z,k) &= \operatorname{aam}(\operatorname{acos}(z),1/k)/k, & \operatorname{asd}(z,k) &= K(k) + \operatorname{aam}(\operatorname{acos}(k_cz),k), \\ \operatorname{ads}(z,k) &= K(k) + \operatorname{aam}(\operatorname{acos}(k_c/z),k), & \operatorname{asn}(z,k) &= \operatorname{aam}(\operatorname{asin}(z),k). \end{aligned}$$

We can easily remove the range restriction for the inverse amplitude function, $\operatorname{aam}(\phi,k) = F(\phi,k)$, if we recall that $K(k) = F(\frac{1}{2}\pi,k)$. We can then split the integral into two parts, the first with an upper limit that is a multiple of $\frac{1}{2}\pi$, and the second an integral from that limit up to ϕ:

$$\begin{aligned} \phi &= n(\tfrac{1}{2}\pi) + r, && \text{for } n = 0,1,2,\dots \text{ and } r \text{ in } [0, \tfrac{1}{2}\pi), \\ \operatorname{aam}(\phi,k) &= \begin{cases} nK(k) + F(r,k), & \text{for } n \text{ even}, \\ (n+1)K(k) - F(\frac{1}{2}\pi - r, k), & \text{for } n \text{ odd}. \end{cases} \end{aligned}$$

If n is odd, we advance to the next multiple of $\frac{1}{2}\pi$, giving the term $(n+1)K(k)$, and then subtract the integral of the advance, $F(\frac{1}{2}\pi - r, k)$. Otherwise, we have $nK(k)$ from the first integral, plus the remaining integral that, by symmetry, is equal to $F(r,k)$. Thanks to the `RPH()` family in the mathcw library (see **Section 9.2** on page 250), we have *exact* argument reduction in binary arithmetic, but we can use it only when n is exactly representable as an `int` data type, as well as an `fp_t` type. For the trigonometric functions, we only require the remainder r and the low-order bits of n, but here, we need both r and n to be exact.

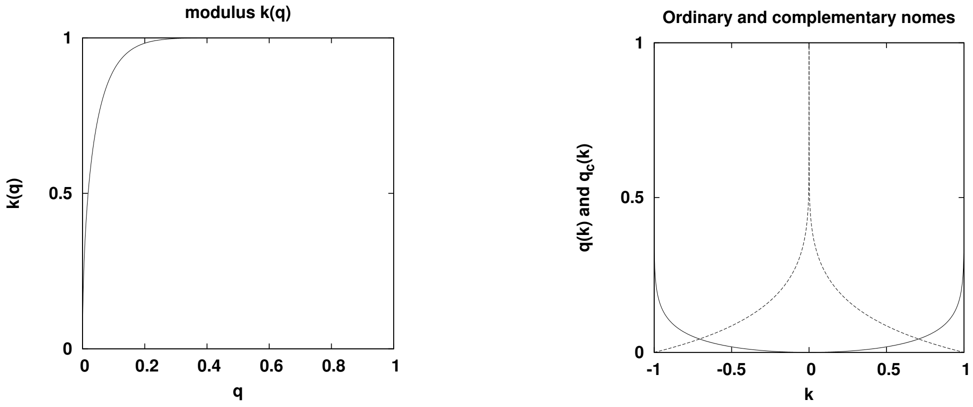


Figure 20.12: Elliptic modulus, ordinary nomes (solid), and complementary nome (dashed). The peaks are steep, but are not poles, because they stop at 1.

20.17 The modulus and the nome

In the next section, we introduce functions defined by power series in a variable q that is called the *nome*, a name derived from *binomial*. The magnitude of q lies in $[0, 1]$, and the nome is defined by the relation

$$\begin{aligned} q &= \exp(-\pi K'(k)/K(k)), && \text{for } k \text{ in } [-1, +1], \\ &\approx \texttt{EXP(-PI * ELLKC(k) / ELLK(k))}. \end{aligned}$$

Thus, q is determined entirely by the elliptic function modulus k, which lies in $[-1, +1]$. When k is real, q is also real, and the converse holds as well.

There is also a complementary nome defined with an inverted ratio of the two complete elliptic functions:

$$q_c = \exp(-\pi K(k)/K'(k)), \qquad \text{for } k \text{ in } [-1, +1].$$

Unlike the ordinary and complementary complete elliptic functions, the two nomes are *not* mirror images. Instead, they have these relations that allow one of them to be obtained from the other with minimal loss of accuracy:

$$\begin{aligned} \log(1/q)\log(1/q_c) &= \pi^2, && \text{for } q \text{ and } q_c \text{ in } (0, 1], \\ \log(q)\log(q_c) &= \pi^2, \\ q &= \exp(\pi^2/\log(q_c)), & q - 1 &= \operatorname{expm1}(\pi^2/\log(q_c)), \\ q_c &= \exp(\pi^2/\log(q)), & q_c - 1 &= \operatorname{expm1}(\pi^2/\log(q)). \end{aligned}$$

When either nome is near 1, the equations containing expm1() allow accurate determination of the difference from 1, effectively representing the nomes in higher precision.

The modulus and the nomes are common in the mathematics of elliptic functions. The limiting relations

$$\begin{aligned} k(0) &= 0, & k(1) &= 1, \\ q(0) &= 0, & q(\pm 1) &= 1, & q(-k) &= q(k), \\ q_c(0) &= 1, & q_c(\pm 1) &= 0, & q_c(-k) &= q_c(k), \end{aligned}$$

allow us handle the cases $k = \pm 1$, $q = 1$, and $q_c = 0$, even when those endpoints are excluded by the mathematics.

The values of the modulus and nomes in real arithmetic are graphed in Figure 20.12. Because there is a one-to-one relationship between k and q, an inverse function exists that converts q to k. We show how to compute that inverse later in this section, and in the next section.

For improved accuracy and computational speed with small arguments, we need these Taylor-series expansions:

$$k(q) = 4\sqrt{q}(1 - 4\,q + 14\,q^2 - 40\,q^3 + 101\,q^4 - 236\,q^5 + 518\,q^6 - 1080\,q^7 + 2162\,q^8 + \cdots),$$

$$q(k) = 2^{-20}(65536\,k^2 + 32768\,k^4 + 21504\,k^6 + 15872\,k^8 + 12514\,k^{10} + 10293\,k^{12} + \cdots),$$
$$q_c(1-d) = 2^{-19}(65536\,d + 32768\,d^2 + 20480\,d^3 + 14336\,d^4 + 10784\,d^5 + 8528\,d^6 + 6994\,d^7 + 5895\,d^8 + \cdots).$$

For tiny k, we have $q(k) \approx k^2/16$, so q underflows for $k < 4\sqrt{\texttt{FP\_T\_MIN}}$, eliminating roughly half of all possible k values. In the other direction, if $q > 0$, then $k(q) > 0$ as well.

Maple supplies `EllipticNome(k)` and `EllipticModulus(q)` for the nome function and its inverse. Mathematica has `EllipticNomeQ[m]` and `InverseEllipticNomeQ[q]`, where the latter returns $m = k^2$. The mathcw library implements them as the families `ELQ(k)` and `ELK(q)`.

In order to handle the nome endpoints accurately, the library also provides `ELKM1(q)` for $k(q) - 1$, `ELQ1P(km1)` for $q(1 + \texttt{km1})$, `ELQC(k)` for $q_c(k)$, and `ELQC1P(km1)` for $q_c(1 + \texttt{km1})$. Like our complete elliptic integral functions, the functions remember their last argument and result for fast handling of repeated calls with the same argument.

The programming of `ELQ(k)` is simple, but developing a suitable algorithm for the inverse, `ELK(q)`, is more difficult. In the next section, we show a general formula for that inverse, but here we develop an alternative to deal with a problem that is evident in the graphs in **Figure 20.12**: when the nome $q > \frac{1}{2}$, the modulus $k \approx 1$. Thus, it is desirable to find a way to compute the difference $k - 1$ directly and accurately, because $k = 1 + (k - 1)$ can then be computed accurately to machine precision.

It is instructive to look at numeric values of the nome for $k \approx 1$ by choosing k values one little machine epsilon below 1 in extended IEEE 754 binary and decimal arithmetic:

```
% maple
> Digits := 80:

> for t in [24, 53, 64, 113, 237] do
>     q := EllipticNome(1 - 2**(-t)):
>     printf("%3d  %8.6f  %9.3e\n", t, q, 1 - q)
> end do:
 24  0.590159  4.098e-01
 53  0.775486  2.245e-01
 64  0.808544  1.915e-01
113  0.884486  1.155e-01
237  0.942397  5.760e-02

> for t in [7, 16, 34, 70] do
>     q := EllipticNome(1 - 10**(-t)):
>     printf("%3d  %8.6f  %9.3e\n", t, q, 1 - q)
> end do:
  7  0.581375  4.186e-01
 16  0.776016  2.240e-01
 34  0.884435  1.156e-01
 70  0.941338  5.866e-02
```

Thus, in the commonly used 64-bit formats, we are almost certain to have $q < 0.777$.

We also examine the dependence of the modulus k on the nome q:

```
> Digits := 1000:

> for q from 0 to 0.7 by 0.1 do
>     printf("%.1f  %.9f\n", q, EllipticModulus(q))
> end do:
0.0  0.000000000      0.4  0.999832060
0.1  0.895769668      0.5  0.999994761
0.2  0.982777795      0.6  0.999999967
0.3  0.997799764      0.7  1.000000000

> for t in [24, 53, 64, 113, 237] do
>     printf("%3d  %12.6e  %12.6e\n", t, EllipticModulus(2**(-t)), 1 - EllipticModulus(0.99))
> end do:
 24  9.765623e-04  2.620380e-426
 53  4.214685e-08  2.620380e-426
 64  9.313226e-10  2.620380e-426
113  3.925231e-17  2.620380e-426
237  8.511488e-36  2.620380e-426

> for t in [7, 16, 34, 70] do
>     printf("%3d  %12.6e  %12.6e\n", t, EllipticModulus(10**(-t)), 1 - EllipticModulus(0.99))
> end do:
  7  1.264911e-03  2.620380e-426
 16  4.000000e-08  2.620380e-426
 34  4.000000e-17  2.620380e-426
 70  4.000000e-35  2.620380e-426
```

We see that $q < 0.5$ for $k < 0.999\,995$, and getting a q value as large as 0.99 requires that k be represented with a precision greater than 425 decimal digits.

We can also investigate the behavior near the underflow limit of the 128-bit IEEE 754 decimal format, which has the widest exponent range of current floating-point systems:

```
> Digits := 10000:

> for q from 0.9990 to 0.9995 by 0.0001 do
>     printf("%.4f  %12.6e\n", q, EllipticModulus(q) - 1)
> end do:
0.9990  -5.393551e-4284
0.9991  -2.982956e-4760
0.9992  -1.422724e-5355
0.9993  -5.492125e-6121
0.9994  -1.543755e-7141
0.9995  -2.611957e-8570
```

The smallest subnormal is reached already just before $q = 0.999\,307$.

A Taylor-series expansion of `EllipticModulus(1 - d)` in Maple produces a complicated result containing the functions of the next section. However, a similar expansion in Mathematica produces a result with interesting, and recognizable, structure:

```
% math
In[1]:= Series[InverseEllipticNomeQ[1 - d] - 1, {d, 0, 3}]

               2                                2         2      3
             Pi /(-d - d /2 - d /3)       (2 Pi )/(-d - d /2 - d /3)
Out[2]= -16 E                       + 128 E

                   2         2      3
              (3 Pi )/(-d - d /2 - d /3)
        -704 E
```

Further experiments with increasing expansion order show that the polynomials in the denominator are simply sums of d^k/k, which is just the power series of $\log(1-d) = \log(q)$. The general result is therefore

$$
\begin{aligned}
a &= \exp(\pi^2/\log(q)) = \exp(\pi^2/\operatorname{log1p}(-d)), \\
m(q) - 1 &= \sum_{n=1}^{\infty} c_n a^n, \qquad \textit{elliptic parameter,} \\
c_n &= -16, +128, -704, +3072, -11488, +38400, -117632, +335872, \\
&\quad -904784, +2320128, -5702208, +13504512, -30952544, \ldots \\
(k(q))^2 &= m(q), \qquad \textit{relation between elliptic modulus and parameter,} \\
(k(q))^2 - 1 &= m(q) - 1,
\end{aligned}
$$

$$\begin{aligned} k(q) - 1 &= (m(q) - 1)/(k(q) + 1) \\ &= (m(q) - 1)/(\sqrt{1 + (m(q) - 1)} + 1). \end{aligned}$$

A check with Sloane's *On-Line Encyclopedia of Integer Sequences*[2] finds the coefficients in sequence *A115977*, where they are identified as those of the expansion of the modulus in powers of the nome.

Because $q < 1$, we have $\log(q) < 0$ and $0 \leq a < 1$. With $q = 0.99$, we have $d = 0.01$ and $a \approx 3.275 \times 10^{-427}$, so the series converges extraordinarily quickly for small d. Even for $q = d = \frac{1}{2}$, we have $a \approx 6.549 \times 10^{-7}$, and a sum of the thirteen terms corresponding to the listed values of c_n converges to 75 decimal digits, more than enough for the extended 256-bit floating-point formats supported by the mathcw library.

Here is our code for computing $k(q) - 1$:

```
fp_t
ELKM1(fp_t q)
{   /* elliptic modulus k, less 1, from nome q in [0,1] */
    fp_t result;
    static fp_t last_q = FP(0.0);
    static fp_t last_result = FP(-1.0); /* elkm1(0) = -1 */

    if (ISNAN(q))
        result = q;
    else if ( (q < ZERO) || (ONE < q) )
        result = SET_EDOM(QNAN(""));
    else if (q == last_q)
        result = last_result;
    else if (q == ZERO)
        result = -ONE;
    else if (q == ONE)
        result = ZERO;
    else if (q < NOME_K_HALF) /* k(q) - 1 has no leading digit loss */
        result = (fp_t)(HP_ELK((hp_t)q) - HP(1.));
    else
    {
        hp_t a, err_inv_log_q, inv_log_q, log_q, sum;
        int k, n;

        log_q = HP_LOG((hp_t)q);
        inv_log_q = HP(1.) / log_q;
        err_inv_log_q = HP_ERRDIV(inv_log_q, HP(1.), log_q);
        a = HP_EXP(HP_FMA(PI_SQUARED_HI, inv_log_q,
                          PI_SQUARED_LO * inv_log_q +
                          PI_SQUARED_HI * err_inv_log_q));

        n = 99;

        if (q < FP(0.25))
        {
            if      (FP_T_DIG > 34) n = 88;
            else if (FP_T_DIG > 16) n = 48;
            else if (FP_T_DIG >  7) n = 26;
            else n = 14;
        }
        else if (q < FP(0.5))
        {
            if      (FP_T_DIG > 34) n = 28;
            else if (FP_T_DIG > 16) n = 15;
```

[2] See http://oeis.org/.

```
            else if (FP_T_DIG >  7) n = 8;
            else n = 5;
        }
        else if (q < FP(0.75))
        {
            if      (FP_T_DIG > 34) n = 14;
            else if (FP_T_DIG > 16) n = 8;
            else if (FP_T_DIG >  7) n = 4;
            else n = 3;
        }
        else
        {
            if      (FP_T_DIG > 34) n = 6;
            else if (FP_T_DIG > 16) n = 4;
            else if (FP_T_DIG >  7) n = 3;
            else n = 2;
        }

        sum = c[n + 1];

        for (k = n; k >= 1; --k)
            sum = QFMA(sum, a, c[k]);

        sum *= a;

        result = (fp_t)(sum / (HP_SQRT(HP(1.) + sum) + HP(1.)));
        last_q = q;
        last_result = result;
    }

    return (result);
}
```

The series converges poorly for $k < \frac{1}{2}$ (or, $q < 0.018$), so in that region, we switch to the function family `ELK(q)` described in the next section. However, there is subtraction loss in forming $k(q) - 1$ for q in $[0.018, 1]$, so we hide that by using the next higher precision of that function. When the series is usable, and $q \to 1$, the argument of the exponential is large and negative, introducing the problem of error magnification (see **Section 4.1** on page 61). We need about five extra decimal digits to hide that error, so here too, we work in the next higher precision. The number of terms needed in the series depends on both q and the floating-point precision, so we save time by setting the loop limit according to those values.

When a higher-precision format is not available, we lose about five or six decimal digits as $q \to 1$, as these examples in the 128-bit decimal format show against values computed in 7000-digit precision with the `EllipticModulus(q)` function in Maple:

```
hocd128> elkm1(0.4999); -5.260509197344328434613414148710826487e-06
-5.260_509_197_344_328_434_613_414_148_710_825e-06
-5.260_509_197_344_328_434_613_414_148_710_826e-06

hocd128> elkm1(0.9993); -5.492124671494925145968662458171583 56e-6121
-5.492_124_671_494_925_145_968_662_458_152_305e-6121
-5.492_124_671_494_925_145_968_662_458_171_584e-6121
```

Losses are similar in the 80-bit format, and are about three decimal digits in the 64-bit format.

Several of the functions discussed in the remaining sections of this chapter require both k and q in their computation, but only one of those values is provided. Because k approaches 1 so quickly as q grows, we soon have $\mathrm{fl}(k) = 1$ to machine precision, even though the exact difference $k - 1$ is nonzero. For that reason, the `mathcw` library includes a family of complete elliptic integral functions of the first kind with the nome as the argument, `ELLKN(q)`. When the computed k in that family is less than $\frac{1}{2}$, the result is just `ELLK(k)`. Otherwise, the result is computed from

`ELLKC(SQRT(-km1 * (TWO + km1)))`, where `km1` = `ELKM1(q)`. The general programming rule is that if q is given, then the *smaller* of k and $k-1$ should first be determined accurately, and the other then derived from it.

20.18 Jacobian theta functions

Four functions known as Jacobian theta functions are related to the Jacobian elliptic functions, and are conventionally defined as sums of infinite series of powers of the nome q, where convergence requires that $q < 1$ [AS64, equations 16.27.1–16.27.4] [OLBC10, §20.2]:

$$\begin{aligned}
\theta_1(z,q) &= 2q^{+1/4}\sum_{n=0}^{\infty}(-1)^n q^{n(n+1)}\sin((2n+1)z) \\
&= 2\sum_{n=0}^{\infty}(-1)^n q^{((n+1/2)^2)}\sin((2n+1)z) \\
&= \theta_2(u-\tfrac{1}{2}\pi,q), \\
\theta_2(z,q) &= 2q^{+1/4}\sum_{n=0}^{\infty}q^{n(n+1)}\cos((2n+1)z) \\
&= 2\sum_{n=0}^{\infty}q^{((n+1/2)^2)}\cos((2n+1)z) \\
&= \theta_1(z+\tfrac{1}{2}\pi,q), \\
\theta_3(z,q) &= 1+2\sum_{n=1}^{\infty}q^{(n^2)}\cos(2nz) \\
&= \theta_4(z-\tfrac{1}{2}\pi,q), \\
\theta_4(z,q) &= 1+2\sum_{n=1}^{\infty}(-1)^n q^{(n^2)}\cos(2nz) \\
&= \theta_3(z+\tfrac{1}{2}\pi,q).
\end{aligned}$$

Notice that the lower limits of the first two differ from those of the last two. The terms in the sums are similar, so it may be worthwhile for software to compute $\theta_1(z,q)$ and $\theta_2(z,q)$ together, and $\theta_3(z,q)$ and $\theta_4(z,q)$ together. The functions are plotted in **Figure 20.13** on the next page for real arguments u and four different real values of q.

The subscript on θ is called the *kind*: thus, $\theta_4(z,q)$ is the theta function of the *fourth* kind. Some books label that function with a zero subscript: $\theta_0(z,q) = \theta_4(z,q)$.

The Jacobian theta functions turn up in several areas of pure and applied mathematics. They are solutions of an equation for heat conduction and another for wave motion, they appear in *Laplace transforms*, and they are related to another family of elliptic functions described in **Section 20.22** on page 682.

The Jacobian theta functions have these periodicity and symmetry relations:

$$\begin{aligned}
\theta_1(z+\pi,q) &= -\theta_1(z,q), & \theta_1(-z,q) &= -\theta_1(z,q), \\
\theta_2(z+\pi,q) &= -\theta_2(z,q), & \theta_2(-z,q) &= +\theta_2(z,q), \\
\theta_3(z+\pi,q) &= +\theta_3(z,q), & \theta_3(-z,q) &= +\theta_3(z,q), \\
\theta_4(z+\pi,q) &= +\theta_4(z,q), & \theta_4(-z,q) &= +\theta_4(z,q).
\end{aligned}$$

Theta functions of two special arguments are related to the complete elliptic integral of the first kind:

$$\begin{aligned}
\theta_1(\tfrac{1}{2}\pi,q) = \theta_2(0,q) &= \sqrt{\frac{2kK(k)}{\pi}}, \\
\theta_3(0,q) = \theta_4(\tfrac{1}{2}\pi,q) &= \sqrt{\frac{2K(k)}{\pi}}, \\
k' &= \sqrt{1-k^2}, && \textit{complementary elliptic modulus,}
\end{aligned}$$

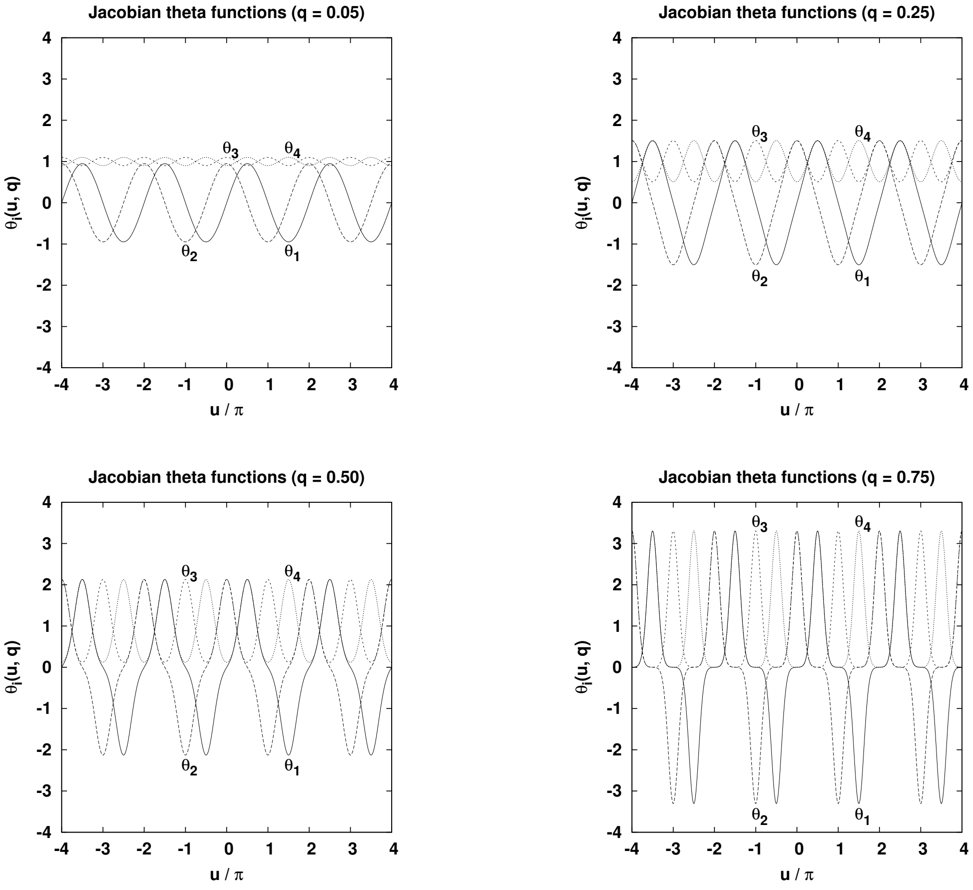


Figure 20.13: Jacobian theta functions. The horizontal axes are scaled to units of π to emphasize the periods of the theta functions. Notice that $\theta_1(u,q)$ and $\theta_2(u,q)$ have both positive and negative values, while the other two functions are never negative.

$$\theta_3(\tfrac{1}{2}\pi, q) = \theta_4(0, q) = \sqrt{\frac{2k'K(k)}{\pi}}.$$

Maple provides the Jacobian theta functions as `JacobiTheta1(z,q)`, ..., `JacobiTheta4(z,q)`. Mathematica supplies a single function `EllipticTheta[n,z,q]`, where $n = 1, 2, 3$, or 4. REDUCE also has a single function, but with different arguments: `EllipticTheta(n, u, m)`. The `mathcw` library implements the theta functions as `ELJT1(u, q)` through `ELJT4(u, q)`, and also provides `ELJTA(result, u, q)` for computing an array of values of all four theta functions.

The argument z in the symbolic-algebra languages is an arbitrary complex number defined over the entire complex plane. The nome argument q is also a complex number.

In **Section 20.17** on page 668, we defined the nome with an exponential function depending on k. Two of the theta functions provide an inverse relation that recovers k from q:

$$k = \frac{(\theta_2(0,q))^2}{(\theta_3(0,q))^2}, \qquad \text{provided } q < 1.$$

The `ELK()` family implements that simple formula, with `ELJT2()` and `ELJT3()` doing the real work.

Many mathematical formulas involving theta functions require both the modulus k and the nome q. Because of the interdependence of k and q, software argument lists require just one of them, but finding the other is a comparatively expensive operation.

The theta functions are related to the Jacobian elliptic functions like this:

$$\begin{aligned}
k' &= \sqrt{1-k^2}, && \textit{complementary elliptic modulus,}\\
q &= \exp(-\pi K'(k)/K(k)), && \textit{elliptic nome,}\\
v &= \frac{\pi u}{2K(k)}, && \textit{scaled argument,}\\
\operatorname{cn}(u,k) &= \frac{\sqrt{k'}}{\sqrt{k}}\,\frac{\theta_2(v,q)}{\theta_4(v,q)}, \\
\operatorname{dn}(u,k) &= \sqrt{k'}\,\frac{\theta_3(v,q)}{\theta_4(v,q)}, \\
\operatorname{sn}(u,k) &= \frac{1}{\sqrt{k}}\,\frac{\theta_1(v,q)}{\theta_4(v,q)}.
\end{aligned}$$

The four theta-function infinite series with sums of sines and cosines are called *Fourier series*, and they are convergent provided $q < 1$. The rapidly increasing powers of q ensure quick convergence of the sums, unless $q \approx 1$. Numerical tests of those functions in symbolic-algebra systems, and our straightforward implementations in C and hoc, show that up to 10,000 terms may be required in the 128-bit formats when $q = 1 - 10^{-6}$. For values of q even closer to 1, the symbolic-algebra systems may return undefined values.

Unfortunately, there do not appear to be any simple argument-reduction relations that can move the case of $q \approx 1$ to smaller q values, where the four series converge more quickly. Computer software therefore is advised to put a reasonable limit on the number of terms that are summed, and to exit the loop as soon as the sum has converged in floating-point arithmetic.

In practical applications, the problem values of q rarely appear. Recall from the relation of k to the eccentricity of an ellipse (see **Section 20.3** on page 630) that the case $k = 1$ corresponds to an infinitely thin ellipse, so we are likely to need only $|k| < 1$. We showed in **Section 20.17** on page 668 that $q < \frac{1}{2}$ corresponds to $k < 0.999\,995$, so convergence of the four theta series is rapid. With the limit $q < 0.777$ in the 64-bit formats, at most a dozen terms are needed, and even in the 256-bit formats, 52 terms suffice.

The representation of $\operatorname{cn}(u,k)$ and $\operatorname{dn}(u,k)$ as ratios of theta functions does *not* provide a solution to the accuracy-loss problem that we discussed in **Section 20.15** on page 659 and **Section 20.15.2** on page 664: if $|k| \to 1$, then $q \to 1$ as well, and the theta-series convergence is too slow to be practical.

When $z = \frac{1}{2}\pi$, we have $(-1)^n \sin((2n+1)z) = +1$, so the sum in the definition of $\theta_1(z,q)$ grows rapidly if $q \approx 1$. When $z = \pi$, we have $\cos((2n+1)z) = -1$, and the sum for $\theta_2(z,q)$ also grows rapidly when $q \approx 1$. Similarly, for $q \approx 1$, $\theta_3(z,q)$ grows when $z = \pi$, and $\theta_4(z,q)$ increases when $z = \pi/2$.

20.19 Logarithmic derivatives of the Jacobian theta functions

Logarithmic derivatives of the theta functions [AS64, §16.29] [OLBC10, §20.5] are sometimes needed. They are defined by these infinite sums:

$$\begin{aligned}
\frac{\theta_1'(u,q)}{\theta_1(u,q)} &= +\cot(u) + 4\sum_{n=1}^{\infty} \frac{q^{2n}}{1-q^{2n}}\sin(2nu), && \textit{for nome } q \textit{ in } [0,1),\\
\frac{\theta_2'(u,q)}{\theta_2(u,q)} &= -\tan(u) + 4\sum_{n=1}^{\infty} (-1)^n \frac{q^{2n}}{1-q^{2n}}\sin(2nu),\\
\frac{\theta_3'(u,q)}{\theta_3(u,q)} &= 4\sum_{n=1}^{\infty} (-1)^n \frac{q^{n}}{1-q^{2n}}\sin(2nu),\\
\frac{\theta_4'(u,q)}{\theta_4(u,q)} &= 4\sum_{n=1}^{\infty} \frac{q^{n}}{1-q^{2n}}\sin(2nu).
\end{aligned}$$

Here, the lower limits on the sums are identical, but notice that the numerator powers differ.

Multiply the left-hand sides by their denominators to produce formulas for the derivatives of the theta functions with respect to u.

The terms in the logarithmic-derivative sums are similar, apart from their signs, so software implementations can compute the four sums more efficiently simultaneously than separately. Also, when either $q = 0$ or $u = 0$, the sums are zero. Software should check for that case, and take care to preserve the correct sign of zero in the results.

When q is real, the denominator $1 - q^{2n}$ is subject to severe accuracy loss if $q^{2n} > \frac{1}{2}$, or equivalently, $q > 0.707$. The polynomial $\mathcal{Q}_{2n}(q) = 1 - q^{2n}$ clearly has at least two real roots, $q = \pm 1$, so it can always be written in the form $\mathcal{Q}_{2n}(q) = (1-q)(1+q)\mathcal{P}_{2n-2}(q)$. The factor $1 - q$ is then exact for $q \geq \frac{1}{2}$, and the product is accurate. We can therefore compute the denominators stably and accurately with this recurrence:

$$\begin{aligned} \mathcal{Q}_2(q) &= 1 - q^2, && \textit{for } q \textit{ in } [0, 1), \\ &= (1-q)(1+q), && \textit{computational form,} \\ \mathcal{Q}_{2n+2}(q) &= q^2 \mathcal{Q}_{2n}(q) + \mathcal{Q}_2(q), && \textit{sum of positive terms.} \end{aligned}$$

The mathcw library function families `ELJTD1(u,q)` through `ELJTD4(u,q)` implement separate computation of the logarithmic derivatives of the four Jacobian theta functions.

The family `ELJTDA(result,u,q)` computes all four logarithmic derivatives simultaneously, returning them in the four-element `result[]` array. Here is its code from the file `eljdax.h`:

```
void
ELJTDA(fp_t result[/* 4 */], fp_t u, fp_t q)
{   /* logarithmic derivative of Jacobian theta function of orders
       1 to 4 for arbitrary u, and q in [0,1): see NBS Handbook of
       Mathematical Functions, equations 16.29.1 to 16.29.4 */
    static const int NMAX = 10000;      /* enough for worst case (q ~= 1) in 34D arithmetic */
    static fp_t last_q = FP_T_MAX;
    static fp_t last_result[4] = { FP_T_MAX, FP_T_MAX, FP_T_MAX, FP_T_MAX };
    static fp_t last_u = FP_T_MAX;

    if (ISNAN(u))
        result[0] = result[1] = result[2] = result[3] = u;
    else if (ISNAN(q))
        result[0] = result[1] = result[2] = result[3] = q;
    else if ( (q < ZERO) || (ONE <= q) )
        result[0] = result[1] = result[2] = result[3] = SET_EDOM(QNAN(""));
    else if ( (u == last_u) && (q == last_q) )
    {
        result[0] = last_result[0];
        result[1] = last_result[1];
        result[2] = last_result[2];
        result[3] = last_result[3];
    }
    else if (q == ZERO)
    {
        result[0] = COTAN(u);
        result[1] = -TAN(u);
        result[2] = result[3] = ZERO;
    }
    else if (u == ZERO)
    {
        result[0] = COPYSIGN(INFTY(), u);
        result[1] = COPYSIGN(ZERO, -COPYSIGN(ONE, u));
        result[2] = result[3] = ZERO;
    }
    else
    {
        fp_t Q_2, Q_2n, q_sq, q_to_n, sum1, sum2, sum3, sum4;
        int n, nc;
        nc = 0;
        q_sq = q * q;
        q_to_n = ONE;
        Q_2 = (ONE - q) * (ONE + q);    /* Q_2(q) = 1 - q**2 */
        Q_2n = Q_2;                     /* Q_2n(q) = 1 - q**(2*n) */
        sum1 = sum2 = sum3 = sum4 = ZERO;

        for (n = 1; n <= NMAX; ++n)
        {
            fp_t f, new_sum1, new_sum2, new_sum3, new_sum4, term;

            q_to_n *= q;

            if (q_to_n == ZERO)
                break;                  /* all remaining powers are zero */

            f = (q_to_n / Q_2n) * SIN((fp_t)(n + n) * (fp_t)u);
            term = q_to_n * f;
            new_sum1 = sum1 + term;
            new_sum2 = sum2 + (IS_EVEN(n) ? term : -term);
            term = f;
            new_sum3 = sum3 + (IS_EVEN(n) ? term : -term);
            new_sum4 = sum4 + term;

            if (new_sum3 == sum3)
            {
                if (++nc > 1)
                    break;
            }
            else
                nc = 0;

            sum1 = new_sum1;
            sum2 = new_sum2;
            sum3 = new_sum3;
            sum4 = new_sum4;
            Q_2n = FMA(q_sq, Q_2n, Q_2);
        }

        result[0] = QFMA(FOUR, sum1, COTAN(u));
        result[1] = QFMA(FOUR, sum2,  -TAN(u));
        result[2] = FOUR * sum3;
        result[3] = FOUR * sum4;

        last_q = q;
        last_result[0] = result[0];
        last_result[1] = result[1];
        last_result[2] = result[2];
        last_result[3] = result[3];
        last_u = u;
    }
}
```

The code in ELJTDA() is reasonably straightforward, except for one important point. Because we compute four sums simultaneously, the loop exit condition must be based on the sum with the slowest convergence, that for the logarithmic derivative of the third kind. However, it is possible that a term is abnormally small because of the sine factor, so we count in nc the number of times that the term is negligible, and exit the loop only when that happens in two successive terms.

20.20 Neville theta functions

The functions of this section were introduced by the English mathematician E. H. Neville in his 1944 book *Jacobian Elliptic Functions* [Nev44, Nev51]. The Neville functions are related to the Jacobian theta functions like this [AS64, §16.36] [OLBC10, §22.2]:

$$\begin{aligned}
u &= F(\phi\backslash\alpha) = F(\phi|m) = F(\phi,k), & v &= \frac{\pi u}{2K(k)},\\
q &= \exp(-\pi K'(k)/K(k)),\\
\theta_c(u,q) &= \frac{\theta_2(v,q)}{\theta_2(0,q)}, & \theta_n(u,q) &= \frac{\theta_4(v,q)}{\theta_4(0,q)},\\
\theta_d(u,q) &= \frac{\theta_3(v,q)}{\theta_3(0,q)}, & \theta_s(u,q) &= \frac{2K(k)}{\pi}\,\frac{\theta_1(v,q)}{\theta_1'(0,q)},\\
\theta_1'(0,q) &= \theta_2(0,q)\theta_3(0,q)\theta_4(0,q).
\end{aligned}$$

The last relation expresses the first derivative of $\theta_1(u,q)$ with respect to u as a product of three other theta functions, all evaluated at $u = 0$. That derivative supplies the denominator in $\theta_s(u,q)$.

The subscripts are chosen from the letters c, d, n, and s, because that convention allows *all twelve* Jacobian elliptic functions to be recovered from the four Neville theta functions with this simple relation:

$$\mathrm{ab}(u,k) = \frac{\theta_a(u,q)}{\theta_b(u,q)}, \qquad \textit{for nome } q\textit{, and } a, b \textit{ any distinct pair of } c, d, n, s.$$

Some software implementations of the Jacobian elliptic functions use that relation to generate them from the Neville theta functions.

Like the Jacobian theta functions, the Neville theta functions have periods $2K(k)$ or $4K(k)$. However, unlike the Jacobian theta functions, two of them grow quickly with increasing q. The Neville functions are shown in **Figure 20.14** on the facing page.

Neville's functions can also be defined as sums of infinite series [AS64, §16.38]:

$$\begin{aligned}
v &= \frac{\pi u}{2K(k)}, \qquad q = \exp(-\pi K'(k)/K(k)), \qquad m = k^2, \qquad m_1 = 1-m,\\
\theta_c(u,q) &= \sqrt{\frac{2\pi\sqrt{q}}{\sqrt{m}K(k)}}\sum_{n=0}^{\infty} q^{n(n+1)}\cos((2n+1)v),\\
\theta_d(u,q) &= \sqrt{\frac{\pi}{2K(k)}}\left(1 + 2\sum_{n=1}^{\infty} q^{(n^2)}\cos(2nv)\right),\\
\theta_n(u,q) &= \sqrt{\frac{\pi}{2\sqrt{m_1}K(k)}}\left(1 + 2\sum_{n=1}^{\infty}(-1)^n q^{(n^2)}\cos(2nv)\right),\\
\theta_s(u,q) &= \sqrt{\frac{2\pi\sqrt{q}}{\sqrt{mm_1}K(k)}}\sum_{n=0}^{\infty}(-1)^n q^{n(n+1)}\sin((2n+1)v).
\end{aligned}$$

The sums have the same convergence behavior as those of the Jacobian theta functions. If code size not an issue, it is better to use the sums instead of the definitions as ratios of Jacobian theta functions.

Mathematica supplies the Neville theta functions as `NevilleThetaC[z,m]` through `NevilleThetaS[z,m]`, where $m = k^2$, but the functions appear to be absent from other algebra systems. We provide them in the mathcw library with the prefix `ELN` as the function families `ELNTC(u,q)`, `ELNTD(u,q)`, `ELNTN(u,q)`, and `ELNTS(u,q)`, and we compute them either from their definitions as ratios of Jacobian theta functions, or directly from their sums. A compile-time preprocessor symbol selects between the two choices, with the default of direct sums.

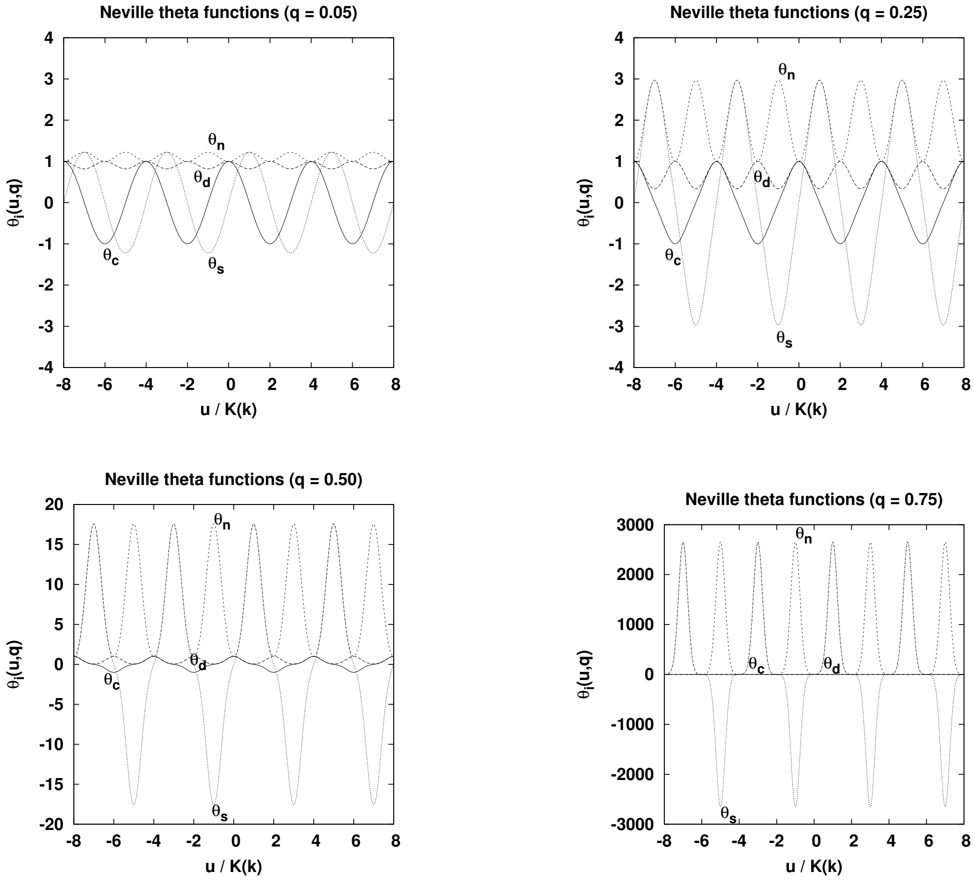


Figure 20.14: Neville theta functions. The horizontal axis is scaled to multiples of $K(k)$ to emphasize the periodicity. For large q, $\theta_s(u,q)$ and $\theta_n(u,q)$ dominate, and the oscillations in $\theta_c(u,q)$ and $\theta_d(u,q)$ are not visible at $q = 0.75$.

20.21 Jacobian Eta, Theta, and Zeta functions

The members of the elliptic-function family that we describe in this section are credited to Jacobi, and named with uppercase Greek letters, two of which have the same letter shapes as Latin letters:

$$\begin{aligned}
v &= \frac{\pi u}{2K(k)}, && \textit{scaled argument,}\\
q &= \exp(-\pi K'(k)/K(k)), && \textit{elliptic nome,}\\
H(u,k) &= \theta_1(v,q), && \textit{Jacobian Eta function,}\\
\Theta(u,k) &= \theta_4(v,q), && \textit{Jacobian Theta function,}\\
Z(u,k) &= \frac{\partial \log(\Theta(u,k))}{\partial u}, && \textit{Jacobian Zeta function,}\\
&= \frac{\Theta'(u,k)}{\Theta(u,k)},\\
&= \frac{\pi}{2K(k)}\frac{\theta_4'(v,q)}{\theta_4(v,q)},\\
&= \frac{v}{u}\frac{\theta_4'(v,q)}{\theta_4(v,q)}.
\end{aligned}$$

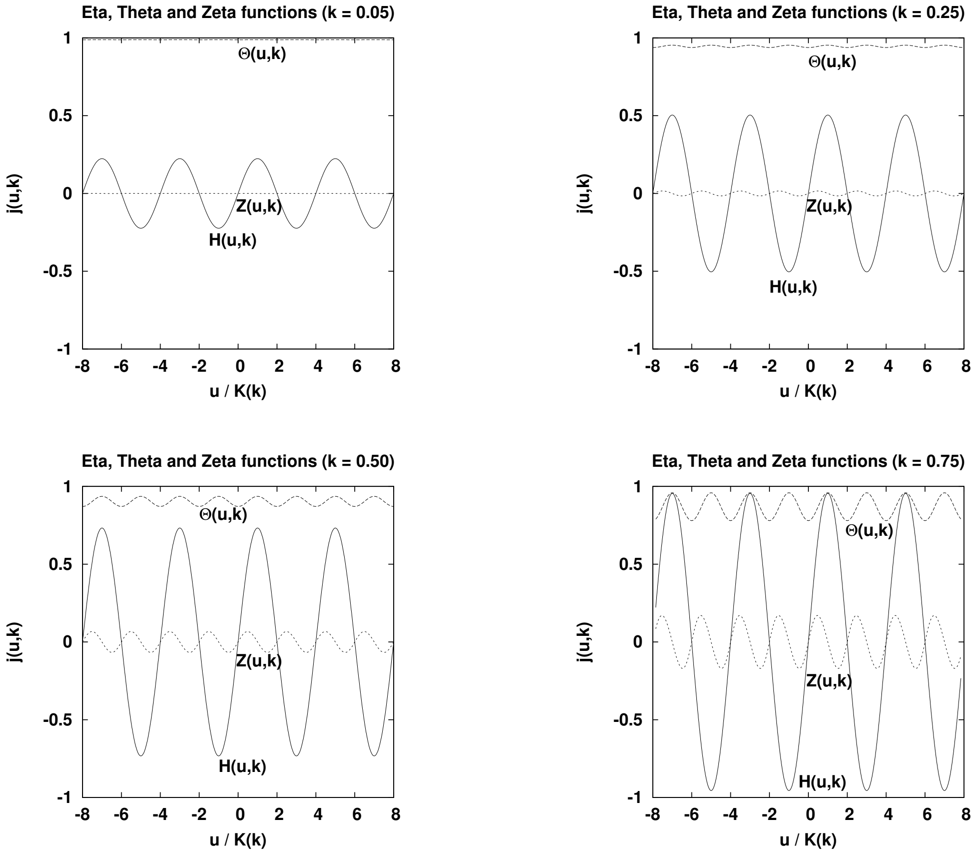


Figure 20.15: Jacobian Eta, Theta, and Zeta functions. The horizontal axis is scaled to multiples of $K(k)$ to emphasize the locations of extrema and zeros at $u = nK(k)$.

Here, ∂ is the symbol for a partial derivative, and is variously pronounced as *curly dee*, *del*, *dye*, or *partial*. The functions are periodic, single valued, and without singularities except at $u = \infty$. The first two are easy to compute from the Jacobian theta functions. The Zeta function needs a derivative of a theta function, and we can find it from the formulas for logarithmic derivatives given in **Section 20.19** on page 675.

Here are some special values of these functions:

$$k' = \sqrt{1-k^2},$$

$$\Theta(0,k) = \Theta(K(k),k) = \sqrt{\frac{2k'K(k)}{\pi}},$$

$$H(0,k) = 0, \qquad H(K(k),k) = \sqrt{\frac{2kK(k)}{\pi}},$$

$$H(u,0) = 0 \times \operatorname{sign}(u), \qquad \Theta(u,0) = 1,$$

$$H(u,1) = 0 \times \operatorname{sign}(u), \qquad \Theta(u,1) = 0,$$

$$Z(0,k) = Z(\pi/2,k) = 0 \qquad Z(u,1) = \operatorname{sn}(u,1) = \tanh(u).$$

The functions are plotted in **Figure 20.15**, and have these symmetry and periodicity properties:

$$\Theta(-u,k) = +\Theta(u,k), \qquad \Theta(u+2K(k),k) = +\Theta(u,k),$$

$$H(-u,k) = -H(u,k), \qquad H(u+2K(k),k) = -H(u,k),$$

$$Z(-u,k) = -Z(u,k), \qquad Z(u+2K(k),k) = +Z(u,k),$$

$$H(2nK(k),k) = 0, \qquad Z(nK(k),k) = 0, \qquad \text{for } n = 0,1,2,\ldots.$$

The Theta and Zeta functions can also be computed from the arithmetic-geometric mean (AGM) algorithm result arrays returned by our `ELJAG()` function [AS64, §16.35]:

$$\begin{aligned}
m &= k^2, \qquad m_1 = 1 - m, \\
a_0 &= 1 \qquad b_0 = \sqrt{m_1} \qquad c_0 = \sqrt{m}, \\
\log(\Theta(u,k)) &= \tfrac{1}{2}\log\left(\frac{2\sqrt{m_1}K(k)}{\pi}\right) + \tfrac{1}{2}\log\left(\frac{\cos(\phi_1 - \phi_0)}{\cos(\phi_0)}\right) + \\
&\quad \tfrac{1}{4}\log(\sec(2\phi_0 - \phi_1)) + \tfrac{1}{8}\log(\sec(2\phi_1 - \phi_2)) + \cdots + \\
&\quad 2^{-(n+1)}\log(\sec(2\phi_{n-1} - \phi_n)), \\
Z(u,k) &= c_1\sin(\phi_1) + c_2\sin(\phi_2) + \cdots + c_n\sin(\phi_n).
\end{aligned}$$

Although those relations are straightforward to evaluate, they are subject to subtraction loss in many places, and the errors are magnified near poles of the secant function (recall that $\sec(\phi) = 1/\cos(\phi)$). It is therefore preferable to compute $\Theta(u,k)$ and $Z(u,k)$ from their definitions in terms of Jacobian theta functions.

The mathcw library function families `ELJH(u,q)`, `ELJT(u,q)`, and `ELJZ(u,q)` compute the Jacobian Eta, Theta, and Zeta functions from their relations to the $\theta_i(u,q)$ functions.

The Jacobian Zeta function is related to the Legendre elliptic functions of the first and second kind like this:

$$\begin{aligned}
u &= F(\phi,k), && \text{for given } \phi \text{ and } k, \\
Z(u,k) &= E(\phi,k) - uE(k)/K(k).
\end{aligned}$$

When the Zeta function is small, it may be the result of cancellation of the terms on the right-hand side, so the formula does not provide a stable algorithm for determining the Zeta function. Nevertheless, it provides an independent test of software implementations of *five* different elliptic functions, provided that the terms are suitably rearranged to avoid subtraction loss.

A large test program, `chkell.c`, evaluates several of the mathematical relations between all of the elliptic functions discussed in this chapter. Here is a fragment of its output on an AMD64 system for the `double` data type:

```
% dgcc -DMAXTEST=100000000 -I.. chkell.c ../libmcw.a && ./a.out

Test of ELJZ() versus ELLFI() & ELLEI() with 100000000 random arguments

Average error   =  0.003 ulps
Maximum error   =  4.057 ulps
phi             =  0.10803920670680306 [  0x1.ba8751b7dd324p-4 ]
k               = -0.97475380456865568 [ -0x1.f312ee408c436p-1 ]
E               =  1.0603326785713263  [  0x1.0f71f661580b9p+0 ]
K               =  2.9097357998829532  [  0x1.747239023fb42p+1 ]
u = F(phi,k)    =  0.10823944089966656 [  0x1.bb5947acd2812p-4 ]
ELJZ(u,k)       =  0.06839625803509389 [  0x1.1826acb6df412p-4 ]
ELLEI(u,k)      =  0.10803876155813456 [  0x1.ba86da397b00bp-4 ]
u * E / K       =  0.03944338049551541 [  0x1.431ec3a8b397dp-5 ]
EI - u * E / K  =  0.06859538106261914 [  0x1.18f778652134cp-4 ]
test(ELJZ(u,k)) =  0.10783963853060929 [  0x1.b9b60e8b390d0p-4 ]
test(ELLEI())   =  0.10783963853060939 [  0x1.b9b60e8b390d7p-4 ]
exact results   =  99536754 / 100000000 =  99.54%
stddev(err)     =  0.049 ulps
```

The test program uses fused multiply-add operations to construct the value of $E(\phi,k) - uE(k)/K(k)$ and recover the error of division. Considering the amount of independent code, and the number of numerical operations required, it is gratifying that all but a half percent of the tests show exact agreement. The worst-case error is only about twice the two-ulps design target of the mathcw library, and the small standard deviation of the error suggests that such large errors are likely to be rare.

20.22 Weierstrass elliptic functions

The Nineteenth Century German mathematician Karl Weierstrass spent much of his long career working on elliptic functions, and is one of those rare mathematicians with a special symbol attached to his name. The letter $\wp$ is a stylized script p that identifies the Weierstrass elliptic function, although in older books, typographical limitations sometimes force it to be written in calligraphic ($\mathcal{P}$, $\mathscr{P}$) or fraktur ($\mathfrak{P}, \mathfrak{p}$) styles.

The *Handbook of Mathematical Functions* devotes an entire chapter to the Weierstrass functions [AS64, Chapter 18] [OLBC10, Chapter 23], whereas the *Handbook of Elliptic Integrals* relegates them to a section of an appendix [BF71, pages 308–315]. There are also brief treatments in [Car77, §8.2] and [GRJZ07, §8.16]. More detail can be found in [Law89, Chapters 6 and 7], which describes the properties of the Weierstrass functions, and treats their applications to the motion of spinning bodies — ballistic projectiles, gyroscopes, spherical pendulums, and tops. Most of the other books that we have cited in this chapter, and those on the computation of elementary and special functions, make no mention at all of the Weierstrass functions, except for a short discussion, with C code, in [Bak92].

The Weierstrass $\wp$-function is normally introduced by defining its *inverse* as a particular elliptic integral that is easily expressed with one of Carlson's R-functions (see **Section 20.8** on page 645):

$$\begin{aligned}
\wp^{-1}(u, e_1, e_2, e_3) &= \int_u^\infty (4(t-e_1)(t-e_2)(t-e_3))^{-1/2}\,dt \\
&= \tfrac{1}{2}\int_u^\infty ((t-e_1)(t-e_2)(t-e_3))^{-1/2}\,dt \\
&= R_F(u-e_1, u-e_2, u-e_3), && \textit{Carlson form,} \\
&\approx \mathrm{ellrf}(u-e_1, u-e_2, u-e_3), && \textit{mathcw library form.}
\end{aligned}$$

The parameters e_n are usually *distinct* real numbers, although symbolic-algebra systems permit them to be complex. The integrand is symmetric in the e_n, so we can always arrange the last three parameters in descending order. Weierstrass chose them under the constraint that their sum is zero, so there are really only two independent parameters that are usually written as g_2 and g_3, with these relations to the e_n values:

$$\begin{aligned}
&e_1 > e_2 > e_3, && \textit{distinct and in descending order,} \\
&g_1 = e_1 + e_2 + e_3 = 0, && \textit{zero-sum constraint,} \\
&g_2 = -4(e_1e_2 + e_1e_3 + e_2e_3), && \textit{sum of distinct pairs,} \\
&\quad = 2(e_1^2 + e_2^2 + e_3^2), && \textit{sum of squares,} \\
&\quad = \sqrt{8(e_1^4 + e_2^4 + e_3^4)}, && \textit{sum of fourth powers,} \\
&g_3 = 4e_1e_2e_3, && \textit{distinct product,} \\
&\quad = \tfrac{4}{3}(e_1^3 + e_2^3 + e_3^3), && \textit{sum of cubes.}
\end{aligned}$$

The constraint on the e_n simplifies the cubic polynomial in the reciprocal square root in $\wp^{-1}(u, e_1, e_2, e_3)$ by eliminating the quadratic term:

$$\begin{aligned}
4(t-e_1)(t-e_2)(t-e_3) &= 4t^3 - 4(e_1+e_2+e_3)t^2 + \\
&\qquad 4(e_1e_2 + e_1e_3 + e_2e_3)t - 4e_1e_2e_3 \\
&= 4t^3 - g_2t - g_3.
\end{aligned}$$

The e_n are clearly the roots of the cubic polynomial, and given only the g_n, recovering the e_n means that we have to solve the cubic for those roots. Explicit formulas are known for the roots of a cubic equation, but they are complicated expressions. Here is what the roots look like for the Weierstrass cubic:

$$\begin{aligned}
a &= 27g_3 + 3\sqrt{-3g_2^3 + 81g_3^2}, && \textit{first intermediate variable,} \\
b &= \sqrt[3]{a}, && \textit{second intermediate variable,} \\
c &= g_2/b, && \textit{third intermediate variable,} \\
e_1 &= \tfrac{1}{6}(b + 3c), && \textit{first root,}
\end{aligned}$$

$$\begin{aligned} e_2 &= -\tfrac{1}{12}\left(b + 3c + \sqrt{3}(b-3c)i\right), && \textit{second root,}\\ e_3 &= -\tfrac{1}{12}\left(b + 3c - \sqrt{3}(b-3c)i\right), && \textit{third root.} \end{aligned}$$

Roots e_2 and e_3 are complex conjugates, and even when the three roots are real, the variables a, b, and c are in general complex.

Although the symbolic-algebra system routines mentioned later use arguments g_2 and g_3, our computational formulas require the roots e_n. It therefore appears most sensible for code that works only in real arithmetic to use arguments e_1 and e_2. If we instead start with g_2 and g_3, then the code requires extra compute time and complex arithmetic to solve the cubic equation, and we introduce unnecessary errors in e_1 and e_2. For programming convenience, we therefore provide two families, `ELWE(&e1, &e2, g2, g3)` and `ELWG(&g2, &g3, e1, e2)` for converting between the differing argument conventions, and hiding the complex arithmetic. The functions remember their last arguments and results for quick handling of the common case of repeated calls with unchanged arguments.

Because of its direct relation to the Carlson $R_F()$ function, the inverse Weierstrass elliptic function has this scaling relation:

$$\wp^{-1}(su, se_1, se_2) = s^{-1/2}\wp^{-1}(u, e_1, e_2), \qquad \textit{for nonzero scale factor } s.$$

With the introduction of some additional variables, the Weierstrass elliptic integral can be converted to standard Legendre form [BF71, page 308]:

$$\begin{aligned} k &= \sqrt{\frac{e_2 - e_3}{e_1 - e_3}}, && \textit{elliptic modulus,}\\ &= \sin(\alpha), && \textit{elliptic angle,}\\ r &= \sqrt{e_1 - e_3}, && \textit{new variable } r,\\ t &= e_1 + (r\cot(\theta))^2, && \textit{new variable } \theta,\\ &= e_3 + (r/s)^2, && \textit{new variable } s,\\ \wp^{-1}(u, e_1, e_2, e_3) &= \wp^{-1}(u, g_2, g_3), && \textit{alternate form,}\\ &= \frac{1}{r}\int_0^{\phi}(1 - (k\sin(\theta))^2)^{-1/2}\,d\theta, && \textit{Legendre form,}\\ &= \frac{1}{r}\int_0^{y}((1-s^2)(1-(ks)^2))^{-1/2}\,ds, && \textit{Jacobi form,}\\ &= \frac{1}{r}F(\phi\backslash\alpha), && \textit{first kind,}\\ \phi &= \operatorname{am}(ru, k), && \textit{integration limit,}\\ y &= \operatorname{sn}(ru, k), && \textit{integration limit.} \end{aligned}$$

However, Carlson's $R_F()$ function is clearly a much simpler representation of the inverse Weierstrass elliptic function.

Because the Jacobian elliptic function amplitude is the inverse of the Legendre incomplete elliptic function of the first kind (see **Section 20.15** on page 657), the Weierstrass elliptic function is related to the Jacobian elliptic functions like this ([AS64, §18.9] and [BF71, page 309]):

$$\wp(z, e_1, e_2, e_3) = e_1 + (r\operatorname{cs}(rz, k))^2 = e_2 + (r\operatorname{ds}(rz, k))^2 = e_3 + (r\operatorname{ns}(rz, k))^2.$$

We switched arguments here from real u to complex z to emphasize that the Weierstrass elliptic function is defined over the entire complex plane, except at its poles.

The last relation may provide an optimal way to evaluate the Weierstrass elliptic function, because the needed Jacobian function is likely to be more accurate than the other two when $k \approx \pm 1$. However, because $e_3 < 0$, the formula may be subject to subtraction loss. If it is, then the first relation is a better choice.

The Weierstrass elliptic function satisfies these symmetry and scaling relations:

$$\begin{aligned} \wp(-z, e_1, e_2) &= \wp(z, e_1, e_2), & \wp(sz, s^{-2}e_1, s^{-2}e_2) &= s^{-2}\wp(z, e_1, e_2),\\ \wp(-z, g_2, g_3) &= \wp(z, g_2, g_3), & \wp(sz, sg_2, sg_3) &= s^{-2}\wp(z, s^5 g_2, s^7 g_3). \end{aligned}$$

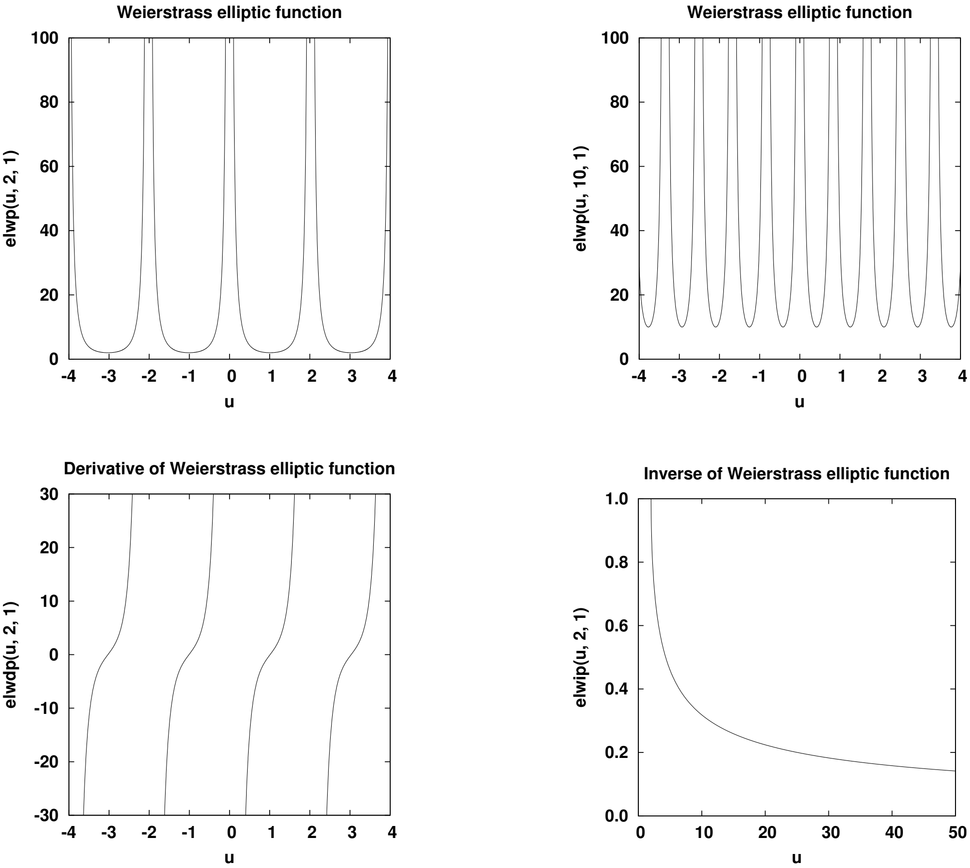


Figure 20.16: The Weierstrass elliptic function, its first derivative, and its inverse. The sharp peaks in $\wp(u, e_1, e_2)$ correspond to poles of the function. For the indicated roots, the inverse function is not defined for $u < 2$, but is finite at $u = 2$.

Derivatives of the Weierstrass elliptic function have relatively simple forms:

$$\begin{aligned} p &= \wp(z, g_2, g_3), && \textit{convenient shorthand,} \\ \wp'(z, g_2, g_3) &= p' = \pm\sqrt{4p^3 - g_2 p - g_3}, && \textit{first derivative,} \\ \wp''(z, g_2, g_3) &= p'' = 6p^2 - \tfrac{1}{2}g_2, && \textit{second derivative,} \\ \wp'''(z, g_2, g_3) &= 12pp', && \textit{third derivative,} \\ \wp^{(4)}(z, g_2, g_3) &= 12((p')^2 + pp''), && \textit{fourth derivative.} \end{aligned}$$

The sign of the first derivative is uncertain, because it is mathematically defined as the square of the relation shown here. Our code sets the sign by estimating the derivative numerically, but that may not be reliable for some argument regions. A mathematically correct sign requires another function, the Weierstrass sigma function, that we describe and implement later.

Figure 20.16 shows the periodicity of the Weierstrass elliptic function for two different choices of roots e_n. It also shows the derivative and inverse for one set of roots.

The *Handbook of Mathematical Functions* uses an alternate parameterization of the Weierstrass function, with two arguments, ω_1 and ω_2, that are the periods of the function:

$$k = \sqrt{\frac{e_2 - e_3}{e_1 - e_3}}, \qquad \textit{elliptic modulus,}$$

$$
\begin{aligned}
\omega_1 &= K(k)/\sqrt{e_1 - e_3}, && \textit{period of real part,} \\
\omega_2 &= iK'(k)/\sqrt{e_1 - e_3}, && \textit{period of imaginary part,} \\
\wp(z|\omega_1, \omega_2) &= \wp(z, g_2, g_3) = \wp(z, e_1, e_2), && \textit{alternate notations,} \\
W &= 2m\omega_1 + 2n\omega_2, && \textit{for integers } m \textit{ and } n, \\
g_2 &= 60 \sum_{m,n=-\infty}^{+\infty} W^{-4}, && \textit{exclude term with } m = n = 0, \\
g_3 &= 140 \sum_{m,n=-\infty}^{+\infty} W^{-6}, && \textit{exclude term with } m = n = 0.
\end{aligned}
$$

Those relations are all that we need to convert between any pair of (ω_1, ω_2), (g_2, g_3), and (e_1, e_2). However, for computation, it is clearly much simpler to work with the e_n and one of three Jacobian elliptic functions. The sums of inverse powers of W converge poorly if $|W| < 1$, which happens if the roots e_n are widely separated. We therefore do not provide any library support for converting from (ω_1, ω_2) to the other argument pairs, but we do supply `ELWO(&omega1, &omega2, e1, e2)` for deriving the periods from the roots. To avoid complex arithmetic in the code, the returned value `omega2` must be multiplied by the imaginary unit, i, to obtain the mathematical value ω_2. In practical applications, it is almost certain that either the e_n or the g_n are known from the problem specification, and the ω_k are then derived values that are of interest only because they determine the periods of the real and imaginary parts of the Weierstrass functions.

Maple provides the Weierstrass elliptic function as `WeierstrassP(z, g2, g3)`, but does not offer the inverse of that function. It also supplies `WeierstrassPPrime(z, g2, g3)` for computing the first derivative. Mathematica has `WeierstrassP[z, {g2, g3}]`, `InverseWeierstrassP[z, {g2, g3}]`, and `WeierstrassPPrime[z, {g2, g3}]`, where the g_n arguments must be grouped in a braced list.

PARI/GP has substantial support for algebra with elliptic curves, but not particularly for elliptic functions. However, it is possible to coax out a value of the Weierstrass elliptic function with a call to `ellwp(ellinit([0, 0, 0, -g2/4, -g3/4]), z)`.

None of the other symbolic-algebra systems mentioned in this book appears to support the Weierstrass elliptic functions.

In the mathcw library, we provide the families `ELWP(u, e1, e2)` and `ELWIP(u, e1, e2)` for the Weierstrass elliptic function and its inverse, and `ELWDP(u, e1, e2)` for the first derivative. Their code is a straightforward implementation of the relations given in this section of those functions to the Jacobian, Carlson, and Weierstrass elliptic functions.

If $e1 \approx e2$, then the derived modulus $k \approx 1$, and accurate determination of $K(k)$ requires a reliable value of $k - 1$. Consequently, the mathcw library provides another conversion family, `ELWK(&k, &km1, e1, e2)` to compute both k and $k - 1$. It works by rewriting the expression inside the square root like this:

$$
\begin{aligned}
k &= \sqrt{(e_2 - e_3)/(e_1 - e_3)} \\
&= \sqrt{(e_1 - e_3 - e_1 + e_2)/(e_1 - e_3)} \\
&= \sqrt{1 - (e_1 - e_2)/(e_1 - e_3)} \\
&= \sqrt{1 + x}, \\
x &= -(e_1 - e_2)/(e_1 - e_3), \\
k - 1 &= \sqrt{1 + x} - 1 \\
&= d.
\end{aligned}
$$

To find d, `ELWK()` defines a function $f(d) = (1 + x) - (1 + d)^2$, and then uses fast Newton–Raphson iterations (see **Section 2.2** on page 8) to improve the initial estimate of d. By ensuring an accurate starting point, the quadratic convergence of the Newton–Raphson algorithm means that at most two iterations are needed. The critical code fragment in file `elwkx.h` looks like this:

```
r = (e2 - e3) / (e1 - e3);
```

```
if (r <= FP(0.25))              /* then k <= 1/2 */
{
    k = SQRT(r);
    km1 = ONE - k;
}
else                            /* 1/2 < k */
{
    fp_t d, x;
    volatile fp_t onepx;

    x = -(e1 - e2) / (e1 - e3);

    onepx = ONE + x;
    STORE(&onepx);

    if (onepx == ONE)           /* then x is tiny */
        d = (FP(0.5) + (FP(-0.125) + FP(0.0625) * x) * x) * x;
    else
        d = SQRT(onepx) - ONE;

    d = (d * d + x) / (TWO + d + d); /* 1st Newton-Raphson iteration */
    d = (d * d + x) / (TWO + d + d); /* 2nd Newton-Raphson iteration */
    km1 = d;
    k = ONE + km1;
}
```

When x is tiny, d is obtained from a third-order Taylor series in compact Horner form. Otherwise, a call to `SQRT()` finds an almost-correct value of d.

Weierstrass also introduced functions related to integrals of $\wp(u, g_2, g_3)$:

$$\zeta_w(u, g_2, g_3) = 1/u - \int_0^u (\wp(z, g_2, g_3) - 1/z^2)\, dz, \qquad \textit{zeta function,}$$

$$\zeta'_w(u, g_2, g_3) = -\wp(u, g_2, g_3), \qquad \textit{derivative of zeta function,}$$

$$\sigma_w(u, g_2, g_3) = u \exp(\int_0^u (\wp(z, g_2, g_3) - 1/z^2)\, dz), \qquad \textit{sigma function,}$$

$$\sigma'_w(u, g_2, g_3)/\sigma_w(u, g_2, g_3) = \zeta_w(u, g_2, g_3). \qquad \textit{logarithmic derivative of sigma function.}$$

We add the w subscript to distinguish them from other, and better-known, functions named by the Greek letters zeta and sigma, although publications about those functions omit that subscript.

The Weierstrass sigma and zeta functions satisfy these symmetry and scaling relations:

$$\begin{aligned}
\sigma_w(-z, e_1, e_2) &= -\sigma_w(-z, e_1, e_2), & \zeta_w(-z, e_1, e_2) &= -\zeta_w(-z, e_1, e_2),\\
\sigma_w(-z, g_2, g_3) &= -\sigma_w(-z, g_2, g_3), & \zeta_w(-z, g_2, g_3) &= -\zeta_w(-z, g_2, g_3),\\
\sigma_w(su, s^{-2}e_1, s^{-2}e_2) &= s\sigma_w(u, e_1, e_2), & \zeta_w(su, s^{-2}e_1, s^{-2}e_2) &= (1/s)\zeta_w(u, e_1, e_2),\\
\sigma_w(su, s^{-4}g_2, s^{-6}g_3) &= s\sigma_w(u, g_2, g_3), & \zeta_w(su, s^{-4}g_2, s^{-6}g_3) &= (1/s)\zeta_w(u, g_2, g_3).
\end{aligned}$$

The sigma and zeta functions are graphed in Figure 20.17 on the facing page. They are not elliptic functions, because they are not periodic. Instead, they are called *quasi-periodic*, because they obey these relations [BF71, pages 150 and 155]:

$$k = \sqrt{\frac{e_2 - e_3}{e_1 - e_3}}, \qquad \textit{elliptic modulus,}$$

$$q = \mathrm{nome}(k), \qquad \textit{elliptic nome,}$$

$$\eta_1 = -\frac{\pi^2}{12\omega_1^2}\frac{\theta_1'''(0, q)}{\theta_1'(0, q)}, \qquad \textit{intermediate constant,}$$

$$\sigma_w(u + 2\omega_1, e_1, e_2) = -\exp(2\eta_1(u + \omega_1))\sigma_w(u, e_1, e_2), \qquad \textit{sigma translation,}$$

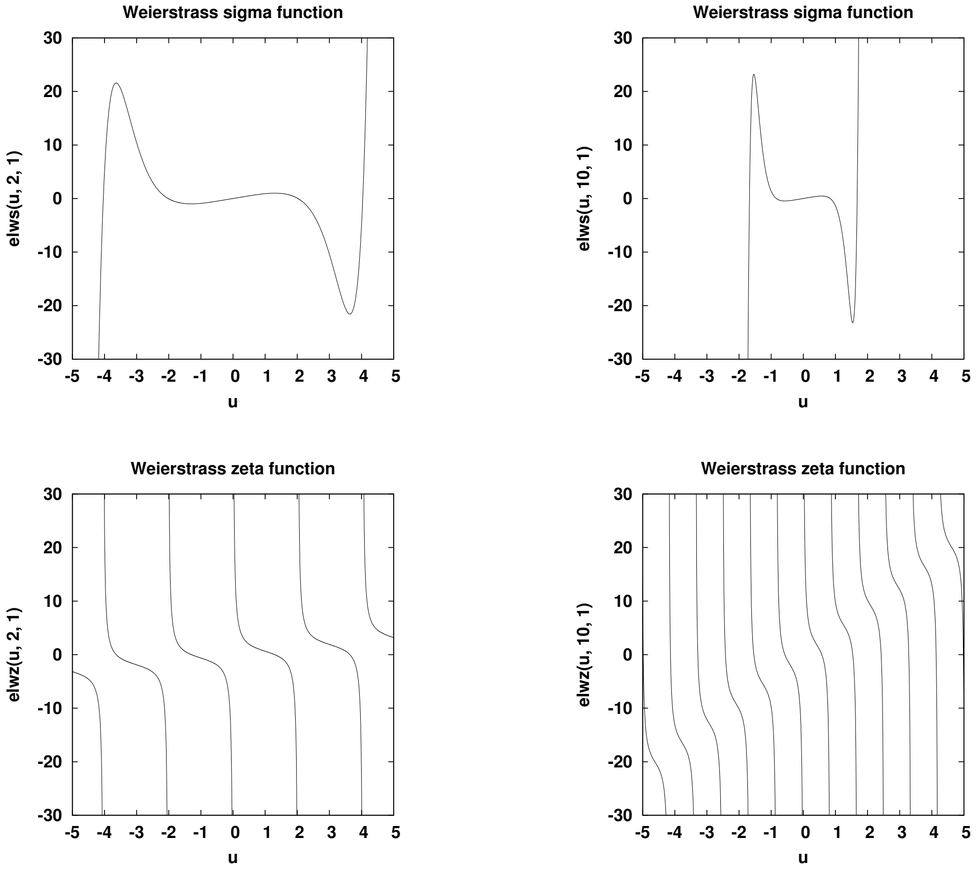


Figure 20.17: The Weierstrass sigma and zeta functions for $(e_1, e_2) = (2, 1)$ and $(10, 1)$.
The sigma function grows rapidly outside the range of u shown here, overflowing in IEEE 754 64-bit arithmetic before $|u| = 49$ (left) and $|u| = 19$ (right).
On the real axis, the zeta function has poles at even multiples of $\omega_1 \approx 1.009$ (left) and 0.419 (right).

$$\zeta_w(u + 2\omega_1, e_1, e_2) = \zeta_w(u, e_1, e_2) + 2\eta_1, \qquad \textit{zeta translation.}$$

The exponential function in the relation for the sigma function explains the growth seen in **Figure 20.17**. The additive constant $2\eta_1$ accounts for the upward movement of the zeta-function curves as u increases.

The Weierstrass sigma and zeta functions can be computed from their relations to the Jacobian Eta function and its derivatives [GRJZ07, §8.193]:

$$\begin{aligned}
k &= \sqrt{\frac{e_2 - e_3}{e_1 - e_3}}, && \textit{elliptic modulus,} \\
\lambda &= e_1 - e_3, && \textit{difference of extremal roots,} \\
\omega_1 &= K(k)/\sqrt{\lambda}, && \textit{period of real part,} \\
\eta_1 &= \zeta(\omega_1) = \frac{-\omega_1 \lambda}{3} \frac{H'''(0, k)}{H'(0, k)}, && \textit{intermediate variable,} \\
\sigma_w(u, e_1, e_2) &= \frac{1}{\sqrt{\lambda}} \exp\left(\frac{\eta_1 u^2}{2\omega_1}\right) \frac{H(u\sqrt{\lambda}, k)}{H'(0, k)}, && \textit{Weierstrass sigma function,} \\
\zeta_w(u, e_1, e_2) &= \frac{\eta_1 u}{\omega_1} + \sqrt{\lambda} \frac{H'(u\sqrt{\lambda}, k)}{H(u\sqrt{\lambda}, k)}, && \textit{Weierstrass zeta function.}
\end{aligned}$$

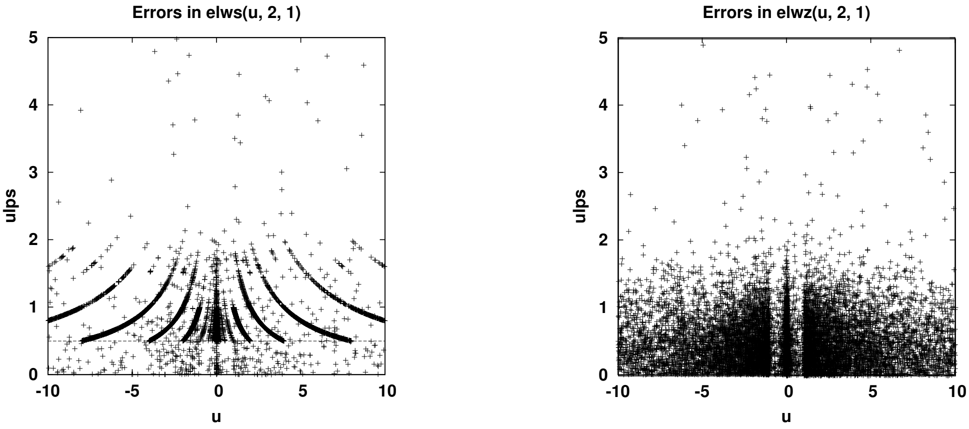


Figure 20.18: Errors in Weierstrass sigma and zeta functions for the IEEE 754 64-bit format, with e_n values chosen randomly from a logarithmic distribution in the interval $[\texttt{FP\_T\_MIN}, 100]$. Although most errors are small for the illustrated range of u, peak errors reach 1550 ulps for the sigma function, and 3750 ulps for the zeta function. Plots for other data types are qualitatively similar, and thus, not shown.

We consequently extend the mathcw library with the function family `ELJH4(result, u, k)` that computes a four-element vector with the values $H(u,k)$, $H'(u,k)$, $H''(u,k)$, and $H'''(u,k)$. Recall from **Section 20.21** on page 679 that $H(u,k)$ is a Jacobian theta function with a scaled argument, and that function in turn is the sum of an infinite Fourier series. The coefficients in that series depend on the elliptic nome, $q(k)$, but are independent of u, so the derivative affects only the trigonometric factor, $\sin((2n-1)v) = \sin(au)$. The first, second, and third derivatives of that factor are $a\cos(au)$, $-a^2\sin(au)$, and $-a^3\cos(au)$. We can therefore compute the n-th term of all four series with one call to our `SINCOS()` family, and when $u = 0$, we can avoid that call entirely, because we know that $\sin(0) = 0$ and $\cos(0) = 1$. The coefficients require powers of the nome, but are identical for all four series. Thus, we can compute the four results with just a bit more work than is needed in `ELJH(u,k)` for finding $H(u,k)$ alone. In addition, because an argument $u = 0$ occurs in three places in our formulas, we include separate code for that case, setting $H(0,k) = H''(0,k) = 0$ without unnecessary computation.

Like others in our elliptic-function family, our code for `ELJH4(result, u, k)` remembers the arguments and result array so that a subsequent call with unchanged arguments can return the saved results quickly, without further computation.

Maple supplies the Weierstrass sigma and zeta functions as `WeierstrassSigma(z, g2, g3)` and `WeierstrassZeta(z, g2, g3)`. Mathematica provides `WeierstrassSigma[z, {g2, g3}]` and `WeierstrassZeta[z, {g2, g3}]`. We provide them in the mathcw library in real arithmetic as the families `ELWS(u, e1, e2)` and `ELWZ(u, e1, e2)`, and both use `ELWK()` to compute accurate values of k and $k-1$. The measured errors in our implementation are shown in **Figure 20.18**.

The rapid growth in the sigma function, and the presence of poles in the zeta function, as shown in **Figure 20.17** on the previous page, tell us that we cannot expect high accuracy in the growth regions without access to high-precision arithmetic. Consequently, our implementations of those two functions cannot meet the accuracy goal of the mathcw library. We do not use the `hp_t` type in the code for the sigma and zeta function, but we do make use of their Taylor-series expansions to handle accurately the case of small arguments. We therefore recommend that users of those functions call only the functions of the highest precision available.

The Weierstrass sigma function provides an answer to the question of the correct sign of the first derivative of the Weierstrass elliptic function [Law89, page 158]:

$$\wp'(u,e_1,e_2) = -\sigma_w(2u,e_1,e_2)/\sigma_w^4(u,e_1,e_2).$$

The sigma function is real for real arguments, so the denominator on the right-hand side is necessarily positive. The sign of the derivative is therefore the opposite of that of the sigma function in the numerator. When an implementation of the sigma function is available, that relation also provides an independent check of our software in the

`ELWDP()` family.

20.23 Weierstrass functions by duplication

The Weierstrass functions satisfy argument-doubling relations ([AS64, §18.4], [OLBC10, §23.10]):

$$\begin{aligned}
\wp(2z, e_1, e_2) &= -2\wp(z, e_1, e_2) + \frac{1}{4}\left(\frac{\wp''(z, e_1, e_2)}{\wp'(z, e_1, e_2)}\right)^2, \\
\sigma_w(2z, e_1, e_2) &= -\wp'(z, e_1, e_2)(\sigma_w(z, e_1, e_2))^4, \\
\zeta_w(2z, e_1, e_2) &= 2\zeta_w(z, e_1, e_2) + \frac{\zeta_w'''(z, e_1, e_2)}{2\zeta_w''(z, e_1, e_2)}.
\end{aligned}$$

The derivatives needed in those equations have simple forms that require only the Weierstrass $\wp$ function:

$$\begin{aligned}
p &= \wp(z, g_2, g_3), & \zeta_w'(z, e_1, e_2) &= -p, \\
p' &= \pm\sqrt{4p^3 - g_2 p - g_3}, & \zeta_w''(z, e_1, e_2) &= -p', \\
p'' &= 6p^2 - \tfrac{1}{2}g_2, & \zeta_w'''(z, e_1, e_2) &= -p''.
\end{aligned}$$

Recall the advice at the end of the previous section on how to determine the correct sign of p'.

Coquereaux, Grossmann, and Lautrup [CGL90] published an algorithm for computing $\wp(z, e_1, e_2)$ that combines the duplication relations with rapidly convergent small-argument series expansions that are easily generated with a symbolic-algebra system:

$$\begin{aligned}
\wp(z, g_2, g_3) &= z^{-2} + \frac{g_2}{20}z^2 + \frac{g_3}{28}z^4 + \frac{g_2^2}{1200}z^6 + \frac{3g_2g_3}{6160}z^8 + \cdots, \\
\sigma_w(z, g_2, g_3) &= z - \frac{g_2}{240}z^5 - \frac{g_3}{840}z^7 - \frac{g_2^2}{161\,280}z^9 - \frac{g_2g_3}{2\,217\,600}z^{11} + \cdots, \\
\zeta_w(z, g_2, g_3) &= z^{-1} - \frac{g_2}{60}z^3 - \frac{g_3}{140}z^5 - \frac{g_2^2}{8400}z^7 - \frac{g_2g_3}{18\,480}z^9 - \cdots.
\end{aligned}$$

If z is small, then for a suitably chosen positive integer n, we divide z by 2^n (an exact operation in binary arithmetic) to get a tiny value for which the series can be summed to machine precision. We then repeatedly apply the duplication formula n times to recover the original z, and the corresponding function value.

For the Weierstrass $\wp$ function, a hoc implementation that uses a three-term series is short:

```
func crlwp(z, e1, e2)                         \
{   # compute Weierstrass P(z, e1, e2) by duplication rule
    global __CRLWP_N, __TWO_TO_CRLWP_N

    e3 = -(e1 + e2)
    g2 = 2 * (e1**2 + e2**2 + e3**2)
    g3 = 4 * e1 * e2 * e3
    z0 = z / __TWO_TO_CRLWP_N
    z0sq = z0 * z0
    pk = 1 / z0sq + ((g2 / 20) + (g3 / 28) * z0sq) * z0sq

    for (k = 1; k <= __CRLWP_N; ++k)          \
    {
        pksq = pk * pk
        t = (6 * pksq - g2 / 2)**2
        v = 4 * ((4 * pksq - g2) * pk - g3)
        pk = -2 * pk + t / v
    }

    return (pk)
}
```

There are, of course, practical considerations that prevent our simple prototype from being a complete solution for computing $\wp(z, e_1, e_2)$:

- Arguments of ± 0, Infinity and NaN are not handled properly.
- What value do we choose for n (the global variable `__CRLWP_N`)? Execution time is proportional to n, so we prefer it to be small. However, a large value of n makes the series variable z_0 small, allowing the series expansion to be truncated to fewer terms.
- If n is too large, then for small z, we may find that $z/2^n$ underflows to subnormals or zero in floating-point arithmetic. That makes the initial `pk` large but inaccurate, or infinite, and the loop produces a NaN result from division of infinities.
- If z is sufficiently small, then only the leading term of the series matters, and we can immediately return z^{-2}. The code must therefore be extended to make such a check, and in a C implementation, also set `errno` to `ERANGE`.
- For a fixed n, numerical experiments for a reasonable range of arguments z, e_1, and e_2 can compare the computed function values with those from a higher-precision implementation by another method, such as the `mathcw` library function family `ELWP(u, e1, e2)`, and determine the average and extreme errors. Repeating the experiments for different n values allows us to pick an optimal n for each floating-point format. Such experiments recommend values $n = 4, 7, 9$, and 15 for a three-term series in the 32-bit, 64-bit, 80-bit, and 128-bit IEEE 754 formats for z in $(0, 1]$ and e_1 and e_2 in $(0, 10]$. Using a five-term series reduces the term count from 15 to 11 in the last case.
- If the floating-point base is $\beta > 2$, then the initial argument reduction and repeated doublings are no longer exact operations, so we introduce argument error.
- How does the error in the initial series, and in a nonbinary base, in the reduced and doubled arguments, propagate?

The cited article shows mathematically that, for the Weierstrass $\wp$ function, error growth is not a serious problem, and our numerical experiments support that, with average errors in the 64-bit IEEE 754 format below 0.85 ulps, although maximum errors reach 1300 ulps for $e_1 \approx 10$.

20.24 Complete elliptic functions, revisited

Fukushima observes that some of the elliptic functions required in astrophysics applications appear in rate-determining computational steps, so fast implementations are desirable [FI94, Fuk09a, Fuk09b, Fuk10, Fuk11, Fuk12, Fuk13a, Fuk13b]. In his recent treatment [Fuk09a] of the complete elliptic functions, $K(k)$ and $E(k)$, Fukushima uses separate polynomial fits centered on intervals of width 0.1 for k in $[0, 0.8]$, and for k in $[0.8, 0.85]$ and $[0.85, 0.9]$. For the more difficult region with k in $[0.9, 1]$, where the functions exhibit logarithmic behavior, $K(k) \to \infty$ and $E(k) \to 1$, he uses a relation that we could not introduce at the start of this chapter because it requires the complementary nome, q':

$$\begin{aligned} K(k) &= -\log(q')K'(k)/\pi, \\ E(k) &= K(k) + (\tfrac{1}{2}\pi - E'(k)K(k))/K'(k). \end{aligned}$$

The relation for $E(k)$ is just a rewriting of the *Legendre relation*. Because there is severe subtraction loss in that relation as $k \to 1$, Fukushima goes to considerable trouble to find an auxiliary function that reduces that error. For $k > 0.9$, we have $q' < 0.014$, so Fukushima uses a rapidly convergent series expansion to compute q'. Because $k \approx 1$ is rare in practice, the extra work to determine $K(k)$ and $E(k)$ for k in $[0.9, 1]$ is unlikely to matter much.

We therefore investigated a similar approach, using fits to rational polynomials with k in intervals of width $\frac{1}{8}$, except for the last interval, which is shortened to $[\frac{7}{8}, \frac{31}{32}]$. The fits are arranged so that the polynomial provides a small correction to an exactly representable value chosen near the average function value on the interval, and the polynomial variable is usually relative to the interval midpoint:

$$x = k - k_{\text{mid}}, \qquad K(k) = K_{\text{mid}} + \mathcal{R}(x), \qquad E(k) = E_{\text{mid}} + \mathcal{S}(x).$$

The polynomial contribution is less than 8% of the function value in all of the intervals, except for the last, where it does not exceed 13%, so there is never subtraction loss. The interval boundaries are exactly representable in both binary and decimal floating-point arithmetic, and the interval can be selected quickly via the branch table compiled from a `switch` statement.

For the final interval $[\frac{31}{32}, 1]$, instead of using the complementary functions and nome, and yet another complicated auxiliary function, we instead use the simple relations to two of Carlson's functions that we presented in **Section 20.10** on page 650:

$$e = 1 - k^2, \qquad K(k) = R_F(0, e, 1), \qquad E(k) = R_F(0, e, 1) - \tfrac{1}{3}k^2 R_D(0, e, 1).$$

The new code provides optional, longer, but faster and more accurate, alternatives to the inner blocks that invoke the arithmetic-geometric functions in the code for `ELLK()` and `ELLE()` shown in **Section 20.5** on page 632 and **Section 20.6** on page 638. Testing of the new code shows that the errors lie below 0.5 ulps, except in the small final interval, k in $[\frac{31}{32}, 1]$, that is rarely needed in applications. In that region, errors up to 1.2 ulps in `ELLK()`, and up to 7 ulps in `ELLE()`, can be removed entirely by computing in the next higher precision, when available.

20.25 Summary

The utility of the arithmetic-geometric mean for computation is not widely appreciated, and few books on numerical analysis even mention it. Despite its grade-school simplicity, we found that considerable care is needed to produce a robust library routine that is free of underflow and overflow, and correctly handles the special values of IEEE 754 arithmetic.

Although we do not use the AGM-like algorithms for the logarithm, exponential, inverse tangent, inverse cosine, and inverse hyperbolic cosine in our library code for those functions, the simplicity of those algorithms makes them useful for generating independent values for function testing. The catch is that, because the AGM is not self correcting, the test values must be computed in higher-than-working precision. The algorithms are also useful for implementing those functions in arbitrary-precision arithmetic, because they avoid precision- and range-dependent polynomial approximations.

We saw that the ordinary AGM is quadratically convergent as long as we handle zero arguments separately. However, the convergence of the AGM-like algorithms for the trigonometric and hyperbolic functions needs to be analyzed mathematically and tested numerically. Generating just one or two bits per iteration is unlikely to be the fastest way to compute a function in software.

Application of the AGM to the computation of the complete elliptic integrals of the first and second kinds appears straightforward at first, but we found that careful analysis is needed to mitigate the problem of severe subtraction loss, and premature overflow and underflow. We also discovered that even more care is required near the poles of $K(m)$ and $K'(m)$ to avoid returning a zero value, instead of Infinity, near those poles.

Computing the difficult incomplete elliptic integral functions was a daunting task before Carlson showed that choosing symmetric functions has great advantages, and that a few iterations of the duplication rule, followed by summing a short Taylor-series expansion, is a practical way to compute the basic four functions. Application of the scaling rules eliminates the problem of premature overflow and underflow, and shortens the inner loops. Carlson's algorithms are not as fast as the AGM algorithm for the easier complete elliptic integral functions, but are quick enough to be practical. They also eliminate the need for different approximations in different argument regions, work for any number of arguments, and carry the additional bonus that the $R_C(x, y)$ auxiliary function provides an easy route to ten elementary functions, and does so with errors below 2.3 ulps in IEEE 754 arithmetic.

The Jacobian elliptic functions are computed with two-term Taylor-series expansions, and the vector AGM. The Jacobian theta functions are handled by summing infinite series to machine precision. The related Neville theta functions, and the Jacobian Eta, Theta, and Zeta functions are then easily obtained from the basic four Jacobian theta functions.

Although the Jacobian and Neville functions are periodic, exact argument reduction requires high-precision values of the complete elliptic function of the first kind, $K(k)$. That is impractical in programming languages that do not supply arbitrary-precision arithmetic, so our implementations in C necessarily suffer accuracy loss for large arguments u and v. There is also accuracy loss in $\mathrm{cn}(u, k)$ and $\mathrm{dn}(u, k)$, and functions that depend on them, when k is close to 1, and inadequate, or no, higher-precision arithmetic is available to compute the Jacobian amplitude, $\phi = \mathrm{am}(u, k)$, with sufficient extra digits to recover its difference from $\pm\frac{1}{2}\pi$ to machine accuracy.

Computation of three of the Weierstrass elliptic functions is straightforward because of their relations to functions developed earlier in the chapter. The major complication is the need for complex arithmetic to find the roots from the parameters g_2 and g_3, but our complex-as-real `fp_cx_t` data type and its support functions provide a portable solution that can be hidden inside the `ELWE()` family. The Weierstrass sigma and zeta functions depend on the complete elliptic function of the first kind, $K(k)$, and on the Jacobian Eta function and its low-order derivatives. Those values are calculated by the `ELJH4()` family.

To supplement the citations to various books and papers on the AGM iteration given in the first part of this chapter, the Web site `http://www.math.utah.edu/pub/tex/bib/index-table-a.html#agm` provides an extensive, and growing, bibliography on that subject.

21 Bessel functions

> EACH CYCLE OF A RECURSIVE PROCESS NOT ONLY GENERATES ITS OWN ROUNDING ERRORS, BUT ALSO INHERITS THE ROUNDING ERRORS COMMITTED IN ALL PREVIOUS CYCLES. IF CONDITIONS ARE UNFAVORABLE, THE RESULTING PROPAGATION OF ERRORS MAY WELL BE DISASTROUS.
>
> — WALTER GAUTSCHI
> *Computational Aspects of Three-Term Recurrence Relations* (1967).

A large family of functions known as Bessel[1] functions is treated in four chapters of the *Handbook of Mathematical Functions* [AS64, Chapters 9–12], with more coverage than any other named functions in that famous compendium. Although those functions were first discovered by Daniel Bernoulli (1700–1782), who in 1732 worked with the order-zero function that is now known as $J_0(x)$, it was Friedrich Wilhelm Bessel who generalized them about 1824 and brought them to mathematical prominence, and they bear his name, instead of that of Bernoulli. Leonhard Euler (1707–1783) discussed their generalizations to arbitrary integer orders, $J_n(x)$, in 1764. The definitive textbook treatment in English, first published in 1922, is *A Treatise on the Theory of Bessel Functions* [Wat95], a revised reprint of the 1944 second edition.

The Bessel functions arise in several areas of astronomy, engineering, mathematics, and physics as solutions of certain equations containing unknown functions and their derivatives. Such equations are known as *differential equations*, a topic that is outside the scope of this book. The Bessel functions appear as analytic solutions of those equations, particularly in problems with cylindrical and spherical geometries, and older literature commonly refers to them as cylinder and sphere functions. Many textbooks on mathematical physics describe applications of Bessel functions to topics such as diffraction of light, electromagnetic potentials, planetary motion, radiation from moving charges, scattering of electromagnetic waves, and solutions of the Helmholtz equation, the Laplace equation and the Poisson equation [AW05, AWH13, Jac75, MH80, MF53, PP05].

Computation of Bessel functions is treated in several books on special functions, including [GDT$^+$05, Chapter 7], [GST07, Chapters 7 and 12], [HCL$^+$68, Section 6.8], [Luk69a], [Luk77, Chapters 17–19], [Mos89, Chapter 6], [Olv74, Chapters 2 and 7], [Tho97, Chapters 14–15], and [ZJ96, Chapters 5–11]. Software algorithms for the Bessel function family are also described in numerous research articles [ADW77a, ADW77b, Amo86, BS92, Cai11, Cam80, CMF77, Cod80, Cod83, CS89, CS91, Cod93b, Gau64, GST02, GST04, HF09, Har09b, Hil77, Hil81, Jab94, JL12, Kod07, Kod08, Kra14, Sko75, VC06, Wie99]. The journals *Mathematics of Computation* and *SIAM Journal on Mathematical Analysis* contain dozens of articles on the computation of Bessel functions, and their derivatives, integrals, products, sums, and zeros, although without software.[2] A search of the journals *Computer Physics Communications* and *Journal of Computational Physics* finds almost 50 articles on the computation of Bessel functions, and many more on their applications. The *MathSciNet* database has entries for about 3000 articles with *Bessel* in their titles, and the *zbMATH* database lists more than 8000 such articles.

Some of the Bessel functions that we treat in this chapter are special cases of more general functions, the *Coulomb wave functions* (see [AS64, Chapter 14] and [OLBC10, Chapter 33]), and some publications in the cited physics journals take that approach. However, it seems unlikely to this author that a three-parameter function that is even more difficult computationally can provide a satisfactory route to the two-parameter Bessel functions for the argument ranges and accuracy required for the mathcw library.

Bessel functions are normally defined in terms of real orders, ν (sometimes restricted to nonnegative values), and complex arguments, z, although complex orders are possible [Kod07]. However, this book deals mostly with the computation of functions of real arguments. Among all the functions treated in this book, the Bessel functions

[1] Friedrich Wilhelm Bessel (1784–1846) was a German astronomer and mathematician. Bessel achieved fame in astronomy, cataloging the positions of more than 50,000 stars, and was the first to use parallax to calculate the distance of stars from the Earth. He was appointed director of the Königsberg Observatory at the age of 26 by the King of Prussia. A large crater on the Moon, and Asteroid 1552 Bessel, are named after him.

[2] Extensive bibliographies of the contents of those journals are available at links from `http://www.math.utah.edu/pub/tex/bib/index-table.html`.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_21

are the most difficult to determine accurately, and their accurate computation for complex arguments is *much harder* than for real arguments. In this chapter, we adopt the convention that mathematical formulas are usually given for real ν and complex z, but when we discuss computation, we consider only integer orders n and real arguments x. Nevertheless, keep in mind that many applications of Bessel functions, particularly in physics problems, require real orders and complex arguments, and sometimes, the orders are large, and the magnitudes of the real and imaginary parts differ dramatically. Our implementations of several of the Bessel functions do not address those needs. Indeed, a completely satisfactory software treatment of those functions in freely available and highly portable software remains elusive. The *GNU Scientific Library* [GDT$^+$05] handles the case of real orders, but at the time of writing this, is restricted to real arguments.

21.1 Cylindrical Bessel functions

There are about two dozen members of the Bessel function family, as shown in **Table 21.1** on the next page. None of them is mentioned in the ISO C Standards, but implementations of the C library on various flavors of UNIX since the mid-1970s have included functions for computing $J_0(x)$, $J_1(x)$, and $J_n(x)$, as well as $Y_0(x)$, $Y_1(x)$, and $Y_n(x)$, but only for integer orders and real arguments. The Fortran 2008 Standard [FTN10] introduces them to that language with names like `bessel_j0(x)`. The POSIX Standards require those six functions, but confusingly, their software names are spelled in lowercase, despite the fact that they compute the *cylindrical*, rather than the *spherical*, Bessel functions. Their prototypes look like this:

```
double j0 (double x);              double y0 (double x);
double j1 (double x);              double y1 (double x);
double jn (int n, double x);       double yn (int n, double x);
```

There are companions for other floating-point types with the usual type suffixes. Particularly in the physics literature, the function of the second kind is commonly referred to as the Neumann function, $N_\nu(z)$, but it is identical to $Y_\nu(z)$. For real arguments, the functions of the first and second kinds have real values, but the functions of the third kind have complex values.

For the `mathcw` library, and this book, we implement the six Bessel functions required by POSIX, in each supported floating-point precision. We also supply the modified cylindrical Bessel companions, and the spherical functions that correspond to each of the supported cylindrical functions. To augment those scalar functions, we provide a family that returns in an array argument the values of sequences of Bessel functions with a single argument x and orders $k = 0, 1, 2, \ldots, n$. Such sequences are needed in series expansions with terms involving Bessel functions. The sequence values may be computable more economically than by separate invocations of the functions for specific orders and arguments.

The mathematical functions and software routines for the ordinary Bessel functions are related as follows:

$$J_0(x) \equiv \texttt{j0(x)} = \texttt{jn(0,x)}, \qquad J_1(x) \equiv \texttt{j1(x)} = \texttt{jn(1,x)},$$
$$Y_0(x) \equiv \texttt{y0(x)} = \texttt{yn(0,x)}, \qquad Y_1(x) \equiv \texttt{y1(x)} = \texttt{yn(1,x)}.$$

For mathematical reasons that are discussed later when we develop computer algorithms, implementations of order-n Bessel functions are almost certain to invoke the functions of a single argument directly for $n = 0$ and $n = 1$, rather than providing independent computational routes to those two particular functions.

Symbolic-algebra systems include many of the Bessel functions listed in **Table 21.1** on the facing page. The functions of interest in the first part of this chapter, $J_\nu(z)$ and $Y_\nu(z)$, are called

- `BesselJ(nu,z)` and `BesselY(nu,z)` in Maple and REDUCE,
- `BesselJ[nu,z]` and `BesselY[nu,z]` in Mathematica,
- `bessel_j(nu,z)` and `bessel_y(nu,z)` in Maxima,
- `besselJ(nu,z)` and `besselY(nu,z)` in Axiom and MuPAD, and
- `besselj(nu,z)` and `besseln(nu,z)` in PARI/GP.

Those systems permit `nu` to be real, and `z` to be complex, rather than restricting them to integer and real values, respectively.

Table 21.1: The Bessel function family. The subscripts are called the *order* of the function, and except as noted, may be positive or negative. The order ν is any real number, the order n is any integer, and the order k is any nonnegative integer. The argument z is any complex number, and for the Kelvin functions, the argument x is any nonnegative real number.
Some authors call the functions of the first kind the *regular* functions, and those of the second kind, *irregular* functions.

Function	Description
$J_\nu(z)$	ordinary Bessel function of the first kind
$Y_\nu(z)$	ordinary Bessel function of the second kind, sometimes called the Neumann function or Weber's function
$H_\nu(z)$	ordinary Bessel function of the third kind, also known as the Hankel function
$I_\nu(z)$	modified (or hyperbolic) Bessel function of the first kind
$K_\nu(z)$	modified (or hyperbolic) Bessel function of the second kind, also called the Basset function and the Macdonald function
$N_\nu(z)$	Neumann function, identical to $Y_\nu(z)$
$Z_\nu(z)$	arbitrary ordinary Bessel function of first, second, or third kinds,
$j_n(z)$	spherical Bessel function of the first kind, equal to $\sqrt{\pi/(2z)}J_{n+1/2}(z)$
$y_n(z)$	spherical Bessel function of the second kind, equal to $\sqrt{\pi/(2z)}Y_{n+1/2}(z)$
$h_n(z)$	spherical Bessel function of the third kind
$i_n(z)$	modified spherical Bessel function of the first kind, equal to $\sqrt{\pi/(2z)}I_{n+1/2}(z)$
$k_n(z)$	modified spherical Bessel function of the second kind, equal to $\sqrt{\pi/(2z)}K_{n+1/2}(z)$
$n_n(z)$	spherical Neumann function, identical to $y_n(z)$
$S_n(z)$	Riccati–Bessel function of the first kind, equal to $zj_n(z)$
$C_n(z)$	Riccati–Bessel function of the second kind, equal to $-zy_n(z)$
$\zeta_n(z)$	Riccati–Bessel function of the third kind, equal to $zh_n(z)$
$\mathrm{ber}_k(x)$	Kelvin (or Thomson) function of the first kind
$\mathrm{bei}_k(x)$	Kelvin (or Thomson) function of the first kind
$\mathrm{ker}_k(x)$	Kelvin (or Thomson) function of the second kind
$\mathrm{kei}_k(x)$	Kelvin (or Thomson) function of the second kind
$\mathrm{Ai}(z)$	Airy function, equal to $(\sqrt{z}/3)\big(I_{-1/3}(\xi) - I_{+1/3}(\xi)\big)$, where ξ is the Greek letter *xi*, and also equal to $(1/\pi)\sqrt{z/3}K_{+1/3}(\xi)$, where $\xi = (2/3)z^{3/2}$
$\mathrm{Bi}(z)$	Airy function, equal to $\sqrt{z/3}\big(I_{-1/3}(\xi) + I_{+1/3}(\xi)\big)$
$\mathbf{H}_\nu(z)$	ordinary Struve function
$\mathbf{L}_\nu(z)$	modified Struve function
$\mathbf{J}_\nu(z)$	Anger's function
$\mathbf{E}_\nu(z)$	Weber's function

21.2 Behavior of $J_n(x)$ and $Y_n(x)$

A few of the ordinary Bessel functions of the first and second kinds are graphed in Figure 21.1 on the next page. From the plots, and selected numerical evaluations, we can make some important observations that are relevant to their computation:

- The functions of the first kind, $J_n(x)$, look like damped cosine ($n = 0$) and sine ($n > 0$) waves, but their zeros are not equally spaced. Instead, the zeros appear to get closer together as x increases, and differ for each value of n. That means that argument reductions like those that we found for the trigonometric functions are not applicable.

- Values of $J_n(x)$ lie in $[-1, +1]$, so intermediate overflow is unlikely to be a problem in their computation.

- The x position of the first positive maximum of $J_n(x)$ and $Y_n(x)$, and their first positive nonzero root, increase as n gets larger.

- For $n > 0$, $J_n(10^{-p}) \approx \mathcal{O}(10^{-np})$, so $J_1(x)$ is representable for tiny x, although the higher-order functions underflow to zero.

- The functions of the second kind, $Y_n(x)$, have single poles at $x = 0$ for all n, and their zeros appear to get closer together as x increases.

- The approach to the single pole in $Y_n(x)$ slows as n increases.

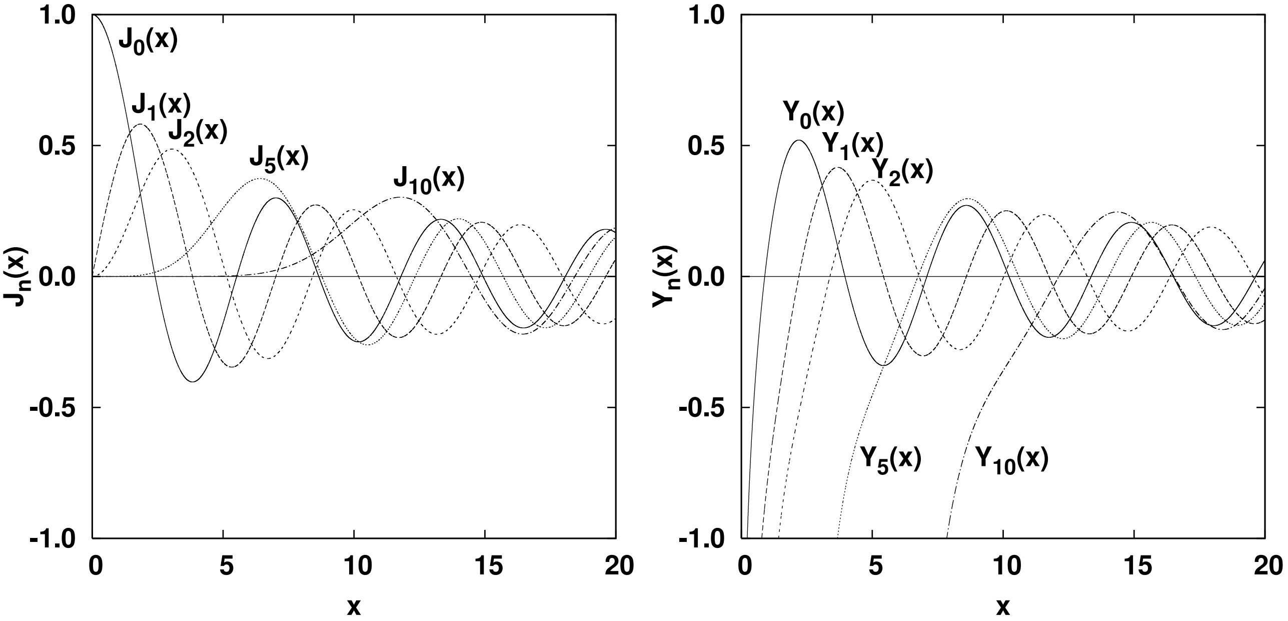


Figure 21.1: Ordinary Bessel functions of the first and second kinds, $J_n(x)$ and $Y_n(x)$. The solid lines, and highest positive maxima, correspond to $n = 0$.
The functions of the first kind can be extended to the negative axis through the relation $J_n(-x) = (-1)^n J_n(x)$, so even orders are symmetric about the origin, and odd orders are antisymmetric.
The functions of the second kind have real values only for $x \geq 0$; they have complex values when x is negative. For integer orders, they all go to $-\infty$ as x approaches zero.

- Values of $Y_n(x)$ lie in $(-\infty, 0.53]$, so overflow must be anticipated and handled for $x \approx 0$. Numerical evaluation with x set to the smallest representable floating-point number shows that $Y_0(x)$ is of modest size, but overflow occurs in $Y_n(x)$ for all $n > 0$.

- Although both $J_n(x)$ and $Y_n(x)$ decay as x increases, the attrition is not rapid. For example, we have $J_0(10^6) \approx 0.000\,331$, $Y_0(10^6) \approx -0.000\,726$, $J_0(10^{600}) \approx 0.449 \times 10^{-300}$, and $Y_0(10^{600}) \approx 0.659 \times 10^{-300}$. Thus, in floating-point arithmetic, the maxima of those functions *cannot underflow* for any representable x. Some deficient software implementations of those functions, such as those in the GNU/LINUX run-time libraries, suffer premature underflow over most of the range of x, or produce spurious NaN results. Here is an example from that operating system on an AMD64 platform when hoc is linked with the native math library:

```
hoc64> for (x = 1.0e7; x <= 1.0e18; x *= 10) \
hoc64>     printf("%.2g  % g\n", x, J0(x))
1e+07  -8.68373e-05
1e+08   3.20603e-05
1e+09  -qnan(0x31)
1e+10  -qnan(0x33)
1e+11  -qnan(0x35)
1e+12  -qnan(0x37)
1e+13  -qnan(0x39)
1e+14  -qnan(0x3b)
1e+15  -qnan(0x3d)
1e+16  -qnan(0x3f)
1e+17   0
1e+18   0
```

Native library improvements on that system removed some of those irregularities as this book was nearing completion. However, tests on more than a dozen flavors of UNIX found that only MAC OS X, OSF/1, and SOLARIS IA-32 (but not SPARC) produced expected results over most of the entire floating-point range, and even those lost all significant digits for large arguments. The conclusion is that implementations of the Bessel functions in UNIX libraries may be neither reliable nor robust.

- The oscillatory nature of the Bessel functions suggests that recurrence formulas are likely to suffer subtraction loss for certain ranges of x.
- When x is large, there is insufficient precision in a floating-point representation to resolve the waves of the functions, because consecutive floating-point numbers eventually bracket multiple wave cycles. The best that we can hope for then is to get the correct order of magnitude of the waves.

In the next few sections, we investigate those graphical observations further, and make them more precise.

21.3 Properties of $J_n(z)$ and $Y_n(z)$

The ordinary Bessel functions of the first and second kinds have these limiting values, where ν is any real number, z is a complex number, and $e = \exp(1)$:

$$\begin{aligned}
J_\nu(z) &\approx (z/2)^\nu/\Gamma(\nu+1), && z \to 0, \nu \neq -1, -2, -3, \ldots, \\
Y_0(z) &\approx (2/\pi)\log(z), && z \to 0, \\
Y_\nu(z) &\approx -(1/\pi)\Gamma(\nu)(2/z)^\nu && z \to 0, \nu > 0, \\
J_\nu(x) &\to \sqrt{2/(\pi x)}\cos(x - \nu\pi/2 - \pi/4), && x \to +\infty, x \gg \nu, \nu \geq 0, \\
Y_\nu(x) &\to \sqrt{2/(\pi x)}\sin(x - \nu\pi/2 - \pi/4), && x \to +\infty, x \gg \nu, \nu \geq 0, \\
J_\nu(x) &\to (1/\sqrt{2\pi\nu})(ex/(2\nu))^\nu, && \nu \to +\infty, \nu \gg x, \\
Y_\nu(x) &\to -(1/\sqrt{2\pi\nu})(ex/(2\nu))^{-\nu}, && \nu \to +\infty, \nu \gg x, \\
J_0(0) &= 1, \qquad Y_0(0) = -\infty, && \\
J_n(0) &= 0, \qquad Y_n(0) = -\infty, && n > 0, \\
J_n(\infty) &= 0, \qquad Y_n(\infty) = 0. &&
\end{aligned}$$

The large-argument limits of $J_\nu(x)$ and $Y_\nu(x)$ answer the question about the spacing of the roots: they are ultimately separated by π, rather than squeezing ever closer together, as might be suggested by the function graphs for small x. Table 21.2 on the following page shows a few of the roots, easily found in Maple with calls to `BesselJZeros(nu,k)` and `BesselYZeros(nu,k)`. For $k \gg \nu$, higher roots of $J_\nu(r_{\nu,k}) = 0$ and $Y_\nu(s_{\nu,k}) = 0$ can be estimated to about three correct figures by the formulas

$$r_{\nu,k} \approx (k + \nu/2 - 1/4)\pi, \qquad s_{\nu,k} \approx (k + \nu/2 - 3/4)\pi.$$

In particular, that relation shows that the roots of the Bessel functions of orders $\nu - 1, \nu, \nu + 1, \ldots$ are well separated, a fact that has significance later in their evaluation by recurrence relations.

The functions have these symmetry relations:

$$J_n(-x) = (-1)^n J_n(x), \qquad J_{-n}(x) = (-1)^n J_n(x), \qquad Y_{-n}(x) = (-1)^n Y_n(x).$$

They allow the computation for real arguments to be done for $n \geq 0$ and $x \geq 0$, followed by a negation of the computed result when n is negative and odd.

For $|x| \gg |\nu|$, the two Bessel functions have asymptotic expansions as linear combinations of cosines and sines, like this:

$$\begin{aligned}
\theta &= x - (\nu/2 + 1/4)\pi, \\
J_\nu(x) &\asymp \sqrt{2/(\pi x)}\,(\mathrm{P}(\nu,x)\cos(\theta) - \mathrm{Q}(\nu,x)\sin(\theta)), \\
Y_\nu(x) &\asymp \sqrt{2/(\pi x)}\,(\mathrm{Q}(\nu,x)\cos(\theta) + \mathrm{P}(\nu,x)\sin(\theta)).
\end{aligned}$$

Table 21.2: Approximate roots of ordinary Bessel functions, $J_\nu(r_{\nu,k}) = 0$ and $Y_\nu(s_{\nu,k}) = 0$.

					k				
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	...
$r_{0,k}$	2.405	5.520	8.654	11.792	14.931	18.071	21.212	24.352	...
$r_{1/2,k}$	3.142	6.283	9.425	12.566	15.708	18.850	21.991	25.133	...
$r_{1,k}$	3.832	7.016	10.173	13.324	16.471	19.616	22.760	25.904	...
$r_{3/2,k}$	4.493	7.725	10.904	14.066	17.221	20.371	23.519	26.666	...
$r_{2,k}$	5.136	8.417	11.620	14.796	17.960	21.117	24.270	27.421	...
$r_{10,k}$	14.476	18.433	22.047	25.509	28.887	32.212	35.500	38.762	...
$r_{100,k}$	108.836	115.739	121.575	126.871	131.824	136.536	141.066	145.453	...
$s_{0,k}$	0.894	3.958	7.086	10.222	13.361	16.501	19.641	22.782	...
$s_{1/2,k}$	1.571	4.712	7.854	10.996	14.137	17.279	20.420	23.562	...
$s_{1,k}$	2.197	5.430	8.596	11.749	14.897	18.043	21.188	24.332	...
$s_{3/2,k}$	2.798	6.121	9.318	12.486	15.644	18.796	21.946	25.093	...
$s_{2,k}$	3.384	6.794	10.023	13.210	16.379	19.539	22.694	25.846	...
$s_{10,k}$	12.129	16.522	20.266	23.792	27.207	30.555	33.860	37.134	...
$s_{100,k}$	104.380	112.486	118.745	124.275	129.382	134.206	138.821	143.275	...

The right-hand sides of $J_\nu(x)$ and $Y_\nu(x)$ are product-sum expressions of the form $ab + cd$ that may be subject to subtraction loss. Our `PPROSUM()` function family (see Section 13.24 on page 386) can evaluate them accurately.

$\mathrm{P}(\nu, x)$ and $\mathrm{Q}(\nu, x)$ are auxiliary functions defined by

$$\begin{aligned}
\mu &= 4\nu^2, \\
\mathrm{P}(\nu, x) &\asymp 1 - \frac{(\mu - 1^2)(\mu - 3^2)}{2!\,(8x)^2} + \frac{(\mu - 1^2)(\mu - 3^2)(\mu - 5^2)(\mu - 7^2)}{4!\,(8x)^4} - \cdots, \\
\mathrm{Q}(\nu, x) &\asymp \frac{\mu - 1^2}{8x} - \frac{(\mu - 1^2)(\mu - 3^2)(\mu - 5^2)}{3!\,(8x)^3} + \\
&\quad \frac{(\mu - 1^2)(\mu - 3^2)(\mu - 5^2)(\mu - 7^2)(\mu - 9^2)}{5!\,(8x)^5} - \cdots.
\end{aligned}$$

Those series look complicated, but their terms can easily be generated by recurrence relations:

$$\begin{aligned}
\mathrm{P}(\nu, x) &\asymp p_0 + p_1 + p_2 + \cdots, \\
p_0 &= 1, \\
p_k &= -\frac{(\mu - (4k-3)^2)(\mu - (4k-1)^2)}{2k(2k-1)} \times \frac{1}{(8x)^2} \times p_{k-1}, && k = 1, 2, 3, \ldots, \\
\mathrm{Q}(\nu, x) &\asymp q_0 + q_1 + q_2 + \cdots, \\
q_0 &= \frac{\mu - 1}{8x}, \\
q_k &= -\frac{(\mu - (4k-1)^2)(\mu - (4k+1)^2)}{2k(2k+1)} \times \frac{1}{(8x)^2} \times q_{k-1}, && k = 1, 2, 3, \ldots.
\end{aligned}$$

As we describe in Section 2.9 on page 19, asymptotic expansions are not convergent, but they can be summed as long as term magnitudes decrease. The error in the computed function is then roughly the size of the omitted next term in the sum, and that size then determines the available accuracy. For large order ν, the factor in the formulas for the terms p_k and q_k is roughly $\nu^4/(16k^2x^2)$, so as long as $\nu^2 < x$, convergence to arbitrary machine precision is rapid, with each term contributing at least four additional bits to the precision of the sum.

Several published algorithms for $J_\nu(x)$ and $Y_\nu(x)$ exploit the forms of the asymptotic expansions for $\mathrm{P}(\nu, x)$ and $\mathrm{Q}(\nu, x)$ by using instead polynomial approximations in the variable $t = 1/x^2$. To compute such approximations, we first need closed forms for $\mathrm{P}(\nu, x)$ and $\mathrm{Q}(\nu, x)$. We can find $\mathrm{P}(\nu, x)$ by multiplying the two equations involving the Bessel functions by $\cos(\theta)$ and $\sin(\theta)$, respectively, adding the equations, and simplifying. The other function is

Table 21.3: Trigonometric formulas needed for the asymptotic formulas for the ordinary Bessel functions $J_n(x)$ and $Y_n(x)$, with $\theta = x - (n/2 + 1/4)\pi$, and $n \geq 0$. See the text for accurate computation of the sums and differences in these formulas.

n mod 4	$\cos(\theta)$	$\sin(\theta)$
0	$+\sqrt{\frac{1}{2}}(\cos(x) + \sin(x))$	$-\sqrt{\frac{1}{2}}(\cos(x) - \sin(x))$
1	$-\sqrt{\frac{1}{2}}(\cos(x) - \sin(x))$	$-\sqrt{\frac{1}{2}}(\cos(x) + \sin(x))$
2	$-\sqrt{\frac{1}{2}}(\cos(x) + \sin(x))$	$+\sqrt{\frac{1}{2}}(\cos(x) - \sin(x))$
3	$+\sqrt{\frac{1}{2}}(\cos(x) - \sin(x))$	$+\sqrt{\frac{1}{2}}(\cos(x) + \sin(x))$

found by swapping the trigonometric multipliers, and subtracting. We then have these results:

$$\begin{aligned}
\mathrm{P}(\nu, x) &= \sqrt{\pi x/2}\,(J_\nu(x)\cos(\theta) + Y_\nu(x)\sin(\theta)), \\
\mathrm{Q}(\nu, x) &= \sqrt{\pi x/2}\,(Y_\nu(x)\cos(\theta) - J_\nu(x)\sin(\theta)).
\end{aligned}$$

Once the replacements for the asymptotic series have been evaluated, along with the two trigonometric functions, both Bessel functions can be produced with little additional cost, and some software packages do just that. Subtraction loss in the cosines and sines of shifted arguments can be prevented by using the double-angle formula for the cosine, and then solving for the problematic sums and differences:

$$\begin{aligned}
\cos(2\theta) &= (\cos(\theta))^2 - (\sin(\theta))^2 \\
&= (\cos(\theta) - \sin(\theta)) \times (\cos(\theta) + \sin(\theta)), \\
\cos(\theta) - \sin(\theta) &= \cos(2\theta)/(\cos(\theta) + \sin(\theta)), \\
\cos(\theta) + \sin(\theta) &= \cos(2\theta)/(\cos(\theta) - \sin(\theta)).
\end{aligned}$$

When $\nu = 0$, we require these functions:

$$\begin{aligned}
\cos(\theta) &= \cos(x - (1/4)\pi) \\
&= \sqrt{\tfrac{1}{2}}\,(\cos(x) + \sin(x)) \\
&= \sqrt{\tfrac{1}{2}}\,\frac{\cos(2x)}{\cos(x) - \sin(x)}, \\
\sin(\theta) &= \sin(x - (1/4)\pi) \\
&= -\sqrt{\tfrac{1}{2}}\,(\cos(x) - \sin(x)) \\
&= -\sqrt{\tfrac{1}{2}}\,\frac{\cos(2x)}{\cos(x) + \sin(x)}.
\end{aligned}$$

For each of them, we use whichever of the second or third formulas that does not involve a subtraction.

When $\nu = 1$, we get a reduction to the two cases just treated:

$$\begin{aligned}
\cos(\theta) &= \cos(x - (3/4)\pi) \\
&= \sin(x - (1/4)\pi), \\
\sin(\theta) &= \sin(x - (3/4)\pi) \\
&= -\cos(x - (1/4)\pi).
\end{aligned}$$

For the general case of nonnegative integer orders, there are just four possibilities, summarized in Table 21.3.

Although our formulas for trigonometric sums and differences may require $\cos(2x)$, we do not normally compute it by invoking the cosine function directly, for these reasons:

- The argument $2x$ overflows when x is near the overflow limit.

- The cosine computation is comparatively expensive when x is large.
- The argument $2x$ may be inexact for floating-point bases other than two, and if x is large, the computed $\cos(2x)$ would then be wildly incorrect.

Instead, we use well-known trigonometric relations to compute it accurately from quantities that we already have:

$$\cos(2x) = \begin{cases} 2(\cos(x))^2 - 1, & \textit{if } |\cos(x)| \leq 1/2, \\ 1 - 2(\sin(x))^2, & \textit{if } |\sin(x)| \leq 1/2, \\ (1 - \tan(x)^2)/(1 + \tan(x)^2), & \textit{if } \tan(x)^2 \textit{ is outside } (\frac{1}{2}, \frac{3}{2}), \\ -2\,\mathrm{fma}(\sin(x), \sin(x), -\frac{1}{2}), & \textit{if } |\sin(x)| \leq |\cos(x)|, \\ 2\,\mathrm{fma}(\cos(x), \cos(x), -\frac{1}{2}), & \textit{otherwise.} \end{cases}$$

The first two cases each cover about a third of the period of the cosine. The third case computes $\tan(x) = \sin(x)/\cos(x)$ instead of invoking the tangent function, and covers about a sixth of the period. Accuracy loss happens in the remaining sixth of the period, but it can be reduced by use of the fused multiply-add function call. However, if $2x$ is finite and exact, as it is when $\beta = 2$, and sometimes is when $\beta \neq 2$, then we call the cosine function in preference to the fused multiply-add function. There is a tradeoff between the last two alternatives, and in practice, we also call the cosine function when $2x$ is not exact, but x is less than some cutoff, set at 200 in our code.

When the host arithmetic has wobbling precision, more care is needed in the application of the tangent formula. We use it only when $\tan(x)^2 < 1/2$, and compute it as $\frac{1}{2}(1 - \tan(x)^2)/(\frac{1}{2} + \frac{1}{2}\tan(x)^2)$, to avoid leading zero bits in the numerator and denominator. The value of $\tan(x)$ lies in $[\sqrt{\frac{1}{3}}, \sqrt{\frac{1}{2}}] \approx [0.577, 0.707]$, so neither it, nor the sine and cosine from which it is computed, has leading zero bits. Were we to use the tangent formula when $\tan(x)^2 > 3/2$, numerical experiments show that leading zero bits occur about 10% of the time.

The code for the computation of $\cos(2x)$ is needed in a half-dozen Bessel-function files, so it is defined as a private function in the header file `cosdbl.h`.

The functions of the second kind have complex values for negative arguments, and the POSIX specification permits implementations to return either $-\infty$, or the negative of the largest normal number if Infinity is not supported, or else a NaN. The global value `errno` is then set to `EDOM`.

The derivatives of the Bessel functions are:

$$dJ_n(x)/dx = nJ_n(x)/x - J_{n+1}(x),$$
$$dY_n(x)/dx = nY_n(x)/x - Y_{n+1}(x).$$

From the derivatives, we find these error-magnification factors (see **Section 4.1** on page 61):

$$\begin{aligned} \mathrm{errmag}(J_n(x)) &= xJ_n'(x)/J_n(x) \\ &= n - xJ_{n+1}(x)/J_n(x), \\ \mathrm{errmag}(Y_n(x)) &= n - xY_{n+1}(x)/Y_n(x). \end{aligned}$$

The error magnifications are therefore large near the zeros of the Bessel functions, and also when n or x is large.

Three-term recurrence relations relate functions of consecutive orders with fixed argument z:

$$J_{\nu+1}(z) = (2\nu/z)J_\nu(z) - J_{\nu-1}(z),$$
$$Y_{\nu+1}(z) = (2\nu/z)Y_\nu(z) - Y_{\nu-1}(z).$$

When $z \gg \nu$, the first term on the right is negligible, and we have $J_{\nu+1}(z) \approx -J_{\nu-1}(z)$. A sequence of those functions for $\nu = 0, 1, 2, \ldots$ then looks like $J_0(x)$, $J_1(x)$, $-J_0(x)$, $-J_1(x)$, $J_0(x)$, $J_1(x)$, ... The same observation applies to $Y_{\nu+1}(z)$ and sequences of the functions of the second kind.

Unfortunately, the recurrence relations are frequently unstable because of subtraction loss, especially for $|x| < \nu$, as illustrated in the graphs of the ratios of the terms on the right shown in **Figure 21.2** on the next page. Whenever those ratios are in $[\frac{1}{2}, 2]$, the terms have the same sign, and similar magnitudes, and the subtraction loses one or more leading bits in binary arithmetic. That happens near the zeros of the functions on the left-hand side. Whether the accuracy loss affects values of higher-order functions depends on the relative magnitudes of the terms on the right in the next iteration. In addition, when the first terms on the right dominate, errors in the computed $J_\nu(z)$ and $Y_\nu(z)$

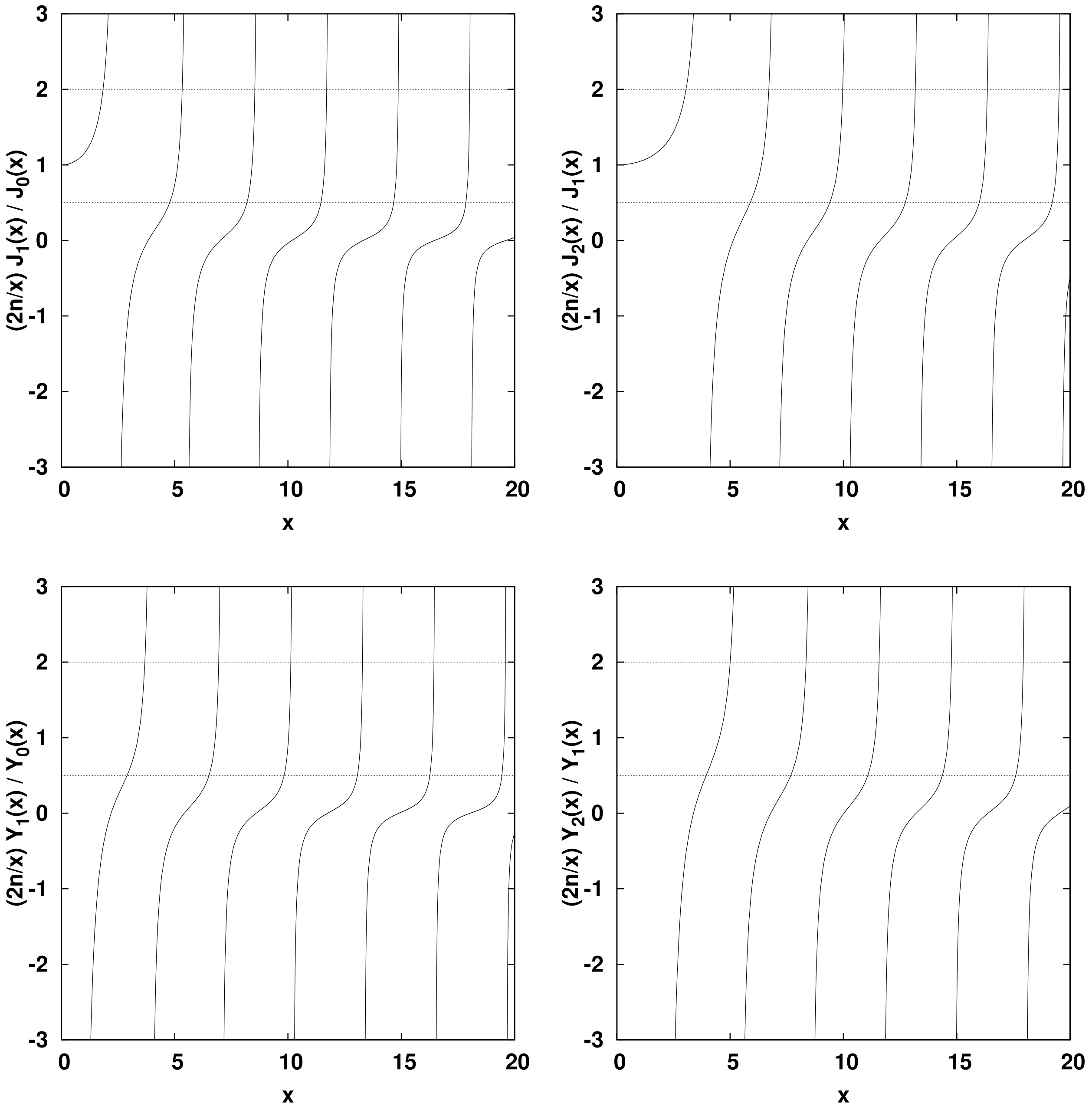


Figure 21.2: Ratios of ordinary Bessel functions of the first and second kinds, $(2n/x)J_n(x)/J_{n-1}(x)$ and $(2n/x)Y_n(x)/Y_{n-1}(x)$, for $n = 1$ and $n = 2$. As n increases, the graphs are qualitatively similar, but shifted to the right.
The recurrence formulas suffer subtraction loss in the region between the horizontal dotted lines.

are magnified by $2\nu/z$, which is large for $\nu \gg |z|$. Under those conditions, just a few iterations of the recurrence relations can easily produce results with *no significant digits* whatever. We show numerical examples of that problem in **Section 21.4** on page 705.

There is a continued fraction (see **Section 2.7** on page 12) for the ratio of two successive orders of the ordinary

Bessel function of the first kind:

$$\begin{aligned}\frac{J_\nu(z)}{J_{\nu-1}(z)} &= 0 + \frac{1}{2\nu/z -}\ \frac{1}{2(\nu+1)/z -}\ \frac{1}{2(\nu+2)/z -}\ \frac{1}{2(\nu+3)/z -}\ \cdots \\ &= 0 + \frac{(1/2)z/\nu}{1 -}\ \frac{(1/4)z^2/\nu(\nu+1)}{1 -}\ \frac{(1/4)z^2/((\nu+1)(\nu+2))}{1 -} \\ &\quad \frac{(1/4)z^2/((\nu+2)(\nu+3))}{1 -}\ \frac{(1/4)z^2/((\nu+3)(\nu+4))}{1 -}\ \cdots .\end{aligned}$$

We show later in **Section 21.6** on page 710 how that continued fraction can be used to find $J_\nu(z)$.

The ordinary Bessel functions of the first and second kinds have these relations:

$$\begin{aligned}\sin(\nu\pi)Y_\nu(z) &= \cos(\nu\pi)J_\nu(z) - J_{-\nu}(z), \\ -2\sin(\nu\pi) &= (\pi z)\big(J_{\nu+1}(z)J_{-\nu}(z) + J_\nu(z)J_{-(\nu+1)}(z)\big), \\ 2 &= (\pi z)\big(J_{\nu+1}(z)Y_\nu(z) - J_\nu(z)Y_{\nu+1}(z)\big).\end{aligned}$$

At least for small ν, those equations can be useful for testing software implementations of the functions, as long as ν and z are chosen so that there is no subtraction loss on the right-hand sides.

The Bessel functions can also be expressed as integrals, although the oscillatory nature of the integrand for large n and/or large z makes it difficult to evaluate them accurately by numerical quadrature:

$$\begin{aligned}J_0(z) &= (1/\pi)\int_0^\pi \cos(z\sin(t))\,dt, \\ &= (1/\pi)\int_0^\pi \cos(z\cos(t))\,dt, \\ J_n(z) &= (1/\pi)\int_0^\pi \cos(z\sin(t) - nt)\,dt, \\ Y_0(z) &= (4/\pi^2)\int_0^{\pi/2} \cos(z\cos(t))\big(\gamma + \log(2z(\sin(t))^2)\big)\,dt.\end{aligned}$$

Here, $\gamma \approx 0.577\cdots$ is the Euler–Mascheroni constant.

For x in $[0,\pi/2]$, the integrand for $J_0(x)$ is smooth and positive, so numerical quadrature could be attractive, and accurate. For example, a 28-point Gauss–Chebyshev quadrature recovers $J_0(\pi/4)$ to within 2 ulps of the correct value in 64-bit IEEE 754 arithmetic, a 32-point Gauss–Legendre quadrature is correct to 30 decimal digits, and a 24-point Simpson's rule quadrature produces 34 correct decimal digits. That too could be useful for software checks.

The summation formula

$$J_\nu(z) = (z/2)^\nu \sum_{k=0}^{\infty} \frac{(-(z^2/4))^k}{k!\,\Gamma(\nu+k+1)}$$

converges rapidly for $|z| < 1$, and the terms fall off sufficiently fast that there is no loss of leading digits in the subtractions of successive terms. Even with larger values of $|z|$, convergence is still reasonable, as shown in **Table 21.4** on the next page. However, for $|z| > 2$ and small n, the first few terms grow, and are larger than $J_n(z)$, so there is necessarily loss of leading digits during the summation. Higher intermediate precision, when it is available, can sometimes hide that loss.

When ν is an integer, the gamma function in the denominator reduces to a factorial, and we then have simpler summations suitable for the Bessel functions of the first kind required by POSIX:

$$\begin{aligned}J_0(z) &= 1 - (z^2/4)/(1!)^2 + (z^2/4)^2/(2!)^2 - (z^2/4)^3/(3!)^2 + \cdots, \\ J_1(z) &= (z/2)(1 - (z^2/4)/(1!\,2!) + (z^2/4)^2/(2!\,3!) - (z^2/4)^3/(3!\,4!) + \cdots), \\ J_n(z) &= (z/2)^n \sum_{k=0}^{\infty} \frac{(-(z^2/4))^k}{k!\,(k+n)!}.\end{aligned}$$

It is convenient to introduce two intermediate variables to simplify the sum for $J_n(z)$:

$$v = z/2, \qquad w = v^2, \qquad J_n(z) = v^n \sum_{k=0}^{\infty} \frac{(-w)^k}{k!\,(k+n)!}.$$

Table 21.4: Series term counts needed to reach a given accuracy for $J_n(x)$. The digit counts correspond to those of extended IEEE 754 decimal floating-point arithmetic, and are close to those of extended IEEE 754 binary arithmetic.

$x = 0.1$	Decimal digits			
n	7	16	34	70
0	3	5	9	17
1	2	5	9	16
5	2	4	8	16
10	2	4	8	15
100	2	4	7	13
1000	2	3	6	11

$x = 1$	Decimal digits			
n	7	16	34	70
0	5	9	15	27
1	5	9	15	26
5	4	8	14	25
10	4	7	13	24
100	3	5	11	20
1000	2	4	8	16

$x = 5$	Decimal digits			
n	7	16	34	70
0	11	17	27	43
1	10	16	26	42
5	9	15	24	40
10	7	13	23	38
100	4	9	16	30
1000	3	6	12	22

For example, for $n = 0$ or $n = 1$, and x in $[0, 2]$, the number of terms required in the four extended IEEE 754 decimal formats is 8, 14, 25, and 45. For fixed n and a chosen interval of x, the sum could also be replaced by a Chebyshev polynomial economization.

We can factor the sum so that the terms t_k can be computed with a simple recurrence relation:

$$J_n(z) = \frac{z^n}{2^n n!}(t_0 + t_1 + t_2 + \cdots) = \frac{v^n}{n!}(t_0 + t_1 + t_2 + \cdots),$$

$$t_0 = 1, \qquad t_k = \frac{-w}{k(k+n)} t_{k-1}, \qquad \text{for } k = 1, 2, 3, \ldots.$$

In the scale factor $v^n/n!$, the numerator can overflow or underflow, and the denominator can overflow, even though the scale factor may be representable if it were computed in exact arithmetic. In those cases, it must be computed with a logarithm and an exponential.

For integer orders only, there is a corresponding, but complicated, sum for $Y_n(z)$:

$$\begin{aligned} Y_n(z) = -\frac{1}{\pi} \Bigg[& v^{-n} \sum_{k=0}^{n-1} \frac{(n-k-1)!}{k!} w^k - 2\log(v) J_n(z) \\ & + v^n \sum_{k=0}^{\infty} \big(\psi(k+1) + \psi(k+n+1)\big) \frac{(-w)^k}{k!\,(k+n)!} \Bigg]. \end{aligned}$$

That sum looks unsuitable for fast computation because of the presence of the psi functions (see **Section 18.2** on page 536). Fortunately, their arguments are integers, for which they have simple forms in terms of the partial sums, h_k, of the *harmonic series* of reciprocal integers, and γ, the Euler–Mascheroni constant:

$$h_0 = 0, \qquad h_k = 1 + 1/2 + 1/3 + \cdots + 1/k, \qquad k > 0,$$

$$\psi(1) = -\gamma, \qquad \psi(k) = -\gamma + \sum_{m=1}^{k-1} \frac{1}{m} = h_{k-1} - \gamma, \qquad k > 1.$$

Substitute those values into the formula for $Y_n(z)$ to get

$$\begin{aligned} Y_n(z) = -\frac{1}{\pi} \Bigg[& v^{-n} \sum_{k=0}^{n-1} \frac{(n-k-1)!}{k!} w^k - 2\log(v) J_n(z) \\ & + v^n \sum_{k=0}^{\infty} (h_k + h_{k+n} - 2\gamma) \frac{(-w)^k}{k!\,(k+n)!} \Bigg]. \end{aligned}$$

A computationally satisfactory formula for the ordinary Bessel function of the second kind results from substitution of the expansion of $J_n(z)$ in that result:

$$\begin{aligned} Y_n(z) = -\frac{1}{\pi}\Bigg[& v^{-n}\sum_{k=0}^{n-1}\frac{(n-k-1)!}{k!}w^k - 2(\log(v)+\gamma)J_n(z) \\ & +v^n\sum_{k=0}^{\infty}(h_k+h_{k+n})\frac{(-w)^k}{k!\,(k+n)!}\Bigg].\end{aligned}$$

From the general case, we can now easily find these formulas for the functions of orders zero and one:

$$\begin{aligned} Y_0(z) &= -\frac{2}{\pi}\left[-(\log(v)+\gamma)J_0(z)+\sum_{k=1}^{\infty}h_k\frac{(-w)^k}{(k!)^2}\right], \\ Y_1(z) &= -\frac{1}{\pi}\left[\frac{1}{v}-2(\log(v)+\gamma)J_1(z)+v\sum_{k=0}^{\infty}(h_k+h_{k+1})\frac{(-w)^k}{k!\,(k+1)!}\right].\end{aligned}$$

The series for $Y_0(z)$ requires about the same number of terms as that for $J_0(z)$. Convergence of the series for $Y_1(z)$ is faster than that for $Y_0(z)$, because corresponding terms are smaller by a factor of $1/(k+1)$.

The presence of the harmonic partial sums, h_k, suggests three possible computational approaches:

- Use small precomputed tables of h_k and $k!$ to compute the series sums for small z.
- Precompute the complete coefficients of w^k. Numerical experiments for x in $[0,2]$ show that 8, 12, 20, and 32 terms suffice for computing the infinite sum in $Y_1(z)$ in the four extended IEEE 754 decimal precisions.
- Replace the infinite sums by Chebyshev economizations; they need 6, 10, 16, and 27 terms for x in $[0,2]$ for those four decimal precisions.

With another intermediate variable, and expansions of $J_0(z)$ and $J_1(z)$, we can further simplify the formulas for $Y_0(z)$ and $Y_1(z)$ to obtain fast formulas suitable for tiny arguments:

$$\begin{aligned} s &= \log(v)+\gamma, \\ Y_0(z) &= -\frac{2}{\pi}(-s-(-s+1)w-(1/8)(2s-3)w^2-(1/216)(-6s+11)w^3- \\ &\qquad \cdots), \\ Y_1(z) &= -\frac{1}{\pi v}(1+(-2s+1)w+(s-5/4)w^2+(1/18)(-3s+5)w^3+\cdots).\end{aligned}$$

In the term $\log(v)+\gamma$, notice that when $z<2$, the logarithm is negative, so there is a subtraction from γ that can lose leading bits. There is complete loss near $z=2\exp(-\gamma)\approx 1.123$. The solution is to rewrite the problem expression like this:

$$\begin{aligned} \log(v)+\gamma &= \log(v)+\log(\exp(\gamma)) \\ &= \log(v\exp(\gamma)) \\ &= \log(v\times 1.781\,072\,417\,990\,197\,985\,236\,504\,103\,107\,179\cdots) \\ &= \log(z\times 0.890\,536\,208\,995\,098\,992\,618\,252\,051\,553\,589\cdots).\end{aligned}$$

The last form is preferred, because in hexadecimal arithmetic, it preserves maximal accuracy in the constant.

Unfortunately, the series for $Y_0(z)$ has two other computational problems:

- When the argument of the logarithm is near one, which happens for $z\approx 1.123$, the logarithm is small, and loses accuracy. One solution is to replace it with the logarithm-plus-one function, log1p(), and compute the argument accurately with the help of a fused multiply-add operation:

$$\begin{aligned} v\exp(\gamma) &= 1+d, \\ d &= v\exp(\gamma)-1 \\ &= \mathrm{fma}(v,\exp(\gamma),-1), \\ \log(v\exp(\gamma)) &= \mathrm{log1p}(d).\end{aligned}$$

We can improve the accuracy of the argument d by splitting the exponential constant into exact high and approximate low parts:

$$\begin{aligned}\exp(\gamma) &= c_{\mathrm{hi}} + c_{\mathrm{lo}},\\ d &= \mathrm{fma}(v, c_{\mathrm{hi}}, -1) + v c_{\mathrm{lo}}\\ &= \mathrm{fma}(v, c_{\mathrm{lo}}, \mathrm{fma}(v, c_{\mathrm{hi}}, -1)).\end{aligned}$$

- The summation contains terms of alternating signs, and numerical experiments show that the ratio of the sums of positive and negative terms exceeds $\frac{1}{2}$ when $2.406 \leq x$, implying loss of leading digits. For $x = 5$, two decimal digits are lost, for $x = 10$, four are lost, and for $x = 20$, eight are lost.

Higher precision, when available, can hide the summation loss if x is not too big, but a better approach is to replace the sum, or the entire expansion of $Y_0(x)$, with a polynomial fit. Several published algorithms for that function do just that, and to limit the polynomial degrees, the range of x where the sum is used is split into several regions, each with its own polynomial fit.

21.4 Experiments with recurrences for $J_0(x)$

The epigraph that begins this chapter comes from an often-cited paper on three-term recurrence relations [Gau67], and its author shows a numeric example of upward recurrence from accurate starting values of $J_0(1)$ and $J_1(1)$. Here is that experiment done in 32-bit decimal arithmetic in hoc, using initial values correctly rounded to seven digits from a high-precision computation in Maple, and augmented with comments showing correct results:

```
hocd32> x = 1
hocd32> J_km1 = 0.7651977
hocd32> J_k   = 0.4400506
hocd32> printf("%2d %.6e\n", 0, J_km1)
hocd32> for (k = 1; k <= 20; ++k) \
hocd32> {
hocd32>     J_kp1 = (2 * k / x) * J_k - J_km1
hocd32>     printf("%2d %.6e\n", k, J_k)
hocd32>     J_km1 = J_k
hocd32>     J_k = J_kp1
hocd32> }
 0 7.651977e-01                # expect  0   7.651977e-01
 1 4.400506e-01                # expect  1   4.400506e-01
 2 1.149035e-01                # expect  2   1.149035e-01
 3 1.956340e-02                # expect  3   1.956335e-02
 4 2.476900e-03                # expect  4   2.476639e-03
 5 2.518000e-04                # expect  5   2.497577e-04
 6 4.110000e-05                # expect  6   2.093834e-05
 7 2.414000e-04                # expect  7   1.502326e-06
 8 3.338500e-03                # expect  8   9.422344e-08
 9 5.317460e-02                # expect  9   5.249250e-09
...
15 7.259898e+06                # expect 15   2.297532e-17
16 2.175373e+08                # expect 16   7.186397e-19
17 6.953934e+09                # expect 17   2.115376e-20
18 2.362163e+11                # expect 18   5.880345e-22
19 8.496833e+12                # expect 19   1.548478e-23
20 3.226435e+14                # expect 20   3.873503e-25
```

Almost half the digits are incorrect at $k = 4$, all are incorrect at $k = 6$, and instead of decreasing towards zero, the values computed with seven-digit arithmetic begin to grow at $k = 7$.

Increasing precision helps only marginally. In 64-bit decimal arithmetic, with 16 digits of precision, half the digits are wrong at $k = 6$, all are wrong at $k = 9$, and values increase at $k = 11$.

In 128-bit decimal arithmetic, with 34 digits of precision, half the digits are in error at $k = 11$, all are erroneous at $k = 16$, and the increase begins at $k = 17$.

Similar experiments with $x = 1/10$ show that the generated values are completely wrong at $k = 4$, and with $x = 1/1000$, at $k = 2$. Clearly, forward recurrence for $J_n(x)$ is extremely unstable when $x \ll n$.

We now repeat the experiment in seven-digit arithmetic using *backward* recurrence:

```
hocd32> x = 1
hocd32> J_kp1 = 9.227622e-27       # J(21,x)
hocd32> J_k   = 3.873503e-25       # J(20,x)
hocd32> printf("%2d %.6e\n", 20, J_k); \
hocd32> for (k = 20; k > 0; --k)       \
hocd32> {
hocd32>     J_km1 = (2*k/x)*J_k - J_kp1
hocd32>     printf("%2d %.6e\n", k - 1, J_km1)
hocd32>     J_kp1 = J_k
hocd32>     J_k = J_km1
hocd32> }
20 3.873503e-25                # expect 20   3.873503e-25
19 1.548478e-23                # expect 19   1.548478e-23
18 5.880342e-22                # expect 18   5.880345e-22
17 2.115375e-20                # expect 17   2.115376e-20
16 7.186395e-19                # expect 16   7.186397e-19
15 2.297531e-17                # expect 15   2.297532e-17
...
 9 5.249246e-09                # expect  9   5.249250e-09
 8 9.422337e-08                # expect  8   9.422344e-08
 7 1.502325e-06                # expect  7   1.502326e-06
 6 2.093833e-05                # expect  6   2.093834e-05
 5 2.497577e-04                # expect  5   2.497577e-04
 4 2.476639e-03                # expect  4   2.476639e-03
 3 1.956335e-02                # expect  3   1.956335e-02
 2 1.149035e-01                # expect  2   1.149035e-01
 1 4.400506e-01                # expect  1   4.400506e-01
 0 7.651977e-01                # expect  0   7.651977e-01
```

Although there are differences in final digits in two-thirds of the results, the largest relative error is just 0.83 ulps, at $k = 11$. The final six results are correctly rounded representations of the exact values. Evidently, backward recurrence is stable for that computation.

For $|x| \gg n$, the stability problem in the upward recurrence largely disappears. Even for $x = 20$ in our seven-digit-arithmetic example, the computed values, not shown here, have a maximum absolute error of 4×10^{-7}. Because the starting values for upward recurrence are simpler than for downward recurrence, we exploit that fact later in code for sequences of Bessel functions of a fixed argument and increasing orders.

Interval arithmetic provides another way to illustrate the instability of upward recurrence for $J_n(x)$ when $x \ll n$, and the return of stability when $x \gg n$. Here is the output of an interval version of the recurrence implemented in the 32-bit binary version of hoc, where the arguments of the test function are the maximum order and the interval bounds for x. The starting values of $J_0(x)$ and $J_1(x)$ are determined from higher-precision computation, then converted to interval form with a half-ulp halfwidth. Each line shows the order n, the lower and upper bounds on the function value, the interval midpoint and its halfwidth, and the expected value determined by computing it in higher precision, and then casting it to working precision:

```
hoc32> load("ijn")    # code for interval version of UPWARD recurrence for Jn(n,x)
hoc32> test_Jn_up(20, 1, 1)
 0  [ 7.651_976e-01,  7.651_978e-01] =  7.651_977e-01 +/- 5.960_464e-08  # expect  7.651_977e-01
 1  [ 4.400_505e-01,  4.400_506e-01] =  4.400_506e-01 +/- 2.980_232e-08  # expect  4.400_506e-01
 2  [ 1.149_033e-01,  1.149_036e-01] =  1.149_035e-01 +/- 1.192_093e-07  # expect  1.149_035e-01
 3  [ 1.956_272e-02,  1.956_373e-02] =  1.956_323e-02 +/- 5.066_395e-07  # expect  1.956_335e-02
 4  [ 2.472_758e-03,  2.479_076e-03] =  2.475_917e-03 +/- 3.159_046e-06  # expect  2.476_639e-03
 5  [ 2.183_318e-04,  2.698_898e-04] =  2.441_108e-04 +/- 2.577_901e-05  # expect  2.497_577e-04
```

```
  6  [-2.957_582e-04,  2.261_400e-04] = -3.480_911e-05 +/- 2.609_491e-04  # expect  2.093_834e-05
 ...
 15  [-1.202_778e+08,  8.013_022e+07] = -2.007_379e+07 +/- 1.002_040e+08  # expect  2.297_532e-17
 16  [-3.611_190e+09,  2.408_198e+09] = -6.014_958e+08 +/- 3.009_694e+09  # expect  7.186_397e-19
 17  [-1.156_382e+11,  7.718_263e+10] = -1.922_779e+10 +/- 9.641_042e+10  # expect  2.115_376e-20
 18  [-3.934_108e+12,  2.627_821e+12] = -6.531_434e+11 +/- 3.280_964e+12  # expect  5.880_344e-22
 19  [-1.417_051e+14,  9.471_720e+13] = -2.349_394e+13 +/- 1.182_111e+14  # expect  1.548_478e-23
 20  [-5.387_421e+15,  3.603_188e+15] = -8.921_166e+14 +/- 4.495_304e+15  # expect  3.873_503e-25
```

Notice that already at $n = 6$, the interval includes zero, and the interval halfwidth is larger than its midpoint. The rapidly growing interval width gives one little confidence in the function values estimated from the interval midpoints.

We then repeat the experiment with $x = 20$, and see that, although the interval widths grow, they remain small compared to the midpoints, and the midpoints are close to the expected values:

```
hoc32> test_Jn_up(20, 20, 20)
  0  [ 1.670_246e-01,  1.670_247e-01] =  1.670_247e-01 +/- 1.490_116e-08  # expect  1.670_247e-01
  1  [ 6.683_312e-02,  6.683_313e-02] =  6.683_312e-02 +/- 7.450_581e-09  # expect  6.683_312e-02
  2  [-1.603_414e-01, -1.603_413e-01] = -1.603_414e-01 +/- 2.235_174e-08  # expect -1.603_414e-01
  3  [-9.890_141e-02, -9.890_138e-02] = -9.890_139e-02 +/- 1.490_116e-08  # expect -9.890_139e-02
  4  [ 1.306_709e-01,  1.306_710e-01] =  1.306_709e-01 +/- 3.725_290e-08  # expect  1.306_709e-01
  5  [ 1.511_697e-01,  1.511_698e-01] =  1.511_697e-01 +/- 3.725_290e-08  # expect  1.511_698e-01
  6  [-5.508_611e-02, -5.508_599e-02] = -5.508_605e-02 +/- 5.960_464e-08  # expect -5.508_605e-02
 ...
 15  [-8.174_032e-04, -8.067_638e-04] = -8.120_835e-04 +/- 5.319_715e-06  # expect -8.120_690e-04
 16  [ 1.451_691e-01,  1.451_905e-01] =  1.451_798e-01 +/- 1.072_884e-05  # expect  1.451_798e-01
 17  [ 2.330_773e-01,  2.331_223e-01] =  2.330_998e-01 +/- 2.249_330e-05  # expect  2.330_998e-01
 18  [ 2.510_408e-01,  2.511_387e-01] =  2.510_898e-01 +/- 4.899_502e-05  # expect  2.510_898e-01
 19  [ 2.187_510e-01,  2.189_724e-01] =  2.188_617e-01 +/- 1.106_933e-04  # expect  2.188_619e-01
 20  [ 1.644_882e-01,  1.650_068e-01] =  1.647_475e-01 +/- 2.593_249e-04  # expect  1.647_478e-01
```

21.5 Computing $J_0(x)$ and $J_1(x)$

For tiny x, summing the first few terms of the Taylor series of $J_n(x)$ provides a correctly rounded result. The mathcw library code uses four-term series for the Bessel functions $J_0(x)$ and $J_1(x)$.

For x in $[\text{tiny}, 2]$, the series can be summed to machine precision, or $J_0(x)$ and $J_1(x)$ can be represented by polynomial approximations. The mathcw library code for those two functions implements both methods, and for each function, uses a single Chebyshev polynomial table (see **Section 3.9** on page 43) that is truncated at compile time to the accuracy needed for the current working precision, avoiding the need for separate minimax polynomials for each precision. For those functions, and the intervals treated in this section, minimax fits are only slightly more accurate than Chebyshev fits of the same total degree.

The polynomial for $J_0(x)$ is chosen to fit the remainder of the Taylor series after the first two terms, and because of the symmetry relation, we assume that x is nonnegative:

$$\begin{aligned}
t &= x^2, && \textit{t in } [0,4], \textit{ x in } [0,2],\\
u &= t/2 - 1, && \textit{Chebyshev variable u in } [-1,+1],\\
f(t) &= (J_0(\sqrt{t}) - (1 - t/4))/t^2, &&\\
&= \sum_{k=0}^{N} c_k T_k(u), && \textit{Chebyshev polynomial fit}\\
J_0(x) &= 1 - t/4 + t^2 f(t) &&\\
&= 1 - x^2/4 + t^2 f(t) &&\\
&= \operatorname{fma}(-x/2, x/2, 1) + t^2 f(t) &&\\
&= (1 - x/2)(1 + x/2) + t^2 f(t), && \textit{use when } \beta \neq 16,
\end{aligned}$$

$$= 2((1 - x/2)(1/2 + x/4)) + t^2 f(t), \qquad \textit{use when } \beta = 16.$$

When $t \approx 4$, the leading sum $1 - t/4$ suffers serious loss of leading digits, unless it is computed with a fused multiply-add operation. However, rewriting it in factored form moves the subtraction loss to the term $1 - x/2$, and for $\beta = 2$, that is computed almost exactly, with only a single rounding error. Using higher precision for intermediate computations, but not for the Chebyshev sum, reduces the worst-case errors by about one ulp.

Two preprocessor symbols, `USE_CHEBYSHEV` and `USE_SERIES`, select the algorithm used in `j0x.h` when x is in $[\text{tiny}, 2]$. Error plots show that both methods are equally accurate for small x, but errors are smaller for the Chebyshev fit and higher intermediate precision, so that is the default if neither symbol is defined.

For $J_1(x)$ with x in $[0, 2]$, t and u are as before, and we develop a polynomial fit for nonnegative x like this:

$$\begin{aligned} g(t) &= (2J_1(\sqrt{t})/\sqrt{t} - (1 - t/8))/t^2 \\ &= \sum_{k=0}^{N} d_k T_k(u), && \textit{Chebyshev polynomial fit,} \\ J_1(x) &= (x/2)(1 - t/8 + t^2 g(t)). \end{aligned}$$

Here, $t/8 \leq 1/2$, so there is no subtraction loss in the first two terms, and the terms can be summed safely from right to left. The major sources of error are the two rounding errors from the final subtraction and the final product with $x/2$.

There are single roots $J_0(2.404\cdots) = 0$ and $J_1(3.831\cdots) = 0$ in the interval $[2, 4]$. Straightforward polynomial approximations on that interval are possible, but near the roots, they must sum to a small number, and as a result, have a large relative error. The Cephes library [Mos89, Chapter 6] removes that error by instead using a polynomial approximation in which the roots in the interval are factored out. For our case, we call the respective roots r and s, and we have:

$$\begin{aligned} J_0(x) &= (x^2 - r^2)p(x), & p(x) &= \sum_{k=0}^{N} p_k T_k(u), && \textit{Chebyshev polynomial fit,} \\ J_1(x) &= x(x^2 - s^2)q(x), & q(x) &= \sum_{k=0}^{N} q_k T_k(u), && \textit{Chebyshev polynomial fit.} \end{aligned}$$

The critical computational issues are that we must avoid massive subtraction loss in the factors with the roots, and we must account for the fact that the exact roots are not machine numbers. That is best done by representing each root as a sum of exact high and approximate low parts, and then computing the factor like this:

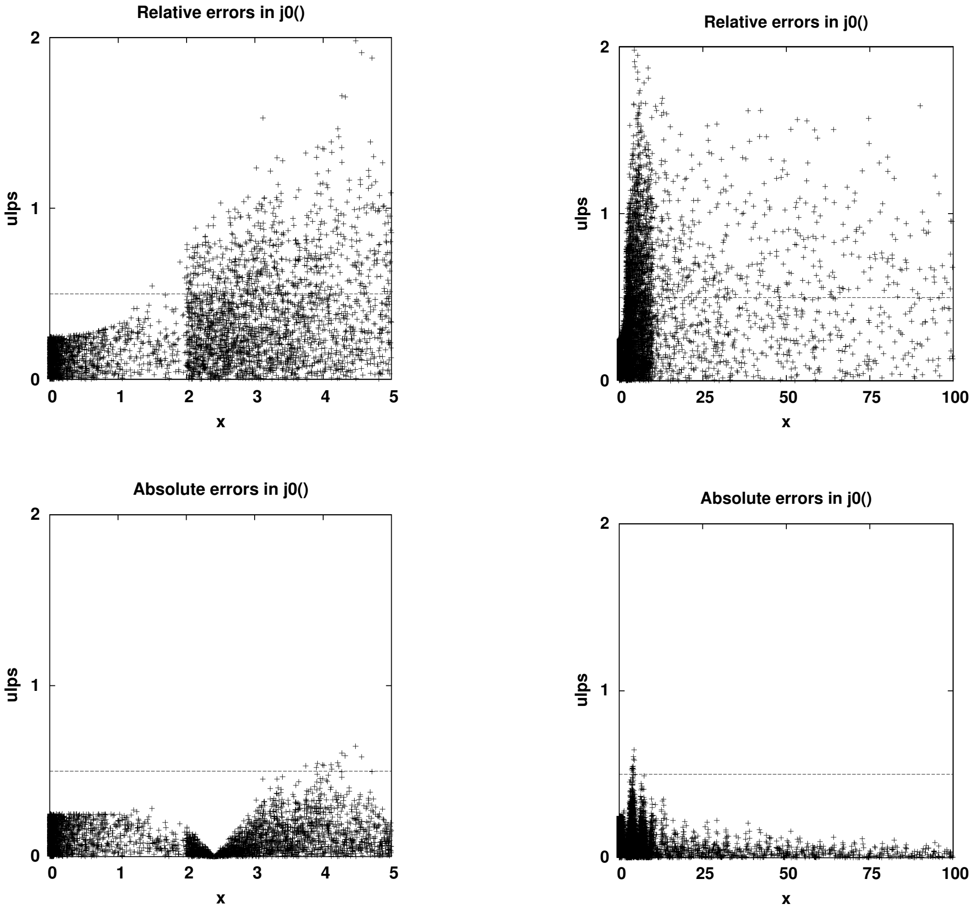
$$\begin{aligned} x^2 - r^2 &= (x - r)(x + r), \\ &= ((x - r_{\mathrm{hi}}) - r_{\mathrm{lo}})((x + r_{\mathrm{hi}}) + r_{\mathrm{lo}}). \end{aligned}$$

When x is near a root, the difference $x - r_{\mathrm{hi}}$ is computed exactly, because both terms have the same floating-point exponent. Subtraction of r_{lo} then provides an important correction to the difference, with only a single rounding error. The second factor is computationally stable because it requires only additions.

We need a different split of each root for each machine precision, and a Maple function in the file `split-base.map` generates the needed C code, with embedded preprocessor conditional statements to select an appropriate set for the current precision. For example, one such pair with its compile-time selector in `j0.h` looks like this for the IEEE 754 and VAX 32-bit formats:

```
#elif (T >= 24) && (B != 16)

static const fp_t J0_R1_HI = FP(10086569.0) / FP(4194304.0);
static const fp_t J0_R1_LO = FP(1.08705905e-07);
```

The one drawback to our factored representation is that the final function value is no longer computed as the sum of an exact value and a small correction. Instead, it has cumulative rounding errors from each of the factors and their products. We therefore expect larger errors on the interval $[2, 4]$ than on $[0, 2]$, but we can nevertheless guarantee a small relative error, instead of a small absolute error.


Relative errors in j0()

Relative errors in j0()

Absolute errors in j0()

Absolute errors in j0()

Figure 21.3: Relative (top) and absolute (bottom) errors in `j0(x)` for two argument ranges. The largest relative errors occur near the zeros of the function that lie in $[4, \infty)$, and can grow arbitrarily large. However, the absolute errors remain small.

For the interval $[4, \infty)$, we use Chebyshev polynomial fits to the functions $\mathrm{P}(0, x)$, $\mathrm{P}(1, x)$, $\mathrm{Q}(0, x)$, and $\mathrm{Q}(1, x)$ (see **Section 21.3** on page 699), and then recover the Bessel functions from the formulas involving those functions and the cosine and sine. There are now cumulative errors from the polynomial evaluations, the trigonometric functions, and their final product sums. We therefore expect higher errors than on the interval $[0, 4]$, and sadly, we can only provide small absolute error near the roots after the first one.

For large x, $\mathrm{P}(\nu, x)$ is $\mathcal{O}(1)$, and dominates $\mathrm{Q}(\nu, x)$, which is $\mathcal{O}(1/x)$. Nevertheless, because the trigonometric multipliers can be small, and of either sign, the two products may be of comparable size, and their sum subject to subtraction loss. Also, for large x, the accuracy of the computed Bessel functions depends critically on that of the trigonometric argument reduction, and it is precisely here that most historical implementations of $J_n(x)$ and $Y_n(x)$ may deliver results where every digit is in error, even if the magnitudes are correct. Thanks to the exact argument reduction used in the `mathcw` library, that is not a problem for us, and as long as x is exactly representable, and not near a Bessel function root, our `j0(x)` and `j1(x)` functions produce results that agree to within a few ulps of high-precision values computed by symbolic-algebra systems, even when x is the largest number representable in the floating-point system.

Figure 21.3 and **Figure 21.4** on the following page show the relative and absolute errors for our implementations in data type `double` of the $J_0(x)$ and $J_1(x)$ functions. Error plots for other binary and decimal precisions are similar, and thus, not shown. The observed errors quantify the rough estimates that we made based on the numerical steps of the computations.

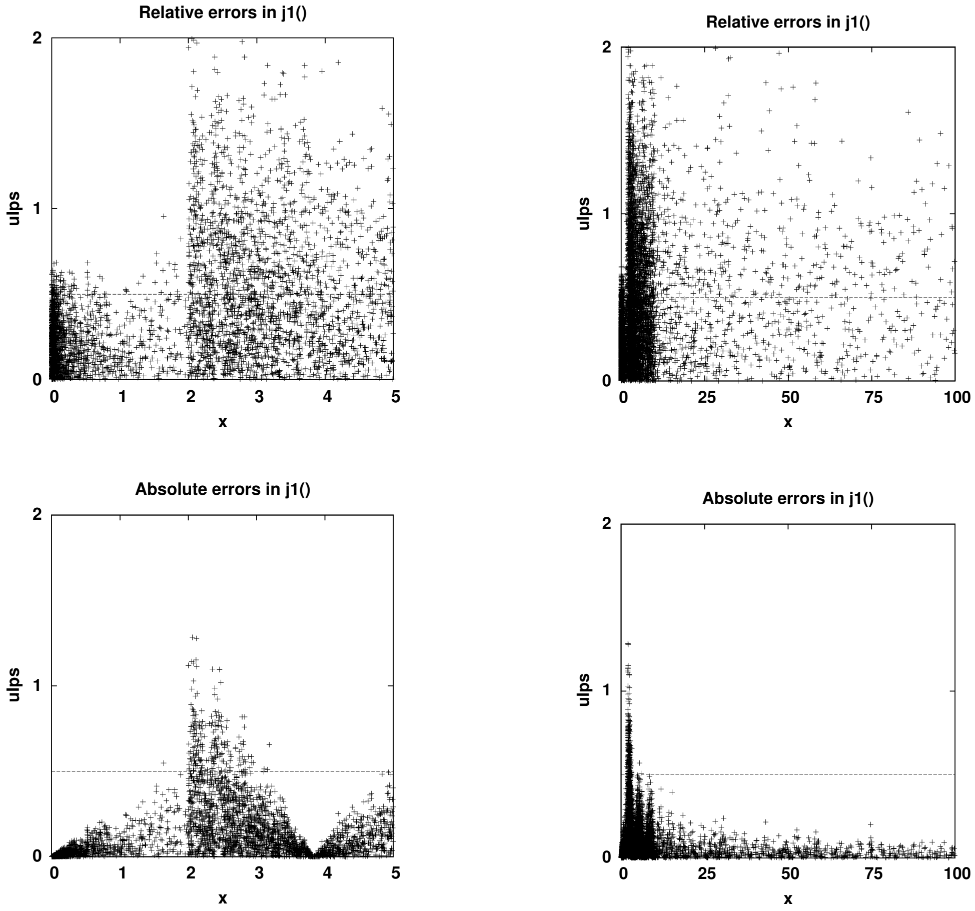


Figure 21.4: Relative (top) and absolute (bottom) errors in `j1(x)` for two argument ranges. The largest relative errors occur near the zeros of the function that lie in $[4, \infty)$, and can grow arbitrarily large. However, the absolute errors remain small.

21.6 Computing $J_n(x)$

We now turn to the problem of computing $J_n(x)$ for $n > 1$. As we observed earlier, when x is small, the series formula converges rapidly, and is the best way to compute the function for arbitrary n.

For larger x values, although upward recurrence for computing $J_n(x)$ is unsuitable, downward recurrence has been found to be stable, but the problem is that we do not have a good way to compute the two starting Bessel functions directly when both n and x are large. The continued fraction presented earlier on page 702 leads to an algorithm for finding those two functions, but the procedure is not immediately obvious, and requires some explanation.

In the notation of **Section 2.7** on page 12, the elements of the continued fraction for $J_\nu(z)/J_{\nu-1}(z)$ are given by

$$\begin{array}{lll} a_1 = +1, & a_k = -1, & k > 1, \\ b_0 = 0, & b_k = 2(\nu + k - 1)/z, & k > 0. \end{array}$$

The numerically preferred backward evaluation of the continued fraction is complicated because we have two parameters that affect the starting value of the iteration. However, either of the Lentz or Steed algorithms allows evaluation in the forward direction with early loop exit as soon as the value of the continued fraction has converged. That gives us the ratio $J_\nu(z)/J_{\nu-1}(z)$, but we still do not know $J_\nu(z)$ itself. To find that value, rewrite the recurrence relation in the downward direction for decreasing *integer* orders, and divide each equation by $J_n(z)$:

$$\begin{aligned}
J_{n-2}(z)/J_n(z) &= (2(n-1)/z)J_{n-1}(z)/J_n(z) - 1,\\
J_{n-3}(z)/J_n(z) &= (2(n-2)/z)J_{n-2}(z)/J_n(z) - J_{n-1}(z)/J_n(z),\\
J_{n-4}(z)/J_n(z) &= (2(n-3)/z)J_{n-3}(z)/J_n(z) - J_{n-2}(z)/J_n(z),\\
\ldots &\quad \ldots\\
J_2(z)/J_n(z) &= (6/z)J_3(z)/J_n(z) - J_4(z)/J_n(z),\\
J_1(z)/J_n(z) &= (4/z)J_2(z)/J_n(z) - J_3(z)/J_n(z),\\
J_0(z)/J_n(z) &= (2/z)J_1(z)/J_n(z) - J_2(z)/J_n(z).
\end{aligned}$$

The right-hand side of the first equation contains known values, so we can easily compute its left-hand side. The right-hand side of the second equation requires two ratios that we now have, so we can find its left-hand side. We need to remember at most two consecutive ratios to repeat the process, and we finally obtain a value for $J_0(z)/J_n(z)$. Calling that result $f_n(z)$, we now compute $J_0(z)$ independently by the methods of **Section 21.5** on page 707, and then recover the desired function value $J_n(z)$ as $J_0(z)/f_n(z)$.

There is an important refinement to be made in that last step. If we are near a zero of $J_0(z)$, then the function value is likely to have a high relative error that propagates into the computed $J_n(z)$. It is then better to use the second-last ratio, and compute $J_n(z)$ from $J_1(z)/(J_1(z)/J_n(z))$. We use that alternative when the magnitude of the ratio $J_1(z)/J_n(z)$ exceeds that of $J_0(z)/J_n(z)$. The root separation of the estimates in **Section 21.3** on page 697 guarantees that successive ratios cannot both be tiny.

Although that procedure may look complicated, the code that implements it is short. Here is a hoc function that computes $J_n(x)$, with checks for special cases of n and x omitted:

```
func Jncf(n,x)                                    \
{   # return J(n,x) via the continued-fraction algorithm
    rinv = cf(n,x)          # J(n,x) / J(n-1,x)
    rk = 1 / rinv           # J(k,x) / J(n,x), for k == n - 1
    rkp1 = 1
    s = 1

    for (k = n - 1; k > 0; --k)                   \
    {
        rkm1 = (2 * k / x) * rk - rkp1
        rkp1 = rk
        rk = rkm1

        if (isinf(rkm1))                          \
        {
            rk *= MINNORMAL
            rkp1 *= MINNORMAL
            s *= MINNORMAL
            rkm1 = (2 * k / x) * rk - rkp1
        }
    }

    if (abs(rkp1) > abs(rk))                      \
        return (s * J1(x) / rkp1)                 \
    else                                          \
        return (s * J0(x) / rk)
}
```

The check for infinity in the downward loop is essential, because when n/x is large, the successive ratios grow quickly toward the overflow limit. The computation cannot be allowed to proceed normally, because the next one or two iterations require subtraction of infinities, producing NaN results. To prevent that, the code instead performs an exact downward scaling that is undone in the `return` statement. Our choice of scale factor forces `s`, and thus, the final result, to underflow if scaling is required more than once. For older floating-point designs where overflow is fatal,

Table 21.5: Iteration counts for evaluating the continued fraction of $J_n(x)/J_{n-1}(x)$ in the IEEE 754 128-bit format (approximately 34 decimal digits) using the forward Lentz algorithm. Counts for the Steed algorithm are almost identical.

	n						
x	**2**	**10**	**100**	**1000**	**10 000**	**100 000**	**1 000 000**
1	16	15	10	7	5	7	4
10	37	30	14	9	7	5	5
100	156	148	58	15	9	7	5
1 000	1120	1112	1021	119	16	9	9
10 000	10 259	10 251	10 165	9255	251	15	9
100 000	>25 000	>25 000	>25 000	>25 000	>25 000	533	15
1 000 000	>25 000	>25 000	>25 000	>25 000	>25 000	>25 000	1122

a safe-arithmetic function like the `is_fmul_safe()` procedure that we introduced in our treatment of the incomplete gamma function (see **Section 18.4** on page 560) can check whether the expression assigned to `rkm1` would overflow without causing overflow, and exit the loop with `rk` set to the largest floating-point number.

The private function `cf(n,x)` evaluates the continued fraction, and an instrumented version of its code stores the iteration count in a global variable that allows recovery of the data shown in **Table 21.5**. Evidently, the continued fraction converges quickly when x/n is small, but it is impractical for $x > 1000$ if $x > n$.

Our simple prototype for computing $J_n(x)$ can be improved in at least these ways, most of which are implemented in the C code in the file `jnx.h`:

- For small x, use the general Taylor series (see **Section 21.3** on page 702). It is particularly effective for large n and tiny x.
- If `s` is zero, or if `1/rk` or `1/rkp1` underflows, the calculation of $J_0(x)$ or $J_1(x)$ is unnecessary.
- Instead of allowing the ratios to grow toward the overflow limit, it is better to rescale earlier. A suitable cutoff is roughly the square root of the largest representable number. With that choice, overflow elsewhere in the loop is prevented, and there is no need to have separate code for older systems where overflow is fatal.
- If possible, use higher working precision in the continued fraction and downward recurrences, to improve accuracy near zeros of $J_n(x)$.
- For large $|x|$, and also whenever the continued fraction is found not to converge, switch to the asymptotic expansion given in **Section 21.3** on page 697. Terminate the summations for $P(n,x)$ and $Q(n,x)$ as soon as either the partial sums have converged to machine precision, or else the terms are found to increase. Then take particular care with the trigonometric argument reductions to avoid needless loss of accuracy.
- In the region where the asymptotic series is used, the quality of the underlying cosine and sine function implementations, coupled with exact argument reduction, are of prime importance for the accuracy of $J_n(x)$, as well as for some of the spherical Bessel functions that we treat later in this chapter. The trigonometric code in many existing libraries performs poorly for large arguments, and may cause complete loss of significance in the Bessel functions.

The algorithm does not require storage of all of the right-hand side ratios, $J_k(z)/J_n(z)$, but if we preserve them, we can quickly recover a vector of values $J_0(z)$, $J_1(z)$, $\ldots$, $J_{n-1}(z)$ simply by multiplying each of the saved ratios by $J_n(z)$.

Alternatively, if we determine $J_{n-1}(z)$ and $J_n(z)$ with that algorithm, we can use downward recurrence to find earlier members of the sequence. That algorithm requires $\mathcal{O}(4n)$ floating-point operations, in addition to the work required for the continued fraction, so if n is large, computation of $J_n(x)$ for large x is expensive.

Figure 21.5 on the next page shows the measured errors in our implementation of $J_n(x)$ for a modest value of n.

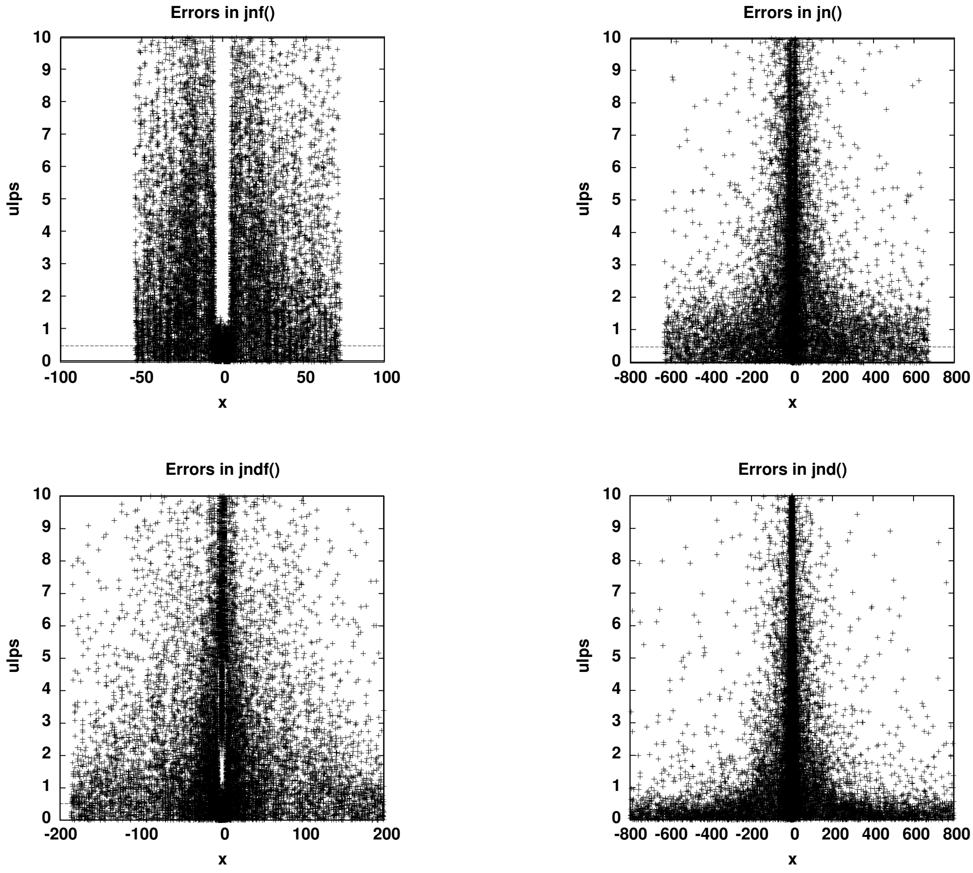


Figure 21.5: Errors in the binary (top) and decimal (bottom) `jn(n,x)` family for $n = 25$.

21.7 Computing $Y_0(x)$ and $Y_1(x)$

As with our implementations of the ordinary Bessel functions of the first kind, for x in $(4, \infty)$, we compute the functions of the second kind, $Y_0(x)$ and $Y_1(x)$, from the trigonometric formulas with the factors $\mathrm{P}(\nu, x)$ and $\mathrm{Q}(\nu, x)$ defined on **Section 21.3** on page 698.

The major difficulties for arguments in $[0, 4]$ are the approach to $-\infty$ as $x \to 0$, and the presence of two zeros of $Y_0(x)$, and one of $Y_1(x)$. Unless the zeros are specifically accounted for, polynomial fits or series sums for the functions have large relative errors near those zeros. Consequently, we prefer to use several different approximations in that region. In the following, we assume that the special arguments of NaN, Infinity, and zero are already handled.

For $Y_0(x)$, we use these regions and Chebyshev polynomial approximations, $f_r(u)$, with u on $[-1, +1]$:

x **in** $(\mathbf{0}, \mathbf{tiny}]$: Sum the four-term Taylor series in order of increasing term magnitudes, with the cutoff chosen so that the magnitude of the last term is below $\frac{1}{2}\epsilon/\beta$, but may affect the rounding of the final result.

x **in** $[\mathbf{3/4}, \mathbf{1}]$: Factor out the root in this region, with $Y_0(x) = (x^2 - s_{0,1}^2) f_2(8x - 7)$. The difference of squares must be factored and computed accurately from a two-part split of the root, as described in **Section 21.5** on page 707.

x **in** $(\mathbf{tiny}, \mathbf{2}]$: Sum the series for $Y_0(x)$ starting with the second term, exiting the loop as soon as the last-computed term no longer affects the sum. Then add the first term. If x is in $[-\exp(-\gamma), -3\exp(-\gamma)]$ (roughly [0.56, 1.69]), the argument of the logarithm lies in $[\frac{1}{2}, \frac{3}{2}]$, and is subject to accuracy loss, so add the term $\mathrm{log1p}(d) \times J_0(x)$, where d is computed accurately with a fused multiply-add operation and a two-part split of the constant $\exp(\gamma)$. Otherwise, add $\log(\mathrm{fma}(v, c_{\mathrm{hi}}, vc_{\mathrm{lo}})) \times J_0(x)$. $Y_0(x)$ is then that sum times $2/\pi$.

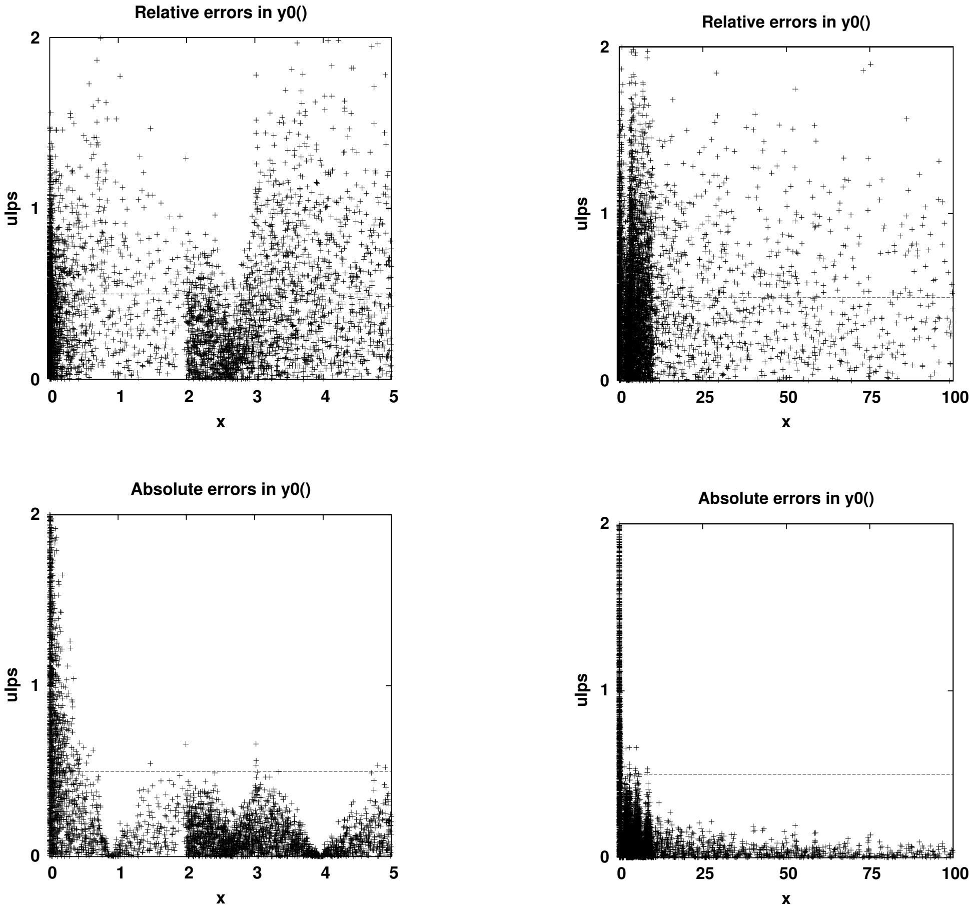


Figure 21.6: Relative (top) and absolute (bottom) errors in `y0(x)` for two argument ranges. The largest relative errors occur near the zeros of the function that lie in $[4, \infty)$, and can grow arbitrarily large. However, the absolute errors remain small.

x **in** $(2, 3]$: The function is positive and decreasing on this interval, so to get an approximation that is the sum of a positive exact value and a positive correction, use $Y_0(x) = Y_0(3) + x^2 f_4((2x^2 - 13)/5)$. The constant $Y_0(3)$ is represented as a two-part split, with the low part added first.

x **in** $(3, 4]$: Factor out the root in this region from the approximation, and compute $Y_0(x) = x^2(x^2 - s_{0,2}^2) f_5(2x - 7)$, with the usual careful handling of the difference of squares.

x **in** $(4, \infty)$: Use the P–Q fit.

The final approximation form in each region is the result of experiments with several alternatives suggested by prior work. The goal is to find auxiliary functions that are almost linear on the region, in order to minimize the length of the Chebyshev expansions. Most published work on computing Bessel functions ignores the issue of the function zeros, and thus, can only achieve small absolute, rather than relative, error. For our approximations, the worst case for 16-digit accuracy is region $(2, 3]$, where Chebyshev terms up to $T_{25}(u)$ are needed.

Cody and Waite generally recommend making the implementation of each elementary or special function independent of related ones. However, for small arguments, eliminating the dependence of $Y_0(x)$ on $J_0(x)$ leads to excessively long polynomial expansions. Further searches for alternate approximation forms are needed to see whether that blemish can be removed without losing accuracy.

Figure 21.6 shows the measured errors in our implementation of $Y_0(x)$. Because the function magnitude grows as its argument approaches zero, the absolute errors must also increase in that region.

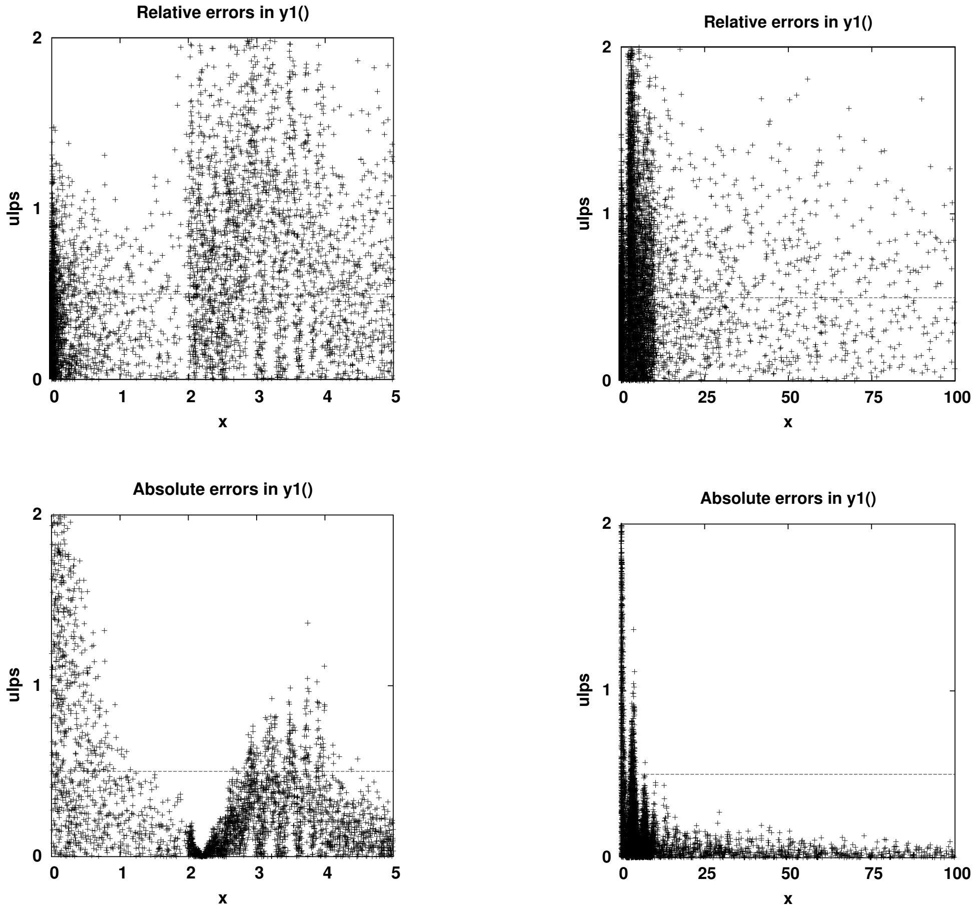


Figure 21.7: Relative (top) and absolute (bottom) errors in `y1(x)` for two argument ranges. The largest relative errors occur near the zeros of the function that lie in $[4, \infty)$, and can grow arbitrarily large. However, the absolute errors remain small.

For $Y_1(x)$, we use these regions and different Chebyshev polynomial fits, $f_r(u)$:

$\boldsymbol{x}$ **in** $\mathbf{(0, tiny]}$: Sum the four-term Taylor series, similar to our approach for tiny arguments in $Y_0(x)$.

$\boldsymbol{x}$ **in** $\mathbf{(tiny, 2]}$: Use $Y_1(x) = (2/\pi)(\log(x) \times J_1(x) - 1/x) + x f_2(x^2/2 - 1)$.

$\boldsymbol{x}$ **in** $\mathbf{(2, 4]}$: Use $Y_1(x) = (x^2 - s_{1,1}^2) f_3(x - 3)$.

$\boldsymbol{x}$ **in** $\mathbf{(4, \infty)}$: Use the P–Q fit.

Here too, we have been unable to remove the dependence of $Y_1(x)$ on $J_1(x)$ for small arguments without unacceptably long polynomial fits.

Figure 21.7 shows the measured errors in our implementation of $Y_1(x)$.

21.8 Computing $Y_n(x)$

The Bessel function $Y_n(x)$ is complex-valued for negative x, so the code should return a quiet NaN for that case. If NaN is not available in the host floating-point system, then a suitable replacement might be the negative of the largest representable number.

For negative n, use the symmetry relation $Y_{-n}(x) = (-1)^n Y_n(x)$ so that only nonnegative n values need to be considered further.

For $n = 0$ and $n = 1$, use the routines for $Y_0(x)$ and $Y_1(x)$.

For small arguments, $Y_n(x)$ is best computed by summing the general Taylor series (see **Section 21.3** on page 703).

Overflow is possible for tiny x, but the complexity of the series is such that it is not practical to precompute cutoffs for general n below which the code could return $-\infty$, or if infinities are not available, the negative of the largest representable number. Numerical experiments show that for a tiny fixed x and increasing n, the magnitude of $Y_n(x)$ is larger than x^{-n}, so if overflow is a fatal error, then it could be necessary to return an indication of a likely overflow when $x < \exp(-\log(\textit{largest representable number})/n)$, even though that is an expensive test to make.

The recurrence relation for $Y_n(x)$ is stable in the upward direction, so once $Y_0(x)$ and $Y_1(x)$ have been computed by the methods of the preceding section, $Y_{|n|}(x)$ is produced in a loop of short code in $|n| - 2$ iterations. The final result is then negated if n is negative and odd, to account for the symmetry relation.

The explicit formula for $Y_n(x)$ contains an n-term sum, a product of a logarithm and $J_n(x)$, and sum of an infinite number of terms. When $x < 2$, the first sum is larger than the product and the second sum, and the terms in both sums fall off quickly. Upward recurrence requires $Y_0(x)$ and $Y_1(x)$, but they are easier to compute than $J_n(x)$. It is therefore unclear which algorithm should be chosen. Timing tests show that the series algorithm is about two to four times faster than upward recurrence on some platforms, whereas on others, it is about twice as slow. On the common IA-32 architecture, the two algorithms have nearly equal performance.

Although the code for the asymptotic series of $J_n(x)$ can be adapted to compute $Y_n(x)$ with only a one-line change, it is not essential, because upward recurrence from $Y_0(x)$ and $Y_1(x)$ works for arguments of any size. However, we provide code for the asymptotic series for $Y_n(x)$, because there is a tradeoff in efficiency between a long recurrence, and a short sum that also requires a complicated argument reduction inside the trigonometric functions. Timing tests on two common platforms for $Y_{25}(x)$ and $Y_{1000}(x)$ with random arguments in the region where the asymptotic code is used shows that code to be two to ten times faster than the code that uses downward recurrence. The asymptotic series is therefore the default algorithm for large arguments.

We do not have a satisfactory algorithm to handle the case of $n \gg x \gg 1$ for either $J_n(x)$ or $Y_n(x)$, because the continued fraction then converges too slowly to be practical, three-term recurrences take too many steps, and the asymptotic series cannot produce sufficient accuracy. Fortunately, for most applications where Bessel functions are needed, such extreme arguments are rare.

Figure 21.8 on the next page shows the measured errors in our implementation of $Y_n(x)$ for modest n.

21.9 Improving Bessel code near zeros

After this chapter, and its software, were completed, John Harrison described a significant improvement in the computation of the Bessel functions $J_0(x)$, $J_1(x)$, $Y_0(x)$, and $Y_1(x)$ that he implemented in the math library for the Intel compiler family [Har09b]. His code uses separate polynomial fits around the zeros and extrema of those functions for arguments x in $[0, 45]$. Above that region, the functions are represented with single trigonometric functions as

$$J_n(x) \approx \mathcal{P}(1/x)\cos(x - (\tfrac{1}{2}n + \tfrac{1}{4})\pi - \mathcal{Q}(1/x)),$$
$$Y_n(x) \approx \mathcal{P}(1/x)\sin(x - (\tfrac{1}{2}n + \tfrac{1}{4})\pi - \mathcal{Q}(1/x)),$$

where $\mathcal{P}(1/x)$ and $\mathcal{Q}(1/x)$ are polynomials in inverse powers of x. His algorithm improves the relative accuracy of the Bessel functions near their zeros, and ensures monotonocity near their extrema.

Harrison also carried out an exhaustive search to find arguments $x < 2^{90} (\approx 10^{27})$ that are the worst cases for determining correct rounding. For the IEEE 754 64-bit binary format, he found that at most *six* additional bits in the Bessel-function values are required for correct rounding decisions. In particular, accurate computations in the 80-bit format provide more than the required additional bits to guarantee correct rounding in the 64-bit format.

Because the mathcw library supports more floating-point formats, and decimal arithmetic, using Harrison's approach would require a large number of polynomial tables. It therefore is reasonable to ask whether there is a simpler way to improve relative accuracy near the zeros of the Bessel functions. We start by generating the Taylor series of $J_0(x)$ near a point $x = z + d$, such that $J_0(z) = 0$:

```
% maple
> alias(J = BesselJ):
```

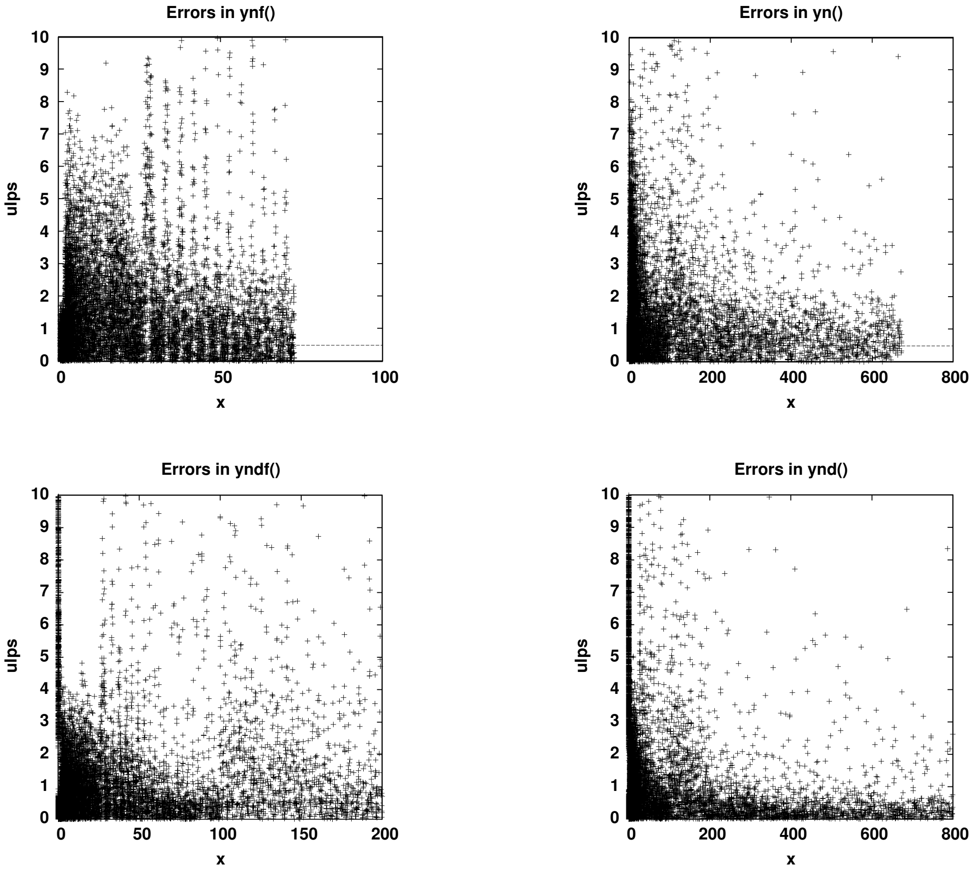


Figure 21.8: Errors in the binary (top) and decimal (bottom) yn(n,x) family for $n = 25$.

```
> taylor(J(0, z + d), d = 0, 5);
                                                   2
J(0, z) - J(1, z) d + (1/4 J(2, z) - 1/4 J(0, z)) d  +
                                   3
    (-1/24 J(3, z) + 1/8 J(1, z)) d  +
                                                   4      5
    (1/192 J(4, z) - 1/48 J(2, z) + 1/64 J(0, z)) d  + O(d )
```

The expansion coefficients require values of higher-order Bessel functions at z, so let us apply the three-term recurrence relation, and then display the numerator and denominator:

```
> t := convert(%, polynom):
> u := subs(J(4,z) = (6/z)*J(3,z) - J(2,z),
            J(3,z) = (4/z)*J(2,z) - J(1,z),
            J(2,z) = (2/z)*J(1,z) - J(0,z),
            J(0,z) = 0, t):
> numer(u);
                3        2      2        2  3      3    3  2
-J(1, z) d (12 z  - 6 d z  + 4 d  z - 2 d  z  - 3 d  + d  z )
> denom(u);
    3
12 z
```

Notice that the Taylor series reduces to a rational polynomial scaled by the constant $J_1(z)$. Next, introduce two intermediate variables and simplify:

```
> simplify(subs(v^2 = w, subs(z = 1/v, u)));
                                   2        2      3        3
1/12 d J(1, 1/v) (-12 + 6 v d - 4 d  w + 2 d  + 3 d  v w - d  v)
```

We now have a simple polynomial in powers of d, scaled by $J_1(z)$.

A more complex Maple program in the file `J0taylor.map` finds the general form of the polynomial coefficients to any desired order. In Horner form, they involve only whole numbers, and look like this:

```
c[0] = 0;
c[1] = -1;
c[2] = (v) / 2;
c[3] = (1 - 2 * w) / 6;
c[4] = ((-1 + 3 * w) * v) / 12;
c[5] = (-1 + (7 - 24 * w) * w) / 120;
c[6] = ((1 + (-11 + 40 * w) * w) * v) / 240;
c[7] = (1 + (-15 + (192 - 720 * w) * w) * w) / 5040;
c[8] = ((-1 + (24 + (-330 + 1260 * w) * w) * w) * v) / 10080;
c[9] = (-1 + (26 + (-729 + (10440 - 40320 * w) * w) * w) * w) / 362880;
...
```

The important point here is that the coefficients for the expansions near *all* of the zeros of $J_0(x)$ are represented with symbolic expressions that can be evaluated at run time for any particular zero, z.

Because v and w are reciprocals, their higher powers decrease and prevent overflow in the coefficients c_k. In addition, the values of the numerators of the leading coefficients are often in $[\frac{1}{2}, 1]$, reducing accuracy loss when the host arithmetic has wobbling precision.

Subtraction loss in the Taylor series is most severe in the sum $c_2d^2 + c_3d^3$, so we can use the series only when $|d| < \frac{1}{2}c_2/c_3 \asymp 3/(2z)$. For the zeros of $J_0(x)$ that we treat, the upper limit on $|d|$ is roughly 0.27.

If we tabulate just the zeros z_k and the corresponding values $J_1(z_k)$, then for any particular zero, z, we can compute numerical values of the coefficients and obtain the Bessel function from its Taylor series as

$$J_0(z + d) = J_1(z)(c_0 + c_1 d + c_2 d^2 + c_3 d^3 + \cdots),$$

where that sum is best evaluated in Horner form.

Given a value x, the Bessel-function zero estimates allow us to find z quickly without having to look at more than three of the tabulated roots. To further enhance accuracy, we store the z_k and $J_1(z_k)$ values as two-part sums of high and low parts, where the high part is exactly representable, correctly rounded, and accurate to working precision.

Similar investigations show that the Taylor series of $J_1(x)$ requires a scale factor of $J_0(z)$, the series for $Y_1(x)$ needs $Y_0(z)$, and that for $Y_1(x)$ needs $Y_0(z)$. The coefficients c_k are identical for $J_0(x)$ and $Y_0(x)$, and another set of c_k values handles both $J_1(x)$ and $Y_1(x)$.

By computing coefficients up to c_{17}, we can use the series for $J_0(x)$ for $|d| < 0.805$ in the IEEE 754 64-bit formats, and $|d| < 0.060$ in the 128-bit formats, subject to the additional limit required to avoid subtraction loss. Otherwise, we fall back to the algorithms described in earlier sections of this chapter.

As long as x lies within the table, we can achieve high accuracy near the zeros. The table requirements are modest: 100 entries handle $x < 311$, and 320 entries suffice for $x < 1000$.

Revised versions of the files `j0x.h`, `j1x.h`, `y0x.h`, and `y1x.h` incorporate optional code that implements the general Taylor-series expansions, and their corresponding header files contain the required data tables of zeros and Bessel-function values at those zeros. The new code is selected by default, but can be disabled with a compile-time preprocessor macro definition. The error reduction in $J_0(x)$ is evident in **Figure 21.9** on the facing page. Plots for other precisions, and the three other Bessel functions, show similar improvements, so they are omitted.

21.10 Properties of $I_n(z)$ and $K_n(z)$

The modified Bessel functions of the first kind, $I_n(z)$, and the second kind, $K_n(z)$, have quite different behavior from the ordinary Bessel functions, $J_n(z)$ and $Y_n(z)$. Instead of taking the form of decaying waves, the modified functions

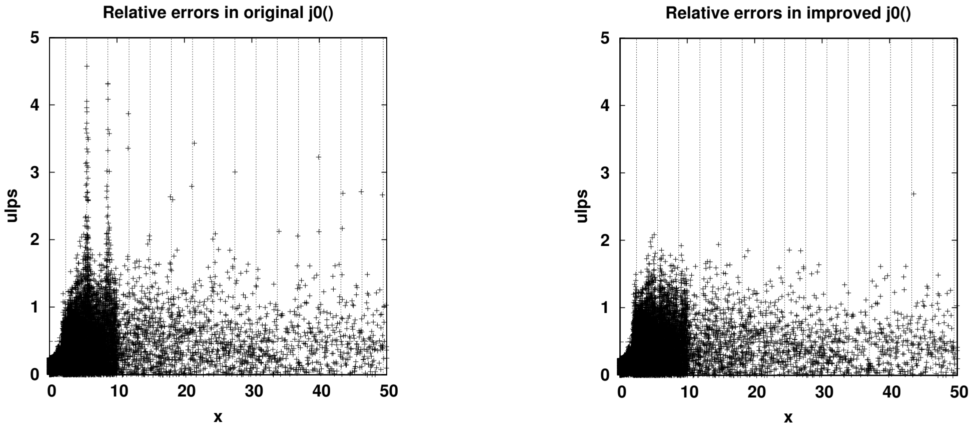


Figure 21.9: Errors in the $J_0(x)$ Bessel function before and after adding code for Taylor-series expansions around the roots indicated by the vertical dotted lines.

resemble rising and falling exponentials, as shown in **Figure 21.10** on the next page. That property makes them computationally easier than the ordinary Bessel functions, because there are no roots where high relative accuracy is difficult to achieve.

The scaled companion functions defined by

$$\mathrm{Is}_\nu(z) = \exp(-z) I_\nu(z), \qquad \mathrm{Ks}_\nu(z) = \exp(+z) K_\nu(z)$$

are finite and representable for arguments over much of the floating-point range, whereas the unscaled ones soon overflow or underflow.

Because the modified Bessel functions are not specified by POSIX or any ISO programming-language standards, we get to choose names for their unscaled and scaled software implementations in the mathcw library. Only one function in Standard C89, and four functions in C99, begin with the letter b, so we choose that letter to prefix our function names, and identify them as members of the Bessel family:

```
double bi0 (double x);             double bk0 (double x);
double bi1 (double x);             double bk1 (double x);
double bin (int n, double x);      double bkn (int n, double x);

double bis0 (double x);            double bks0 (double x);
double bis1 (double x);            double bks1 (double x);
double bisn (int n, double x);     double bksn (int n, double x);
```

They have the usual suffixed companions for other floating-point types.

The modified functions satisfy these symmetry relations:

$$\begin{aligned} I_n(-z) &= (-1)^n I_n(z), && \text{for integer } n, \\ I_{-n}(z) &= I_n(z), && \text{for integer } n, \\ K_{-\nu}(z) &= K_\nu(z), && \text{for real } \nu. \end{aligned}$$

Like $Y_\nu(z)$, the $K_\nu(z)$ functions are complex-valued for negative z. For such arguments, our software implementations therefore call `QNAN("")` to produce a quiet NaN as the return value, and set `errno` to `EDOM`.

Their limiting forms for small arguments, $z \to 0$, are

$$\begin{aligned} I_\nu(z) &\to (z/2)^\nu / \Gamma(\nu+1), && \text{if } \nu \neq -1, -2, -3, \ldots, \\ I_0(0) &= 1, \\ I_n(0) &= 0, && \text{for } n = \pm1, \pm2, \pm3, \ldots, \end{aligned}$$

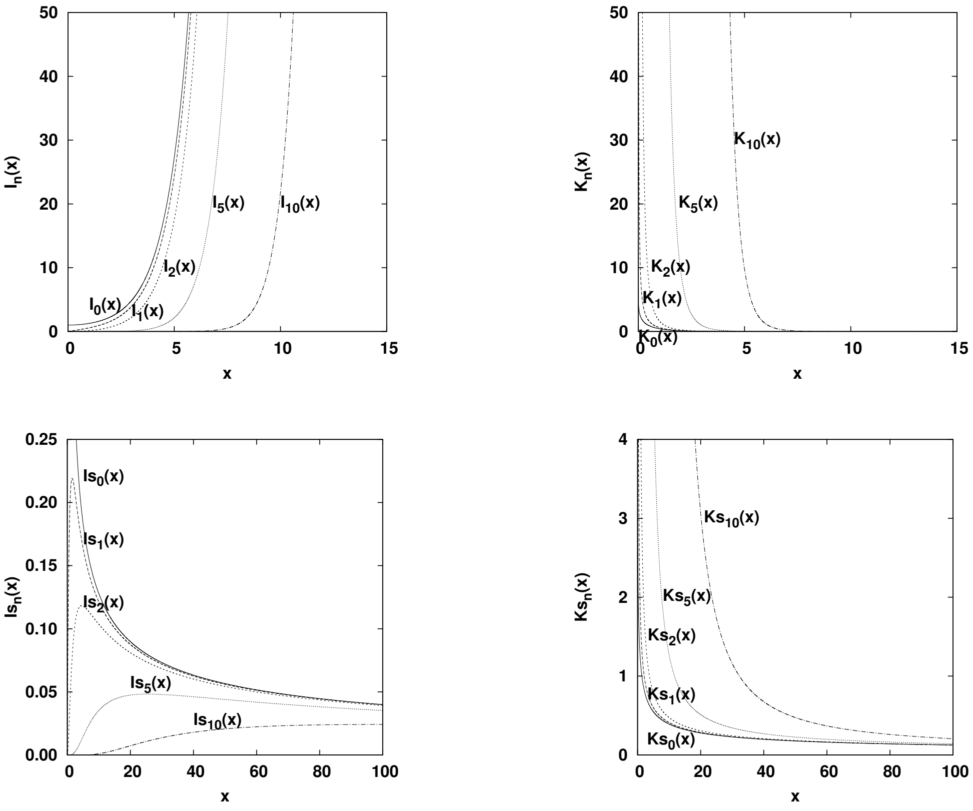


Figure 21.10: Modified Bessel functions, $I_n(x)$ and $K_n(x)$, and scaled modified Bessel functions, $e^{-x}I_n(x)$ and $e^{x}K_n(x)$.

$$\begin{aligned} K_0(z) &\to -\log(z), \\ K_\nu(z) &\to \tfrac{1}{2}\Gamma(\nu)/(z/2)^\nu, && \textit{if } \nu > 0. \end{aligned}$$

For large arguments, $z \to \infty$, the functions behave like this:

$$\begin{aligned} I_\nu(z) &\to \frac{\exp(z)}{\sqrt{2\pi z}} \to +\infty, \\ I_n(-z) &\to (-1)^n \frac{\exp(z)}{\sqrt{2\pi z}} \to (-1)^n\infty, && \textit{for integer } n, \\ K_\nu(z) &\to \exp(-z)\sqrt{\frac{\pi}{2z}} \to 0. \end{aligned}$$

The functions satisfy these three-term recurrence relations:

$$\begin{aligned} I_{\nu+1}(z) &= -(2\nu/z)I_\nu(z) + I_{\nu-1}(z), \\ K_{\nu+1}(z) &= +(2\nu/z)K_\nu(z) + K_{\nu-1}(z). \end{aligned}$$

They are stable in the downward direction for $I_\nu(z)$, and in the upward direction for $K_\nu(z)$. With those direction choices, there is never subtraction loss, because for real arguments and integer orders, the recurrences require only addition of positive terms.

Similar to what we observed on page 700 for $J_\nu(x)$ and $Y_\nu(x)$, when $|x| \gg \nu$, a sequence of $I_\nu(x)$ values for $\nu = 0, 1, 2, \ldots$ looks like $I_0(x)$, $I_1(x)$, $I_0(x)$, $I_1(x)$, $I_0(x)$, $I_1(x)$, ..., and similarly for sequences of $K_\nu(x)$.

Ratios of modified functions of the first kind satisfy a continued fraction similar to that for $J_\nu(z)/J_{\nu-1}(z)$ (see **Section 21.3** on page 702), except that minus signs become plus signs:

$$\begin{aligned}\frac{I_\nu(z)}{I_{\nu-1}(z)} &= 0 + \frac{1}{2\nu/z+}\ \frac{1}{2(\nu+1)/z+}\ \frac{1}{2(\nu+2)/z+}\ \frac{1}{2(\nu+3)/z+}\ \cdots\\ &= 0 + \frac{(1/2)z/\nu}{1+}\ \frac{(1/4)z^2/\nu(\nu+1)}{1+}\ \frac{(1/4)z^2/((\nu+1)(\nu+2))}{1+}\\ &\qquad \frac{(1/4)z^2/((\nu+2)(\nu+3))}{1+}\ \frac{(1/4)z^2/((\nu+3)(\nu+4))}{1+}\ \cdots.\end{aligned}$$

The same computational technique that we described for the ordinary Bessel functions of integer orders allows us to compute ratios down to $I_n(z)/I_0(z)$, and then recover $I_n(z)$ by multiplying that ratio by a separately computed $I_0(z)$. That solves the problem of the unstable upward recurrence for $I_n(z)$. Furthermore, because the ratios are invariant under uniform scaling, the continued fraction is also valid for ratios of scaled functions, $\mathrm{Is}_\nu(z)/\,\mathrm{Is}_{\nu-1}(z)$.

The derivatives of the modified Bessel functions

$$\begin{aligned} dI_n(x)/dx &= +I_{n+1}(x) + nI_n(x)/x,\\ dK_n(x)/dx &= -K_{n+1}(x) + nK_n(x)/x,\end{aligned}$$

lead to these error-magnification factors (see **Section 4.1** on page 61):

$$\begin{aligned}\mathrm{errmag}(I_n(x)) &= xI'_n(x)/I_n(x)\\ &= n + xI_{n+1}(x)/I_n(x),\\ \mathrm{errmag}(K_n(x)) &= n - xK_{n+1}(x)/K_n(x).\end{aligned}$$

The ratios of the Bessel functions are modest, so the general behavior of those factors is $n \pm rx$, where r is the ratio. Thus, errors in the computed functions are expected to grow almost linearly with x, and are large when either n or x is large.

The error-magnification factors of the scaled functions look like this:

$$\begin{aligned}\mathrm{errmag}(\mathrm{Is}_n(x)) &= n - x + xI_{n+1}(x)/I_n(x),\\ \mathrm{errmag}(\mathrm{Ks}_n(x)) &= n + x + xK_{n+1}(x)/K_n(x).\end{aligned}$$

Plots of those functions for fixed n and increasing x show that the first has decreasing magnitude, whereas the second grows. However, for arguments of modest size, the scaled functions should be accurately computable.

These relations between the two functions

$$\begin{aligned} I_\nu(z)K_{\nu+1}(z) + I_{\nu+1}(z)K_\nu(z) &= 1/z,\\ \mathrm{Is}_\nu(z)\,\mathrm{Ks}_{\nu+1}(z) + \mathrm{Is}_{\nu+1}(z)\,\mathrm{Ks}_\nu(z) &= 1/z,\end{aligned}$$

where the terms on the left are positive for positive orders and positive real arguments, are useful for checking software implementations.

Checks can also be made with relations to integrals that can be evaluated accurately with numerical quadrature, as long as their integrands are not too oscillatory:

$$\begin{aligned} I_0(z) &= (1/\pi)\int_0^\pi \cosh(z\cos t)\,dt,\\ I_n(z) &= (1/\pi)\int_0^\pi \exp(z\cos t)\cos(nt)\,dt,\\ K_0(z) &= -(1/\pi)\int_0^\pi \exp(\pm z\cos t)\big(\gamma + \log(2z(\sin(t))^2)\big)\,dt,\\ &= \int_0^\infty \exp(-z\cosh(t))\,dt, && \Re(z) > 0,\\ K_\nu(z) &= \int_0^\infty \exp(-z\cosh(t))\cosh(\nu t)\,dt, && \Re(z) > 0.\end{aligned}$$

There does not appear to be a similar integral for $K_n(z)$ on $[0, \pi]$.

The functions inside the infinite integrals fall off extremely rapidly. For $n = 0$, $x = 1$, and $t = 10$, the integrand is $\mathcal{O}(10^{-4782})$, so quadrature over a small finite range can be used. For example, a 40-point Simpson's rule quadrature for t on $[0, 5]$ produces 34 correct decimal digits for $K_0(1)$.

The modified Bessel function of the first kind has a series that looks like that for $J_\nu(z)$ (see **Section 21.3** on page 702), but without sign changes:

$$I_\nu(z) = (z/2)^\nu \sum_{k=0}^{\infty} \frac{(z^2/4)^k}{k!\,\Gamma(\nu+k+1)}.$$

We can again simplify the formula with the help of two intermediate variables, and also exhibit special cases for positive integer orders, and for $\nu = 0$ and $\nu = 1$:

$$\begin{aligned}
v &= z/2, \\
w &= v^2, \\
I_\nu(z) &= v^\nu \sum_{k=0}^{\infty} \frac{w^k}{k!\,\Gamma(\nu+k+1)}, \\
I_n(z) &= v^n \sum_{k=0}^{\infty} \frac{w^k}{k!\,(k+n)!}, && \text{for integer } n = 0, 1, 2, 3, \ldots, \\
I_0(z) &= \sum_{k=0}^{\infty} \frac{w^k}{(k!)^2}, \\
I_1(z) &= v \sum_{k=0}^{\infty} \frac{w^k}{k!\,(k+1)!}.
\end{aligned}$$

The modified Bessel function of the second kind has a series for integer orders that looks like that for $Y_n(z)$ (see **Section 21.3** on page 703):

$$\begin{aligned}
K_n(z) = {}& \tfrac{1}{2} v^{-n} \sum_{k=0}^{n-1} \frac{(n-k-1)!}{k!} (-w)^k - (-1)^n \log(v) I_n(z) \\
& + (-1)^n \tfrac{1}{2} v^n \sum_{k=0}^{\infty} \big(\psi(k+1) + \psi(k+n+1)\big) \frac{w^k}{k!\,(k+n)!}.
\end{aligned}$$

As with the ordinary Bessel functions, we can replace the psi functions of integer arguments by differences of the partial sums of the harmonic series, h_k (see **Section 21.3** on page 703), and the Euler–Mascheroni constant, γ:

$$\begin{aligned}
K_n(z) = {}& \tfrac{1}{2} v^{-n} \sum_{k=0}^{n-1} \frac{(n-k-1)!}{k!} (-w)^k - (-1)^n \log(v) I_n(z) \\
& + (-1)^n \tfrac{1}{2} v^n \sum_{k=0}^{\infty} (h_k + h_{k+n} - 2\gamma) \frac{w^k}{k!\,(k+n)!}.
\end{aligned}$$

We can then recognize part of the infinite sum to be $I_n(z)$, giving a further simplification:

$$\begin{aligned}
K_n(z) = {}& \tfrac{1}{2} v^{-n} \sum_{k=0}^{n-1} \frac{(n-k-1)!}{k!} (-w)^k - (-1)^n (\log(v) + \gamma) I_n(z) \\
& + (-1)^n \tfrac{1}{2} v^n \sum_{k=0}^{\infty} (h_k + h_{k+n}) \frac{w^k}{k!\,(k+n)!}.
\end{aligned}$$

The special cases for the first two orders are:

$$\begin{aligned}
K_0(z) &= -(\log(v) + \gamma) I_0(z) + \sum_{k=0}^{\infty} h_k \frac{w^k}{(k!)^2}, \\
K_1(z) &= \frac{1}{2v} + (\log(v) + \gamma) I_1(z) - \frac{v}{2} \sum_{k=0}^{\infty} (h_k + h_{k+1}) \frac{w^k}{k!\,(k+1)!}.
\end{aligned}$$

The terms in the infinite sums are all positive, providing better numerical stability than we have for the sums in $Y_n(z)$, where the terms have alternating signs. Unfortunately, there is subtraction loss when the infinite sums are added to the remaining terms. The formula for $K_0(z)$ is only stable when $z < 0.825$, and that for $K_1(z)$, when $z < 1.191$, provided that the logarithmic factor is handled properly.

The series for the modified Bessel functions of the first kind have these leading terms, suitable for use with small arguments:

$$\begin{aligned} I_0(z) &= 1 + w + w^2/4 + w^3/36 + w^4/576 + w^5/14\,400 + \cdots, \\ I_1(z) &= v(1 + w/2 + w^2/12 + w^3/144 + w^4/2880 + w^5/86\,400 + \cdots). \end{aligned}$$

All terms are positive, and convergence is rapid for small arguments. With the six terms shown, values of $z = 1$, $z = 1/10$, and $z = 1/1000$ produce results correct to 7, 17, and 37 decimal digits, respectively.

On systems with hexadecimal floating-point arithmetic, the series coefficients should be halved, and the final result doubled, so as to reduce accuracy loss from wobbling precision.

To simplify the series for the modified Bessel functions of the second kind, we again introduce the intermediate variable

$$s = \log(v) + \gamma.$$

To avoid loss of leading digits, s must be computed carefully as described in **Section 21.3** on page 704. We then have these formulas for fast computation for small arguments, without the need for values of other Bessel functions:

$$\begin{aligned} K_0(z) &= -s + (1 - s)w + (1/8)(-2s + 3)w^2 + (1/216)(-6s + 11)w^3 + \\ &\quad (1/6912)(-12s + 25)w^4 + (1/864\,000)(-60s + 137)w^5 + \cdots, \\ K_1(z) &= \frac{1}{2v}(1 + (2s - 1)w + (1/4)(4s - 5)w^2 + (1/18)(3s - 5)w^3 + \\ &\quad (1/1728)(24s - 47)w^4 + (1/86\,400)(60s - 131)w^5 + \cdots). \end{aligned}$$

The terms in those series diminish rapidly for $z < 2$. For $z = 1$, the six terms shown produce results correct to nearly six decimal digits. For $z = 1/10$, they give function values good to 16 decimal digits. For $z = 1/1000$, the results are accurate to 36 digits.

There are asymptotic expansions of the modified Bessel functions for large arguments:

$$\begin{aligned} \mu &= 4\nu^2, \\ I_\nu(z) &\asymp \frac{\exp(z)}{\sqrt{2\pi z}}\Bigg(1 - \frac{\mu - 1^2}{8z} + \frac{(\mu - 1^2)(\mu - 3^2)}{2!\,(8z)^2} \\ &\qquad - \frac{(\mu - 1^2)(\mu - 3^2)(\mu - 5^2)}{3!\,(8z)^3} + \cdots\Bigg), \\ K_\nu(z) &\asymp \sqrt{\frac{\pi}{2z}}\exp(-z)\Bigg(1 + \frac{\mu - 1^2}{8z} + \frac{(\mu - 1^2)(\mu - 3^2)}{2!\,(8z)^2} \\ &\qquad + \frac{(\mu - 1^2)(\mu - 3^2)(\mu - 5^2)}{3!\,(8z)^3} + \cdots\Bigg). \end{aligned}$$

Asymptotic expansions usually limit the attainable accuracy (see **Section 2.9** on page 19), but provided that $|z| > \nu^2$, the sums can be computed to machine precision.

Although the asymptotic formulas look complex, the parenthesized sums are straightforward to compute, using the term recurrences

$$t_0 = 1, \qquad t_k = (-1)^p \frac{\mu - (2k - 1)^2}{8kz} t_{k-1}, \qquad k = 1, 2, 3, \ldots,$$

where $p = 1$ for $I_\nu(z)$, and $p = 0$ for $K_\nu(z)$. For fixed ν and large z, convergence is rapid. As usual, accuracy is improved by omitting the first one or two terms, summing the remaining terms until the result has converged to machine precision, and finally, adding the omitted terms. For a hexadecimal base, the series for $K_\nu(z)$ has the undesirable form $r \times (1 + \delta)$: compute it as $2 \times (r \times (\frac{1}{2} + \frac{1}{2}\delta))$ to avoid unnecessary loss of leading bits from wobbling precision.

We use the asymptotic formulas directly for large arguments with $\nu = 0$ and $\nu = 1$. For smaller z values, we use their forms as a guide, and for the parenthesized sums, compute polynomial expansions in the variable $t = 1/z$. That way, the exponential behavior is handled entirely by the exponential functions, and the parenthesized sums are $\mathcal{O}(1)$, far from the overflow and underflow limits.

21.11 Computing $I_0(x)$ and $I_1(x)$

After much experimentation based on published algorithms for computing the modified Bessel functions of the first kind, and examination of their error plots, this author concluded that many of those recipes are inadequate, and more care is needed to achieve the accuracy expected of functions in the mathcw library.

The lack of an argument-reduction formula for the Bessel functions means that we need to handle arguments over the entire floating-point range, and that requires more intervals, each with separate polynomial approximations.

Although Maple is able to compute Chebyshev fits to formulas involving the Bessel functions over a reasonable range of arguments, high-precision minimax fits in some argument regions are infeasible. For example, Maple reports failure like this:

```
% maple
> with(numapprox):
> alias (BI = BesselI):
> BIS := proc(n, x) return exp(-x) * BI(n, x) end proc:
> Digits := 800:
> minimax((BIS(1,x) - 13/128), x = 10 .. 25, [11, 11], 1, 'maxerror'):
  Error, (in numapprox:-remez) error curve fails to oscillate
  sufficiently; try different degrees
```

That error can often be made to disappear by increasing the value of `Digits`, but its value here of 800 is the result of several earlier unsuccessful experiments with lower precisions, and the computation time at the point of failure is excessive.

The two scaled functions decay smoothly, and slowly, for $x > 2$. That suggests using fits on intervals $[a, b]$ to functions such that

$$\mathrm{Is}_n(x) = d + \begin{cases} f_1(x), \\ f_2(x)/x, \\ f_3(x)/x^2, \\ f_4(\sqrt{x})/x. \end{cases}$$

Here, the constant d is chosen to be roughly the function average on the interval, $\frac{1}{2}(\mathrm{Is}_n(a) + \mathrm{Is}_n(b))$, and the interval $[a, b]$ is selected to make the fitting function a *small* correction to d. That way, we can get results that are often correctly rounded, remedying a deficiency of most published algorithms for those functions. Because the functions decay, the correction is positive for $x \approx a$, and negative for $x \approx b$. We adjust d to a nearby value that is exactly representable in both binary and decimal `float` formats, such as $d = 25/128 = 0.195\,312\,5\dot{0}$, and so that the magnitude of a negative correction is smaller than $d/2$, preventing loss of leading bits in the subtraction.

In practice, there is little difference in the lengths of the Chebyshev expansions for $f_1(x)$, $f_2(x)$, $f_3(x)$, and $f_4(\sqrt{x})$, so we pick the simplest form, $f(x) = f_1(x) = \mathrm{Is}_n(x) - d$. A short private function then makes it easy to compute the scaled Bessel function on any of the required intervals:

```
static fp_t
evch (fp_t x, fp_t a, fp_t b, fp_t d, const int nc, const fp_t c[])
{   /* compute Is(n,x) = d + f(x), where f(x) is a Chebyshev fit */
    fp_t sum, u;

    u = (x + x - (b + a)) / (b - a);
    sum = ECHEB(u, nc, c);
    return (d + sum);
}
```

That function is used in a series of range tests, one of which looks like this:

```
    else if (x < FP(10.0))
        result = evch(x, FP(1.0), FP(10.0), FP(25.0) / FP(128.0), NC1_10, C1_10);
                      /* I0(x) = 25/128 + f(x), for x in [1,10] */
```

The symmetry relations allow us to compute only for positive arguments. For $\mathrm{Is}_1(x)$, if the input argument is negative, we must reverse the sign of the result computed with $|x|$ before returning.

After the usual checks for special arguments, the final algorithm adopted for the scaled functions `bis0(x)` and `bis1(x)` looks like this:

***x* in $(\mathbf{0}, \mathbf{small}]$** : Use a *six*-term loop-free Taylor series. That is a larger term count than we usually handle for small arguments, but doing so allows a larger cutoff, and faster computation.

***x* in $(\mathbf{small}, \mathbf{1}]$** : Evaluate a Chebyshev fit to the series in powers of w (see Section 21.10 on page 722).

***x* in $(\mathbf{1}, \mathbf{10}]$** : Use $\mathrm{Is}_n(x) = d + f(x)$, where d is chosen as described earlier, and $f(x)$ is a Chebyshev fit.

***x* in $(\mathbf{10}, \mathbf{25}]$** : Similar to previous interval, with different constant and fitting function.

***x* in $(\mathbf{25}, \mathbf{100}]$** : Ditto.

***x* in $(\mathbf{100}, \mathbf{1000}]$** : Ditto.

***x* in $(\mathbf{1000}, \mathbf{10^8}]$** : Use $\mathrm{Is}_n(x) = \sqrt{x}\, p(1/x)$, where $p(1/x)$ is a Chebyshev fit.

***x* in $(\mathbf{10^8}, \boldsymbol{\infty})$** : Sum the parenthesized asymptotic series to machine precision, where at most nine terms suffice for 70 decimal digits of accuracy. The final result is the product of that sum and `RSQRT((x + x) * PI)`, unless the base is 16, in which case, we must avoid leading zero bits in the stored constant π, so we compute the multiplier as $\sqrt{1/\pi} \times (\sqrt{\frac{1}{2}} \times$ `RSQRT(x)`$)$, where the two leading square-root constants are precomputed.

The modified Bessel functions of the first kind, `bi0(x)` and `bi1(x)`, without scaling, are computed like this:

***x* in $(\mathbf{0}, \mathbf{tiny}]$** : Use a fast three-term Taylor series.

***x* in $(\mathbf{tiny}, \mathbf{5}]$** : Sum the series in powers of w (see Section 21.10 on page 722). At most 43 terms are needed for 70 decimal digits of accuracy.

***x* in $(\mathbf{5}, \boldsymbol{\infty})$** : The relations to the unscaled functions are

$$\texttt{bi0(x)} = \exp(x) \times \texttt{bis0(x)}, \qquad \texttt{bi1(x)} = \exp(x) \times \texttt{bis1(x)}.$$

Compute the exponential function and the scaled Bessel function, but if the exponential function overflows, compute their product indirectly from a logarithm and another exponential. For example, the code for $I_0(x)$ looks like this:

```
s = EXP(x);
t = BIS0(x);
result = (s < FP_T_MAX) ? (s * t) : EXP(x + LOG(t));
```

For large x, $\mathrm{Is}_0(x)$ is smaller than one, so there is a small region near the overflow limit of $\exp(x)$ where exact computation of $s \times t$ might produce a finite representable result, but s overflows. In such a case, we switch to the indirect form to avoid that premature overflow.

A sensibly implemented exponential function returns the largest floating-point number to indicate overflow when Infinity is not available in the floating-point design, so we compare s with that largest value, instead of calling the usual `ISINF()` wrapper.

When $x \leq 5$, no elementary or special functions are required, so the computational speed is largely determined by the number of series terms summed. That argument-dependent count is reasonably close to the values in Table 21.4 on page 703, and is less than 20 for data type `double` on most systems.

Figure 21.11 on the following page through Figure 21.15 on page 730 show the measured errors in our implementations of the modified Bessel functions of the first kind.

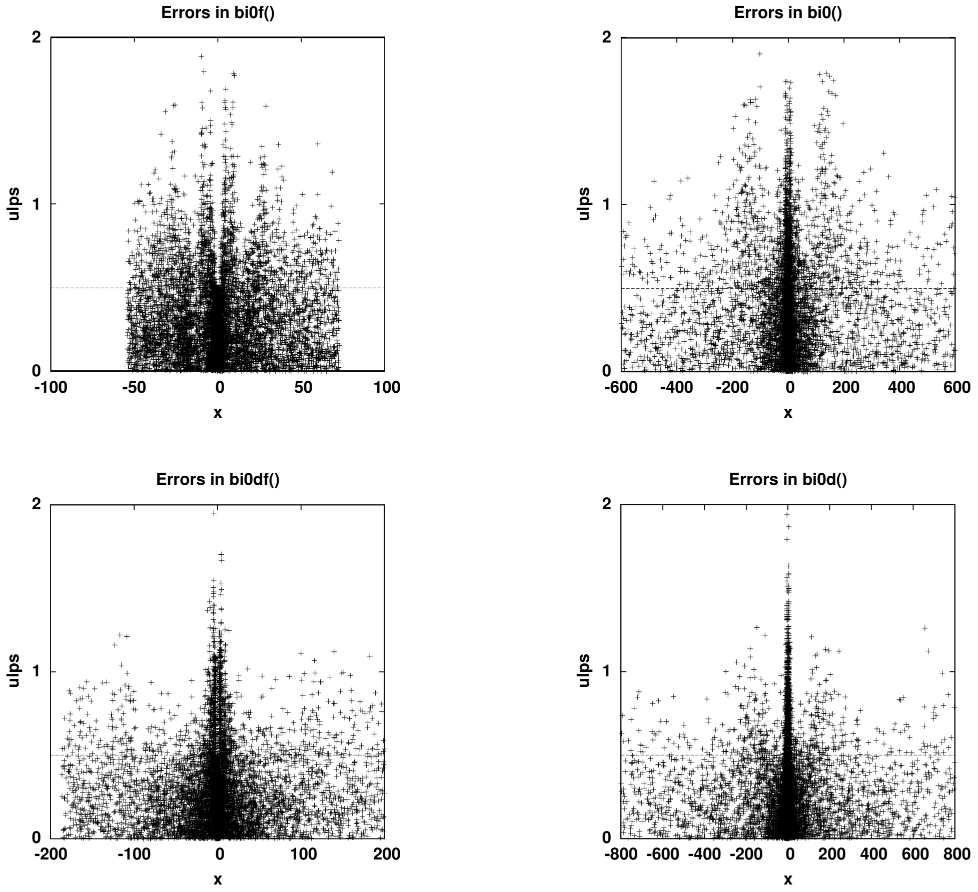


Figure 21.11: Errors in the binary (top) and decimal (bottom) `bi0(x)` family.

21.12 Computing $K_0(x)$ and $K_1(x)$

For technical reasons of exponent-size limitations, and how the `chebyshev()` function accesses the user-provided function to be fit to a Chebyshev expansion, Maple is unable to compute fits to the Bessel functions $K_0(x)$ and $K_1(x)$ for expansions in $1/x$ when $x > 10^8$. Although it seems reasonable to ask for a fit to the function $f(t) = \exp(-1/t)K_0(1/t)\sqrt{t}$, which is smooth and nearly linear for tiny t, Maple reports failure:

```
% maple
> with(numapprox):
> Digits := 30:
> chebyshev(exp(1/t) * BesselK(0, 1/t) * sqrt(t),
            t = 0 .. 1/100,
            1.0e-16);
Error, (in numapprox:-chebyshev) function does not evaluate to numeric
```

Attempts to incorporate a conditional test in the argument function to check for tiny t, and switch to the asymptotic formula, fail with the same error report.

After dealing with the usual special arguments, the final algorithm adopted for the scaled functions `bks0(x)` and `bks1(x)` follows these steps:

x in $(0, a]$: Here $a = \frac{3}{4}$ for $\mathrm{Ks}_0(x)$ and $a = 1$ for $\mathrm{Ks}_1(x)$. Evaluate the infinite sums involving harmonic-number coefficients using a Chebyshev expansion in the variable $t = x^2$, which produces fewer terms than an expan-

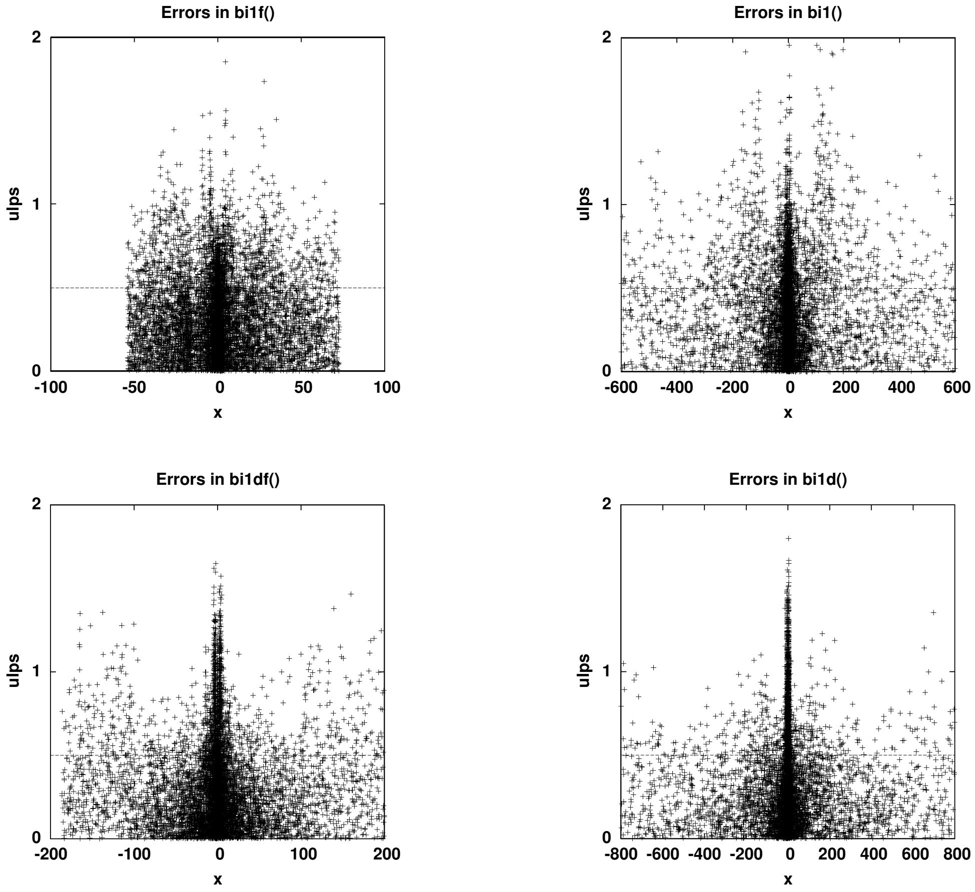


Figure 21.12: Errors in the binary (top) and decimal (bottom) `bi1(x)` family.

sion in the variable x. Add the remaining terms, computing the factor $\log(v) + \gamma$ carefully as described in Section 21.3 on page 704. Multiply the final sum by $\exp(x)$.

***x* in $(a, 5]$** : For $\mathrm{Ks}_0(x)$, use a Chebyshev fit to $(\mathrm{K}_0(t^2) - \frac{3}{4}) \times t^2$ for $t = \sqrt{x}$ in $[\frac{27}{32}, \frac{5}{4}]$. The result is the value of that expansion divided by x, plus $\frac{3}{4}$.

The fit interval is slightly larger than $(a, 5]$ to get a simple form with exact coefficients of the mapping to the Chebyshev variable interval $[-1, +1]$.

For $\mathrm{Ks}_1(x)$, use a Chebyshev fit to $(\mathrm{K}_1(x) - \frac{9}{8}) \times x$. The result is the value of that expansion divided by x, plus $\frac{9}{8}$.

***x* in $(5, 10]$** : Evaluate a Chebyshev fit to $\mathrm{Ks}_0(x) - \frac{5}{32}$ or $\mathrm{Ks}_1(x) - \frac{1}{2}$ on that interval.

***x* in $(10, 25]$** : Similar to previous interval, with different shift constants and fitting function.

***x* in $(25, 100]$** : Ditto.

***x* in $(100, 1000]$** : Ditto.

***x* in $(1000, 10^8]$** : Evaluate a Chebyshev fit with the variable $t = 1/x$.

***x* in $(10^8, \infty)$** : Sum all but the first two terms of the asymptotic expansion to machine precision, then add those two terms.

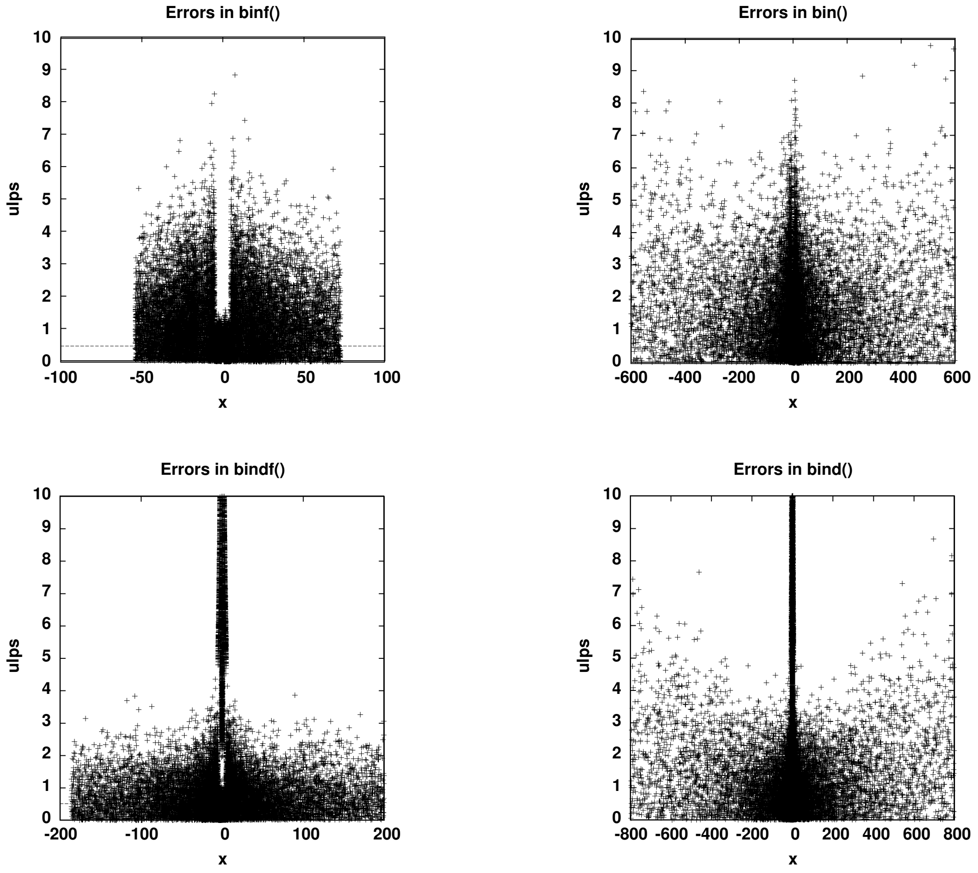


Figure 21.13: Errors in the binary (top) and decimal (bottom) `bin(n,x)` family for $n = 25$.

Although we dislike the dependence in the first region on the exponential function, and in the first two regions on the logarithm and on $I_0(x)$ or $I_1(x)$, no obvious simple functions accurately handle the approach to the singularity at $x = 0$.

We compute the unscaled functions `bk0(x)` and `bk1(x)` in two regions:

$\boldsymbol{x}$ **in** $(\boldsymbol{0}, \boldsymbol{a}]$: Same as for the scaled functions, but omit the final multiplication by $\exp(x)$.

$\boldsymbol{x}$ **in** $(\boldsymbol{a}, \boldsymbol{\infty}]$: In this region, the danger is from underflow instead of overflow. Because the scaled functions are smaller than one for large x, if $\exp(-x)$ underflows, the modified Bessel function of the second kind does as well, and the computation of the scaled function can then be avoided entirely.

Here too, when $x \leq a$, there is direct and hidden dependence on the logarithm, and on $I_0(x)$ or $I_1(x)$. In both regions, there is also dependence on the scaled functions.

Figure 21.17 on page 732 through Figure 21.22 on page 737 show the measured errors in our implementations of the modified Bessel functions of the second kind. An extended vertical scale is used for the tests with $n > 1$.

21.13 Computing $I_n(x)$ and $K_n(x)$

For the modified Bessel function of the first kind for general integer order, $I_n(x)$, and its scaled companion, $\mathrm{Is}_n(x)$, after the usual handling of special arguments, and accounting for any sign change mandated by symmetry relations, we use two computational procedures in `binx.h` and `bisnx.h`:

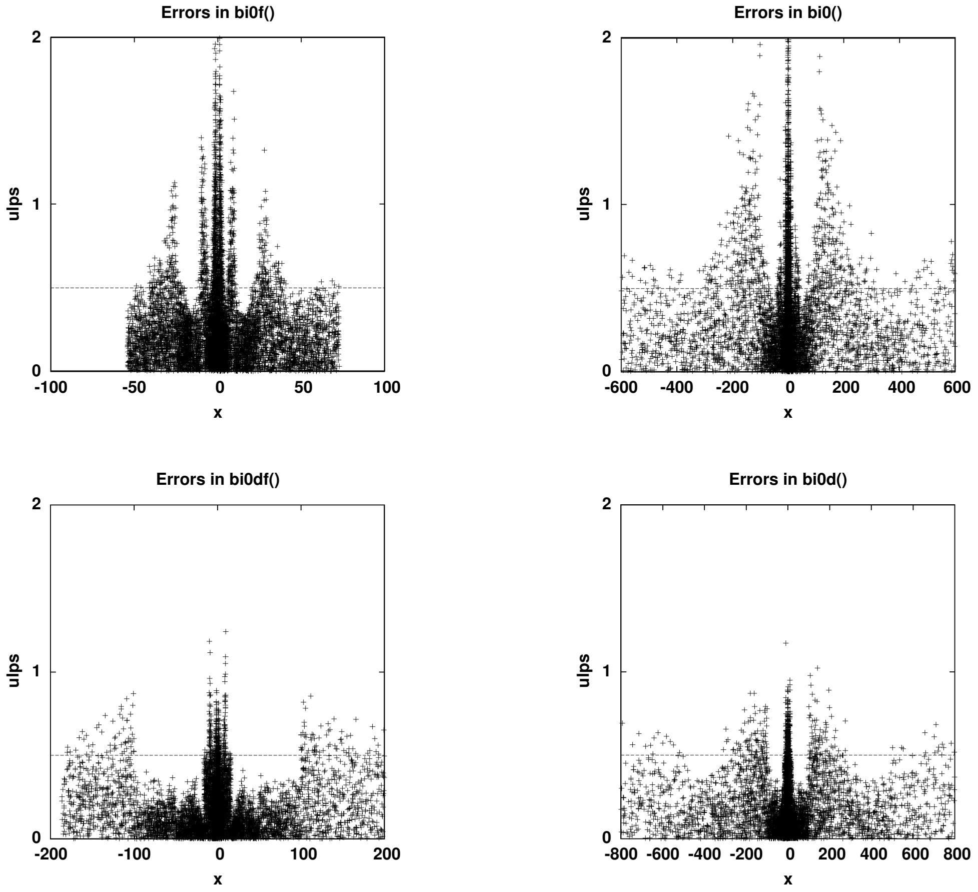


Figure 21.14: Errors in the binary (top) and decimal (bottom) `bis0(x)` family.

***x* in $[0, 5]$** : Sum the series for $I_n(x)$ (see **Section 21.10** on page 722) to machine precision, adding the first two terms last to improve accuracy for small arguments. For the scaled function, multiply the result by $\exp(-x)$.

***x* in $(5, \infty)$** : Evaluate the continued fraction for the ratios $I_n(x)/I_{n-1}(x)$ or $\mathrm{Is}_n(x)/\mathrm{Is}_{n-1}(x)$ (see **Section 21.10** on page 721) using the forward Lentz algorithm. Then use the downward recurrence for the ratios, as in **Section 21.6** on page 711, to find the ratio $I_n(x)/I_0(x)$ or $\mathrm{Is}_n(x)/\mathrm{Is}_0(x)$. The final result is the product of that ratio and the separately computed $I_0(x)$ from `bi0(x)`, or $\mathrm{Is}_0(x)$ from `bis0(x)`.

In both cases, computation time is proportional to the order n, so large orders are costly to compute.

Vector versions of the Bessel functions that we describe later in **Section 21.18** on page 755 compute all orders from 0 to n for fixed x using only the continued fraction algorithm, preserving the ratios in the argument vector for the final scaling by the zeroth-order function.

For the modified Bessel function of the first kind for general integer order, we consider the scaled function $\mathrm{Ks}_n(x)$ as the computational kernel. Because upward recurrence is stable, we can use starting values of `bks0(x)` and `bks1(x)` to obtain the final result needed in `bksn(n,x)`.

For the unscaled function, `bkn(x)`, there are argument regions where the factor $\exp(-x)$ underflows, and $\mathrm{Ks}_n(x)$ overflows, yet their product in exact arithmetic is finite and representable. For example, if $n = 200$ and $x = 110$, then in single-precision IEEE 754 arithmetic, $\exp(-x) \approx 10^{-48}$ underflows, and $\mathrm{Ks}_{200}(110) \approx 10^{66}$ overflows, but the exact product is $\mathcal{O}(10^{18})$, so $K_n(x)$ is representable. Similarly, both $\mathrm{Is}_{200}(110) \approx 10^{-68}$ and $\exp(+x) \approx 10^{48}$ are out of range, yet $I_{200}(110) \approx 10^{-20}$ is representable.

We cannot handle the extremes satisfactorily without intermediate scale factors, or having a separate function to

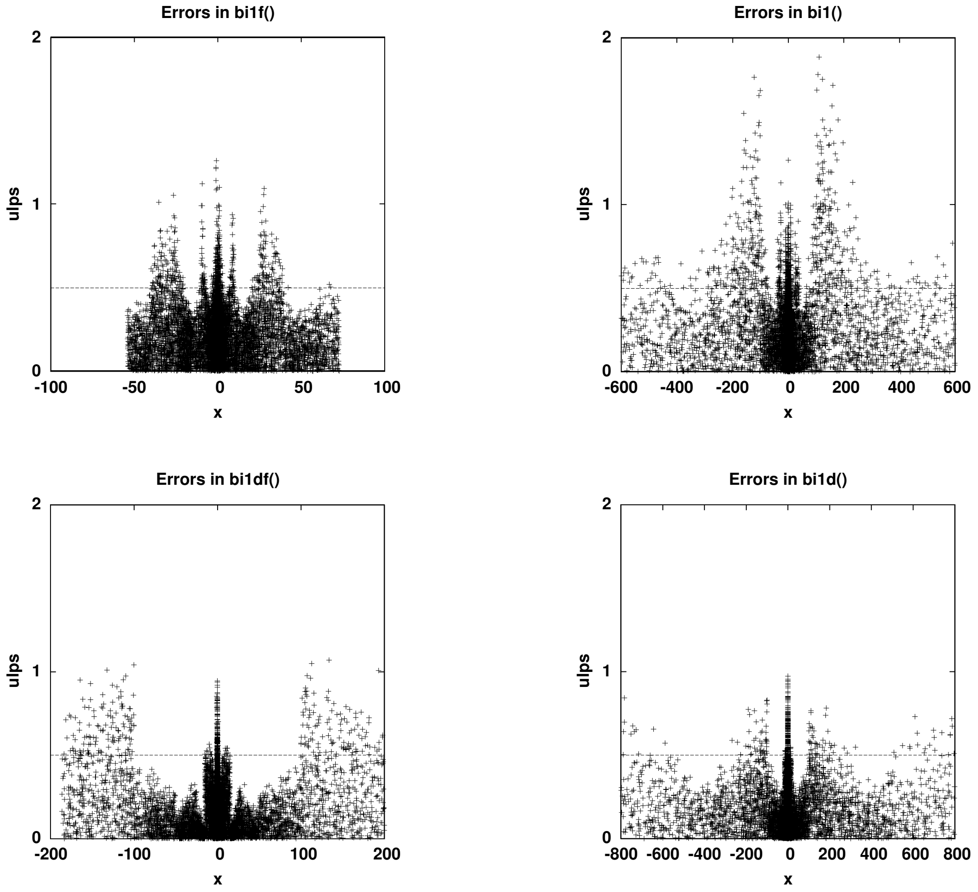


Figure 21.15: Errors in the binary (top) and decimal (bottom) `bis1(x)` family.

compute the logarithms of the scaled functions. The best that we can do is detect and report the problem through the global variable `errno`, with code like this in `bknx.h`:

```
s = EXP(-x);
t = BKSN(n, x);

if (t >= FP_T_MAX)              /* overflow in scaled function */
    result = (s == ZERO) ? SET_ERANGE(QNAN("")) : SET_ERANGE(INFTY());
else
    result = s * t;
```

The Bessel functions are good examples of the need for wider exponent ranges in floating-point designs. User code that requires Bessel functions for large orders or arguments may find it necessary to invoke versions of the functions in the highest available precision, if that format provides more exponent bits that might eliminate out-of-range intermediate results.

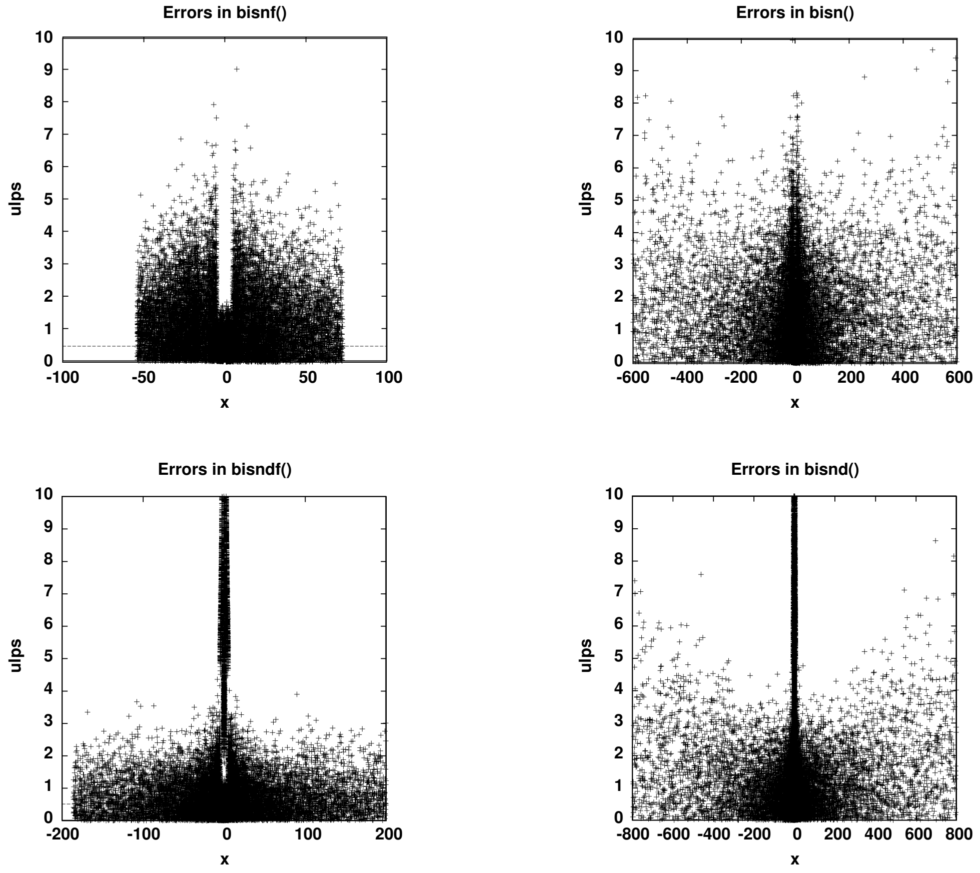


Figure 21.16: Errors in the binary (top) and decimal (bottom) `bisn(n,x)` family for $n = 25$.

21.14 Properties of spherical Bessel functions

The spherical Bessel functions are conventionally denoted by lowercase letters, and are related to the ordinary and modified Bessel functions of *half-integral order*, like this:

$$j_n(z) = \sqrt{\pi/(2z)}J_{n+\frac{1}{2}}(z), \qquad \textit{ordinary spherical Bessel function (first kind)},$$

$$y_n(z) = \sqrt{\pi/(2z)}Y_{n+\frac{1}{2}}(z), \qquad \textit{ordinary spherical Bessel function (second kind)},$$

$$i_n(z) = \sqrt{\pi/(2z)}I_{n+\frac{1}{2}}(z), \qquad \textit{modified spherical Bessel function (first kind)},$$

$$k_n(z) = \sqrt{\pi/(2z)}K_{n+\frac{1}{2}}(z), \qquad \textit{modified spherical Bessel function (second kind)}.$$

Some books, including the *Handbook of Mathematical Functions* [AS64, §10.2], call $k_n(z)$ a function of the *third* kind, even though they refer to its cylindrical companion $K_n(z)$ as a function of the *second* kind. That is another instance of the lack of standardization of Bessel-function terminology.

Unlike the cylindrical Bessel functions, the spherical Bessel functions have closed forms in terms of trigonometric, hyperbolic, and exponential functions, and **Table 21.6** on page 738 shows a few of them. However, the common factor $\sqrt{\pi/(2z)}$ used in most textbook presentations of those functions needs to be rewritten as $\sqrt{\frac{1}{2}\pi}/\sqrt{z}$ to agree with the closed forms. The two factors are identical for both real and complex z, *except* for negative real z, where they differ in

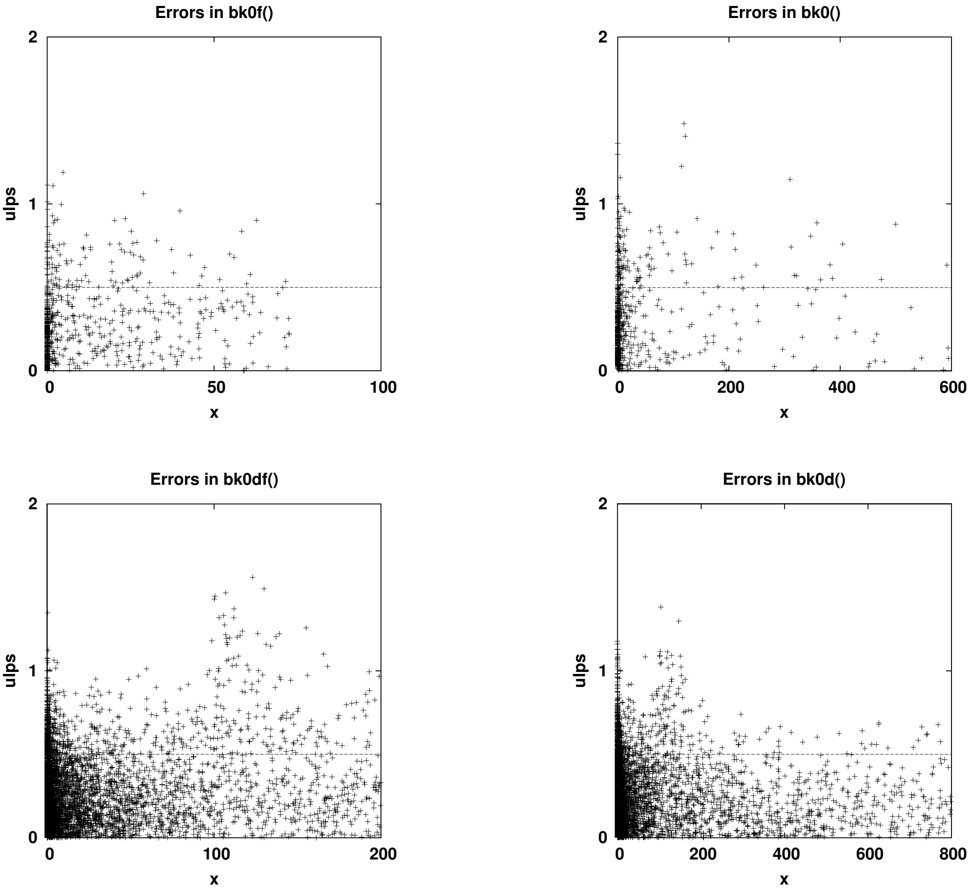


Figure 21.17: Errors in the binary (top) and decimal (bottom) `bk0(x)` family.

sign. The negative real axis here is the *branch cut* of the complex square-root function (see Section 17.3 on page 476). Users of software for computation of spherical Bessel functions with negative arguments should be careful to check the implementation's sign conventions.

The spherical Bessel functions have these symmetry relations:

$$i_n(-z) = (-1)^n i_n(z), \qquad j_n(-z) = (-1)^n j_n(z), \qquad y_n(-z) = (-1)^{n+1} y_n(z).$$

The values $k_n(\pm z)$ do not have a simple relation, although from the tabulated closed forms, we can see that for small $|z|$, where the exponential term is nearly one and the function values are large, we have $k_n(-z) \approx (-1)^{n+1} k_n(z)$.

The modified spherical Bessel functions $i_n(z)$ grow exponentially, and the $k_n(z)$ functions fall exponentially, so scaled companions are commonly used. For large arguments, the scaled functions $\mathrm{is}_n(z)$ and $\mathrm{ks}_n(z)$ are proportional to $z^{-(n+1)}$, and $\mathrm{is}_{-n}(z)$ to z^{-n}, so they are representable in floating-point arithmetic over much of the argument range when n is small. For large n and z, they soon underflow to zero.

Although the square root in their definitions in terms of the cylindrical functions might suggest a restriction to $z \geq 0$, the spherical Bessel functions have real, rather than complex, values for both positive and negative real arguments. Figure 21.23 on page 739 shows plots of the low-order spherical functions.

The error-magnification formulas for the spherical Bessel functions of integer order look like this:

$$\begin{aligned}
\mathrm{errmag}(i_n(x)) &= -\tfrac{1}{2} + \tfrac{1}{2}x(i_{n-1}(x) + i_{n+1}(x))/i_n(x), \\
\mathrm{errmag}(j_n(x)) &= -\tfrac{1}{2} + \tfrac{1}{2}x(j_{n-1}(x) - j_{n+1}(x))/j_n(x), \\
\mathrm{errmag}(k_n(x)) &= -\tfrac{1}{2} - \tfrac{1}{2}x(k_{n-1}(x) + k_{n+1}(x))/k_n(x),
\end{aligned}$$

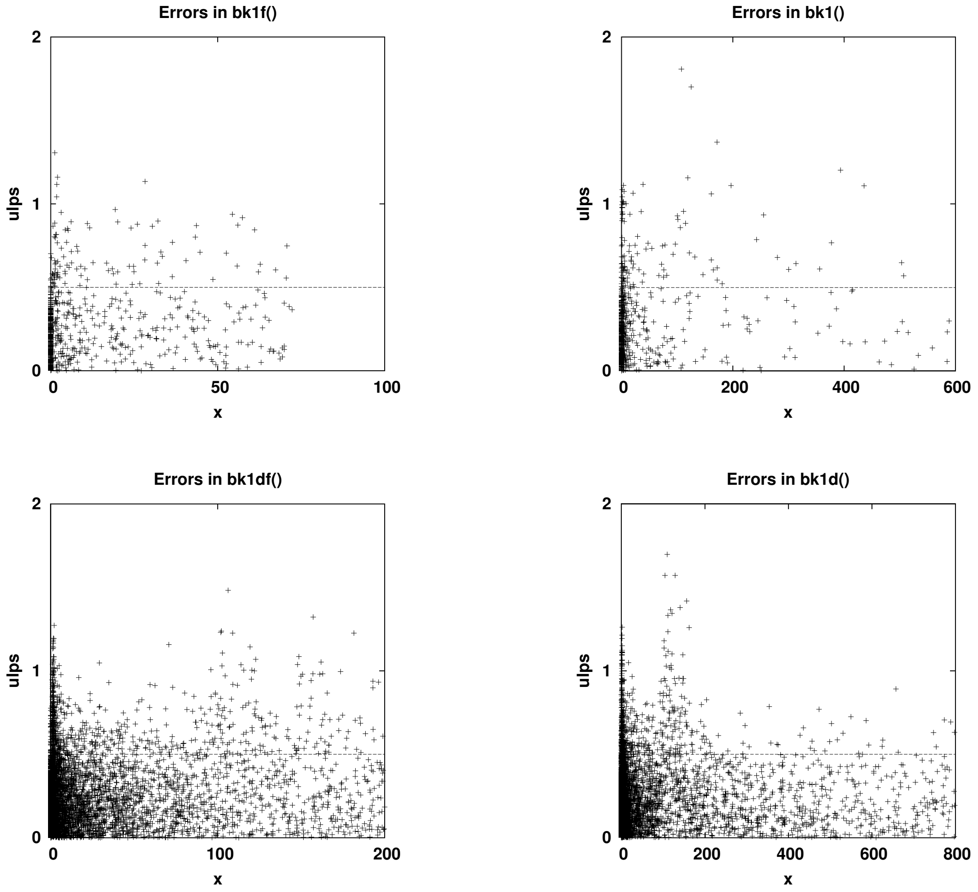


Figure 21.18: Errors in the binary (top) and decimal (bottom) `bk1(x)` family.

$$\mathrm{errmag}(y_n(x)) = -\tfrac{1}{2} + \tfrac{1}{2}x(y_{n-1}(x) - y_{n+1}(x))/y_n(x).$$

Plots of those formulas for various n and modest ranges of x show that errors grow roughly linearly in x for $i_n(x)$ and $k_n(x)$. For the other two, the error factor lies in $[-n, n]$, and away from the zeros of the function, but, of course, grows without bound near those zeros.

The first spherical Bessel function, $j_0(z) = \sin(z)/z$, is identical to the sinc function, `sinc(z)`, which has important applications in approximation theory [LB92, Ste93, KSS95]. Considering its simple form, further mathematical development of the theory of the sinc function is surprisingly complex, but its rewards are rich, leading to rapidly convergent approximations that can produce results of arbitrarily high precision. However, we do not consider the sinc-function approach further in this book.

Because the recurrence relations for the ordinary and modified Bessel functions are valid for arbitrary order ν, the spherical Bessel functions of integer order have similar relations:

$$\begin{aligned}
j_{n+1}(z) &= ((2n+1)/z)j_n(z) - j_{n-1}(z),\\
y_{n+1}(z) &= ((2n+1)/z)y_n(z) - y_{n-1}(z),\\
i_{n+1}(z) &= (-(2n+1)/z)i_n(z) + i_{n-1}(z),\\
k_{n+1}(z) &= ((2n+1)/z)k_n(z) + k_{n-1}(z).
\end{aligned}$$

Those relations are numerically stable in the upward direction for $y_{n+1}(z)$ and $k_{n+1}(z)$, and in the downward direction for the other two.

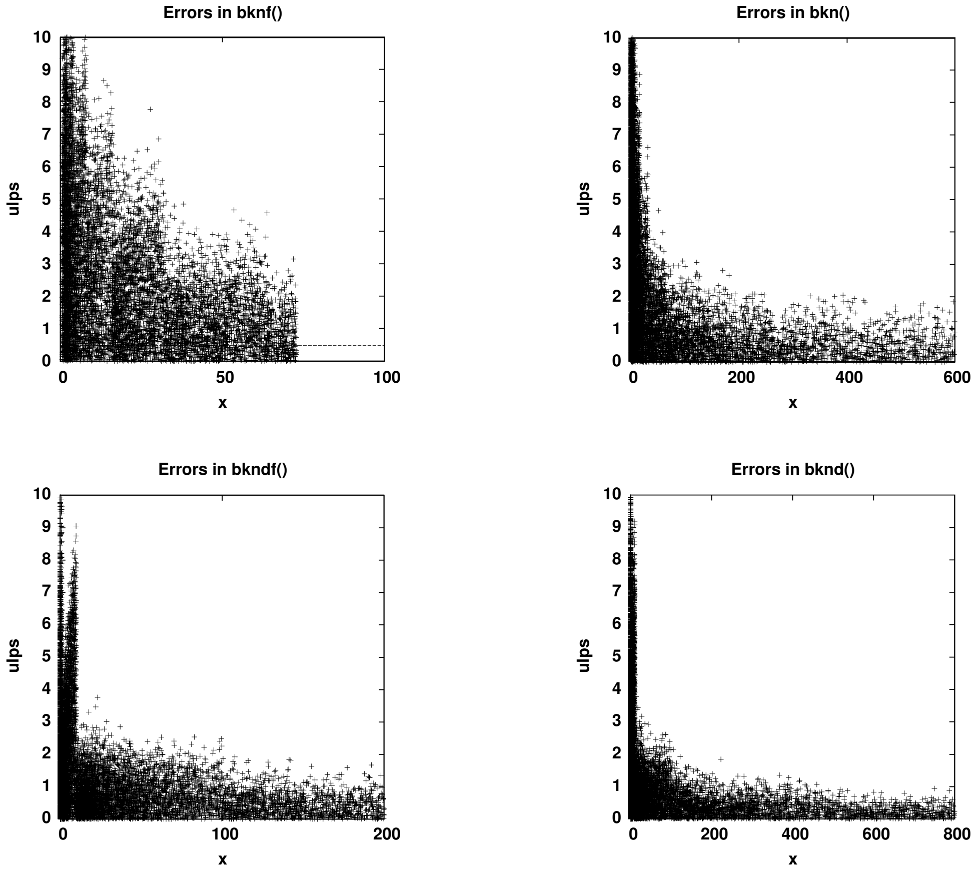


Figure 21.19: Errors in the binary (top) and decimal (bottom) `bkn(n,x)` family for $n = 25$.

As long as the implementations of the trigonometric functions employ *exact argument reduction*, as those in the mathcw library do, $j_0(z)$ and $y_0(z)$ can be computed accurately from the formulas of **Table 21.6** on page 738 for *any* representable value z. However, the higher-order functions $j_n(z)$ and $y_n(z)$ suffer serious subtraction loss when $n > 0$. For real arguments, the $k_n(x)$ functions are well-behaved for all n and $x \geq 0$ because all terms are positive, and the $i_n(x)$ functions have a dominant term for large x, so they too are computationally reasonable. We investigate the stability of the computation of $k_n(x)$ for negative arguments later when we develop computer algorithms for those functions in **Section 21.17.7** on page 754.

Three of the spherical Bessel functions of negative order are simply related to those of positive order:

$$j_{-n}(z) = (-1)^n y_{n-1}(z), \qquad y_{-n}(z) = (-1)^{n+1} j_{n-1}(z), \qquad k_{-n}(z) = k_{n-1}(z).$$

The function $i_{-n}(z)$ does not have a simple relation to $i_{n-1}(z)$, because the hyperbolic cosine and sine are exchanged in the closed forms shown in **Table 21.6** on page 738. However, is z is large, $\cosh(z) \approx \sinh(z)$, so we can conclude that when $z \gg 1$, $i_{-n}(z) \approx i_{n-1}(z)$. For example, $i_{-4}(10)$ and $i_3(10)$ agree to eight decimal digits.

The case of $n < 0$ for $i_n(z)$ is most easily handled by the stable downward recurrence starting from $i_1(z)$ and $i_0(z)$.

The spherical Bessel functions have the limiting behaviors summarized in **Table 21.7** on page 740. The argument of the sine function in the large-argument limit for $j_n(z)$ cannot be determined accurately when n is large unless high precision is available, but the angle sum formula allows it to be replaced by $\sin(z)\cos(n\pi/2) - \cos(z)\sin(n\pi/2)$, and because n is an integer, that collapses to one of $\pm\sin(z)$ or $\pm\cos(z)$. Similar considerations allow accurate reduction of the cosine in the large-argument limit for $y_n(z)$.

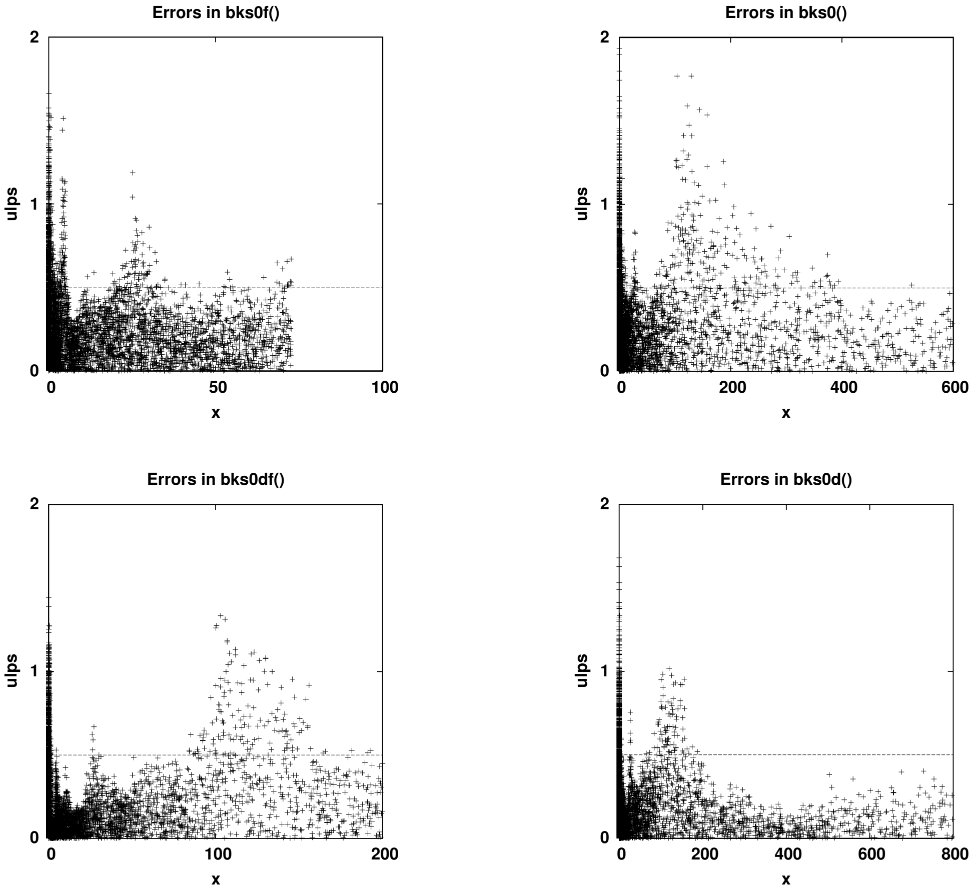


Figure 21.20: Errors in the binary (top) and decimal (bottom) `bks0(x)` family.

21.15 Computing $j_n(x)$ and $y_n(x)$

Because the spherical Bessel functions of the first and second kinds have simple relations to their cylindrical companions, the same computational techniques can be used for both types. However, the closed forms $j_0(x) = \sin(x)/x$ and $y_0(x) = -\cos(x)/x$, together with our exact argument reduction inside the trigonometric functions, suggest the use of those formulas, except for small arguments, where truncated Taylor series provide a faster route. The needed series are trivially obtained from those of the trigonometric functions, so that we have

$$j_0(x) \approx 1 - x^2/6 + x^4/120 - x^6/5040 + x^8/362\,880 - \cdots,$$

$$y_0(x) \approx -\frac{1}{x}(1 - x^2/2 + x^4/24 - x^6/720 + x^8/40\,320 - \cdots).$$

The closed forms $j_1(x) = (-x\cos(x) + \sin(x))/x^2$ and $y_1(x) = -(\cos(x) + x\sin(x))/x^2$ look simple, but suffer massive subtraction loss near the zeros of those functions. For small arguments, series expansions solve the accuracy-loss problem:

$$j_1(x) \approx \frac{x}{3}(1 - x^2/10 + x^4/280 - x^6/15\,120 + x^8/1\,330\,560 - \cdots),$$

$$y_1(x) \approx -\frac{1}{x^2}(1 + x^2/2 - x^4/8 + x^6/144 - x^8/5760 + \cdots).$$

In binary floating-point arithmetic, it is advisable to compute the series for $j_1(x)$ as $x/4 + (x/12 - x^3/30 + \cdots)$, so that the first term is exact, and the second term is a small correction. For other bases, the error from the division by

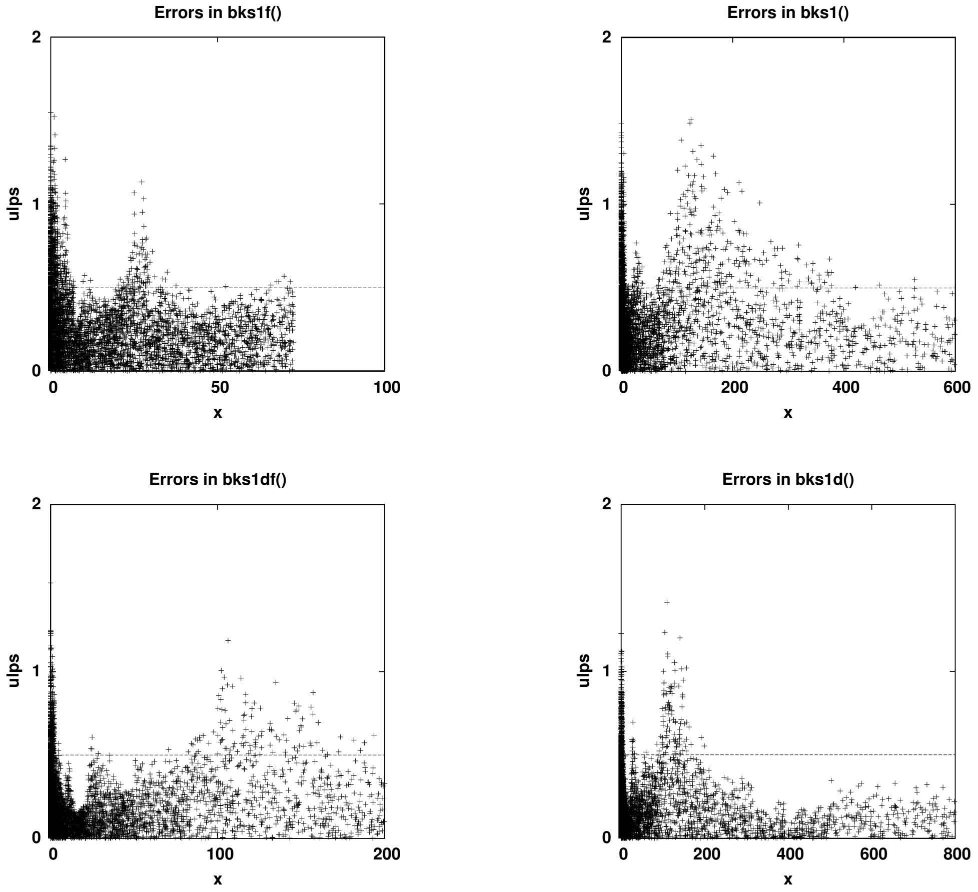


Figure 21.21: Errors in the binary (top) and decimal (bottom) `bks1(x)` family.

three can be reduced if higher precision is used.

By techniques discussed later in **Section 21.17.5** on page 750, we can find the general form of the Taylor series for arbitrary integer $n \geq 0$:

$$\begin{aligned}
j_n(x) = {} & \frac{x^n}{(2n+1)!!}(1 - \frac{1}{2(2n+3)}x^2 + \frac{1}{8(2n+3)(2n+5)}x^4 - \\
& \frac{1}{48(2n+3)(2n+5)(2n+7)}x^6 + \\
& \frac{1}{384(2n+3)(2n+5)(2n+7)(2n+9)}x^8 - \cdots), \\
y_n(x) = {} & -\frac{(2n-1)!!}{x^{n+1}}(1 + \frac{1}{2(2n-1)}x^2 + \frac{1}{8(2n-1)(2n-3)}x^4 + \\
& \frac{1}{48(2n-1)(2n-3)(2n-5)}x^6 + \\
& \frac{1}{384(2n-1)(2n-3)(2n-5)(2n-7)}x^8 + \cdots).
\end{aligned}$$

The series for $j_n(x)$ is free of leading bit loss only for $x < \sqrt{2n+3}$, so the minimal cutoff for using the series with $n \geq 2$ in a computer program is $x_{\mathrm{TS}} = \sqrt{7}$. Although it might not be immediately evident, the terms in the series for

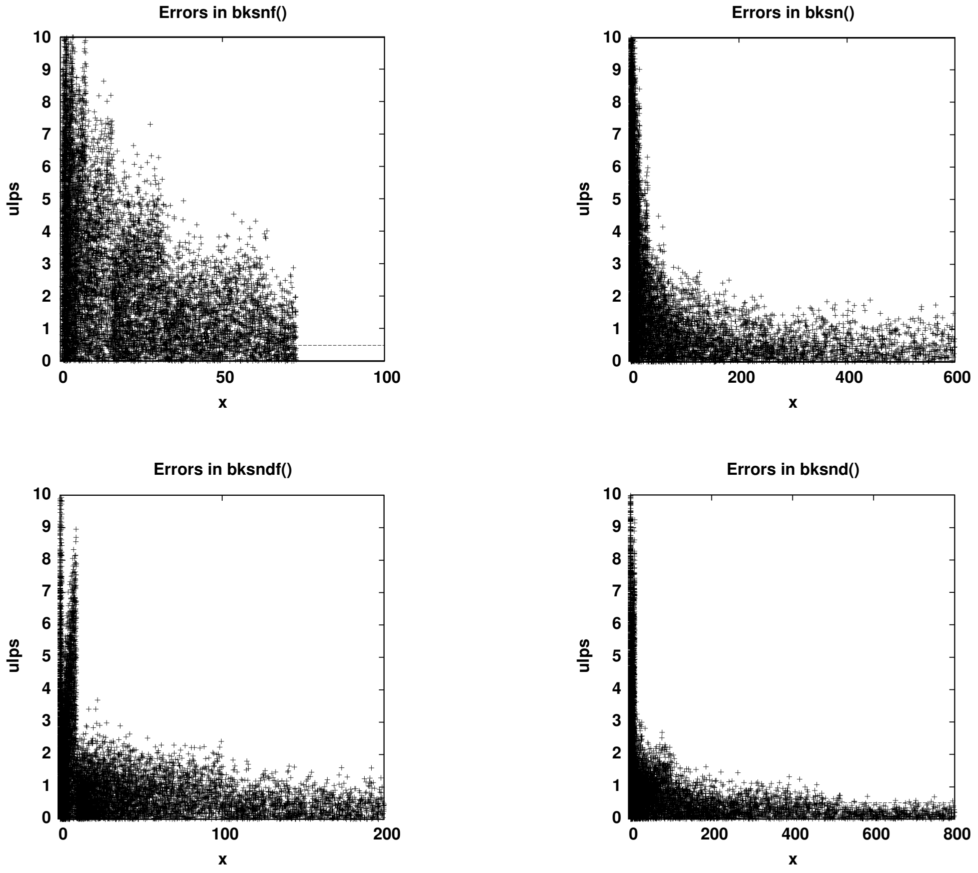


Figure 21.22: Errors in the binary (top) and decimal (bottom) `bksn(n,x)` family for $n = 25$.

$y_n(x)$ can also alternate in sign. For $n = 2$, the term containing x^6 is negative, and subtraction loss is prevented by using the series only for $x < \sqrt{3}$.

The series expansions of $j_n(x)$ and $y_n(x)$ lead to simple recurrence formulas for successive terms:

$$t_0 = 1, \qquad t_k = -\left(\frac{1}{2k}\right)\left(\frac{1}{2n+2k+1}\right) x^2\, t_{k-1}, \qquad \text{for } k = 1, 2, 3, \ldots.,$$

$$u_0 = 1, \qquad u_k = \left(\frac{1}{2k}\right)\left(\frac{1}{2n+1-2k}\right) x^2\, u_{k-1}, \qquad \text{for } k = 1, 2, 3, \ldots,$$

$$j_n(x) = \frac{x^n}{(2n+1)!!}(t_0 + t_1 + t_2 + \cdots),$$

$$y_n(x) = -\frac{(2n-1)!!}{x^{n+1}}(u_0 + u_1 + u_2 + \cdots).$$

Certainly for $|x| < 1$, and also for $n \gg x^2$, the terms fall off so rapidly that there is no subtraction loss when they alternate in sign. Provided that the terms are accumulated in order of increasing magnitudes, the major source of inaccuracy in all but $j_0(x)$ is from the outer multiplication and division. We can further reduce rounding error by factoring out x^2 in the sums after the first term so that, for example, we compute

$$y_1(x) \approx -\left(\frac{1}{x^2} + (1/2 - x^2/8 + x^4/144 - x^6/5760 + \cdots)\right).$$

The widest interval where the closed form for $j_1(x)$ loses leading bits is $[0, 1.166]$, and that for $y_1(x)$ is $[2.458,$

Table 21.6: Explicit forms of low-order spherical Bessel functions, with their order-symmetry relations in **bold**.

$j_0(z) = \sin(z)/z$	$i_0(z) = \sinh(z)/z$
$j_1(z) = (-z\cos(z) + \sin(z))/z^2$	$i_1(z) = (z\cosh(z) - \sinh(z))/z^2$
$j_2(z) = (-3z\cos(z) - (z^2 - 3)\sin(z))/z^3$	$i_2(z) = (-3z\cosh(z) + (z^2 + 3)\sinh(z))/z^3$
$j_3(z) = ((z^3 - 15z)\cos(z) - (6z^2 - 15)\sin(z))/z^4$	$i_3(z) = ((z^3 + 15z)\cosh(z) - (6z^2 + 15)\sinh(z))/z^4$
$y_0(z) = -\cos(z)/z$	$k_0(z) = (\pi/(2z))\exp(-z)$
$y_1(z) = (-\cos(z) - z\sin(z))/z^2$	$k_1(z) = (\pi/(2z^2))(z + 1)\exp(-z)$
$y_2(z) = ((z^2 - 3)\cos(z) - 3z\sin(z))/z^3$	$k_2(z) = (\pi/(2z^3))(z^2 + 3z + 3)\exp(-z)$
$y_3(z) = ((6z^2 - 15)\cos(z) + (z^3 - 15z)\sin(z))/z^4$	$k_3(z) = (\pi/(2z^4))(z^3 + 6z^2 + 15z + 15)\exp(-z)$
$j_{-1}(z) = \cos(z)/z$	$i_{-1}(z) = \cosh(z)/z$
$j_{-2}(z) = (-\cos(z) - z\sin(z))/z^2$	$i_{-2}(z) = (-\cosh(z) + z\sinh(z))/z^2$
$j_{-3}(z) = -((z^2 - 3)\cos(z) - 3z\sin(z))/z^3$	$i_{-3}(z) = ((z^2 + 3)\cosh(z) - 3z\sinh(z))/z^3$
$j_{-4}(z) = ((6z^2 - 15)\cos(z) + (z^3 - 15z)\sin(z))/z^4$	$i_{-4}(z) = ((-6z^2 - 15)\cosh(z) + (z^3 + 15z)\sinh(z))/z^4$
$\boldsymbol{j_{-n}(z) = (-1)^n y_{n-1}(z)}$	
$y_{-1}(z) = \sin(z)/z$	$k_{-1}(z) = (\pi/(2z))\exp(-z)$
$y_{-2}(z) = -(-z\cos(z) + \sin(z))/z^2$	$k_{-2}(z) = (\pi/(2z^2))(z + 1)\exp(-z)$
$y_{-3}(z) = (-3z\cos(z) - (z^2 - 3)\sin(z))/z^3$	$k_{-3}(z) = (\pi/(2z^3))(z^2 + 3z + 3)\exp(-z)$
$y_{-4}(z) = -((z^3 - 15z)\cos(z) - (6z^2 - 15)\sin(z))/z^4$	$k_{-4}(z) = (\pi/(2z^4))(z^3 + 6z^2 + 15z + 15)\exp(-z)$
$\boldsymbol{y_{-n}(z) = (-1)^{n+1} j_{n-1}(z)}$	$\boldsymbol{k_{-n}(z) = k_{n-1}(z)}$

2.975]. We could use polynomial approximations in those regions, but that provides only a partial solution to a problem that occurs uncountably often for larger x values. The code in `sbj1x.h` uses a rational polynomial fit in the first loss region.

For larger arguments, however, the approach in most existing implementations of $j_1(x)$ and $y_1(x)$ is to suffer the subtraction loss from straightforward application of the closed formulas, which means accepting small absolute error, rather than small relative error. That does not meet the accuracy goals of the `mathcw` library, and because no obvious rearrangement of the closed forms for arguments $|x| > 1$ prevents the subtraction loss analytically, the only recourse is then to use higher precision, when available. In practice, that means that the `float` functions can achieve high accuracy, and on some systems, the `double` functions as well. However, the relative accuracy for longer data types can be expected to be poor near the function zeros.

Because two trigonometric functions are usually needed for each of $j_1(x)$ and $y_1(x)$, some libraries compute both spherical Bessel functions simultaneously. Ours does not, but we use the `SINCOS()` function family to get the sine and the cosine with a single argument reduction, and little more than the cost of just one of the trigonometric functions.

For sufficiently large x, the terms containing the factor x in $j_1(x)$ and $y_1(x)$ dominate, and we can avoid one of the trigonometric functions, computing $j_1(x) \approx -\cos(x)/x$, and $y_1(x) \approx -\sin(x)/x$. A suitable cutoff for $j_1(x)$ is found by setting $\sin(x) \approx 1$ and $\cos(x) \approx \epsilon$ (the machine epsilon), for which we have $j_1(x) \approx (-x\epsilon + 1)/x^2$, and 1 is negligible in that sum if $1/(x\epsilon) < \frac{1}{2}\epsilon/\beta$. We solve that to find the cutoff $x_c = 2\beta/\epsilon^2 = 2\beta^{2t-1}$, where t is the number of base-β digits in the significand. The same cutoff works for $y_1(x)$ too.

An alternate approach is to evaluate the continued fraction for the ratio $j_1(x)/j_0(x)$ (see **Section 21.6** on page 710), and then determine $j_1(x)$ from the product of that ratio with an accurate value of $j_0(x)$. Numerical experiments with that technique show that the errors are higher than with direct use of the trigonometric closed forms, even when those forms cannot be computed in higher precision.

For orders $n \geq 2$, $j_n(x)$ is computed using the same algorithm as for $J_n(x)$: series summation for small arguments, and downward recurrence with the continued fraction to find the ratio $j_n(x)/j_0(x)$, from which $j_n(x)$ can be easily found.

Unfortunately, the continued fraction for the Bessel function ratios converges poorly for large arguments: the

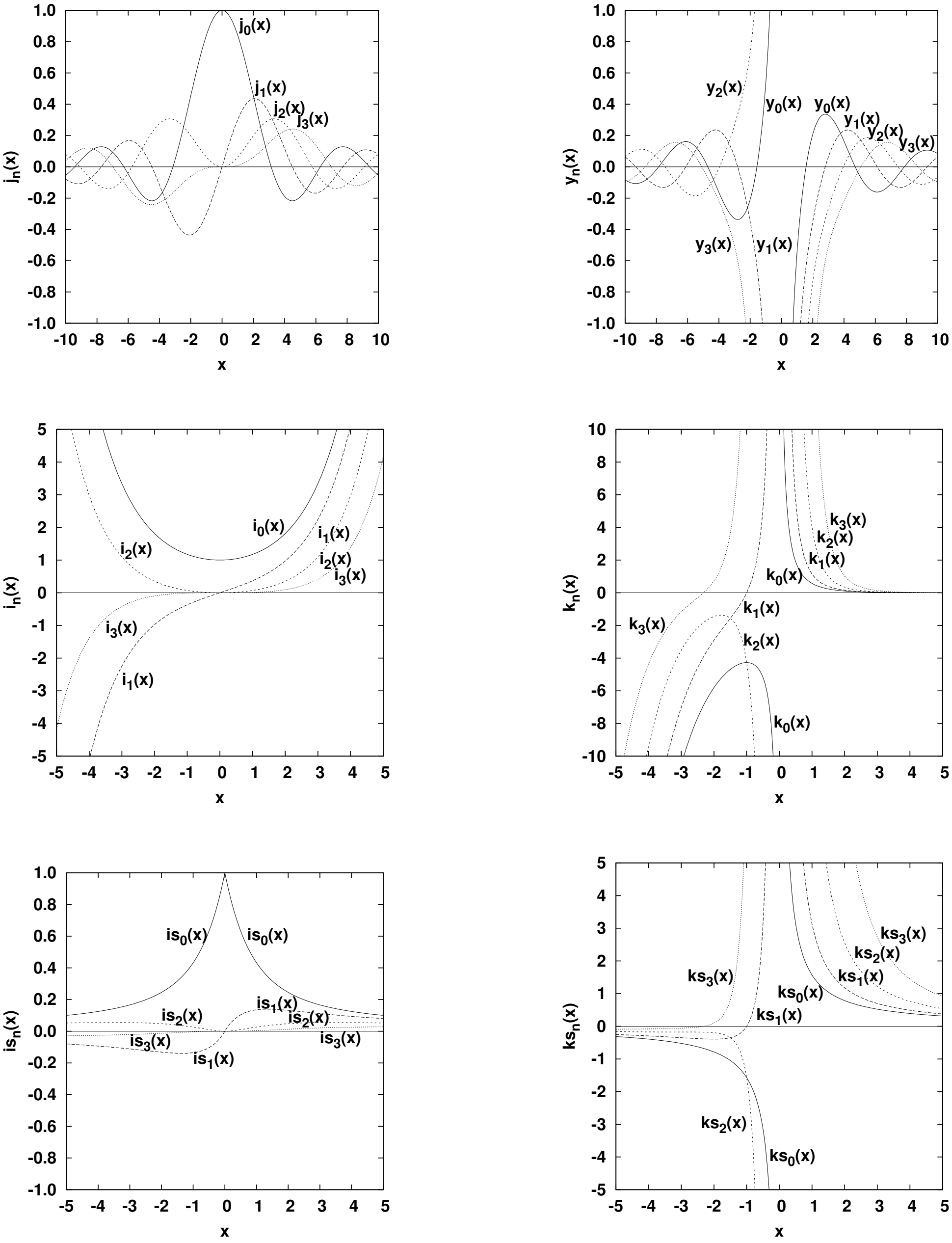


Figure 21.23: Spherical Bessel functions, $j_n(x)$ and $y_n(x)$, modified spherical Bessel functions, $i_n(x)$ and $k_n(x)$, and scaled modified spherical Bessel functions, $e^{-|x|}i_n(x)$ and $e^x k_n(x)$.

number of iterations required is roughly $\lfloor x \rfloor$ (see **Section 6.7** on page 136). It is then better to switch to the formulas suggested by the asymptotic relations involving the functions $\mathrm{P}(\nu, x)$ and $\mathrm{Q}(\nu, x)$ that we introduced in **Section 21.3** on page 698.

For $n \geq 2$, $y_n(x)$ can be computed stably by upward recurrence, or when $(n + \frac{1}{2})^2 < |x|$, by the asymptotic expansion. However, if x lies near a zero of $y_1(x)$, namely, the values for which $x \sin(x) = \cos(x)$, or $x = \cot(x)$, then as we noted, $y_1(x)$ may be inaccurate. Its error may then contaminate all higher $y_n(x)$ values computed with the upward recurrence. The only simple solution to that problem is to use higher-precision arithmetic for the recurrence,

Table 21.7: Limiting values of spherical Bessel functions.

$j_0(0) = 1,$	
$j_n(0) = 0,$	$n > 0,$
$j_n(z) \to (1/z)\sin(z - n\pi/2),$	$\lvert z\rvert \to \infty,$
$j_n(z) \to z^n/(2n+1)!!,$	$\lvert z\rvert \to 0,$
$y_n(0) = -\infty,$	*for all integer* $n \geq 0,$
$y_n(z) \to -(2n-1)!!/z^{n+1},$	$\lvert z\rvert \to 0,$
$y_n(z) \to -(1/z)\cos(z - n\pi/2),$	$\lvert z\rvert \to \infty,$
$i_0(0) = 1,$	
$i_n(0) = 0,$	*for all integer* $n > 0,$
$i_n(z) \to z^n/(2n+1)!!,$	$\lvert z\rvert \to 0,$
$i_{-n}(z) \to (-1)^{n+1}(2n-3)!!/z^n,$	$\lvert z\rvert \to 0,$
$i_n(z) \to \exp(z)/(2z),$	$\lvert z\rvert \to \infty,$
$k_n(0) = +\infty,$	
$k_n(z) \to (\frac{1}{2}\pi)(2n-1)!!/z^{n+1},$	$\lvert z\rvert \to 0,$
$k_n(z) \to (\frac{1}{2}\pi/z)\exp(-z),$	$\lvert z\rvert \to \infty.$

but that is not possible at the highest-available precision.

Figure 21.24 through Figure 21.29 on page 746 show the measured errors in our implementations of the ordinary spherical Bessel functions.

21.16 Improving $j_1(x)$ and $y_1(x)$

In Section 21.9 on page 716, we showed how to use a symbolic-algebra system to find the general form of the Taylor-series expansions near the zeros of low-order ordinary cylindrical Bessel functions. We can do the same for the spherical Bessel functions, but we can omit the order-zero functions because their closed forms can be evaluated accurately near all of their zeros, as long as exact trigonometric argument reduction is available, as it is in the `mathcw` library.

The Maple files `sbj1taylor.map` and `sby1taylor.map` generate the expansions around a particular zero, z:

$$\begin{aligned} j_1(z+d) &= a_0 + a_1 d + a_2 d^2 + a_3 d^3 + a_4 d^4 + \cdots, \\ y_1(z+d) &= b_0 + b_1 d + b_2 d^2 + b_3 d^3 + b_4 d^4 + \cdots. \end{aligned}$$

Four intermediate variables

$$v = 1/z, \qquad w = v^2, \qquad c = \cos(z), \qquad s = \sin(z),$$

simplify the coefficients, the first few of which look like this:

$$\begin{aligned} a_0 &= 0, \\ a_1 &= (((1-2w)s + 2cv)v)/1, \\ a_2 &= (-((3-6w)vs + (-1+6w)c)v)/2, \\ a_3 &= (((-1+(12-24w)w)s + (-4+24w)vc)v)/6, \\ a_4 &= (-((-5+(60-120w)w)vs + (1+(-20+120w)w)c)v)/24, \\ b_0 &= 0, \end{aligned}$$

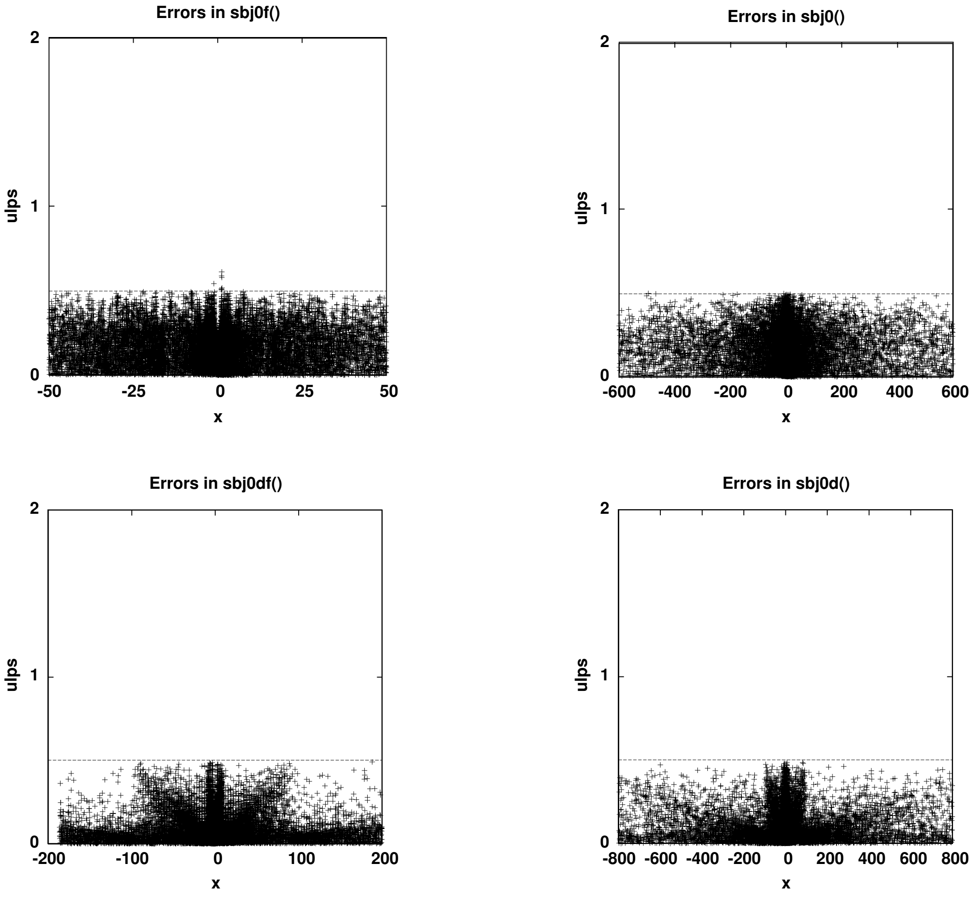


Figure 21.24: Errors in the binary (top) and decimal (bottom) `sbj0(x)` family.

$$
\begin{aligned}
b_1 &= ((2sv + (-1 + 2w)c)v)/1,\\
b_2 &= (-((-1 + 6w)s + (-3 + 6w)vc)v)/2,\\
b_3 &= (((-4 + 24w)vs + (1 + (-12 + 24w)w)c)v)/6,\\
b_4 &= (-((1 + (-20 + 120w)w)s + (5 + (-60 + 120w)w)vc)v)/24.
\end{aligned}
$$

Our improved code for `sbj1(x)` and `sby1(x)` computes the coefficients up to $k = 17$, and then evaluates the polynomials in Horner form using the `QFMA()` wrapper.

Sign alternations in the Taylor-series expansions may lead to loss of leading digits when the series are summed, and digit loss is also possible in the computation of the individual coefficients a_k and b_k. The maximum size of $|d|$ for which the expansions can be used without digit loss depends on z, and is best determined by high-precision numerical experiment. The two Maple programs do just that, and their output shows that loss is almost always most severe in the sums $a_2d^2 + a_3d^3$ and $b_2d^2 + b_3d^3$. Tables of the first 318 zeros handle $x < 1000$, for which a cutoff of $|d| < 0.003$ suffices for both functions. In practice, a somewhat larger cutoff, such as $|d| < 0.05$, could probably be used, because the terms for $k = 1$ dominate the sums.

As in the improved code for the cylindrical functions, we tabulate the zeros as pair sums of exact high and accurate low parts so that d can be determined to machine precision from $(x - z_{\rm hi}) - z_{\rm lo}$. We could also tabulate the values of the sine and cosine at the zeros, but instead conserve storage at the expense of somewhat longer compute times when $|d|$ is small enough for the series to be used. In addition, we suppress the use of the Taylor expansions entirely when the number of digits in the `hp_t` data type is at least twice that in the `fp_t` data type, because our normal algorithm then has sufficient precision to provide low relative error near the zeros.

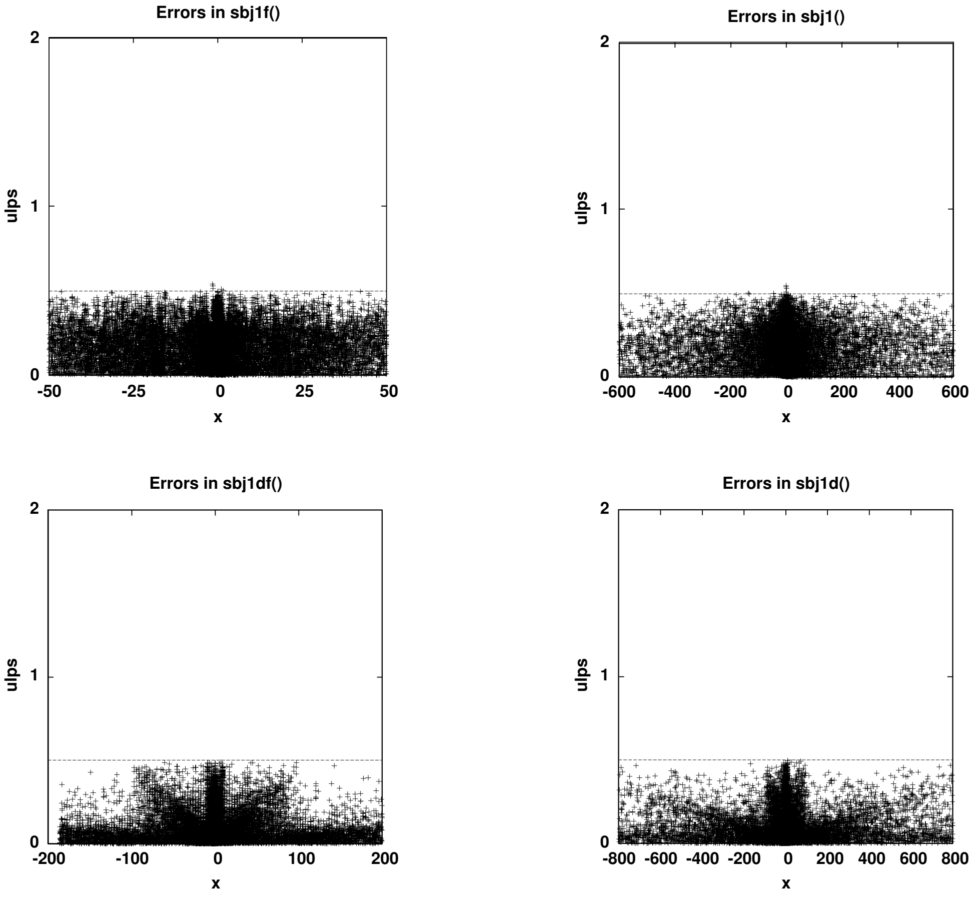


Figure 21.25: Errors in the binary (top) and decimal (bottom) `sbj1(x)` family.

The error plots in **Figure 21.25** for `sbj1(x)`, and in **Figure 21.28** on page 745 for `sby1(x)`, exhibit low relative error because of our use of higher intermediate precision, so we do not show plots from the improvements of this section. Instead, we do a short numerical experiment at the adjacent zeros bracketing the table end, using high-precision values from a symbolic-algebra system for comparison:

```
% hocd128 -lmcw
hocd128> x318 = 1_000.596_260_764_587_333_582_227_925_178_11

hocd128> x319 = 1_003.737_856_546_205_566_637_222_858_984_913

hocd128> sbj1(x318);  0.670873169872646828693521368291249352922e-34
         6.708_731_698_726_468_286_935_213_682_912_493e-35
         6.708_731_698_726_468_286_935_213_682_912_494e-35

hocd128> sbj1(x319); -0.371136263838085458455232524434396532682e-33
         -3.711_362_459_535_284_444_620_381_903_540_337e-34
         -3.711_362_638_380_854_584_552_325_244_343_965e-34

hocd128> x319a = 1_003.737_856_546_206

hocd128> sbj1(x319a); -0.431748747189598548200146130945033677271e-15
         -4.317_487_471_895_985_482_001_461_299_690_095e-16
```

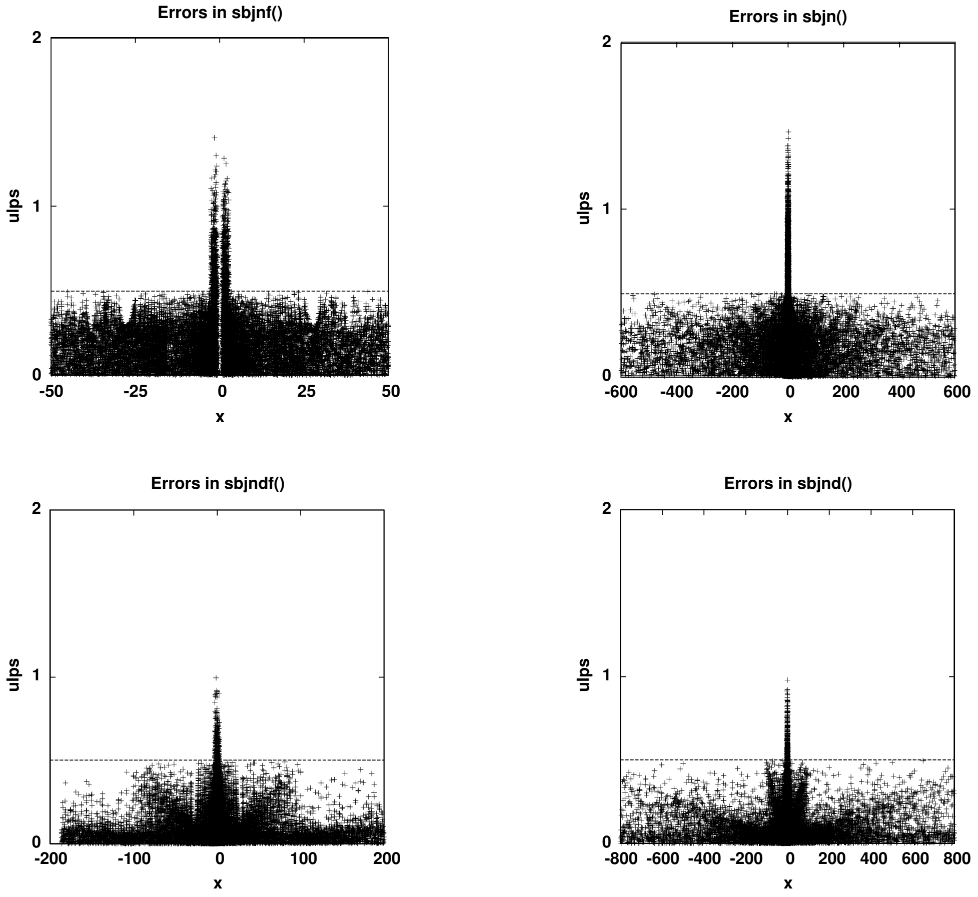


Figure 21.26: Errors in the binary (top) and decimal (bottom) `sbjn(n,x)` family for $n = 25$.

```
-4.317_487_471_895_985_482_001_461_309_450_337e-16
```

In the region covered by the tabulated zeros, we find relative errors of about one ulp or less, but closest to the first zero outside that region, all but seven of the computed digits are wrong. The last experiment shows the improvement when we have a 16-digit approximation to the zero.

21.17 Modified spherical Bessel functions

The unscaled and scaled modified spherical Bessel functions of the first and second kinds are provided by functions with these prototypes:

```
double sbi0  (double);          double sbk0  (double);
double sbi1  (double);          double sbk1  (double);
double sbin  (int, double);     double sbkn  (int, double);
double sbis0 (double);          double sbks0 (double);
double sbis1 (double);          double sbks1 (double);
double sbisn (int, double);     double sbksn (int, double);
```

They have companions with the usual type suffixes for other precisions and bases.

As the function names suggest, the cases $n = 0$ and $n = 1$ receive special treatment. Code for arbitrary n can then use those functions internally.

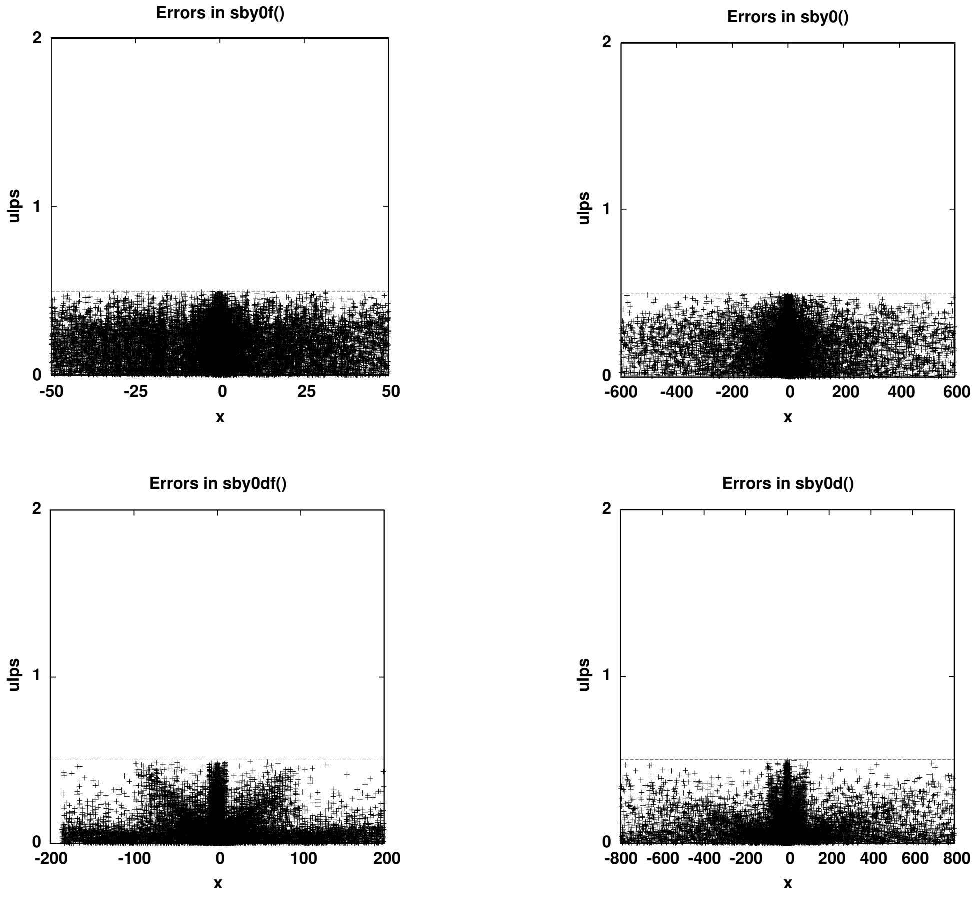


Figure 21.27: Errors in the binary (top) and decimal (bottom) `sby0(x)` family.

21.17.1 Computing $i_0(x)$

To compute the modified function of the first kind of order zero, `sbi0(x)`, a one-time initialization block first determines two cutoffs for Taylor series expansions, and the arguments above which $\sinh(x)$ and $i_0(x)$ overflow. The symmetry relation $i_0(-x) = i_0(x)$ allows us to work only with nonnegative x.

For small arguments, the Taylor-series expansion and its term recurrence looks like this:

$$\begin{aligned}
i_0(x) &= 1 + (1/6)x^2 + (1/120)x^4 + (1/5040)x^6 + (1/362\,880)x^8 + \\
&\qquad (1/39\,916\,800)x^{10} + \cdots, \\
&= t_0 + t_1 + t_2 + \cdots, \\
t_0 &= 1, \qquad t_k = \left(\frac{1}{2k(2k+1)}\right) x^2 t_{k-1}, \qquad \text{for } k = 1, 2, 3, \ldots.
\end{aligned}$$

That series enjoys rapid convergence, and all terms are positive, so there is never subtraction loss. The series is usable for x values larger than one. If $x = 4$, then 24 terms recover 34 decimal digits, and 39 terms produce 70 decimal digits.

The Taylor-series cutoffs are straightforward expressions that determine the value of x for which the k-th series term is smaller than half the rounding error: $c_k x_{\mathrm{TS}}^k < \frac{1}{2}\epsilon/\beta$. Thus, we have $x_{\mathrm{TS}} < \sqrt[k]{\frac{1}{2}\epsilon/(\beta c_k)}$, and we can choose k so that the k-th root needs only square roots or cube roots.

Unlike the implementations of functions described in most other chapters of this book, for the spherical Bessel functions, we use Taylor series over a wider range. For $i_0(x)$, in the four extended IEEE 754 decimal formats, the

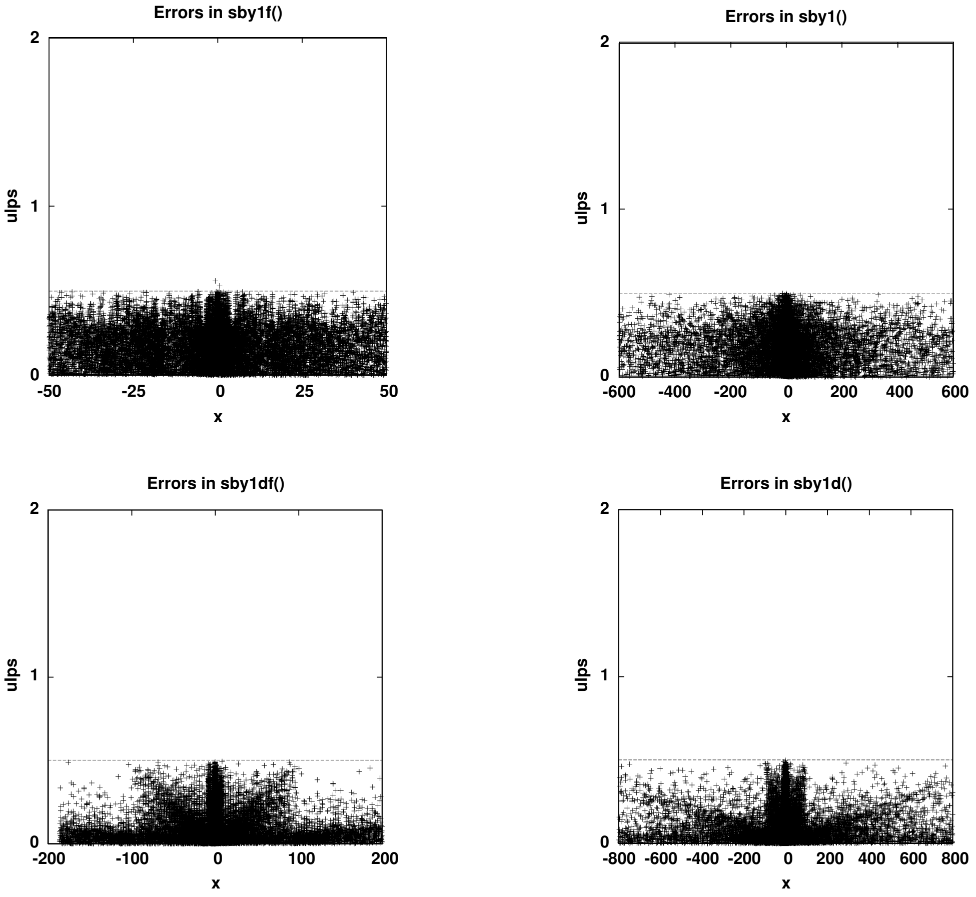


Figure 21.28: Errors in the binary (top) and decimal (bottom) `sby1(x)` family.

cutoffs for a nine-term series are about 2.84, 0.777, 0.0582, and 0.000 328. For $i_1(x)$, the cutoffs are larger, and importantly, for the lowest precision, cover the range where subtraction loss in the closed form of the function is a problem. Having both a short and a long series allows tiny arguments to be handled at lower cost.

For large x, $\sinh(x) \approx \exp(x)/2$, so overflow happens for $x > \log(2) + \log(\texttt{FP\_T\_MAX})$, where the argument of the last logarithm is the largest representable value. However, experiments on various systems show that implementations of the hyperbolic sine in some vendor libraries suffer from premature overflow, so we reduce the cutoff by $1/16$.

For large x, $i_0(x) = \sinh(x)/x \approx \exp(x)/(2x)$, but that does not give a simple way to determine x when the left-hand side is set to `FP_T_MAX`. We can find x by using Newton–Raphson iteration to solve for a root of $f(x) = \log(i_0(x)) - \log(\texttt{FP\_T\_MAX}) \approx x - \log(2x) - \log(\texttt{FP\_T\_MAX})$. Starting with x set to $\log(\texttt{FP\_T\_MAX})$, convergence is rapid, and five iterations produce a solution correct to more than 80 digits.

With the initialization complete, we first handle the special cases of NaN, zero, and $|x|$ above the second overflow cutoff. For small x, we sum three-term or nine-term Taylor series expansions in nested Horner form in order of increasing term magnitudes. For x below the first overflow cutoff, we use the hyperbolic sine formula. For x between the two overflow cutoffs, we can replace $\sinh(x)$ by $\exp(x)/2$, but we need to proceed carefully to avoid premature overflow from the exponential. Preprocessor conditionals select base-specific code for $\beta = 2, 8, 10$, and 16. For example, for decimal arithmetic, we compute $\exp(x)/(2x) = \exp(x/10)^{10}/(2x)$ like this:

```
volatile fp_t t;
fp_t u, u2, u4;
```

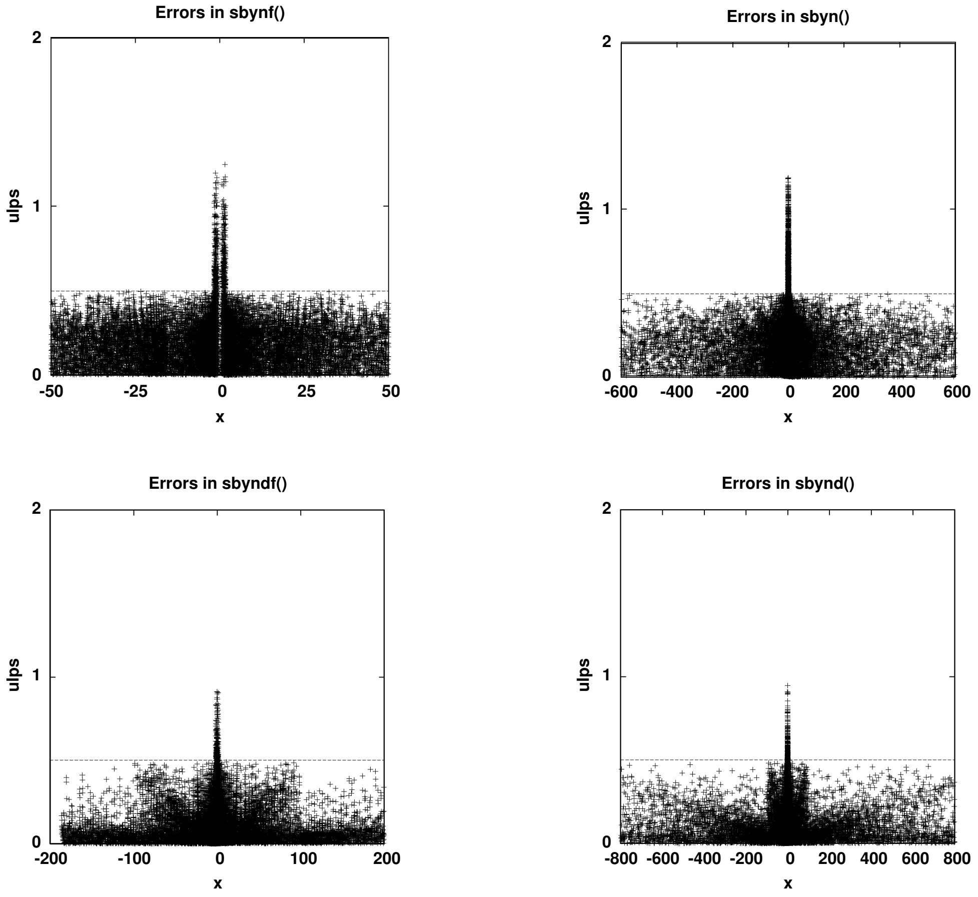


Figure 21.29: Errors in the binary (top) and decimal (bottom) `sbyn(n,x)` family for $n = 25$.

```
u = EXP(x * FP(0.1));   /* exact argument scaling */
u2 = u * u;
u4 = u2 * u2;
t = HALF * u2 / x;
STORE(&t);
result = t * u4 * u4;
```

Because of the error magnification of the exponential, it is imperative to scale its argument exactly. The `volatile` keyword and the `STORE()` macro force intermediate expression evaluation to prevent an optimizing compiler from delaying the division by x until the products have been computed, and overflowed. Avoidance of premature overflow therefore costs at least *seven* rounding errors in a decimal base (or *four* when $\beta = 2$), but that happens only near the overflow limit.

Figure 21.30 on the facing page shows the measured accuracy in two of the functions for computing $i_0(x)$. Plots for their companions are similar, and thus, not shown.

21.17.2 Computing $\mathrm{is}_0(x)$

The scaled modified spherical Bessel function of order zero is defined by

$$\begin{aligned}\mathrm{is}_0(x) &= \exp(-|x|)i_0(x)\\ &= \exp(-|x|)\sinh(x)/x\end{aligned}$$

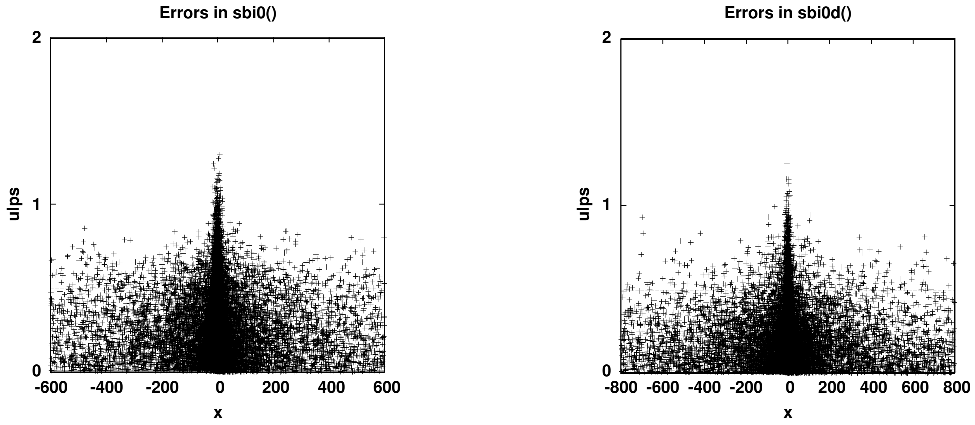


Figure 21.30: Errors in the binary (left) and decimal (right) `sbi0(x)` family.

$$= (1 - \exp(-2|x|))/(2|x|).$$

For $|x|$ in $[0, -\frac{1}{2}\log(\frac{1}{2})] \approx [0, 0.347]$, there is subtraction loss in the numerator. We could handle that by using a polynomial approximation in that region, but we have already done so elsewhere in the library, because we can rewrite the function as $\mathrm{is}_0(|x|) = -\,\mathrm{expm1}(-2|x|)/(2|x|)$.

For small arguments, the Taylor series is useful:

$$\mathrm{is}_0(x) = 1 - x + (2/3)x^2 - (1/3)x^3 + (2/15)x^4 - (2/45)x^5 + (4/315)x^6 - \\ (1/315)x^7 + (2/2835)x^8 - (2/14\,175)x^9 + \cdots .$$

There is no loss of leading bits if we use it only for $|x| < 1/2$.

The series terms can be computed with this recurrence:

$$t_0 = 1, \qquad t_k = -\left(\frac{2}{k+1}\right) x t_{k-1}, \qquad \textit{for } k = 1, 2, 3, \ldots,$$
$$\mathrm{is}_0(x) = t_0 + t_1 + t_2 + \cdots .$$

For sufficiently large $|x|$, the exponential function is negligible, and the function reduces to $1/(2|x|)$. That happens for $|x|$ in $[-\frac{1}{2}\log(\frac{1}{2}\epsilon/\beta), \infty)$. In IEEE 754 arithmetic, the lower limit is about 8.664 in the 32-bit format, and 39.510 in the 128-bit format. Thus, over most of the floating-point range, we do not even need to call an exponential function.

The code in `sbis0x.h` has a one-time initialization block that computes two Taylor-series cutoffs, the subtraction-loss cutoff, and the upper cutoff where the exponential function can be omitted. It then handles the case of NaN and zero arguments. Otherwise, it records whether x is negative, and forces it positive, and then splits the computation into five regions: a four-term Taylor series, a nine-term Taylor series, the subtraction loss region, the nonneglible exponential region, and the final region for large x where the result is just $1/(2x)$. The final result is then negated if x was negative on entry to the function.

There are subtle dependencies on the base in the handling of the denominator $2x$. In the loss region, for $\beta \neq 2$, compute the result as $-\frac{1}{2}\,\mathrm{expm1}(-(x+x))/x$ to avoid introducing an unnecessary rounding error in the denominator. In the exponential region, the result is $\frac{1}{2}((1 - \exp(-(x+x)))/x)$ when $\beta \neq 2$. In the overflow region, the result is $\frac{1}{2}x$.

Figure 21.31 on the next page shows the measured accuracy in two of the functions for computing $\mathrm{is}_0(x)$.

21.17.3 Computing $i_1(x)$

The code for the spherical Bessel function $i_1(x) = (\cosh(x) - \sinh(x)/x)/x$ begins with a one-time initialization block that computes two Taylor series cutoffs, and two overflow cutoffs where the hyperbolic functions overflow,

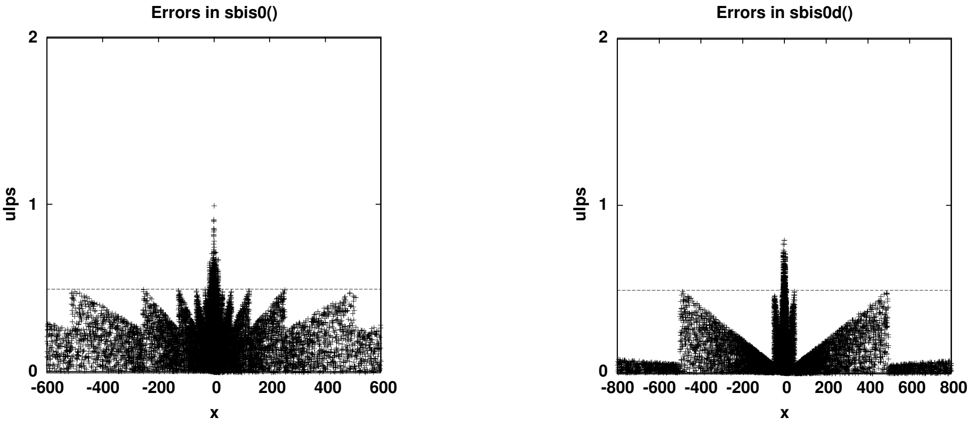


Figure 21.31: Errors in the binary (left) and decimal (right) `sbis0(x)` family.

and where $i_1(x)$ itself overflows. As in the code for $i_0(x)$, Newton–Raphson iteration is needed to find the second cutoff.

The code is easy for arguments that are NaN, zero, or have magnitudes beyond the second overflow cutoff. Otherwise, if x is negative, the symmetry relation $i_1(-x) = -i_1(x)$ allows x to be forced positive, as long as we remember to negate the final result. For small x, we sum three-term or nine-term Taylor series in Horner form.

For x between the second Taylor series cutoff and the first overflow limit, we could evaluate the hyperbolic functions that define $i_1(x)$. However, there is subtraction loss in the numerator for x in $[0, 1.915]$, and all digits can be lost when x is small. There does not appear to be a simple alternative expression for $i_1(x)$ that avoids the loss, and requires only functions that we already have. We therefore have two choices: sum the general Taylor series until it converges, or use a polynomial approximation.

The Taylor-series expansion and its term recurrence looks like this:

$$\begin{aligned}
i_1(x) &= (1/3)x + (1/30)x^3 + (1/840)x^5 + (1/45\,360)x^7 + \cdots \\
&= \sum_{k=1}^{\infty} \frac{2k}{(2k+1)!} x^{2k-1}, \\
&= (x/3)(t_1 + t_2 + t_3 + + \cdots) \\
t_1 = 1, \qquad & t_{k+1} = \left(\frac{1}{2k(2k+3)}\right) x^2 t_k, \qquad k = 1, 2, 3, \ldots.
\end{aligned}$$

All terms have the same sign, so no subtraction loss is possible. It is best to start with $k = 2$ and sum the series until it has converged, and then add the first term. Convergence is slowest for the largest x, and numerical experiments show that we need at most 7, 12, 13, 19, and 32 terms for the five binary extended IEEE 754 formats. Accumulation of each term costs two adds, two multiplies, and one divide. However, if we store a precomputed table of values of $1/(2k(2k+3))$, the sum costs only one add and two multiplies per term.

In most cases, only the last two rounding errors affect the computed function value. In binary arithmetic, one of those errors can be removed by rewriting $x/3$ as $x/4 + x/12$, and then moving the term $x/12$ into the sum of the remaining terms. That sum is then added to the *exact* value $x/4$.

For the polynomial-fit alternative, we can compute the Bessel function as

$$\begin{aligned}
i_1(x) &\approx x/3 + x^3 \mathcal{R}(x^2) \\
&\approx x/4 + (x/12 + x^3 \mathcal{R}(x^2)), \\
\mathcal{R}(x) &= (i_1(\sqrt{x}) - \sqrt{x}/3)/\sqrt{x^3}, \qquad \textit{fit to rational polynomial.}
\end{aligned}$$

For x on $[0, 1]$, the term ratio $x^3\mathcal{R}(x^2)/(x/3)$ lies on $[0, 0.103]$, so the polynomial adds at least one decimal digit of precision. For x on $[1, 1.915]$, that ratio reaches 0.418.

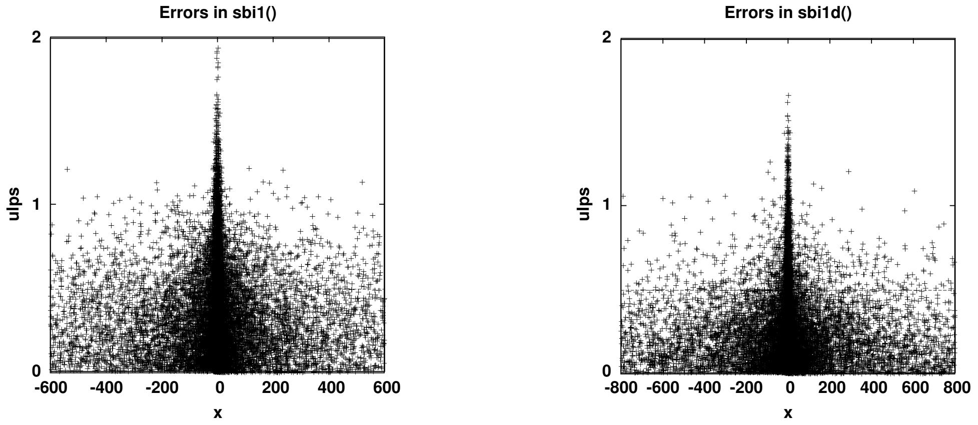


Figure 21.32: Errors in the binary (left) and decimal (right) `sbi1(x)` family.

Numerical experiments in Maple show that rational polynomial fits of orders ⟨2/1⟩, ⟨4/3⟩, ⟨4/4⟩, ⟨7/6⟩, and ⟨12/12⟩ are needed for the five binary extended IEEE 754 formats. Those fits require less than a quarter of the operation counts of Taylor-series sums for the same accuracy. Consequently, rational polynomials are the default evaluation method in `sbi1(x)`, although the others can be selected at compile time by defining preprocessor symbols.

Above the interval $[0, 1.915]$, we can safely use the hyperbolic functions, until we reach the region between the two overflow cutoffs, where we have $i_1(x) \approx \exp(x)(1-1/x)/(2x) = (\exp(x)/(2x^2))(x-1)$. Having a single function that computes both `sinh(x)` and `cosh(x)` simultaneously is clearly useful here, and our library supplies `sinhcosh(x,psh,pch)` for that purpose. As we did for $i_0(x)$, to avoid premature overflow in the exponential, expand it as $\exp(x/\beta)^\beta$, and include the factor $1/(2x)$ early in the product of the factors. For example, for a hexadecimal base, the code looks like this:

```
volatile fp_t t;
fp_t u, u2, u4, v;

u = EXP(x * FP(0.0625));    /* exact argument scaling */
u2 = u * u;
u4 = u2 * u2;
t = HALF * u4 / (x * x);
STORE(&t);
v = t * u4 * u4 * u4;
result = v * (x - ONE);
```

Avoidance of premature overflow costs *ten* rounding errors (or *six* when $\beta = 2$), but only near the overflow limit.

Figure 21.32 shows the measured accuracy in two of the functions for computing $i_1(x)$.

21.17.4 Computing $\mathrm{is}_1(x)$

The scaled modified spherical Bessel function of order one has the symmetry relation $\mathrm{is}_1(-x) = -\mathrm{is}_1(x)$, so we henceforth assume that x is nonnegative, and we set a flag to negate the final result if the argument is negative. The function is then defined by

$$\begin{aligned}
\mathrm{is}_1(x) &= \exp(-x) i_1(x), \qquad \text{for } x \ge 0, \\
&= \exp(-x)(\cosh(x) - \sinh(x)/x)/x \\
&= \exp(-x)((\exp(x) + \exp(-x)) - (\exp(x) - \exp(-x))/x)/(2x) \\
&= (1 + \exp(-2x) - (1 - \exp(-2x))/x)/(2x)
\end{aligned}$$

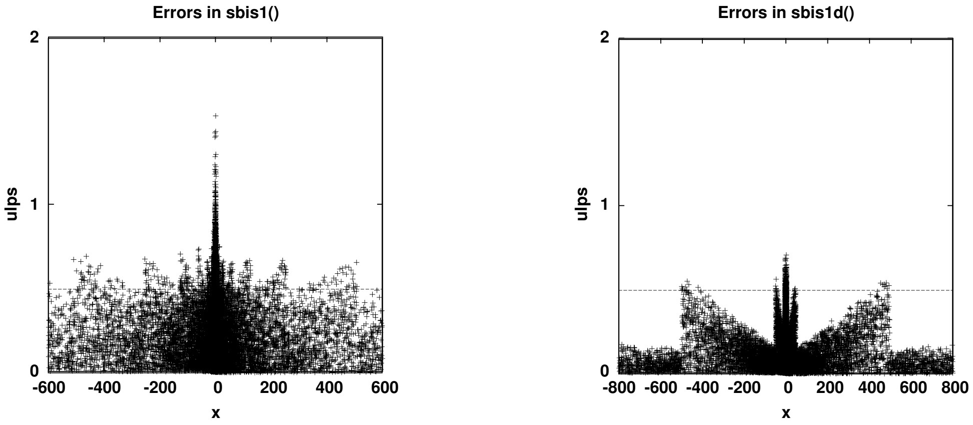


Figure 21.33: Errors in the binary (left) and decimal (right) `sbis1(x)` family.

$$= ((1 - 1/x) + (1 + 1/x)\exp(-2x))/(2x).$$

As we observed in Section 21.17.3 on page 747, there is subtraction loss in the hyperbolic form of the numerator for x in $[0, 1.915]$, and that region is best handled by summing a Taylor series, or with a polynomial approximation.

The Taylor series for $\mathrm{is}_1(x)$, and its term recurrence relation, look like this:

$$\begin{aligned}
\mathrm{is}_1(x) &= (x/3)(1 - x + (3/5)x^2 - (4/15)x^3 + (2/21)x^4 - (1/35)x^5 + \\
&\qquad (1/135)x^6 - (8/4725)x^7 + (2/5775)x^8 - (2/31\,185)x^9 + \cdots), \\
&= (x/3)(t_0 + t_1 + t_2 + \cdots), \\
&= (x/4)(t_0 + t_1 + t_2 + \cdots) + (x/12)(t_0 + t_1 + t_2 + \cdots), \qquad \text{when } \beta = 2, \\
t_0 = 1, \qquad & t_k = -\left(\frac{2(k+1)}{k(k+3)}\right) x t_{k-1}, \qquad k = 1, 2, 3, \ldots.
\end{aligned}$$

Unfortunately, that series converges slowly, so its use is limited to $|x| \ll 1$. In the 32-bit IEEE 754 formats, the nine-term series can be used only for $|x| < 0.31$. Higher precisions reduce that cutoff. Just as we did for $i_1(x)$ in a binary base, replacing the inexact $x/3$ factor by $x/4 + x/12$ reduces the outer rounding error.

In the exponential form, the second term is almost negligible for x above the value for which $\exp(-2x)$ is smaller than the rounding error $\frac{1}{2}\epsilon/\beta$. That cutoff is then $-\frac{1}{2}\log(\frac{1}{2}\epsilon/\beta)$. However, that is a slight underestimate, and a better choice that works in all IEEE 754 formats is larger by $1/8$. We could, of course, use Newton–Raphson iteration to find the precise cutoff value, but a simple increment by $1/8$ is easier, and of little significance for later computation.

The code in `sbis1x.h` first handles NaN and zero arguments. Otherwise, it enforces the symmetry relation by forcing x to be positive with a negation flag set for later use, and then splits the computation into five regions: three-term or nine-term Taylor series for small x, a polynomial approximation in the loss region, the exponential form below the upper cutoff, and above that cutoff, simply $(\frac{1}{2} - (\frac{1}{2})/x)/x$. In the last region, to reduce rounding error, multiplication by the reciprocal of x should be avoided.

Figure 21.33 shows the measured accuracy in two of the functions for computing $\mathrm{is}_1(x)$.

21.17.5 Computing $i_n(x)$

To find the unscaled modified spherical Bessel functions of arbitrary order, `sbin(n,x)`, we often need to use the recurrence relation when $n > 1$. As we observed in Section 21.13 on page 728, stable computation of the $i_n(x)$ Bessel functions requires using the continued-fraction form to find the ratio $i_n(x)/i_{n-1}(x)$, and then downward recurrence to find $i_n(x)/i_0(x)$, from which the function value can be found by multiplying the ratio by the result returned by `sbi0(x)`.

For small arguments, it is desirable for speed and accuracy to sum a series, but Maple is unable to find one for arbitrary n. However, Mathematica is successful:

```
% math
In[1]:= sbin = Function[{n, x}, BesselI[n + 1/2,x] * Sqrt[Pi/(2*x)]];

In[2]:= Simplify[Series[sbin[n,x], {x, 0, 6}]]

             -1 - n              -2 - n           2
         n  2       Sqrt[Pi]    2       Sqrt[Pi] x
Out[2]= x  (---------------- + ---------------------- +
                    3                          3
              Gamma[- + n]     (3 + 2 n) Gamma[- + n]
                    2                          2

              -4 - n          4
             2       Sqrt[Pi] x
>      ------------------------------- +
                       2        3
       (15 + 16 n + 4 n ) Gamma[- + n]
                                2

                    -5 - n          6
                   2       Sqrt[Pi] x                    7
>      ------------------------------------------- + O[x] )
                            2      3        3
       3 (105 + 142 n + 60 n  + 8 n ) Gamma[- + n]
                                            2
```

That expansion looks horrid, but we recognize some common factors on the right-hand side, and try again with those factors moved to the left-hand side:

```
In[3]:= Simplify[Series[sbin[n,x] * 2^(n+1) * Gamma[n + 3/2] /
                        Sqrt[Pi], {x, 0, 6}]]

                   2               4
         n        x               x
Out[3]= x  (1 + ------- + ------------------- +
                6 + 4 n                    2
                          120 + 128 n + 32 n

                      6
                     x                       7
>      ------------------------------- + O[x] )
                             2      3
       48 (105 + 142 n + 60 n  + 8 n )
```

The series coefficients are reciprocals of polynomials in n with integer coefficients. For $n > x^2$, the leading term is the largest, and the left-hand side grows like x^n.

The factor in the left-hand side looks ominous, until we remember that half-integral values of the gamma function have simple values that are integer multiples of $\sqrt{\pi}$. We recall some results from **Section 18.1** on page 522 to find the form of the factor:

$$\begin{aligned}
\Gamma(\tfrac{1}{2}) &= \sqrt{\pi}, \\
\Gamma(n+\tfrac{1}{2}) &= (2n-1)!!\,\Gamma(\tfrac{1}{2})/2^n, && \textit{if } n \ge 0, \\
&= (2n-1)!!\sqrt{\pi}/2^n, \\
\Gamma(n+\tfrac{3}{2}) &= (2(n+1)-1)!!\sqrt{\pi}/2^{n+1} \\
&= (2n+1)!!\sqrt{\pi}/2^{n+1}, \\
\boldsymbol{(2n+1)!!} &= \boldsymbol{2^{n+1}\Gamma(n+\tfrac{3}{2})/\sqrt{\pi}}, && \textit{left-hand side factor.}
\end{aligned}$$

With a larger limit in the series expansion in Mathematica, and application of its polynomial-factorization function, `Factor[]`, we therefore find the first few terms of the Taylor series as

$$\begin{aligned} i_n(x) = \frac{x^n}{(2n+1)!!}(1 + &\frac{1}{2(3+2n)}x^2 + \frac{1}{8(3+2n)(5+2n)}x^4 + \\ &\frac{1}{48(3+2n)(5+2n)(7+2n)}x^6 + \\ &\frac{1}{384(3+2n)(5+2n)(7+2n)(9+2n)}x^8 + \\ &\frac{1}{3840(3+2n)(5+2n)(7+2n)(9+2n)(11+2n)}x^{10} + \\ &\frac{1}{46\,080(3+2n)(5+2n)(7+2n)(9+2n)(11+2n)(13+2n)}x^{12} + \\ &\cdots). \end{aligned}$$

Notice the additional structure that coefficient factorization exposes, and that all terms are positive. Taking ratios of adjacent coefficients shows that the terms of the Taylor series have a delightfully simple recurrence:

$$\begin{aligned} t_0 &= 1, \qquad t_k = \left(\frac{1}{2k(2k+2n+1)}\right) x^2 t_{k-1}, && \textit{for } k = 1, 2, 3, \ldots, \\ i_n(x) &= \frac{x^n}{(2n+1)!!}(t_0 + t_1 + t_2 + t_3 + \cdots). \end{aligned}$$

The general Taylor-series term and its coefficient can be written explicitly like this:

$$\begin{aligned} t_k &= c_k x^{2k}, && \textit{for } k = 0, 1, 2, \ldots, \\ c_0 &= 1, \\ c_k &= 1/\prod_{j=1}^{k}(2j(2n+1+2j)), && \textit{for } k = 1, 2, 3, \ldots, \\ &= 1/\bigl(2^k\, k!\, \prod_{j=1}^{k}(2n+1+2j)\bigr) \\ &= 1/\bigl(2^{2k}\, k!\, \prod_{j=1}^{k}(n+\tfrac{1}{2}+j)\bigr) \\ &= 1/\bigl(2^{2k}\, k!\, (n+\tfrac{3}{2})_k\bigr), && \textit{see text for this notation.} \end{aligned}$$

In the last equation, the notation $(a)_k$ stands for the product $a \times (a+1) \times (a+2) \times \cdots \times (a+k-1) = \Gamma(a+k)/\Gamma(a)$. It is called the *rising factorial function*, or sometimes, the *Pochhammer symbol*. Maple provides it as the function `pochhammer(a,k)`, and Mathematica as the function `Pochhammer[a,k]`.

Normally, we apply the Taylor series only for small x, so that only a few terms are needed to reach a given accuracy. However, here we can see that the series also needs only a few terms if $n > x^2$, because the denominator of the k-th term is larger than $2^{2k}k!$, effectively providing more than $2k$ additional bits to the sum. That is an important observation, because our other computational route to $i_n(x)$ involves a recurrence whose execution time is proportional to n, plus the separate computation of $i_0(x)$. To illustrate how well the series converges, Table 21.8 on the next page shows the accuracy obtainable with modest numbers of terms for various choices of n and x. The limited exponent range of most floating-point systems means that we usually cannot compute $i_n(x)$ for n values as large as those shown in that table before the function overflows.

With care, we can use the Taylor series for larger values of x than permitted by the condition $n > x^2$. If we are prepared to sum up to k terms, and if $x^2 > n$, then the first few terms grow, but eventually they get smaller because of the rapid growth of the denominator. If term k is smaller than the rounding error compared to the first term, then the sum has surely converged to machine precision, so we have the requirements that

$$t_k = c_k x^{2k} < \epsilon/(2\beta), \qquad x^2 < \sqrt[k]{\epsilon/(2\beta c_k)}, \qquad x^2/n < \sqrt[k]{2^{2k}k!\epsilon/(2\beta)}.$$

Table 21.8: Convergence of the Taylor series of $i_n(x)$, showing the relative error in a sum of N terms.

n	relerr	n	relerr	n	relerr
		$x = 1$			
$N = 10$		$N = 20$		$N = 30$	
10	2.82e-25	10	3.02e-57	10	7.47e-93
100	1.47e-33	100	4.76e-72	100	4.14e-113
1000	2.48e-43	1000	3.00e-91	1000	2.03e-141
10 000	2.61e-53	10 000	3.66e-111	10 000	3.12e-171
		$x = 10$			
$N = 10$		$N = 20$		$N = 30$	
10	2.82e-05	10	3.02e-17	10	7.47e-33
100	1.47e-13	100	4.76e-32	100	4.14e-53
1000	2.48e-23	1000	3.00e-51	1000	2.03e-81
10 000	2.61e-33	10 000	3.66e-71	10 000	3.12e-111
		$x = 100$			
$N = 10$		$N = 20$		$N = 30$	
1000	0.00248	1000	3.00e-11	1000	2.03e-21
10 000	2.61e-13	10 000	3.66e-31	10 000	3.12e-51
100 000	2.63e-23	100 000	3.73e-51	100 000	3.25e-81
1 000 000	2.63e-33	1 000 000	3.74e-71	1 000 000	3.27e-111

The last inequality follows from the replacement in c_k of $(n + \frac{3}{2})_k$ by the *smaller* value n^k. For a large fixed k, computation of the right-hand requires rewriting it in terms of logarithms and an exponential to avoid premature overflow. However, it gives us a scale factor, s, that needs to be computed only once for the chosen limit on k, and that can then be used to determine how large n can be compared to x^2 to use the series.

When $x^2 > n$, some term after the first is the largest, and because each term suffers four rounding errors, those errors can have a large affect on the computed sum. One solution would be to accumulate the sum in higher precision. Alternatively, we can just reduce the scale factor to force a switch to an alternative algorithm for smaller values of x, and that is what we do in the code in `sbinx.h`.

After a one-time initialization block to compute a Taylor-series cutoff and the limit on x^2/n, the code in `sbinx.h` for computing $i_n(x)$ has several blocks. First, there are checks for x is a NaN, Infinity, or zero, then checks for $n = 0$ or $n = 1$. Otherwise, we record a negation flag that tells us whether x is negative and n is even, and then we force x to be positive. The computation is then split into four regions, the first where $n < 0$ and the downward recurrence is stable, the second where the four-term Taylor series can be summed, the third where the general Taylor series is effective because $x^2/n < s$, and the last where the Lentz algorithm evaluates the continued-fraction ratio $i_n(x)/i_{n-1}(x)$, then downward recurrence is used to find $i_n(x)/i_0(x)$, and the magnitude of the final result is obtained by multiplying that ratio by `sbi0(x)`. If the negation flag is set, the last result must be negated.

Figure 21.34 on the following page shows the measured accuracy in two of the functions for computing $i_{25}(x)$. The extended vertical range is needed to show how numerical errors increase in the recurrence.

21.17.6 Computing $\mathrm{is}_n(x)$

The scaled modified spherical Bessel functions of arbitrary order, implemented in the function `sbisn(n,x)`, are defined as $\mathrm{is}_n(x) = \exp(-|x|)i_n(x)$, and satisfy the argument symmetry relation $\mathrm{is}_n(-x) = (-1)^n \mathrm{is}_n(x)$. The exponential scaling damps the growth of $i_n(x)$, making $\mathrm{is}_n(x)$ representable over more of the floating-point range. For example, $i_{10}(1000) \approx 10^{431}$, but $\mathrm{is}_{10}(1000) \approx 10^{-4}$.

As happens with the other spherical Bessel functions of order n, Maple is unable to find a Taylor series expansion of the scaled functions for arbitrary n, but Mathematica can do so, and we can display its results like this:

$$\begin{aligned}\mathrm{is}_n(x) = \frac{x^n}{(2n+1)!!}(1 - x + {} \\ \frac{2+n}{3+2n}x^2 - \frac{1}{3}\frac{3+n}{3+2n}x^3 + {}\end{aligned}$$

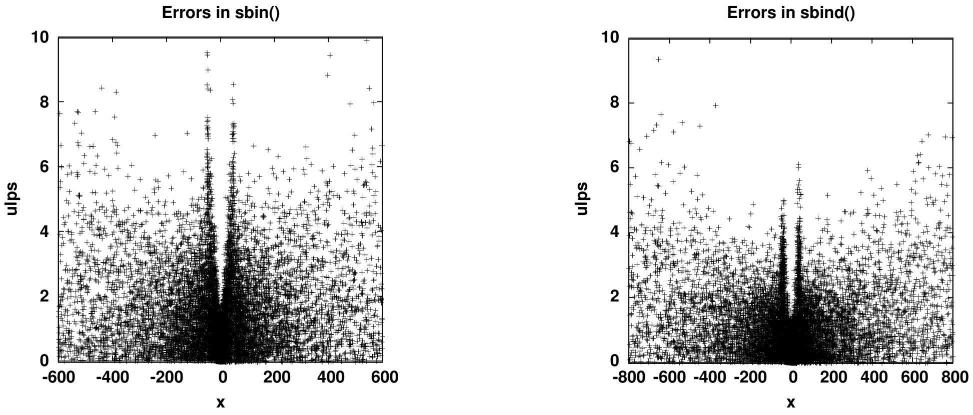


Figure 21.34: Errors in the binary (left) and decimal (right) `sbin(n,x)` family for $n = 25$.

$$\begin{aligned}
&\frac{1}{6}\frac{(3+n)(4+n)}{(3+2n)(5+2n)}x^4 - \frac{1}{30}\frac{(4+n)(5+n)}{(3+2n)(5+2n)}x^5 + \\
&\frac{1}{90}\frac{(4+n)(5+n)(6+n)}{(3+2n)(5+2n)(7+2n)}x^6 - \\
&\frac{1}{630}\frac{(5+n)(6+n)(7+n)}{(3+2n)(5+2n)(7+2n)}x^7 + \\
&\cdots) \\
= {} & \frac{x^n}{(2n+1)!!}(t_0 + t_1 + t_2 + \cdots).
\end{aligned}$$

Successive terms can be produced with this recurrence:

$$t_0 = 1, \qquad t_k = -\frac{2}{k}\left(\frac{k+n}{k+1+2n}\right)x t_{k-1}, \qquad k = 1, 2, 3, \ldots.$$

Convergence is slower than that of the series for $i_n(x)$, and because the signs alternate, subtraction loss is a problem unless $|x| < \frac{1}{2}$. Nevertheless, for $x = \frac{1}{2}$ and $n = 1$, only 10, 17, 20, 31, and 53 terms are required for the five binary extended IEEE 754 formats, and convergence is faster for smaller $|x|$ or larger n.

The computational approach in `sbisnx.h` is similar to that in `sbinx.h`: a one-time initialization block to compute a Taylor series cutoff, special handling when x is NaN, zero, or Infinity, or $n = 0$ or $n = 1$, downward recurrence if $n < 0$, a four-term Taylor series for small x, a general Taylor series for $|x| < \frac{1}{2}$, and otherwise, the Lentz algorithm for the continued fraction, downward recurrence, and then normalization by `sbis0(x)`.

Figure 21.35 on the next page shows the measured accuracy in two of the functions for computing $\mathrm{is}_{25}(x)$.

21.17.7 Computing $k_n(x)$ and $\mathrm{ks}_n(x)$

Because the modified spherical Bessel function of the second kind for order n, $k_n(x)$, is the product of $(\pi/(2x^{n+1}))\exp(-x)$ and a polynomial of order n in x with positive integer coefficients, its upward recurrence relation is certainly stable for positive arguments. There *is* subtraction loss for negative arguments, but the error is always small compared to the dominant constant term in the polynomial factor, so in practice, the computation is also stable for negative arguments.

The two lowest-order unscaled functions have these Taylor-series expansions:

$$\begin{aligned}
k_0(x) = (\pi/(2x))(1 - x + (1/2)x^2 - (1/6)x^3 + (1/24)x^4 - {}\\
(1/120)x^5 + (1/720)x^6 - (1/5040)x^7 + \cdots),
\end{aligned}$$

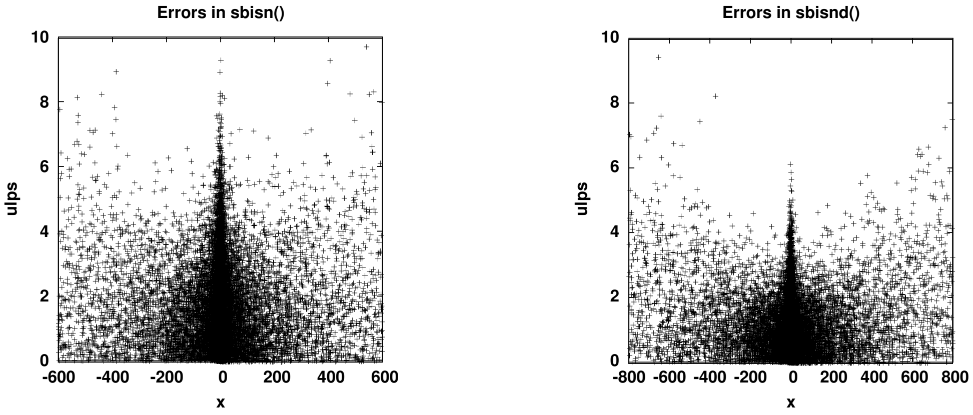


Figure 21.35: Errors in the binary (left) and decimal (right) `sbisn(n,x)` family for $n = 25$.

$$k_1(x) = (\pi/(2x^2))(1 - (1/2)x + (1/3)x^2 - (1/8)x^3 + (1/30)x^4 - \\ (1/144)x^5 + (1/840)x^6 - (1/5760)x^7 + \cdots).$$

Leading bit loss in their sums is avoided if we choose the cutoffs $x_{\mathrm{TS}} = \frac{1}{2}$ for $k_0(x)$ and $x_{\mathrm{TS}} = \frac{3}{4}$ for $k_1(x)$.

The series for the scaled functions are just the polynomials given in **Table 21.6** on page 738.

For large x, the value of $k_n(x)$ approaches $(\pi/(2x))\exp(-x)$, so the only significant concern in its computation is the optimization of avoiding a call to the exponential when x is large enough that $k_n(x)$ underflows to zero. Because that value of x is independent of n, it is a cutoff that we can compute in a one-time initialization block. The scaled companion, $\mathrm{ks}_n(x) = \exp(x)k_n(x)$, needs no exponential, and thus, no cutoff test.

Code for the cases $n = 0$ and $n = 1$ is implemented in the files `sbk0x.h`, `sbks0x.h`, `sbk1x.h`, and `sbks1x.h`. The functions contain a one-time initialization block to compute overflow and underflow cutoffs. They then check for the special cases of x is negative, a NaN, or small enough to cause overflow. The unscaled functions also check whether x is above the underflow cutoff. Otherwise, the functions are simply computed from their definitions, but taking care to represent $\pi/2$ as $2(\pi/4)$ in any base with wobbling precision.

The files `sbknx.h` and `sbksnx.h` contain the code for arbitrary n, and differ only in their references to the unscaled or scaled functions of orders 0 and 1. Once n and x have been determined to be other than one of the special cases, upward recurrence starting from function values for $n = 0$ and $n = 1$ finds the result in at most n steps. The loop test at the start of iteration k includes the expected test $k < n$, and a second test that the function value is still nonzero, so that early loop exit is possible once the underflow region has been reached.

Figure 21.36 on the next page through **Figure 21.41** on page 757 show the measured accuracy in two of the functions for computing the modified spherical Bessel functions of the second kind.

21.18 Software for Bessel-function sequences

Some applications of Bessel functions require their values for a fixed argument x, and a consecutive sequence of integer orders. The existence of three-term recurrence relations suggests that, at least for some argument ranges, it should be possible to generate members of a Bessel function sequence for little more than the cost of computing two adjacent elements. Many of the software implementations published in the physics literature cited in the introduction to this chapter produce such sequences, but the POSIX specification of the Bessel functions provides only for single function values of $J_n(x)$ and $Y_n(x)$.

For the `mathcw` library, we implement several families of functions that return a vector of $n + 1$ Bessel function values for a fixed argument x, starting from order zero:

```
void vbi  (int n, double result[], double x); /* I(0..n,x) */
void vbis (int n, double result[], double x); /* I(0..n,x)*exp(-|x|) */
```

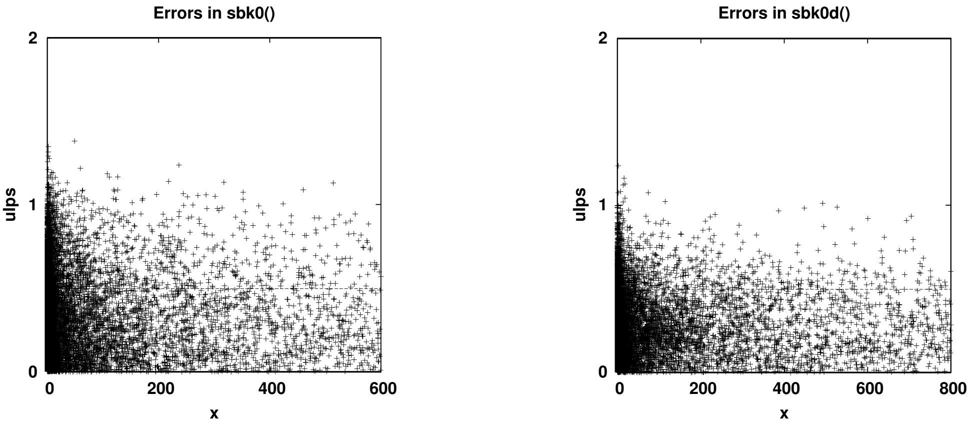


Figure 21.36: Errors in the binary (left) and decimal (right) `sbk0(x)` family.

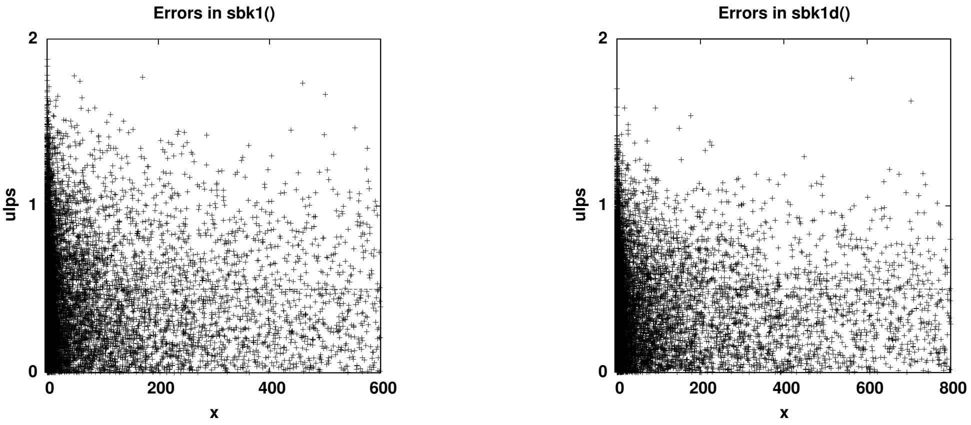


Figure 21.37: Errors in the binary (left) and decimal (right) `sbk1(x)` family.

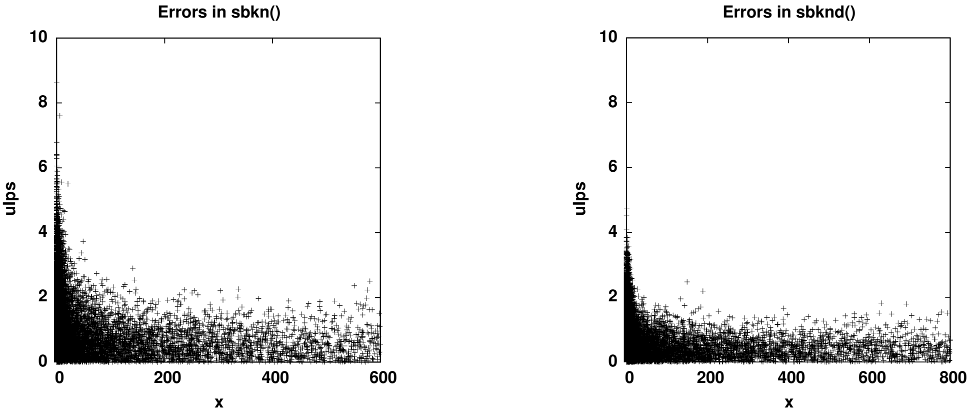


Figure 21.38: Errors in the binary (left) and decimal (right) `sbkn(n,x)` family for $n = 25$.

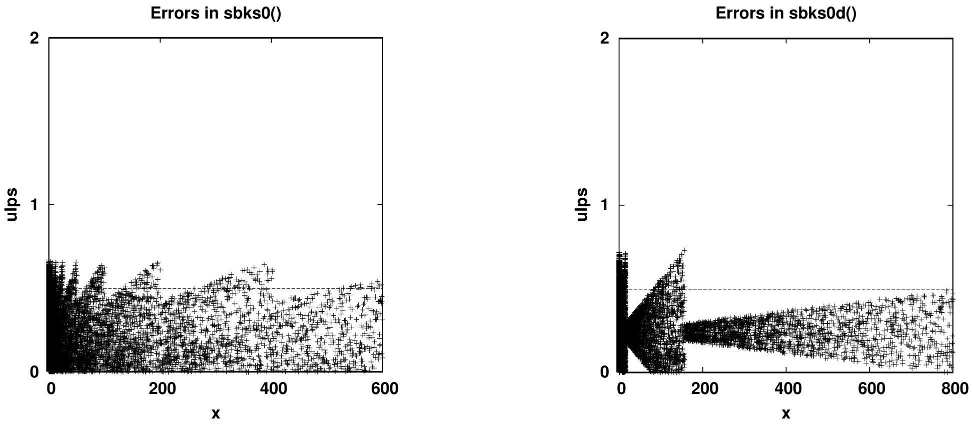


Figure 21.39: Errors in the binary (left) and decimal (right) `sbks0(x)` family.

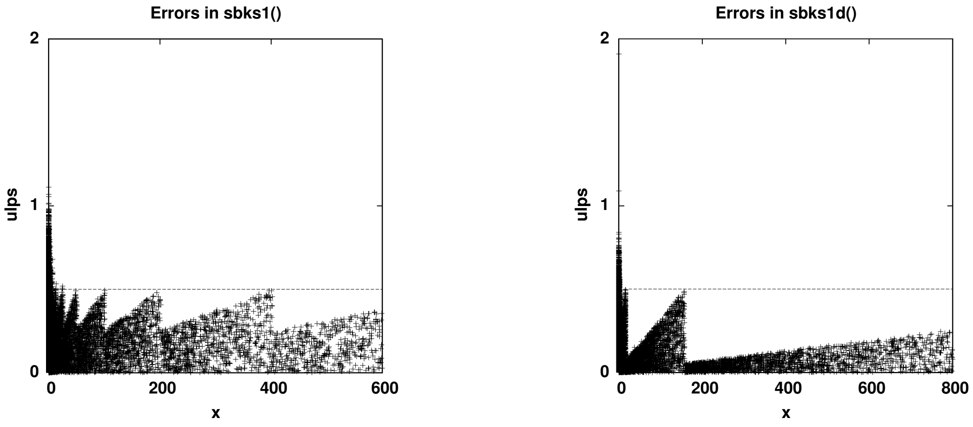


Figure 21.40: Errors in the binary (left) and decimal (right) `sbks1(x)` family.

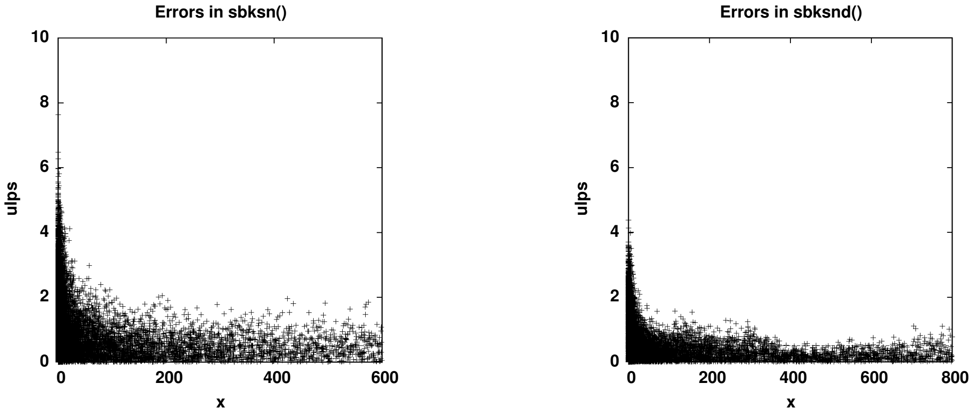


Figure 21.41: Errors in the binary (left) and decimal (right) `sbksn(n,x)` family for $n = 25$.

```
void vbj  (int n, double result[], double x); /* J(0..n,x) */
void vbk  (int n, double result[], double x); /* K(0..n,x) */
void vbks (int n, double result[], double x); /* K(0..n,x) * exp(x) */
void vby  (int n, double result[], double x); /* Y(0..n,x) */

void vsbi (int n, double result[], double x); /* i(0..n,x) */
void vsbis(int n, double result[], double x); /* i(0..n,x)*exp(-|x|) */
void vsbj (int n, double result[], double x); /* j(0..n,x) */
void vsbk (int n, double result[], double x); /* k(0..n,x) */
void vsbks(int n, double result[], double x); /* k(0..n,x) * exp(x) */
void vsby (int n, double result[], double x); /* y(0..n,x) */
```

Each of those functions has companions with the usual precision suffices. For example, after a call to the decimal_double function vsbyd(n,result,x), the array result[] contains the ordinary spherical Bessel function values $y_0(x), y_1(x), \ldots, y_n(x)$.

The vector functions follow a common design:

- Check for the condition $n < 0$, indicating an empty output array, in which case, set errno to ERANGE, and return without referencing the array. There is no provision for arrays of negative orders, even though such orders are well defined for all of the functions.

- Check for the special cases of x is a NaN, Infinity, and zero and handle them quickly, taking care to preserve symmetry properties for negative arguments, and handle signed zeros properly. Within a single case, the elements of the returned arrays have identical magnitudes, but may differ in sign.

- For functions that have complex values on the negative axis, if $x < 0$, set the array elements to the quiet NaN returned by QNAN(""), and set the global variable errno to EDOM.

- Handle the special cases of $n = 0$ and $n = 1$ by calling the corresponding scalar Bessel functions. We can then later assume that $n > 1$, and avoid bounds checks on storage into the output array.

- For small arguments, when the general Taylor series is known and has simple term recurrences, use it to compute all array entries. That is easily done by an outer loop over orders $m = 0, 1, 2, \ldots, n$, and an inner loop that sums series terms after the first to machine precision, after which the first term, and any outer scale factor, are incorporated. The last step may require special handling for systems that have wobbling precision.

  If there is an outer scale factor, such as $x^m/(2n+1)!!$, update it. If that factor overflows or underflows, the remaining array elements can be quickly supplied, and the outer loop exited early.

  To illustrate that description, here is the code block for the Taylor region for $I_n(x)$ in the file vbix.h:

  ```
  else if (QABS(x) < FIVE)    /* x in (0,5): use Taylor series */
  {
      fp_t f_m, scale, v, w;
      int m;

      v = x * HALF;
      w = v * v;
      scale = ONE;            /* k = 0: scale = v**m / m! */

      for (m = 0, f_m = ZERO; m <= n; ++m, ++f_m)
      {
          fp_t f_k, sum, t_k, u;
          int k;
          static const int MAX_TERMS = 43; /* enough for 70D */

          sum = ZERO;
          t_k = ONE;

          for (k = 1, f_k = ONE; k <= MAX_TERMS; ++k, ++f_k)
  ```

```
          {                   /* form sum = t_1 + t_2 + ... */
              fp_t new_sum;

              t_k *= w / (f_k * (f_k + f_m));
              new_sum = sum + t_k;

              if (new_sum == sum)
                  break;      /* converged: early loop exit */

              sum = new_sum;
          }

#if defined(HAVE_WOBBLING_PRECISION)
          u = HALF * scale;
          u = u + u * sum;    /* u = (1/2)*scale*(t_0 + ...) */
          result[m] = u + u;  /* scale * (t_0 + t_1 + ...) */
#else
          u = scale * sum;    /* scale * (t_1 + t_2 + ...)  */
          result[m] = scale + u; /* scale * (t_0 + t_1 + ...) */
#endif /* defined(HAVE_WOBBLING_PRECISION) */

          scale *= v / (f_m + ONE);

          if (scale == ZERO)  /* remaining elements underflow */
          {
              if (n > m)
                  VSET(n - m, &result[m + 1], ZERO);

              break;          /* early loop exit on underflow */
          }
      }
  }
```

Our algorithm leads to fast code, but for large n values, the use of repeated multiplication for the computation of an outer scale factor that involves a power and a factorial is *less accurate* than we could produce by calling our `IPOW()` function and using a stored table of correctly rounded factorials.

Separate handling of small arguments with Taylor series is *essential* for those Bessel functions, such as $J_n(x)$, that decay with increasing n and require downward recurrence. Otherwise, we could start the recurrence with zero values for $J_n(x)$ and $J_{n-1}(x)$, and all lower elements would then also be set to zero, producing unnecessary premature underflow in the algorithm.

- If upward recurrence is always stable, or x is sufficiently large compared to n that upward recurrence is safe, call the scalar functions for $n = 0$ and $n = 1$ and save their values in the output array, and then quickly fill the remainder of the array using the recurrence relation. The code block for $I_n(x)$ in the file `vbix.h` looks like this:

```
else if (HALF * QABS(x) > (fp_t)n)
{                /* n > 1: use stable UPWARD recurrence */
    fp_t f_k, two_over_x;

    two_over_x = TWO / x;
    result[0] = BI0(x);
    result[1] = BI1(x);

    if (ISINF(result[0]) || ISINF(result[1]))
    {
        fp_t inf_val;
```

```
            inf_val = INFTY();

            for (k = 2; k <= n; ++k)
                result[k] = ( (x < ZERO) && IS_ODD(k) ) ? -inf_val
                                                        : inf_val;
        }
        else
        {
            for (k = 1, f_k = ONE; k < n; ++k, ++f_k)
                result[k + 1] = QFMA(-f_k * two_over_x, result[k],
                                     result[k - 1]);
        }
    }
```

If x is large, then the reciprocal could underflow to subnormal or zero in IEEE 754 arithmetic. We could prevent that by replacing $(2k)/x$ by $(2ks)/(xs)$, with the scale factor s set to a suitable power of the base, such as ϵ^2. However, $I_n(x)$ grows quickly to the overflow limit, so we are unlikely to use x values large enough to require scaling, but infinite function values need special handling to avoid generating NaNs in later subtractions. The major source of accuracy loss in the recurrence is from subtractions near Bessel-function roots, and fused multiply-add operations can reduce that loss.

- Otherwise, use downward recurrence. Generate the first two array values by calls to the scalar functions of orders zero and one. Even though they could be also found from the recurrence, the scalar functions are likely to be more accurate, and because the initial terms are larger for some of the Bessel functions, that may be beneficial in the later use of the results. Call the general scalar functions for orders n and $n-1$ to get the last two array elements, and finally, use the downward recurrence relation to compute the intervening elements.

  Care is needed to prevent unnecessary generation of Infinity, and subsequent NaN values from subtractions of Infinity, or multiplications of Infinity by zero. To see how that is done, here is the code block for $I_n(x)$ in the file vbix.h:

```
    else                /* n > 1: use stable DOWNWARD recurrence */
    {
        fp_t two_k, one_over_x;

        one_over_x = ONE / x;

        if (ISINF(one_over_x)) /* prevent NaNs from Infinity * 0 */
            one_over_x = COPYSIGN(FP_T_MAX, x);

        result[0] = BI0(x);
        result[1] = BI1(x);
        result[n] = BIN(n, x);

        if (n > 2)
            result[n - 1] = BIN(n - 1, x);

        for (k = n - 1, two_k = (fp_t)(k + k); k > 2;
             --k, two_k -= TWO)
        {
            volatile fp_t b_k_over_x;

            b_k_over_x = one_over_x * result[k];
            STORE(&b_k_over_x);
            result[k - 1] = QFMA(two_k, b_k_over_x, result[k + 1]);
        }
    }
```

For speed, division by x is replaced by multiplication by its reciprocal, introducing another rounding error.

If $1/x$ overflows, as it can when x is near the underflow limit on important historical architectures such as the DEC PDP-10, PDP-11, and VAX, and the IBM System/360, and in IEEE 754 arithmetic if x is subnormal, replace it by the largest representable number of the correct sign to prevent later subtractions of Infinity that produce NaN values. Overflow cannot happen if we have already handled small arguments with Taylor series.

We avoid calling `BIN(n - 1, x)` if we already have its value.

Because the calls `BIN(n, x)` and `BIN(n - 1, x)` each generate sequences of Bessel function ratios that could be used to recover the complete function sequence, the work is repeated three times, and an improved implementation of the vector functions would copy that code from the scalar functions, or make it available from them via a separate private function.

We terminate the downward loop once `result[2]` has been generated.

In the loop, we use the `volatile` qualifier and the `STORE()` macro to prevent compilers from reordering the computation $(2k)((1/x)I_k(x))$ to $(1/x)((2k)I_k(x))$, possibly producing premature overflow because $I_k(x)$ can be large.

If either, or both, of $I_{n-1}(x)$ and $I_n(x)$ overflow to Infinity, then that overflow propagates down to $I_2(x)$, even though some of those elements could be finite. A more careful implementation would check for infinite elements generated by the downward recurrence, and then recompute them individually. In practice, applications that need that Bessel function for arguments large enough to cause overflow are better rewritten to use the scaled functions `BISN(n,x)` or `VBIS(n,result,x)`.

- Although the recurrences involve expressions like $2k/x$, throughout the code, we are careful to avoid, or minimize, integer-to-floating-point conversions. Although machines with hardware binary arithmetic can do such conversions quickly, conversions for decimal arithmetic are slower, especially if the arithmetic is done in software.

Comparison of results from the vector Bessel functions with those from the scalar Bessel functions shows two main problems: accuracy loss near Bessel-function zeros, and accuracy loss for large n from the use of repeated multiplication for powers and factorials. When those issues matter, it is better to stick with the slower scalar functions, possibly from the next higher precision, or else to call a vector function of higher precision, and then cast the results back to working precision.

21.19 Retrospective on Bessel functions

The large number of books and research articles on the computation of Bessel functions that we reported at the beginning of this chapter reflects the interest in, and importance of, those functions, as well as the difficulty in computing them accurately.

Many of the recurrence relations and summation formulas presented in this chapter are subject to loss of leading digits in subtractions, particularly for the ordinary Bessel functions, $J_\nu(z)$ and $Y_\nu(z)$, and their spherical counterparts, $j_\nu(z)$ and $y_\nu(z)$, when the argument is near one of their uncountably many roots.

Fortunately, for the Bessel functions that we treat, when upward recurrence is unstable, downward recurrence is stable, and vice versa. The continued fraction for the ratio of Bessel functions is an essential starting point for downward recurrence, but we saw in **Table 21.5** on page 712 that convergence of the continued fraction may be unacceptably slow for large values of the argument z.

The modified functions $I_\nu(z)$ and $K_\nu(z)$ have no finite nonzero roots, but they quickly reach the overflow and underflow limits of conventional floating-point number representations. Subtraction loss often lurks elsewhere, as we saw when the finite and infinite sums for $K_\nu(z)$ are added, and when the term $\log(v) + \gamma$ is computed for $Y_\nu(z)$ and $K_\nu(z)$.

The exponentially scaled functions, $\mathrm{Is}_\nu(z)$ and $\mathrm{Ks}_\nu(z)$, delay the onset of overflow and underflow, but do not prevent it entirely. In addition, the unscaled modified Bessel functions have argument ranges where the function values are representable, yet either, or both, of the exponential factor or the scaled function are out of range, making the unscaled function uncomputable without access to arithmetic of wider range, or messy intermediate scaling, or independent implementations of logarithms of the scaled modified Bessel functions.

The presence of two parameters, the order ν and the argument z, often requires separate consideration of the cases $|\nu| \gg z$ and $|\nu| \ll z$, and the computation time may be proportional to either ν or z.

The spherical Bessel functions of integer order have closed forms that require only trigonometric or hyperbolic functions, and as long as the underlying trigonometric functions provide exact argument reduction, as ours do, can be computed accurately for some argument intervals over the entire floating-point range. Unfortunately, for $n > 0$, subtraction loss is a common problem, and no obvious rewriting, as we did in **Section 21.3** on page 699 for the sines and cosines of shifted arguments, seems to provide a simple and stable computational route that guarantees low *relative* error, instead of low absolute error, for arguments near the function zeros. Higher working precision is then essential.

With complex arguments, and real or complex orders, all of those difficulties are compounded, and we have therefore excluded those cases from the mathcw library and this book.

22 Testing the library

PROGRAM TESTING CAN BE A VERY EFFECTIVE WAY
TO SHOW THE PRESENCE OF BUGS, BUT IT IS HOPELESSLY
INADEQUATE FOR SHOWING THEIR ABSENCE.

— EDSGER W. DIJKSTRA
1972 ACM TURING AWARD LECTURE.

The Cody/Waite book, *Software Manual for the Elementary Functions* [CW80], made two extremely important contributions to the quality of elementary-function libraries, which were frankly dismal on many computing systems until the mid to late 1980s.

First, the book's authors showed how to design accurate algorithms for each elementary function that take account of the properties of binary fixed-point, binary floating-point, and decimal floating-point arithmetic systems, being careful to point out the trouble spots where subtractions, especially in the argument-reduction phase, cause catastrophic bit loss, and where the nasty effects of wobbling precision can be reduced by rescaling. Of course, every computer vendor already had libraries for all of the elementary functions in Fortran and often other languages of the time (Algol, COBOL, Jovial, Pascal, PL/1, ...), but Cody and Waite showed how to improve those libraries.

Second, and perhaps more importantly, Cody and Waite showed how to design *portable* test software that can assess the quality of any elementary function library without knowing anything at all about how the functions are implemented, and without requiring large tables of accurate numbers to compare against. Their book contains no explicit Fortran code for implementing their algorithms, but it does contain complete Fortran code for testing existing implementations, later released in machine-readable form as *ELEFUNT: The Elementary Function Test Package*.

With the help of a student assistant, this author translated the entire ELEFUNT package from Fortran to C in 1987, and later adapted it for use with C++ as well. That work predated automated Fortran-to-C translators, such as David Gay's f2c, begun at AT&T Bell Laboratories in 1989.

Later, in 2002, this author translated ELEFUNT to Java, providing a clean formatted-I/O class library that Java sorely lacks. All of those versions of ELEFUNT are freely available on the Web,[1] and they have been used to test scores of elementary-function libraries in Fortran, C, C++, and Java. For example, this author used ELEFUNT to uncover severe deficiencies of the DEC TOPS-20 PDP-10 Fortran library in 1981, and after he filed problem reports with the vendor, the next release of the compiler shipped with a superb elementary-function library.

For our implementation of the Cody/Waite algorithms, the test software is taken almost verbatim from the 1987 C translation, but updated to include checks on the setting of the C errno global variable, and additional tests in the spirit of Cody and Waite for some of the new functions added to the mathcw library.

By default, when the library is built with the standard recipe

```
% ./configure && make all check install
```

the precision of the host arithmetic is used to select appropriate polynomial approximations, maintaining both speed and accuracy. In that application, bigger is not necessarily better: a higher-degree polynomial offers higher precision, but takes more computation, and because the hardware precision is fixed, the additional unnecessary computation merely contributes to increasing the rounding errors, eventually making the results worse, and always making them slower.

You can test the three versions of a single function with targets check-asin, check-atan, check-exp, and so on. In those targets, acos() is tested with asin(), atan2() with atan(), cos() with sin(), cosh() with sinh(), cotan() with tan(), and log10() with log(). Thus, there are fewer test programs than elementary functions.

It is possible to test *all* of the embedded polynomial data like this:

```
% make check-all
```

[1] See http://www.math.utah.edu/pub/elefunt.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_22

That runs separate checks for each of the elementary functions through targets `check-all-asin`, `check-all-atan`, `check-all-exp`, ..., and each of those loops over all polynomial precisions embedded in, and dynamically extracted from, the source code, deletes object files for the function to be tested, recompiles them with the current polynomial precision, rebuilds the library, and then steps into each of the three subdirectories to rebuild and rerun the tests with the just-updated library.

The original ELEFUNT tests used 2000 random arguments in each test interval, a number that was reasonable for machines of the 1970s, and that remains the default. However, in the C version of the ELEFUNT tests, you can override that number at run time with a new value in the environment variable `MAXTEST`. Similarly, you can provide an alternate initial random-number generator seed in the environment variable `INITIALSEED` to change the sequence of random numbers used for test arguments:

```
% env INITIALSEED=987654321 MAXTEST=1000000 make check
```

Increasing `MAXTEST` of course makes the tests run proportionally longer, but usually also slightly increases the worst-case error reports, because the more arguments that are tested, the more likely it is that bigger errors are found.

The ELEFUNT tests have voluminous output: about 1500 lines for one run of all of the tests in the original ELEFUNT package, and 6750 lines in the extended tests in the mathcw package. It is therefore impractical to assess the output results quickly without further filtering. The default filter extracts just the bit-loss reports, which are the key measure of library accuracy, reducing the output for tests of one precision of the entire mathcw package to about a hundred lines. Thus, a typical report for several tests of an elementary function of one floating-point data type looks like this:

```
ELEFUNT test matan
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.66
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.00
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   1.00
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.00
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   1.92
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.19
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   1.34
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.00
```

Those reports are repeated for the `float`, `double`, and `long double` versions of each elementary function. In practice, the reports are shortened even further by excluding those below the cutoff `MAXLOSS`, set to a default of 2 in the `Makefile`. The previous example would then be reduced to just the test header, but running `make MAXLOSS=0 check` would produce the example shown.

The ELEFUNT reports contain important information about the handling of special arguments, function symmetry properties, and behavior near the underflow and overflow limits, so it is worth examining them in more detail. To capture the complete ELEFUNT test output in a file, simply override the filter setting:

```
% make CHECK_FILTER=cat check-all > big-output-file
```

When the selected polynomial precision is below the hardware precision, the tests warn about that immediately before the test output:

```
=== WARNING: P = 24 is below 64-bit working precision: expect loss of 40 bit(s)

ELEFUNT test matan
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS  35.23
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS  34.21
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS  35.81
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS  35.15
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS  35.66
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS  34.75
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS  34.21
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS  33.02
```

Here, all of the bit-loss reports were below the expected 40-bit loss, so the Cody/Waite rational polynomials for `atan()` are actually about four bits more precise than their book indicates. That may have been intentional, but as we noted earlier, their book does not discuss the computation or derivation of the polynomials.

On a fast workstation (2 GHz AMD64 processor), the `make check-all` run takes about twenty minutes, and a `make check` run takes about three minutes.

Of course, after running `make check-all`, the last version compiled for each function is the one with the highest-degree polynomials. That version of the library is needlessly slower than it should be, so if you have not installed the package yet, clean up, rebuild, revalidate, and install it with:

```
% make clean all check install
```

You can also use the mathcw test procedures to check the quality of your native C math library, just by changing the library variable:

```
% make LIBS='-lm -lmcw' check
```

The mathcw library option `-lmcw` must usually be specified after the native library `-lm` to provide a source of functions that are not supplied by the host, such as the additional routines described in **Section 3.16** and listed in **Table 3.4** on page 58.

Convenience targets in the `Makefiles` let you specify testing of just the C89 functions in your native math library like this:

```
% make LIBS='-lm' check-c89
```

or like this:

```
% make check-native-c89
```

The targets `check-c99` and `check-native-c99` provide similar testing of the C99 repertoire in mathcw and in the native math library. Targets `check-w` and `check-q` test the two extra precisions available on Hewlett–Packard HP-UX on IA-64, and target `check-hp` combines those tests with the normal `check` tests.

The ELEFUNT tests have all been adapted for decimal floating-point arithmetic as well, with code in separate subdirectories for each supported decimal format. The command

```
% make check-decimal
```

runs those decimal versions of the tests.

22.1 Testing `tgamma()` and `lgamma()`

The `test-*` subdirectories in the mathcw library distribution contain the ELEFUNT code suites, augmented with new code written in a similar style for testing several additional functions provided by our library that were not covered in the Cody/Waite book. In this section, we describe some of the issues that affect the testing of the gamma and log-gamma functions.

There are dramatic changes in the gamma function values near the poles at large negative arguments, as suggested by **Figure 18.1** on page 522, and shown by this numerical experiment in IEEE 754 128-bit arithmetic, where the first four arguments are spaced one machine epsilon apart:

```
hoc128> gamma(-100)
+Inf

hoc128> gamma(-100 + macheps(-100))
1.071_510_288_125_466_923_183_546_759_519_241e-126

hoc128> gamma(-100 + 2*macheps(-100))
5.357_551_440_627_334_615_917_733_797_596_452e-127

hoc128> gamma(-100 + 3*macheps(-100))
```

```
3.571_700_960_418_223_077_278_489_198_397_799e-127

hoc128> gamma(-99.5)
3.370_459_273_906_717_035_419_140_191_178_166e-157

hocd128> gamma(-99.5 + macheps(-99.5))
 3.370_459_273_906_717_035_419_140_191_178_321e-157

hocd128> gamma(-99.5 + 2*macheps(-99.5))
 3.370_459_273_906_717_035_419_140_191_178_475e-157

hocd128> gamma(-99.5 + 3*macheps(-99.5))
 3.370_459_273_906_717_035_419_140_191_178_631e-157
```

Without special algorithms near poles, or arithmetic of much higher precision, we cannot expect high accuracy in pole regions. However, away from the poles on the negative axis, only a few low-order digits of the gamma-function values change as we move in steps of one machine epsilon.

Because of the varying behavior of the gamma and log-gamma functions, the test methods implemented in `ttgamm.c` and `tlgamm.c` require several different regions and algorithms. Ideally, they should compute values of $\Gamma(x)$ and $\log|\Gamma(x)|$ by methods that differ from those in the implementations of `tgamma()` and `lgamma()`.

For $\Gamma(x)$, `ttgamm.c` uses the Taylor series for a small interval around $x = 1$, and the recurrence relation $\Gamma(x) = (x-1)\Gamma(x-1)$ for several intervals on the positive and negative axes, where $\Gamma(x-1)$ is computed as `tgamma(x-1)`. That is reasonably straightforward, but requires care to purify the random test values so that both x and $x-1$ are exactly representable.

Because of the rapidly changing function values near the poles, the test intervals exclude a region of size $\pm 1/32$ around negative integers. If that precaution is not taken, then the reported bit losses are large, but that cannot be avoided unless the pole regions are excluded.

When possible, the Taylor series is evaluated in the next higher precision, because the terms alternate in sign, and their sum is thus subject to leading bit loss. The Taylor-series coefficients do not have any simple rational form, so they are simply supplied as high-precision 36-digit values in arrays. To ensure that they are exact to machine precision, the coefficients are computed at run time from exactly representable parts and small corrections, giving an effective precision of about 43 decimal digits that is sufficient for all current hardware floating-point architectures. Split coefficients are provided for both decimal and binary bases, because the exact parts are given as rational numbers that cannot be exact in both bases. They assume that at least 20 bits or 6 decimal digits are available in single precision, a property that holds for all current and historical floating-point architectures.

Because of the rapid growth of $\Gamma(x)$, the Taylor-series test can cover only a relatively small interval, but the reported average bit loss should be below one bit for careful implementations of `tgamma()`.

The test of $\log|\Gamma(x)|$ in `tlgamm.c` also uses a Taylor series on a small interval, and the recurrence relations on several different intervals on the positive and negative axis.

The Taylor series of $\log|\Gamma(x)|$ around $x = 1$ converges slowly, so instead, we use the series around $x = 5/2$. There are no simple rational representations of the coefficients, so as in `ttgamm.c`, they are split into exact values and small corrections, giving an effective precision of about 43 decimal digits.

Application of the recurrence relations for $\log|\Gamma(x)|$ is more difficult than it is for $\Gamma(x)$, and requires that the implementation of $\log(x)$ be highly accurate. For $\Gamma(x)$, the recurrence involves only a single multiply, introducing at worst a half-bit or one-bit error in the test value, depending on the rounding characteristics of the host arithmetic. For $\log|\Gamma(x)|$, the recurrence relations suffer serious subtractive bit loss in certain regions, as shown in **Figure 22.1** on the next page. The more detailed view in **Figure 22.2** shows the loss regions discussed earlier in **Section 4.19** on page 89.

In particular, the recurrence relations *cannot* be used for the approximate intervals $[-3.99, -2.24]$, $[0.46, 1.83]$, and anywhere close to negative integer values. As with `ttgamm.c`, pole regions are excluded from testing anyway, but that still leaves the important region $[-4, +2]$ where some other technique is called for.

The obvious approach of using `log(abs(tgamma(x)))` for evaluation of $\log|\Gamma(x)|$ in the region $[-4, +2]$ would presumably provide an accurate value of `lgamma()` that is not subject to subtraction loss. However, that is probably what any reasonable implementation of `lgamma()` does anyway for at least part of this region, so the test could erroneously report zero bit loss, even if there are serious errors in `tgamma()`.

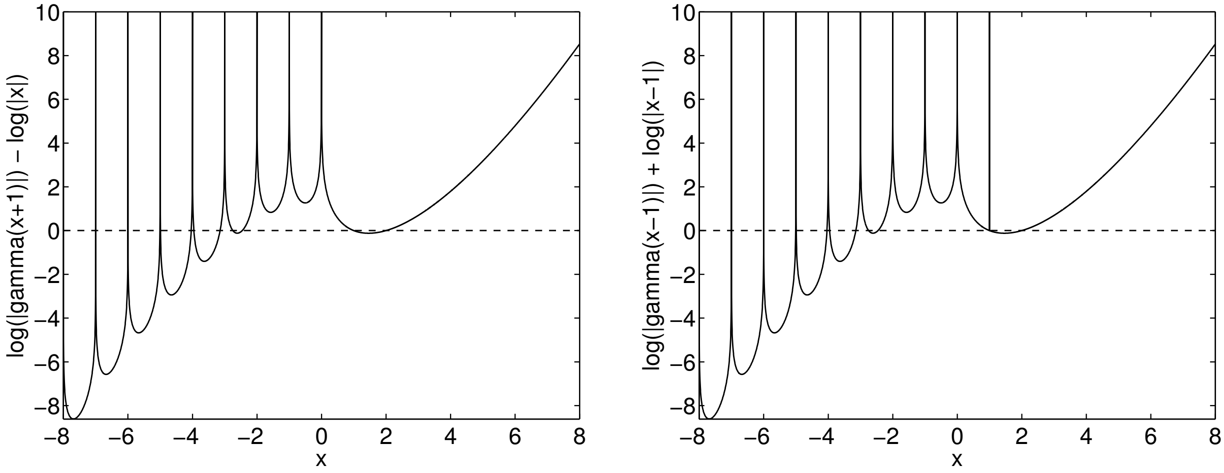


Figure 22.1: Recurrence relations for $\log|\Gamma(x)|$. Whenever the difference or sum is near zero, there is catastrophic bit loss.

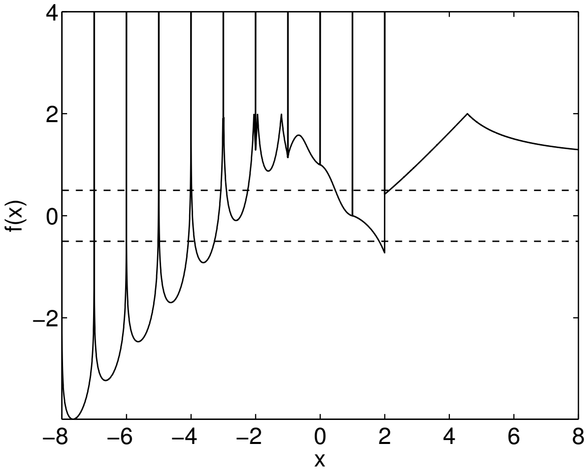


Figure 22.2: Significance loss in recurrence relations for $\log|\Gamma(x)|$ as the value of $(\log|\Gamma(x-1)| + \log|x-1|)/\max(\log|\Gamma(x-1)|, \log|x-1|)$. Whenever the plotted value falls into the range $[-1/2, +1/2]$ marked by the dashed lines, one or more leading bits are lost in the addition of values of opposite sign.

An alternative approach that `tlgamm.c` uses instead in the region $[-4, +2]$ is rational-polynomial approximations of a better-behaved function, $1/\Gamma(x)$, that is plotted in **Figure 22.3** on the next page. Because the logarithm of the reciprocal requires just a sign change, there is no additional error introduced by a division to recover $\Gamma(x)$. A rational minimax polynomial of degree $\langle 16/16 \rangle$ approximates $1/\Gamma(x) - 1$ on the interval $[-3.9999, -3.0001]$ with worst-case accuracy of 129 bits, sufficient even for the 113-bit precision of IEEE 754 128-bit arithmetic, and the `log1p()` function recovers an accurate value of $\log|\Gamma(x)|$. For the interval $[-2.9999, -2.0001]$, a $\langle 16/16 \rangle$-degree approximation for $1/\Gamma(x) + 1$ (notice the sign change) provides worst-case accuracy of 128 bits, and for the interval $[0.46, 1.83]$, a $\langle 14/14 \rangle$-degree approximation for $1/\Gamma(x) - 1$ provides at least 118 bits. The function `lgamma()` is one of just two in the entire `mathcw` test suite for which it has been necessary to resort to use of rational polynomial approximations. The drawback is that they need to be generated for the highest precision to be tested, and for portability, that precision must be high enough to encompass all current floating-point architectures. The test program issues a warning if the precision is not adequate.

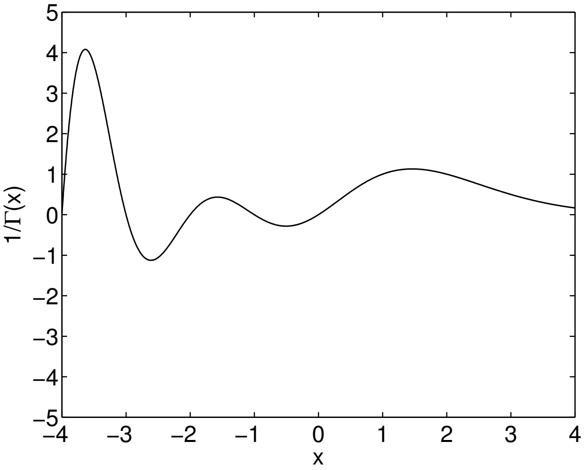


Figure 22.3: Reciprocal of the gamma function, $1/\Gamma(x)$.

22.2 Testing `psi()` and `psiln()`

For testing purposes on the positive axis, we can use either of the first two recurrence relations shown in **Section 18.2.2** on page 539 with suitably purified test arguments. On the negative axis, we can use the reflection formula. As with $\Gamma(x)$ and $\log \Gamma(x)$, the Taylor-series coefficients do not have a simple form, so they are stored as exact values and corrections, and then summed at run time to ensure that they are correct to the last bit.

The test programs `tpsi.c` and `tpsiln.c` are quite straightforward, and test a dozen or so regions along the x axis.

22.3 Testing `erf()` and `erfc()`

The ordinary and complementary error functions, `erf()` and `erfc()`, are graphed in **Figure 19.1** on page 594, and are computed in the `mathcw` library as described earlier in **Section 19.1** on page 593. Testing them in isolation is difficult, because there is no simple addition rule relating function values at one point to those at another point. The functions are defined on the entire real axis, so a Taylor-series evaluation is of limited utility for testing more than a small interval. In addition, it is particularly important to assess the accuracy of the functions in the regions where they are close to the underflow limit, because it is precisely here that some historical recipes for the computation of the error functions are poor.

The procedure recommended by Hart et al. [HCL$^+$68] for brute-force evaluation of the error functions is to use one of two continued fractions (see their description in **Section 2.7** on page 12), depending on the value of x. Although continued fractions can be manipulated to reduce the number of divides to just one, thereby speeding the computation, the test programs do not do so in the interests of keeping the continued-fraction code clear and simple. Also, the coefficients in the reduced-division formulation can suffer from premature overflow or underflow, requiring special scaling. The continued fractions for $\operatorname{erf}(x)$ and $\operatorname{erfc}(x)$ have particularly simple, small, and exactly representable coefficients, so direct evaluation avoids the problems of the alternate algorithm.

The `terf.c` and `terfc.c` files use multiple test regions, but the regions and test algorithms differ substantially, so separate test files are appropriate, and facilitate testing those related functions independently of each other.

Near the origin, the tests compute the functions from the first six terms of the Taylor series, which fortunately has relatively simple rational coefficients that can be factored to a common denominator so that all numerators are exactly representable. Even better, the series involves only odd terms, and the coefficients fall by nearly a factor of ten with each term, so there is little problem with subtraction loss in summing the series.

For $|x| < 1$, the test file `terf.c` sums the continued fraction, using a term count that depends only on the required precision.

The remaining intervals have $|x| \geq 1$, and they investigate the arguments up to where $\mathrm{fl}(\mathrm{erf}(x)) = 1$ to machine precision. In each test region, `terf.c` temporarily uses $\mathrm{erf}(x) = 1 - \mathrm{erfc}(x)$ as the accurate function value, because the subtraction is stable for x in the interval $[0.477, \infty)$. However, a separate check that does not depend on an accurate `erfc()` function is desirable, so more work is still needed to remove that dependence.

Testing the complementary error function is more difficult. In `terfc.c`, a five-term Taylor series with exactly representable coefficients handles the case of small x. For $x \geq 3$, a continued fraction of modest length provides an accurate estimate of $\mathrm{erfc}(x)$; the required term count is determined by a quadratic fit to the results of a numerical experiment. The difficult region for that function is the interval $[0, 3]$. In part of that interval, the continued fraction for $\mathrm{erf}(x)$ converges quickly, but the subsequent subtraction $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$ loses many leading bits. In the remainder of the interval, the continued fraction for $\mathrm{erfc}(x)$ converges slowly, requiring as many as 3500 terms for small x. A compromise between the two is called for, and higher precision is used when available to reduce the effect of leading bit loss. That problem could be avoided by reverting to a rational polynomial approximation, but we prefer to avoid that approach.

The asymptotic expansion for $\mathrm{erfc}(x)$ is only usable for 128-bit arithmetic when $x > 10$, where simpler and more accurate tests can be used instead, so that series is not useful for testing purposes.

As additional checks, `terfc.c` uses the reflection formula, $\mathrm{erfc}(x) = 2 - \mathrm{erfc}(-x)$, for negative arguments and for small positive arguments in the region where the Taylor series applies, and it also checks the region $(-\infty, -9]$, where $\mathrm{fl}(\mathrm{erfc}(x)) = 2$ in all floating-point systems with 113 or fewer bits of precision.

22.4 Testing cylindrical Bessel functions

A few years after the ELEFUNT code was published, Cody and Stolz [CS89, CS91] prepared similar test programs for the cylindrical Bessel functions $J_0(x)$, $J_1(x)$, $Y_0(x)$, $Y_1(x)$, $I_0(x)$, $I_1(x)$, $K_0(x)$, and $K_1(x)$, and made them freely available in the Netlib repository.[2]

This author translated them from Fortran to C, and added them to the `test-*` subdirectories of the mathcw library distribution. We leave the testing details to the original articles, but note here that the Cody/Stolz code continues the practice of assessing accuracy by numerical evaluation of mathematical identities where function arguments are carefully reduced to eliminate argument error.

22.5 Testing exponent/significand manipulation

The `frexp(x, &n)` function has long been used in the C language for *exact* decomposition of a finite nonzero floating-point number into product of a significand in $[1/\beta, 1)$ and a power of the base, β^n.

The companion function `ldexp(s,n)` returns the exact product $s \times \beta^n$, providing the reconstruction operation that allows exact scaling by powers of the base.

Neither of those operations requires any numerical approximations, but their correct behavior is essential for higher-level code that uses them. Their testing is therefore mandatory.

The file `tfrexp.c`, and its companions for other floating-point types, follows the ELEFUNT coding style, and samples the entire floating-point range, including IEEE 754 subnormals when they are supported. When the floating-point symbolic constants from the system header file `<float.h>` suggest IEEE 754 arithmetic, the tests also include run-time generation of a NaN and signed Infinity.

Similarly, the file `tldexp.c` and its companions supply test code for checking the correct behavior of the `ldexp()` function family.

22.6 Testing inline assembly code

About 35 source code files in the mathcw library distribution contain inline assembly code to allow access to native hardware instructions for fast and accurate elementary-function evaluation. Although that code is expected to be reliable, it is always possible that compiler optimization rearranges instructions, making them invalid. The test

[2] See `http://www.netlib.org/specfun/`.

program in chkasm.c therefore does some limited tests in data types float, double, and long double that should suffice to uncover such errors.

The root functions are easily verified, but the others are handled by measuring the relative error of computed values against stored constant tables of about 25 triples of values of $(x, f_{\rm hi}(x), f_{\rm lo}(x))$ in long double format for a limited range of exactly representable arguments. The two-part representation of the function value supplies up to 20 bits of additional effective precision, counteracting rounding errors from decimal-to-binary conversions. A Maple program in the file chkasm.map generates the header file chkasm.h with the data tables. A typical test looks like this:

```
% cc -DMAXULPS=50.0 -I.. chkasm.c ../libmcw.a && ./a.out
chkasm: test of functions with possible inline-assembly code

MAXTEST = 2000
MAXULPS = 50.00

OK      sqrtf
OK      sqrt
OK      sqrtl

OK      rsqrtf
OK      rsqrt
OK      rsqrtl

... output omitted ...

OK      sinhcoshf
OK      sinhcosh
OK      sinhcoshl

SUCCESS: All 114 functions pass their tests
```

22.7 Testing with Maple

Software testing with the ELEFUNT package depends mostly on measuring how closely certain mathematical relations are satisfied, without using arithmetic of higher precision. Even with careful argument purification, numerical computation of those identities is usually subject to a few rounding errors.

The testing approach illustrated throughout this book with graphs of errors in units in the last place compares results computed in working precision and in the next higher precision. In most functions whose computation is described in this book, the algorithms are independent of precision, except for the particular polynomial approximations used. That means that an algorithmic error would likely not be detected by the error plots: the computation might be accurate, even though it does not correspond to the desired function. Also, the functions at the highest available precision cannot be tested that way.

It is therefore desirable to carry out separate tests against completely independent implementations of the functions. On modern computers, any of several symbolic-algebra systems provide most of the software necessary to do that, because they generally permit computations to be carried out with any reasonable user-specified precision, and they have a large repertoire of mathematical functions.

The main problem is that it must be possible to guarantee *exact transfer* of test arguments and function values from the library under test to the symbolic-algebra system, where the two may be on different computers. The only reliable way of doing that is to represent floating-point numbers in exact rational integer form. In addition, implementation of new computational algorithms in symbolic algebra should not normally be required, so as to avoid the prove-the-proof problem.

For example, the function log1p(x) is generally absent from most programming languages, including all of the algebra systems available to this author. Its mathematical definition as $\log(1 + x)$ is usable numerically only if the argument can be computed exactly, or if x is sufficiently small that the Taylor-series expansion can be reduced to just one or two terms. In Maple, that can be done with a function that is easy to understand, does not require excessive computational precision, and is short enough to be unlikely to harbor bugs:

```
log1p := proc(x)
             local w:
             w := 1 + x:
             return ‘if‘(evalb(w = 1), x, log(w) * x / (w - 1))
         end proc:
```

Here, we use a one-term series if x is tiny, and otherwise, we compute a formula derived in **Section 10.4** on page 290.

The `genmap-*` subdirectories in the `mathcw` distribution normally each contain just two files: a `Makefile`, and a template file, `msbis0.c`. Declarations and output format items in the template file are customized for a particular floating-point data type, but the templates in each subdirectory are similar. In particular, the code in the template file contains argument test ranges that cover all of the elementary and special functions in the library, as well as a large collection of special arguments that are known to cause trouble for one or more functions, such as fractional multiples of π. The nearest representable neighbors of the special arguments are also used in the tests.

The template file includes three header files that are shared among all of the test subdirectories. Those files provide a table of special arguments, and prototype declarations for test functions that map library functions with multiple arguments to wrapper functions with a single argument. The wrappers permit a single template file to be used for all functions.

The `Makefile` contains lists of function names that are used by the stream-editor utility, `sed`, to generate new test programs from the template file, and, by default, all of the functions would be tested with a single command. More commonly, results are needed only for a function subset, so here is how two particular functions of type `float` can be tested:

```
% cd genmap-s
% make FUNS=’expm1 log1p’ MAXTEST=10000 MAXULPS=0.5 all check

Making C test programs
-r--r--r--  1 user group 12649 Apr 28 08:36 check-expm1f.c
-r--r--r--  1 user group 12649 Apr 28 08:36 check-log1pf.c
Making Maple test programs
-r--r--r--  1 user group 13104324 Apr 28 08:36 check-expm1f.map
-r--r--r--  1 user group 11469880 Apr 28 08:36 check-log1pf.map
Running Maple test programs
-rw-rw-r--  1 user group 2531061 Apr 28 08:39 check-expm1f.dat.maple
-rw-rw-r--  1 user group    9501 Apr 28 08:39 check-expm1f.log
expm1
FAIL: expm1f: 61 of 148670 tests have errors above 0.5 ulps
Maximum error = -0.563412 ulps for x = 0.696086287 at line 109559
-rw-rw-r--  1 user group 3098628 Apr 28 08:41 check-log1pf.dat.maple
-rw-rw-r--  1 user group    1721 Apr 28 08:41 check-log1pf.log
log1p
PASS: log1pf: all 131002 tests have relative errors below 0.5 ulps
```

The `FUNS` list gives the function names in their unsuffixed form used for data type `double`, so that the function lists do not need to be changed in each subdirectory. The `MAXTEST` and `MAXULPS` settings override defaults that are set in the `Makefile`. The target `all` builds the test programs, and the target `check` runs the tests. The separation of building and testing is essential, because Maple is not available on most of the systems where library testing is needed.

The functions to be tested are by default found in the library file `libmcw.a` in the parent directory. However, it is possible to test functions in other libraries by supplying additional command-line assignments. For example, we could test a function in the vendor-supplied math library like this:

```
% make FUNS=expm1 LIBS="-lm -lmcw" SUFFIX=-lm
```

The specified suffix is applied to the created filenames to distinguish them from the default ones. The `mathcw` library option, `-lmcw`, is needed to supply additional functions that are absent from the native library, but required for the tests.

In the sample test, the `make` command first generated two C programs from the template file, and then compiled, linked, and ran them to produce two Maple programs. It then ran those programs from the `../maple` subdirectory to produce the `*.log` summary files, and the `*.dat.maple` data files that are suitable for error plotting.

Here is a small fragment of one of the Maple programs:

```
interface(quiet = true):
printf("Test of log1pf(x)\n\n"):
C_NAME := "log1pf":
...
Digits := 40;
...
read("htod.map"):
read("dtoh.map"):
read("mathcw.map"):
...
x := htod("-0x1.ffb882p-1"): check(htod("-0x1.e0e3f8p+2"), log1p(x)):
x := htod("-0x1.fe3e62p-1"): check(htod("-0x1.6b34eep+2"), log1p(x)):
...
```

Most of the program consists of lines with argument assignments and a call to a function that receives the values computed in C and in Maple. Here, the C99 hexadecimal format ensures exact representation of values in both languages, and the `htod()` function handles the exact conversion from hexadecimal to decimal. The `htod()` function and its companion `dtoh()` for the reverse conversion are available in files in the `maple` subdirectory. The `mathcw.map` file provides the critical interface between functions known by their names in C, and corresponding functions in Maple.

For tests of functions in decimal arithmetic, decimal values are used instead of hexadecimal, because Maple's internal representation of multiple-precision numbers is decimal, rather than binary.

The Maple program is carefully formatted so that it can track input line numbers. In the event that a computed error is larger than the specified value of `MAXULPS`, a warning message identifies the offending line with its values of x and $f(x)$. The largest error found above the tolerance is recorded and the last output line then reports the corresponding argument and its line number, as shown earlier for the test of `expm1(x)`.

Part of one of the output log files looks something like this:

```
Test of log1pf(x)

ULP = 2 ** (-(24 - 1))
MINSUBNORMAL = 2 ** (-149)
Digits := 40
MAXULPS := 0.5
PLOTFILE := "check-log1pf.dat.maple"

Tests with 10000 random arguments on [ -3.40282e+38, -1e+07 ]
SKIP: log1pf

Tests with 10000 random arguments on [ -1e+07, -100000 ]
SKIP: log1pf

... output omitted ...

Tests with 10000 random arguments on [ -1, -0.001 ]
PASS: log1pf

Tests with 10000 random arguments on [ -0.001, -1.19209e-07 ]
PASS: log1pf

... output omitted ...

Tests with 10000 random arguments on [ 1e+07, 3.40282e+38 ]
PASS: log1pf

Tests with special arguments
PASS: log1pf: 11003 of 11003
```

```
PASS: log1pf: all 131002 tests have relative errors below 0.5 ulps
```

Test ranges for $x < -1$ were automatically skipped because the test function returns a NaN in those regions. That practice is essential to avoid having to tailor the test ranges to each function. The last output line shows that no test arguments resulted in an error above the specified tolerance.

For the exponential function less one, there are several test failures, and the worst case is readily extracted from the Maple file using the reported line number:

```
% sed -n 109559p check-expm1f.map
x := htod("0x1.64656cp-1"): check(htod("0x1.0181ccp+0"), expm1(x)):
```

A separate computation in Maple finds a more accurate function value that can then be correctly rounded to the 24-bit significand:

```
exact:    0x1.0181_cd22_2a39_a385_4b7c_...p+0
rounded:  0x1.0181_cep+0
computed: 0x1.0181_ccp+0
```

Once the Maple testing facility was operational, confidence in the general correctness and accuracy of the library improved significantly, and it became much easier to test functions for which no ELEFUNT-like test programs exist yet. Those programs average more than 400 lines each, and they require expert analysis to find suitable mathematical identities that can be computed accurately, and expert, and portable, programming to implement the tests. By contrast, the symbolic-algebra alternative requires just one, or at most a few, lines to be written by a human, with the machine doing all of the rest of the work automatically.

Eventually, similar tests should be developed using other symbolic-algebra systems, particularly those that are available under open-source licenses, such as Axiom and Maxima.

Using high-precision arithmetic in symbolic-algebra languages for testing can be expensive for some functions. For Maple's implementation of the beta function, testing betam1(x) with high-precision values of $\beta(x) - 1$ is impractical. That problem could probably be solved by writing a separate implementation of the direct computation of $\beta(x) - 1$ in Maple, but we have yet to do so.

22.8 Testing floating-point arithmetic

Modern computer hardware is generally reliable, and users expect computers to operate correctly for long periods until interrupted for software or hardware upgrades, or by extended loss of the electrical supply. Nevertheless, chip failures due to manufacturing defects, heat, or aging are occasionally seen, so it is desirable for users to be able to validate correct operation of at least arithmetic operations.

In the early 1980s, William Kahan wrote a floating-point arithmetic test program called paranoia, and that program has since been translated into other languages and made more portable [Kar85]. This author maintains a copy on the Web[3] and other versions are readily found by Web search engines. Here is a typical run of the program:

```
% make paranoia
% ./paranoia-sp
Lest this program stop prematurely, i.e. before displaying

    ‘END OF TEST’,

try to persuade the computer NOT to terminate execution when an
error like Over/Underflow or Division by Zero occurs, but rather
to persevere with a surrogate value after, perhaps, displaying some
warning.  If persuasion avails naught, don’t despair but run this
program anyway to see how many milestones it passes, and then
amend it to make further progress.

... lengthy output omitted ...
```

[3] See http://www.math.utah.edu/~beebe/software/ieee/.

```
No failures, defects nor flaws have been discovered.
Rounding appears to conform to the proposed IEEE standard P754.
The arithmetic diagnosed appears to be Excellent!
END OF TEST.

% ./paranoia-dp
... more lengthy output omitted ...
The arithmetic diagnosed appears to be Excellent!
END OF TEST.
```

22.9 The Berkeley Elementary Functions Test Suite

In the late 1980s, Kahan's student Zhi-Shun Alex Liu developed the *Berkeley Elementary Functions Test Suite* [Liu87, Liu88] as part of his thesis project.[4] The *BeEF tests* are designed to follow paranoia and ELEFUNT tests, and make more extensive tests of elementary functions in both DEC VAX and IEEE 754 arithmetic.

The BeEF and paranoia tests are now incorporated in another Berkeley test package, UCBTEST[5] [Stu95]. The package has regrettably not been brought to the software portability level that it deserves, and carries license restrictions that forbid redistribution, but on the few architectures on which it can be built, a typical unpacking, build, and test run looks like this:

```
% cd /tmp
% tar xfz ucbtest.tgz
% cd ucb/ucbtest
% make SRC=/tmp/ucb
... lengthy output omitted ...
Totals clib_DP.output :

Total   60 tests: pass   60,  flags err 0, value err 0, acosd
Total  352 tests: pass  352,  flags err 0, value err 0, addd
Total   77 tests: pass   77,  flags err 0, value err 0, asind
Total  104 tests: pass  101,  flags err 0, value err 3, atan2d
...
Total  321 tests: pass  321,  flags err 0, value err 0, subd
Total   54 tests: pass   54,  flags err 0, value err 0, tand
Total   72 tests: pass   72,  flags err 0, value err 0, tanhd
...
csin_DP.output :

 ucbtest START   in /tmp/ucb/ucbtest/beef.c at line 448 for double
 6250 tests on 16 regions with 53 significant bits

 NME  =  Negative Maximum Error observed in ULPs
 PME  =  Positive Maximum Error observed in ULPs
 NMC  =  Non-monotonicity count
 {SYM}=  Non-symmetry count if nonzero

           [      From     ,        to     )  N.M.E.  P.M.E. NMC {SYM}

SIN(X)   1 [      0.0000000,      0.8750000)  -0.629   0.617 0
SIN(X)   2 [      0.8750000,      2.2500000)  -0.598   0.634 0
SIN(X)   3 [      2.2500000,      4.0000000)  -0.618   0.605 0
SIN(X)   4 [      4.0000000,      5.4375000)  -0.623   0.600 0
SIN(X)   5 [      5.4375000,      7.0000000)  -0.628   0.629 0
```

[4] Available at http://www.ucbtest.org/.

[5] Available at http://www.netlib.org/fp/ucbtest.tgz.

```
SIN(X) ALL [      0.0000000,      7.0000000)  -0.629   0.634 0

 ucbtest UCBPASS in SIN(X) at line 442 for generic
...
```

22.10 The AT&T floating-point test package

AT&T Bell Labs scientist Norman Schryer developed one of the earliest portable software test packages for checking the behavior of floating-point arithmetic [Sch81]. The fptest package was once available under license from his employer, but sadly, may no longer be; fortunately, this author is a licensee.

The fptest package is written in portable Fortran, and the cited report contains summaries of its findings for more than 30 mainframe architectures available about 1980. Those results are an outstanding collection of the kinds of problems with floating-point arithmetic that programmers routinely faced before IEEE 754 arithmetic became widely available. Among the gems recorded in the report are ones like these, with vendor identities omitted:

> Early versions of this machine had serious errors. For example, in double precision, $\mathrm{fl}(1 - 2^{-55}) \geq 1$.
>
> It is ... possible for two numbers that differ in bit 48 to appear equal.
>
> As a result of the week-long bug chase that ended at the backplane, the computing center installed a tastefully designed use of FPTST to be run at 3am every day ... Since its installation it has found that FPA's have partially fallen out of the backplane on three occasions and sent that information to the system gurus.
>
> In single precision, comparison of 0 with 0 was incorrect. That is, things like `x .lt. y` were `.true.` when $x = y = 0$.
>
> The compiler used ... was so buggy that not much of the test could be run, but we got some information. The first fact we got was that 1 is not the multiplicative identity, that is, $\mathrm{fl}(1 \times x) \neq x$ for some x. ... There are numbers $x < 0$ and $y < 0$ such that $\mathrm{fl}(x \times y) < 0$!

A build and test run of the fptest package looks something like this on a modern system:

```
% f77 *.f && ./a.out
Long Real Floating-point Test With

  B =   2,   T =    53
  Emin =   -1021,   Emax =    1024

 BEX =  -1021   -53     0    53  1024
 BEY =  -1021   -53     0    53  1024
 BIX =      1    27    53
 BIY =      1    27    53
nBEX =      3     3     3     3     3
nBEY =      3     3     3     3     3
nBIX =      3     3     3
nBIY =      3     3     3

nEX, nEY =    21    21
nIX, nIY =    11    11

This test will produce and test results which underflow.
If overflow or divide check errors occur,
then the basic machine constants have been incorrectly set.

Floating-point test completed successfully.
```

22.11 The Antwerp test suite

The Research Group CANT (Computer Arithmetic and Numerical Techniques) at the University of Antwerp has made many useful contributions to the testing and use of floating-point arithmetic. Their IEEE 754 Compliance Checker test suite [VCV01a, VCV01b], and related tools for multiple-precision arithmetic, and for continued fractions for special functions [CPV$^+$08, BC09], are available on the Web.[6]

The test suite is written mostly in C++, and is driven by data files that can be adapted for many different floating-point architectures. Here is a sample unpacking, build, and test run that checks for correct single-precision addition operations:

```
% cd /tmp
% tar xfz IeeeCC754.tgz
% cd IeeeCC754
% ./configure -platform AMD && make
... output omitted ...
% ./IeeeCC754 -s BasicOp/testsets/add
Taking input from BasicOp/testsets/add.
.......................................................................
... more lines of dots omitted ...
...........................................................
Counting 611 lines.
```

Successful runs produce only dots and test counts.

22.12 Summary

The epigraph that begins this chapter expresses the difficulty of software testing: the job is never done. Software validation often requires writing other software, and that new software is itself likely to have bugs. In some cases, the test software is longer than that which it checks, and itself needs to be tested.

Branches of computer science and mathematics deal with proving the correctness of programs and theorems, but the proofs, which themselves are programs or theorems, are often longer than the original: the difficulties may then grow without bound.

Theories in mathematics are based on *axioms*, which are simple statements whose truth is (believed to be) self evident. In software, we are expected to believe that computer hardware and compilers are correct. Although hardware is likely to be reliable, because it cost a lot to develop, test, and produce, compilers are complex programs that are certain to have bugs. Both problems can be dealt with by testing with other hardware, and other compilers.

That last remark contains an important lesson for programmers. *For nontrivial projects, avoid use of programming languages that have only single implementations or that are available only on a single platform.* The useful life of software can often be much longer than that of the hardware on which it was initially developed, so designing software from the beginning to be portable is often the wisest approach. For major programming languages, there are usually additional tools, such as debuggers, prettyprinters, run-time profilers, security scanners, static and dynamic code-flow analyzers, and syntax checkers. They are often essential for large software projects.

Even when software is carefully written and widely used, obscure bugs can surface years later, as we found in the computation of the relative error in the ELEFUNT test suite, and discussed in **Section 4.25.4** on page 99, and in the binary-search algorithm flaw described in **Appendix I.5** on page 976.

The ELEFUNT tests report average and worst-case errors found by testing with arguments chosen from logarithmic distributions over intervals of interest. The tests also investigate special cases with exactly known results. However, the ELEFUNT tests give little information about how the errors are distributed. For that reason, throughout this book, we include graphs of measured errors for the elementary functions in the mathcw library. In some cases, the graphs of early implementations exposed argument regions with higher-than-desired errors, and led to algorithm modifications to reduce those errors. Comparison of the error graphs for different functions helped to identify functions whose algorithms needed improvement.

[6] Available at `http://cant.ua.ac.be/`.

23 Pair-precision elementary functions

DOUBLE, DOUBLE, TOIL AND TROUBLE.

— SHAKESPEARE'S *MacBeth* (1606).

DOUBLE YOUR PLEASURE, DOUBLE YOUR FUN.

— ADVERTISING SLOGAN (1956).

We discussed some of the limitations of pair-precision arithmetic in **Chapter 13** on page 353. Although our pair-precision routines for the basic operations of add, subtract, multiply, divide, square root, and cube root are *on average* correctly rounded in IEEE 754 arithmetic with the default *round-to-nearest* mode, we cannot guarantee correct rounding in general. Indeed, as the tables in **Section 13.20** on page 379 show, worst-case errors of almost four ulps can be discovered by extensive testing with random arguments. In the non-default rounding modes of IEEE 754, errors of almost 10 ulps are found. We should therefore expect errors at least that large on those historical architectures that have truncating arithmetic.

The inability to guarantee always-correct rounding in pair-precision arithmetic means that accurate conversion of constants stored as decimal strings is difficult. Polynomial approximations are then hard to use, unless we go to the trouble of expressing their coefficients in exact rational form, which depends on both the host precision, and the polynomial precision.

In this chapter, we therefore take a simpler approach to the computation of elementary functions, sacrificing speed by summing Taylor series on small intervals, and computing those sums until the ratio of the last term to the sum falls below half the pair-precision machine epsilon. We then use experimental testing to assess the accuracy of the functions over their argument range, and when necessary, make algorithmic adjustments to attempt to limit the worst-case errors to a few ulps. That is not satisfactory for general computation, but it is often sufficient for applications where pair-precision arithmetic is needed to get a few more digits beyond what is available in ordinary floating-point arithmetic.

Our choice of algorithms for some of the elementary functions in this chapter is guided by the prior published experience of Hida, Li, and Bailey for quad-precision arithmetic [HLB00]. In several cases, however, we improve their algorithms to enhance accuracy.

23.1 Pair-precision integer power

For evaluations of series, we sometimes need to compute integer powers of floating-point numbers. We saw in **Chapter 14** on page 411 that the general power function is one of the most difficult to compute accurately. However, if we only need exponents that are small integers, then repeated multiplication is likely to be the best approach.

In **Section 14.3** on page 414, we showed that we do not need to compute the n-th power by $n-1$ separate multiplications. Instead, we can use bitwise decomposition of the power and repeated squaring. For example, if $n = 25$, we have $n = 2^4 + 2^3 + 2^0 = 16 + 8 + 1$, and therefore, $x^{25} = x^{16}x^8x$. The only powers of x that we need are x, x^2, x^4, x^8, and x^{16}, which takes only four multiplications, and then we use two more to obtain x^{25}. We also showed in **Section 14.3** that it is sometimes possible to reduce the multiplication count even further, but in this section, we ignore that improvement to keep the code manageable.

The only additional complication that we need to handle is negative powers, which we do by first computing the positive power, then inverting the result. That is preferable to inverting first, because the error from the division then propagates into every factor of the product, and magnifies in each iteration. If underflow or overflow happens in that final inversion, then, with high probability, it would have happened in the last iteration anyway, had we inverted first.

Here is how the integer-power algorithm works in hoc code:

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_23

```
func ipow(x, n)                     \
{   # return x**n, for integer n
    v = 1
    k = fabs(int(n))

    while (k > 0)                   \
    {
        if (k & 1) v *= x

        k >>= 1

        if (k > 0) x *= x
    }

    if (n < 0) v = 1/v

    return (v)
}
```

The check for a positive k value before the final squaring in the loop adds to the cost, because it is executed on every iteration, and saves at most one multiplication for the entire loop. However, it prevents premature setting of the *overflow* or *underflow* exception flags in IEEE 754 arithmetic.

Zero arguments are handled by defining $0^0 \equiv 1$, which is what our function produces. There is no consensus among numerical programmers for what should be done for that special case. Our choice is simple to explain, and requires no extra code.

Negative arguments need no special treatment, because the power operation for integer exponents involves only multiplications, not logarithms.

No checks for Infinity or NaN arguments are needed, because any such arguments simply propagate to the returned function value, as expected. However, our handling of zero powers means that we also define $\infty^0 \equiv 1$, and $\mathrm{NaN}^0 \equiv 1$. That decision could be controversial, but we prefer the simplicity of our design choice that $x^0 \equiv 1$ for *all* possible floating-point values of x.

If underflow or overflow happens, the function result is subnormal, zero, or Infinity, as expected.

When working with pair-precision arithmetic in a language that does not allow operator overloading, it is best to first create a working implementation in ordinary arithmetic, and then transcribe the code, replacing operator expressions with pair-precision function calls. Our C translation of the hoc version looks like this:

```
void
PIPOW(fp_pair_t result, const fp_pair_t x, int n)
{
    /* return x**n in result */

    fp_pair_t t, v;
    static const fp_pair_t one = { FP(1.), FP(0.) };
    int k;

    PCOPY(v, one);
    PCOPY(t, x);

    k = (n < 0) ? -n : n;

    while (k > 0)       /* ceil(log2(|n|)) iterations */
    {
        if (k & 1)
            PMUL(v, v, t);

        k >>= 1;
        if (k > 0)
            PMUL(t, t, t);
    }

    if (n < 0)
        PDIV(v, one, v);

    PCOPY(result, v);
}
```

Apart from the addition of type declarations, and the use of function calls for arithmetic operations, the only significant change that we have to make is the introduction of the temporary variable t in the loop to take the place of x, because x is now a constant that we cannot modify.

The worst-case expected error in the computed integer power is roughly bounded by the maximum number of multiplications, $2\lceil \log_2 n \rceil - 1$, times the average error in PMUL(). For negative powers, add to that the average error in PDIV(). Our applications in the mathcw library require only small values of $|n|$, for which PIPOW() is acceptably accurate.

23.2 Pair-precision machine epsilon

In **Section 13.1** on page 354, we observed that the normal algorithm for finding the machine epsilon cannot be used with pair-precision arithmetic because the exponents of the high and low parts are not related by a constant offset, so the calculation produces a low part that is too small.

We therefore provide a function to compute the generalized machine epsilon in pair-precision arithmetic, using construction, rather than iteration, to arrive at the desired result. With t-digit arithmetic, the normal machine epsilon is β^{1-t}, and the pair-precision machine epsilon is β^{1-2t}. The generalized machine epsilon, which is the smallest *positive* number whose sum with x differs from x, can then be obtained for positive x from the computation $\mathrm{fl}(x \times (1+\epsilon)) - \mathrm{fl}(x)$, where the subtraction serves to trim unwanted trailing digits. The code looks like this:

```
void
PEPS(fp_pair_t result, const fp_pair_t x)
{
    fp_pair_t tmp;
    static fp_pair_t one_minus_eps, one_plus_eps;
    static fp_t minfloat;
    static int do_init = 1;

    if (do_init)
    {
        fp_pair_t eps;
        volatile fp_t t;

        PSET(eps, (fp_t)B, ZERO);
        PIPOW(eps, eps, (1 - 2 * T));
        PSET(one_minus_eps, ONE, -eps[0] / (fp_t)B);
        PSET(one_plus_eps, ONE, eps[0]);
        minfloat = FP_T_MIN;
        t = minfloat;

        while (t > ZERO)
        {
            minfloat = t;
            t = minfloat / (fp_t)B;
            STORE(&t);
        }
```

```
            do_init = 0;
        }

        if (ISINF(x[0]))
        {
            fp_t q;

            q = SET_EDOM(QNAN(""));
            PSET(result, q, q);
        }
        else if (ISNAN(x[0]))
            PSET(result, x[0], x[0]);
        else if (x[0] == ZERO)
            PSET(result, minfloat, ZERO);
        else if (x[0] > ZERO)
        {
            PMUL(tmp, x, one_plus_eps);
            PSUB(result, tmp, x);
        }
        else /* x[0] < ZERO */
        {
            PMUL(tmp, x, one_minus_eps);
            PSUB(result, tmp, x);
        }
    }
```

On the first call only, the initialization block computes some needed constants. The smallest positive *normal* floating-point value is available as a compile-time constant in C, but there is no separate constant for systems that support gradual underflow to subnormal values, so we have to compute `minfloat` via exact division by the base, with the usual subterfuges to disable higher intermediate precision.

The cases of Infinity and NaN arguments require special handling: both produce NaN results. A zero argument produces the smallest representable floating-point number. Nonzero normal arguments require a single multiplication and subtraction, but to ensure a positive result, for a negative argument, we compute $\mathrm{fl}(x \times (1 - \epsilon/\beta)) - \mathrm{fl}(x)$. The factor $(1 - \epsilon/\beta)$ is the representable number nearest to one that is less than one.

23.3 Pair-precision exponential

The convergence of the Taylor series of the exponential function, $\exp(x) = \sum_{n=0}^{\infty} x^n/n!$, can be improved by making x small. One way to do that is to split x into a sum of parts, one of which we handle with a power operation, and the other with a short sum. If we pick suitable integers k and m, then, for base β, we can do the split like this:

$$\begin{aligned} x &= m\ln(\beta) + kr, \\ \exp(x) &= \exp(m\ln(\beta))\exp(kr) \\ &= \big(\exp(\ln(\beta))\big)^m \big(\exp(r)\big)^k \\ &= \beta^m \big(\exp(r)\big)^k. \end{aligned}$$

Multiplication by β^m is an *exact* scaling, and the `SCALBN()` function family does the job. We pick m such that $m\ln(\beta)$ is closest to x, because that guarantees that $|kr| \le \frac{1}{2}$, reducing the number of terms needed from the Taylor series. It then allows us to predict that number, because the largest value of the n-th term is $1/(2^n n!)$, and in *round-to-nearest* mode, we can stop as soon as a term falls below half the pair-precision machine epsilon.

The best choice of k is a power of the base, because that means that r can be most accurately determined by exact scaling of $x - m\ln(\beta)$. If, in addition, the base is 2, then the k-th power takes a minimal number of multiplications, $\log_2(k)$, as we showed in our analysis of the integer power function in **Section 14.3** on page 414 and **Section 23.1** on page 777.

Increasing k reduces r, speeding the convergence of the Taylor series, but the fewer terms that we sum, the higher the power of $\exp(r)$ that we need. There is clearly a tradeoff to be measured by timing experiments, or worked out manually with operation counts. However, there is a nasty problem lurking: accuracy loss. When r is small, $\exp(r) = 1 + \delta$, where $\delta \approx r$ is also small. Then $\big(\exp(r)\big)^k = (1+\delta)^k = 1 + k\delta + k(k-1)\delta^2/2 + \cdots$. Because of finite precision in the storage of the value $1 + \delta$, δ is determined comparatively inaccurately, and its error is magnified by k in forming the power. The original algorithm [HLB00] had $k = 512$, but that introduces about nine bits of error into the power. Numerical experiments over the argument range for which the exponential is representable show that a choice of $k = 1$ produces the lowest average error for binary arithmetic; larger values of k are always worse. We make some further comments on the choice of k at the end of this section on page 785.

The value of r can be either negative or positive, because we choose m to minimize r. The series contains both odd and even terms, so there can be subtraction loss if r is negative. However, because $\exp(-|r|) = 1/\exp(|r|)$, we could sum the series for the positive case, and then take the reciprocal if r is negative. We examine that choice on page 785.

As with many of the elementary functions, the argument reduction step is critical. Mathematically, we have two simple operations:

$$\begin{aligned} m &= \operatorname{round}(x/\ln(\beta)), \\ r &= (x - m\ln(\beta))/k. \end{aligned}$$

The computation of m is easy, and even if, in nearly halfway cases, the value of $x/\ln(\beta)$ is rounded in the wrong direction, then r changes sign, but has a similar magnitude. The difficult part is the second operation, for which accurate computation requires a double-length product $m\ln(\beta)$. If we have a *correct* fused multiply-add library function, then the computation can be done accurately using that function:

$$r = \operatorname{fma}(-m, \ln(\beta), x)/k.$$

Unfortunately, that facility is not widely available. Cody and Waite observed that m is in practice small, because the limited floating-point exponent range means that the exponential function soon reaches the underflow and overflow regions. For example, with the 32-bit IEEE 754 format, the argument range for which the function is finite and nonzero is about $(-103.28, +88.73)$, and even for the 128-bit format, the argument range is about $(-11433, +11357)$. They correspond to m ranges of $[-149, +128]$ and $[-16494, +16384]$, respectively, and hold for pair-precision formats as well, because they have the same exponent ranges. Those values of m can be represented in 8 to 16 bits. If we split $\ln(\beta)$ into a high part with a few bits, and a low part with the remaining bits, then the product of m and the high part is exactly representable. The computation can then be done like this, as long as there is a guard digit in the adder:

$$r = \big((x - m\ln(\beta)_{\mathrm{hi}}) - m\ln(\beta)_{\mathrm{lo}}\big)/k.$$

The first two terms, their difference, and the division by k are exact, so the only error comes from the product $m\ln(\beta)_{\mathrm{lo}}$ and its subtraction.

When there is no guard digit, the argument must be split into integer and fractional parts, $x = x_{\mathrm{int}} + x_{\mathrm{fract}}$, and the computation rearranged like this:

$$r = \big(((x_{\mathrm{int}} - m\ln(\beta)_{\mathrm{hi}}) + x_{\mathrm{fract}}) - m\ln(\beta)_{\mathrm{lo}}\big)/k.$$

The split of x is an exact operation. The first two terms are nearly equal, and when their exponents are identical, the subtraction is exact. The final two terms are smaller, and produce the only rounding errors in the expression. For binary pair-precision arithmetic, the difference between the two formulas for r is dramatic, with errors as large as 50 ulps in `exp()` from the simpler argument reduction formula.

For base 2, Cody and Waite recommend a split with $\ln(\beta)_{\mathrm{hi}} = 355/512$, which takes just nine bits. For IEEE 754 arithmetic, we could even use a few more bits, with $\ln(\beta)_{\mathrm{hi}} = 22\,713/32\,768$, which needs 14 bits. Numerical experiments in ordinary arithmetic show little difference between the two splits in the accuracy of the exponential function. For pair-precision arithmetic, we split the constant $\ln(\beta)$ into three successive chunks. The first chunk forms the high part of $\ln(\beta)_{\mathrm{hi}}$, with a zero low part, and the remaining two chunks make up $\ln(\beta)_{\mathrm{lo}}$. For example, in single-precision decimal arithmetic, we have $\ln(10) = 2.302\,585\,092\,994\,045\,684\,017\ldots$, and the initializer looks like this:

```
static const fp_pair_t ln_base_hi = { FP(2.302585), FP(0.) };
static const fp_pair_t ln_base_lo = { FP(9.299400e-08), FP(4.568402e-14) };
```

The underflow limit for the exponential function is reached when the argument is the logarithm of the smallest positive nonzero representable number. The overflow limit is at the logarithm of the largest representable number. Checking those two cutoffs first eliminates unnecessary computation for most of the possible floating-point arguments.

When x is sufficiently small, the Taylor series can be truncated to its first term, apart from issues of rounding and the *inexact* exception flag. For *round to nearest* operation, for positive x, that happens when x lies below half the machine epsilon, and for negative x, when $|x|$ is below $\frac{1}{2}\epsilon/\beta$. For other rounding modes, that economization to a single term is wrong, but we can correct it by returning $1 + x$ instead of 1. That action also sets the *inexact* flag.

With those details taken care of, we can now show a simple implementation of the algorithm in hoc:

```
__LOG_B := +0x1.62e42fefa39ef35793c7673007e5ed5e81e69p-1
__INV_LOG_B := +0x1.71547652b82fe1777d0ffda0d23a7d11d6aefp+0
__LOG_B_HI := +0x1.63p-1
__LOG_B_LO := -0x1.bd0105c610ca86c3898cff81a12a17e1979b3p-13
__LN_MAXNORMAL := log(MAXNORMAL)
__LN_MINSUBNORMAL := log(MINSUBNORMAL)

func pexp(x)                                  \
{   # Return exp(x)

    if (isnan(x))                             \
        return (x)                            \
    else if (x < __LN_MINSUBNORMAL)           \
        return (0)                            \
    else if (x > __LN_MAXNORMAL)              \
        return (INF)                          \
    else if (fabs(x) < (0.25 * __MACHEPS__))  \
        return (1 + x)                        \
    else                                      \
    {
        k = BASE
        m = round( x * __INV_LOG_B )
        r = ((x - m * __LOG_B_HI) - m * __LOG_B_LO) / k
        rabs = fabs(r)
        tn = rabs
        exp_r = tn

        for (n = 2; n <= 24; ++n)             \
        {
            tn *= rabs / n
            new_exp_r = exp_r + tn

            if (exp_r == new_exp_r) break

            exp_r = new_exp_r
        }

        exp_r += 1

        if (r < 0) exp_r = 1 / exp_r

        return (scalbn(ipow(exp_r, k), m))
    }
}
```

That code is valid only for base-2 systems, because of the values that we assign to the four constants involving the logarithm of the base, but that could easily be repaired. It is important that those constants be accurate to the last bit, so we cannot portably rely on the value of `log(BASE)` and related run-time expressions.

Although we limit the sum to 24 terms (enough for the precision of the 128-bit IEEE 754 binary format when r reaches its maximum value of $\frac{1}{2}\log(2)$), in practice, many fewer are needed, and the `break` statement ensures that the loop terminates as soon as full accuracy has been reached.

We improve accuracy at little cost by suppressing addition of the leading term of the series until after the loop.

The translation to pair-precision code in C is reasonably obvious:

```
void
PEXP(fp_pair_t result, const fp_pair_t x)
{
    fp_t tmp, xval;
    static const fp_pair_t one = { FP(1.), FP(0.) };
    static fp_t cuthi, cutlo, cutseries, k_inv;
    static int do_init = 1;
    static int k;
    static volatile fp_t tiny = FP_T_MIN;

    if (do_init)
    {
        cuthi = HALF * FP_PAIR_T_EPSILON;
        cutlo = -cuthi / (fp_t)B;
        cutseries = HALF * cutlo;

#if defined(KEXP)
        k = KEXP;
#else
        k = 1;
#endif

        k_inv = ONE / (fp_t)k;    /* exact */
        do_init = 0;
    }

    xval = PEVAL(x);

    if (ISNAN(xval))
    {
        tmp = SET_EDOM(QNAN(""));
        PSET(result, tmp, tmp);
    }
    else if (xval < LN_MINSUBNORMAL)
    {
        STORE(&tiny);
        tmp = tiny * tiny;        /* intentional underflow! */
        PSET(result, tmp, ZERO);
    }
    else if (xval > LN_MAXNORMAL)
    {
        tmp = SET_ERANGE(INFTY());
        PSET(result, tmp, tmp);
    }
    else if ( (cutlo < xval) && (xval < cuthi) )
        PADD(result, one, x);
    else
    {
        fp_pair_t exp_r, r, s, sum, t, tn, tnu, u;
        int m, n;
        static const fp_pair_t half = { FP(0.5), FP(0.) };
        static const fp_pair_t six = { FP(6.), FP(0.) };

#if B != 2
        int is_r_neg;
#endif

        m = (FABS(xval) < (HALF * LN_B)) ? 0 :
            (int)LROUND(xval * INV_LN_B);

        if (m == 0)
            PCOPY(r, x);
        else
        {
            fp_pair_t mm, x_int, x_fract;

            PSET(x_int, TRUNC(x[0]), ZERO);
            PSUB(x_fract, x, x_int);
            PSET(mm, (fp_t)m, ZERO);
            PMUL(s, mm, ln_base_hi);
            PMUL(t, mm, ln_base_lo);
            PSUB(r, x_int, s);
            PADD(r, r, x_fract);
            PSUB(r, r, t);
        }

        if (k > 1)
        {
            r[0] *= k_inv;
            r[1] *= k_inv;
        }

#if B != 2
        is_r_neg = (r[0] < ZERO);

        if (is_r_neg)
            PABS(r, r);
#endif

        PDIV(tn, r, six);
        PADD(sum, half, tn);

        for (n = 4; n <= 100; ++n)
        {
            PSET(u, (fp_t)n, ZERO);
            PDIV(tnu, tn, u);
            PMUL(tn, r, tnu);

#if B == 2
            if ( (-cutseries < tn[0]) && (tn[0] < cutseries) )
                break;
#else
            if (tn[0] < cutseries)
                break;
#endif
```

```
                PADD(sum, sum, tn);
            }

            PMUL(sum, sum, r);
            PMUL(sum, sum, r);
            PADD(sum, r, sum);
            PADD(exp_r, one, sum);

#if B != 2
            if (is_r_neg)
                PDIV(exp_r, one, exp_r);
#endif

            if (k > 1)
                PIPOW(result, exp_r, k);
            else
                PCOPY(result, exp_r);

            if (m != 0)
            {
                result[0] = SCALBN(result[0], m);
                result[1] = SCALBN(result[1], m);
            }
        }
}
```

The cutoff values *should* be compile-time constants, but C89 and C99 permit the machine-epsilon macros in <float.h> to expand to run-time values, rather than numerical constants, and some C implementations work that way. We therefore have to compute the values during the first call to the routine, even though on many systems, the compiler could compute them at translation time.

The special cases are handled by testing just the high-order component. We follow the C99 Standard in setting errno for NaN and large positive arguments. For large-magnitude negative arguments, computation of the square of the smallest normal number sets the *underflow* exception flag, and produces a zero result in three of the four IEEE 754 binary rounding modes. In the fourth mode, *round-to-plus-infinity*, the result is the smallest representable number.

Small arguments in the range (cutlo, cuthi) result in evaluation of $1 + x$, which then properly reflects the current rounding mode: the result can be $1 - \epsilon/\beta$, 1, or $1 + \epsilon$, depending on the mode, and on the sign of x.

When the special tests all fail, control enters the final else block, where we sum the Taylor series for $\exp(r)$, the exponential of the reduced argument, r. The sum delays addition of the first two terms until last, so that the result is usually correct to within rounding error. In addition, the leading term in the loop summation is $\frac{1}{2}$, so there is no bit loss from wobbling precision in a hexadecimal base.

In the hoc prototype, we summed the series for $\exp(|r|)$, and then computed its reciprocal when r is negative. Graphs of the error in numerical experiments with the pair-precision code show that it is better to avoid introducing the additional error from that division. Because we know that r lies in $[-\frac{1}{2}\log(\beta), +\frac{1}{2}\log(\beta)]$, by examination of the ratios of successive terms of the series, it is easy to show that there is no subtraction loss possible for base $\beta = 2$. Loss of a single digit is possible for larger bases, but only from the first two terms. The safe solution is therefore to sum the series for $\exp(r)$ for binary arithmetic, and that for $\exp(|r|)$ for other bases. The loop limit must be adjusted upward to 100, large enough for the worst case of 256-bit hexadecimal arithmetic.

For about half the floating-point range, we have $m = 0$, so we check for that case in two places to avoid doing unnecessary work.

As we noted in Section 13.1 on page 354, using comparisons in sums of pair-precision values is unwise. To avoid that problem, and lower the cost of the loop-termination test, we do it in ordinary precision, using only the high part in the test.

Figure 23.1 on the next page shows plots of the measured errors in the pair-precision exponential functions for the *round to nearest* case, where all but about 2% of the results are correctly rounded. Plots for data from other rounding modes are qualitatively similar, so we omit them.

Figure 23.2 compares the accuracy of the 64-bit decimal function pexpd() with $k = 1$ and $k = 10$ in the argument

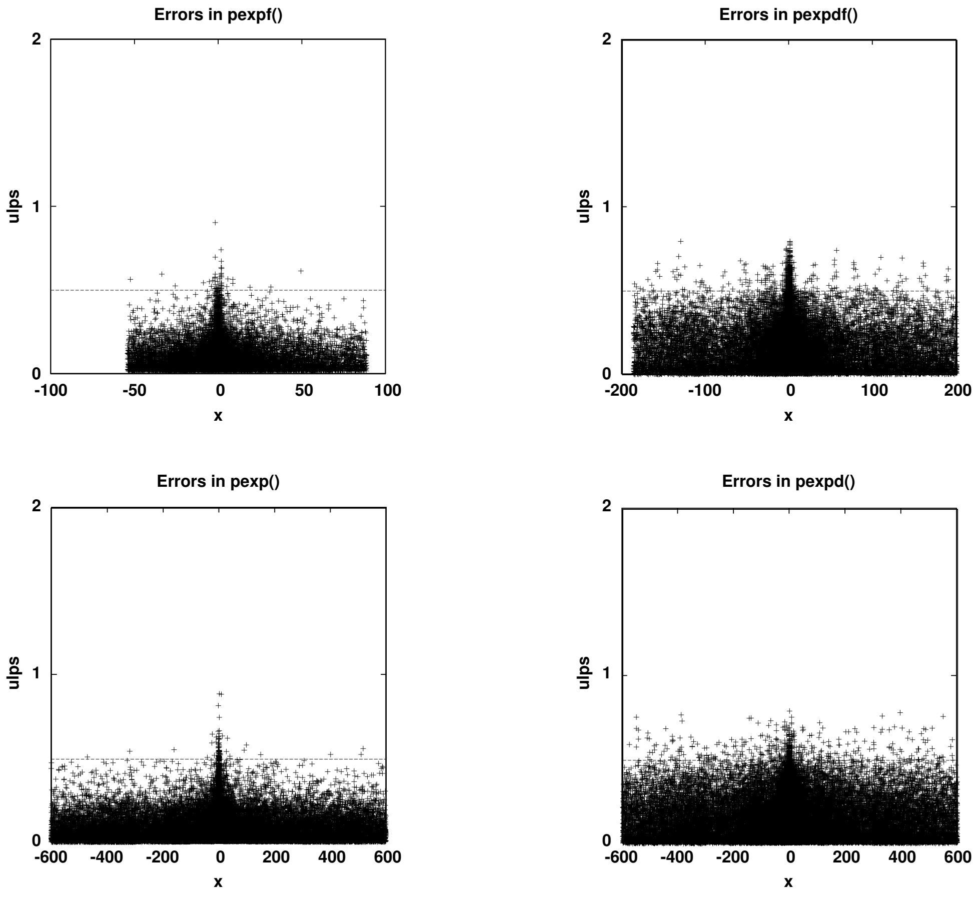


Figure 23.1: Errors in pair-precision exponential functions. The horizontal dotted line at 0.5 ulps marks the boundary below which results are correctly rounded.

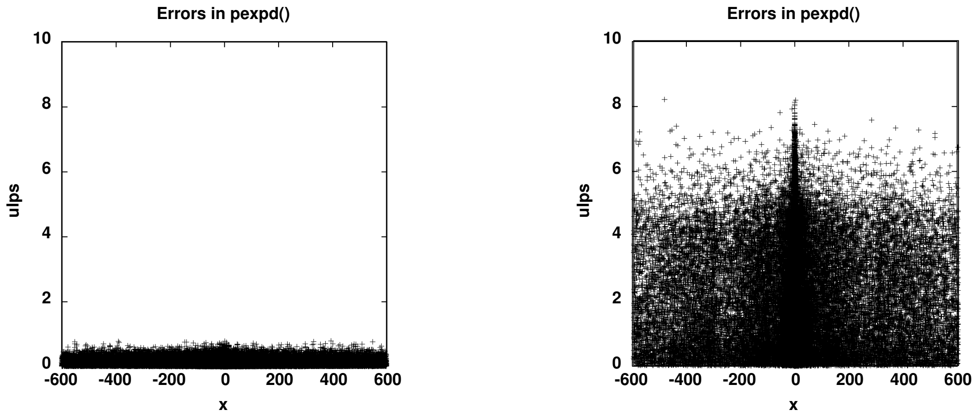


Figure 23.2: Errors in the decimal pair-precision `PEXPD()` function. The left-hand plot is for the default $k = 1$ in the argument reduction in `pexpd()`, whereas the right-hand plot is an experiment with $k = 10$, where k is defined at the start of Section 23.3 on page 780.

Plots for the single-precision decimal function, `pexpdf()`, are similar, and thus, not shown.

reduction. The penalty for a faster-converging sum is an unacceptable ten-fold increase in the error.

23.4 Pair-precision logarithm

We saw in **Section 2.6** on page 10 that the Taylor series for $\log(x)$ around $x = 1$ has terms of the form $(-1)^{k+1}(x-1)^k/k$. Unlike the exponential function, there are no factorials in the denominators of the logarithm series to make the terms fall off quickly. Brute-force summation of the series is therefore not a suitable way to compute the logarithm, unless x is close to 1.

Because the logarithm and exponential are inverse functions, we can use Newton–Raphson iteration to solve for a root of $f(y) = \exp(y) - x$ for fixed x. The solution, $y = \log(x)$, is the logarithm that we seek. The iteration is

$$\begin{aligned} y_{n+1} &= y_n - f(y_n)/f'(y_n) \\ &= y_n - (\exp(y_n) - x)/(\exp(y_n)) \\ &= y_n + x\exp(-y_n) - 1. \end{aligned}$$

The logarithm of the high part is a suitable starting value, and because convergence is quadratic, just one or two iterations suffice. Accuracy should be only a bit worse than that of the exponential, because there are only three additional floating-point operations.

Near convergence, $x\exp(-y_n) - 1 \approx 0$, so we again have a case where there is massive subtraction loss that can be computed accurately if we have a fused multiply-add operation. Unfortunately, we lack pair-precision code to compute it.

The error magnification of the exponential function, which grows with the argument magnitude, is another concern. However, that only matters when $|y_n|$ is large, and that value dominates the much smaller $x\exp(-y_n) - 1$ term in the sum for y_{n+1}. Nevertheless, it is an issue that we can investigate with numerical experiments.

The simplicity of the iteration suggests that the logarithm code should be short. However, we need tests for NaN, negative, and zero arguments so that we can handle them according to the C99 specification.

Testing of an initial version of the logarithm code exposed a problem with the Newton–Raphson iteration: large accuracy loss for $x \approx 1$, where the logarithm is small. That suggests switching to the Taylor series, but its terms alternate in sign, and converge slowly.

The solution to that problem is another series that can be found in standard mathematical tables, or with a symbolic-algebra system:

$$\begin{aligned} x &= (1+\delta)/(1-\delta), \\ \delta &= (x-1)/(x+1), \\ \log(x) &= \log\big((1+\delta)/(1-\delta)\big) \\ &= 2\delta\big(1 + \delta^2/3 + \delta^4/5 + \cdots + \delta^{2m}/(2m+1) + \cdots\big). \end{aligned}$$

For x in $[\frac{1}{2}, \frac{3}{2}]$, the new variable δ varies smoothly over the interval $[-\frac{1}{3}, +\frac{1}{5}]$. When $x \approx 1$, δ is small, and the parenthesized series now has only positive terms, and converges more quickly than the Taylor series in x because it contains only even powers.

Here is hoc code to compute the logarithm from the new series, assuming only positive nonzero arguments:

```
func plog(x)                          \
{
    if (fabs(x - 1) > 0.25)           \
    {
        y = log(0.9999 * x)             # starting estimate
        y += x * exp(-y) - 1
        y += x * exp(-y) - 1
        y += x * exp(-y) - 1
    }                                 \
    else                              \
    {
        d = (x - 1) / (x + 1)
        dd = d * d
        d_to_k = dd
        sum = d_to_k / 3

        for (k = 4; k <= 40; k += 2)  \
        {
            d_to_k *= dd
            term = d_to_k / (k + 1)

            if ((sum + term) == sum) break

            sum += term
        }

        y = 2 * d * (1 + sum)
    }

    return (y)
}
```

As we did for PEXP(), we omit the leading term of the sum until after the loop, thereby gaining some extra precision, and ensuring that the sum is almost always correctly rounded.

In hexadecimal arithmetic, the phenomenon of wobbling precision requires that the final assignment to y be rewritten

```
        y = 4 * d * (0.5 + 0.5 * sum)
```

to avoid unnecessary loss of up to three bits in the significand.

The term limit is set at $k = 40$, based on the worst case in 128-bit arithmetic of $x = \frac{3}{4}$ where $\delta = -\frac{1}{7}$, so 21 terms are summed. For the 80-bit, 64-bit, and 32-bit binary formats, we need 12, 10, and 5 terms, respectively.

Graphical results from numerical experiments with an initial pair-precision version of the hoc prototype reveal another problem: the error grows to about 10 ulps as x approaches the δ series region from either side, although the error remains below about 3 ulps away from that region. The slow convergence of the series prevents its use much outside the interval $[\frac{3}{4}, \frac{5}{4}]$.

One possibility is to expand the logarithm about, say, 0.5 and 1.5, but that has the problem that the coefficients are no longer easily representable in pair-precision arithmetic.

A workable alternative is to use the identity $\log(x) = \log(x/s) + \log(s)$ for some suitable scale factor s to shift the computation into the series region. For $x > \frac{5}{4}$, pick $s = 2$, and for $x < \frac{3}{4}$, choose $s = \frac{1}{2}$. The $\log(s)$ terms then have identical magnitude, so only one pair-precision constant is needed. Furthermore, the scaling is *exact* in the common case of binary floating-point arithmetic, so the shift introduces only a single additional rounding error from the addition of $\log(s)$. For decimal and hexadecimal arithmetic, we may incur two rounding errors, from the addition and the multiplication.

The constant $\log(2)$ can be computed in a one-time initialization block by summing a series of suitable exactly representable stored rational numbers. For decimal arithmetic, the numerators are exact seven-digit sequences from the decimal constant, and the denominators are successive powers of 10^7. Otherwise, we use denominators that are powers of 2^{24}, based on the data in Table H.1 on page 948 that show that all current, and most historical, architectures provide at least 24 significand bits. In addition, the value 2^{24} is an exact power of the base for $\beta = 2, 4, 8, 16$, and 256. Only one denominator needs to be stored, and at most 21 numerators allow us to represent the constant exactly up to 147 digits, enough for pair-precision versions of the code in 70D octuple-precision arithmetic. Similar reconstructions are needed in other pair-precision functions, so we provide the PCON() function family to do the job:

```
void
PCON(fp_pair_t result, int n, const fp_t table[/* n */],
     const fp_t scale)
{   /* result = sum(k=0:(n-1)) table[k] * scale**k */
    fp_pair_t t;
    fp_t scale_to_k;
    int k;

    PSET(result, ZERO, ZERO);
    scale_to_k = scale * scale;

    for (k = 2; (k < n) && (scale_to_k != ZERO); ++k)
    {
        fp_pair_t t1, t2, t3;

        PSET(t1, scale_to_k, ZERO);
        PSET(t2, table[k], ZERO);
        PMUL(t3, t1, t2);
        PADD(result, result, t3);
        scale_to_k *= scale;           /* exact */
    }

    PSET(t, table[1] * scale, ZERO);
    PADD(result, result, t);
    PSET(t, table[0], ZERO);
    PADD(result, result, t);
}
```

The loop terminates early if `scale_to_k` underflows to zero. The initial table entry is zero for some of the constants that we need, so the first *two* terms are added last to enhance accuracy.

Further graphical analysis of numerical experiments from pair-precision code that implements the δ series, two shift regions, and the Newton–Raphson iteration show that further improvement is possible by extending the δ-series region downward, and widening the shift regions. For the latter, that means multiple recursive calls to the pair-precision logarithm function, each of which incurs one or two rounding errors until the final call sums the δ series. The shift-region cutoffs are chosen so that the errors do not change noticeably at the boundaries with the Newton–Raphson iteration regions. The cutoff values are all exactly representable in the available precision of current and historical floating-point systems, and are selected so as to prevent an argument from being moved from one shift region into the other, because that would produce infinite recursion.

The graphical and numerical experiments show that the δ-series code produces errors of up to 3 ulps in binary arithmetic, and up to 2 ulps in decimal arithmetic, notably worse than we found for the exponential function. Switching to the normal Taylor series for the logarithm about $x = 1$ pushes the errors in both binary and decimal arithmetic down below 0.8 ulps, and most results are below 0.5 ulps, and thus, correctly rounded. The penalty for the enhanced accuracy is increased computation time.

Here is the final C translation of the hoc program for the pair-precision logarithm. It uses the result of `PCMP()` as a convenient `case` statement selector, and replaces the δ series by the normal Taylor series:

```
void
PLOG(fp_pair_t result, const fp_pair_t x)
{
    fp_t r, xval;
    int k;

#if B != 2
    static const fp_pair_t half  = { FP(0.5), FP(0.) };
    static const fp_pair_t two   = { FP(2.), FP(0.) };
#endif

    static const fp_pair_t one   = { FP(1.), FP(0.) };
    static const fp_pair_t zero  = { FP(0.), FP(0.) };
    static fp_pair_t Log_2       = { FP(0.), FP(0.) };;
    static int do_init = 1;
    fp_pair_t sum, t1, t2, term;
    if (do_init)
    {
        PCON(Log_2, (int)elementsof(log_2_table), log_2_table,
             log_2_scale);
        do_init = 0;
    }

    switch (PCMP(x, zero))
    {
    default:        /* should not happen */ /* FALLTHROUGH */
    case -2:        /* x is a NaN */        /* FALLTHROUGH */
    case -1:        /* x < 0 */
        r = SET_EDOM(QNAN(""));
        PSET(result, r, r);
        break;

    case 0:         /* x == 0 */

#if defined(HAVE_IEEE_754)
        r = -INFTY();
#else
        r = SET_ERANGE(-INFTY());
#endif

        PSET(result, r, r);
        break;

    case 1:             /* x > 0 */
        xval = PEVAL(x);

        if ( (FP(0.5) < xval) && (xval < FP(1.5)) )
        {
            fp_pair_t xm1_to_k, xm1, xm1sq;

            PSUB(xm1, x, one);
            PMUL(xm1_to_k, xm1, xm1);
            PCOPY(xm1sq, xm1_to_k);
            PCOPY(sum, zero);

            for (k = 3; k <= 460; ++k)
            {   /* sum log(x) Taylor series about x = 1 */
                fp_t lo_abs;

                PMUL(xm1_to_k, xm1_to_k, xm1);
                PSET(t1, (fp_t)k, ZERO);
                PDIV(term, xm1_to_k, t1);
                lo_abs = FABS(sum[1]);

                if (lo_abs == (lo_abs + term[0]))
                    break;

                if (IS_ODD(k))
                    PADD(sum, sum, term);
                else
                    PSUB(sum, sum, term);
            }
#if B == 2
```

```
            term[0] = HALF * xm1sq[0];
            term[1] = HALF * xm1sq[1];
#else
            PMUL(term, half, xm1sq);
#endif

            PSUB(sum, sum, term);
            PADD(result, sum, xm1);
        }
        else if ( (FP(1.25) < xval) && (xval <= FP(128.)) )
        {   /* compute log(x) = log(x / 2) + log(2) */

#if B == 2
            PSET(t1, x[0] * HALF, x[1] * HALF); /* exact */
#else
            PMUL(t1, x, half);
#endif

            PLOG(t2, t1);
            PADD(result, t2, Log_2);
        }
        else if ( (FP(0.0078125) <= xval) && (xval < FP(0.625)) )
        {   /* compute log(x) = log(x * 2) - log(2) */

#if B == 2
            PSET(t1, x[0] * TWO, x[1] * TWO);   /* exact */
#else
            PMUL(t1, x, two);
#endif

            PLOG(t2, t1);
            PSUB(result, t2, Log_2);
        }
        else
        {
            fp_pair_t y;

            PSET(y, LOG(xval), ZERO);

            for (k = 0; k < 2; ++k)
            {   /* Newton--Raphson iteration */
                fp_pair_t t, yneg;

                PNEG(yneg, y);
                PEXP(t, yneg);
                PMUL(t, x, t);
                PSUB(t, t, one);
                PADD(y, y, t);
            }

            PCOPY(result, y);
        }
        break;
    }
}
```

For NaN or negative arguments, C99 requires a domain error, and for IEEE 754 arithmetic, the raising of the *invalid* exception flag. The QNAN() function sets the flag, and the SET_EDOM() wrapper sets the errno global value.

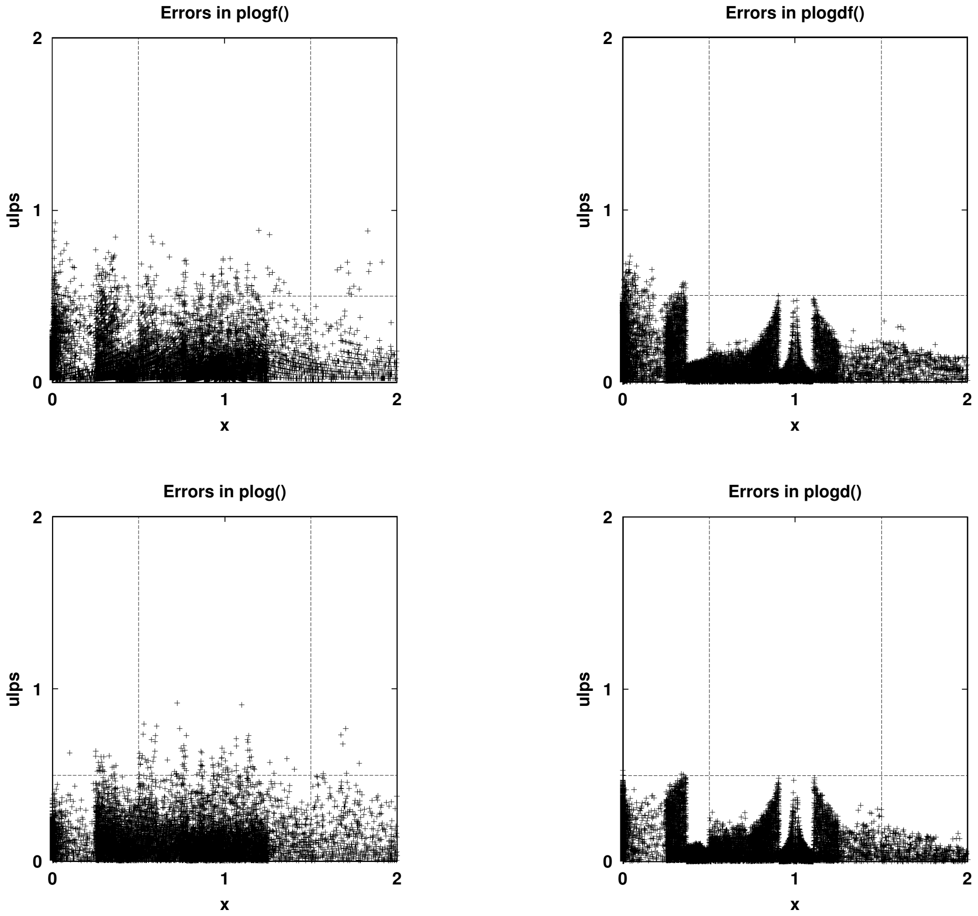


Figure 23.3: Errors in pair-precision logarithm functions near the series region. Results between the vertical dotted lines are from the Taylor series.

For an argument of ±0, in IEEE 754 arithmetic, C99 mandates the raising of the *divbyzero* exception flag, and a return value of $-\infty$. The `INFTY()` function in the `mathcw` library ensures that both conditions are met. For other floating-point architectures, C99 allows a range error to be set. On such systems, our `INFTY()` function returns the largest representable floating-point number, and we set the `errno` global value so that the caller can detect the range error.

We use the Taylor series for arguments in $(\frac{1}{2}, \frac{3}{2})$, region shifts for arguments in $[\frac{1}{128}, \frac{5}{8})$ and $(\frac{5}{4}, 128]$, and otherwise, the Newton–Raphson iteration. The loop limit is increased to $k = 460$ (the value needed for the 256-bit formats) because of the widened series range, but we make an early exit from the loop when the high component of the term is small enough that it does not contribute to the absolute value of the low component of the sum.

As in the exponential function, the loop sums terms after the second to machine precision, and the larger first two terms are added last. Most of the rounding error comes from that final operation.

More often, the code uses two steps of the self-correcting Newton–Raphson iteration. The final error in that case consists of the small errors from the add, subtract, and multiply, and the larger error from the exponential. The error in the logarithm is then expected to be at most a few ulps.

Figure 23.3 shows the errors in the logarithm functions around the series region, and Figure 23.4 shows the errors over the entire argument range. They demonstrate that the pair-precision logarithm functions are correctly rounded, except near $x = 1$, where errors up to 1 ulp are possible. Test with other rounding modes produce similar plots.

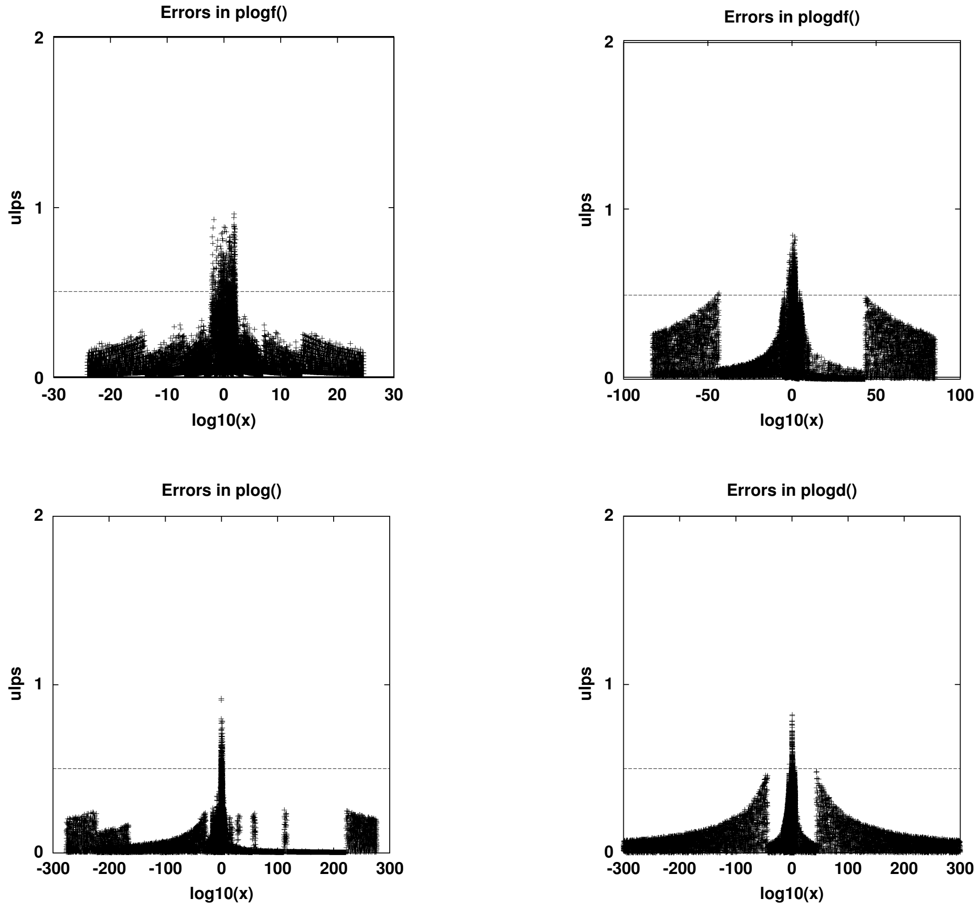


Figure 23.4: Errors in pair-precision logarithm functions.

23.5 Pair-precision logarithm near one

When a computation requires $\log(1 + x)$ for small x, direct use of the logarithm function loses accuracy from the addition $1 + x$. In Section 10.4 on page 290, we introduce a new function, $\mathrm{log1p}(x)$, that provides a simple solution to that problem. Unfortunately, the trick used there to recover $\log(1 + x)$ from $\log(\mathrm{fl}(1 + x))$ is unsatisfactory in pair-precision arithmetic: an initial implementation of `PLOG1P()` with that algorithm exhibited errors of up to 3 ulps. Instead, we use a two-term Taylor series for tiny arguments, the normal Taylor series when $|x| < \frac{1}{2}$, and otherwise, call `PLOG()` to evaluate $\log(1 + x)$. The series convergence is poor unless $|x|$ is small, so we sacrifice speed for accuracy.

We omit the code for `PLOG1P()`, but Figure 23.5 on the following page shows the measured errors near the series region.

23.6 Pair-precision exponential near zero

In Section 10.2 on page 273, we describe the computation of `expm1()`, which evaluates $\exp(x) - 1$ accurately for small arguments, where direct computation of the right-hand side would otherwise lose accuracy from the subtraction of two nearly equal values.

For small-magnitude arguments, the $\mathrm{expm1}(x)$ function reduces to just the first term of its Taylor series. For larger argument magnitudes, we use a rational polynomial approximation. Otherwise, when there is no possibility

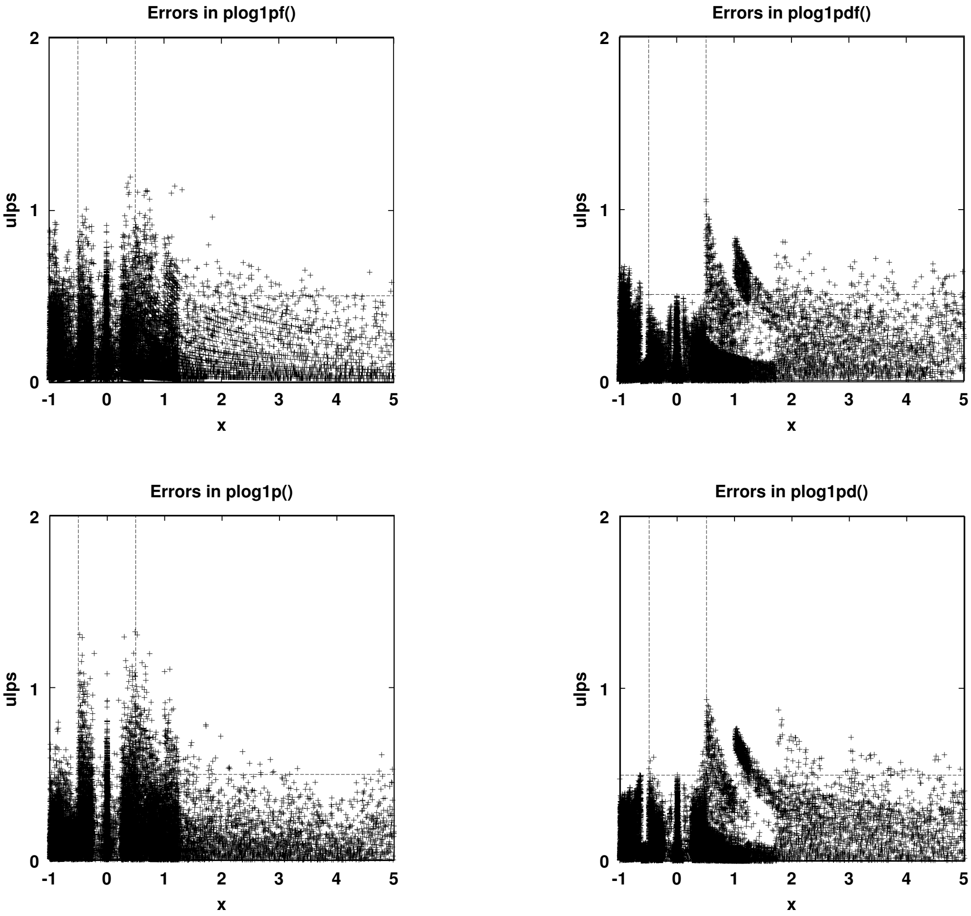


Figure 23.5: Errors in pair-precision logarithm-plus-one functions near the series region. Results between the vertical dotted lines are from the Taylor series.

of subtraction loss, we can compute it directly from $\exp(x) - 1$.

A rational polynomial approximation is unsuitable for pair-precision arithmetic because of the difficulty of representing the polynomial coefficients correctly. One possible solution is to use Newton–Raphson iteration. Given x, to find $y = \exp(x) - 1$, rearrange and take logarithms to get $\log(1 + y) = x$, and then set $f(y) = \log(1 + y) - x = \operatorname{log1p}(y) - x$. The first derivative of that function is $f'(y) = 1/(1 + y)$, so the iteration looks like this:

$$y_{n+1} = y_n - f(y_n)/f'(y_n) = y_n - (\operatorname{log1p}(y_n) - x)(1 + y_n)$$

A reasonable starting point for the iteration is $y_0 = \operatorname{expm1}(\operatorname{peval}(x))$, and we need the pair-precision function `plog1p()` from **Section 23.5** on the preceding page. The code for the pair-precision computation of `pexpm1()` with that algorithm is simple enough that we do not show it here. Plots of the error show that it remains well below 1 ulp over most of the argument range, but in the interval $[-\frac{1}{2}, +\frac{1}{2}]$, it rises to about 5 ulps, much higher than we would like.

A better solution proves to be modification of the code for `PEXP()`: reduce the argument via $x = m \ln(\beta) + r$, use the Taylor series to compute $\exp(r) - 1$, and then recover the final result via exact scaling by β^m. The code in `PEXPM1()` is similar to that for `PEXP()`, so we omit it here.

Figure 23.6 on the next page shows the measured errors in the `PEXPM1()` functions, demonstrating that the results are usually correctly rounded, except in the interval $[-2, +2]$, where errors can reach 1.3 ulps. Outside the interval shown in the plots, the errors remain mostly below 0.5 ulps, and never get above 0.8 ulps.

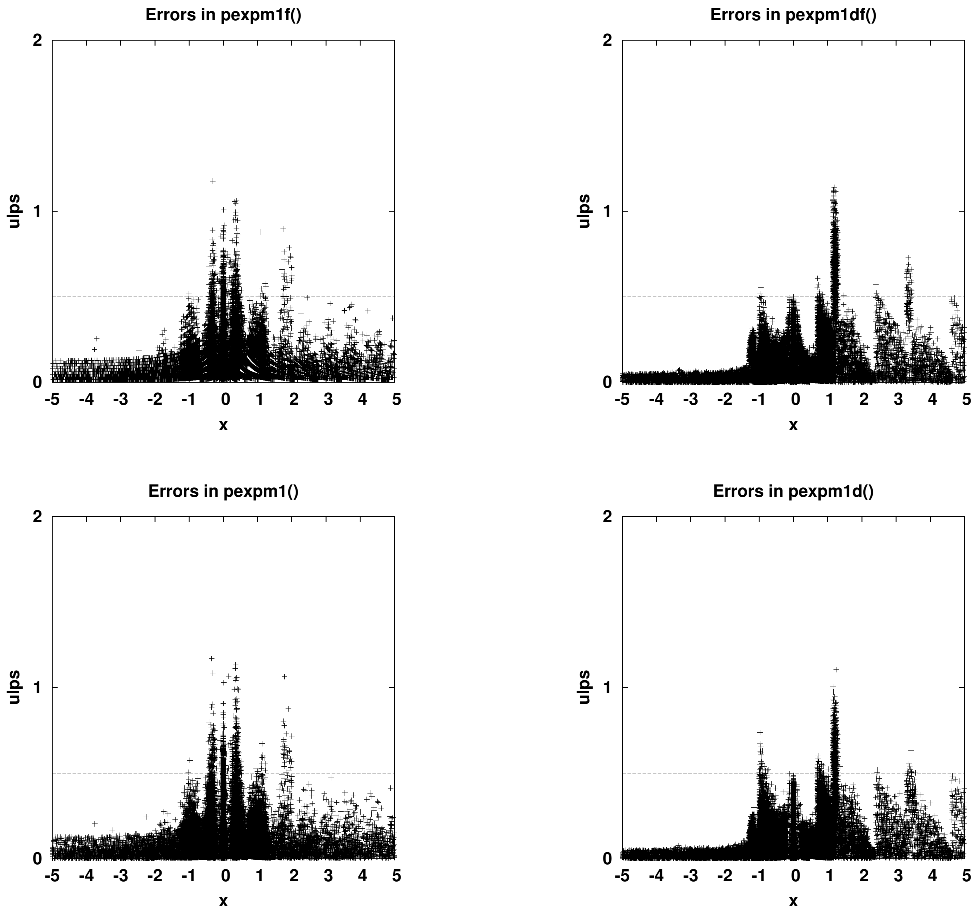


Figure 23.6: Errors in pair-precision `PEXPM1()` functions.

23.7 Pair-precision base-*n* exponentials

The function families `pexp2()`, `pexp8()`, `pexp10()`, and `pexp16()` provide pair-precision base-2, 8, 10, and 16 exponentials, respectively. Let $k^x = \exp(y)$ for some fixed k, then take logarithms of both sides to find $y = x \log(k)$. We therefore have these definitions:

$$\text{pexp2}(x) = \text{pexp}(x \log(2)), \qquad \text{pexp10}(x) = \text{pexp}(x \log(10)),$$
$$\text{pexp8}(x) = \text{pexp}(x \log(8)), \qquad \text{pexp16}(x) = \text{pexp}(x \log(16)).$$

Unfortunately, straightforward application of those relations is unsatisfactory because of the problem of error magnification (see **Section 4.1** on page 61). The inexact scaling of x by the logarithm term introduces an error in the exponential functions that grows with $|x|$, so errors of hundreds or thousands of ulps are likely for the higher-precision functions near their underflow and overflow limits.

The solution that we adopt is to split x into a sum of integer and fractional parts: $x = n + f$. That is an exact operation provided by the `modf()` family that we can apply to the high part. We then have

$$\text{pexp2}(x) = 2^n \,\text{pexp}(f \log(2)), \qquad \text{pexp10}(x) = 10^n \,\text{pexp}(f \log(10)),$$
$$\text{pexp8}(x) = 8^n \,\text{pexp}(f \log(8)), \qquad \text{pexp16}(x) = 16^n \,\text{pexp}(f \log(16)),$$

where the error magnification is largely eliminated because of the reduced argument size, provided that we can compute integer powers of 2, 8, 10, and 16 accurately. The `ldexp()` family provides exact computation of those

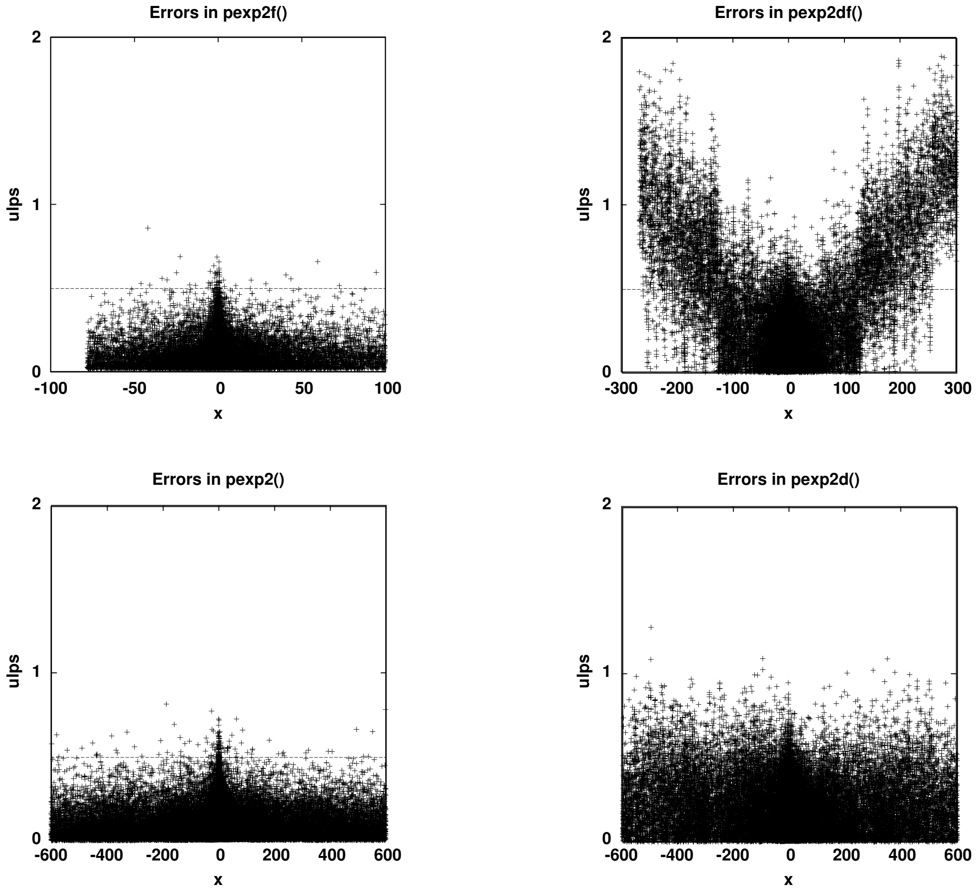


Figure 23.7: Errors in pair-precision `PEXP2()` functions. The error growth in `pexp2df()` is due to the use of the inexact `pipowdf()` function for computing 2^n in decimal arithmetic.

powers when the base is k. Otherwise, we can sometimes decompose them into an exact scaling and a small accurate power. For example, when $\beta = 16$, we have $8^n = 8^{4m+r} = 16^{3m} \times 8^r$, where r lies in $[-3, +3]$. If that reduction is not possible, we fall back to computing the power with the `pipow()` family.

The logarithms of the constants in pair precision must be accurate to the last digit. We use the same technique as in `plog()`, representing the values as series of exact rational numbers that are summed to working precision by `pcon()` in an initialization code block that is executed only on the first entry.

Figure 23.7 through **Figure 23.10** on page 799 show the measured errors in our implementations of the exponential functions.

23.8 Pair-precision trigonometric functions

The cos, sin, and tan functions satisfy these periodicity and symmetry relations, where n is an integer:

$$\begin{aligned} \cos(x) &= \cos(x + 2n\pi) & \tan(x) &= \tan(x + n\pi), \\ \cos(x) &= -\cos(x + (2n+1)\pi), & \cos(x) &= \cos(-x), \\ \sin(x) &= \sin(x + 2n\pi), & \sin(x) &= -\sin(-x), \\ \sin(x) &= -\sin(x + (2n+1)\pi), & \tan(x) &= -\tan(-x). \end{aligned}$$

The function ranges are $[-1, +1]$ for cos and sin, and $(-\infty, +\infty)$ for tan.

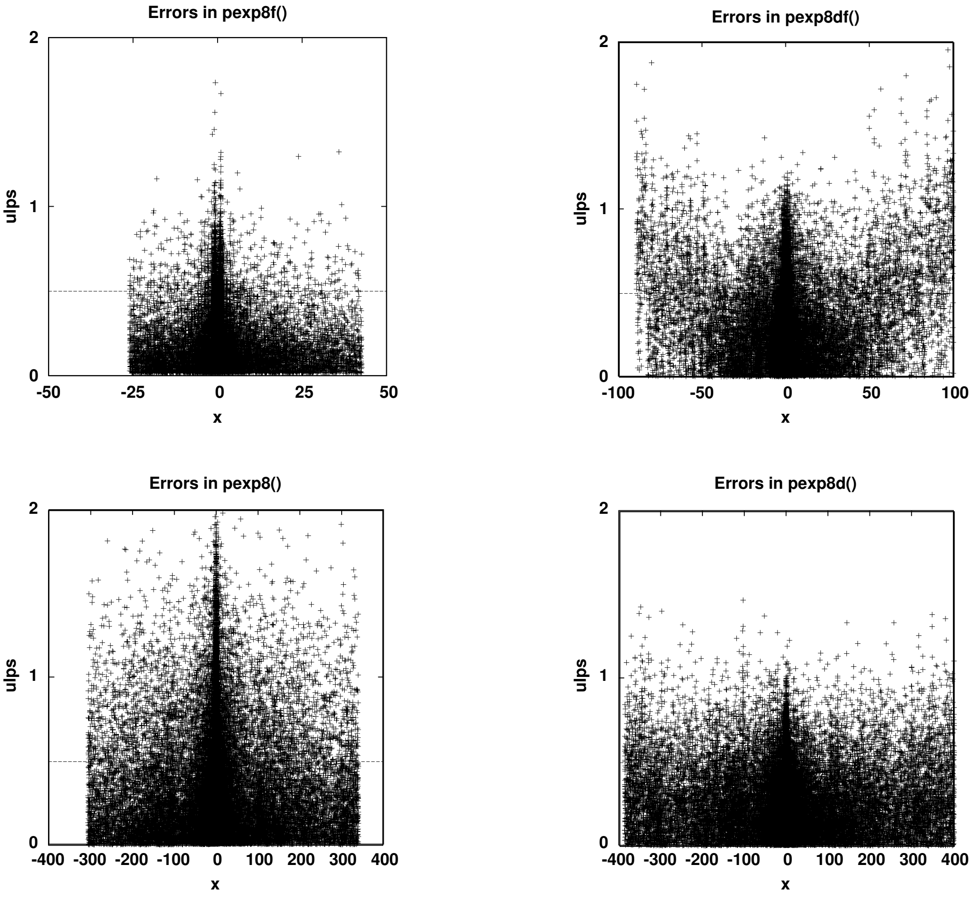


Figure 23.8: Errors in pair-precision PEXP8() functions.

We guarantee symmetry by computing the functions only for positive arguments, and then inverting the sign of the result for sin and tan of negative arguments.

The hardest problem in their computation is argument reduction via $x = n\pi + r$. If x is large, then n is too, and the small residual r in $[-\frac{1}{2}\pi, +\frac{1}{2}\pi]$ is the result of the subtraction of two large numbers. In IEEE 754 64-bit decimal arithmetic, n can require almost 400 digits, and in 128-bit decimal arithmetic, more than 6000 digits. We require π to similar accuracy. The exact argument-reduction algorithm that we describe in **Chapter 9** on page 243 is complicated, and would be difficult to extend for pair-precision arithmetic.

Applications of trigonometric functions can usually be manipulated mathematically to avoid the need for arguments outside small ranges like $[0, \pi]$, $[0, 2\pi]$, or $[-\frac{1}{2}\pi, +\frac{1}{2}\pi]$. Most implementations of trigonometric functions, including the recipes of Cody and Waite, therefore do not attempt accurate argument reduction beyond a few multiples of π.

Polynomial approximations are impractical for pair-precision arithmetic, so we need to resort to series summation. The Taylor series for $\cos(r)$, $\sin(r)$, and $\tan(r)$ converge reasonably rapidly because of the factorials in the denominators: when $r = \frac{1}{2}\pi$, a sum of 50 terms recovers about 140 decimal digits.

Although the series terms for $\cos(r)$ and $\sin(r)$ alternate in sign, the ratios of the magnitudes of successive terms remain below $\frac{1}{2}$ for $|r| \leq \frac{1}{2}\pi$, so there is never subtraction loss in the argument region where we use the series.

Only the series for $\sin(r)$ is suitable for computation: it begins $r - r^3/3! + r^5/5! + \cdots$. Successive terms t_k can be

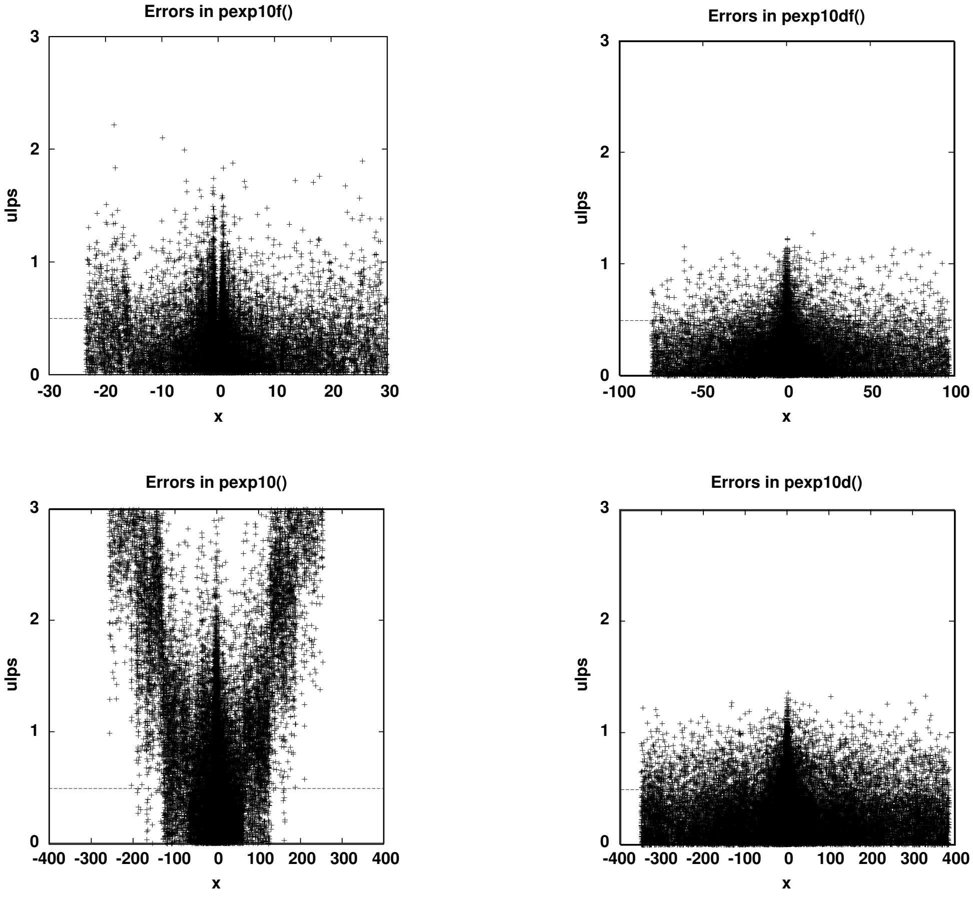


Figure 23.9: Errors in pair-precision `PEXP10()` functions. The error growth in `pexp10f()` is due to the use of the inexact `pipowf()` function for computing 10^n in binary arithmetic, and the errors rise to about 8 ulps. Notice that the range of the vertical axis is larger than in other figures in this chapter.

computed quickly from a recurrence relation:

$$t_0 = 0, \qquad t_{2k} = 0, \qquad t_1 = r, \qquad t_{2k+1} = \frac{r^2}{(2k+1)(2k)} t_{2k-1}, \qquad \text{for } k = 1, 2, 3, \ldots .$$

We omit the first two terms while accumulating the sum to machine precision after factoring out r^5, then multiply that sum by r^5 and add it to the leading terms, producing a result with small relative error.

The series for $\cos(r)$ starts with $1 - r^2/2! + r^4/4! + \cdots$. Although that is reasonable when $|r| \ll 1$, the range of the function, $[-1, +1]$, makes it clear that massive subtraction loss must occur to reach a result of 0. Instead, we use the relation $\cos(r) = \sin(r + \frac{1}{2}\pi)$, taking care to apply it only for $r \geq 0$ to avoid subtraction loss.

Cody and Waite recommend using a higher-precision value of $\frac{1}{2}\pi \approx c_1 + c_2$, where c_1 is chosen so that nc_1 is exact for n values of interest, and c_2 is a correction accurate to full precision. In the mathcw library, we set a target of accurate reduction for $|n| < 100$. The computation then proceeds as follows:

$$\begin{aligned}
n &= \operatorname{round}(|x|/\pi + \tfrac{1}{2}), \\
x_n &= n - \tfrac{1}{2}, \\
x_1 &= \operatorname{floor}(|x|), \\
x_2 &= |x| - x_1,
\end{aligned}$$

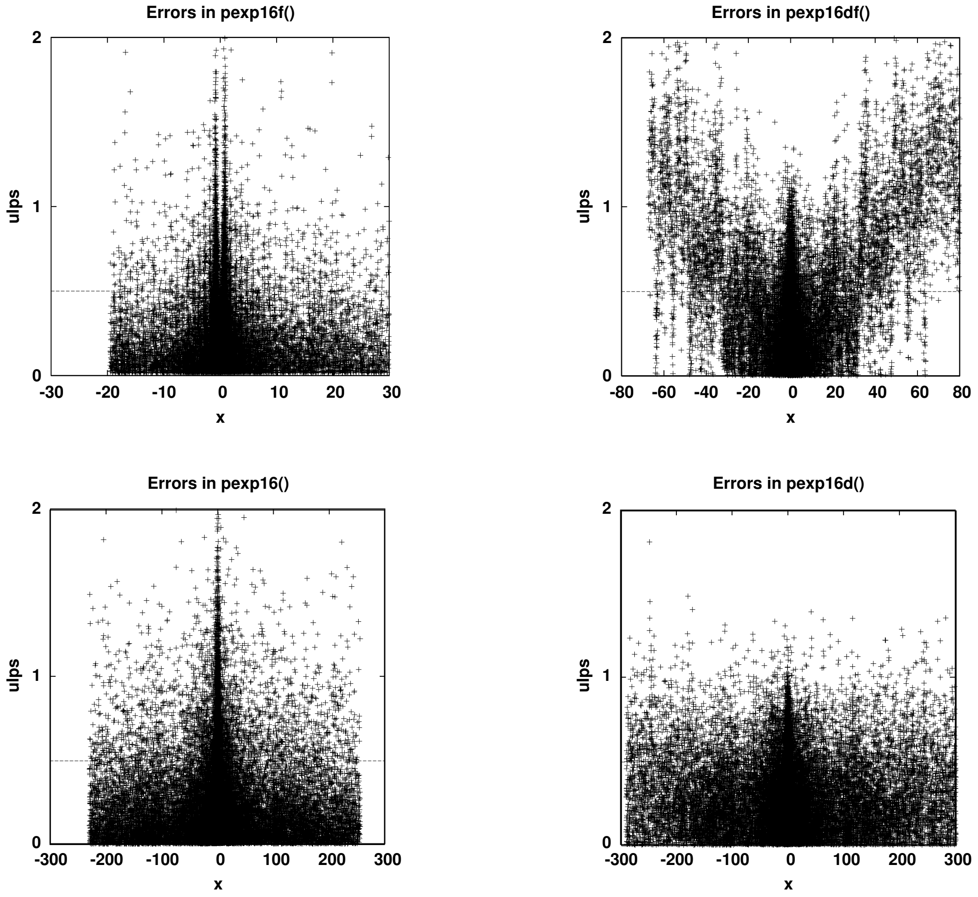


Figure 23.10: Errors in pair-precision `PEXP16()` functions.

$$r = \big((x_1 - x_n c_1) + x_2\big) - x_n c_2.$$

Notice that the explicit computation of $x + \frac{1}{2}\pi$ is avoided by folding the term $\frac{1}{2}\pi$ into the computation of x_n, because $\frac{1}{2}\pi$ is not exactly representable, but $n - \frac{1}{2}$ is exact if n is not too large.

The pair-precision computation of `PCOS()` from `PSIN()` is then a straightforward transcription of those steps:

```
PABS(xabs, x);
n = (int)ROUND(PEVAL(xabs) * ONE_OVER_PI + HALF);
PSET(xn, (fp_t)n - HALF, ZERO);
PSET(x1, FLOOR(PEVAL(xabs)), ZERO);
PSUB(x2, xabs, x1);
PMUL(t, xn, C1);
PSUB(r, x1, t);
PADD(r, r, x2);
PMUL(t, xn, C2);
PSUB(r, r, t);

if (ISODD(n))
    PNEG(r, r);

PSIN(result, r);
```

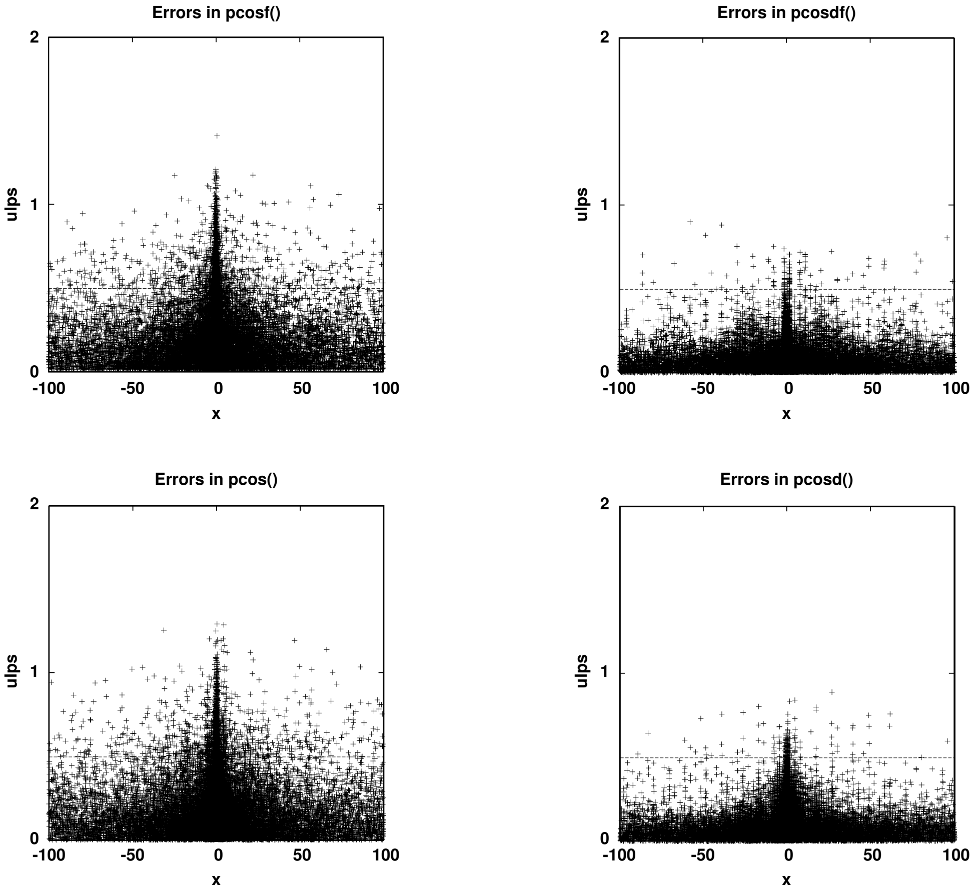


Figure 23.11: Errors in pair-precision `PCOS()` functions.

The Taylor series for $\tan(r)$ does not have simple coefficients that can easily be generated from earlier ones. Instead, they depend on the Bernoulli numbers that we met in earlier chapters (see **Section 18.5** on page 568):

$$\begin{aligned}\tan(x) &= x + (1/3)x^3 + (2/15)x^5 + (17/315)x^7 + \cdots + \\ &\quad (B_{2n}(-4)^n(1-4^n)/(2n)!)x^{2n-1} + \cdots .\end{aligned}$$

For pair-precision arithmetic, we therefore use that series only when the argument is small enough that the first two terms suffice. Otherwise, we fall back to the definition $\tan(x) = \sin(x)/\cos(x)$, making the pair-precision tangent about twice as costly as the corresponding cosine or sine.

Figure 23.11 through **Figure 23.13** on page 802 show the measured errors in our pair-precision implementations of the trigonometric functions. The errors are almost entirely due to inaccuracies in the initial argument reduction, and are largest for arguments that lie near integer multiples of $\frac{1}{2}\pi$, reaching several tens of ulps in testing with random arguments, and much larger with specially selected ones.

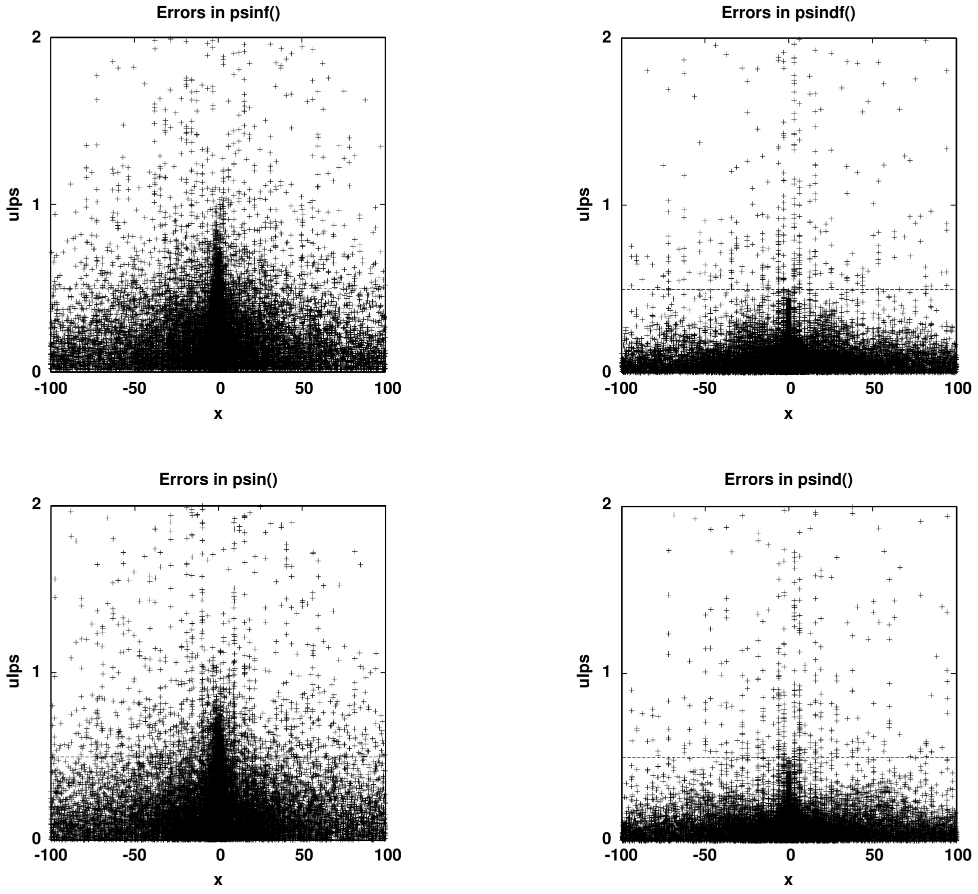


Figure 23.12: Errors in pair-precision `PSIN()` functions.

23.9 Pair-precision inverse trigonometric functions

The Taylor series for the inverse trigonometric functions are:

$$\begin{aligned}
\operatorname{acos}(x) &= \tfrac{1}{2}\pi - \bigl(x + (1/6)x^3 + (3/40)x^5 + (5/112)x^7 + \cdots + \\
&\qquad ((2n)!/(4^n(n!)^2(2n+1)))x^{2n+1} + \cdots\bigr), \\
\operatorname{asin}(x) &= x + (1/6)x^3 + (3/40)x^5 + (5/112)x^7 + \cdots + \\
&\qquad ((2n)!/(4^n(n!)^2(2n+1)))x^{2n+1} + \cdots, \\
\operatorname{atan}(x) &= x - (1/3)x^3 + (1/5)x^5 - (1/7)x^7 + (1/9)x^9 - \cdots.
\end{aligned}$$

The argument range of $\operatorname{acos}(x)$ is $[-1,+1]$, and the corresponding function range is $[\pi, 0]$. The argument range of $\operatorname{asin}(x)$ is $[-1,+1]$, and the function range is $[-\frac{1}{2}\pi, +\frac{1}{2}\pi]$. The argument range of $\operatorname{atan}(x)$ is $(-\infty, +\infty)$, and the function range is $[-\frac{1}{2}\pi, +\frac{1}{2}\pi]$.

The series for $\operatorname{atan}(x)$ was found by Gregory[1]about 1667 from complicated geometric arguments, about four decades before Taylor in 1715 used the calculus of Newton and Leibniz to show how to derive such series systematically. In 1755, Euler found another series that converges more rapidly for $|x| \approx 1$, about the same as the normal

[1] The Scottish scientist James Gregory (1638–1675) also described the first practical reflecting telescope, and invented the diffraction grating for splitting sunlight into its colors, a year after Isaac Newton did the same with a prism.

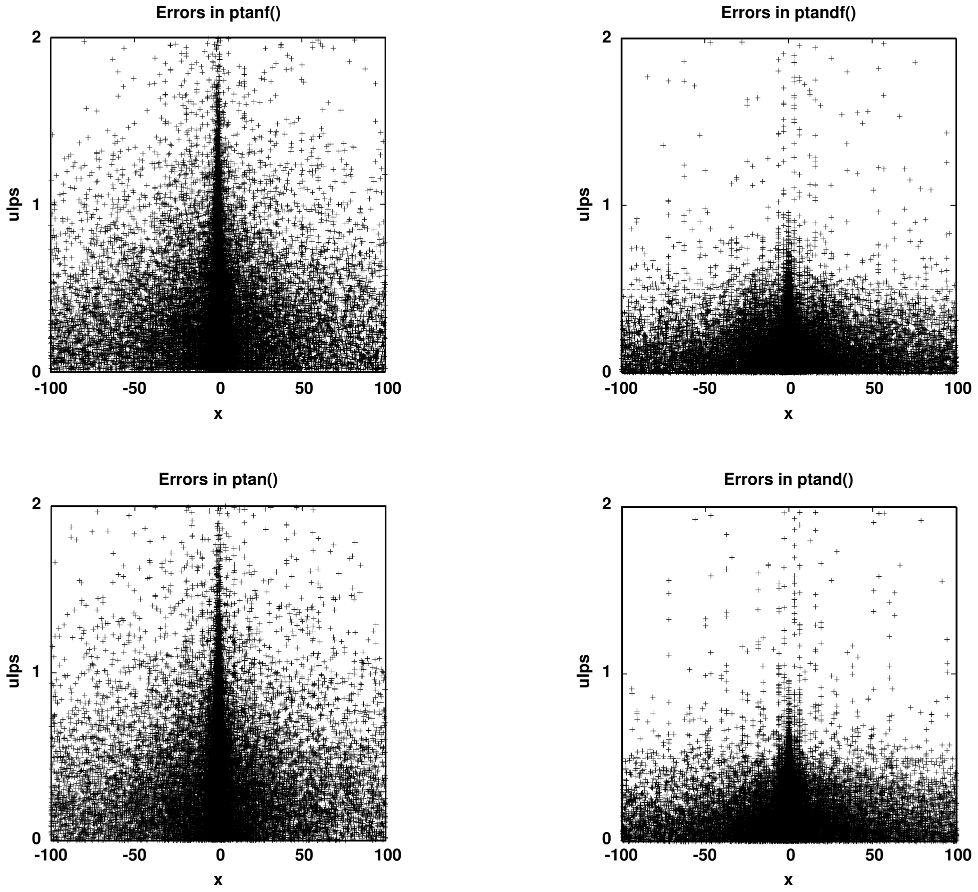


Figure 23.13: Errors in pair-precision `PTAN()` functions.

Taylor series for $|x| \approx 0$, and has only positive terms:

$$\begin{aligned} z &= x^2/(1+x^2), \\ \operatorname{atan}(x) &= (z/x)(1 + (2!!/3!!)z + (4!!/5!!)z^2 + (6!!/7!!)z^3 + \cdots) \\ &= (z/x)\sum_{n=0}^{\infty}\prod_{k=1}^{n} 2kz/(2k+1), \end{aligned}$$

Lambert[2] (1770) and Lagrange[3] (1776) independently discovered a simple, and visually pleasing, continued-fraction formula for the arc tangent:

$$\operatorname{atan}(x) = \frac{x}{1+}\;\frac{x^2}{3+}\;\frac{4x^2}{5+}\;\frac{9x^2}{7+}\;\frac{16x^2}{9+}\;\frac{25x^2}{11+}\;\frac{36x^2}{13+}\;\frac{49x^2}{15+}\;\frac{64x^2}{17+}\cdots.$$

The general term in the denominators of the continued fraction is

$$d_k = (2k-1) + k^2x^2/d_{k+1},$$

[2] The Swiss–German scientist Johann Heinrich Lambert (1728–1777) was the first to introduce hyperbolic functions into trigonometry and the first to prove that π is irrational (1761). He also developed the first practical *hygrometer* and *photometer*, devices for measuring water vapor and light. The widely used *Lambert cylindrical equal-area* map projection is named after him.

[3] The French mathematician and astronomer Joseph-Louis Lagrange (1736–1813) made many important contributions, including fundamental work on analytical mechanics and wave propagation. He developed the *calculus of variations*, the theory of *differential equations*, the *mean value theorem*, and worked on the *three-body problem* for the calculation of the orbits of the Earth, Moon, and Sun, and Jupiter and its satellites. His tomb is in the Panthéon in Paris, France.

and the fraction can be evaluated in backward order, starting from a fixed term k, by setting $1/d_{k+1} = 0$. All terms are positive, so there is no possibility of subtraction loss. Numerical experiments, done once and for all, determine the number of terms required to reach the desired precision for a given maximum value of $|x|$. They show that for $|x| < \frac{1}{2}$, the continued fraction requires about half as many terms as the Taylor series. The hoc code to evaluate the continued fraction is short:

```
xx = x * x
d_k = Infinity

for (k = KMAX; k > 0; --k) \
    d_k = (k + k - 1) + (k * k * xx / d_k)

result = x / d_k
```

Forward evaluation with early loop exit is also possible (see **Section 2.7** on page 12), but we do not show code for it here.

The Taylor-series recurrence formula for terms t_k in the series for asin(x) is

$$t_0 = 0, \qquad t_{2k} = 0, \qquad t_1 = x, \qquad t_{2k+1} = \frac{(2k-1)^2}{2k(2k+1)} x^2 t_{2k-1}, \qquad \text{for } k = 1, 2, 3, \ldots .$$

The symmetry relations are

$$\begin{aligned}\operatorname{acos}(-x) &= \tfrac{1}{2}\pi + \operatorname{asin}(x),\\ \operatorname{asin}(-x) &= -\operatorname{asin}(x),\\ \operatorname{atan}(-x) &= -\operatorname{atan}(x).\end{aligned}$$

These relations are essential for argument reduction:

$$\operatorname{acos}(x) = \begin{cases} 2\operatorname{asin}\left(\sqrt{\tfrac{1}{2}(1-x)}\right), & \text{use for } x > \tfrac{1}{2},\\ \pi - 2\operatorname{asin}\left(\sqrt{\tfrac{1}{2}(1+x)}\right), & \text{use for } x < -\tfrac{1}{2},\\ \tfrac{1}{2}\pi - \operatorname{asin}(x), & \text{use for } |x| \le \tfrac{1}{2},\end{cases}$$

$$\operatorname{asin}(x) = \tfrac{1}{2}\pi - 2\operatorname{asin}\left(\sqrt{\tfrac{1}{2}(1-x)}\right), \qquad \text{use for } |x| > \tfrac{1}{2}, \text{ but see text,}$$

$$\operatorname{atan}(x) = \begin{cases} \tfrac{1}{2}\pi - \operatorname{atan}(1/x), & \text{use for } |x| > 1,\\ \tfrac{1}{6}\pi - \operatorname{atan}(f), & \text{use for } |x| > 2 - \sqrt{3}.\end{cases}$$

The intermediate variable in the last equation is defined by

$$\begin{aligned} f &= \left((\sqrt{3})x - 1\right)/(x + \sqrt{3}), && \text{use for } \beta \neq 16,\\ &= \left((((\sqrt{3} - 1)x - \tfrac{1}{2}) - \tfrac{1}{2}) + x\right)/(x + \sqrt{3}), && \text{use for } \beta = 16.\end{aligned}$$

The factors $(1 \pm x)$ lose leading digits in the regions where they are used, but they are nevertheless computed exactly, because x is exact, and because neither shifting nor rounding is required to perform the operations.

For a NaN argument, or an argument that lies outside the valid range, the code in all three functions returns the NaN argument, or else a quiet NaN, and sets `errno` to `EDOM`.

For asin(x), the code handles a zero argument separately so as to preserve the sign of the argument in the result, and processes tiny arguments quickly by a two-term Taylor series that provides the required rounding, and sets the *inexact* flag. We use the symmetry relation to handle negative arguments, argument reduction to handle arguments in $(\frac{1}{2}, 1]$, and otherwise, we sum the Taylor series for arguments in $(0, \frac{1}{2}]$.

Graphs of the errors in an initial implementation of `PASIN()` exposed a problem with that algorithm, which is based on traditional recipes. The graphs had spikes in the errors reaching up to about 2.2 ulps for $x \approx \pm\frac{1}{2}$, whereas elsewhere, the errors remained mostly below 0.75 ulps. The problem is subtraction loss in forming $\frac{1}{2}\pi - 2\operatorname{asin}\left(\sqrt{\frac{1}{2}(1-x)}\right)$. Leading bits are lost for x values where the second term exceeds $\frac{1}{4}\pi$. That relation is readily solved to find that bit loss occurs for x in $[\frac{1}{2}, \sqrt{\frac{1}{2}}] \approx [0.5, 0.7]$. Increasing the cutoff from $\frac{1}{2}$ to $\sqrt{\frac{1}{2}}$ roughly doubles the number of terms needed in the Taylor series in the worst case, but notably improves the error plots by completely eliminating the two spikes. We therefore adopt that improvement, sacrificing speed for accuracy.

The magnification factors in **Table 4.1** on page 62 for $\operatorname{acos}(x)$ and $\operatorname{asin}(x)$ have denominators $\sqrt{1-x^2}$, and thus, those functions are increasingly sensitive to argument errors as $|x|$ approaches 1. As with other functions having large error-magnification factors, the only solution when such arguments are commonly needed is to invoke higher-precision versions of the functions.

For $\operatorname{acos}(x)$, the computation diverts to $\operatorname{asin}\left(\sqrt{\frac{1}{2}(1 \pm x)}\right)$ when $|x| > \frac{1}{2}$. Otherwise, the result is $\frac{1}{2}\pi - \operatorname{asin}(x)$. The argument ranges in each case prevent inexact loss of leading digits in the subtractions. The only extra consideration is that the constants π and $\frac{1}{2}\pi$ have leading zero bits in hexadecimal arithmetic, so when $\beta = 16$, the computation is done in steps with $\frac{1}{4}\pi$, which has no leading zero bits.

The $\operatorname{atan}(x)$ computation is the most difficult, and the code in `PATAN()` closely follows that for `ATAN()`. Negative arguments are handled as positive ones, with a sign change in the final result. As with $\operatorname{asin}(x)$, zero and tiny arguments are handled by separate code for preservation of the sign of zero, rounding, setting of the *inexact* flag, and speed.

The reciprocal argument relation for $\operatorname{atan}(x)$ allows large arguments to be reduced to smaller ones in $[0, 1]$. The intermediate variable f allows further reduction to the approximate range $[0, 0.268]$ where the Taylor series is finally summed. Unfortunately, the series converges slowly: 28 terms are needed for 34 decimal digits, and 120 terms produce 140 digits. The multistep computation of f avoids precision loss from leading zero bits in hexadecimal arithmetic. Cody and Waite's careful arrangement of the algorithm steps avoids direct computation of $\frac{1}{2}\pi - \frac{1}{6}\pi$ by reducing it analytically to $\frac{1}{3}\pi$.

In all three functions, header files contain values of the constants π, $\frac{1}{2}\pi$, $\frac{1}{3}\pi$, $\sqrt{3} - 1$, and so on, each expressed as an exact pair-precision high part that must be normalized by `PSUM2()` before use, and a pair-precision low part defined as a table that must be summed by a call to `PCON()`. Those steps are handled in an initialization block that is executed only on the first call to each function. When one of those constants is needed in a sum, the low part is added before the exact high part, sometimes helping to increase the effective precision of subtractions.

Figure 23.14 on the next page through **Figure 23.16** on page 806 show the measured errors in our implementations of the inverse trigonometric functions in pair-precision arithmetic. Compared to the routines for binary arithmetic, the decimal routines exhibit smaller errors because of their more accurate low-level pair-precision primitives.

23.10 Pair-precision hyperbolic functions

We treated the hyperbolic companions of the standard trigonometric functions in detail in **Section 12.1** on page 341, observing there that the two functions $\cosh(x)$ and $\sinh(x)$ need careful argument shifting to avoid premature overflow. The shift code for the pair-precision version is a straightforward transcription of the code for ordinary precision, except that the shift constants $\frac{1}{2}v - 1$ and $1/v^2$ must be reconstructed by the `PCON()` family from a tabular representation.

For the hyperbolic cosine, we use a pair-precision transcription of the algorithm described earlier in **Section 12.1** on page 341, because there is no possibility of subtraction loss, and the major computational problem is to avoid premature overflow.

Figure 23.17 on page 807 shows the measured errors in the pair-precision hyperbolic cosine functions.

The functions $\sinh(x)$ and $\tanh(x)$ cannot be computed from their definitions in the region where $\exp(x) - \exp(-x)$ suffers bit loss from the subtraction. However, their Taylor series expansions both begin with x, showing that the loss is primarily from the leading term of the series for $\exp(x) = 1 + x + x^2/2! + \cdots$. That suggests that it may be useful to modify their definitions to work with $\operatorname{expm1}(x) = \exp(x) - 1$, a function for which we have accurate implementations for both ordinary and pair-precision arithmetic. We find

$$\begin{aligned}
\sinh(x) &= \tfrac{1}{2}(\exp(x) - \exp(-x)) \\
&= \tfrac{1}{2}\big((\exp(x) - 1) - (\exp(-x) - 1)\big) \\
&= \tfrac{1}{2}(\operatorname{expm1}(x) - \operatorname{expm1}(-x)).
\end{aligned}$$

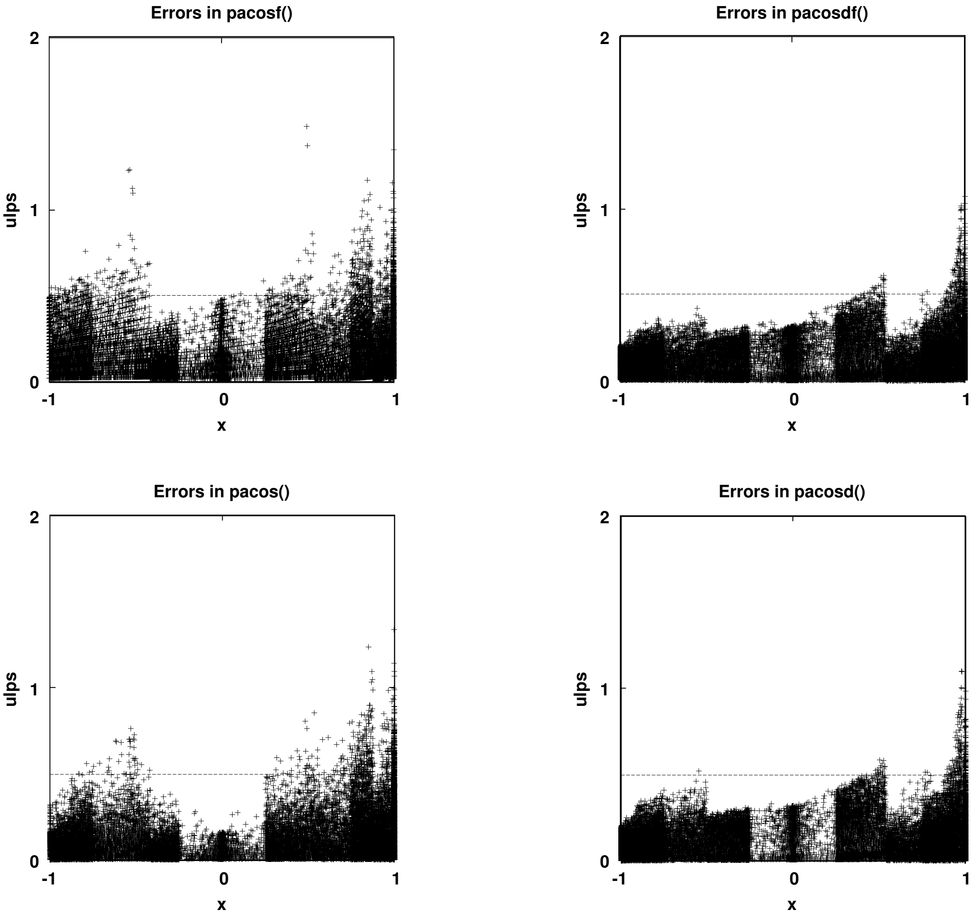


Figure 23.14: Errors in pair-precision `PACOS()` functions.

For x in $[0,\infty)$, the range of $\operatorname{expm1}(-x)$ is $[0,-1]$, so the subtraction is really addition of positive terms, and there can be no bit loss. We can eliminate one of the functions on the right by introducing a temporary variable as follows:

$$\begin{aligned}E &= \exp(x) - 1\\ &= \operatorname{expm1}(x),\\ \sinh(x) &= \tfrac{1}{2}\big((E+1) - 1/(E+1)\big)\\ &= \tfrac{1}{2}\big(((E+1)^2 - 1)/(E+1)\big)\\ &= \tfrac{1}{2}(E + E/(E+1)).\end{aligned}$$

Because E is nonnegative for $x \geq 0$, that allows us to compute $\sinh(x)$ directly from $\operatorname{expm1}(x)$ for x in the region $[0,1]$ where we used a rational polynomial approximation in ordinary arithmetic. The drawback is that we cannot easily prevent bit loss from wobbling precision in hexadecimal arithmetic.

Figure 23.18 on page 807 shows the measured errors in the pair-precision hyperbolic sine functions.

The hyperbolic tangent quickly reaches its limits of ±1, which we can handle as in the code in `tanhx.h`. For smaller arguments where that code uses a polynomial approximation, we instead build on the pair-precision algorithm for $\sinh(x)$, and write

$$\begin{aligned}\tanh(x) &= \sinh(x)/\cosh(x)\\ &= (E + E/(E+1))/\big((E+1) + 1/(E+1)\big)\end{aligned}$$

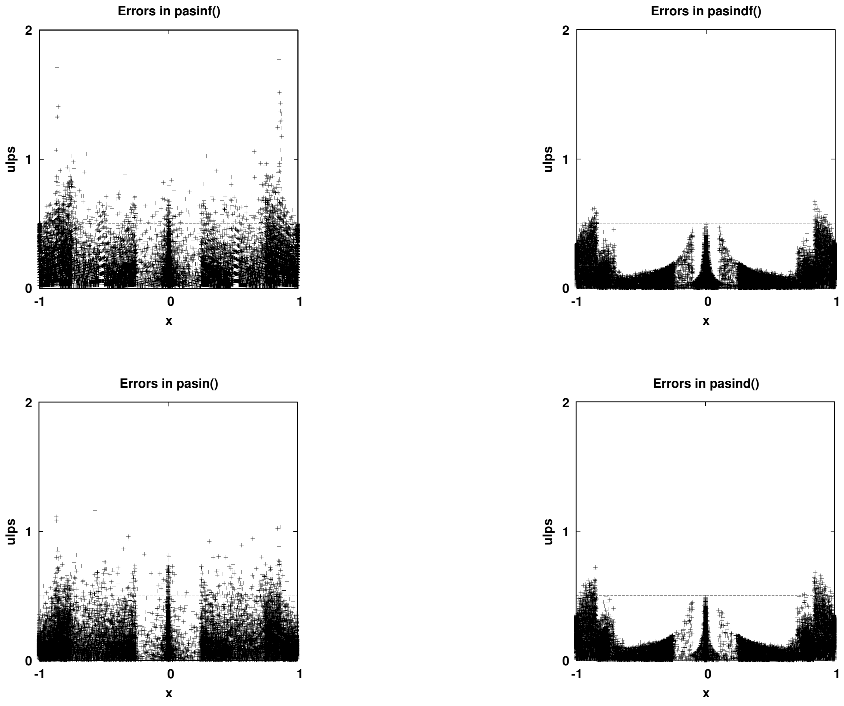


Figure 23.15: Errors in pair-precision `PASIN()` functions.

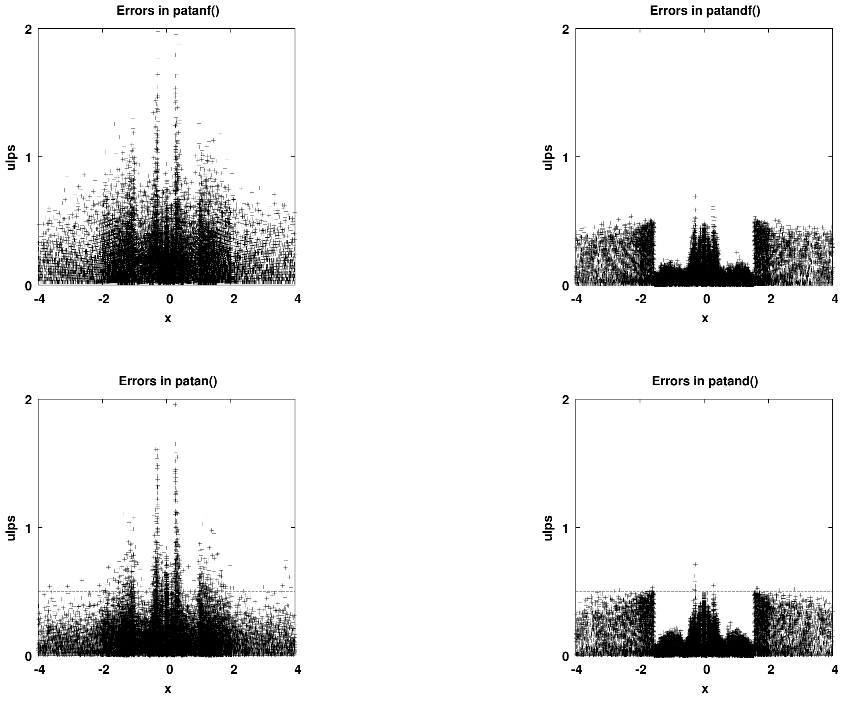


Figure 23.16: Errors in pair-precision `PATAN()` functions. Outside the interval shown in the plots, the errors remain below $\frac{1}{2}$ ulps, indicating correct rounding.

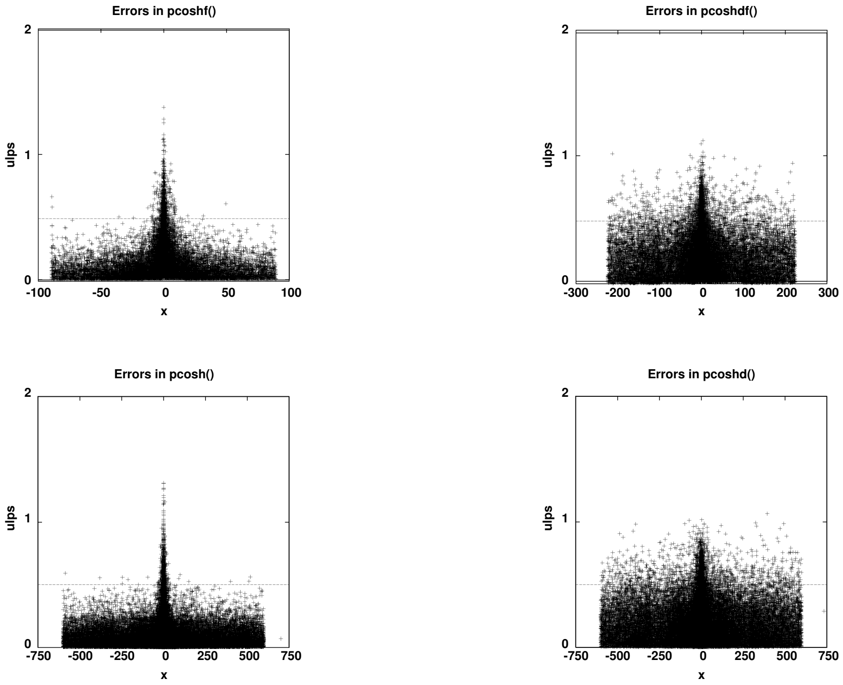


Figure 23.17: Errors in pair-precision PCOSH() functions.

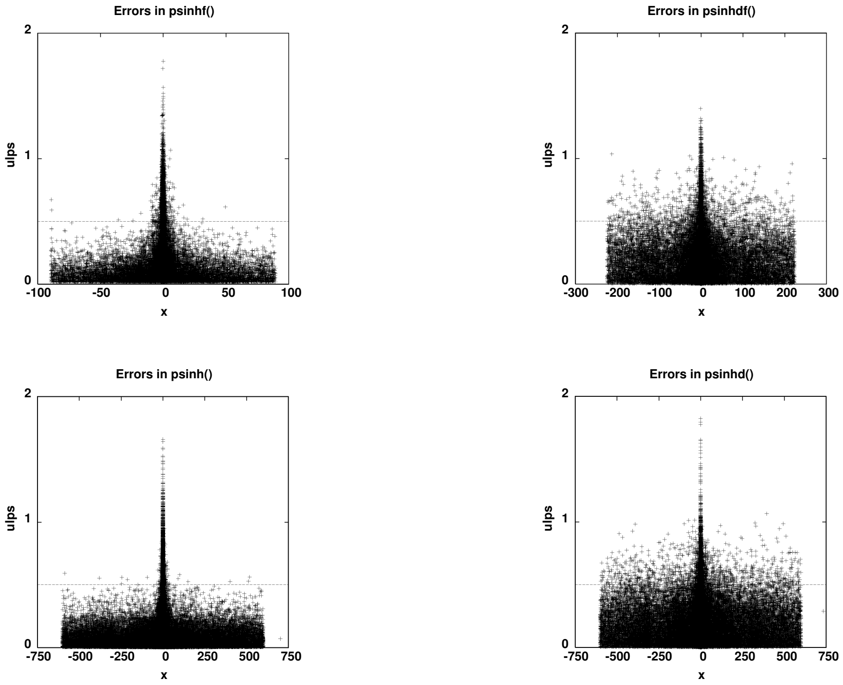


Figure 23.18: Errors in pair-precision PSINH() functions.

To counteract the effect of wobbling precision in hexadecimal arithmetic for expressions of the form $1 + \cdots$, introduce two intermediate variables, F and G, defined by

$$F = \tfrac{1}{2}(E+1) = \tfrac{1}{2}E + \tfrac{1}{2}, \qquad\qquad G = \tfrac{1}{2}/(E+1) = \tfrac{1}{4}/(\tfrac{1}{2}E + \tfrac{1}{2}) = \tfrac{1}{4}/F.$$

We then have

$$\tanh(x) = E(\tfrac{1}{2} + G)/(F+G),$$

where all of the terms are positive when $x \geq 0$.

There is still room for improvement, however. Add and subtract the denominator from the numerator, and simplify to find

$$\begin{aligned}\tanh(x) &= E(1 + (\tfrac{1}{2} - F)/(F+G)) \\ &= 2E(\tfrac{1}{2} - E/(4(F+G))).\end{aligned}$$

Although we have now introduced a subtraction, there is no bit loss, because the range of the term subtracted is approximately $[0, 0.16]$ for x in the interval $[0, |\frac{1}{2}\ln(\frac{1}{3})|]$ where that formula applies. We now have a small correction to an exact value, $\frac{1}{2}$, instead of a ratio of two inexact values, so the cumulative error is primarily from E alone. Numerical experiments show that our simple change reduces the worst-case error by as much as 0.7 ulps.

Figure 23.19 on the next page shows the measured errors in the pair-precision hyperbolic tangent functions.

23.11 Pair-precision inverse hyperbolic functions

The inverse hyperbolic functions discussed in **Section 12.4** on page 348 have simple definitions in terms of log(), log1p(), and sqrt() that are suitable for direct computation. The pair-precision implementations of those functions are straightforward transcriptions of the code for `acosh()`, `asinh()`, and `atanh()`, so we do not display the code here. **Figure 23.20** on the next page through **Figure 23.22** on page 810 show the measured errors in our pair-precision versions of the inverse hyperbolic functions. The argument range for `acosh()` is $[1, \infty)$ and for `asinh()` is $(-\infty, +\infty)$ but the graphs show only a small interval near the origin. Outside that region, the errors remain below 0.7 ulps and 1.2 ulps, respectively.

23.12 Summary

In this chapter, we have given a flavor of how some of the elementary functions can be computed in pair-precision arithmetic. The essential difference from the algorithms used for ordinary arithmetic is that we do not have the luxury of compact rational polynomial approximations, primarily because of the difficulty of representing their coefficients portably, accurately, and without run-time overhead of decoding long decimal or hexadecimal string representations. Instead, we are usually forced to sum truncated Taylor series, which can require several hundred terms for the highest precision supported by the `mathcw` library, about 140 decimal digits,

Newton–Raphson iteration can sometimes replace polynomial approximation, as we found for the `pexpm1()` and `plog()` function families. However, the iteration requires an accurate inverse of the function that we seek to compute, so at least one of them must be found by a different algorithm.

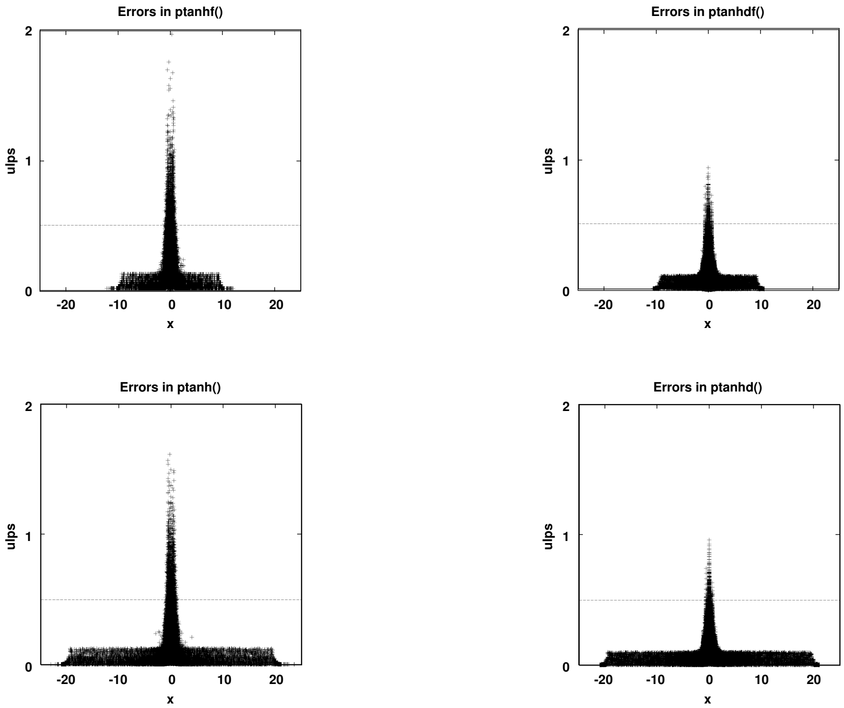


Figure 23.19: Errors in pair-precision `PTANH()` functions.

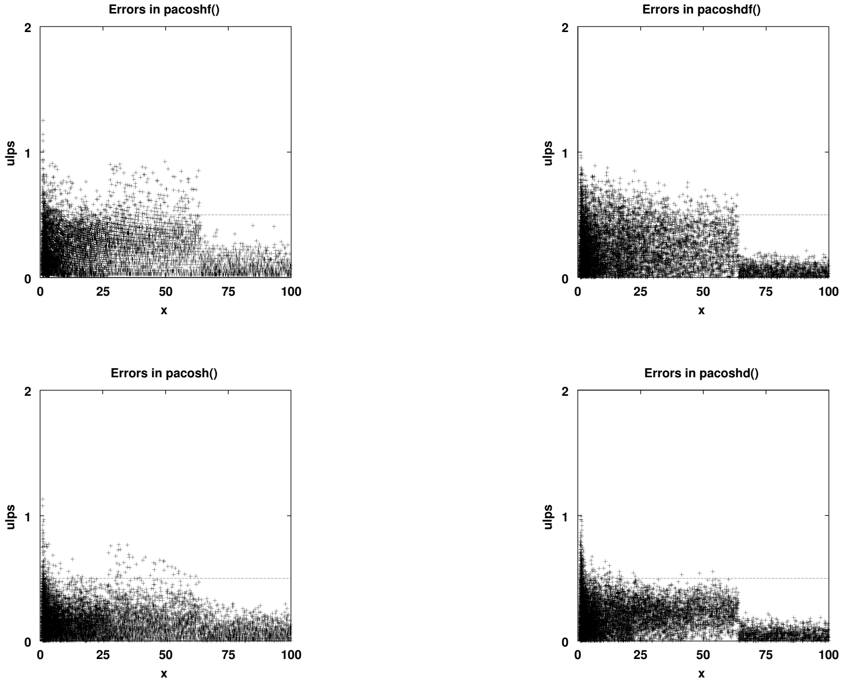


Figure 23.20: Errors in pair-precision `PACOSH()` functions.

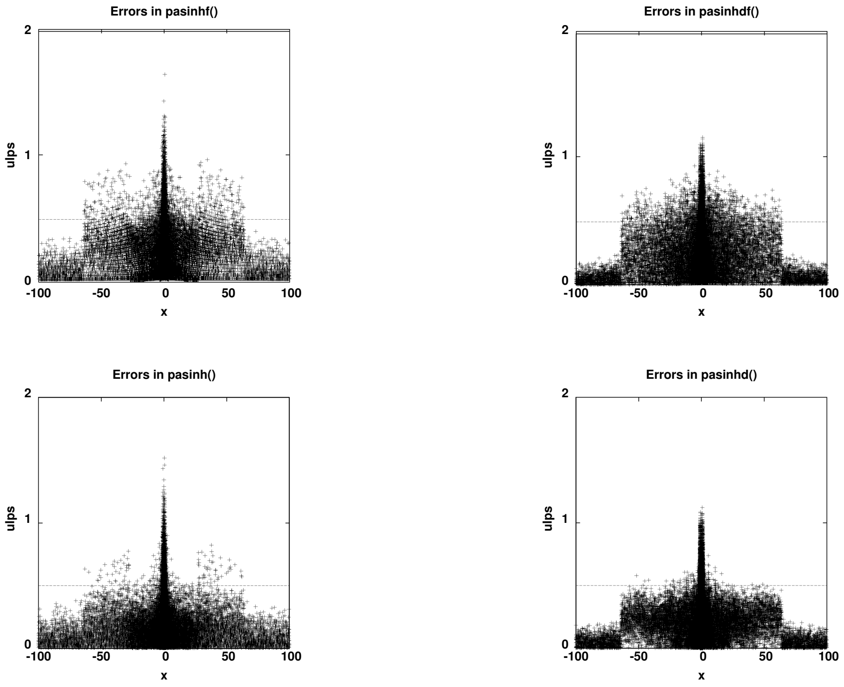


Figure 23.21: Errors in pair-precision `PASINH()` functions.

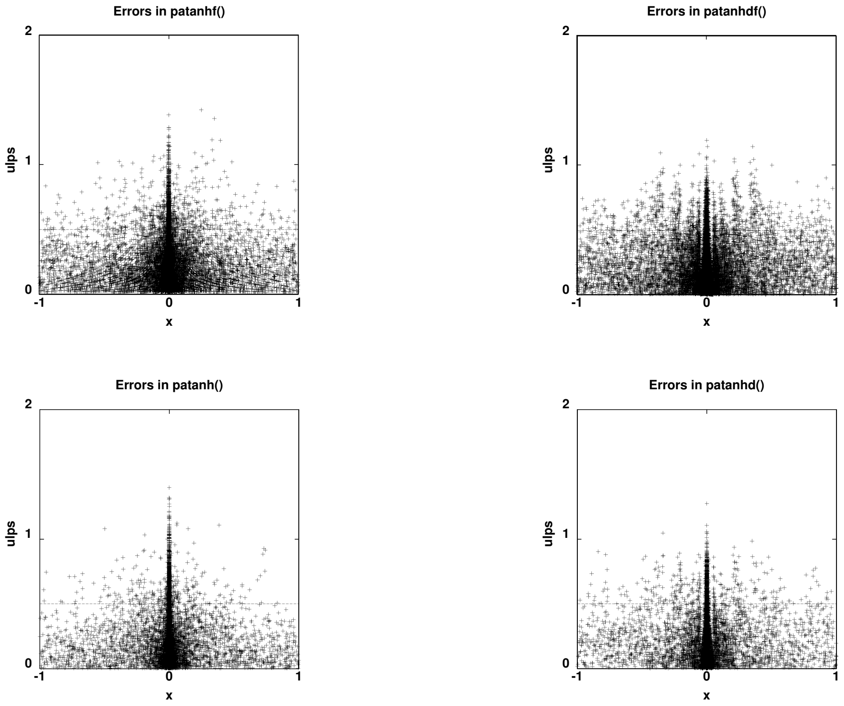


Figure 23.22: Errors in pair-precision `PATANH()` functions.

24 Accuracy of the Cody/Waite algorithms

> ... SOME DEGREE OF ACCURACY MUST BE SACRIFICED TO CONCISENESS.
>
> — SAMUEL JOHNSON, LEXICOGRAPHER (1760).

To assess the accuracy of the elementary functions provided by the mathcw library, it is best to let the ELEFUNT numbers speak for themselves. **Table 24.1** on the next page through **Table 24.23** on page 822 show typical results obtained at this author's site and the Hewlett–Packard Test Drive Laboratory. The tables are prepared with the help of a small awk program that filters and reformats the output of `make check` to avoid errors from manual transcription.

Results should be quite similar with *any* compiler on any system with IEEE 754 arithmetic, although those from older architectures with poor floating-point rounding behavior may be a bit or two worse. Any larger errors should be investigated, because they may be due to incorrect code generation, or to invalid optimizations.

For comparison with other libraries, see the results for:

- the GNU / LINUX native math library (`-lm`) on IA-64 in **Table 24.16** on page 819;
- the native Sun Microsystems library (`-lm`) in **Table 24.17** on page 820;
- IBM APMathLib (`-lultim`) in **Table 24.19** on page 820;
- Sun Microsystems' fdlibm (`-lfdm`) (see **Chapter 25.3** on page 824) in **Table 24.20** on page 821;
- Sun Microsystems' libmcr (`-lmcr`) (see **Chapter 25.3** on page 824) in **Table 24.21** on page 821; and
- Moshier's Cephes function library (`-lmf`) (see **Chapter 25.2** on page 823): **Table 24.22** and **Table 24.23** on page 822.

Because APMathLib and libmcr both claim to return correctly rounded results, you might expect that the ELEFUNT test reports would show worst-case errors of zero. However, that does not happen because of the way ELEFUNT measures errors. It does so by exploiting identities where the quantities involved, apart from the function values, are all exactly representable.

For example, to test $\sin(x)$, ELEFUNT uses the identity

$$\sin(x) = 3\sin(x/3) - 4(\sin(x/3))^3.$$

It then selects 2000 random values logarithmically distributed in the interval to be tested and purified so that both x and $x/3$ are exact, and compares the two sides of the identity, accumulating average and worst-case errors.

Even if the two function values are correctly rounded, the right-hand side requires four multiplies and one add, and it is likely that those five operations introduce additional rounding errors. Indeed, in the worst case with default IEEE 754 *round-to-nearest* behavior, each operation would have an error of 0.5 bits, and the cumulative error could be five times larger, or 2.5 bits.

Of course, error accumulation is rarely that bad, but without using higher-precision computation in the test package, tests of the identities incur some rounding error that could amount to a bit or so. To assess the size of the rounding errors in the identities, in the ELEFUNT test of the single-precision $\sin(x)$ on Sun Microsystems SOLARIS 10 SPARC, a simple change of the data type from `float` to `double` for the three variables involved in the error computation reduced the average reported errors by 0.47 bits.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_24

Table 24.1: ELEFUNT bit-loss report for mathcw compiled with native cc on Hewlett–Packard/Compaq/DEC Alpha OSF/1 5.1.
The asin tests cover acos, the atan tests cover atan2, the sin tests cover cos, and the tan tests cover cot.
Even though the worst-case errors are a few bits, the average errors are all zero, which is excellent. Similar behavior is seen on almost every platform on which the mathcw library has been tested.
Losses in boldface text exceed 2 bits.

float			**double**			**long double**		
name	**rms**	**worst**	**name**	**rms**	**worst**	**name**	**rms**	**worst**
asinf	0.00	1.63	asin	0.00	1.00	asinl	0.00	1.00
atanf	0.00	**2.44**	atan	0.00	**2.49**	atanl	0.00	1.93
cbrtf	0.00	1.00	cbrt	0.00	1.00	cbrtl	0.00	1.00
expf	0.00	1.60	exp	0.00	1.55	expl	0.00	**2.55**
expm1f	0.00	**2.35**	expm1	0.00	**2.41**	expm1l	0.00	**2.38**
logf	0.00	**2.40**	log	0.00	**2.45**	logl	0.00	**2.34**
log1pf	0.00	**3.56**	log1p	0.00	**3.39**	log1pl	0.00	**3.58**
powf	0.00	1.26	pow	0.00	**3.92**	powl	0.00	**8.06**
rsqrtf	0.00	1.17	rsqrt	0.00	1.42	rsqrtl	0.00	1.70
sinf	0.00	1.83	sin	0.00	1.93	sinl	0.00	1.75
sinhf	0.00	**2.00**	sinh	0.00	1.93	sinhl	0.00	**2.35**
sqrtf	0.00	1.00	sqrt	0.00	1.00	sqrtl	0.00	1.00
tanf	0.00	**2.54**	tan	0.00	**2.53**	tanl	0.00	**2.50**
tanhf	0.00	1.43	tanh	0.00	1.58	tanhl	0.00	1.97

Table 24.2: ELEFUNT bit-loss report for mathcw compiled with native cc on Hewlett–Packard/Compaq/DEC Alpha OSF/1 5.1.
Here, the default test count of 2000 was increased by setting the environment variable `MAXTEST` to 3 000 000, beyond the period of 2 796 202 of the Hansson–Pike–Hill random-number generator used in ELEFUNT.
Compare these results with those in **Table 24.1** for the default test count.

float			**double**			**long double**		
name	**rms**	**worst**	**name**	**rms**	**worst**	**name**	**rms**	**worst**
asinf	0.00	1.90	asin	0.00	1.00	asinl	0.00	1.00
atanf	0.00	**2.52**	atan	0.00	**2.52**	atanl	0.00	**2.52**
cbrtf	0.00	1.00	cbrt	0.00	1.00	cbrtl	0.00	1.00
expf	0.00	**2.08**	exp	0.00	**2.08**	expl	0.00	**2.81**
expm1f	0.00	**3.05**	expm1	0.00	**2.86**	expm1l	0.00	**2.75**
logf	0.00	**2.71**	log	0.00	**2.58**	logl	0.00	**2.75**
log1pf	0.00	**3.73**	log1p	0.00	**3.97**	log1pl	0.00	**3.92**
powf	0.00	**2.18**	pow	0.00	**4.54**	powl	0.00	**8.48**
rsqrtf	0.00	1.17	rsqrt	0.00	1.57	rsqrtl	0.00	1.83
sinf	0.00	**2.00**	sin	0.00	**7.17**	sinl	0.00	**9.74**
sinhf	0.00	**2.50**	sinh	0.00	**2.21**	sinhl	0.00	**2.70**
sqrtf	0.00	1.00	sqrt	0.00	1.00	sqrtl	0.00	1.00
tanf	0.00	**2.85**	tan	0.00	**2.90**	tanl	0.00	**2.90**
tanhf	0.00	**2.33**	tanh	0.00	**2.34**	tanhl	0.00	**2.28**

Table 24.3: ELEFUNT bit-loss report for mathcw compiled with GNU `gcc` on GNU/Linux on AMD64.

float			double			long double		
name	**rms**	**worst**	**name**	**rms**	**worst**	**name**	**rms**	**worst**
asinf	0.00	1.63	asin	0.00	1.00	asinl	0.00	1.00
atanf	0.00	1.92	atan	0.00	**2.49**	atanl	0.00	1.92
cbrtf	0.00	1.00	cbrt	0.00	1.00	cbrtl	0.00	1.00
expf	0.00	1.60	exp	0.00	1.55	expl	0.00	**2.43**
expm1f	0.00	**2.35**	expm1	0.00	**2.41**	expm1l	0.00	**2.44**
logf	0.00	**2.40**	log	0.00	**2.45**	logl	0.00	**2.50**
log1pf	0.00	**3.56**	log1p	0.00	**3.39**	log1pl	0.00	**3.39**
powf	0.00	1.26	pow	0.00	**3.92**	powl	0.00	**8.06**
rsqrtf	0.00	1.17	rsqrt	0.00	1.42	rsqrtl	0.00	1.50
sinf	0.00	1.83	sin	0.00	1.93	sinl	0.00	1.98
sinhf	0.00	**2.00**	sinh	0.00	1.93	sinhl	0.00	**2.53**
sqrtf	0.00	1.00	sqrt	0.00	1.00	sqrtl	0.00	1.00
tanf	0.00	**2.54**	tan	0.00	**2.53**	tanl	0.00	**2.53**
tanhf	0.00	1.43	tanh	0.00	1.58	tanhl	0.00	1.75

Table 24.4: ELEFUNT bit-loss report for mathcw compiled with GNU `gcc` on GNU/Linux on AMD64.
Here, the default test count of 2000 was increased by setting the environment variable `MAXTEST` to 3 000 000, beyond the period of 2 796 202 of the Hansson–Pike–Hill random-number generator used in ELEFUNT.
The large worst-case error in `sin()` happens for a random argument close to a multiple of $\pi/2$: $x = 23.5619444748828748 = 15\pi/2 - 0.00000042704057449$. Even with careful argument reduction, some loss of accuracy is unavoidable for such arguments.

float			double			long double		
name	**rms**	**worst**	**name**	**rms**	**worst**	**name**	**rms**	**worst**
asinf	0.00	1.90	asin	0.00	1.00	asinl	0.00	1.00
atanf	0.00	**2.52**	atan	0.00	**2.52**	atanl	0.00	**2.52**
cbrtf	0.00	1.00	cbrt	0.00	1.00	cbrtl	0.00	1.00
expf	0.00	**2.08**	exp	0.00	**2.08**	expl	0.00	**2.67**
expm1f	0.00	**3.05**	expm1	0.00	**2.86**	expm1l	0.00	**2.88**
logf	0.00	**2.71**	log	0.00	**2.58**	logl	0.00	**2.73**
log1pf	0.00	**3.73**	log1p	0.00	**3.97**	log1pl	0.00	**4.02**
powf	0.00	**2.18**	pow	0.00	**4.54**	powl	0.00	**8.79**
rsqrtf	0.00	1.17	rsqrt	0.00	1.57	rsqrtl	0.00	1.50
sinf	0.00	**2.00**	sin	0.00	**7.17**	sinl	0.00	**2.01**
sinhf	0.00	**2.50**	sinh	0.00	**2.21**	sinhl	0.00	**2.80**
sqrtf	0.00	1.00	sqrt	0.00	1.00	sqrtl	0.00	1.00
tanf	0.00	**2.85**	tan	0.00	**2.90**	tanl	0.00	**2.90**
tanhf	0.00	**2.33**	tanh	0.00	**2.34**	tanhl	0.00	**2.00**

Table 24.5: ELEFUNT bit-loss report for mathcw compiled with GNU gcc on GNU/Linux on IA-32. Losses in boldface text exceed 2 bits.

float			double			long double		
name	**rms**	**worst**	**name**	**rms**	**worst**	**name**	**rms**	**worst**
asinf	0.00	1.64	asin	0.00	1.00	asinl	0.00	1.00
atanf	0.00	1.15	atan	0.00	1.86	atanl	0.00	1.92
cbrtf	0.00	1.00	cbrt	0.00	1.00	cbrtl	0.00	1.00
expf	0.00	1.00	exp	0.00	1.00	expl	0.00	**2.43**
expm1f	0.00	**2.04**	expm1	0.00	1.94	expm1l	0.00	**2.44**
logf	0.00	**2.33**	log	0.00	**2.45**	logl	0.00	**2.50**
log1pf	0.00	**3.56**	log1p	0.00	**3.39**	log1pl	0.00	**3.39**
powf	0.00	1.23	pow	0.00	**3.74**	powl	0.00	**8.06**
rsqrtf	0.00	1.01	rsqrt	0.00	1.10	rsqrtl	0.00	1.50
sinf	0.00	1.41	sin	0.00	1.26	sinl	0.00	1.98
sinhf	0.00	1.63	sinh	0.00	1.49	sinhl	0.00	**2.53**
sqrtf	0.00	1.00	sqrt	0.00	1.00	sqrtl	0.00	1.00
tanf	0.00	**2.20**	tan	0.00	**2.16**	tanl	0.00	**2.53**
tanhf	0.00	0.99	tanh	0.00	1.00	tanhl	0.00	1.75

Table 24.6: ELEFUNT bit-loss report for mathcw compiled with Intel icc on GNU/Linux on IA-64 (Itanium-2), without use of multiply-add wrappers.
Compare these results with those for multiply-add wrappers in Table 24.7 on the next page.

float			double			long double		
name	**rms**	**worst**	**name**	**rms**	**worst**	**name**	**rms**	**worst**
asinf	0.00	1.63	asin	0.00	1.00	asinl	0.00	1.00
atanf	0.00	**2.44**	atan	0.00	**2.49**	atanl	0.00	1.92
cbrtf	0.00	1.00	cbrt	0.00	1.00	cbrtl	0.00	1.00
expf	0.00	1.60	exp	0.00	1.55	expl	0.00	**2.43**
expm1f	0.00	**2.35**	expm1	0.00	**2.41**	expm1l	0.00	**2.44**
logf	0.00	**2.40**	log	0.00	**2.45**	logl	0.00	**2.50**
log1pf	0.00	**3.56**	log1p	0.00	**3.39**	log1pl	0.00	**3.39**
powf	0.00	1.26	pow	0.00	**3.92**	powl	0.00	**8.06**
rsqrtf	0.00	1.17	rsqrt	0.00	1.42	rsqrtl	0.00	1.50
sinf	0.00	1.83	sin	0.00	1.93	sinl	0.00	1.98
sinhf	0.00	**2.00**	sinh	0.00	1.93	sinhl	0.00	**2.53**
sqrtf	0.00	1.00	sqrt	0.00	1.00	sqrtl	0.00	1.00
tanf	0.00	**2.54**	tan	0.00	**2.53**	tanl	0.00	**2.53**
tanhf	0.00	1.43	tanh	0.00	1.58	tanhl	0.00	1.75

Table 24.7: ELEFUNT bit-loss report for mathcw compiled with Intel `icc` on GNU/Linux on IA-64 (Itanium-2), with multiply-add wrappers enabled.
Results with worst-case errors of zero bits are flagged with a check mark (✓).
Compared to the results without multiply-add wrappers in Table 24.6 on the facing page, in no case are the results worse, and the `pow()` function is dramatically improved for `float` and `double`, although curiously, not for `long double`.
Contrast these results with the always-better ones for HP-UX on IA-64 in Table 24.8.

	float			**double**			**long double**	
name	**rms**	**worst**	**name**	**rms**	**worst**	**name**	**rms**	**worst**
asinf	0.00	1.64	asin	0.00	1.00	asinl	0.00	1.00
atanf	0.00	1.92	atan	0.00	1.91	atanl	0.00	1.92
cbrtf	0.00	1.00	cbrt	0.00	1.00	cbrtl	0.00	1.00
expf	0.00	1.00	exp	0.00	1.00	expl	0.00	1.62
expm1f	0.00	**2.17**	expm1	0.00	**2.14**	expm1l	0.00	**2.19**
logf	0.00	**2.12**	log	0.00	**2.10**	logl	0.00	**2.50**
log1pf	0.00	**3.15**	log1p	0.00	**3.39**	log1pl	0.00	**3.39**
powf	0.00	0.95	pow	0.00	0.85	powl	0.00	**8.06**
rsqrtf	0.00	1.17	rsqrt	0.00	1.42	rsqrtl	0.00	1.50
sinf	0.00	1.08	sin	0.00	1.17	sinl	0.00	1.54
sinhf	0.00	1.84	sinh	0.00	1.60	sinhl	0.00	1.45
sqrtf	0.00	0.00✓	sqrt	0.00	0.00✓	sqrtl	0.00	1.00
tanf	0.00	**2.54**	tan	0.00	**2.53**	tanl	0.00	**2.49**
tanhf	0.00	1.43	tanh	0.00	1.58	tanhl	0.00	1.65

Table 24.8: ELEFUNT bit-loss report for mathcw compiled with native `cc` on Hewlett–Packard HP-UX 11.23 on IA-64 (Itanium-2).

	float			**double**			**long double**	
name	**rms**	**worst**	**name**	**rms**	**worst**	**name**	**rms**	**worst**
asinf	0.00	1.00	asin	0.00	1.00	asinl	0.00	1.00
atanf	0.00	1.00	atan	0.00	1.00	atanl	0.00	1.00
cbrtf	0.00	0.00✓	cbrt	0.00	0.00✓	cbrtl	0.00	0.00✓
expf	0.00	1.00	exp	0.00	1.00	expl	0.00	1.00
expm1f	0.00	1.13	expm1	0.00	1.07	expm1l	0.00	1.08
logf	0.00	**2.12**	log	0.00	**2.10**	logl	0.00	**2.02**
log1pf	0.00	**3.56**	log1p	0.00	**3.39**	log1pl	0.00	**3.58**
powf	0.00	0.98	pow	0.00	0.85	powl	0.00	0.00✓
rsqrtf	0.00	0.00✓	rsqrt	0.00	0.00✓	rsqrtl	0.00	0.00✓
sinf	0.00	1.42	sin	0.00	1.40	sinl	0.00	1.28
sinhf	0.00	1.62	sinh	0.00	1.60	sinhl	0.00	1.50
sqrtf	0.00	0.00✓	sqrt	0.00	0.00✓	sqrtl	0.00	0.00✓
tanf	0.00	1.97	tan	0.00	1.95	tanl	0.00	1.97
tanhf	0.00	1.43	tanh	0.00	1.62	tanhl	0.00	1.86

Table 24.9: ELEFUNT bit-loss report for mathcw compiled with native `cc` on Hewlett–Packard HP-UX 11.23 on PA-RISC.
That system has no `long double` math library routines at all, although the compiler and I/O library support that data type. Consequently, it was necessary to use a renamed copy of `mathcw.h` as a stand-in for the system header file `<math.h>` in order to compile the `long double` test suite with correct math library prototypes.

float			**double**			**long double**		
name	**rms**	**worst**	**name**	**rms**	**worst**	**name**	**rms**	**worst**
asinf	0.00	1.63	asin	0.00	1.00	asinl	0.00	1.00
atanf	0.00	**2.44**	atan	0.00	**2.49**	atanl	0.00	1.93
cbrtf	0.00	1.00	cbrt	0.00	1.00	cbrtl	0.00	0.00✓
expf	0.00	1.60	exp	0.00	1.55	expl	0.00	**2.55**
expm1f	0.00	**2.35**	expm1	0.00	**2.41**	expm1l	0.00	**2.38**
logf	0.00	**2.40**	log	0.00	**2.45**	logl	0.00	**2.34**
log1pf	0.00	**3.56**	log1p	0.00	**3.39**	log1pl	0.00	**3.58**
powf	0.00	1.26	pow	0.00	**3.92**	powl	0.00	**8.06**
rsqrtf	0.00	1.17	rsqrt	0.00	1.42	rsqrtl	0.00	1.70
sinf	0.00	1.83	sin	0.00	1.93	sinl	0.00	1.75
sinhf	0.00	**2.00**	sinh	0.00	1.93	sinhl	0.00	**2.35**
sqrtf	0.00	1.00	sqrt	0.00	1.00	sqrtl	0.00	1.00
tanf	0.00	**2.54**	tan	0.00	**2.53**	tanl	0.00	**2.50**
tanhf	0.00	1.43	tanh	0.00	1.58	tanhl	0.00	1.97

Table 24.10: ELEFUNT bit-loss report for mathcw compiled with native `c89` on IBM AIX 4.2 on POWER.

float			**double**		
name	**rms**	**worst**	**name**	**rms**	**worst**
asinf	0.00	1.64	asin	0.00	1.00
atanf	0.00	1.15	atan	0.00	1.91
cbrtf	0.00	1.00	cbrt	0.00	1.00
expf	0.00	1.00	exp	0.00	1.64
expm1f	0.00	**2.04**	expm1	0.00	**2.14**
logf	0.00	**2.33**	log	0.00	**2.45**
log1pf	0.00	**3.18**	log1p	0.00	**3.39**
powf	0.00	1.23	pow	0.00	**3.74**
rsqrtf	0.00	0.77	rsqrt	0.00	1.42
sinf	0.00	1.41	sin	0.00	1.58
sinhf	0.00	1.63	sinh	0.00	1.93
sqrtf	0.00	1.00	sqrt	0.00	1.00
tanf	0.00	**2.20**	tan	0.00	**2.53**
tanhf	0.00	0.99	tanh	0.00	1.58

Table 24.11: ELEFUNT bit-loss report for mathcw compiled with GNU `gcc` on GNU/LINUX on MIPS R4400SC.
Compare these results with those for the Silicon Graphics IRIX MIPS R10000 in Table 24.12.
There is no support for `long double` on this system.

	float	
name	**rms**	**worst**
asinf	0.00	1.63
atanf	0.00	1.92
cbrtf	0.00	1.00
expf	0.00	1.60
expm1f	0.00	**2.35**
logf	0.00	**2.40**
log1pf	0.00	**3.56**
powf	0.00	1.26
rsqrtf	0.00	1.17
sinf	0.00	1.83
sinhf	0.00	**2.00**
sqrtf	0.00	1.00
tanf	0.00	**2.54**
tanhf	0.00	1.43

	double	
name	**rms**	**worst**
asin	0.00	1.00
atan	0.00	**2.49**
cbrt	0.00	1.00
exp	0.00	1.55
expm1	0.00	**2.41**
log	0.00	**2.45**
log1p	0.00	**3.39**
pow	0.00	**3.92**
rsqrt	0.00	1.49
sin	0.00	1.68
sinh	0.00	1.93
sqrt	0.00	1.00
tan	0.00	**2.29**
tanh	0.00	1.58

Table 24.12: ELEFUNT bit-loss report for mathcw compiled with native `cc` on Silicon Graphics IRIX 6.5 on MIPS R10000.
Like IBM AIX on RS/6000, this system implements `long double` as a pair of `double` values, so although precision increases from 53 to 106 significand bits, the exponent range is unchanged. On both systems, the exponents of each member of the pair are not clamped to a constant offset, so it is possible to represent a number with unknown garbage in the middle of its significand.
By default, subnormals are not supported on IRIX, and can only be enabled with a little-known run-time system call, rather than a compile-time option. They are *not* present in the ELEFUNT tests in the mathcw validation suite.
Nonzero average errors are shaded, and are rarely seen on other systems.

	float	
name	**rms**	**worst**
asinf	0.00	1.98
atanf	0.00	1.92
cbrtf	0.00	1.00
expf	0.00	1.60
expm1f	0.00	**2.35**
logf	0.00	**2.40**
log1pf	0.00	**3.56**
powf	0.00	1.26
rsqrtf	0.00	1.17
sinf	0.00	1.83
sinhf	0.00	**2.00**
sqrtf	0.00	0.00✓
tanf	0.00	**2.54**
tanhf	0.00	1.43

	double	
name	**rms**	**worst**
asin	0.00	1.98
atan	0.00	**2.44**
cbrt	0.00	1.00
exp	0.00	1.55
expm1	0.00	**2.41**
log	0.00	**2.45**
log1p	0.00	**3.39**
pow	0.00	**3.92**
rsqrt	0.00	1.55
sin	0.00	1.68
sinh	0.00	1.93
sqrt	0.00	0.00✓
tan	0.00	**2.29**
tanh	0.00	1.95

	long double	
name	**rms**	**worst**
asinl	0.00	**2.94**
atanl	0.00	**2.93**
cbrtl	0.00	0.00✓
expl	0.00	**2.00**
expm1l	0.00	**3.59**
logl	0.00	**2.41**
log1pl	1.07	**4.98**
powl	0.00	**5.36**
rsqrtl	0.00	**2.35**
sinl	0.47	**2.58**
sinhl	0.00	**2.42**
sqrtl	0.00	**2.44**
tanl	0.00	**3.53**
tanhl	0.50	**3.59**

Table 24.13: ELEFUNT bit-loss report for mathcw compiled with `gcc` on Apple Mac OS X on PowerPC. This system provides neither `float` nor `long double` routines in the native math library, nor `long double` support in the compiler and library.

	float	
name	**rms**	**worst**
asinf	0.00	1.63
atanf	0.00	**2.44**
cbrtf	0.00	1.00
expf	0.00	1.60
expm1f	0.00	**2.35**
logf	0.00	**2.40**
log1pf	0.00	**3.56**
powf	0.00	1.26
rsqrtf	0.00	1.17
sinf	0.00	1.83
sinhf	0.00	**2.00**
sqrtf	0.00	1.00
tanf	0.00	**2.54**
tanhf	0.00	1.43

	double	
name	**rms**	**worst**
asin	0.00	1.00
atan	0.00	**2.49**
cbrt	0.00	1.00
exp	0.00	1.55
expm1	0.00	**2.41**
log	0.00	**2.45**
log1p	0.00	**3.39**
pow	0.00	**3.92**
rsqrt	0.00	1.49
sin	0.00	1.93
sinh	0.00	1.93
sqrt	0.00	1.00
tan	0.00	**2.53**
tanh	0.00	1.58

Table 24.14: ELEFUNT bit-loss report for mathcw compiled with native `c99` on Sun Microsystems Solaris 10 on SPARC.

The nonzero average error for `sinhf()`, marked by shading, is one of the rare cases where that small infelicity has been seen.

Compare these results with those for Solaris 10 on IA-32 in **Table 24.15** on the next page.

	float	
name	**rms**	**worst**
asinf	0.00	0.91
atanf	0.00	0.96
cbrtf	0.00	0.76
expf	0.00	0.80
expm1f	0.00	1.47
logf	0.00	1.91
log1pf	0.00	**2.83**
powf	0.00	0.87
rsqrtf	0.00	0.39
sinf	0.00	0.90
sinhf	0.02	1.58
sqrtf	0.00	0.91
tanf	0.00	1.87
tanhf	0.00	0.80

	double	
name	**rms**	**worst**
asin	0.00	0.97
atan	0.00	1.02
cbrt	0.00	0.90
exp	0.00	0.83
expm1	0.00	1.39
log	0.00	**2.09**
log1p	0.00	**2.09**
pow	0.00	1.36
rsqrt	0.00	0.86
sin	0.00	1.24
sinh	0.00	0.99
sqrt	0.00	0.83
tan	0.00	1.88
tanh	0.00	1.00

	long double	
name	**rms**	**worst**
asinl	0.00	0.86
atanl	0.00	1.40
cbrtl	0.00	0.00✓
expl	0.00	0.97
expm1l	0.00	1.16
logl	0.00	1.33
log1pl	0.00	**2.36**
powl	0.00	**6.07**
rsqrtl	0.00	1.48
sinl	0.00	1.03
sinhl	0.00	1.43
sqrtl	0.00	0.81
tanl	0.00	1.62
tanhl	0.00	0.98

Table 24.15: ELEFUNT bit-loss report for mathcw compiled with native `c99` on Sun Microsystems SOLARIS 10 on IA-32.
In almost every case, the longer registers on IA-32 produce higher accuracy than the SOLARIS SPARC results in Table 24.14 on the preceding page.

float			**double**			**long double**		
name	**rms**	**worst**	**name**	**rms**	**worst**	**name**	**rms**	**worst**
asinf	0.00	1.64	asin	0.00	1.00	asinl	0.00	1.00
atanf	0.00	1.15	atan	0.00	1.86	atanl	0.00	1.92
cbrtf	0.00	1.00	cbrt	0.00	0.00✓	cbrtl	0.00	1.00
expf	0.00	1.00	exp	0.00	1.00	expl	0.00	**2.43**
expm1f	0.00	**2.04**	expm1	0.00	1.94	expm1l	0.00	**2.44**
logf	0.00	**2.40**	log	0.00	**2.45**	logl	0.00	**2.50**
log1pf	0.00	**3.18**	log1p	0.00	**3.36**	log1pl	0.00	**3.39**
powf	0.00	1.23	pow	0.00	**3.74**	powl	0.00	**8.06**
rsqrtf	0.00	0.78	rsqrt	0.00	1.10	rsqrtl	0.00	1.50
sinf	0.00	1.41	sin	0.00	1.38	sinl	0.00	1.98
sinhf	0.00	1.63	sinh	0.00	1.49	sinhl	0.00	**2.53**
sqrtf	0.00	1.00	sqrt	0.00	1.00	sqrtl	0.00	1.00
tanf	0.00	**2.20**	tan	0.00	**2.04**	tanl	0.00	**2.53**
tanhf	0.00	0.99	tanh	0.00	1.00	tanhl	0.00	1.75

Table 24.16: ELEFUNT bit-loss report for the native GNU math library on GNU/LINUX (Red Hat Advanced Server 2.1AS) on IA-64.

float			**double**			**long double**		
name	**rms**	**worst**	**name**	**rms**	**worst**	**name**	**rms**	**worst**
asinf	0.00	1.00	asin	0.00	1.00	asinl	0.00	1.00
atanf	0.00	1.31	atan	0.00	1.00	atanl	0.00	1.00
cbrtf	0.00	0.00✓	cbrt	0.00	0.00✓	cbrtl	0.00	0.00✓
expf	0.00	1.00	exp	0.00	1.00	expl	0.00	1.00
expm1f	0.00	1.13	expm1	0.00	1.07	expm1l	0.00	1.08
logf	0.00	**2.12**	log	0.00	**2.10**	logl	0.00	**2.07**
log1pf	0.00	**3.15**	log1p	0.00	**3.39**	log1pl	0.00	**3.16**
powf	0.00	0.98	pow	0.00	0.85	powl	0.00	0.00✓
sinf	0.00	1.42	sin	0.00	1.40	sinl	0.00	1.45
sinhf	0.00	1.62	sinh	0.00	1.60	sinhl	0.00	1.44
sqrtf	0.00	0.00✓	sqrt	0.00	0.00✓	sqrtl	0.00	0.00✓
tanf	0.00	1.97	tan	0.00	1.95	tanl	0.00	**2.03**
tanhf	0.00	**2.13**	tanh	0.00	**2.13**	tanhl	0.00	**2.08**

Table 24.17: ELEFUNT bit-loss report for the native Sun Microsystems math library for SOLARIS 10 on SPARC. Sun Microsystems has superb floating-point support, and the library quality is excellent.
Compare these results with those in Table 24.18 for the native SOLARIS IA-32 library.

float			double			long double		
name	**rms**	**worst**	**name**	**rms**	**worst**	**name**	**rms**	**worst**
asinf	0.00	0.99	asin	0.00	0.97	asinl	0.00	0.86
atanf	0.00	0.79	atan	0.00	0.94	atanl	0.00	0.78
cbrtf	0.00	0.00✓	cbrt	0.00	0.00✓	cbrtl	0.00	0.00✓
expf	0.00	0.69	exp	0.00	0.83	expl	0.00	0.93
expm1f	0.00	0.96	expm1	0.00	0.93	expm1l	0.00	0.92
logf	0.00	0.66	log	0.00	**2.09**	logl	0.00	1.33
log1pf	0.00	**2.83**	log1p	0.00	**2.09**	log1pl	0.00	1.90
powf	0.00	0.05	pow	0.00	0.20	powl	0.00	0.00✓
rsqrtf	0.00	0.39	rsqrt	0.00	0.86	rsqrtl	0.00	1.17
sinf	0.00	0.76	sin	0.00	0.82	sinl	0.00	0.60
sinhf	0.00	0.95	sinh	0.00	0.98	sinhl	0.00	0.90
sqrtf	0.00	0.00✓	sqrt	0.00	0.00✓	sqrtl	0.00	0.00✓
tanf	0.00	1.96	tan	0.00	1.28	tanl	0.00	1.29
tanhf	0.00	0.80	tanh	0.15	1.03	tanhl	0.00	0.98

Table 24.18: ELEFUNT bit-loss report for the native Sun Microsystems math library for SOLARIS 10 on IA-32. Results are similar to those in Table 24.17 for SPARC, but the `log()` function, and `long double` `powl()`, `tanl()`, and `tanhl()` functions, need improvement.

float			double			long double		
name	**rms**	**worst**	**name**	**rms**	**worst**	**name**	**rms**	**worst**
asinf	0.00	1.00	asin	0.00	1.00	asinl	0.00	1.98
atanf	0.00	1.00	atan	0.00	1.00	atanl	0.00	1.00
cbrtf	0.00	0.00✓	cbrt	0.00	0.00✓	cbrtl	0.00	0.00✓
expf	0.00	1.00	exp	0.00	1.00	expl	0.00	1.00
expm1f	0.00	1.00	expm1	0.00	1.00	expm1l	0.00	1.72
logf	0.00	**2.12**	log	0.00	**2.10**	logl	0.00	**2.41**
log1pf	0.00	**3.14**	log1p	0.00	**3.17**	log1pl	0.00	**3.60**
powf	0.00	0.98	pow	0.00	0.99	powl	0.00	**12.93**
rsqrtf	0.00	0.78	rsqrt	0.00	1.10	rsqrtl	0.00	1.50
sinf	0.00	0.63	sin	0.00	0.69	sinl	0.00	1.41
sinhf	0.00	1.43	sinh	0.00	1.49	sinhl	0.00	1.67
sqrtf	0.00	0.00✓	sqrt	0.00	0.00✓	sqrtl	0.00	0.00✓
tanf	0.00	1.78	tan	0.00	1.61	tanl	0.00	**2.03**
tanhf	0.00	0.99	tanh	0.00	1.56	tanhl	0.00	**2.40**

Table 24.19: ELEFUNT bit-loss report for IBM APMathLib compiled with native `cc` on Sun Microsystems SOLARIS 10 on SPARC. See the text in Section 24 on page 811 for why these results are not all zero. The library provides only `double` versions of the elementary functions.

double		
name	**rms**	**worst**
asin	0.00	1.00
atan	0.00	1.00
exp	0.00	1.00
log	0.00	**2.45**
pow	0.00	**5.96**
sin	0.00	1.40
sinh	0.00	1.83
sqrt	0.00	0.00✓
tan	0.00	1.95
tanh	0.00	**2.06**

Table 24.20: ELEFUNT bit-loss report for Sun Microsystems fdlibm compiled with native `cc` on Sun Microsystems SOLARIS 8 on SPARC and run on SOLARIS 10. The library provides only `double` versions of the elementary functions.

	double	
name	**rms**	**worst**
asin	0.00	1.00
atan	0.00	1.00
exp	0.00	1.00
log	0.00	**2.45**
pow	0.00	**5.97**
sin	0.00	1.45
sinh	0.00	1.83
sqrt	0.00	0.00✓
tan	0.00	1.95
tanh	0.00	**2.06**

Table 24.21: ELEFUNT bit-loss report for Sun Microsystems libmcr compiled with native `cc` on Sun Microsystems SOLARIS 10 on SPARC. See the text in **Section 24** on page 811 for why these results are not all zero. Version 0.9 of libmcr contains only `double` versions of the elementary functions.

	double	
name	**rms**	**worst**
asin	0.00	1.00
atan	0.00	1.00
exp	0.00	1.00
log	0.00	**2.45**
pow	0.00	**5.96**
sin	0.00	1.40
sinh	0.00	1.60
sqrt	0.00	1.00
tan	0.00	1.95
tanh	0.00	1.58

Table 24.22: ELEFUNT bit-loss report for Moshier's Cephes library compiled with GNU `gcc` on Sun Microsystems SOLARIS 8 on SPARC, and tested on SOLARIS 10. The Cephes library does not support 128-bit `long double`, so those functions are not tested.

	float	
name	**rms**	**worst**
asinf	0.00	0.99
atanf	0.00	0.96
cbrtf	0.00	0.00✓
expf	0.00	0.69
expm1f	0.00	1.47
logf	0.00	0.91
log1pf	0.00	**2.51**
powf	0.00	0.88
sinf	0.00	0.90
sinhf	0.00	0.95
sqrtf	0.00	0.00✓
tanf	0.00	1.06
tanhf	0.00	0.80

	double	
name	**rms**	**worst**
asin	0.00	0.97
atan	0.00	0.94
cbrt	0.00	0.00✓
exp	0.00	1.63
expm1	0.00	0.91
log	0.00	1.67
log1p	0.00	1.78
pow	0.00	**2.57**
sin	0.00	1.24
sinh	0.00	0.99
sqrt	0.00	0.37
tan	0.00	1.35
tanh	0.15	1.31

Table 24.23: ELEFUNT bit-loss report for Moshier's Cephes library compiled with native `c99` on Sun Microsystems Solaris 10 on IA-32.

float			double			long double		
name	**rms**	**worst**	**name**	**rms**	**worst**	**name**	**rms**	**worst**
asinf	0.00	1.00	asin	0.00	1.00	asinl	0.00	1.00
atanf	0.00	1.93	atan	0.00	1.00	atanl	0.00	1.96
cbrtf	0.00	0.00✓	cbrt	0.00	0.00✓	cbrtl	0.00	0.00✓
expf	0.00	1.00	exp	0.00	1.66	expl	0.00	1.70
expm1f	n/a	n/a	expm1	0.00	1.99	expm1l	0.00	1.74
logf	0.00	**2.25**	log	0.00	**2.24**	logl	0.00	**2.17**
log1pf	n/a	n/a	log1p	0.00	**3.17**	log1pl	0.00	**3.39**
powf	0.00	1.60	pow	0.00	**3.84**	powl	0.00	**6.64**
sinf	0.00	1.41	sin	0.00	1.39	sinl	0.00	1.98
sinhf	0.00	1.43	sinh	0.00	1.90	sinhl	0.00	1.46
sqrtf	0.00	0.25	sqrt	0.00	0.00✓	sqrtl	0.00	0.50
tanf	0.00	1.78	tan	0.00	1.58	tanl	0.10	1.99
tanhf	0.00	0.99	tanh	0.00	1.00	tanhl	0.00	1.75

25 Improving upon the Cody/Waite algorithms

IN THIS ABSENCE OF NINE YEARS I FIND A GREAT IMPROVEMENT IN THE CITY
. . . SOME SAY IT HAS IMPROVED BECAUSE I HAVE BEEN AWAY.

— MARK TWAIN (1900).

The Cody/Waite recipes in their book *Software Manual for the Elementary Functions* [CW80] are an excellent source of portable methods for computation of elementary functions. It is possible to do better, but sometimes only at the expense of portability, and often only by doing at least parts of the computation in higher precision, which may mean software arithmetic and a significant performance penalty.

This author maintains a frequently updated bibliography[1] on the computation of elementary functions, and selected special functions. Together with the floating-point arithmetic bibliography cited later in this chapter, it provides a valuable resource for locating published work in that area, and one that is likely to be more up-to-date than a completed book like this one.

In the remainder of this chapter, we survey some important sources of software for elementary and special functions.

25.1 The Bell Labs libraries

The AT&T Bell Laboratories PORT library [Cow84, Chapter 13] is written in Portable Fortran. It supplies about 1500 functions covering several areas of numerical analysis. The special-functions section includes hyperbolic and inverse hyperbolic functions, inverse trigonometric functions, the gamma function, and the Bessel functions $J_n(x)$ and $I_n(x)$. Both single- and double-precision versions are supplied, and a few of the functions also have versions for complex arithmetic.

The PORT library was developed in the mid 1970s, and derives its portability by limiting the code to a portable subset of Fortran [Ryd74], and by obtaining machine-dependent constants from a family of low-level architecture-dependent primitives [FHS78a]. Constants for many different floating-point designs, including IEEE 754 arithmetic, are recorded in program comments in those primitives.

Bell Labs researcher Wayne Fullerton developed a separate library of special functions in Portable Fortran, called FNLIB. It contains about 180 functions in single- and double-precision, supplying gamma and log-gamma functions, Airy functions, Bessel functions for integer and fractional order ($J_\nu(x)$, $Y_\nu(x)$, $I_\nu(x)$, and $K_\nu(x)$), hyperbolic and trigonometric functions and their inverses, cube root, and several others. Many functions in FNLIB are also available in versions for complex arithmetic.

Both libraries are available online,[2] including a vector version of FNLIB with additional special functions [BS92, BS93].[3] Fullerton's extensive bibliography of special-function research is included, with credits, in ours.

As with most older libraries that were developed before IEEE 754 arithmetic, some of the code in the PORT and FNLIB libraries needs adjustment to behave sensibly for arguments of NaN, Infinity, and signed zero.

25.2 The Cephes library

Moshier [Mos89] provides book-length careful treatment of the elementary functions, with internal code for multiple-precision computation. His library, called Cephes, is available commercially, and free without support. Its `long double` support is restricted to the 80-bit format of IA-32, IA-64, and the now-obsolete Motorola 68000 processor family. Its distribution packaging needs considerable polishing to make it easily installable on more systems, and

[1] See `http://www.math.utah.edu/pub/tex/bib/index-table-e.html#elefunt`.

[2] See `http://www.netlib.org/port` and `http://www.netlib.org/fn`.

[3] See `http://www.netlib.org/vfnlib`.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_25

avoid the need for manually setting preprocessor symbol definitions in `mconf.h` to identify the host arithmetic and byte order in storage. Version 2.9 (April 2001) contains more than 52 200 lines of C code.

25.3 The Sun libraries

The Sun Microsystems Freely Distributable Math Library, fdlibm,[4] is designed specifically for machines with IEEE 754 64-bit arithmetic. It contains pervasive knowledge of that arithmetic system in the form of integer hexadecimal constants that represent a view of particular floating-point numbers, frequently uses bit masking and shifting on integer overlays of the high and low parts of floating-point numbers, and contains extensive tests of standards conformance. Version 5.3.2 contains 23 865 lines of C code.

The fdlibm library contains only `double` versions of the elementary functions. In most cases, `float` versions could be supplied as wrappers that use the `double` routines, if accuracy is more important than speed. The fdlibm coding practices make it difficult to tweak the algorithms or to extend coverage to other precisions or other floating-point formats.

The extensive fdlibm test suite includes checks for monotonicity, something that few others do. When polynomial approximations are used, monotonic results are probably quite hard to achieve unless correct rounding can also be guaranteed.

In early 2005, Sun Microsystems released a preliminary version of an open-source library, libmcr, for correctly rounded elementary functions in IEEE 754 arithmetic.[5] Version 0.9 contains 11 400 lines of C code, provides only `double` versions of `atan()`, `cos()`, `exp()`, `log()`, `pow()`, `sin()`, and `tan()`, and validates on only a few platforms beyond the vendor's own supported AMD64, IA-32 and SPARC products.

25.4 Mathematical functions on EPIC

In the mid-1990s, Hewlett–Packard and Intel began a joint development project to produce a new architecture class that they named *EPIC* (Explicitly Parallel Instruction Computer). Intel's product family for that architecture is named IA-64, with current implementations called Itanium-1 and Itanium-2. EPIC is similar to RISC in some respects, but differs in many others. Here are some of its notable features:

- The IA-32 computational format is extended from 80 bits to 82 bits. The extra two bits go in the exponent field, and serve to reduce the incidence of underflow and overflow in intermediate computations.
- Large register sets reduce expensive memory traffic. There are 128 82-bit floating-point registers and 128 64-bit integer registers.
- Fused multiply-add instructions are used for many purposes, including software-pipelined divide and square root.
- Instructions are packaged for simultaneous execution in bundles of three, preferably with independent inputs and outputs. Hardware locks force stalls when the input of one instruction depends on the output of another instruction in the same bundle.
- Predicate bits allow *nullification* of instruction results based on the data-dependent outcome of earlier instructions. One important use of that feature is to prevent conditional branches in loops from flushing the instruction pipeline.

For a time, EPIC systems achieved the highest performance of any current processor, and as compilers for that challenging architecture improve, those systems will do even better. However, it is not straightforward to get high performance, mostly because doing so requires keeping the instruction bundles full of useful work, and getting needed data fetched from memory early enough to avoid CPU stalls. In an excellent book about the EPIC architecture, Markstein [Mar00] shows how C code, with occasional excursions to single lines of assembly code via C-compiler `asm()` directives, can be used to write provably correctly rounded divide and square root, and produce high-performance implementations of all of the elementary functions.

[4] See `http://www.netlib.org/fdlibm/`.

[5] Search for *libmcr* at `http://www.sun.com/`.

Markstein's code makes extensive use of the fused multiply-add operation for both speed and accuracy. IBM introduced that operation in 1990 on the POWER architecture, but sadly, it was not until the 2008 revision of the IEEE 754 Standard [IEEE08, ISO11] that it was formally standardized. Most models of the most widely used desktop architectures, IA-32 and AMD64/EM64T, lack hardware support for fused multiply-add instructions at the time of writing this.

25.5 The GNU libraries

Richard Stallman began the now well-known *GNU Project* in 1983 as a reaction against restrictive commercial licensing of software. The goal was to develop a freely distributable operating system, and software suite, that could be used on many different architectures, and allow programmers to freely build on each other's work. The GNU C compiler, `gcc`, was first released in 1987, and has since been renamed the *GNU compiler collection*, because it supports numerous programming languages with separate front ends for each language, a common middle portion for code generation and optimization, and separate back ends for further code generation and optimization on dozens of processor architectures.

The accompanying library, glibc, supplies the interface to the host operating system, as well as a family of mathematical functions specified by various programming languages. It provides the native run-time library on HURD and most GNU/LINUX systems, but may not be needed on other platforms where such support already exists from other sources. The `gcc` and glibc software packages have an extensive group of software developers, many of them volunteers, and the wide use of those packages means that they are generally highly reliable.

A few GNU/LINUX distributions, such as ALPINE and DRAGORA, replace glibc by musl,[6] a new implementation of the Standard C library that strives for memory compactness, and conformance to both POSIX and the 2011 ISO C Standard. Its math library functions are borrowed from fdlibm, and from FREEBSD and OPENBSD. For the IA-32 family and its 64-bit extensions, musl uses assembly-language code to exploit the native 8087-style hardware for elementary functions. The musl library is unusual in the UNIX world of storing the mathematical functions in the basic C library, `-lc`, instead of holding them in a separate mathematical library, `-lm`. The latter exists, but is just a dummy file.

The GNU multiple-precision library, GMP [GSR$^+$04], can be installed easily on most modern platforms, and it provides highly optimized code for some architectures. It requires either 32-bit or 64-bit architectures, and is not usable with older CPU designs.

The *GNU Scientific Library*, GSL [GDT$^+$05], is a huge library with broad coverage of elementary and special functions, curve fitting, linear algebra, numerical integration, random numbers, root finding, sorting, statistics, and more. At the time of writing this, it contains more than 4500 functions, and consists of more than 415 000 lines of code.

25.6 The French libraries

The large research group LIP,[7] at l'École Nationale Supérieure in Lyon, France, has made extensive progress both on the problem of implementing correctly rounded functions, and on machine-assisted correctness proofs of algorithms and arithmetic. Their many publications and reports are covered in an online bibliography of floating-point arithmetic[8] maintained by this author.

Zimmermann, Revol, and Pélissier [ZRP05] released the mpcheck package in 2005[9] for testing binary floating-point mathematical libraries for correct rounding, output range, monotonicity, and symmetry. Version 1.1.0 supports only the IEEE 754 64-bit format, but work is in progress to extend it to the 32-, 80-, and 128-bit formats.

A collaborative effort by French research groups has produced another library, MPFR, the *Multiple Precision Floating-Point Reliable Library* [MPFR04]. MPFR extends the GMP library to provide multiple-precision floating-point computations with correct rounding.

Yet another library from French and Danish groups, MPC,[10] supplies complex arithmetic with correct rounding [ETZ09], and builds on the MPFR and GMP libraries. The three libraries are used in recent versions of the GNU

[6] See `https://www.musl-libc.org/`.

[7] See `http://www.ens-lyon.fr/LIP/`.

[8] Available at `http://www.math.utah.edu/pub/tex/bib/index-table-f.html#fparith`.

[9] Available at `http://www.loria.fr/~zimmerma/mpcheck/`.

[10] See `http://www.multiprecision.org/index.php?prog=mpc`.

gcc family to improve the compile-time conversion of floating-point numbers from decimal to binary. At the time of writing this, the MPC library provides dozens of low-level primitives for arithmetic, and another two dozen routines for elementary functions and complex operations, including `acos()`, `acosh()`, `arg()`, `asin()`, `asinh()`, `atan()`, `atanh()`, `conj()`, `cos()`, `cosh()`, `exp()`, `imag()`, `log()`, `norm()`, `pow()`, `proj()`, `real()`, `sin()`, `sinh()`, `sqr()`, `sqrt()`, `strtoc()`, `tan()`, `tanh()`, and `urandom()`, each name prefixed with `mpc_`.

The MPFR library provides a subset of those functions for real arithmetic, plus additional trigonometric and hyperbolic functions, cube root, reciprocal square root, error functions, gamma and log-gamma functions, the ordinary Bessel functions for $J_n(x)$ and $Y_n(x)$, and the Riemann zeta function. The repertoire is large enough that a wrapper library could provide all of the elementary functions required by most standard programming languages.

In another important project, a French team has built the MPFI library[11] for multiple-precision interval arithmetic.

25.7 The NIST effort

The DLMF Project (*Digital Library of Mathematical Functions*)[12] [Loz03] at the US National Institute of Standards and Technology has the ambitious goal of providing accurate implementations of all of the functions defined in the famous *Handbook of Mathematical Functions* [AS64], as well as providing a complete rewrite of that book, in printed form, and as a free electronic digital library [OLBC10]. That new edition includes pointers to software implementations of many of the functions, but there is no single software library for them with a uniform design.

25.8 Commercial mathematical libraries

The commercial *Numerical Algorithms Group* NAG library [Cow84, Chapter 14] [Phi86] has been available for more than four decades.[13] The Visual Numerics *IMSL Numerical Library*[14] is another long-lived commercial library [Cow84, Chapter 10]. Both libraries have a reputation for high quality and reliability, and library variants may be available for Fortran, C, C++, C#, and Java. However, in most cases, source code and algorithmic details may not be supplied, the libraries are supported only on the most popular platforms, and are only usable during their license period. They are of no use to other software that needs to run on less-common platforms, or is written in an unsupported language, or is made available under other kinds of licenses, or requires floating-point support beyond the single- and double-precision arithmetic supplied by those libraries. There is also the real danger of software that uses them being marooned when the vendor drops library support for a platform that is still in use at customer sites.

25.9 Mathematical libraries for decimal arithmetic

With the exception of the Cody/Waite book, and the mathcw library described in this book, there is little to be found in existing libraries for decimal floating-point arithmetic. Although we have shown in this book that, in most cases, similar, or identical, algorithms can be used for both binary and decimal arithmetic, the coding practices of most existing libraries make it unlikely that they can be quickly extended to support decimal arithmetic. For example, Intel compiler developers chose to provide decimal-function wrappers that call the 80-bit binary routines [Har09a], converting from decimal to binary on entry, and from binary to decimal on exit. That can only be a temporary solution because it does not cover either the higher precision of the 128-bit decimal format, or its wider exponent range.

25.10 Mathematical library research publications

The journal *ACM Transactions on Mathematical Software*,[15] informally known as *TOMS*, continues publication of ACM algorithms after they moved from the society's lead journal *Communications of the ACM* in 1974. TOMS contains

[11] See `https://gforge.inria.fr/projects/mpfi/`.

[12] See `http://dlmf.nist.gov/`.

[13] See `http://www.nag.com/`.

[14] See `http://www.imsl.com/`.

[15] For a complete hypertext-linked bibliography, see `http://www.math.utah.edu/pub/tex/bib/index-table-t.html#toms`.

numerous articles on, and often with source code for, the computation of elementary and special functions. Notable among those articles are several by Ping Tak Peter Tang on table-driven algorithms that use many different polynomial approximations in small intervals to reduce the operation count, and increase speed and accuracy. TOMS also contains articles on the computation of the elementary functions in complex arithmetic, in interval arithmetic (a computational technique where all values are represented by pairs of numbers providing exact lower and upper bounds), and in multiple-precision arithmetic.

A small number of researchers have worked on the difficult problem of providing *correctly rounded* elementary functions. Here are some of the highlights of that work published in several TOMS articles:

- The first work on correctly rounded elementary functions may be that of Hull and Abrham for the square-root function in 1985 [HA85] and the exponential in 1986 [HA86].
- It was followed in 1991 by work by Gal and Bachelis [GB91] at the IBM Israel Scientific Center on an entire elementary-function library that produces correctly rounded results for almost all arguments. Later that year at the same laboratory, Ziv extended his colleagues' work to always correctly rounded results [Ziv91].
- Also in 1991, Smith produced almost-always correctly rounded elementary functions for multiple-precision arithmetic [Smi91].
- In 1998, Smith extended his own work to multiple-precision complex arithmetic [Smi98].
- In 2001, Verdonk, Cuty and Verschaeren published software for compliance testing of IEEE 754 (binary) and 854 (decimal) arithmetic [IEEE85a, ANS87, VCV01a, VCV01b].
- That same year, the IBM Israel researchers made their *Accurate Portable Mathematical Library*, `APMathLib`, freely available on the Web [ZOHR01]. It supplies just the `double` versions of `acos()`, `asin()`, `atan()`, `atan2()`, `cos()`, `cot()`, `exp()`, `exp2()`, `log()`, `log2()`, `pow()`, `remainder()`, `sin()`, `sqrt()`, and `tan()`, and contains 10 730 lines of code. Unfortunately, the current release builds only on a few platforms, but that can probably be radically improved by packaging changes.

25.11 Books on computing mathematical functions

The earliest important books about computation of mathematical functions are *Approximations for Digital Computers* [Has55] and *Computer Approximations* [HCL$^+$68]. The first predated high-level programming languages and standardized mathematical libraries, and because of the tiny memory sizes of early computers, emphasized compactness over accuracy. The second was written by a team of expert numerical analysts, and it sketched algorithms, but did not implement them in any programming language.

The lead author of the books *The Standard C Library* [Pla92] and *The C++ Standard Template Library* [PSLM00] was intimately involved with development of the UNIX operating system, commercial compilers, and the ISO C and C++ language standardization efforts. The books show a lot of software, and the code is available is under license. The floating-point functions are written exclusively for the IEEE 754 64-bit format, and the entire mathematical library portion is squeezed into about 50 pages, including excerpts from the 1990 ISO Standard for C [C90].

The *C Mathematical Function Handbook* [Bak92] is an early attempt to implement much of the function repertoire of the NBS *Handbook of Mathematical Functions*. The code is licensed with the book, but fails to address many important aspects of floating-point computation, accuracy, and software portability.

There are two books on the computation of elementary and special functions that may be helpful for supplemental reading [ZJ96, Tho97], but *not* for a source of computer code. Their computational accuracy is poor, and they do not address features of IEEE 754 arithmetic that must be considered in reliable numerical software on modern systems.

More limited books include *Numerical Methods for Special Functions* [GST07] and the *Handbook of Continued Fractions for Special Functions* [CPV$^+$08]. Both books concentrate on the mathematics, and algorithms are sketched only briefly, without software implementations. There is, however, an extensive Maple package of routines for evaluation of special functions using continued fractions [BC09].

There are several other books on elementary and special functions that we record here for further reading, from oldest to newest [Rai60, Bel68, Luk69a, Luk69b, Olv74, Car77, Luk77, WG89, Tem96, Mul97, And98, AAR99, Mar00, Bel04, MvA06, Mul06, CKT07, Bry08, MH08, HBF09, BW10, Mul16].

25.12 Summary

In this chapter, we surveyed some of the important prior work on computing mathematical functions, especially where the software for doing so is freely or commercially available.

Perhaps the most encouraging aspect of recent research publications in that area is the development of algorithms in IEEE 754 arithmetic that are provably correctly rounded. At present, such proofs must generally be accompanied by an exhaustive search for the worst cases for correct rounding. Some of the research teams have pushed that work even further, developing formal proofs of algorithm correctness that can be automatically checked by theorem-proving systems. That is not straightforward, and some of the proofs reported are much longer than the algorithms themselves, and require great care on the part of experts in computer-based theorem proving. Proofs of correctness are intellectually satisfying, but are in practice insufficient, because they must assume the correctness of a lot of other software and hardware over which the user has little or no control. Software testing with real compilers on real (or virtual!) computers therefore remains indispensable.

26 Floating-point output

> Input and output for this machine are expressed in standard teletype code, with a coded symbol for the operation required.
>
> — Douglas R. Hartree
> *Calculating Instruments & Machines* (1950).

Conversions of numerical values to string representations, and the reverse, are surprisingly complex and difficult. Older programming languages provide limited support, such as Fortran's `FORMAT` statement specifier `I`*w* for integer conversions, and `E`*w*`.`*d*, `F`*w*`.`*d*, and `G`*w*`.`*d* for floating-point conversions. Pascal provides a subset of those capabilities with modifiers on output variables, such as `writeln(x:10:5)`, to produce an output string of the form `␣␣␣3.14159`.

The C language also follows the Fortran model, offering format specifiers `%`*w*`.`*d*`d` and `%`*w*`.`*d*`i` for signed integers, `%`*w*`.`*d*`u`, `%`*w*`.`*d*`o`, and `%`*w*`.`*d*`x` for unsigned decimal, octal, and hexadecimal integers, and `%`*w*`.`*d*`f`, `%`*w*`.`*d*`e`, and `%`*w*`.`*d*`g` for floating-point values.

In Fortran and Pascal, the format specifiers are part of the language definition, and thus, the compiler can match variables in the I/O statement with format specifiers, and generate code to call an appropriate library routine for the conversion: different routines are needed for each data type.

In C, input and output are *not* part of the language, but instead, are handled by the run-time library. That means that the programmer needs to supply format specifiers of the correct type, and the library conversion routines recognize additional format modifier letters, such as `L` for `long double`, and `h` for `short int`. Although there are modifier letters to support `float` in input formats, there are none for output formats, so `float` data must be cast to `double` for output; C compilers do that automatically for functions with variable numbers of arguments, such as `printf()`. Although that might seem innocuous, we shall see that the unwanted type conversion can be a problem.

The standard Fortran format specifiers provide no mechanisms for filling floating-point output fields with zeros instead of blanks, or for left-justifying or centering the string in the output field. If a nonzero field width is insufficient to hold the output string, the language requires that the user be punished by destroying the output field and overwriting it with asterisks. Fortran 90 allows the field width to be zero, in which case the system picks a width that is large enough. Later languages generally view the field width as a *minimum* width, and expand it if needed.

Format specifiers in C support additional flag characters that request left-justification, leading-zero filling, suppression of a decimal point when no fractional digits follow, and so on.

Numerical programs often need to be able to specify input numbers precisely. Most computers used for numerical work have a nondecimal floating-point base, and the accuracy of conversion from decimal strings is almost never specified by programming-language standards, so decimal representations are unreliable. If you want the constant 2^{-1022} (the smallest IEEE 754 64-bit normal value) in your program, then you really do not want to have to express it as `2.2250738585072013830902`$\cdots$`e-308`, nor can you write it in exact rational form without using a number of 308 digits, nor can you rely on a library power routine, `pow(2.0,-1022.0)`, or the Fortran operator form, `2.0d0**(-1022)`, to compute it precisely on all systems. It is even worse to express it in native floating-point form, possibly as integer values in decimal or hexadecimal in a `union` with floating-point values, because that depends on the platform and its byte order.

The C99 Standard introduced a convenient notation for hexadecimal floating-point constants, by extending the long-standing hexadecimal integer constant syntax with a power of two: `0x1p-1022` is 1×2^{-1022}. The C99 I/O library recognizes a new format item, `%`*w*`.`*d*`a`, for such conversions.

Neither the C language nor the C library provides alternatives for number bases that are not a power of two or ten. Ada has a convenient source-code notation for such cases: a base from 2 to 16 written as a decimal integer, followed by a sharp-delimited string of digits, optionally followed by the letter `E` and an exponent of the base, again written as a decimal integer. Thus, our constant can be written in Ada source code as `2#1#E-1022`, or as `4#1#E-511`, or as `16#4.0#E-256`. Unfortunately, there is limited I/O support in Ada for such values: although they can be input, there is no way to specify a base for floating-point output.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_26

The sharp character is already assigned another meaning in the C language, but the related hoc language provides a notation similar to that in Ada by replacing the sharp delimiter with the otherwise-unused at-sign, and allowing any base from 2 to 36. Our constant can then be written approximately in hoc as `36@3.4lmua2oeuvp@e-198`, or exactly as either `0x1p-1022` or `2@1@e-1022`. Based numbers like those examples can be output to files and strings using format-specifier extensions in the hoc `printf` and `sprintf` statements.

Few programming languages provide for digit grouping, despite the fact that grouping has been used in typeset numerical tables for centuries. Groups of three, four, or five digits are much easier for humans to read than a long string of digits. The Ada language supports digit grouping in program source code by permitting any pair of digits to be separated by a single underscore, but it has no provision for input and output of such values. The Eiffel language [ECM05] also allows digit-separating underscores, including repeated ones. The D language does as well [Ale10].

This author is convinced that digit grouping is of sufficient importance that it needs to be universally available, both in source programs, and for input and output. Remarkably little additional code is needed to support it, which makes it even more regrettable that it was not provided at the dawn of the computing age, and incorporated in every subsequent programming language.

Until the 1978 ANSI Standard, Fortran did not separate the job of format conversion from input and output, and many older languages had similar limitations. In the `mathcw` library, format conversion is provided by a family of functions whose result is either a string or a number, without any reference to the I/O system. It is then possible to implement the higher-level `printf()` and `scanf()` functions in terms of those conversion routines.

26.1 Output character string design issues

In software that produces character-string output, there are at least four possible design choices:

- The caller provides two additional arguments: a pointer to a character string array that is 'big enough' to hold the output, and the size of that array.

  That is flexible, but burdensome on the programmer, because it affects every call to the conversion routine. Undetected buffer overflow conditions are an all-too-frequent cause of software failure, and having the caller provide working storage is poor practice in the design of reliable software.

- Dynamically allocate as much space as is required on each call, ensuring that there are no artificial limits on the size of the returned string.

  The problem with that approach is that it burdens the caller with the job of freeing the dynamic memory (recall that Standard C does not reclaim no-longer-used dynamic memory). Failure to do so results in memory leaks that can be hard to find and repair, and that can cause unexpected termination of long-running jobs. It also requires some sort of failure indication, such as a `NULL`-pointer return, in the event that sufficient memory is not available, and the caller is obligated to check for that indicator after *every* call, in the absence of an exception-handling facility like the `throw` and `catch/try` functions or statements in C++, C#, Java, and Lisp.

  Dynamic memory allocation adds substantial run-time overhead that can sometimes cost much more than the conversion operation itself.

- Allocate memory dynamically inside the conversion routine on the first call, but reuse it on subsequent calls, and reallocate it whenever it needs to grow. That is a reasonable choice, but there is still a single memory leak over which the caller has no control.

- Allocate memory in static storage that is reused on each call, and made 'big enough' for almost all practical purposes. That is the approach that we take here, and the C99 Standard suggests that choice by mandating that format conversions shall be able to handle a field width of at least 4095 characters.

With the last two choices, the caller must take care to avoid overwriting the internal buffer, such as by using it more than once in a single statement, for example, as the arguments to a member of the `printf()` function family, or by using it in a multithreaded program without a synchronization wrapper to guarantee exclusive access.

With the last choice, we can guarantee successful operation, with no possibility of an out-of-memory error at run time, so that is what we do.

26.2 Exact output conversion

Before we discuss the routines that handle floating-point output conversion in the mathcw library, it is worthwhile to investigate the arithmetic steps needed to convert a number from one base to another. Such conversions are needed for both input and output on computers when the external human-readable data are in decimal, and their forms inside the computer are in a base that is most commonly $\beta = 2$ or $\beta = 16$. Internally, we have at most t digits in base β, with an exponent range $[\mathrm{emin}, \mathrm{emax}]$. Externally, we have base B with an exponent range $[\mathrm{EMIN}, \mathrm{EMAX}]$. Even though the number of internal digits is fixed, the number of external digits may be unbounded. For example, in base 3, the value $1/3 = 0.1_3$ exactly, but in base 10, $1/3 \approx 0.333\,333\,333 \cdots$.

When B is an integral multiple of β, all base-B expansions of base-β numbers are *finite*. In particular, all binary floating-point numbers have finite decimal expansions, although the reverse is *not* true. The expansions can be long: for the IEEE 754 64-bit binary format, the smallest and largest numbers are 2^{-1074} and $2^{1024} \times (1 - 2^{-52})$, with exact decimal representations of 751 and 309 digits, respectively, ignoring power-of-ten exponents. For the 128-bit binary format, the corresponding decimal values require 4931 and 11 530 digits.

To understand the claim of the last paragraph, notice that if $B = k\beta$, with $k = 2, 3, 4, \ldots$, then any number x that is exactly representable in base β can be written in that base as $x = m/\beta^p$, where m is an integer. Substituting β by B/k shows that $x = (mk^p)/B^p$. Because the parenthesized numerator is also an integer, x is exactly representable in both bases β and B.

The output base-conversion problem can be viewed as finding the base-B digits, D_j, that satisfy

$$(d_n d_{n-1} \cdots d_2 d_1 d_0 . d_{-1} d_{-2} \cdots d_{n+1-t})_\beta = (D_N D_{N-1} \cdots D_2 D_1 D_0 . D_{-1} D_{-2} \cdots)_B$$

for a given number x. If we want to find a particular digit D_k for $k \geq 0$, we can easily do so as follows:

- Multiply x by B^{-k} to move the fractional point to just *after* the desired digit:

$$x \times B^{-k} = (D_N D_{N-1} \cdots D_k . D_{k-1} D_{k-2} \cdots D_1 D_0 D_{-1} D_{-2} \cdots)_B.$$

- Take the floor of that value to discard fractional digits:

$$\lfloor x \times B^{-k} \rfloor = (D_N D_{N-1} \cdots D_{k+1} D_k)_B.$$

- Divide that result by B, take the floor again, and multiply by B to convert the rightmost digit to a zero:

$$\lfloor (\lfloor x \times B^{-k} \rfloor / B) \rfloor \times B = (D_N D_{N-1} \cdots D_{k+1} 0)_B.$$

- Subtract the two numbers to recover the desired digit:

$$(D_N D_{N-1} \ldots D_{k+1} D_k)_B - (D_N D_{N-1} \ldots D_{k+1} 0)_B = (D_k)_B.$$

We can write those steps more compactly as

$$\begin{aligned} C &= \lfloor x \times B^{-k} \rfloor, \\ D_k &= C - B \times \lfloor C/B \rfloor. \end{aligned}$$

A little thought shows that the algorithm also works for the fractional digits, where $k < 0$. That nice result allows us to find the digits in any convenient order, and assuming that the powers of B are available in a precomputed table, each digit costs two multiplies, one add, and two floor operations.

Two obvious questions must be answered to complete the digit-conversion algorithm:

- What is the value of N that determines the index of the leading output digit?

- How many digits do we need to compute?

The first is easy to answer: $N = \lfloor \log_B(x) \rfloor$. We may not require a precise value of the logarithm, unless it happens to be close to an integer value, so it may be possible to replace it by, for example, a fast low-order polynomial approximation.

The second is harder, because it might be "user specified" (perhaps as the width or precision value of a C or Fortran format specifier), or it could be determined by the requirement "just enough digits to recover the original x on input", or it might be "as many digits as are needed to represent x *exactly*." If fewer than the exact number of output digits are produced, we also need to specify the rounding algorithm. We address those points later in this chapter.

Unfortunately, the apparent simplicity of the output digit-generation algorithm is illusory, because it requires *exact* arithmetic in the worst case when an exact output representation is required. Even when fewer digits are needed, the final subtraction that produces D_k necessarily loses $N - k$ leading digits, so we must have C and $B \times \lfloor C/B \rfloor$ accurate to at least $N - k + 1$ digits.

We saw earlier that hundreds or thousands of decimal digits could be required to represent $x \times B^k$, and thus also C and C/B, when $B = 10$. For a future 256-bit binary format, more than a million decimal digits might be needed for exact representation.

There is a special case worth pointing out, however. When B is an integral power of β, say $B = \beta^m$, such as for octal or hexadecimal output of internal binary values, then $x \times B^{-k}$, and all of the other intermediate values, are *exact* operations. The output digits D_k can then be produced exactly with ordinary floating-point arithmetic, only $\lceil t/m \rceil$ output digits can be nonzero, and $N = \lfloor \log_\beta(x)/m \rfloor = \lfloor (\text{exponent}(x) + \log_\beta(\text{significand}(x)))/m \rfloor$ can be computed cheaply.

26.3 Hexadecimal floating-point output

Although it is a new feature in the C99 language, hexadecimal floating-point output is a good place to start our discussion of format conversions, because it is conceptually the easiest, and with care, it can be done exactly on systems where the floating-point base is 2, 4, 8, or 16. On such systems, we could write a function that takes a single floating-point value, and returns an exact hexadecimal floating-point string of sufficient length to represent all of the significand bits. However, that function would soon prove limiting, because we are likely to need to control at least the width and precision, just as we do with decimal output.

26.3.1 Hexadecimal floating-point output requirements

To understand the design requirements, we quote from the specification of hexadecimal floating-point output in the C99 Standard, as modified by two subsequent technical corrigenda. Because the specification is mixed with that for decimal output and other format items, text pertaining to them has been elided:

> *Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:*
>
> - *Zero or more* flags *(in any order) that modify the meaning of the conversion specification.*
> - *An optional minimum* field width. *If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk `*` (described later) or a nonnegative decimal integer. [Footnote: Note that `0` is taken as a flag, not as the beginning of a field width.]*
> - *An optional precision that gives the … number of digits to appear after the decimal-point [sic] character for `a`, `A`, … conversions, …. The precision takes the form of a period (`.`) followed either by an asterisk `*` (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero.*
> - *An optional* length modifier *that specifies the size of the argument.*
> - *A* conversion specifier *character that specifies the type of conversion to be applied.*
>
> *The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a `-` flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.*
>
> *The flag characters and their meanings are:*
>
> `-` *The result of the conversion is left-justified within the field. (It is right-justified if the flag is not specified.)*

> `+` *The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if the flag is not specified.)*
>
> space *If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the* space *and* `+` *flags both appear, the* space *flag is ignored.*
>
> `#` *The result is converted to an "alternative form". … For* `a`, `A`, *… conversions, the result of converting a floating-point number always contains a decimal-point [sic] character, even if no digits follow it. (Normally, a decimal-point [sic] character appears in the result of these conversions only if a digit follows it.)*
>
> `0` *For …* `a`, `A` *… conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the* `0` *and* `-` *flags both appear, the* `0` *flag is ignored.*
>
> *The conversion specifiers and their meanings are:*
>
> `a`,`A` *A* `double` *argument representing a floating-point number is converted in the style* `[-]0xh.hhhhp±d`, *where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point [sic] character [Footnote: Binary implementations can choose the hexadecimal digit to the left of the decimal-point [sic] character so that subsequent digits align to nibble (4-bit) boundaries.] and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and* `FLT_RADIX` *is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and* `FLT_RADIX` *is not a power of 2, then the precision is sufficient to distinguish [Footnote: The precision p is sufficient to distinguish values of the source type if* $16^{p-1} > b^n$ *where b is* `FLT_RADIX` *and n is the number of base-b digits in the significand of the source type. A smaller p might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point [sic] character.] values of type* `double`, *except that trailing zeros may be omitted; if the precision is zero and the* `#` *flag is not specified, no decimal-point [sic] character appears. The letters* `abcdef` *are used for* `a` *conversion and the letters* `ABCDEF` *for* `A` *conversion. The* `A` *conversion specifier produces a number with* `X` *and* `P` *instead of* `x` *and* `p`. *The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.*
>
> *A double argument representing an infinity or NaN is converted in the style of an* `f` *or* `F` *conversion specifier.*
>
> *If a conversion specification is invalid, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.*
>
> *In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.*

That is already a rather complex specification, but the Standard goes on to mandate rounding behavior:

> *For* `a` *and* `A` *conversions, if* `FLT_RADIX` *is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.*
>
> *For* `a` *and* `A` *conversions, if* `FLT_RADIX` *is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.*

26.3.2 Remarks on hexadecimal floating-point output

One significant use of hexadecimal floating-point output is to reveal the exact bit sequence in a result, and it is important that it be possible to compare output from different compilers and different platforms. Thus, although the Standard permits the leading hexadecimal digit to be any value, it is unwise to use a digit other than one, unless the result is zero. Otherwise, output comparison is impractical. Compare these hexadecimal and decimal representations of π in IEEE 754 32-bit arithmetic:

```
0x1.921fb6p+1    0x3.243f6cp+0   0x6.487ed8p-1   0xc.90fdb0p-2
0.314159274e+1   3.14159274      31.4159274e-1   314.159274e-2
```

Unless you expand the hexadecimal digits to bits, you cannot easily determine that the numbers in the first row have identical values.

Regrettably, the GNU/LINUX run-time library does not guarantee a unit leading digit. Our implementation does.

The Standard leaves the representation of subnormal numbers unspecified. At the time of writing this, there are comparatively few C99 implementations to test. GNU/LINUX and HP-UX systems print leading zeros, whereas SOLARIS systems start with a leading unit digit. Thus, the smallest IEEE 64-bit subnormal prints as `0x0.0000000000001p-1022` on the former, and on the latter, as `0x1.0000000000000p-1074`.

Although it can sometimes be helpful to distinguish subnormal values from normal ones, `float` data cannot enjoy that distinction. The lack of suitable type specifiers in C output formats means that `float` must be cast, or automatically promoted, to `double`, and all `float` subnormals are normal values in wider precisions. A better approach would be to mandate a leading unit digit as the default on all platforms, and then provide an additional flag character to request properly aligned leading zeros in subnormal values. Our implementation provides both styles.

The presence of several format flag characters, and the requirement that the `+` flag overrides the *space* flag, and that the `-` flag cancels the `0` flag, adds complexity to the software.

26.3.3 Hexadecimal floating-point output-conversion code

The code for the conversion to hexadecimal floating-point is lengthy, and we have split it into three internal helper functions, and a large one that does the rest of the work. However, it would overwhelm the reader if it were displayed as a single block of code, as we normally do in this book, so instead we present smaller chunks of code in sequential order, surrounding each code block with explanatory prose.

We want to support octal and binary floating-point output in companion conversion routines, so we take care to parametrize the base dependence to minimize the differences in their code. The usual header file, and definitions of four base-dependent constants, begin the code in the file `cvtohx.h`:

```
#if !defined(CVTOHX_H)
#define CVTOHX_H

#include "cvtoh.h"

static const char   BASE_LETTER   = 'x';
static const char * digit_chars   = "0123456789abcdef";
static const int    DIGIT_BITS    = 4;
static const int    OUTPUT_BASE   = 16;
```

A simple array-indexing operation, `digit_chars[d]`, converts a digit value `d` in $[0, 15]$ to its representation as a hexadecimal character. We also need a fast reverse conversion:

```
#define DIGIT_VALUE(c)  ( (((int)'0' <= c) && (c <= (int)'9')) ? \
                          (c - (int)'0') :                       \
                          ((c - (int)'a') + 10) )
```

The code assumes that digits occupy adjacent slots of the character table, and that the first six alphabetic letters do so as well. That is true in all important current and historical character sets. An alternate definition avoids that assumption, but is somewhat slower:

```
#define DIGIT_VALUE(c)  (int)(strchr(digit_chars, c) - &digit_chars[0])
```

Our later use of the conversion operations ensures in-range indexes and arguments, so range checks are not needed here.

The critical task in output conversion is to deliver digits one at a time. For hexadecimal, octal, and binary conversion, we can exploit the fact that in bases that are powers of two, the exponent can be treated independently of the significand. The task is more complex in a decimal base, so we supply some file-global variables, and an internal initialization function to prepare for digit delivery:

```
static fp_t f_bits;   /* fraction bits: 0 or in [1/B,1) */
static int k_bits;    /* number of bits delivered so far */
static int n_zeros;   /* number of leading zero bits to deliver */

#if B == 10
static fp_t f_hi;
static fp_t f_lo;
static const fp_t F_SCALE = FP(1.) / FP_T_EPSILON;
#endif

static void
init_bits(fp_t x, int * pn_exp, int show_subnormals)
{
    /*
    ** Decompose NONNEGATIVE x [unchecked!] into
    **
    **    x = 0.0 * B**0                // when x == 0
    **    x = f_bits * 2**n_exp         // when x != 0
    **
    ** where f_bits is in [1/B, 1).
    **
    ** If B is of the form 2**k (B = 2, 4, 8, 16, ...), then the
    ** decomposition is EXACT.
    **
    ** Otherwise, when B is not of the form 2**k (notably, B == 10),
    ** the decomposition is NOT exact, but we attempt to do it
    ** accurately as
    **
    **     x = f * 10**n
    **       = (f * 5**n) * 2**n
    **       = g * 2**n
    **       = (g / 2**m) * 2**(n + m)
    **       = f_bits * 2**n_exp
    **
    ** by working in the next higher precision.
    **
    ** If show_subnormals is nonzero (true), then x is KNOWN [but not
    ** checked!] to be subnormal, and subnormals are to be indicated
    ** by one or more leading zero bits in the output of CVTOH().
    **
    ** Successive fraction bits can be retrieved by next_bit().
    **
    ** If B > 2, then it is possible for up to log2(B) - 1 leading
    ** bits returned by next_bit() to be zero.
    */

    int n_exp;

    k_bits = 0;
    n_exp = 0;                          /* keep optimizers happy */

    if (x == ZERO)
    {
        f_bits = ZERO;

#if B == 10
        f_hi = ZERO;
        f_lo = ZERO;
#endif

    }
    else
    {
```

```
#if B == 2

        f_bits = FREXP(x, &n_exp);      /* f_bits in [1/2,1) */

#elif B == 8

        f_bits = FREXPO(x, &n_exp);     /* f_bits in [1/8,1) */
        n_exp *= 3;                     /* convert to power of two */

#elif B == 10

        hp_t f, g, h;
        int n, m;

        f = HP_FREXP((hp_t)x, &n);      /* x = f * 10**n */
        g = f * HP_IPOW(HP(5.), n);     /* g = f * 5**n */
        m = (int)CEIL((fp_t)HP_LOG2(g));/* smallest m for g < 2**m */
        h = g / HP_IPOW(HP(2.), m);     /* h = g / 2**m */

        if (h >= ONE)
        {
            h *= HP(0.5);
            m++;
        }

        f_bits = (fp_t)h;

        if (f_bits >= ONE)
        {
            f_bits = HALF;
            m++;
        }

        n_exp = n + m;

        assert( (TENTH <= f_bits) && (f_bits < ONE) );

        f_hi = FLOOR(f_bits * F_SCALE) / F_SCALE;
        f_lo = f_bits - f_hi;

        assert(f_bits == (f_hi + f_lo));

#elif B == 16

        f_bits = FREXPH(x, &n_exp);     /* f_bits in [1/16,1) */
        n_exp *= 4;                     /* convert to power of two */

#else
#error "cvtoh() family not yet implemented for bases other than 2, 8, 10, or 16"
#endif

    }

    if (show_subnormals)
        n_zeros = EMIN + 1 - n_exp;
    else
```

```
        n_zeros = 0;

    if (pn_exp != (int *)NULL)
        *pn_exp = n_exp;
}
```

Checks for Infinity and NaN arguments in that function are not required, because we treat those special values separately.

For bases $\beta = 2, 8$, and 16, the FREXP() family members handle the exact decomposition into an exponent of the base, and a significand that is either zero, or lies in $[1/\beta, 1)$.

The decimal case is harder, and requires a base-two logarithm, and integer powers, to accomplish the reduction to a significand and a power of two. The required operations are *not* exact, so we work in the next higher precision to reduce the effect of rounding errors. A more thorough treatment would resort to multiple-precision arithmetic to guarantee correct rounding. Once an accurate value of the significand f_bits is available, we split it into the sum of a high part with $t - 1$ digits, and a one-digit low part. The assertions in the decimal block are sanity checks, and we use a few more assert calls later.

The show_subnormals flag allows us to handle subnormals with and without leading zeros, and n_zeros records the number of leading zero bits.

All subsequent digit-extraction arithmetic works on the significand stored in the file-global variable f_bits, and the limited range of that value eliminates the possibility of either underflow or overflow.

Digit extraction proceeds from left to right in order of descending digit values so that we can output digits as they are produced. That job is handled by two functions, one that returns the next bit, and another that returns the next digit. Here is the first of them:

```
static int
next_bit(void)
{
    /*
    ** Return the next bit from a cache of leading zero bits (in the
    ** event that init_bits() had a nonzero show_subnormals argument),
    ** and then from f_bits, taking bits from left to right.  If B > 2,
    ** there may be up to log2(B) - 1 leading zero bits, in addition to
    ** any cached zero bits.
    */

    int bit;

    if (n_zeros > 0)                    /* take next bit from cache */
    {
        bit = 0;
        n_zeros--;
    }
    else
    {

#if B == 2

        f_bits += f_bits;               /* exact */
        bit = (f_bits >= ONE);

        if (bit)
            f_bits--;                   /* exact */

#elif B == 8

        static fp_t g_bits = FP(0.);
        if ((k_bits % 3) == 0)
        {
            f_bits *= FP(8.);            /* exact */
            g_bits = ((fp_t)(int)f_bits);/* 3-bit value in [1,8) */
            f_bits -= g_bits;            /* f_bits in [1/8,1) */
            g_bits *= FP(0.125);         /* 3-bit value in [1/8,1) */
        }

        g_bits += g_bits;                /* exact */
        bit = (g_bits >= ONE);

        if (bit)
            g_bits--;                    /* exact */

#elif B == 10

        fp_t err, sum;

        f_hi += f_hi;                    /* exact */
        f_lo += f_lo;                    /* possibly inexact */

        sum = f_hi + f_lo;
        err = ERRSUM(sum, f_hi, f_lo);

        bit = (sum >= ONE);

        if (bit)
        {
            f_hi = sum - ONE;            /* exact */
            f_lo = err;                  /* possibly inexact */
        }

#elif B == 16

        static fp_t g_bits = FP(0.);

        if ((k_bits % 4) == 0)
        {
            f_bits *= FP(16.);           /* exact */
            g_bits = ((fp_t)(int)f_bits);/* 4-bit value in [1,16) */
            f_bits -= g_bits;            /* f_bits in [1/16,1) */
            g_bits *= FP(0.0625);        /* 4-bit value in [1/16,1) */
        }

        g_bits += g_bits;                /* exact */
        bit = (g_bits >= ONE);

        if (bit)
            g_bits--;                    /* exact */

#else
#error "cvtoh() family not yet implemented for bases other than 2, 8, 10, or 16"
#endif

    }

    k_bits++;
    assert( (0 <= bit) && (bit <= 1) );
    assert(f_bits >= ZERO);

    return (bit);
}
```

For a subnormal, there may be leading zero bits in the cache; if so, we produce a zero bit, and reduce the count `n_zeros`. Otherwise, we have to get the next bit by arithmetic operations.

Bit extraction is easiest in a binary base: double the reduced significand to get its leading bit, and then reduce it by one if we got a 1-bit. All operations are exact.

For octal and hexadecimal bases, more care is required because doubling is no longer an exact operation. Instead, we count the bits delivered, and every third or fourth call, do an exact scaling by the base. We then reduce the integer value in `g_bits` to a fraction by exact scaling by the reciprocal of the base, and use that variable, instead of `f_bits`, to deliver bits. Because `g_bits` has at most four nonzero bits, its subsequent doubling is an exact operation.

In a decimal base, we operate as we did with a binary base, except that `f_bits` is represented as a sum of `f_hi` and `f_lo` so that doubling of the high part is always exact. Doubling of the low part is exact until it has t digits with a leading digit in $[5, 9]$. Because the low part began with only a single digit, we have $t - 1$ additional digits to hide the effects of rounding, so as long as we do not request an excessive number of output bits, we expect accurate conversion to a bit sequence. In **Section 26.6** on page 851, we discuss the number of bits needed to represent a decimal value sufficiently accurately to allow exact recovery of that value from its bit string.

Once a bit has been extracted, we count it in `k_bits`, and issue two final assertions to ensure that sanity still reigns.

With bit extraction operational, we can now easily collect a hexadecimal digit:

```
static int
next_digit(void)
{   /* return next digit in base OUTPUT_BASE */
    int digit;

    digit = next_bit();
    digit += digit + next_bit();
    digit += digit + next_bit();
    digit += digit + next_bit();

    assert( (0 <= digit) && (digit < OUTPUT_BASE) );

    return (digit);
}
```

The arithmetic is simple and exact in all bases. The final assertion guarantees an in-range digit for later calls to `DIGIT_VALUE()`. Although the function delivers a hexadecimal digit, we name it generically to minimize code differences from the octal and binary companions.

We are now ready for the public function. Its name, `CVTOH()`, stands for several precision-dependent versions of the code: `cvtohf()`, `cvtoh()`, `cvtohl()`, and so on. The name `CVTOH()` is an acronym for ***C**on**V**er**T** for **O**utput in **H**exadecimal*. The code begins like this:

```
const char *
CVTOH(fp_t x, int min_width, int p_digits, int e_digits, int g_digits, int flags)
{   /* convert native floating-point to output in hexadecimal */
```

The first argument, `x`, is the floating-point value to be converted. The remaining arguments are all integers.

The argument `min_width` determines the minimum output width. In many applications, its value is zero, ensuring that no whitespace surrounds the returned value. Otherwise, padding spaces or zeros are supplied, according to options supplied in the final argument. A negative value for `min_width` is converted internally to a zero value.

The argument `p_digits` is the output precision: the number of digits following the base point. There is always one digit before that point. A zero value for `p_digits` means that there is only one significant digit in the output. A negative value is given a special meaning: its magnitude is ignored, and it is converted internally to the number of

Table 26.1: Symbolic flags for formatted output. The alternate names relate them to format flags for the C `printf()` function family.

Recommended name	Alternate name	Description
CVTO_NONE	none	placeholder for no-flags-set
CVTO_FILL_WITH_ZERO	CVTO_FLAG_ZERO	fill with zeros
CVTO_JUSTIFY_CENTER	CVTO_FLAG_EQUALS	center justify
CVTO_JUSTIFY_LEFT	CVTO_FLAG_MINUS	left justify
CVTO_JUSTIFY_RIGHT	none	right justify
CVTO_SHOW_EXACT_ZERO	none	exact zero is single digit
CVTO_SHOW_MIXEDCASE	none	Infinity and NaN in mixed case
CVTO_SHOW_PLUS	CVTO_FLAG_PLUS	always show sign
CVTO_SHOW_PLUS_AS_SPACE	CVTO_FLAG_SPACE	use space for plus
CVTO_SHOW_POINT	CVTO_FLAG_SHARP	always show point
CVTO_SHOW_SUBNORMAL	none	subnormals have leading zeros
CVTO_SHOW_UPPERCASE	none	all letters are uppercase
CVTO_TRIM_TRAILING_ZEROS	none	drop trailing fractional zeros

digits, excluding the leading digit, needed for exact round-trip conversion [Gol67, Mat68a, Mat68b]. We discuss that issue later in Section 26.6 on page 851. Most applications of the `CVTOH()` family should supply a negative value for `p_digits`.

The argument `e_digits` defines the minimum output exponent width. If needed, our code supplies leading zeros to meet that width. A negative or zero value ensures an exponent of minimal width. The exponent of a zero value for `x` is always reported as zero, even though other values are possible in some historical formats, and in IEEE 754-2008 decimal floating-point arithmetic.

The argument `g_digits` argument specifies the number of digits in a group in the significand and exponent. Underscores separate groups, counting away from the base point of the significand, and from the right of the exponent. A zero or negative value for `g_digits` suppresses digit grouping.

The final argument, `flags`, is the bitwise OR of symbolic flags summarized in Table 26.1. The flag names are defined in the header file `cvtocw.h`, which also provides prototypes for all of the output functions. For languages that lack the bitwise-OR operation, the argument can be constructed as the integer sum of flag values, as long as no flag is repeated. The names are intentionally verbose and descriptive, but their use is expected to be rare. To make their relationship to the `printf()` format flags clearer, some have synonyms shown in the table.

The `CVTO_SHOW_SUBNORMAL` flag is an extension that requests that subnormals be shown with leading zero digits, and a fixed exponent corresponding to that of the *smallest normal number*. Otherwise, all nonzero values begin with a leading digit of one. There is no corresponding standard format flag.

The flags are accessed with two private macros defined in `cvtoh.h`:

```
#define CLEAR(name)     flags &= ~(name)
#define IS_SET(name)    (flags & (name))
```

The `CLEAR()` and `IS_SET()` macros are wrappers to clear a flag bit, and test whether a flag bit is set. For simplicity, they hide the name of the `flags` argument.

In summary, the call `cvtoh(x,0,-1,0,0,0)` produces exactly what the `%a` format does, and the call `cvtoh(x,0,-1,0,0,CVTO_SHOW_UPPERCASE)` matches the output from the `%A` format. A call `cvtoh(x,0,-1,4,0,0)`, with `x` having the closest value to π, produces output like `0x1.921f_b544_42d1_8p+1`.

The next code chunk declares local variables, and internal buffer space for the returned string:

```
    char *retbuf;
    char *s;
    char sign;
    fp_t f_in;
    int lenbuf, n_exp, n_exp_in, pass, positive, show_subnormals;
    static char bigbuf[MAXBIGBUF];
    static char buf[MAXBUF];
```

There are two working buffers for the output string. The buffers are declared with the `static` attribute so that they have fixed memory locations, and are preserved across calls. The first, `buf`, has a size that is easily predictable from the number of bits in the floating-point storage format. The second is required only if the specified field width is larger than that needed for the converted number; its much larger size is determined by the C99 specification. The variable `retbuf` holds a pointer to one of those buffers for the function return value.

In order to preserve correct output justification, when a plus sign is to be shown as a space, we instead temporarily employ an otherwise-unused character for the job. That character is later replaced by a space after the output string has been justified.

```
static const char PLUS_AS_SPACE = '?';
```

The first task in the code is to limit the value of `e_digits` to a sensible range:

```
if (e_digits < 0)
    e_digits = 0;           /* default: minimal-width exponent */
else if (e_digits > MAXEXPWIDTH)
    e_digits = MAXEXPWIDTH;
```

The next job is to record the sign of the argument, `x`, to be converted. Because the value of that argument can be a NaN or a negative zero, we cannot just compare it with zero to determine the sign. Instead, the `SIGNBIT()` function is the essential tool for safely extracting the sign. There are four possibilities, two of them affected by the flags that must be tested in the order mandated by the C99 Standard:

```
positive = 1;

if (SIGNBIT(x))
{
    sign = '-';
    x = COPYSIGN(x, ONE);
    positive = 0;
}
else if (IS_SET(CVTO_SHOW_PLUS))
    sign = '+';
else if (IS_SET(CVTO_SHOW_PLUS_AS_SPACE))
    sign = PLUS_AS_SPACE;
else
    sign = '\0';
```

In order to properly handle rounding in the output string, we need a two-pass algorithm. On the first pass, we compute an extra digit that is not output, but is used to determine the rounding for the second pass. The second pass is skipped if no rounding adjustment is needed.

The most common use of the conversion routine is to duplicate the action of the `%a` format: it represents all bits exactly, and therefore requires no rounding, and just a single pass.

```
for (pass = 1; pass <= 2; ++pass)
{
    s = &buf[0];

    if (sign != '\0')
        *s++ = sign;
```

The variable `s` points to the next available position in the output string. The notation `*s++` is a common idiom in the C-language family for storing a character, and then advancing the pointer to the next available output position. However, we have to guarantee that the pointer does not advance outside the buffer into which it points. We can do that either by precomputing the maximal buffer size, or by adding bounds checks on the pointer. We proceed with caution, and do both in the code that follows.

The next task is to handle the special cases of Infinity and NaN:

```
        if (ISINF(x))
        {
            (void)strlcpy(s, CVTOI(x, IS_SET(CVTO_SHOW_UPPERCASE) ? CVTO_SHOW_UPPERCASE : CVTO_NONE),
                          sizeof(buf) - 1);
            CLEAR(CVTO_FILL_WITH_ZERO);
            retbuf = &buf[0];
            break;
        }
        else if (ISNAN(x))
        {
            (void)strlcpy(s, CVTON(x, IS_SET(CVTO_SHOW_UPPERCASE) ? CVTO_SHOW_UPPERCASE : CVTO_NONE),
                          sizeof(buf) - 1);
            CLEAR(CVTO_FILL_WITH_ZERO);
            retbuf = &buf[0];
            break;
        }
```

The standard `strcpy()` function is deprecated because it cannot prevent buffer overruns. We instead use the safe OPENBSD-style variant, `strlcpy()`. The details of how Infinity and NaN are converted are hidden in two other output-conversion family members, `CVTOI()` and `CVTON()`, described in **Section 26.9** on page 866 and **Section 26.10** on page 866. We clear the zero-fill flag, because it does not apply to Infinity and NaN values. The `s` pointer does not need to be adjusted, because the job is complete after the string copy, and the `break` statements skip the unneeded second pass.

This author would prefer to return the mixed-case values `"Infinity"` and `"NaN"` for those string results, but the C99 Standard does not permit that. In its description of the `f` and `F` format items, it says:

> *A `double` argument representing an infinity is converted in one of the styles `[-]inf` or `[-]infinity` — which style is implementation-defined. A `double` argument representing a NaN is converted in one of the styles `[-]nan` or `[-]nan(`n-char-sequence`)` — which style, and the meaning of any* n-char-sequence*, is implementation-defined. The `F` conversion specifier produces `INF`, `INFINITY`, or `NAN` instead of `inf`, `infinity`, or `nan`, respectively. [Footnote: When applied to infinite and NaN values, the `-`, `+`, and* space *flag characters have their usual meaning; the `#` and `0` flag characters have no effect.]*

Nevertheless, our code provides a flag option to get mixed-case values; see **Table 26.1** on page 840.

The specification does not distinguish between quiet and signaling NaNs, except possibly via the *n-char-sequence*, but that feature cannot be used in code that is intended to be portable

The real work of the conversion to a hexadecimal floating-point string happens in the long `else` block that comes next. It begins with a few variable declarations:

```
        else                            /* finite x */
        {
            fp_t r;
            int n_digits, n_drop, n_out;
            unsigned int digit;
```

Special handling of subnormals is only needed when the value to be converted is known to be a subnormal:

```
            show_subnormals = (B == 2) && IS_SET(CVTO_SHOW_SUBNORMAL) && ISSUBNORMAL(x);
```

We further restrict that handling to binary arithmetic, even though subnormals are available in decimal arithmetic. That is an arbitrary design decision, and could be changed in a future version of the code.

Next, we decompose the value into a significand and an exponent of two, and save those results for reconstructing the value to be converted in the second pass:

```
            init_bits(x, &n_exp, show_subnormals);
            f_in = f_bits;
            n_exp_in = n_exp;
```

A few more initializations are needed before we can begin digit collection:

```
digit = 0;
n_drop = 0;
n_out = 0;
r = HALF;
*s++ = ’0’;
*s++ = BASE_LETTER;
```

The variable `r` records a half unit in the last place, and is updated each time a digit is output. We need it later to handle rounding.

We are now ready to output the first digit, and there are three cases to handle.

For a zero value, we output the first digit. If exact zeros are requested, we output an optional base point, and a rounding digit that is later removed, and then exit the two-pass loop:

```
if (x == ZERO)
{
    *s++ = digit_chars[next_digit()];

    if (IS_SET(CVTO_SHOW_EXACT_ZERO))
    {
        if (IS_SET(CVTO_SHOW_POINT))
            *s++ = ’.’;

        *s++ = ’0’;         /* rounding digit */
        break;
    }
}
```

The `CVTO_SHOW_EXACT_ZERO` flag provides a useful service that some Fortran implementations offer. In large tables of numbers with decimal formatting, it can be helpful for exact zeros to be represented as `0.0`, rather than as a string of fractional zero digits. The corresponding form here is thus `0x0p+0`, or `0x0.p+0` if a base point is required.

For a subnormal value, we output the first bit, rather than the first hexadecimal digit:

```
else if (show_subnormals)
{
    *s++ = digit_chars[next_bit()];
    r *= HALF;
}
```

Otherwise, the value is nonzero, so we discard leading zero bits and output the first nonzero bit:

```
else
{
    while (next_bit() == 0) /* 1 to 4 times */
    {
        r *= HALF;
        n_exp--;
        n_drop++;
    }

    *s++ = ’1’;
}
```

Next, we output any required base point, and compute the number of remaining output digits required:

```
if (IS_SET(CVTO_SHOW_POINT) || (p_digits != 0))
    *s++ = ’.’;

n_digits = (TBITS - k_bits + n_drop + DIGIT_BITS - 1) /
           DIGIT_BITS;
```

```
            if ( (0 <= p_digits) && (p_digits < (n_digits + MAXEXTRAPREC)) )
                n_digits = p_digits;

            n_digits++;         /* need rounding digit */
            assert(n_digits > 0);
```

Here, TBITS is a value set in the header file cvtoh.h to the number of bits in the significand. For a decimal base, it is determined by a formula [Gol67] that we discuss later in **Section 26.6** on page 851.

If a nonnegative p_digits value is specified, and is smaller than the number of digits needed for exact representation, we reduce the digit count. We accept precisions larger than are significant, but only up to MAXEXTRAPREC additional digits; that value is included in the definition of MAXBUF.

Output of the remaining digits, and any requested digit separators, is now straightforward:

```
            while (n_digits-- > 0)
            {
                r /= (fp_t)OUTPUT_BASE;
                digit = (unsigned int)next_digit();
                *s++ = digit_chars[digit];
                n_out++;

                if ( (g_digits > 0) && (n_digits > 0) && ((n_out % g_digits) == 0) )
                    *s++ = '_';
            }
```

Digit grouping takes only *three* extra lines of code: we count the number of digits output with n_out, and if grouping is requested, output an underscore when the count is a multiple of the group size and at least one more digit remains to be output in the next iteration. As we said earlier, digit grouping should be universally available in all programming languages: its implementation is nearly trivial, and its cost is negligible compared to the rest of the conversion code.

A candidate digit string with one more than the required number of fractional digits has just been produced, and we now have the difficult task of rounding in preparation for the second pass. We begin by handling the common, and easiest case, where all bits have been used:

```
            if (pass == 1)              /* check for rounding */
            {
                fp_t adjust;
                int mode;

                if ( (digit == 0) && (f_bits == ZERO) )
                    break;      /* loop exit: no rounding needed */
```

Otherwise, we determine the rounding direction dynamically (and independently of whether IEEE 754 arithmetic is in effect), and take one of four actions:

```
                mode = _cvtdir();

                if (mode == CVT_FE_UPWARD)
                    adjust = positive ? (r + r) : ZERO;
                else if (mode == CVT_FE_DOWNWARD)
                    adjust = positive ? ZERO : (r + r);
                else if (mode == CVT_FE_TOWARDZERO)
                    break;              /* loop exit */
                else                    /* mode == CVT_FE_TONEAREST */
                {
                    int prev_digit;

                    if (digit == 0)
                        break;   /* loop exit: no rounding needed */
                    prev_digit = (int)(((s[-2] == '.') || (s[-2] == '_')) ? s[-3] : s[-2]);
                    prev_digit = DIGIT_VALUE(prev_digit);

                    assert( (0 <= prev_digit) && (prev_digit < OUTPUT_BASE) );

                    if ( (digit == (unsigned int)(OUTPUT_BASE / 2)) &&
                         (f_bits == ZERO) &&
                         ((prev_digit & 1) == 0) )
                        break;   /* loop exit: no rounding needed */

                    adjust = r;
                }
```

How the rounding mode is determined is described later in **Section 26.3.5** on page 848.

We consider only four rounding modes here, but we must note that complete implementation of the IEEE 754-2008 specification for decimal arithmetic has one more mode that rounds ties away from zero. It is omitted here because at the time of writing this, compiler support for decimal rounding is lacking.

Rounding upward requires adding one ulp, `r + r`, if the number is positive. Otherwise, we can simply truncate the digit string.

Rounding downward is the mirror image of rounding upward, but applies only to negative numbers. Because the sign is handled separately, the adjustment is positive, rather than negative.

Rounding to zero (truncation) is easiest: we drop the extra digit, and skip the second pass.

The *round-to-nearest* case is the most difficult, which is why most floating-point designs before IEEE 754 do not supply it. There are three subsidiary cases to consider:

- If the extra hexadecimal digit is less than 8, then the remainder is less than half, truncation of the collected digits produces the desired result, and we can skip the second pass.

- If the extra digit is 8, and the reduced significand is zero, then we have a halfway case, where we have to determine whether the previous digit is even or odd. If it is even, then by the *round-to-nearest-with-ties-to-even* rule, no further adjustments are needed, and we can skip the second pass.

  The extra digit is stored at `s[-1]`, and we find the value of the digit before it in `s[-2]`, unless that has a point or underscore, in which case we find it in `s[-3]`.

- We have a halfway case with an odd last digit or a nonzero reduced significand, or we have an extra digit that is larger than 8. In either case, we need to round upward with an adjustment by `r`.

It is worth noting that there is another nasty multithreading problem hidden in our rounding code. The tests for the rounding direction and application of the rounding adjustment assume that the rounding direction is constant for the duration of the rounding code. That might not be the case in a multithreaded application where threads modify the rounding direction. If the rounding mode is changed during our rounding-decision checks, correct rounding of the output string cannot be guaranteed.

As with many other members of the C run-time library, we simply have to document that the functions in the `CVTOH()` family are *not* thread safe. They can be safely used in the presence of threads only if they are treated as resources that require exclusive access that is synchronized by suitable thread primitives.

If a rounding pass is needed, we construct a new `x`, and try again:

```
                /* Reconstruct rounding-adjusted x for second pass.
                   Because r is updated by 1/OUTPUT_BASE for each
                   output digit, we need (OUTPUT_BASE/2)*r to get the
                   bit that forces the rounding. */

                f_bits = f_in + (fp_t)(OUTPUT_BASE / 2) * adjust;

#if B == 2
                x = LDEXP(f_bits, n_exp_in);
#else
```

```
                x = (fp_t)((hp_t)f_bits * HP_IPOW(HP(2.), n_exp_in));
#endif

            }                   /* end if (pass == 1) */
        }                       /* end if (finite x) */
    }                           /* end for (pass ...) loop */
```

In binary arithmetic, the needed scaling is exact, and handled by the LDEXP() family. Otherwise, we need to scale by a possibly inexact power of two, using higher precision when available. Most historical systems round by truncation, so the two-pass loop is exited on the first pass, and that inexactness is not encountered.

On exit from the two-pass loop, we have the final digit string, and all that remains is to obey any request to trim trailing zeros, supply the exponent, and then handle any required justification. We start with the zero-trimming and exponent processing:

```
    if (!ISNAN(x) && !ISINF(x))
    {
        s--;            /* discard rounding digit */

        if (s[-1] == '_')
            s--;        /* discard digit separator */

        if (IS_SET(CVTO_TRIM_TRAILING_ZEROS))
        {
            while ( (s[-1] == '0') || (s[-1] == '_') )
                s--;

            if ( (s[-1] == '.') && !IS_SET(CVTO_SHOW_POINT) )
                s--;
        }

        *s++ = 'p';

        if (show_subnormals)
            n_exp = EMIN;
        else if (x != ZERO)
            n_exp--;    /* because point FOLLOWS first nonzero bit */

        (void)strlcpy(s, _cvtits(s, sizeof(buf) - (size_t)(s - &buf[0]), n_exp, 1, 1 + e_digits, g_digits),
                      sizeof(buf) - (size_t)(s - &buf[0]));
    }
```

An internal library function, _cvits(), handles the exponent conversion without relying on any external routines. The safe string-copy function, strlcpy(), once again protects against buffer overrun.

The final block of code handles justification, and returns to the caller. The common case is a zero min_width argument, which requires no filling, so we check for that first, and avoid an unnecessary string copy.

```
    lenbuf = (int)strlen(buf);

    assert ((size_t)lenbuf < elementsof(buf));
    assert (sizeof(buf) <= sizeof(bigbuf));

    if (lenbuf >= min_width)
        retbuf = &buf[0];       /* usual case: no fill needed */
```

The zero-fill case requires more code, because we first have to locate the insertion point after the hexadecimal prefix, which we do by copying to the end of the prefix, then using two standard library routines to complete the job. Also, we must limit the amount of fill to stay inside the internal buffer.

```
    else if (IS_SET(CVTO_FILL_WITH_ZERO))
    {
        char *t;
        int n_fill;

        n_fill = min_width - lenbuf;

        if ( (lenbuf + n_fill) >= (int)(elementsof(bigbuf) - 1) )
            n_fill = (int)(elementsof(bigbuf) - 1) - lenbuf;

        s = &buf[0];
        t = &bigbuf[0];

        do
        {
            *t++ = *s++;
        }
        while (s[-1] != BASE_LETTER);

        (void)memset(t, (int)'0', (size_t)n_fill);
        t += n_fill;
        (void)strncpy(t, s, sizeof(bigbuf) - (size_t)(t - &bigbuf[0]));
        bigbuf[elementsof(bigbuf) - 1] = '\0';
        retbuf = &bigbuf[0];
    }
```

By an arbitrary design decision, zero-fill digits are *not* subject to digit grouping. We use the safe version of the string-copy library function, `strncpy()`, and because it does not supply a trailing NUL string terminator if the target string is not big enough, we make sure to store a terminator in the last slot of `bigbuf`, and then set `retbuf`.

Otherwise, the result string must be justified (to the left, right, or center) with padding spaces. That operation is needed in several output functions, so we implement it in a separate internal library function, `_cvtjus()`, whose code is straightforward, and thus not shown here.

```
    else
        retbuf = (char *)_cvtjus(bigbuf, sizeof(bigbuf), buf, min_width, flags, 0);
```

With justification complete, we can now replace any plus sign that should be a space:

```
    if (IS_SET(CVTO_SHOW_PLUS_AS_SPACE))
    {
        s = strchr(retbuf, PLUS_AS_SPACE);

        if (s != (char *)NULL)
            *s = ' ';
    }
```

The final bit of code in `CVTOH()` handles any requested uppercasing (described in **Section 26.3.4** on the following page), and then returns a pointer to the converted string:

```
    if (IS_SET(CVTO_SHOW_UPPERCASE))
        retbuf = _cvtupper(retbuf);

    return ((const char*)retbuf);
}
#endif /* !defined(CVTOHX_H) */
```

That completes our description of `CVTOH()`, a single function of about 210 nonempty lines, with another 140 lines in three private functions, that provides a capability that has been absent from almost all computers and programming languages since their invention. It is the simplest of the output-conversion routines in the `mathcw` library, but there is clearly a lot more to the job than most programmers would have expected before looking at the code.

26.3.4 Conversion to uppercase

Although the C library provides macros and functions for converting single characters between uppercase and lowercase, it has no standard functions for doing that operation on a string, so we provide a private helper routine to do the uppercasing work that we needed at the end of `CVTOH()`:

```
char *
_cvtupper(char *s)
{
    char *s_in;

    for (s_in = s; *s; ++s)
        *s = TOUPPER(*s);

    return (s_in);
}
```

For our simple needs, the uppercase conversion can be done in place, overwriting the input with the output.

The `TOUPPER()` macro is a wrapper around the standard `toupper()` function. `TOUPPER()` first checks whether its argument is a lowercase letter, because some older C implementations of `toupper()` do not work correctly for arguments that are not lowercase letters.

For the special case of hexadecimal formatting, we have one-to-one mappings, because the only characters that change case are *abcdefpx*. That avoids the complication of lettercase conversion in some languages:

- The mapping changes the number of characters, as for the German lowercase letter sharp-*s*, ß, which becomes *SS* in uppercase.
- The mapping is context dependent, such as the uppercase Greek *sigma*, Σ, which maps to ς at the end of a word, and otherwise to σ.
- The mapping is locale dependent, as in Turkish, which has both dotted and undotted forms of the Latin letter I: ı ↔ I and i ↔ İ.

26.3.5 Determining rounding direction

For both input and output conversions, we need to know how the hardware rounds on addition. Although older architectures generally supply only a single rounding mode, in systems that fully conform to IEEE 754 arithmetic, at run time, four different binary rounding modes are available, and one additional mode is defined for decimal arithmetic. IBM processors that implement decimal floating-point arithmetic include the five IEEE 754 decimal modes, plus three others.

One way to determine the rounding direction is to invoke the C99 environment inquiry function, `fegetround()`. However, the mathcw library code is designed to work on a range of architectures, including historical ones that predate IEEE 754 arithmetic.

We therefore need to determine the rounding direction by run-time experiments, which we can do by adding and subtracting a quantity smaller than the machine epsilon, and then comparing the sum or difference with the original number. That original number should be chosen away from the exponent boundaries of the floating-point system, where the machine epsilons for addition and subtraction differ. We choose the halfway value, 1.5.

In IEEE 754 arithmetic, any value not larger than half the machine epsilon would suffice for the test; we could even use the smallest representable normal number. However, most historical machines have peculiar rounding behavior. If the added value is too small, it may not affect the rounding at all, and if it is too big, it may produce unexpected rounding. The PDP-10 is an example of the latter; its architecture manual says [DEC76, page 2-36]:

> *Rounding is away from zero: if the part of the normalized answer being dropped (the low order part of the fraction) is greater than or equal in magnitude to* one half the LSB[1] *of the part being retained, the magnitude of the latter part is increased by one LSB.*

[1] LSB means *Least Significant Bit*.

To illustrate that, we make a simple experiment in the 36-bit version of hoc on TOPS-20. The machine epsilon is defined to be β^{-t+1} for a t-digit significand, and on the PDP-10, we have $\beta = 2$ and $t = 27$ (see **Table H.1** on page 948):

```
hoc36> eps = 2**(-26)

hoc36> println hexfp(1.5+eps), hexfp(1.5+eps/2), hexfp(1.5+eps/4)
       +0x1.8000004p+0 +0x1.8000004p+0 +0x1.8p+0

hoc36> println hexfp(1.5-eps), hexfp(1.5-eps/2), hexfp(1.5-eps/4)
       +0x1.7fffffcp+0 +0x1.8p+0 +0x1.8p+0
```

The surprise in the output is the result for `1.5 + eps/2`; that should be 1.5 in a *round to nearest* mode, but on the PDP-10, the sum rounds up. Adding `eps/4` produces the expected result. Because of the aberrant rounding on the PDP-10, the normal loop that we have used elsewhere in this book to compute the machine epsilon produces a result that is half the predicted value.

Consequently, we use a value $\delta = \epsilon/4$, and compute $1.5 + \delta$, $-1.5 - \delta$, and $1.5 - \delta$. The computation is encapsulated in a private helper function that returns a small integer value represented by one of four symbolic names defined in the header file `cvtdir.h`:

```
int
_cvtdir(void)
{   /* Return the current rounding direction */
    fp_t delta;
    int result;
    volatile fp_t sum1, sum2, sum3;

    delta = FP_T_EPSILON * FOURTH;

    sum1 = FP(1.5);
    STORE(&sum1);
    sum1 += delta;
    STORE(&sum1);

    sum2 = -FP(1.5);
    STORE(&sum2);
    sum2 -= delta;
    STORE(&sum2);

    sum3 = FP(1.5);
    STORE(&sum3);
    sum3 -= delta;
    STORE(&sum3);

    if (sum1 > FP(1.5))
        result = CVT_FE_UPWARD;
    else if (sum2 < -FP(1.5))
        result = CVT_FE_DOWNWARD;
    else if (sum3 < FP(1.5))
        result = CVT_FE_TOWARDZERO;
    else
        result = CVT_FE_TONEAREST;

    return (result);
}
```

The presence of long registers on some architectures complicates the test expressions for determining the rounding direction. We again need the `volatile` qualifier in variable declarations, and the `STORE()` macro, to force *run-time* *evaluation* of the expressions, rather than compile-time evaluation that is immune to changes in the rounding direction.

Comparisons of the three sums with ±1.5 allow us to identify which of the four IEEE 754 rounding directions is in effect, and should cause no harm (and behave sensibly) on older architectures.

For decimal arithmetic, the output-conversion code should be extended to identify, and handle, four additional decimal rounding modes: IEEE 754-2008 *round-ties-away-from-zero*, and the IBM extensions *round-ties-toward-zero*, *round-away-from-zero*, and *round-to-prepare-for-shorter-precision*.

26.4 Octal floating-point output

The C language has always supported octal integer constants, a legacy of the origins of UNIX on the 18-bit DEC PDP-7 and the influence of the 36-bit and 60-bit computers of the 1960s. An octal digit encodes three bits, so word sizes that are multiples of three are well suited to octal notation.

The use of octal notation was so common in the computing industry that it was carried forward from earlier PDP models with 12-bit and 18-bit word sizes to the 16-bit DEC PDP-11, the machine on which the C language was first developed, even though hexadecimal would have been more appropriate, because it is suited to word sizes that are multiples of four bits. The source-code listing of Sixth Edition UNIX is riddled with octal integer constants [Lio96].

However, until the C99 Standard introduced the hexadecimal floating-point notation, such numbers have almost always been written in decimal in computer programs and their input and output files, even though most computers represent them in binary.

Floating-point conversion to octal is similar to that for hexadecimal, and the code in file `cvtoox.h` differs from that in `cvtohx.h` by only a few lines:

```
static const char * digit_chars = "01234567";
static const char BASE_LETTER = 'o';
static const int DIGIT_BITS = 3;
static const int OUTPUT_BASE = 8;

#define DIGIT_VALUE(c)  ((int)c - (int)'0')
```

In addition, the `next_digit()` function collects only three bits. The body of `CVTOO()` is identical to that of `CVTOH()`, apart from the difference in function names.

Because there is no prior art for octal floating-point, we choose a prefix of `0o`, by analogy with the hexadecimal prefix `0x`. An uppercase form of octal constants is clearly undesirable, because the prefix characters might not be distinguishable.

The `printf()` family format letters already use `o` for octal integers, so we propose `q` and `Q` for octal floating-point.

hoc supports decimal, octal, and hexadecimal floating-point numbers like this:

```
hoc32> println PI, octfp(PI), hexfp(PI)
       3.14159274 +0o1.44417666p+1 +0x1.921fb6p+1

hoc32> printf("%g  %q  %a\n", E, E, E)
       2.71828  0o1.2677025p+1  0x1.5bf0a8p+1

hoc32> printf("%....10@  %....8@  %....16@\n", GAMMA, GAMMA, GAMMA)
       10@5.77215672@e-1  8@4.47421500@e-1  16@9.3c4680@e-1
```

Note the difference: exponents of based literals are *powers of the explicit base*, whereas those for the octal and hexadecimal forms are *powers of two*:

```
hoc32> for (k = 1; k < 13; ++k) printf("%q\t%....8@\n", 10**k, 10**k)
         0o1.2p+3       8@1.20000000@e+1
         0o1.44p+6      8@1.44000000@e+2
         0o1.75p+9      8@1.75000000@e+3
         0o1.161p+13    8@2.34200000@e+4
         0o1.4152p+16   8@3.03240000@e+5
         0o1.72044p+19  8@3.64110000@e+6
```

```
0o1.142264p+23 8@4.61132000@e+7
0o1.372741p+26 8@5.75360400@e+8
0o1.6715312p+29        8@7.34654500@e+9
0o1.12402762p+33       8@1.12402762@e+11
0o1.35103556p+36       8@1.35103556@e+12
0o1.64324512p+39       8@1.64324512@e+13
```

26.5 Binary floating-point output

Just as code for octal floating-point conversion is an easy modification of that for hexadecimal floating-point, binary is too, and we adopt a prefix of `0b`, and propose the `printf()` family format letters `b` and `B`.

hoc supports binary floating-point numbers like this:

```
hoc32> println PI, binfp(PI), hexfp(PI)
       3.14159274 0b1.10010010000111111011011p+1 +0x1.921fb6p+1

hoc32> printf("%g  %b  %a\n", E, E, E)
       2.71828  0b1.010110111111000010101p+1  0x1.5bf0a8p+1

hoc32> printf("%....10@ %....2@ %....16@\n", GAMMA, GAMMA, GAMMA)
       10@5.77215672@e-1 2@1.00100111100010001101000@e-1 16@9.3c4680@e-1
```

Long strings of bits are hard to read, but digit-group separators fix that problem:

```
hoc32> printf("%...4b\n", PI)
        0b1.1001_0010_0001_1111_1011_011p+1

hoc32> __GROUP__ = 4

hoc32> binfp(E)
        0b1.0101_1011_1111_0000_1010_1p+1
```

26.6 Decimal floating-point output

The complexity in decimal conversion arises from the handling of multiple output formats (`%e`, `%f`, `%g`, and their uppercase companions), and from additional code for scaling of the input value, determination of the maximum decimal precision, optimization of the computation of powers of ten, trimming of trailing zeros, digit grouping, and justification in a specified field width.

For decimal arithmetic, we can guarantee always-correctly rounded output, but we cannot do so for nondecimal bases without access to arithmetic of much higher precision, as we discuss in Section 27.4 on page 895. Instead, we do internal scaling and digit extraction in the highest supported precision. Alas, on some platforms, and with some compilers, that is only the C type `double`.

To illustrate the numerical difficulties, consider Table 26.2, which shows just four numbers taken from a much larger tabulation in work by Hough and Paxson [HP91] that describes how to find difficult cases for base conversion.

Fortunately, such problematic numbers are uncommon, and it may be more relevant to require only that enough decimal digits be produced that input conversion can recover the original number exactly. Goldberg [Gol67] and Matula [Mat68a, Mat68b] solved that problem long ago. Until their work, most programmers believed that all that is necessary to find the decimal precision needed for a b-bit integer is to express its maximum value, $2^b - 1$, as a decimal number, and count digits. Mathematically, the count is just $\lceil \log_{10}(2^b - 1) \rceil \approx \lceil b \log_{10}(2) \rceil$.

Matula proved that the expression is sometimes off by one: the correct output-conversion formula is $\lceil b \log_{10}(2) + 1 \rceil$. For conversion from n base-β digits, where $\beta \neq 10$, the general formula for the number of decimal digits is $\lceil n \log_{10}(\beta) + 1 \rceil$.

Goldberg derived the correct formula for input conversion from d-digit decimal to general base $\beta \neq 10$: $\lceil d \log_{\beta}(10) + 1 \rceil$ base-β digits are needed.

Table 26.2: Hard cases for binary to decimal conversion. Each binary value, shown here in product form, can be exactly represented with a 53-bit significand, corresponding to approximately 16 decimal digits, yet as many as 42 decimal digits need to be generated to determine correct rounding.

Exact binary value	Approximate decimal value
$8\,51103\,00202\,75656 \times 2^{-342}$	$9.49999\,99999\,99999\,99986\,79752\,83150\,92877\,15063\,14238\ldots \times 10^{-088}$
$8\,09145\,05872\,92794 \times 2^{-473}$	$3.31771\,01181\,60031\,08151\,84999\,99999\,99999\,99998\,17539\ldots \times 10^{-127}$
$6\,56725\,88820\,77402 \times 2^{+952}$	$2.50000\,00000\,00000\,00040\,44126\,91966\,14109\,55316\,71615\ldots \times 10^{+302}$
$8\,54949\,74112\,94502 \times 2^{-448}$	$1.17625\,78307\,28540\,37998\,95000\,00000\,00000\,00000\,26745\ldots \times 10^{-119}$

More precisely, for conversion of p digits in base α to q digits in base β, this relation must hold:

$$\alpha^p < \beta^{q-1}.$$

Taking base-2 logarithms, that is equivalent to

$$p \log_2(\alpha) < (q-1)\log_2(\beta).$$

Solving for q, we find the general Goldberg/Matula condition for exact round-trip base conversion:

$$q > 1 + (p \log_2(\alpha)/\log_2(\beta)).$$

The inequality suggests evaluation with the ceiling function, but because the logarithms can take integral values, some care is needed in translating the relation into code. One way is to compute the right-hand side, then adjust the result if it is a whole number:

```
rhs = 1.0 + (double)p * log2(alpha) / log2(beta);
q = (int)( (rhs == ceil(rhs)) ? ceil(rhs + 1.0) : ceil(rhs) );
```

Another way is to increase the right-hand side by a tiny amount, and then apply the ceiling function:

```
q = (int)ceil( (1.0 + DBL_EPSILON) * (1.0 + (double)p * log2(alpha) / log2(beta)) );
```

Of course, the input and output bases are usually known, so the ratio of logarithms can then be replaced by a precomputed numerical value.

The simpler formulas given earlier for conversion to and from decimal do not require the adjustment if we restrict β to the range $[2, 99]$.

Typical values from the Goldberg/Matula formulas for current and important historical systems (see Table H.1 on page 948) are summarized in Table 26.3 on the facing page. That table shows that 17 decimal digits suffice to recover the 53-bit values in Table 26.2. For the first of them, the input value `9.5e-88` correctly recovers the exact value `+0x1.e3cb_c990_7fdc_8p-290`.

Another important observation from the table is that we cannot convert exactly between the IEEE decimal and binary formats within the same storage size. For example, for the 32-bit decimal storage format, 7 decimal digits require 25 bits, one more than provided by the 32-bit binary format. The first and fourth pairs of table rows show that the reverse is also true: a 32-bit binary format cannot be exactly represented by a 32-bit decimal format, and similarly for the higher precisions.

We therefore take the view that, in the absence of the multiple-precision arithmetic software needed to produce decimal strings that are always correctly rounded, we should limit the output precision to the Matula predictions, and if the requested precision is larger, we then supply trailing zeros. There are several reasons for that decision:

- Novices often fail to understand the precision limitations of computer arithmetic, and may falsely assume that a computed number printed to a large number of decimal places must be correct to the last digit. For them, trailing zeros are more likely to teach them about the limitations.

- It should always be possible to output computed results, and then use those values as input to recover the original internal values exactly. Provided that the output decimal precision follows the Matula rule, exact round-trip conversion is possible.

Table 26.3: Base-conversion precisions according to the Goldberg/Matula formulas for common floating-point formats. The extended IEEE 754 precisions are shaded.

Binary in	24	27	53	56	59	62	64	106	113	237
Decimal out	9	10	17	18	19	20	21	33	36	73
Octal in	13	26								
Decimal out	13	25								
Hexadecimal in	6	14	28							
Decimal out	9	18	35							
Decimal in	7	16	34	70						
Binary out	25	55	114	234						
Decimal in	7	16	34	70						
Octal out	9	19	39	79						
Decimal in	7	16	34	70						
Hexadecimal out	7	15	30	60						

- The conversion code is more efficient when superfluous output digits can be handled quickly. That is particularly important for software implementations of decimal floating-point arithmetic, where large output precisions could otherwise be costly.
- The existence of the Goldberg/Matula formulas means that we can offer their predictions as easy-to-use defaults for the precision argument in our code. A negative precision, which is otherwise meaningless, is simply replaced by the default.

  It is regrettable that the ISO C Standards continue to follow a historical mistake in the `printf()` function family in the C library of supplying a default precision of six when the precision is omitted in output format conversions. Goldberg's formula tells us that six decimal digits correctly determine only 21 bits, a value that is smaller than the single-precision significand size of all current and historical floating-point formats.

  Future code should use the Goldberg/Matula predictions, and we do so in Section 26.8 on page 865 for output conversion in a general base, and in our extended conversion specifiers for the C `printf()` function family.

26.6.1 The decimal-conversion program interface

The outline of the mathcw library code for decimal conversion looks like this:

```
const char *
CVTOD(fp_t d, int min_width, int p_digits, int e_digits, int g_digits, int flags)
{
    /* code body shown later */
}
```

As usual, the uppercase macro name expands to any of ten variants for binary and decimal floating-point arithmetic: the same code applies to all of them.

The function converts the input number `d` to a decimal representation in an internal `static` character-string buffer, and returns a pointer to that string. Subsequent calls to the function overwrite the buffer, so in a threaded application, either a suitable locking mechanism must be used to ensure exclusive access, or more typically, only a single thread should be given the task of doing I/O.

The `min_width` argument supplies the minimum field width. Shorter output strings are padded to that length with spaces or leading zeros, according to flags described shortly. A negative or zero field width means that no padding is required.

The `p_digits` argument specifies the precision. For `%g`-style conversion, it means the total number of significand digits. For `%e`- and `%f`-style conversion, it means the number of fractional digits. As we noted earlier, our code supplies trailing zeros beyond the Matula precision.

If the precision is negative, the code uses the Matula formula to reset the internal value of `p_digits`.

Table 26.4: Additional symbolic flags for decimal formatted output, as defined in the `cvtocw.h` header file. These supplement the flags in Table 26.1.

Name	Description
`CVTO_E_STYLE`	`%e`-style conversion
`CVTO_F_STYLE`	`%f`-style conversion
`CVTO_G_STYLE`	`%g`-style conversion

If the precision is zero, for `%e`- and `%f`-style conversion, the code produces neither a decimal point (unless requested by a flag), nor any fractional digits. For `%g`-style conversion, it resets the internal precision value to one.

To get the default behavior of the C `printf()` function family for a zero precision, set `p_digits` to six, but realize that it is a poor choice.

The `e_digits` argument specifies the minimum number of digits in the exponent field. For Standard C behavior, set it to two, but recognize that modern systems require at least three digits for type `double`, four for type `long double`, and a future 256-bit decimal format needs seven.

A positive nonzero `g_digits` argument requests insertion of underscores between groups of that many digits, counting away from the decimal point in the significand, and from the right in the exponent. Values of three to five follow common practice in mathematical tables. Negative or zero values suppress digit grouping.

The final `flags` argument is the logical OR (or arithmetic sum, provided that replications are avoided) of the output conversion flags given earlier in Table 26.1 on page 840 for other conversion functions, and additional ones specific to decimal conversion, shown in Table 26.4.

The most significant settings in `flags` are `CVTO_E_STYLE`, `CVTO_F_STYLE`, and `CVTO_G_STYLE`. If more than one of them are set, the code obeys the first one found, in alphabetical order. If none of them is set, `CVTO_G_STYLE` is the default.

The code in `CVTOD()` begins with the usual local variable declarations:

```
static char jusbuf[MAXBUF]; /* return value is ptr to jusbuf[] */
static const char PLUS_AS_SPACE = '?';
char buf[sizeof(jusbuf)];
char digitbuf[sizeof(jusbuf)];
char *retbuf;
xp_t scale, x, y, z;
int f_digits, flag_e, flag_exact_zero, flag_f, flag_g, flag_plus,
    flag_sharp, flag_space, flag_trim_zeros, flag_uppercase,
    flag_zero, k, max_f_digits, max_precision, n_digits,
    n_exponent, n_zero_digits, need, q;
```

The unusual feature here is a new data type, `xp_t`. It stands for *extreme precision*: the highest available floating-point precision. Inexactness in binary-to-decimal conversion is a difficult problem, but computation in higher precision can help to make most of the errors negligible.

Next we have sanity checks and initializations:

```
/* Verify and enforce C99 and internal limits on
   format-conversion sizes */

assert(sizeof(buf) > 4095);
assert(sizeof(digitbuf) > 4095);
assert(sizeof(buf) > (size_t)(1 + FP_T_MAX_10_EXP + 2*T + 1 + 1));

/* Make local copies of flags for shorter code */

flag_exact_zero = IS_SET(CVTO_SHOW_EXACT_ZERO);
flag_plus       = IS_SET(CVTO_FLAG_PLUS);
flag_sharp      = IS_SET(CVTO_FLAG_SHARP);
flag_trim_zeros = IS_SET(CVTO_TRIM_TRAILING_ZEROS);
```

```
    flag_space      = IS_SET(CVTO_FLAG_SPACE);
    flag_uppercase  = IS_SET(CVTO_SHOW_UPPERCASE);
    flag_zero       = IS_SET(CVTO_FLAG_ZERO);

    /* Because the formatting style is selected by flag bits, we guard
       against conflicting requests by prioritizing them in
       alphabetical order (e before f before g), and fall back
       to g if none is set */

    flag_e = IS_SET(CVTO_E_STYLE);
    flag_f = flag_e ? 0 : IS_SET(CVTO_F_STYLE);
    flag_g = (flag_e || flag_f) ? 0 : IS_SET(CVTO_G_STYLE);

    if (!(flag_e || flag_f || flag_g))
        flag_g = 1;

    /* Handle defaults for digits */

    max_precision = cvt_precision();
    f_digits = p_digits;

    if (f_digits < 0) /* default: minimum for exact round-trip conv. */
        f_digits = flag_e ? (max_precision - 1) : max_precision;
                /* warning: f_digits is reset below for %g conv. */

    if (e_digits < 0)
        e_digits = 0;   /* default: minimal-width exponent */
    else if (e_digits > MAXEXPWIDTH)
        e_digits = MAXEXPWIDTH;

    if (min_width < 0)  /* default: minimal-width result */
        min_width = 0;

    z = (xp_t)ZERO;     /* prevent warnings about use before set */

    if (d < ZERO)
        x = -(xp_t)d;
    else
        x = (xp_t)d;
```

The private function `cvt_precision()` implements the Matula formula, but we delay its presentation until **Section 26.6.10** on page 864.

26.6.2 Fast powers of ten

The `CVTOD()` code requires integer powers of ten in several places, and because the power operation is relatively expensive, and possibly less accurate (in some implementations) than we would like, we need a fast way to compute them. The easiest way is by table lookup, because generally only small integer powers are needed. Because we work internally in the highest available precision, which is always at least of type `double`, the code includes three compile-time tables of powers of ten, only one of which is selected on a given system, according to values of the standard macros `DBL_MIN_10_EXP` and `DBL_MAX_10_EXP` in `<float.h>`. The tables handle the power ranges $[-308, +308]$ for IEEE 754 `double`, $[-75, +75]$ for IBM System/360 hexadecimal, and $[-38, +38]$ for the DEC PDP-10 and VAX. The range for IEEE 754 arithmetic intentionally *excludes* subnormal numbers, because their effective precision is less than that of normal numbers.

The private function `power_of_10()` (see **Section 26.6.11** on page 864) provides access to the table, and resorts to calling the general power function only for arguments outside the tabulated range. In most cases, the function finds the needed powers quickly, and they are exact to machine precision, as long as the compiler's input-conversion facilities are not defective.

An approach that we have not implemented is to economize on table storage by using several small tables of powers of ten, the first with powers $1, 2, \ldots, 9$, the second with powers $10, 20, \ldots, 90$, and so on, and then recursively construct a large power by repeated multiplication. For example, to compute the power 10^{4932} near the IEEE 754 128-bit overflow limit, decompose the exponent as $4932 = 4000 + 900 + 30 + 2$, find the four powers in the small tables, and form the product of the powers to construct the needed power. In nondecimal arithmetic, that involves seven rounding errors, four from the stored constants, and three from the multiplications. In the future 256-bit format with seven-digit exponents, there could be thirteen rounding errors. To reduce the cumulative rounding error, we would have to store the powers as sums of exact high and approximate low parts, and use pair-precision arithmetic to generate the products. The mathcw library pow() function produces better accuracy, and guarantees exact powers of the base, so for decimal arithmetic, all powers are exact.

Another possibility is to store just a small table of exactly representable powers, and a few correctly rounded higher ones that are closest to their exact mathematical values, then find requested powers at run time by either exact table lookup, or from a small number of multiplications. Because all of the possible powers are known, for a given architecture, it is possible to identify computed results that differ from correctly rounded values, and correct them. That correction can be done by supplying a compile-time constant compact array of flags that identify the erroneous values, and tell whether they must be rounded up or down. In IEEE 754 arithmetic, numerical experiments on several CPU architectures show that the computed values are never wrong by more than one ulp, so a two-bit flag suffices to identify whether the computed result should be rounded up, rounded down, or kept unchanged. That idea may have first been suggested by Jerome Coonen in his doctoral thesis at the University of California, Berkeley [Coo84, Chapter 7]. That thesis supplies the mathematical and computational rationale behind the IEEE 754 Standard.

Test programs constructed by this author, and inspired by Coonen's descriptions, show that 17 stored positive powers can replace the 39 normally needed for the 32-bit format. For the 64-bit format, just 52 stored powers and exponents, augmented by a 14-word integer array containing packed two-bit flags, allow recovery of correctly rounded values of the 309 possible positive powers. The storage economization to about 20% of a full table is unlikely to be significant on modern systems, but it can be important for older architectures with limited memory, or when the exponent range is large, as it is for the 128-bit and 256-bit formats.

A variant of the test using long double powers in both 80-bit and 128-bit IEEE 754 formats shows that the table space can be reduced to just 5% of the full table, by storing 28 exact powers, and then every 45th power, along with a compact array of two-bit rounding-adjustment flags.

A final experiment examined all possible pairs and triples of partitions of the positive integer powers, selecting the minimal sets that could be used to reconstruct all powers without error. The storage requirements are about 54% of the full table for pairs, and 40% for triples, which is inferior to Coonen's approach.

26.6.3 Preliminary scaling

The code first converts the input argument d to a positive value x of type xp_t, the highest available precision, which we use for computations inside CVTOD(). The conversion is exact because it either widens the value by adding trailing zero digits, or else preserves the original digits when the input precision is equivalent to xp_t.

The key to correct digit extraction is conversion of the input argument to a whole number from which we can accurately extract decimal digits from right to left. Let $x = f \times 10^n$ with f in $[1, 10)$. Then for %e-style conversion, $y = f \times 10^q$ has $1 + q$ decimal digits before the point, and is the floating-point number nearest the whole number that we seek. For %f-style conversion, we need $n + 1 + q$ digits in the whole number.

We determine the initial exponent with this code:

```
if (ISNAN(x))
    n_exponent = 0;
else if (ISINF(x))
    n_exponent = 0;
else if (x == ZERO)
    n_exponent = 0;
else
{
```

```
#if B == 10
        (void)XP_EXP(x, &n_exponent);
        n_exponent--;           /* adjust for f in [1,10) */
#else
        n_exponent = (int)XP_FLOOR(XP_LOG10(x));
#endif

    }
```

In decimal arithmetic, the exponent extraction is an exact operation provided by XP_FREXP(), a macro wrapper that calls the frexp() family member that corresponds to type xp_t. Its return value corresponds to a fraction f in $[1/10, 1)$, so we decrement the exponent to move f into the needed range. We discard the fractional value, as indicated by the (void) cast, because we later scale the original input value.

In nondecimal arithmetic, the floor of the base-10 logarithm recovers the exponent, and the macro wrappers XP_FLOOR() and XP_LOG10() call the corresponding function family members of type xp_t. In borderline cases, it is possible that the computed exponent is off by one; we take care of that later when we complete the reduction of the input value to a whole number.

26.6.4 Support for %g-style conversion

The %g format conversion in the printf() function family provides a general numeric conversion that is intended to be human friendly by avoiding exponential notation for numbers that are neither too big nor too small. However, it is rarely suitable for producing neatly aligned tables of numbers. Roughly, the intent is that the %f style is used when there are no more than three leading zero fractional digits for magnitudes smaller than one and for larger magnitudes, no more than p_digits digits before the point, not counting filler zeros. Trailing fractional zeros, and any trailing decimal point, are removed unless CVTO_SHOW_POINT (or its alternate name CVTO_FLAG_SHARP) is set in flags.

Here is the precise specification from Technical Corrigendum 2 of the C99 Standard (section 7.19.6.1, p. 278), which clarifies the opaque description in the original text:

> g, G *A* double *argument representing a floating-point number is converted in style* f *or* e *(or in style* F *or* E *in the case of a* G *conversion specifier), depending on the value converted and the precision. Let* P *equal the precision if nonzero,* 6 *if the precision is omitted, or* 1 *if the precision is zero. Then, if a conversion with style* E *would have an exponent of* X*:*
>
> ❏ *if* $P > X \geq -4$*, the conversion is with style* f *(or* F*) and precision* $P - (X + 1)$*.*
>
> ❏ *otherwise, the conversion is with style* e *(or* E*) and precision* $P - 1$*.*
>
> *Finally, unless the* # *flag is used, any trailing zeros are removed from the fractional portion of the result and the decimal-point character is removed if there is no fractional portion remaining.*
>
> *A* double *argument representing an infinity or NaN is converted in the style of an* f *or* F *conversion specifier.*

The translation of those requirements into code is straightforward, but as we documented earlier, the inappropriate value *six* for p_digits must be supplied explicitly if needed:

```
    if (flag_g)
    {
        if (p_digits < 0)
            p_digits = max_precision;
        else if (p_digits == 0)
            p_digits = 1;

        if ( (p_digits > n_exponent) && (n_exponent >= -4) )
        {
            flag_f = 1;
            f_digits = p_digits - (n_exponent + 1);
        }
        else
        {
            flag_e = 1;
            f_digits = p_digits - 1;
        }

        flag_g = 0;
        flag_trim_zeros = flag_sharp ? 0 : 1;
    }
```

From now on, we only need to handle `%e`- and `%f`-style conversion.

Some final adjustments are needed in case the caller passed an excessively large value of `f_digits`:

```
    max_f_digits  = flag_e ? (max_precision - 1) : max_precision;
    max_f_digits  = MIN(max_f_digits, f_digits);
    n_zero_digits = MAX(f_digits - max_f_digits, 0);
```

Henceforth, the number of computed fractional digits is `max_f_digits`, and the remaining `n_zero_digits` digits are just trailing zeros that we can assign without computation.

26.6.5 Buffer sizes

A short computation determines the buffer sizes required in the worst case:

```
    if (flag_e)
        need = 1 + (1 + 1 + f_digits) + 1 + 1 + (e_digits > 6 ? e_digits : 6) + 1;
    else
        need = 1 + (FP_T_MAX_10_EXP + f_digits) + 1 + 1;

    if (g_digits > 0)
        need *= 2;          /* assume worst case with g_digits == 1 */

    if (need > (int)elementsof(buf))
        n_zero_digits -= need - (int)elementsof(buf);

    if (min_width >= (int)elementsof(buf))
        min_width = (int)elementsof(buf) - 1;
```

26.6.6 Special cases

NaN, Infinity, and exact zeros are easy to deal with:

```
    if (ISNAN(d))
        return (_cvtjus(jusbuf, elementsof(jusbuf), CVTON(d, flags), min_width, flags, 0));
    else if (ISINF(d))
        return (_cvtjus(jusbuf, elementsof(jusbuf), CVTOI(d,flags), min_width, flags, 0));
    else if (flag_exact_zero && (d == ZERO))
    {
        (void)strlcpy(buf, (COPYSIGN(ONE, d) == -ONE) ? "-" : "+", sizeof(buf));
        (void)strlcat(buf, "0", sizeof(buf));
        (void)strlcat(buf, (flag_uppercase ? "E" : "e"), sizeof(buf));
        (void)strlcat(buf, _cvtits(jusbuf, sizeof(jusbuf), 0, 1, 1 + e_digits, 0), sizeof(buf));
        return (_cvtjus(jusbuf, elementsof(jusbuf), buf, min_width, flags, 0));
    }
```

The safe string functions, `strlcat()` and `strlcpy()`, protect against buffer overrun.

26.6.7 Scaling and rounding adjustment

The preliminary scaling determined the power of ten of the input number, so we can combine that with the requested precision to determine the final scaling. The code finds the required power of ten like this:

```
q = flag_e ? (max_f_digits - n_exponent) : max_f_digits;
```

The problem now is that if the input number is near the underflow limit, then 10^q can overflow, even though $x \times 10^q$ is finite. Fortunately, `<float.h>` defines macros whose values are the maximum representable powers of ten for each precision. We can therefore compare q with a suitable one of those values, and do the scaling in two steps if necessary. The code looks like this:

```
if (q > FP_T_MAX_10_EXP)
{
    int r, s;

    r = q / 2;
    s = q - r;
    scale = power_of_10((s < r) ? s : r);
    y = x * scale;
    y *= scale;

    if (r != s)
        y *= XP_TEN;
}
else
{
    scale = power_of_10(q);
    y = x * scale;
}
```

Notice that the stepwise scaling is done only when needed to prevent overflow, because it introduces as many as three additional rounding errors in a nondecimal base. The `power_of_10()` function is presented in **Section 26.6.11** on page 864.

For `%e` conversion, we need to ensure that we have exactly the right number of digits, and rounding errors in the determination of the exponent, and the subsequent power and scaling, can produce a final result that is slightly off. We therefore require a check and possible adjustment:

```
    if (flag_e)
    {
        z = power_of_10(max_f_digits);

        if (y >= (z * XP_TEN))
        {
            do
            {
#if B == 10
                y *= XP_TENTH;
#else
                y /= XP_TEN;
#endif
                n_exponent++;
            }
            while (y >= (z * XP_TEN));

            /* If numeric error causes y to fall below z, reset
               it to z: d must then be an exact power of 10. */
            if (y < z)
                y = z;
        }
    }
```

In a nondecimal base, we divide by an exact constant, rather than multiplying by an inexact one, to avoid introducing extra rounding errors.

The whole number nearest the scaled value is now easily determined from a new rounding function introduced in C99, and described in **Section 6.4** on page 132:

```
    y = XP_RINT(y);
```

We have not quite completed the job, however. Consider output of the value 9.9999 with a `%.3e` format. The scaling by 10^q produces the number 9999.9, and `Rint()` converts that to 10000, which would then be converted to `10.000e+00` instead of the expected `1.000e+01`. Another scaling check is therefore required:

```
    if (flag_e)
    {
        xp_t ymin, yred;

        ymin = power_of_10(max_f_digits);

#if B == 10
        yred = y * XP_TENTH;             /* exact scaling */
#else
        yred = y / XP_TEN;               /* inexact scaling */
#endif

        if (ymin <= yred) /* y rounded out of range [ymin, ymax) */
        {
            y = yred;     /* y now safely back in [ymin, ymax) */
            n_exponent++;
        }
    }
```

Once again, in a nondecimal base, we use division instead of multiplication to avoid introducing an extra rounding error.

26.6.8 Digit generation

At this point, y contains the correctly rounded whole number from which we extract digits from right to left. The extraction must be an exact operation. In the decimal case, we could do it with simple arithmetic and type conversions between decimal and integer types. However, in the nondecimal case, the only way to guarantee exactness is to use the comparatively expensive `fmod()` function family. Integer operations would be exact, but in the C language, we lack an integer type of sufficient width to handle all possible floating-point significand sizes.

An implementation consideration forces use of `fmod()` also in the decimal case, because type conversions between decimal floating-point and either binary floating-point or integer types in the GNU C library use `sprintf()`. We use `CVTOD()` in our own `printf()` family implementation, so we need to prevent circular use of those functions.

Because we work from right to left, we start at the end of the character-string buffer, and begin by storing a string terminator, and any required filler zeros:

```
    k = (int)elementsof(digitbuf);
    digitbuf[--k] = '\0';

    while (n_zero_digits-- > 0) /* zero fill beyond max_f_digits */
        digitbuf[--k] = '0';
```

We are now ready for the digit-extraction loop, which is the most expensive part of `CVTOD()`, because of the `XP_FMOD()` calls:

```
    n_digits = 0;

    while (k > 2)       /* collect digits from right to left, */
    {                   /* leaving space for sign and point */
        static const char *digits = "0123456789";
        int digit;

        digit = (int)XP_FMOD(y, XP_TEN);
        assert( (0 <= digit) && (digit < 10) );
        digitbuf[--k] = digits[digit];

#if B == 10
        y *= XP_TENTH;    /* exact scaling */
#else
        y /= XP_TEN;      /* inexact scaling */
#endif

        n_digits++;

        if (n_digits == max_f_digits)
            digitbuf[--k] = '.';
        else if ( (n_digits > max_f_digits) && (y < ONE) )
            break;
    }
```

We count the computed digits, but not the filler zeros, and when we have converted all of the required fractional digits, we store a decimal point. Otherwise, we generate at least one more digit, and exit the loop when y contains no more digits. That procedure handles both `%e`- and `%f`-style conversion.

The assertion that the computed digit is a decimal digit is important; on at least two different platforms, it exposed errors in compiler code generation for the `fmod()` function.

In the decimal case, multiplication by $1/10$ is the fastest way to reduce y, and the operation is exact.

In the nondecimal case, the factor $1/10$ is inexact, and its use would introduce an unwanted additional rounding error, so we instead use a division by the exact constant 10. That operation introduces a rounding error, but because we started with a whole number, the error affects only the fractional digits, and is unlikely to produce an incorrect low-order integer digit, especially because we are working in higher precision.

26.6.9 Completing decimal output conversion

The hard part of the work, digit conversion, is now complete, and all that remains is some cleanup. First, we supply any required sign:

```
    if (SIGNBIT(d))
        digitbuf[--k] = '-';
    else if (flag_plus)
        digitbuf[--k] = '+';
    else if (flag_space)
        digitbuf[--k] = ' ';
```

We handle zero trimming next:

```
    if (flag_trim_zeros && (strchr(&digitbuf[k], '.') != (char *)NULL))
    {
        int j;

        for (j = (int)elementsof(digitbuf) - 2; digitbuf[j] == '0';j--)
            digitbuf[j] = '\0';

        if (digitbuf[j] == '.')
```

```
            digitbuf[j] = '\0';
    }
```

We then copy the digit string into the output buffer, supply any required final decimal point, and attach the exponent field:

```
    (void)strlcpy(buf, &digitbuf[k], sizeof(buf));

    if (flag_sharp && (strchr(buf, '.') == (char *)NULL))
        (void)strlcat(buf, ".", sizeof(buf));

    if (flag_e)
    {
        (void)strlcat(buf, flag_uppercase ? "E" : "e", sizeof(buf));
        (void)strlcat(buf, _cvtits(jusbuf, sizeof(jusbuf), n_exponent, 1, 1 + e_digits, g_digits),
                      sizeof(buf));
    }
```

If zero filling is required, we do it in `digitbuf[]`, and then copy the result back to `buf[]`:

```
    if (flag_zero && !IS_SET(CVTO_FLAG_MINUS))
    {
        int fill_count;

        fill_count = min_width - (int)strlen(buf);

        if (fill_count > 0)
        {
            const char *p;
            int need_zero_fill;

            need_zero_fill = 1;
            p = &buf[0];

            for (k = 0; *p; ++k)
            {
                if ( need_zero_fill && ( isdigit(*p) || (*p == '.') ) )
                {
                    while (fill_count-- > 0)
                        digitbuf[k++] = '0';

                    need_zero_fill = 0;
                }
                digitbuf[k] = *p++;
            }

            digitbuf[k] = '\0';
            (void)strlcpy(buf, digitbuf, sizeof(buf));
        }
    }
```

We next supply any requested digit grouping:

```
    if (g_digits > 0)
    {
        char *p;
        size_t start;
        char tmpbuf[sizeof(buf)];
        char save;
```

```
          for (p = &buf[0]; !isdigit(*p); ++p)
              /* NO-OP */ ;

          /* handle prefix of number (spaces and optional sign) */

          start = (size_t)(p - &buf[0]); /* buf[start] holds 1st digit */
          (void)strlcpy(tmpbuf, buf, start + 1);

          /* handle number */

          for (; (*p == ’.’) || isdigit(*p); ++p)
              /* NO-OP */;    /* find character AFTER number */

          save = *p;
          *p = ’\0’;          /* temporarily terminate number string */
          (void)strlcat(tmpbuf, _cvtgpn(jusbuf, sizeof(jusbuf), &buf[start], g_digits), sizeof(tmpbuf));
          *p = save;

          if (flag_e)
          {
              char e[3];

              /* handle "e+" */

              e[0] = p[0];
              e[1] = p[1];
              e[2] = ’\0’;
              (void)strlcat(tmpbuf, e, sizeof(tmpbuf));

              /* handle exponent digits */

              (void)strlcat(tmpbuf, _cvtgpn(jusbuf, elementsof(jusbuf), p + 2, g_digits), sizeof(tmpbuf));
          }

          (void)strlcpy(buf, tmpbuf, sizeof(buf));
      }
```

The final task is to supply any requested justification, replace any plus-as-space substitute with a space, and return to the caller:

```
      retbuf = (char *)_cvtjus(jusbuf, sizeof(jusbuf), buf, min_width, flags, 0);

      if (IS_SET(CVTO_SHOW_PLUS_AS_SPACE))
      {
          char * s;

          s = strchr(retbuf, PLUS_AS_SPACE);

          if (s != (char *)NULL)
              *s = ’ ’;
      }

      return (retbuf);
  }                               /* end of CVTOD() */
```

26.6.10 Computing the minimum desirable precision

The Matula formula given on page 851 predicts the minimum number of decimal digits needed to guarantee correct round-trip conversion between nondecimal and decimal bases. Although it is not difficult to implement in general, we can economize by supplying numerical values of the logarithm ratios for common bases, with a fallback to the general formula that in practice is never used in the code.

A private function in the file cvtodx.h hides the base dependence:

```
static int
cvt_precision(void)
{
    int t;

#define _logb(b,x) ((log)(x) / (log)(b))

    /* Return the minimal number of digits in base 10 needed to
       represent a T-digit base-B number.  The Matula/Goldberg result
       is ceil(1 + T/log[B](base)), but the leading "1" term is not
       needed when B is a power of b, or vice versa.  We handle common
       cases explicitly to avoid a costly base-B logarithm. */

#if B == 2

    t = (int)ceil(1.0 + (double)T * 0.3010299956639812);

#elif B == 8

    t = (int)ceil(1.0 + (double)T * 0.90308998699194343);

#elif B == 10

    t = (int)T;

#elif B == 16

    t = (int)ceil(1.0 + (double)T * 1.2041199826559246);

#else

    t = (int)ceil(1.0 + (double)T / _logb((double)B, 10.0));

#endif  /* B == 2 */

    return (t);
}
```

26.6.11 Coding fast powers of ten

In Section 26.6.2 on page 855, we described the need for fast powers of ten, but temporarily omitted the implementation so as not to interrupt the presentation of the code for CVTOD(). Here is how we handle the job with a private function in the file cvtodx.h:

```
static xp_t
power_of_10(int n)
{   /* interface to high-precision XP_EXP10(n) with fast table lookup
       for 10**n */
    xp_t result;
    int k;

    k = n + p10_bias;

    if ( (0 <= k) && (k < (int)elementsof(p10)) )
        result = p10[k];
    else
    {

#if B == 10
        result = XP_LDEXP(XP(1.), n);
#else
        result = XP_EXP10((xp_t)n);
#endif

    }

    return (result);
}
```

In a binary base, the `p10[]` array is initialized with correctly rounded hexadecimal floating-point values when compiler support is available. Otherwise, the initializers are wrapped constants like `XP(1e23)`. Trailing zeros are suppressed in order to avoid changing the quantization of decimal floating-point values.

That completes our presentation of the important mathcw library function family, `CVTOD()`, for converting native floating-point values to human-readable decimal representations. The total amount of code required is about 275 nonblank lines, somewhat less than that required for conversion to binary, octal, and hexadecimal strings. The significant difference is that, when the native floating-point arithmetic is not decimal, the digit conversion process is subject to rounding error, and thus, is inexact. We examine that problem further in the next section.

26.7 Accuracy of output conversion

The code in `CVTOD()` is complex, and the conversion flags introduce many small variations in behavior. Also, for large-magnitude exponents outside the tabulated range, scaling accuracy depends critically on the quality of the `pow()` function family. Thus, extensive testing is imperative.

For the `float` data type on 32-bit systems, an exhaustive test with every possible significand value shows that `cvtodf()` and its companion for input, `cvtidf()` described in **Section 27.4** on page 895, provide *exact* round-trip conversion.

Such a test is infeasible in higher precisions, but testing with hundreds of millions of logarithmically distributed random arguments on several different architectures shows exact conversions in the functions for the binary data types `float` and `double` when a higher-precision type is available. For the data type `long double`, the largest error does not exceed 3.6 ulps, and the average error is below 0.4 ulps.

For binary, octal, and hexadecimal conversions with the 32-bit and 64-bit decimal types, about 1% of the results show round-trip conversion errors larger than 0.5 ulps, but none above 1.8 ulps. For the 128-bit `decimal_long_double` data type, where no higher intermediate precision is currently available, about 1% of the tests with random arguments have errors above 1.0 ulps, but never bigger than 2.3 ulps.

26.8 Output conversion to a general base

At the beginning of this chapter, we describe based numbers, which represent floating-point values in any base from 2 to 36, using decimal digits and English letters as extended digits. For example, the decimal value 1 234 567 can be written in base 25 as `25@3.407h@e+4`. The constant begins with the base in decimal, its digits are delimited by `@` characters, and the power of the base is written in decimal following the letter `e`. The sample can be decoded like this: $(3 + 4/25 + 0/25^2 + 7/25^3 + 17/25^4) \times 25^4 = 3 \times 25^4 + 4 \times 25^3 + 7 \times 25 + 17 = 1\,171\,875 + 62\,500 + 175 + 17 = 1\,234\,567$.

The function that provides the general output conversion has this outline:

```
const char *
CVTOG(fp_t d, int min_width, int p_digits, int e_digits, int g_digits, int base, int flags)
{
    /* code body omitted */
}
```

As usual, the name CVTOG() represents ten variants for binary and decimal floating-point arithmetic.

The function prototype differs from that of CVTOD() by the addition of the base argument. If base is out of the range $[2, 36]$, the code resets it internally to 10, because a decimal base is most familiar to humans. The two functions handle all of the other arguments identically.

The code in CVTOD() and CVTOG() has much in common, so we do not show any more of it here. CVTOG() requires the general Matula formula to determine the maximum precision needed for correct round-trip conversion, including the omission of the extra term when the input base is a power of the output base, or vice versa. Powers of the base are generated by exact scaling when the base matches an integral power of the native base, or by table lookup for common powers of ten for a decimal base, and otherwise, by calling the pow() function family directly, but in the highest available precision.

26.9 Output conversion of Infinity

A short function provides conversion of Infinity to a string representation:

```
const char *
CVTOI(fp_t x, int flags)
{
  /* code body omitted */
}
```

The uppercase macro name represents any of ten variants for binary and decimal floating-point arithmetic.

In the flags argument, only sign and lettercase flags are obeyed, because CVTOI() is intended primarily for internal use by other conversion functions that handle justification separately.

If the argument is not an Infinity, CVTOI() returns a NULL pointer. Otherwise, it returns a pointer to a constant string ("Inf", "inf", or "INF"), possibly signed or with a leading space, depending on the settings in the flags argument.

26.10 Output conversion of NaN

The last low-level conversion function that we describe in this chapter handles conversions of NaNs, showing their type and their payloads:

```
const char *
CVTON(fp_t x, int flags)
{
  /* code body omitted */
}
```

The uppercase macro name expands to any of ten variants for binary and decimal floating-point arithmetic.

In the flags argument, only sign and lettercase flags are obeyed, because CVTON() is intended primarily for internal use by other conversion functions that handle justification separately.

If the argument is not a NaN, CVTON() returns a NULL pointer. Otherwise, it returns a pointer to a string stored in an internal static buffer that is overwritten on the next call to CVTON(), so the usual caveats about multithreaded applications apply.

For example, on one system, the call cvtonf(snanf("0xfeedface"), 0) returns the string "snan(0x2dface)". The 32-bit format has room for only a 22-bit payload, so the 32-bit payload passed to snanf() is truncated, producing the storage value 0xffadface on return. cvtonf() then determines that its argument is a signaling NaN, and returns the value shown.

CVTON() drops leading zero digits from the payload and discards the parenthesized payload altogether if it is zero, so on many systems, cvton(0.0/0.0, 0) produces the string "qnan". Some architectures, such as MIPS and SPARC, produce default NaNs with all significand bits set to one; for them, the sample call produces the string "qnan(0x7ffffffffffff)".

When the host architecture supports only one kind of NaN, the returned value begins with "nan". Examples of such systems include the Intel IA-32 architecture, the Java Virtual Machine, and the Microsoft .NET Framework Common Language Infrastructure (CLI) virtual machine.

NaN payloads cannot be expected to be the same across architectures, or to be preserved in the output of a numeric CPU instruction with a NaN operand. However, on most platforms, payloads remain intact when NaNs are simply copied from one location to another, and programmers may therefore find it useful to produce distinctive payloads in explicitly generated NaNs so that they can later be identified in output conversions.

We omit further description of the internals of the cvton() function family. The code requires messy and tedious bit manipulation for ten variants of arithmetic, two endian addressing conventions, and for 80-bit long double, suffers additional complications from systems that store them in 10-, 12-, or 16-byte fields. The code also has to handle systems that use paired double for long double, and platforms for which <float.h> incorrectly declares 80-bit long double constants, but the compiler maps that type to 64-bit double.

26.11 Number-to-string conversion

It is often useful to have simple functions for converting numbers to strings, where the format and precision are chosen automatically to allow exact round-trip conversion. The Standard C library does not have such a function, but the mathcw library does. Its ntos() function family provides this user interface:

```
const char *
NTOS(fp_t x)
{
    static char result[MAXOUT];

    /* ... code omitted ... */
}
```

The returned value is a pointer to the internal buffer result[], and as such, that storage area is overwritten on the next call to the function. Thus, the function must not be used more than once in the argument list of a call to another function, and threaded applications need to limit the use of that function to a single thread, or else protect calls to it with an exclusive lock.

For binary arithmetic, the ntos() functions are simply wrappers around a call to snprintf() with a %+.*g-style format, where the precision is chosen according to Matula's formula (see page 851). For decimal arithmetic, the code instead calls the appropriate conversion function in the IBM decNumber library.

One critical feature of the decimal conversion provided by the ntos() family is that it preserves the *quantization*, a vital aspect of decimal arithmetic that we discuss in **Section D.3** on page 931. That information is lost when the C-style library conversion routines, such as the printf() and strtod() families, are used.

26.12 The printf() family

The last set of functions that we describe in this chapter are the most complex for the user, and also for the programmer. The general notion is straightforward: a format string describes the output stream with a little language in which ordinary characters are output verbatim, and special sequences, called *conversion specifiers*, control the conversion and output of the next argument.

The output stream is written to a user-provided character-string buffer, or to a file, depending on which member of the printf() function family is called. There are eight family members, with these prototypes:

```
#include <stdio.h>
```

```
int fprintf   (FILE * restrict stream, const char * restrict format, ...);

int printf    (const char * restrict format, ...);

int snprintf  (char * restrict s, size_t n, const char * restrict format, ...);

int sprintf   (char * restrict s, const char * restrict format, ...);

#include <stdarg.h>
#include <stdio.h>

int vfprintf  (FILE * restrict stream, const char * restrict format, va_list arg);

int vprintf   (const char * restrict format, va_list arg);

int vsnprintf (char * restrict s, size_t n, const char * restrict format, va_list arg);

int vsprintf  (char * restrict s, const char * restrict format, va_list arg);
```

The four functions beginning with v were added in C89.

The restrict qualifier, introduced in C99, is an optimization hint to the compiler. It does not change the behavior of the function, but it does mean that it is the programmer's responsibility to ensure that the restrict-qualified arguments cannot overlap in memory with any other arguments.

The first two functions in each block of four write their output to a file. printf() and vprintf() are convenience functions that are equivalent to calling fprintf() and vfprintf(), respectively, with stdout as the initial argument. The remaining functions in each block send their output to a string.

All of the functions return the number of characters produced, even if some of them cannot be transmitted to a destination string because it is too small. For output to a string, the return value does *not* count the NUL string-terminator character. That character is always written, unless the output buffer size is zero. On error, the functions return EOF, a standard macro defined in <stdio.h> that expands to an unspecified negative value of type int.

Before C89, the return values from the printf() family varied between implementations, and consequently, most programmers ignore the return value, and careful ones indicate that fact by prefixing the calls with a (void) cast. Nevertheless, the return value can be useful. An initial call to snprintf() with a small string buffer can dynamically determine how much storage is really needed. The program can then allocate a suitable memory block and retry the call with the new storage.

For file output, an EOF return is a useful indication that something has gone wrong: the cause is often a full or inaccessible filesystem, or a write-protected file or directory. Programmers who are in the habit of ignoring printf() return values should at least call ferror() after a series of output statements to check that all is still well.

A return of EOF is also required for an erroneous conversion specifier. Because most implementations process the format string in a single pass, in such a case, earlier parts of the format may have already produced output, but the remainder of the format string is not processed, and in particular, is not examined for further errors, and produces no further output. In the mathcw implementation, erroneous specifiers elicit a report on stderr.

26.12.1 Dangers of printf()

The historical sprintf() function, and its newer companion vsprintf(), are deprecated, because they fail to provide an argument that defines the size of the output string. Buffer overruns from calls to those functions have been, and continue to be, a serious source of both program failure, and security breaches.

Modern code should scrupulously avoid those deprecated functions in favor of snprintf() and vsnprintf(). All modern systems have both of those functions, but up until the late 1990s, some vendors did not provide them, and they are likely to be absent from older systems that now run only in simulators.

Writing beyond the end of the output string is not the only insecurity of those functions. There are several other problems with all of the printf() family that programmers need to be aware of:

- If the format argument is not a compile-time constant, but instead can be specified at run time by the user, or otherwise modified in memory through another security breach, then it can be used to subvert the software.

  That problem is likely to increase, in part because software internationalization requires replacement of all program strings that could be visible in output with calls to library functions that look them up in translation libraries, and return a locale-specific alternative. For example, internationalization allows the famous greeting, `"Hello, world"`, from the original book on C to be output as `"Bonjour, le monde"` and `"Hej, verden"` in French and Danish locales.[2]

  Another example of that kind of insecurity arises when the programmer replaces a simple `printf("%s", t)` with the more compact `printf(t)`: if `t` contains conversion specifiers, their interpretation by `printf()` has unpredictable consequences. The safe way is to use the original call, or else `fputs(t, stdout)`.

- When there are fewer arguments after the format string than conversion specifiers, the `printf()` code attempts to retrieve nonexistent arguments: program failure is likely. The design of the C calling conventions makes it *impossible* in any portable way for `printf()` to detect missing arguments.

- Extra arguments beyond the last one consumed by a conversion specifier are ignored. That is generally harmless, but may nevertheless be puzzling when expected output does not appear. Fortran programmers are used to format strings being reused cyclicly when there are extra arguments, but that does not happen with `printf()`.

  Programmers are advised that there is so much processing behind input and output functions that there is no reason to write giant format strings and long lists of arguments; they just make format debugging harder. A series of function calls, each with just a few conversion specifiers, is likely to be as fast, and the code is easier to write, and to read.

- If conversion-specifier widths or precisions are provided at run time by additional arguments, it is impossible to predict by source-code analysis or at compile time how much output can be produced.

- If the `%n` conversion specifier, which writes the current output character count into a memory location defined by a pointer argument, is used, then an attacker may be able to exploit that feature to write arbitrary text at arbitrary memory locations, completely subverting security. Once again, the design of the C language makes it impossible for `printf()` to validate an argument pointer before storing data into the memory location that it references.

- All `printf()` implementations need internal storage to develop temporary copies of format conversions, but they may not be universally reliable in preventing their code from writing beyond the bounds of that storage. C89 requires that any *single* conversion specifier be able to produce at least 509 characters, and C99 raises that limit to 4095 characters. Ours supports 10239 characters. Older implementations are unlikely to even document such limits.

  String conversions with the `%s` specifier might be handled by simply copying characters from an argument to the output, without needing an intermediate storage buffer, but modifiers in the specifier that set justification, precision, or minimum width may force use of an internal buffer. A program that successfully outputs long strings with `printf()` on one system may therefore fail on another where the implementation technique differs, or internal buffers are smaller. Long strings might be handled more efficiently and more safely with `fputs()` or `strncpy()`.

- Ignored return values can mask or delay recognition of an error in a conversion specifier, or a filesystem problem, until much later, when the error can be much harder to locate and diagnose. Often, that means that the program reports successful completion, when in fact it failed to perform as expected.

- In C, the `scanf()` input conversion specifiers and the `printf()` output specifiers look similar, but behave differently. That usually surprises Fortran programmers, who are used to identical behavior for input and output format items.

[2] See `ftp://ftp.gnu.org/gnu/hello/` for a sample implementation of string internationalization.

- The width modifier in specifiers sets the *minimum* output width, but the output field expands if needed to contain a long result. No mechanism exists in printf() to specify a maximum field width that works for all defined specifiers.

Tools like cppcheck, its4, lint, rats, and splint, and some compilers with additional options, can analyze source code and report potential vulnerabilities, including writable format strings, and mismatches in argument counts and data types in the printf() family.

In older languages, like Fortran and Pascal, I/O is part of the language, rather than relegated to the run-time library as it is in C, so compilers interpret I/O statements, and can prevent mismatches in argument counts and types. Output to strings was not part of the original definitions of those languages, so the buffer-overrun security hole was absent as well.

The designers of some later languages, such as Ada and Java, chose to eliminate convenient formatted I/O entirely; they force the programmer to call object-to-string conversion functions, largely eliminate pointers, and make safe strings a part of the language. Nevertheless, the demand for better control over output formatting led to the introduction of a printf()-like string-formatting facility in Java 1.5, more than a decade after the language was first introduced.

Many scripting languages developed in the UNIX world include a printf() function or statement, although some exclude snprintf(), the %n conversion specifier, and run-time setting of widths and precision, in an attempt to improve security and reliability. See Seacord's comprehensive treatment [Sea05] of security problems in C and C++ for that example, and many others.

The view of many seasoned programmers is that the convenience and the power of the printf() family are too important to give up, particularly because most uses of those functions are completely safe. Highways, oceans, and skies, and computer and telephone networks, carry lots of safe and legal traffic, but they also have accidents, failures, and illegal activities. The best approach with all of those technologies is to be informed of the dangers, and proceed with caution.

26.12.2 Variable argument lists

The printf() family, and the scanf() family input companions, and their wide-character variants, are the only functions with variable argument lists in the Standard C run-time library. The feature is useful for user code, often for specialized functions that are wrappers that ultimately call I/O functions. The four new functions beginning with v were added in C99 to facilitate the writing of user functions that act that way.

For example, in a large program, it may be desirable to make error reporting uniform, without needing to include <stdio.h> everywhere, and without committing the output destination to a file or a string: it might instead need to go to a network connection, or to a window system. Here is how such a wrapper can be written:

```
#include <stdarg.h>
#include <stdio.h>

void
error(int severity_code, const char * format, ...)
{
    va_list args;

    va_start(args, format);
    (void)fprintf (stderr, "Error code %d: ", severity_code);
    (void)vfprintf(stderr, format, args);
    (void)fprintf (stderr, "\n");
    va_end(args);

    exit(EXIT_FAILURE);
}
```

A typical call might then look like this:

```
error(ENOSPC, "device full in output to %.255s", filename);
```

Before C89, variable arguments were handled by a slightly different syntax, and a different header file, <varargs.h>. Modern code should use the new style defined in <stdarg.h> exclusively, and some recent compilers have dropped support for the old style entirely, so we do not describe it further here.

26.12.3 Implementing the printf() family

The key observation about the printf() functions is that they behave identically, apart from what happens to their output: it may go to a file or to a string. If it goes to a string, the specification of the snprintf() and vsnprintf() functions requires that it *not* be an error to produce more characters than the string buffer can hold, but execution must continue with storage suppressed when it would be beyond the end of the buffer.

In the mathcw library, we handle all eight family members identically, by introducing a special destination data structure called a *sink*, and defined like this:

```
typedef struct sink_s
{
    FILE *stream;
    char *s;
    size_t n;
    size_t next;
} sink_t;
```

Exactly one of the two pointers is NULL, and for a string, the two remaining structure members record the maximum string size, and the index of the next available slot in the string.

The four functions beginning with v then have simple code bodies:

```
int
(vfprintf)(FILE * restrict stream, const char * restrict format, va_list arg)
{
    sink_t sink;

    return (vprt(new_sink_file(&sink, stream), format, arg));
}

int
(vprintf)(const char * restrict format, va_list arg)
{
    sink_t sink;

    return (vprt(new_sink_file(&sink, stdout), format, arg));
}

int
(vsnprintf)(char * restrict s, size_t n, const char * restrict format, va_list arg)
{
    sink_t sink;

    return (vprt(new_sink_string(&sink, s, n), format, arg));
}

int
(vsprintf)(char * restrict s, const char * restrict format, va_list arg)
{
    sink_t sink;

    return (vprt(new_sink_string(&sink, s, SIZE_T_MAX), format, arg));
}
```

The parenthesized function names prevent preprocessor function-like macro expansion. The private functions new_sink_file() and new_sink_string() handle the initialization of a sink object, which is then not examined further until a character is ready for output. vsprintf() deals with the lack of an output size by declaring the output string to have the largest possible value that can be represented in a value of type size_t, even though that is certainly wrong.

The remaining four functions are only slightly more complicated, and they all call their counterparts that start with v:

```
int
(fprintf)(FILE * restrict stream, const char * restrict format, ...)
{
    int count_or_eof;
    va_list ap;

    va_start(ap, format);
    count_or_eof = (vfprintf)(stream, format, ap);
    va_end(ap);

    return (count_or_eof);
}

int
(printf)(const char * restrict format, ...)
{
    int count_or_eof;
    va_list ap;

    va_start(ap, format);
    count_or_eof = (vprintf)(format, ap);
    va_end(ap);

    return (count_or_eof);
}

int
(snprintf)(char * restrict s, size_t n, const char * restrict format, ...)
{
    int count_or_eof;
    va_list ap;

    va_start(ap, format);
    count_or_eof = (vsnprintf)(s, n, format, ap);
    va_end(ap);

    return (count_or_eof);
}

int
(sprintf)(char * restrict s, const char * restrict format, ...)
{
    int count_or_eof;
    va_list ap;

    va_start(ap, format);
    count_or_eof = (vsprintf)(s, format, ap);
    va_end(ap);

    return (count_or_eof);
```

Table 26.5: Output conversion specifiers (part 1). Each may be enhanced with additional modifiers following the percent.
Using brackets to indicate optional values, the general syntax of a conversion specifier is
%[*flags*][*width*][.*precision*[.*exponentsize*[.*groupsize*[.*base*]]]]*letter*.
The *exponentsize*, *groupsize*, and *base* modifiers, and the %@ conversion specifier, are important extensions in the mathcw library and the hoc language; they do not exist in other languages in the C family.
Specifiers in the second part of the table are extensions to C89 and C99.

Item	Description
%%	Literal percent character. For C99, the optional values *must* be omitted for that specifier, but that is not required by the mathcw library implementation.
%A	Hexadecimal floating-point value with uppercase digits (e.g., -0X*d*.*ddd*...P+*d*). The exponent has a minimum number of digits, and is zero for a zero value. [C99]
%a	Hexadecimal floating-point value with lowercase digits (e.g., -0x*d*.*ddd*...p+*d*). The exponent has a minimum number of digits, and is zero for a zero value. [C99]
%c	unsigned char, or with the l (ell) modifier, unsigned wchar_t.
%d	Decimal integer.
%E	Floating-point value (e.g., -*d*.*ddd*...E+*dd*). The exponent has at least two digits unless an exponent width is specified. Infinity and NaN are as for %F.
%e	Floating-point value (e.g., -*d*.*ddd*...e+*dd*). The exponent has at least two digits unless an exponent width is specified. Infinity and NaN are as for %f.
%B	Base-2 floating-point value with uppercase prefix and exponent letter (e.g., -0B*d*.*ddd*...P+*d*). The exponent has a minimum number of digits, and is zero for a zero value. [mathcw library and hoc]
%b	Base-2 floating-point value with lowercase prefix and exponent letter (e.g., -0b*d*.*ddd*...p+*d*). The exponent has a minimum number of digits, and is zero for a zero value. [mathcw library and hoc]

```
}
```

The private function vprt() handles the real work of controlling the formatted-output processing. With its roughly two dozen private subsidiary functions, it contains about 1800 lines of code whose complexity is largely related to managing control flow, with a score of switch statements and about 230 case statements. Its private functions handle integer and string conversions, but it leaves all of the floating-point conversions to about two dozen mathcw library functions whose names are prefixed with cvto, and whose code amounts to more than 3000 lines.

The code sizes that we quote demonstrate that input and output are complex operations that often can be hidden behind a single statement in user code. I/O facilities need to be powerful, standardized, and applicable to all supported data types.

Some historical languages, such as Algol and Bliss, sidestepped the complexity by simply omitting I/O, thereby dooming those languages to suffer from idiosyncratic and nonstandard additions of vendor-specific I/O facilities, and loss of portability. Ultimately, they fell out of use, and became dead languages.

For all of their flaws, the formatted I/O facilities of Fortran and C have survived for several decades. In C, the fact that they are part of the run-time library, rather than the language itself, makes it possible to extend them for new capabilities, such as decimal floating-point arithmetic, without invalidating a *single line* of existing software, and without requiring modification of compilers.

26.12.4 Output conversion specifiers

For programmers new to C, the printf() conversion specifiers represent a considerable obstacle in the learning process, but because they appear in numerous scripting languages, and even in the UNIX shells [RB05a], it is worthwhile to learn how to use them effectively. Their concise specification occupies about ten pages of the ISO C Standards, and they are usually reasonably summarized in online documentation, such as the UNIX manual pages, so in this section, we concentrate on a description of their extensions in the mathcw library versions of the printf() family to support decimal floating-point arithmetic, and other new features.

A conversion specifier begins with a percent character, followed by zero or more flag characters which may occur in any order and for which repetitions are permitted, but carry no additional meaning. Flag characters in turn

Table 26.6: Output conversion specifiers (part 2). Each may be enhanced with additional modifiers following the percent. Specifiers in the second part of the table are extensions to C89 and C99.

Item	Description
`%F`	Floating-point value (e.g., `-`*`ddd.ddd`*`...`). Infinity and NaN are coded as described in the text. If a decimal point is present, at least one digit precedes it. [C99]
`%f`	Floating-point value (e.g., `-`*`ddd.ddd`*`...`). Infinity and NaN are coded as described in the text. If a decimal point is present, at least one digit precedes it.
`%G`	`%E` or `%f` format with trailing zeros removed. See Section 26.6.4 on page 857 for details of how the choice is made. Infinity and NaN are as for `%F`.
`%g`	`%e` or `%f` format with trailing zeros removed. Infinity and NaN are as for `%f`.
`%i`	Decimal integer. For output conversions, it is identical to `%d`.
`%n`	Argument is an `int *` pointer to a location where the current output character count is written.
`%o`	Unsigned octal integer value.
`%p`	`void *` pointer (implementation-dependent format, but suitable for input with `scanf()`). May not be usable for pointers to functions on some systems. [C89].
`%s`	`signed char *` or `unsigned char *`, or with the `l` (ell) modifier, `wchar_t *`.
`%u`	Unsigned decimal integer value.
`%X`	Unsigned hexadecimal integer. Letters `A`-`F` represent 10 to 15.
`%x`	Unsigned hexadecimal integer. Letters `a`-`f` represent 10 to 15.
`%Q`	Octal floating-point value with uppercase prefix and exponent letter (e.g., `-0B`*`d.ddd`*`...P+`*`d`*). The exponent has a minimum number of digits, and is zero for a zero value. [mathcw library and hoc]
`%q`	Octal floating-point value with lowercase prefix and exponent letter (e.g., `-0b`*`d.ddd`*`...p+`*`d`*). The exponent has a minimum number of digits, and is zero for a zero value. [mathcw library and hoc]
`%Y`	Unsigned binary integer. [mathcw library and hoc]
`%y`	Unsigned binary integer. [mathcw library and hoc]
`%@`	Based-number floating-point value (e.g., `-`*`base`*`@`*`d.ddd`*`...@e+`*`d`*). [mathcw library and hoc]

are followed by an optional sequence of nonnegative integers, or asterisks, separated by dots, then by an optional data-type modifier, and finally end with a character that selects the desired conversion.

Except for percent conversion, each specifier consumes the next unused argument in the function call. In the absence of errors, format and argument processing continues until the parsing meets the final `NUL` terminator in the format string. For output to a string, the function then writes a terminating `NUL`, and returns the output character count, *excluding* that `NUL`, to the caller. For the functions that produce an output string, the return value is exactly what `strlen()` would return for that string.

Table 26.5 on the preceding page through Table 26.13 on page 877 summarize the conversion specifiers. In each table, bracketed notes in the caption or descriptions identify features introduced after the original definition of `printf()` in the early 1970s.

The new format features first introduced and tested in hoc, and then incorporated in the mathcw library, are:

- `%B` and `%b` for binary floating-point output,
- `%Q` and `%q` for octal floating-point output,
- `%Y` and `%y` for binary integer output,
- `%@` for based-number output,
- `=` flag for centering,
- `/` flag for trimming trailing zeros,
- `^` flag for uppercasing based-number output,
- exponent-width, grouping and base modifiers,
- type modifiers `h`, `hL`, `lL`, and `LL` for binary floating-point data,
- type modifiers `H`, `DD`, `DL`, and `DLL` for decimal floating-point data.

Table 26.7: Flag-character format modifiers for conversion specifiers. There may be zero or more flags in any order following the percent sign that starts the specifier, and flags may be repeated without changing their meaning. Flags in the second part of the table are extensions to C89 and C99.

Item	Description
`-`	Value is left-justified in field width. If the flag is not specified, the default is right-justified. A minus preceding a field width is interpreted as the flag followed by a positive width.
`+`	Signed conversion always begins with plus or minus sign. Otherwise, only negative values are signed. Negative floating-point values include negative zero, and negative values that round to zero.
space	Use space instead of + for positive values. Ignore the flag if the + flag is present.
`#`	Alternate conversion. For all floating-point conversions, force the result to contain a base point, even if no digits follow it. For `%G` and `%g`, preserve trailing zeros. For `%o`, guarantee a leading zero digit. For `%X` and `%x`, prefix a *nonzero* result with `0X` or `0x`, respectively. For `%Y` and `%y`, prefix results with `0B` or `0b`, respectively.
`0`	For all numeric conversions, use leading zeros, instead of spaces, to pad to the field width, except when converting Infinity or NaN. Ignore the flag if the - flag is present. For all integer conversions, ignore the flag if the precision is specified.
`=`	Value is centered in field width. [mathcw library and hoc]
`/`	Strip trailing fractional zero digits. [mathcw library and hoc]
`\`	Show subnormals with leading zeros. [mathcw library]
`^`	Convert output to uppercase. [mathcw library and hoc]

The exponent-width modifier follows the precision, as it does in Fortran 77, and remedies a long-standing omission in C that makes it difficult to use `printf()` to properly align tables of numbers written with `%E` or `%e` conversion. Standard C's default of a two-digit exponent reflects floating-point designs of machines of the 1960s, and is another example where a historical mistake in the design of the original `printf()` was incorporated into two international standards.

The grouping modifier solves a problem that exists in almost all programming languages: long strings of digits are hard to read and check. For centuries, professional typesetters of numerical tables have inserted thin spaces every three, four, or five digits, making the tabulations much more usable, because humans can usually remember such short groups with little effort. Ada allows underscores for digit grouping in program source code, but provides no support for them in input and output. hoc supports them in both code and data. Once it is more widely available, digit grouping is likely to become commonplace in program code, input, and output, and the group size therefore follows the exponent width.

The base modifier is needed for only one conversion specifier, `%@`, and its rareness mandates that it be the last modifier.

As we noted earlier on page 853, the choice of a default of six for an omitted precision in `%E`, `%e`, `%F`, `%f`, `%G`, and `%g` conversions is a historical mistake that is unsuited for all current and historical floating-point architectures.

For standards conformance, our `printf()` also uses that default for those six conversions, but for the new `%A`, `%a`, `%B`, `%b`, `%Q`, `%q`, and `%@` conversions, Matula's formula (see page 851) determines the default. That choice ensures that sufficient digits are generated either to represent the internal number exactly, or when the internal and external bases differ, to allow exact round-trip conversion.

For mnemonic purposes, it would have been most convenient for the decimal floating-point type modifiers to match the corresponding suffixes on constants. Alas, the introduction of the `%F` conversion in C99 to allow production of uppercase names for Infinity and NaN prevents that: a `decimal_float` value would have an ambiguous conversion specifier of `%DFe` that could be interpreted as a `%e`-style conversion, or an erroneous `%F`-style conversion. The proposals for decimal arithmetic in C [Kla05, C06b, C06c, C06d] use modifiers `H`, `D`, and `DD` for the 32-bit, 64-bit, and 128-bit formats. However, we find the last two of those choices confusing, and undesirable when a 256-bit format is added.

Until C89, the language did not even support a single-precision data type, a situation that arose because its primary use during its early years was for programming operating systems and text utilities, where floating-point arithmetic is only of occasional use. C89 added the `float` type, but failed to include corresponding mathematical library functions. That deficiency was not remedied until C99.

Table 26.8: Precision format modifiers for conversion specifiers. The precision modifier is optional; default values are supplied as indicated. A negative precision is treated as a missing precision modifier.

Item	Description
`.number`	*Minimum* number of digits for all integer conversions.
	Number of digits after the base point for `%A`, `%a`, `%B`, `%b`, `%E`, `%e`, `%F`, `%f`, `%Q`, `%q`, and `%@` conversions.
	Maximum number of significant digits for `%G` and `%g` conversions.
	Maximum number of bytes for `%s` conversions.
	If a numeric value can be represented in fewer characters than the precision, it is padded with leading zeros to the precision.
	A base point appears in numeric values only if it is followed by a digit, unless the `#` flag is present.
	Floating-point values are *rounded* to the number of digits implied by the precision.
	If `number` is omitted, it is taken as 0.
	If the precision modifier is missing in `%A`, `%a`, `%B`, `%b`, `%Q`, `%q`, or `%@`, the precision used is sufficient for an exact representation of the value.
	If the precision modifier is missing in `%E`, `%e`, `%F`, `%f`, `%G`, or `%g`, it is taken as 6.
	If the precision modifier is missing in integer conversions, it is taken as 1.
	If the precision modifier is missing in `%s`, it is taken as the length of the corresponding string argument.
	For `%E`, `%e`, `%F`, and `%f`, if the precision is zero, no decimal point appears unless the `#` flag is present.
	For `%G` and `%g`, a precision of zero is taken as 1.
	For `%G` and `%g`, trailing zeros in the fractional part are dropped unless the `#` flag is present.
	Warning: For all integer conversions, converting a zero value with zero precision results in an *empty* string.
`.*`	Precision supplied by corresponding integer argument.

Table 26.9: Minimum field-width format modifiers for conversion specifiers. The modifier is optional; if it is omitted, then the result has the minimum width needed to hold the value.

Item	Description
`number`	Field width in characters. The converted value is padded on the left with spaces, on the right if the `-` flag is present, and centered with the `=` flag. If the value is too big, the field is automatically expanded to hold it, and no padding is provided.
`*`	Field width supplied by corresponding integer argument

Table 26.10: Exponent-width format modifiers for conversion specifiers. [mathcw library and hoc]

Item	Description
`.number`	Minimum number of digits in an exponent.
	If omitted, negative, or zero, the default is 1 for `%A`, `%a`, `%B`, `%b`, `%Q`, `%q`, and `%@`, and 2 for `%E` and `%e`. If needed, leading zeros pad exponents to the width.
`.*`	Exponent width supplied by corresponding integer argument.

Table 26.11: Digit-grouping format modifiers for conversion specifiers. Digit groups are separated by a single underscore character. The integer and fractional parts, and the exponent, but not the base, are candidates for grouping. [mathcw library and hoc]

Item	Description
`.number`	Number of digits in a group, counting away from the base point.
	Omitted and negative values are taken as 0 (no digit grouping).
`.*`	Group count supplied by corresponding integer argument.

Table 26.12: Number-base format modifiers for conversion specifiers. Ignored for all but `%@` conversion. [mathcw library and hoc]

Item	Description
`.number`	Number base for `%@` conversion. If omitted, or outside the interval $[2, 36]$, assume base 10.
`.*`	Number base supplied by corresponding integer argument.

Table 26.13: Data-length format modifiers for output conversion specifiers. Modifiers are required if the corresponding argument has a type other than the default for the conversion specifier.

Item	Description
	integer conversion
`hh`	`char` argument [C99]
`h`	`short int` argument
`j`	`intmax_t` or `uintmax_t` argument [C99]
`l`	`long int`, `wint_t`, or `wchar_t` argument
`ll`	`long long int` argument [C99]
`t`	`ptrdiff_t` argument [C99]
`z`	`size_t` argument [C99]
	binary floating-point conversion
`h`	`float` argument [mathcw library]
`L`	`long double` argument
`LL`	`long_long_double` argument [mathcw library]
`hL`	extended (`__float80`) argument [mathcw library and HP-UX]
`lL`	quad (`__float128`) argument [mathcw library and HP-UX]
	decimal floating-point conversion
`H`	`decimal_float` argument [mathcw library]
`DD`	`decimal_double` argument [mathcw library]
`DL`	`decimal_long_double` argument [mathcw library]
`DLL`	`decimal_long_long_double` argument [mathcw library]

Table 26.14: Binary and octal floating-point output, with decimal, hexadecimal, and based-number output for comparison. The test value is $x = 1.5625 \times 8^{-3}$. Notice that exponents in the binary, octal, and hexadecimal formats represent powers of two, but exponents in based numbers specify powers of the base.

Function call	Output
`printf("%b\n", x)`	`+0b1.1001p-9`
`printf("%q\n", x)`	`+0o1.44p-9`
`printf("%e\n", x)`	`3.051758e-03`
`printf("%.12.6.3e\n", x)`	`3.051_757_812_500e-000_003`
`printf("%/a\n", x)`	`0x1.9p-9`
`printf("%/....2@\n", x)`	`2@1.1001@e-9`
`printf("%/....8@\n", x)`	`8@1.44@e-3`
`printf("%/....10@\n", x)`	`10@3.0517578125@e-3`
`printf("%/..2.3.10@\n", x)`	`10@3.051_757_812_5@e-03`
`printf("%/....16@\n", x)`	`16@c.8@e-3`

Neither C89 nor C99 provides for output of `float` values with the `printf()` family, although they do allow input of such values with the `scanf()` functions. When the `float` data type was introduced by some compilers prior to its standardization in C89, function prototypes were still absent from the language, and compiler writers chose to simplify argument passing by converting `float` arguments to `double`. Similar widening promotions had long been done for `char` and `short int` arguments, both of which are passed as `int` values. In the absence of a function prototype, that practice remains the standard behavior in the C language. Since C89, function prototypes eliminate the need for such widening, *except* in the case of functions with variable argument lists. Unfortunately, that widening causes two problems for IEEE 754 arithmetic:

- It is no longer possible to distinguish subnormal values from normal ones, even though it would be useful to have a flag for numerical format conversions that requests that subnormals be shown in exponential form with leading zeros.
- Widening is a numerical operation that destroys the payloads of NaNs on some architectures, and on most, converts a signaling NaN to a quiet NaN. In both cases, important information is lost.

  Tests on all of the platforms available to this author at the time of writing this show that a narrowing cast

> always converts a signaling NaN to a quiet NaN, and except on the PowerPC CPU, a widening cast causes the same conversion.

In the first implementation of decimal floating-point arithmetic in C, that unwanted widening does not happen with `decimal_float`.

If future ISO C Standards eliminate the widening of `float` arguments, they will make `float` a first-class data type, instead of a poor afterthought. In the `mathcw` library, the new `h` and `H` data type modifiers identify arguments of type `float` and `decimal_float`, and for the latter type only, avoid loss of information when those arguments undergo output conversion.

Four of the new conversion specifiers provide for output of floating-point numbers in binary and octal form. Table 26.14 shows some examples of those specifiers, and comparisons with other formats.

For the `%a` conversion specifier introduced in C99, the Standard requires that there be a single nonzero hexadecimal digit before the point, as long as the number is normalized. The formatting of subnormals is not specified by the Standard. It is regrettable that it did not require that digit to be one, because an important use of that specifier is to reveal exact bit patterns for comparison of output across platforms, and different vendors have made different choices that make such comparisons impractical. For example, few people can tell at a glance that `0xc.90fdap-2` and `0x1.921fb4p+1` are identical representations of π in the IEEE 754 32-bit format. The `mathcw` implementation of `printf()` prevents that problem because the `cvtob()`, `cvtoh()`, and `cvtoo()` function families that handle `%b`, `%a`, and `%q` conversions guarantee a unit leading digit for all precisions and all bases.

The Standard requires that the `#` flag provide a `0X` prefix in `%X` conversion, and `0x` for `%x` conversion, *except* when the value is zero. That too is a historical mistake that causes alignment problems in tables. As a workaround, supply the prefix manually, such as in the format `"0x%.08x\n"`. Our `printf()` follows the Standard in that case, but for the new binary integer `%Y` and `%y` conversions, it preserves the `0B` and `0b` prefixes on zero values.

26.13 Summary

In this chapter, we described how to perform accurate output conversion, and we identified the critical areas of digit production where accuracy can be lost. In the next chapter, we discuss the accuracy issue further, and give references to important historical work on the problem.

We separated the job of number-to-string conversion from the conventional output routines by providing a family of conversion functions that give the user more control of output formatting.

We discussed the dangers of the `printf()` family, and argued that its capabilities outweigh the risks. Nevertheless, programmers need to be careful in their use of functions for output formatting.

We generalized the Standard C `printf()` function by supporting additional format flags and specifiers, allowing further control over field justification, trimming of trailing zeros, digit grouping, exponent width, and output as binary, octal, and based numbers. When the extensions are not exploited, our `printf()` function follows the 1990 and 1999 ISO C Standards exactly, so the increased power is transparent to *all* existing C and C++ code.

The coding effort for implementing the `printf()` family is substantial: its algorithm file, `vprtx.h`, is by far the largest of all of the more than 500 such files in the `mathcw` library, and it requires our separate output-conversion functions. Its companion for input that we describe in the next chapter is the second largest in the library.

27 Floating-point input

> WHILE WORKING ON THE BEEF TESTS FOR
> TRANSCENDENTAL FUNCTIONS, IT WAS DISCOVERED THAT
> THE TURBO C 1.0 COMPILER DID NOT CONVERT 11.0 EXACTLY
> INTO A FLOATING POINT NUMBER EQUAL TO 11!
>
> — GUY L. STEELE JR. AND JON L. WHITE
> *Retrospective: How to Print Floating-Point Numbers Accurately* (2003).

The introduction to the last chapter briefly described format specifiers in C, Fortran, and Pascal. When used with the input facilities of those languages, they allow decimal data to be read easily, but binary, hexadecimal, and octal formats are unlikely to be widely supported.

C99 extends the `strtod()` library function to recognize hexadecimal as well as decimal strings, and also Infinity and NaN strings. C99 also adds the companion `strtof()` and `strtold()` functions for conversion of such strings to `float` and `long double` values. The naming of the `strtod()` family is regrettably irregular, but it is too late to repair that mistake.

In this chapter, we describe the `mathcw` library support for floating-point input. The library supplies conversion utilities for all supported precisions and floating-point types, and for input in bases 2, 4, 8, 16, and 32, guarantees correct rounding to the internal binary storage format. Decimal input is always correctly rounded for decimal data types.

In several respects, the output problem is easier than the input problem. Output formatting is done by software, and is generally reliable. It deals with a small, and fixed, number of bits in the value to be converted. Input formatting is often done by humans, and software that converts input strings to internal formats must be carefully written to be robust against input errors, and must be able to handle strings of arbitrary and unpredictable length. Some of the Standard C library routines that convert input strings assume that they are well formed and of modest length, and provide no error indication when they are not.

All of the input routines described in this chapter can return a pointer to the next character to be processed, and that character should be checked after each conversion to make sure that it is one that can follow a correctly formatted input string.

The header file `cvticw.h` contains prototypes for the input functions, and if the decimal floating-point functions are needed, that header file must be included after `mathcw.h`.

27.1 Binary floating-point input

We start our description of the input-conversion functions in the `mathcw` library with the one for a binary floating-point string, because that is one of the simplest floating-point formats to handle, and the code that we present for it is not difficult to adapt for octal and hexadecimal strings.

Besides digit collection for the integer, fractional, and exponent parts, we have to deal with several extra complications:

- input errors;
- rounding to storage format;
- overflow in the digit sequences;
- overflow and underflow in the final value; and
- Infinity and NaN values.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_27

We model the input-conversion software interface on the standard library routine `strtod(s, &endptr)`. If the input string in the first argument is properly formatted, that function returns the floating-point result of the conversion, and if the second argument is not `NULL`, stores in that object a pointer to the character *following* the last character processed in the conversion. Otherwise, the function returns zero, and if the second argument is not `NULL`, stores the first argument in that object. Conversion is therefore successful if `*endptr` differs from `s` on return, and in addition, `**endptr` is a character that can legally follow a number.

In order to program the conversion, we have to specify what valid binary floating-point input looks like:

- optional leading whitespace (as identified by `isspace()`);
- optional plus or minus sign;
- `inf` or `infinity` (letter case is not significant), or
- `nan`, `qnan`, or `snan` (letter case is not significant), optionally followed by a parenthesized sequence of zero or more characters, or
- the binary prefix `0b` or `0B` followed by a string of digits 0 or 1 containing at most a single binary point, with digits perhaps separated by an underscore, optionally followed by `p` or `P` and a possibly signed decimal integer representing a power of two, optionally followed by a type suffix.

That informal description can be tightened by writing it as a sequence of complicated `egrep`-style regular expressions, which are string-matching patterns where square brackets enclose character sets, parentheses delimit subexpressions, and backslash-letter combinations are the usual C-language escape sequences that represent unprintable characters. The metacharacter `*` means *zero-or-more* of the regular expression that precedes it, and `?` means *zero-or-one*. An initial caret inside a square-bracketed set complements the set, and a hyphen between two characters in a set denotes a character range.

The patterns that we need to recognize are these:

- `[ \f\n\r\t\v]*[-+]?[Ii][Nn][Ff][Ii][Nn][Ii][Tt][Yy]`
- `[ \f\n\r\t\v]*[-+]?[Ii][Nn][Ff]`
- `[ \f\n\r\t\v]*[-+]?[QqSs]?[Nn][Aa][Nn]([(][^)]*[)])?`
- `[ \f\n\r\t\v]*[-+]?0[Bb]([01](_?[01])*)*([.]([01]_?)*[01])?`
  `[Pp][-+]?([0-9]_?)*[0-9]([FfLl]|LL|ll)?`
- `[ \f\n\r\t\v]*[-+]?0[Bb]([01](_?[01])*)*([.]([01]_?)*[01])?`
  `(([Uu](LL|ll|L|l)?)|((LL|ll|L|l)[Uu]?))`

In the last two items, the long expressions have been split across lines, and their length tells us that code that interprets them is likely to be complex.

In each case, the longest valid match is chosen. For example, the input string `inflate` consists of the valid string `inf`, followed by the string `late` which is ignored. Similarly, `0b1011LUMP` is treated as the valid string `0b1011LU` followed by `MP`.

Several of the subexpressions are needed in the conversion code for other bases, so it makes sense to break the job into several separately compiled functions. We start with the user-visible routine for binary floating-point input conversion, and because of its size, we present it in parts, delaying the helper functions until later:

```
fp_t
CVTIB(const char *s, char **endptr)
{
    const char *s_in;
    fp_t result;
    int nonzero_digits;
    int sign;
```

As usual, the function-wrapper macro stands for one of several precision-dependent functions, `cvtibf()`, `cvtib()`, `cvtibl()`, and so on. The name `CVTIB()` is an acronym for ***ConVerT** from **I**nput in **B**inary*. The first argument points to the string to be converted, and the second argument, if non-`NULL`, points to an object that, on return, holds a pointer to the character *following* the last one successfully processed.

The first task is to clear a counter, and record the start of the input string, because we require its value if the conversion fails.

```
nonzero_digits = 0;
s_in = s;
```

The next job is to skip over any leading space:

```
while (isspace(*s))
    ++s;
```

Helper functions are named with a leading underscore, indicating that they are for internal use only, and are not intended to be called directly from user code. The first of the helper functions handles recognition of the optional sign:

```
sign = _cvtsgn(s, (char**)&s);
```

If a sign is present, `s` is advanced by one character on return from `_cvtsgn()`.

The main part of the code is a large block that we implement as a one-trip loop with `do {...} while (0)`, purely for the convenience of being able to terminate the processing early with a `break` statement when a conversion succeeds:

```
do                                  /* one-trip loop */
{
    const char *s_val;

    s_val = s;
```

We save the starting value of `s` so that we can detect successful conversions later in the block.

We handle the special cases of Infinity and NaN immediately with two helper functions:

```
    result = CVTINF(s, (char**)&s);

    if (s != s_val)
        break;

    result = CVTNAN(s, (char**)&s);

    if (s != s_val)
    {
        (void)SET_EDOM(result);
        break;
    }
```

If either input conversion succeeds, `s` has advanced, and we leave the block.

We are now ready to convert a binary floating-point string. That is the largest part of the code, but before we present the first code chunk, we need to discuss how the binary point and rounding are handled.

The digit string could be treated as integer and fractional parts, with separate digit-collection loops for each. As we encounter each new digit, we double the accumulator variable, and add the value of the new digit. A sufficiently long input string for the integer part could cause premature overflow, so we have to guard against that by rescaling the accumulator when its value becomes too big. However, we can simplify the code and avoid the need for rescaling, if we use a single loop, keep track of the position of the binary point, and pretend that the binary point follows the first digit. That way, the accumulating sum never exceeds 2.0.

The rounding action needed is determined by the last bit that fits in the storage format, and the bits that follow it in the input digit string. Floating-point hardware for IEEE 754 arithmetic keeps track of four special bits (see **Section 4.6** on page 66, and [Omo94, Chapter 6], [Par00, Section 18.3], [Kor02, Section 4.6], [EL04a, Section 8.4.3], or [MBdD^{+}10, Section 8.2 and Chapter 10]):

L: the last bit in the storage format;

G: the *guard bit*, which follows the L-bit;

R: the *rounding bit*, which follows the G-bit; and

S: the *sticky bit*, which represents all of the bits following the R-bit, but records only their logical OR.

We show later how those four bits determine the rounding action.

The code in the big block for digit accumulation begins like this:

```
    if (_cvtcmp("0b", s) == 0)
    {
        const char *s_power;
        fp_t scale;
        int G_bit, L_bit, R_bit, S_bit;
        int dot_seen, exp_overflow, n_after_dot;
        int n_before_dot, n_bit, n_total, one_bit_seen;
        int power_of_2;
```

We first initialize all of the block-local variables:

```
        G_bit = 0;
        L_bit = 0;
        R_bit = 0;
        S_bit = 0;
        dot_seen = 0;
        exp_overflow = 0;
        n_after_dot = 0;
        n_before_dot = 0;
        n_bit = 0;
        n_total = 0;
        one_bit_seen = 0;
        power_of_2 = 0;
        result = ZERO;
        scale = TWO;
```

Next, we advance past the binary prefix that we just matched:

```
        s += 2;
```

The digit-collection loop begins next:

```
        while ( ISBDIGIT(*s) || (*s == '.') )
        {
            int bit;

            if (*s == '.')
            {
                if (dot_seen)
                    break;

                dot_seen = 1;
                ++s;
                continue;
            }
```

If we have a binary point, we record that fact, advance the string pointer, and continue with the next loop iteration. However, meeting a second binary point mandates a loop exit.

Otherwise, we have a binary digit. We convert it to a number, advance the string pointer, and count the digit, because we need to know later whether we found any digits at all:

```
bit = DIGIT_VALUE(*s);
++s;
++n_total;
```

Some complex logic is now required to handle the tricky cases of long strings of 0 bits preceding or following a 1 bit.

A straightforward digit accumulation here could easily produce premature overflow or underflow. For example, `0b0.000...001p+1000006`, where the ellipsis indicates a million consecutive 0 digits, is a legal input string, and easily representable, but if we do not skip the leading 0 digits, it cannot easily be evaluated without underflow, unless we periodically rescale the result. Similarly, the input `0b1000...000p-1000006`, where the ellipsis has the same meaning as before, is representable, but overflows if we continue digit accumulation without rescaling.

The solution to the overflow and underflow problems is to delay the accumulation until we have seen the first 1 bit, and to terminate digit accumulation, apart from updating of the four special bits, once we have seen `T` digits. We also need to know the relative position of the binary point. Two logical values and two counters allow us to record that information, and the code to set them looks like this:

```
if (bit == 1)
    one_bit_seen = 1;

if (one_bit_seen)
{
    ++n_bit;

    if (!dot_seen)
        ++n_before_dot;
}
else
{
    if (dot_seen)
        ++n_after_dot;
}
```

Notice that we only increment `n_bit` and `n_before_dot` after we have seen a 1 bit, and we count bits after the binary point only if we have not yet encountered a 1 bit.

Because we need to recognize the four special bits, we have to compare the digit count with the precision to choose how to handle each digit:

```
if (n_bit == 0)
    /* NO-OP */ ;
else if (n_bit <= T)
{
    scale *= HALF;

    if (bit == 1)
        result += scale;

    if (n_bit == T)
        L_bit = bit;
}
```

As long as we have seen at least one nonzero digit, and no more than `T` digits, we halve the scale factor, accumulate a scaled digit, and possibly record the L-bit. Because we accumulate at most `T` digits that way, there is no possibility of premature overflow from a long digit string. Because the scale factor is initialized to 2.0, the first digit accumulated has a value 1.0, which puts the implicit binary point immediately after the first 1 bit.

Once we have seen more than `T` digits, we can set the remaining three special bits:

```
else if (n_bit == (T + 1))
    G_bit = bit;
```

```
            else if (n_bit == (T + 2))
                R_bit = bit;
            else
                S_bit |= bit;
```

Next, we skip over any digit separator, completing the digit-collection loop:

```
            if ( (s[0] == '_') && ISBDIGIT(s[1]) )
                ++s;
        }
```

If we accumulated no digits, then the input string is invalid, and we can leave the one-trip loop:

```
        if (n_total == 0)
        {
            s = s_in;
            break;
        }
```

Otherwise, the conversion is successful so far, and we process the optional exponent:

```
        s_power = s;
        power_of_2 = _cvtpow(s, (int)'p', &exp_overflow, (char **)&s);
```

The helper function returns the exponent as the function value, and also sets a flag to record whether the exponent itself overflows, because we need that information later. If no exponent is present, the function returns a zero value.

Finally, we process an optional suffix, either a floating-point suffix if we had an exponent or saw a binary point, or else an integer suffix:

```
        if ( (s != s_power) || dot_seen )
            (void)_cvtfsf(s, (char**)&s);
        else
            (void)_cvtisf(s, (char**)&s);
```

The suffix helper functions return a type code, but we do not require it here, so we discard it with a `(void)` cast.

If we successfully processed characters after the optional sign, then the last two steps of the conversion handle the rounding and the scaling to account for the position of an explicit binary point and a specified exponent:

```
        if (s != s_val)
        {
            int expon, n_scale;

            result = CVTRND(result, scale, sign, L_bit, G_bit, R_bit, S_bit);

            if (n_before_dot > 0)
                n_scale = n_before_dot - 1;
            else
                n_scale = -(n_after_dot + 1);

            if (!exp_overflow && is_add_safe(power_of_2, n_scale))
                expon = power_of_2 + n_scale;
            else
                expon = (power_of_2 < 0) ? INT_MIN : INT_MAX;

            result = LDEXP(result, expon);
        }

        nonzero_digits = one_bit_seen;
    }
```

We cannot safely compute the final exponent by simply adding power_of_2 and n_scale until we have checked for overflow in both the power and the sum. If either of them overflows, we set the exponent to the largest representable integer of the correct sign. The LDEXP() function is exact, and its result underflows or overflows only if the input number is not representable after it has been scaled.

If we did not have a valid binary prefix, then the input string is invalid, so we reset s to its original value, and finish off the one-trip loop:

```
        else                    /* unrecognized input */
            s = s_in;
    }
    while (0);                  /* end one-trip loop */
```

The conversion is complete, so we set the object pointed to by the second argument:

```
    if (endptr != (char**)NULL)
        *endptr = (char *)s;
```

We set the sign with the COPYSIGN() function to prevent a signaling NaN from being converted to a quiet NaN, and to protect negative zeros from compiler mishandling:

```
    if (s == s_in)
        result = ZERO;
    else
    {
        result = COPYSIGN(result, (fp_t)sign);
```

We would prefer to return a NaN if the string does not contain a number, because *not-a-number* is the most appropriate value in an IEEE 754 environment. However, C99 mandates that strtod() return a zero, so we follow that requirement. Finally, we set errno to ERANGE if underflow or overflow occurred, and return:

```
        if ( (nonzero_digits > 0) && (result == ZERO) )
            (void)SET_ERANGE(result);   /* record underflow */
        else if (ISINF(result))
            (void)SET_ERANGE(result);   /* record overflow */
    }

    return (result);
}
```

We assume that underflows flush to zero, rather than wrapping to a large value. For IEEE 754 arithmetic, overflow is detected by checking whether the final result after scaling is Infinity. On some older architectures, overflow detection is more difficult, and we leave that problem unsolved for now.

Apart from helper functions, CVTIB() requires about 140 lines of code. The only machine constants in the code are T (the floating-point precision in bits for a binary base), and INT_MAX and INT_MIN (the integers of largest magnitude). We present the helper functions in the following subsections.

27.1.1 Sign input conversion

Recognition of an optional sign is a straightforward task with obvious code:

```
int
_cvtsgn(const char *s, char **endptr)
{   /* convert optional sign */
    int sign;

    if (*s == '+')
    {
        ++s;
        sign = 1;
    }
    else if (*s == '-')
    {
        ++s;
        sign = -1;
    }
    else
        sign = 1;

    if (endptr != (char**)NULL)
        *endptr = (char *)s;

    return (sign);
}
```

27.1.2 Prefix string matching

The Standard C library does not have a function for case-insensitive string comparisons. We need such a function to simplify recognition of Infinity and NaN input strings as prefixes of longer strings, and the code is not difficult, because we can ignore the lettercase problems described earlier on page 848:

```
int
_cvtcmp(const char *prefix, const char *s)
{   /* compare prefix with s, ignoring case */
    int c1, c2, result;

    result = 0;

    while ((*prefix != '\0') && (*s != '\0'))
    {
        c1 = TOLOWER(*prefix);
        c2 = TOLOWER(*s);

        if (c1 < c2)
        {
            result = -1;
            break;
        }
        else if (c1 > c2)
        {
            result = 1;
            break;
        }
        ++prefix;
        ++s;
    }

    if ( (result == 0) && (*prefix != '\0') && (*s == '\0') )
        result = 1;

    return (result);
}
```

Like the TOUPPER() macro introduced on page 848, TOLOWER() is a wrapper to ensure correct behavior with older C implementations.

The return values of _cvtcmp() are like those for the standard library routine strcmp(). In particular, a zero return value indicates a match, and that is all that we require later. Return values of -1 and $+1$ indicate less-than and greater-than ordering, respectively.

27.1.3 Infinity input conversion

Infinity is represented by two different input strings, and the longest-match rule for input conversion requires that we check for the longer one first:

```
fp_t
CVTINF(const char *s, char **endptr)
{
    fp_t result;

    if      (_cvtcmp("infinity", s) == 0) { result = INFTY(); s += 8; }
    else if (_cvtcmp("inf", s)      == 0) { result = INFTY(); s += 3; }
    else                                    result = ZERO;

    if (endptr != (char**)NULL)
        *endptr = (char *)s;

    return (result);
}
```

The `INFTY()` function computes Infinity dynamically on IEEE 754 systems, and that in turn sets the *overflow* exception flag as a side effect. On non-IEEE-754 systems, the function returns the largest representable floating-point number.

27.1.4 NaN input conversion

The Standard C library function `strtod()` recognizes only one kind of NaN, but because many systems have both quiet and signaling NaNs, that limitation in `strtod()` is a design flaw that we avoid repeating.

We recognize `qnan`, `snan`, and `nan` in any letter case, optionally followed by a parenthesized string whose contents we collect and hand off to a NaN-generator function for further processing.

```
fp_t
CVTNAN(const char *s, char **endptr)
{
    const char *s_in;
    fp_t result;
    char *p;
    char tag[sizeof("0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffff")];

    s_in = s;

    if (_cvtcmp("qnan", s) == 0)
    {
        (void)strlcpy(tag, (s[4] == ’(’) ? s + 5 : "", sizeof(tag));
        p = strchr(tag, ’)’);

        if (p == (char *)NULL)
            tag[0] = ’\0’;
        else
            *p = ’\0’;

        result = QNAN(tag);
        s += 4;
    }
    else if (_cvtcmp("snan", s) == 0)
    {
        (void)strlcpy(tag, (s[4] == ’(’) ? s + 5 : "", sizeof(tag));
        p = strchr(tag, ’)’);
        if (p == (char *)NULL)
            tag[0] = ’\0’;
        else
            *p = ’\0’;

        result = SNAN(tag);
        s += 4;
    }
    else if (_cvtcmp("nan", s) == 0)
    {
        (void)strlcpy(tag, (s[3] == ’(’) ? s + 4 : "", sizeof(tag));
        p = strchr(tag, ’)’);

        if (p == (char *)NULL)
            tag[0] = ’\0’;
        else
            *p = ’\0’;

        result = NAN_(tag);
        s += 3;
    }
    else
        result = ZERO;

    if (s > s_in)
    {
        if (*s == ’(’)          /* skip (n-char-sequence) */
        {
            const char *s_open;
            int level;

            level = 1;
            s_open = s;
            ++s;

            while (*s)
            {
                if (*s == ’(’)
                    level++;
                else if (*s == ’)’)
                    level--;

                if (level == 0)
                    break;

                ++s;
            }

            s = (level == 0) ? s + 1 : s_open;
        }
    }

    if (endptr != (char**)NULL)
        *endptr = (char *)s;

    return (result);
```

```
}
```

If the string following the NaN has balanced parentheses, our code uses the contents as a NaN payload, and sets `*endptr` to the address of the character following the close parenthesis. Otherwise, the code uses an empty payload, and sets `*endptr` to the address of the open parenthesis.

The 1999 ISO C Standard does not specify that the parenthesized string must contain balanced parentheses, even though that would seem to be a reasonable expectation. Thus, `"NaN((((((()"` could be considered a valid string that contains a parenthesized 5-char-sequence. Our code accepts the NaN, and rejects the rest.

We ignore an *n*-char-sequence that does not have a closing right parenthesis. Thus, `"SNaN(abc"` is recognized as a valid signaling `NaN` followed by the string `"(abc"`.

27.1.5 Power input conversion

Recognition of a floating-point exponent is only a little harder than collecting a decimal integer string. However, we must avoid the C library routines that convert decimal strings to integers, because they do not detect integer overflow. Undetected exponent overflow is unacceptable, because it produces nonsensical conversions.

We anticipate using that function for several kinds of floating-point numbers, so we provide the exponent letter as an argument, rather than hard coding it in the function:

```
int
_cvtpow(const char *s, int letter, int *oflptr, char **endptr)
{
    const char *s_in;
    int overflow, power;

    s_in = s;

    overflow = 0;
    power = 0;

    if (TOLOWER(*s) == TOLOWER(letter))
    {
        int n_dec, sign;

        ++s;
        n_dec = 0;
        power = 0;

        sign = _cvtsgn(s, (char**)&s);

        while (isdigit(*s))
        {
            int last_power;

            n_dec++;
            last_power = power;
            power = 10 * power - (int)DIGIT_VALUE(*s);

            if (power > last_power)
                overflow = 1;

            ++s;

            if ( (s[0] == '_') && isdigit(s[1]) )
                ++s;
        }
        if (overflow)
            power = INT_MIN;

        if (sign > 0)
        {
            power = -power;

            if (power < 0)
            {
                overflow = 1;
                power = INT_MAX;
            }
        }

        if (n_dec == 0)         /* incomplete input */
            s = s_in;
    }

    if (endptr != (char**)NULL)
        *endptr = (char *)s;

    if (oflptr != (int*)NULL)
        *oflptr = overflow;

    return (power);
}
```

There are some subtle points in the code that must be noted:

- Because two's-complement arithmetic has one more negative number than positive numbers, the most negative integer cannot be accumulated as a positive value. We therefore accumulate the negative of the decimal digit string. Most textbooks that show code for conversion of decimal strings get that wrong.
- Integer overflow is possible in two places. Inside the loop, overflow is detected when the updated accumulator wraps to a nonnegative value, and thus, exceeds its previous value. It is *not* sufficient to check for a positive accumulator, because that mishandles the case of a zero exponent. The second place that overflow is possible is outside the loop, where the sign is applied.
- The C language (and most others) guarantees that integer overflow is ignored, so we can safely continue digit accumulation after detecting overflow.
- Because the floating-point exponent field uses many fewer bits than an integer can hold, we can safely replace an overflowed value by the closest representable integer. It will precipitate a floating-point underflow or overflow when the exponent is later applied by a call to `LDEXP()`.

27.1.6 Floating-point suffix conversion

The type of floating-point numbers in C is indicated by a suffix letter: `F` or `f` for `float`, none for `double`, and `L` or `l` for `long double`. Curiously, the Standard C library input and output functions neither recognize nor produce such suffixes.

We anticipate possible future extensions to C by recognizing input suffixes, including `LL` and `ll` for a `long long double` type. As required by the C language, the suffixes must be in a uniform lettercase, even though that restriction serves no purpose, and complicates parsing.

Our function return values are small integer values encoded as symbolic names defined as macros in the header file `cvtsfx.h`. The code is then straightforward:

```
int
_cvtfsf(const char *s, char **endptr)
```

```
{   /* optional binary floating-point suffix: ([FfLl]|LL|ll)? */
    /* optional decimal floating-point suffix: (DF|DD|DL|DLL)? */
    int result;

    if (TOLOWER(s[0]) == (int)’f’)
    {
        ++s;
        result = CVT_SUFFIX_F;
    }
    else if ( (s[0] == ’L’) && (s[1] == ’L’) )
    {
        s += 2;
        result = CVT_SUFFIX_LL;
    }
    else if ( (s[0] == ’l’) && (s[1] == ’l’) )
    {
        s += 2;
        result = CVT_SUFFIX_LL;
    }
    else if (TOLOWER(s[0]) == (int)’l’)
    {
        ++s;
        result = CVT_SUFFIX_L;
    }
    else if (TOLOWER(s[0]) == (int)’d’)
    {
        if ( (s[0] == ’d’) && (s[1] == ’f’) )
        {
            s += 2;
            result = CVT_SUFFIX_DF;
        }
        else if ( (s[0] == ’D’) && (s[1] == ’F’) )
        {
            s += 2;
            result = CVT_SUFFIX_DF;
        }
        else if ( (s[0] == ’d’) && (s[1] == ’d’) )
        {
            s += 2;
            result = CVT_SUFFIX_DD;
        }
        else if ( (s[0] == ’D’) && (s[1] == ’D’) )
        {
            s += 2;
            result = CVT_SUFFIX_DD;
        }
        else if ( (s[0] == ’d’) && (s[1] == ’d’) )
        {
            s += 2;
            result = CVT_SUFFIX_DD;
        }
        else if ( (s[0] == ’D’) && (s[1] == ’L’) && (s[2] == ’L’) )
        {
            s += 3;
            result = CVT_SUFFIX_DLL;
        }
        else if ( (s[0] == ’d’) && (s[1] == ’l’) && (s[2] == ’l’) )
        {
            s += 3;
            result = CVT_SUFFIX_DLL;
        }
        else if ( (s[0] == ’D’) && (s[1] == ’L’) )
        {
            s += 2;
            result = CVT_SUFFIX_DL;
        }
        else if ( (s[0] == ’d’) && (s[1] == ’l’) )
        {
            s += 2;
            result = CVT_SUFFIX_DL;
        }
        else
            result = CVT_SUFFIX_NONE;
    }
    else
        result = CVT_SUFFIX_NONE;

    if (endptr != (char**)NULL)
        *endptr = (char *)s;

    return (result);
}
```

27.1.7 Integer suffix conversion

Integer constants in C have type suffixes `U` or `u` for `unsigned int`, `L` or `l` for `long int`, and `LL` or `ll` for `long long int`. The standard library input and output functions do not support those suffixes, but our code does.

An unsigned suffix can precede or follow the length suffix, and the length suffix must be in a uniform lettercase. Both of those features complicate the code. The lettercase restriction increases the number of possible suffixes from 7 to 22, and means that matching with a series of nested conditionals is complex and error-prone, so we adopt a table-driven approach instead.

As with the recognition of floating-point suffixes, our function return values are small integer values with symbolic names defined as macros in the header file `cvtsfx.h`. The code looks like this:

```
int
_cvtisf(const char *s, char **endptr)
{    /* integer suffix: ([Uu](LL|ll|L|l)?)? or ((LL|ll|L|l)[Uu]?)? */
    int k, result;
    typedef struct
    {
        const char *name;
        size_t size;
        int code;
    } match_table_t;
    static match_table_t t[] =
    {
        { "LLU", 3, CVT_SUFFIX_ULL }, { "LLu", 3, CVT_SUFFIX_ULL }, { "ULL", 3, CVT_SUFFIX_ULL },
        { "Ull", 3, CVT_SUFFIX_ULL }, { "llU", 3, CVT_SUFFIX_ULL }, { "llu", 3, CVT_SUFFIX_ULL },
        { "uLL", 3, CVT_SUFFIX_ULL }, { "ull", 3, CVT_SUFFIX_ULL },

        { "LL", 2, CVT_SUFFIX_LL },   { "ll", 2, CVT_SUFFIX_LL },

        { "LU", 2, CVT_SUFFIX_UL },   { "Lu", 2, CVT_SUFFIX_UL },   { "UL", 2, CVT_SUFFIX_UL },
        { "Ul", 2, CVT_SUFFIX_UL },   { "lU", 2, CVT_SUFFIX_UL },   { "lu", 2, CVT_SUFFIX_UL },
        { "uL", 2, CVT_SUFFIX_UL },   { "ul", 2, CVT_SUFFIX_UL },

        { "L", 1, CVT_SUFFIX_L },     { "l", 1, CVT_SUFFIX_L },

        { "U", 1, CVT_SUFFIX_U },     { "u", 1, CVT_SUFFIX_U },
    };

    result = CVT_SUFFIX_NONE;

    if ( (TOLOWER(*s) == 'l') || (TOLOWER(*s) == 'u') )
    {
        for (k = 0; k < (int)elementsof(t); ++k)
        {
            if (strncmp(t[k].name, s, t[k].size) == 0)
            {
                s += t[k].size;
                break;
            }
        }
    }

    if (endptr != (char**)NULL)
        *endptr = (char *)s;

    return (result);
}
```

Notice that the table entries must be ordered so that we check for the longest suffixes first.

In the usual case that a suffix is absent, a straightforward table-driven version would have to compare the string against all 22 suffixes, and would be considerably slower than a version with conditionals, which requires only four tests on the first character of the string to find out that no suffix matches. However, the code with the suffix table is much easier to understand, and the outer `if` statement avoids the search entirely in most cases.

27.1.8 Input rounding adjustment

The last helper function adjusts an accumulated value by adding a small adjustment equal to a half or whole unit in the last place. It uses the rounding direction computed dynamically by another helper function, `_cvtdir()`, that we described in Section 26.3.5 on page 848.

We assume that the value to be adjusted, `x`, is nonnegative, with a sign supplied in a separate argument. The four special bits that we described in Section 27.1 on page 881 make up the final arguments:

```
fp_t
CVTRND(fp_t x, fp_t adjust, int sign, int L_bit, int G_bit, int R_bit, int S_bit)
{
    switch (_cvtdir())
    {
    case CVT_FE_DOWNWARD:
        if ((G_bit + R_bit + S_bit) > 0)
        {
            if (sign < 0)
                x += adjust + adjust;
        }
        break;

    default:                    /* FALLTHROUGH */
    case CVT_FE_TONEAREST:
        if (G_bit == 1)
        {
            if ((R_bit + S_bit) == 0)
            {
                if (L_bit == 1)
                    x += adjust;
            }
            else
                x += adjust;
        }
        break;

    case CVT_FE_TOWARDZERO:
        break;

    case CVT_FE_UPWARD:
        if ((G_bit + R_bit + S_bit) > 0)
        {
            if (sign > 0)
                x += adjust + adjust;
        }
        break;
    }

    return (x);
}
```

Rounding toward zero is the easiest case, because it corresponds to truncation of superfluous bits.

Rounding to $-\infty$ is like rounding to zero if x is nonnegative, but otherwise, requires adding a whole unit in the last place to the positive x.

Rounding to $+\infty$ is similar, but requires an adjustment only when x is positive.

The case of rounding to nearest, with ties broken by rounding to even, is the most complex. It is the only one that requires knowledge of the last storable bit. When the following special bits are 100, adjustment by a half ulp is required only if the L-bit is 1. Otherwise, the adjustment is a half ulp only if the G-bit is 1.

Although `_cvtdir()` does not return any values but the four cases listed, it is always a good idea to guard against future code changes, and supply a `default` case in `switch` statements. We handle that case by falling through into the *round-to-nearest* code.

27.2 Octal floating-point input

With the binary floating-point input problem solved by `CVTIB()`, a companion for octal input has this outline:

```
fp_t
CVTIO(const char *s, char **endptr)
{
    /* ... code omitted ... */
}
```

As usual, the macro represents several precision-dependent names: `cvtiof()`, `cvtio()`, `cvtiol()`, and so on.

The key to its implementation is the realization that correct rounding requires precise knowledge of bit boundaries, and computation of the L, G, R, and S bits described in Section 27.1 on page 881. The easiest way to do that is to replace the inner loop over bits with a pair of loops, the first processing octal digits, and the second extracting bits from each digit.

We do not show the complete code, but the general idea is evident from this fragment:

```
            while (ISODIGIT(*s) || (*s == '.'))
            {
                int digit, k;
                if (*s == ’.’)
                {
                    if (dot_seen)
                        break;

                    dot_seen = 1;
                    ++s;
                    continue;
                }

                digit = DIGIT_VALUE(*s);
                ++s;

                for (k = 2; k >= 0; --k)
                {
                    int bit;

                    bit = (digit >> k) & 1;
                    /* ... code omitted ... */
                }

                /* skip over optional digit separators */
                if ( (s[0] == ’_’) && ISODIGIT(s[1]) )
                    ++s;
            }
```

A test program exercises many special cases of octal floating-point input, and then tests round-trip conversion with `CVTOO()` and `CVTIO()` for large numbers of random arguments. No differences have been found in thousands of millions of such tests, so we can be reasonably confident that both input and output conversions for octal floating-point data work correctly.

27.3 Hexadecimal floating-point input

With the octal floating-point input code as a model, a companion for hexadecimal input is straightforward, and has this outline:

```
fp_t
CVTIH(const char *s, char **endptr)
{
    /* ... code omitted ... */
}
```

The macro `CVTIH()` represents several precision-dependent names: `cvtihf()`, `cvtih()`, `cvtihl()`, and so on.

`CVTIH()` differs from the octal routine, `CVTIO()`, in just *five lines* of code: the prefix is now `0x` instead of `0o`, the digit test uses `ISXDIGIT()` instead of `ISODIGIT()`, the `DIGIT_VALUE()` macro is redefined, and the bit-extraction loop index starts with `k = 3` instead of `k = 2`.

27.4 Decimal floating-point input

In this chapter, and the preceding one on floating-point output (see **Chapter 26** on page 829), we showed how binary, octal, and hexadecimal floating-point strings can be handled exactly, and with correct rounding, without requiring access to higher-precision arithmetic.

The decimal input and output problems when the host arithmetic is not itself decimal are much harder. Although solutions were worked out in the 1970s and 1980s, they were unpublished until 1990 [Cli90, Knu90, Gri90, SW90], and those results were later improved upon [Gay90, BD96, ABC$^+$99, Loi10]. Two recent retrospectives [Cli04, SW04] review those developments, and are worth reading; they preface the original papers [Cli90, SW90] with additional historical commentary. The original algorithms were written in dialects of Lisp that provide multiple-precision arithmetic. Gay provides a package in C for both input and output, but his code is specific to IEEE 754 systems, and not usable at all on older architectures.

Conversion of human-readable text strings to numbers in internal binary formats has gone through several stages in the development of the C language:[1]

- In the early years of C, conversion of strings to integers and floating-point numbers was available only through the functions `atoi()` and `atof()`, producing results of type `int` and `double`, respectively. Those functions were written in assembly code in UNIX V3 in 1973, but C versions appeared in UNIX V6 in 1975, and in 1979, UNIX V7 added `atol()` for `long int` results.
- UNIX V7 added the formatted-input conversion function `sscanf()`. It handled integer strings with inline code, but called `atof()` for floating-point strings.
- UNIX SYSTEM V, released in 1983, included two new functions `strtod()`, for conversion to data type `double`, and `strtol()`, for conversion to `long int`.
- C89 incorporated the two new SYSTEM V functions in the ANSI and ISO Standards, and added `strtoul()` for `unsigned long int` results.
- C99 adds `strtof()` and `strtold()` for conversion to `float` and `long double` floating-point types, and `strtoimax()`, `strtoll()`, `strtoull()`, and `strtoumax()` for conversion to `intmax_t`, `long long int`, `unsigned long long int`, and `uintmax_t` integer types.
- C99 requires that `strtof()`, `strtod()` and `strtold()` recognize hexadecimal floating-point values, Infinity, and NaN, and mandates correct rounding according to the current rounding direction.

The early conversion code was workable, but not reliable. For integer conversions, the original `atoi()` accumulates a positive sum, then negates the result if a negative sign prefixes the digit string. The code does not check for overflow, and in two's-complement arithmetic, for which there is no corresponding positive number, it is impossible to correctly input the most negative integer.

The converter from string to floating point, `atof()`, tries a bit harder. As long as the sum is less than a constant `big` (the largest exactly representable whole number), `atof()` continues digit accumulation. The last digit collected in the loop can therefore produce an inexact result. `atof()` then skips subsequent digits, but increments the exponent for each such digit until it finds a nondigit. If the next character is a decimal point, `atof()` continues digit accumulation if the sum is below `big`, and decrements the exponent for each digit. Otherwise, it skips remaining digits. If the following character is an exponent letter, it collects the explicit exponent value itself, without checking for overflow, or even that at least one digit is present. The sum of the two exponents determines the scale factor 10^n, which `atof()` expresses as $2^n \times 5^n$, then computes 5^n by successive squaring and bitwise reduction of n, and applies the final factor of 2^n exactly by calling `ldexp()`. That implicitly assumes a base-2 floating-point system. On the Interdata 8/32, which has IBM-style hexadecimal floating-point arithmetic, a revised version of the `atof()` code computes 10^n with $n - 1$ multiplications, introducing serious accuracy loss from rounding and from wobbling precision.

Neither `atof()` nor `atoi()` nor `sscanf()` provides the caller with any indication of how much of the string was parsed successfully, so they silently return 123 for the string `"123abc"`, and 0 for the string `"hello"`.

Ideally, a string-to-number conversion function should have these properties:

- Erroneously formatted strings must be reported to the caller.
- Overflow in either the number or the exponent must be detected and reported to the caller.
- Overflow must produce the largest representable magnitude of the appropriate sign. For floating-point arithmetic systems that support an Infinity, that value must be used in place of the signed finite number of largest magnitude.
- In the absence of overflow, integer conversion, even of the most negative representable number, must be exact.

[1] See the Web site of *The Unix Heritage Society* at `http://minnie.tuhs.org/` for source code and charts of development history.

- In the absence of overflow and underflow, the result of a floating-point conversion must be correctly rounded to the nearest representable machine number, possibly according to the current rounding mode, when more than one mode is available.
- Aberrant input, such as a long string of digits with a large-magnitude exponent (`1.000...e-10000` or `0.000...001e+10000`) must be converted correctly if the mathematical number lies in the range of representable machine numbers.

As we observed earlier, those problems can all be handled, but always-correct conversion is only possible if the result is first computed exactly, then converted to working precision with a single rounding. That in turn requires access to multiple-precision arithmetic. Without that luxury, digit accumulation and the computation of the exponent power are both subject to rounding errors, and must be handled carefully to avoid introducing additional errors.

The Standard C conversion functions with the `str` prefix have an indirect pointer as a second argument, so that the caller can examine the substring that follows any successfully parsed characters. In most cases, that substring should begin with a `NUL`, whitespace, or a punctuation character that could legally follow numbers in the context in which they are used. The functions report overflow by setting the global value `errno` to `ERANGE`, and return the largest-magnitude representable value of the appropriate type and sign. For floating-point underflow, they may return either zero or the smallest representable number, and they may or may not set `errno` to `ERANGE`. If conversion is not possible, they return a zero value.

We have noted elsewhere that global variables, such as `errno`, may be unreliable in the presence of threads, but that problem cannot be solved without introducing additional arguments in the conversion routines.

As in the conversion functions for binary, hexadecimal, and octal floating-point strings, we adopt the same calling sequence as the `strtod()` function family. The function outline looks like this, with the usual macro wrapper standing for one of several precision-dependent functions:

```
fp_t
CVTID(const char *nptr, char **endptr)
{
  /* ... code omitted ... */
}
```

After skipping any leading whitespace and recognizing an optional sign, a one-character lookahead allows us to identify candidates for Infinity and NaN conversion which we handle by calls to one of the conversion functions described earlier in this chapter.

Otherwise, we have a possible decimal number string. For decimal floating-point arithmetic, we copy it into an internal buffer, eliminating any digit-separating underscores, and collecting only as much of it that is a valid number string. If the collection was successful, we call `_cvtfsf()` (see **Section 27.1.6** on page 890) to scan over any floating-point suffix string, and set `endptr`, if it is not `NULL`. We then call functions in the `decNumber` library [Cow07] that do the conversion exactly.

The need for an internal buffer introduces an undesirable limit on the size of a decimal number string. That limit could be removed by allocating the string dynamically, but the `mathcw` library design specifications do not permit that. The limit that we choose is guaranteed to be at least 4095 characters, a minimum limit mandated by C99 for formatted output fields. The buffer is a local variable, so it resides on the stack, and is in use only during the execution of `CVTID()`.

For binary arithmetic, we do the digit accumulation in the highest available precision, even if that means software arithmetic on some systems.

We provide for exact digit accumulation by using an accumulator array defined by this data type:

```
typedef struct
{
    xp_t a[MAX_ACC];    /* accumulator cells */
    int  n[MAX_ACC];    /* numbers of digits */
} acc_t;
```

A precomputed limit that is ten times smaller than the maximum representable whole number serves as a cutoff for moving to the next cell. Each cell holds the value of as many digits as can fit exactly, effectively compressing the digit string, and the accompanying count is the number of digits up to, and including, that cell.

On completion of digit collection, we have a representation of the input string as a sum of products of exactly representable whole numbers and associated integral powers of ten, except that digits beyond what can fit in the penultimate cell are ignored. However, the powers of ten are as yet unevaluated.

The value of MAX_ACC is set in the cvtid.h header file. In exact arithmetic, it needs to be large enough for the accumulator to contain a number as wide as a fixed-point number spanning the entire exponent range (see **Table 4.2** on page 65, **Table D.1** on page 929, and **Table D.2** on page 929). In the 80-bit IEEE 754 binary format, that is 9884 digits, which requires MAX_ACC to be 521. The 128-bit format needs 9899 digits, and MAX_ACC set to 291. For the future 256-bit binary format, MAX_ACC must be at least 35 105, and in the corresponding decimal format, 44 939. However, in inexact arithmetic, a value of 5 usually suffices, because other errors exceed the rounding correction from the extra accumulator cells.

We evaluate the exponent digit string with _cvtpow() (see **Section 27.1.5** on page 889) so that exponent overflow can be detected.

The final result is then obtained by summing the accumulator products in order of increasing magnitude, where the explicit exponent is added to the cell exponents so that only a single power of ten is required for each cell. The is_safe_add() function provides an essential check for integer overflow in the exponent arithmetic.

Alas, it is here in that final sum that we fail to meet the goal of exact rounding, unless we can handle it with an exact multiple-precision arithmetic package. Until a suitable portable package is identified, we use a fused multiply-add to minimize errors.

Accuracy is catastrophically lost if any product falls into the subnormal region, or worse, underflows to zero. To prevent that, we check the combined exponent of our multiword accumulator, and if it drops below a cutoff that we set to about twice the number of decimal digits in the significand above the power of ten of the smallest normal number, we offset the combined exponent by a suitable constant. After the accumulation, one further scaling by ten to the negative of that offset recovers the final value.

Here is what the final reconstruction code looks like:

```
if (exp_overflow)
    number = (exponent < 0) ? XZERO : INFTY();
else
{
    int e_bias;
    int offset;

    e_bias = total_digits - fractional_digits;

    if (is_add_safe(e_bias, exponent))
        e_bias += exponent;
    else
        e_bias = (exponent < 0) ? INT_MIN : INT_MAX;

    offset = (e_bias < e_small) ? e_offset : 0;
    number = XZERO;

    for (k = n_acc; k >= 0; --k)
    {
        if (is_add_safe(e_bias, offset - acc.n[k]))
            number = XP_FMA(acc.a[k], power_of_10(offset + e_bias - acc.n[k]), number);
        else
            number += (e_bias < 0) ? XZERO : INFTY();
    }

    if (offset != 0)
        number *= power_of_10(-offset);
}
```

Only a few lines of the reconstruction code need to be replaced with calls to a multiple-precision package to reduce the round-trip conversion errors to zero for binary arithmetic.

In the code shown, the two largest sources of error are in the product at $k = 0$ with a power of ten that may not be exactly representable, and in the final scaling for numbers near the subnormal region, where both the multiplication and the power may introduce additional rounding errors.

The `power_of_10()` function is a private one that uses fast table lookup for most common powers, and otherwise calls the exact function `XP_LDEXP()` in a decimal base, or the approximate function `XP_EXP10()` in a nondecimal base. In the `mathcw` library, tests show that the latter is almost always correctly rounded, and that is superior to the bitwise reduction and squaring used in the original `atof()` for computing 5^n. Of course, in those days, most machines that C ran on had much smaller exponent ranges, so that approach was acceptably accurate because roughly the first ten multiplications needed for 5^n were exact.

In summary, conversion in decimal arithmetic is always exact, or correctly rounded. For binary arithmetic, exhaustive testing of every possible IEEE 754 32-bit floating-point value, and with millions of 64-bit floating-point values selected from logarithmic distributions over the entire floating-point range, finds no round-trip conversion errors. However, in the highest available precision, the test program reports maximum errors of under four ulps, and average errors below 0.4 ulps. Simpler code that accumulates the result in a single number produces maximum errors about one ulp larger.

27.5 Based-number input

It is occasionally desirable to be able to input numbers written in any of several number bases. As we observed in the introduction to **Chapter 26**, the Ada language allows program source code to contain *based literals* of the form `base#digits[.digits]#[exponent]`, where `base` is a decimal value from 2 to 16, and brackets indicate optional fields. Later Ada standards allow the `#` character to be replaced by a colon, although that practice is rare. The digits are in the indicated number base, with letters `A` through `F` representing digit values 10 through 15. The exponent is the letter `E`, followed by an optionally signed decimal integer that specifies the power of the base by which the fixed-point number specified between the `#` delimiters is to be multiplied. Lettercase is ignored in the digits and exponent, and the digit values must be in the range $[0, \texttt{base} - 1]$. For example, `5#2.01#e+3`, `8#3.77#e2`, `10#255#`, and `16#ff#` all represent the decimal value 255.

However, the *sharp* character, `#`, identifies preprocessor directives and operators in the C language family, and is also commonly used as a comment-start symbol in input files for UNIX utilities, so it would be unwise to employ it for yet another purpose. Instead, for the `mathcw` library, we use the *at* character, `@`. For historical reasons,[2] that character is not considered part of the C character set, even though it has been present in computer character sets for decades, and now sees wide use in electronic-mail addresses. We generalize the Ada practice by allowing any base in $[2, 36]$, using the 26 letters of the English alphabet to provide the needed digits. Thus, `36@aBc.XyZ@E3` is the decimal value 623 741 435.

The `mathcw` library conversion function for general based numbers has this outline:

```
fp_t
CVTIG(const char *s, char **endptr)
{
  /* ... code omitted ... */
}
```

Its code is lengthy, so we do not exhibit it here. There are similarities to other input conversion functions, but there are also complications because we want to handle exactly representable inputs exactly. That means separate handling of bases that are powers of two when the native base is a (possibly different) power of two, and of base-10 input when decimal floating-point arithmetic is in use. For other number bases, the conversions may require scaling by a power of that base, where the power may not be represented exactly in the native base, introducing additional rounding errors. The code uses the highest available precision to reduce those errors.

[2] On early UNIX systems, some input terminals had limited character sets, and it was necessary to usurp printing symbols for the line-cancel and erase characters: `@` and `#` were sacrificed.

27.6 General floating-point input

The functions described in the preceding sections provide the tools for writing a general floating-point input function that automatically recognizes based-number, binary, octal, decimal, and hexadecimal floating-point strings, as well as Infinity, and generic, quiet, and signaling NaNs.

All that is required is a small amount of lookahead to determine which specific conversion routine is needed, and the code should be reasonably obvious:

```
fp_t
CVTIA(const char *s, char **endptr)
{   /* convert any numeric input string to a number */
    char *g_endptr;
    const char *s_in;
    const char *s_val;
    fp_t result, value;
    int sign;

    s_in = s;

    while (isspace(*s))    /* skip optional leading space */
        ++s;

    sign = _cvtsgn(s, (char**)&s);
    s_val = s;

    if (isalpha(*s) && (value = CVTINF(s, &g_endptr), g_endptr != s_val))
        result = COPYSIGN(value, (fp_t)sign);
    else if (isalpha(*s) && (value = CVTNAN(s, &g_endptr), g_endptr != s_val))
        result = SET_EDOM(COPYSIGN(value, (fp_t)sign));
    else if (_cvtcmp("0x", s) == 0)
        result = CVTIH(s_in, &g_endptr);
    else if (_cvtcmp("0o", s) == 0)
        result = CVTIO(s_in, &g_endptr);
    else if (_cvtcmp("0b", s) == 0)
        result = CVTIB(s_in, &g_endptr);
    else                /* expect based or decimal number */
    {
        s = s_val;

        while (isdigit(*s))
            ++s;

        if ( (*s == '@') && (s > s_val) )
        {              /* probable based number */
            result = CVTIG(s_in, &g_endptr);

            if (g_endptr == s_in) /* expect decimal number */
                result = CVTID(s_in, &g_endptr);
        }
        else
            result = CVTID(s_in, &g_endptr);
    }

    if (endptr != (char **)NULL)
        *endptr = g_endptr;

    return (result);
}
```

If the input string contains neither an Infinity nor a NaN, then we try hexadecimal, octal, binary, based-number, and decimal strings. An at-sign after a digit suggests a based number, but if that conversion fails, we try a decimal conversion, so `"10@ibm.com"` is converted to 10 with an unprocessed suffix of `"@ibm.com"`.

The design of the code means that anywhere the `strtod()` family is used in existing software, the functions can be transparently replaced with the corresponding members of the `cvtia()` family, and that is easily done *without code changes* by using a compile-time macro definition. The older C library function `atof(s)` can be replaced by `cvtia(s, NULL)`. Those simple changes make it trivial to retrofit into older C programs support for input strings in based-number, binary, octal, and hexadecimal formats, in addition to decimal number strings, and allow recognition of Infinity, NaNs with payloads, and digit-separating underscores.

27.7 The scanf() family

The input-conversion functions described earlier in this chapter give us the tools needed to complete the job of supporting formatted input to programs in the C language family. The Standard C library functions in the `scanf()` family are the input companions to the `printf()` family that we described in **Section 26.12** on page 867.

The input functions have these prototypes:

```
#include <stdio.h>

int fscanf(FILE * restrict stream, const char * restrict format, ...);

int scanf(const char * restrict format, ...);

int sscanf(const char * restrict s, const char * restrict format, ...);

#include <stdarg.h>

int vfscanf(FILE * restrict stream, const char * restrict format, va_list arg);

int vscanf(const char * restrict format, va_list arg);

int vsscanf(const char * restrict s, const char * restrict format, va_list arg);
```

Those six functions are accompanied by six more that take string arguments of type `wchar_t`, but we do not treat those additional functions further in this book, because they are not widely supported, and rarely found in existing software.

As with the `printf()` family, there are two groups of functions. The first group is the more commonly used, and a call to the convenience function `scanf()` is equivalent to a call to `fscanf()` with an initial argument of `stdin`. The second group gives access to the formatted-input facilities from user functions with variable numbers of arguments.

All six functions return the number of arguments to which values have successfully been assigned, or `EOF` if an input failure occurs before any conversion. Input failures may reflect an absence of data from end-of-file or end-of-string conditions, or an error return from a file-input function. By contrast, a match failure *never* produces an `EOF` return value.

Because the return value indicates the degree of success, it is commonly used in software to control input processing. Here is a typical example:

```
char s[10 + 1];
float x;
int n;

/* ... code omitted ... */

while (scanf("%d %g %10c", &n, &x, &s) == 3)
    process(n,x,s);
```

Here, three items — an integer, a floating-point number, and a ten-character string — are read from `stdin` and processed. The loop terminates when three items can no longer be converted.

To `scanf()`, an input file is a continuous stream of data. If the input items are required on separate lines, the sample loop body must consume input until a newline is found. That could be done with a loop

```
do c = getchar(); while ( (c !='\n') && (c != EOF) );
```

but is more compactly handled with a scanset conversion:

```
(void)scanf("%*[^\n]");
```

Scansets are described later in **Section 27.7.3** on page 905.

27.7.1 Implementing the `scanf()` family

The functions listed in the previous section provide for input from either a file or a string, and we implement all of them with a common core. By analogy with the `sink_t` data structure that we used in **Section 26.12.3** on page 871, we introduce a `source_t` data structure:

```
typedef struct source_s
{
    FILE *stream;
    const char *s;
    int count;
    int next_empty_pushback;
    int width;
    size_t next_s;
    unsigned char pushback[MAXPUSHBACK];
} source_t;
```

That structure is more complex than we needed for the output problem, because we now have *three* sources of data: the argument file, the argument string, and an internal buffer of characters that were read, examined, and then pushed back into the input source. When that buffer is not empty, it takes precedence over file or string input.

The need for input pushback is common in parsers, because they often have to examine a few characters of pending input before deciding which of several processing alternatives applies. The easiest way to handle that lookahead is to read characters until a decision can be made, then push them back into the input stream, most recently read data first. Although Standard C provides the `ungetc()` function to do that for file input, it only guarantees a single character of pushback. For Infinity and NaN inputs, we require many more than one: our buffer size is set to 4096, conforming to the minimum limit required by C99 for the number of characters produced by a single formatted-output specifier.

The lookahead required for Infinity and NaN poses a potential problem of loss of data. For example, if the input stream contains `index` instead of `inf` or `infinity`, then at least three characters must be read before match failure is certain, and the three must then be pushed back into the input stream. As long as input processing continues inside a single call to `scanf()`, that is not a problem. However, just prior to return, `scanf()` must transfer its private pushback buffer to that of `ungetc()`, and if that function provides only a single-character buffer, data loss is certain. With NaN payloads, the potential for data loss is even greater. The only way to eliminate the problem is to rewrite the system-dependent implementation of input handling inside `<stdio.h>` on all platforms to which the `mathcw` library is ported, and that is impractical. Although the C library contains functions `ftell()` and `fseek()` to control input/output positioning, those functions fail on data streams that are not seekable, such as interactive files, shell pipes, and network connections. Thus, they cannot provide a solution to the lost-data problem. That appears to be a situation where the ramifications of a new feature in C99 were not fully considered, or at least, it was not imagined that separate implementations of high-level I/O might be provided by other libraries.

The `count` member of the `source_t` data structure records the number of input characters read from the data sources, because that value is needed for the `%n` format specifier. A specifier width sets the `width` member so that a maximum input field length can be enforced. The `next_s` member indexes the string member, `s`.

The three main input functions are wrappers that call functions from the second group:

```
int
(fscanf)(FILE * restrict stream, const char * restrict format, ...)
{
    int         item_count_or_eof;
    va_list     ap;

    va_start(ap, format);
    item_count_or_eof = vfscanf(stream, format, ap);
    va_end(ap);

    return (item_count_or_eof);
}

int
(scanf)(const char * restrict format, ...)
{
    int         item_count_or_eof;
    va_list     ap;

    va_start(ap, format);
    item_count_or_eof = vfscanf(stdin, format, ap);
    va_end(ap);

    return (item_count_or_eof);
}

int
(sscanf)(const char * restrict s, const char * restrict format, ...)
{
    int         item_count_or_eof;
    va_list     ap;

    va_start(ap, format);
    item_count_or_eof = vsscanf(s, format, ap);
    va_end(ap);

    return (item_count_or_eof);
}
```

User-defined functions analogous to the sample function that we showed in **Section 26.12.2** on page 870 can be written in much the same way as those functions.

The functions in the second group have even simpler definitions:

```
int
(vfscanf)(FILE * restrict stream, const char * restrict format, va_list arg)
{
    source_t  source;

    return (vscan(new_source_file(&source, stream), format, arg));
}

int
(vscanf)(const char * restrict format, va_list arg)
{
    source_t  source;

    return(vscan(new_source_file(&source, stdout), format, arg));
}

int
(vsscanf)(const char * restrict s,
            const char * restrict format, va_list arg)
{
    source_t  source;

    return (vscan(new_source_string(&source, s), format, arg));
}
```

The private functions `new_source_file()` and `new_source_string()` initialize the `source_t` structure, and the private function `vscan()` manages the real work of controlling format scanning and input conversion. It is assisted by a score of smaller functions with about two dozen `switch` statements and about 100 `case` statements, collectively amounting to about 1200 lines of code.

The code in `vscan()` is straightforward, but full of tedious details that we omit here. As with `vprt()`, its task is to work its way through the format string, finding format specifiers and converting them to a more convenient internal form, and matching them with the next unprocessed argument. However, because `vscan()` provides input, all of its arguments after the format are pointers, and thus, all of the dangers with `printf()` that we described in **Section 26.12.1** on page 868 apply here, but even more so, because *every* input conversion is a candidate for either buffer overrun, or writing data to arbitrary memory locations.

27.7.2 Whitespace and ordinary characters

A format string for input may contain whitespace characters (as defined by the Standard C function `isspace()`), ordinary printing characters, and format specifiers beginning with a percent character.

Whitespace characters in the format string are skipped, and except for `%c`, `%n`, and `%[...]` conversions, any whitespace characters occurring next in the input stream are read and ignored. It is *not* an error if the input has no whitespace characters, so that effectively means that *whitespace in format strings is purely for improved readability*.

Because carriage returns and newlines are also whitespace by the definition of the `isspace()` function, line breaks in the input stream have no significance, apart from separating input items. That is quite different from Fortran, where `READ` statements start with a new input record. It also differs from `printf()`, where a space or a newline in a format string is transmitted verbatim to the output stream.

When the scan of the format string meets an ordinary printing character, any input whitespace is skipped, and then the next input character is matched with the format character. If they differ, `vscan()` terminates further processing, and returns the number of items assigned so far.

Here is a small test program that can be used to see how whitespace and ordinary characters are handled:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int
main(void)
{
    int code;
    int n;

    (void)printf("Input data (a b c 123): ");

    while ((code = scanf(" a b c %d", &n), code) == 1)
    {
        (void)printf("n = %d\n", n);
        (void)printf("Input data: ");
    }

    (void)printf("Last return code from scanf() = %d\n", code);

    return (EXIT_SUCCESS);
}
```

On a UNIX system, we can compile and run the program like this:

```
% cc test.c && ./a.out
Input data (a b c 123): abc12345
n = 12345
Input data: a   b   c   456
n = 456
Input data: a
b
c
789
n = 789
Input data: A B C 123
Last return code from scanf() = 0
```

After the first prompt, the input `abc12345` matches the format, despite the absence of input whitespace. The loop body reports the value 12345 for `n`, and prompts for more input.

After the second prompt, the input `a␣␣␣b␣␣␣c␣␣␣456`, where ␣ indicates a space, also matches the format, even though this time, the input has additional whitespace. The loop body reports the value 456 and prompts again.

After the third prompt, the input is supplied on four successive lines, but still matches the format and produces the expected report.

However, at the last prompt, the input `A B C 123` fails to match the format, because `A` is not the same character as `a`. Thus, `scanf()` returns zero, and the loop terminates. A subsequent `getchar()` call would return the character `A`.

27.7.3 Input conversion specifiers

The format specifiers recognized by the `scanf()` family are summarized in **Table 27.1** on the next page, and their data type modifiers in **Table 27.2** on page 907. Although there are strong similarities with the `printf()` specifiers (see **Table 26.5** on page 873 and **Table 26.6** on page 874), there are significant differences as well.

Except for their data types, the integer conversions work as defined for `strtol()` and `strtoul()`, with their base argument set appropriately. The `%i` conversion corresponds to a base argument of zero in `strtol()`, where the input form of the number determines the base.

The floating-point conversion specifiers are equivalent, apart from their data types. In Standard C, they work like `strtod()`, recognizing decimal and hexadecimal floating-point values, as well as Infinity and NaN representations. In the `mathcw` library, floating-point conversions are generalized to work like `cvtia()`, so they also support based-number, binary, and octal floating-point values.

The *scanset* specifier, `%[...]`, requires some explanation. It normally takes the form of a bracketed list of characters, called the *scanlist*, and input matching continues until a character is found that is not in the scanlist. At least one input character must match, or else there is a match failure and an immediate function return. Thus, `%[aeiou]` matches a sequence of one or more lowercase English vowels. However, if the first character after the left bracket is a caret (^), the scanset contains all characters that are *not* in the scanlist. The specifier `%[^aeiou]` therefore matches an input sequence that does not contain a lowercase vowel. If the specifier begins with `%[]` or `%[^]`, the right bracket is part of the scanlist, rather than acting as a list delimiter. The specifier `%[][x]` matches an input sequence of any number of brackets or `x` characters.

Standard C leaves the interpretation of a hyphen in the scanlist up to the implementation. Historically, some treated a scanlist of `a-z` as a range specification, equivalent to the set of lowercase English letters in the ASCII character set, whereas others considered it to be a three-character list. In other character sets, such as EBCDIC used in some IBM mainframe operating systems, letters do not form a single contiguous block, so the range `a-z` also contains nonletters. Scanset ranges in `scanf()` family format conversions have never been portable, and are not supported by the `mathcw` library.

The `scanf()` `%n` conversion specifier causes the current number of input characters to be stored in the integer pointed to by the next argument, just as the same specifier in `printf()` reports the current output-character count, However, `%n` conversion is peculiar in that it results in an argument assignment, but does not increment the assignment count that the function ultimately returns. Standard C says that a field width or an assignment suppression in a `%n` conversion are implementation defined; in the `mathcw` library, a field width is silently ignored, and assignment is suppressed. Thus, `%*10n` serves merely as useless filler in a format string.

Table 27.1: Input conversion specifiers for the `scanf()` family.
Using brackets to indicate optional values, the general syntax of a conversion specifier is `%[*][width][datatype]letter`.
An asterisk suppresses assignment to an argument. Any width given must be greater than zero, and specifies the *maximum* field width in characters (or wide characters). The data type modifiers are defined in Table 27.2.
Specifiers in the second part of the table are extensions to C89 and C99.

Item	Description
`%%`	Literal percent character. No argument is assigned.
`%[...]`	Match a scanset; see text for a description. The argument must be as for `%s` conversion.
`%A`, `%a`	Optionally signed floating-point number, Infinity, or NaN, in the format accepted by `strtod()` (for `mathcw` library, `cvtia()`). Argument is pointer to floating point.
`%c`	Sequence of characters of exactly the number specified by the field width (default: 1). Argument is pointer to array of type `char` or `wchar_t` large enough for the sequence. No `NUL` is appended.
`%d`	Optionally signed decimal integer. Argument is pointer to signed integer.
`%E`, `%e`	Same as `%a`.
`%F`, `%f`	Same as `%a`.
`%G`, `%g`	Same as `%a`.
`%i`	Optionally signed integer in decimal or hexadecimal form. Argument is pointer to signed integer.
`%n`	Current input character count is assigned to argument, (pointer to signed integer). Does not contribute to count returned by the function.
`%o`	Optionally signed octal integer. Argument is pointer to unsigned integer.
`%p`	Pointer value in the format produced by `printf()` with `%p` conversion. Argument is pointer to `void`.
`%s`	Sequence of nonwhitespace characters. Argument is pointer to array of type `char`, or with the `l` (ell) modifier, `wchar_t`, large enough for the sequence, and a trailing `NUL`.
`%X`, `%x`	Optionally signed hexadecimal integer. Argument is pointer to unsigned integer.
`%u`	Optionally signed decimal integer. Argument is pointer to unsigned integer.
`%@`	Same as `%a` [hoc and `mathcw` library].
`%B`, `%b`	Same as `%a` [hoc and `mathcw` library].
`%Q`, `%q`	Same as `%a` [hoc and `mathcw` library].
`%Y`, `%y`	Optionally signed binary integer [hoc and `mathcw` library]. Argument is pointer to unsigned integer.

The `%n` specifier might seem to be rarely needed, but it serves an important, albeit unexpected, purpose. Consider the simple format `"x"`, which matches an ordinary character `x`. If one is found, the `scanf()` return value is zero, because there are no assignments. If no `x` is found, there is a match failure, and the return value is again zero. The two cases can easily be distinguished by using a `%n` specifier, as in this code fragment:

```
int n;

n = -1;
(void)scanf("x%n", &n);
(void)printf("%s\n", (n < 0) ? "missing x" : "found x");
```

Thus, judiciously placed `%n` specifiers can provide extra information about conversion success for ordinary characters that may not be obtainable from just the returned assignment count.

An asterisk serves a quite different purpose in conversion specifiers for the `printf()` and `scanf()` families. In `printf()`, it requests retrieval of an integer value from the next argument, whereas for `scanf()`, it specifies assignment suppression. Input matching and data conversion proceed normally, but the final assignment and incrementing of the return count are both prevented. For example, to read the third integer on each line of an input file, use something like this:

```
while (scanf("%*i %*i %i %*[^\n]", &n) == 1)
    process(n);
```

Table 27.2: Data type format modifiers for input conversion specifiers. Modifiers are required if the corresponding argument has a type other than the default for the conversion specifier.

Item	Description
	integer conversion
hh	`signed char *` or `signed char *` argument [C99]
h	`short int *` or `unsigned short int *` argument
j	`intmax_t *` or `uintmax_t *` argument [C99]
l	`long int *` or `unsigned long int *`, `wint_t *`, or `wchar_t *` argument
ll	`long long int *` or `unsigned long long int *` argument [C99]
t	`ptrdiff_t *` argument [C99]
z	`size_t *` argument [C99]
	binary floating-point conversion
l	`double *` argument [mathcw library]
L	`long double *` argument
LL	`long_long_double *` argument [mathcw library]
	decimal floating-point conversion
H	`decimal_float *` argument [mathcw library]
DD	`decimal_double *` argument [mathcw library]
DL	`decimal_long_double *` argument [mathcw library]
DLL	`decimal_long_long_double *` argument [mathcw library]

A field width following the percent or asterisk limits the scan to at most that many characters, and that maximum does not include any leading whitespace that was skipped. The primary use of field widths is for string conversions with `%c` and `%s`, and to prevent buffer overrun, those specifiers should *always* have field widths.

Unfortunately, `scanf()` format specifiers provide no mechanism for dynamic field widths, so that means that formats contain magic constants related to an array dimension elsewhere in the code, which is a recipe for a disaster if the dimension is later reduced. Careful programmers create such formats dynamically in strings declared nearby:

```
char s[100];
float x;
int n;
/* ... code omitted ... */
{
    char fmt[sizeof("%i %g %99999999999999999999s %*[^\\n]")];

    (void)snprintf(fmt, sizeof(fmt), "%%i %%g %%%zus %%*[^\\n]", sizeof(s));

    if (scanf(fmt, &n, &x, s) == 3)
       process(n, x, s);
}
```

The size of the `fmt` array is specified by a template, rather than yet another magic integer constant. The template shows what the format would look like in the worst case, and automatically includes space for the string-terminating `NUL` character. Although a field width as big as the 20-digit one shown would be impractical on most systems, that number of digits is possible from the conversion of a 64-bit unsigned integer, which is likely to be the largest size encountered for the `size_t` value produced by the `sizeof` operator on current systems.

The troublesome extra programming that is needed here for safety clearly indicates poor software design in the `scanf()` family.

The data type modifiers shown in Table 27.2 are frequently needed in `scanf()` format specifiers, because they control the size of the assigned arguments. For example, a snippet of the result-assignment code for signed integer conversion in `vscan()` looks like this:

```
switch (fmt.datatype)
{
case FMT_DATATYPE_CHAR:
    *va_arg(ap, char *) = (char)CLAMP(CHAR_MIN, number, CHAR_MAX);
    break;

case FMT_DATATYPE_SHORT:
    *va_arg(ap, short int *) = (short int)CLAMP(SHRT_MIN, number, SHRT_MAX);
    break;

case FMT_DATATYPE_PTRDIFF_T:
    *va_arg(ap, ptrdiff_t *) = (ptrdiff_t)CLAMP(PTRDIFF_MIN, number, PTRDIFF_MAX);
    break;

case FMT_DATATYPE_SIZE_T:
    *va_arg(ap, size_t *) = (size_t)CLAMP((size_t)0, number, SIZE_MAX);
    break;

    /* ... code omitted ... */
}
```

In each assignment, an argument pointer is coerced to a pointer to the data type matching that given in the format specifier, dereferenced, and assigned to. An erroneous type modifier may result in overwritten data in the caller, and it is *impossible* for `vscan()` to detect that error. Similarly, a missing argument results in writing to an unpredictable memory location, with serious implications for correctness, reliability, and security. That error too is impossible for `vscan()` to guard against in any portable fashion. Utilities like `lint` and `splint`, and some C compilers, perhaps with additional options, can detect both kinds of errors in source code.

The variable `number` holds the result of the conversion, and has the longest supported integer type. The macro `CLAMP(low, number, high)` ensures that the stored result lies in [`low`, `high`]. The design of the `scanf()` family provides no way to warn its caller of an intermediate overflow, not even through the usual global variable `errno`, so it seems most sensible to return an extreme value, rather than an arbitrary result whose decimal representation gives no hint that it arises from bit truncation of the input number.

The C Standards specify that the behavior of integer type-narrowing is undefined when the value to be converted lies outside the range of the target type. Common compiler practice, and CPU instruction-set design, extract as many of the low-order bits of the longer type as are needed to fill the shorter type. However, variations in the size of `long int`, and whether `char` is signed or unsigned, lead to differences across platforms, and between compilers. As we discussed in **Section 4.10** on page 72, integer overflow requires careful handling by software, and is usually best treated by preventing it altogether, as we do in `vscan()` with the `CLAMP()` macro. Few other implementations of the `scanf()` family provide such protection against overflow in input conversions.

The conversion from a longer to a shorter integer type is a convenient implementation choice that simplifies the code in `vscan()`. However, separate conversions are needed for each floating-point data type to avoid the problems of double rounding, loss of NaN payloads, and the distinction between quiet and signaling NaNs.

The `scanf()` `%p` specifier is guaranteed to read a pointer value output by the `printf()` family with `%p`, and provided that the input and output are in the same process, the pointers refer to the same memory location. Pointers exchanged that way between different processes are unlikely to be meaningful. On modern systems, pointers are normally expressed in hexadecimal with a `0x` suffix. On older architectures, such as the PDP-10 and PDP-11, octal pointers are conventional.

There are glaring differences between the `printf()` and `scanf()` families in type modifiers for floating-point conversions. In `printf()`, a `%g` conversion consumes a `double` argument, and instead of adding a new type specifier for `float` arguments, C89 and C99 instead require the compiler to promote a `float` argument to `double` in functions with variable numbers of arguments, when the argument is represented in the function prototype by the ellipsis. By contrast, in `scanf()`, `%g` conversion assigns to a `float` value, and `%lg` to a `double` value. Arguments of type `long double` are handled with `%Lg` conversion in both `printf()` and `scanf()`.

In `printf()`, a `%10c` conversion outputs a single `unsigned char` value that is pushed on the argument list as a fullword integer, and output right justified in a field width of 10 characters. By contrast, the same conversion in `scanf()` reads the next 10 characters from the input stream into a character array, *without supplying a terminating* `NUL`.

Similarly, in `printf()`, a `%10s` conversion outputs a string, possibly containing whitespace, right-justified in a 10-character field, expanding the field automatically if the string is long. In `scanf()`, that same conversion skips whitespace, then reads up to 10 nonwhitespace characters, and stores them in a character array *with a terminating*

NUL. That array must therefore contain at least 11 elements. Any whitespace encountered during the string collection terminates the scan.

The differences between input and output processing with %c and %s specifiers mean that %c is likely to be commonly needed for input, although remaining rare in output, and also that additional code is almost always needed to add a string terminator. Even worse, if the input stream does not contain the number of characters specified in the field width, some characters of the argument array will not be assigned to, so it is *not* sufficient to store a NUL at the end of the array. In practice, then, programs should first initialize input character arrays with NUL characters, with code like this:

```
char name[50 + 1];

(void)memset(name, 0, sizeof(name));

if (scanf("%50c", name) == 1)
    process(name);
```

The character array name is then guaranteed to be properly terminated, and can safely be used as a C string.

27.7.4 Retrospective on the scanf() family

Examination of almost any significant body of C code shows that the printf() family is widely used, whereas the scanf() family is rarely seen. Part of that is certainly due to the application areas where C is commonly used, notably, for text processing. Input handling may then be based on simple getchar() calls, or entire lines might be read with fgets(), and then split into input fields, possibly with the assistance of the standard tokenizer functions, strcspn(), strpbrk(), strsep(), strspn(), and strtok(). Numeric fields can then be converted to internal form with functions in the strtod() and strtol() families.

In window-system applications, the software design usually ensures that each input string contains only a single value of known type. In database and network programming, input data is already likely to be in internal form, or requires only the conversion of a text string to a single value.

Scientific and engineering computing applications are likely to be ones in which software must deal with large input files containing a mixture of integers, floating-point values, and character strings. Use of the scanf() family may then be a reasonable way to solve the input problem. However, as we noted in earlier sections, great care is needed to avoid confusion between conversion specifiers that are similar to ones for printf(), but behave differently. Because all arguments after the format are pointers, even more care is needed to avoid overwriting memory due to erroneous data-type modifiers, incorrect or omitted maximum field widths for string conversions, and missing arguments.

In addition, the program must vigorously defend against input errors that are caused by missing or incorrectly formatted data. It is imperative to check that the function return values match the expected number of conversions, and to include a reasonable error-reporting and recovery mechanism, so that most of the input can be checked on a single run, instead of just terminating at the first conversion error.

Interactive programs may find it helpful to prompt the user for input, and retry with a fresh prompt when conversion fails.

The fixed field widths of 80-character punched cards should now be just a historical curiosity, and any input that is prepared by humans should be insensitive to data widths, horizontal spacing, and perhaps even to line breaks.

It is also helpful to users if input data files can contain comments. For example, if a sharp sign marks an optional comment that continues to end of line, then after collecting data values from a line, the comment can easily be consumed with simple formats in two calls like these:

```
(void)scanf("#");        /* gobble start of comment, if any */
(void)scanf("%*[^\n]");  /* eat rest of input line */
```

Most UNIX utilities that support configuration files permit such comments, and the practice deserves to be more widely employed in user programs as well.

Some UNIX configuration files, and the Fortran NAMELIST facility, allow variable assignments in the input stream. A simplified version of a scanner for such data can be implemented with a loop like this:

```
char name[100 + 1];
long double value;

initialize();

while (scanf("%100s = %Lg", name, &value) == 2)
    install(name, value);
```

That has several advantages for humans:

- If the program initializes all of the input variables with suitable defaults, most uses of the program require only the specification of a few nondefault values.
- Values are associated with variable names, rather than with positions in the input stream.
- Assignments can be done in any convenient order, rather than the sequence demanded by the program.
- Modifications to input values need not destroy previous values, because a new assignment added to the end of the input block overrides earlier ones.
- Input data can be self documenting, especially if comments are permitted in the input stream.
- Suitable whitespace in the form of indentation and line breaks can make the input data easier for a human to prepare and read.

27.8 Summary

Input conversion is astonishingly complex, and robust code for that task is intricate, lengthy, and difficult to write correctly. The input facilities provided by the functions described in this chapter are powerful, and provide capabilities well beyond those of most existing program languages and libraries.

Our design carefully arranges for many of the routines to have the same programming interface as the venerable `strtod()` routine in the C library, allowing existing code to be trivially recompiled to use `cvtid()`, or even better, `cvtia()`, instead.

The mathcw library implementation of the `scanf()` family handles all of the formatted-input requirements of the C Standards, albeit without the little-used companions for wide characters, as well as allowing more number formats, and digit-grouping underscores for improved readability of long digit strings.

Implementations of the C string-to-integer conversion functions `strtol()`, `strtoll()`, `strtoul()`, and `strtoull()` in the mathcw library extend their standard features with support for binary and octal prefixes, based numbers, and digit grouping.

All of our integer input-conversion code correctly handles the most negative number in two's-complement arithmetic, something that most other code fails to do.

Our analysis of the specification in the C Standards of `scanf()` reveals its poor design, and its confusing lack of format compatibility with the widely used `printf()` family. The `scanf()` routines are unable to detect programming errors of inadequate sizes of string buffers, argument types that fail to match format items, and argument counts that differ from format-item counts. Buffer overruns and pointer misuse continue to be a major source of security holes and other failures of commercial and open-source software, including that inside operating systems, device drivers, and code that is embedded in electronic consumer products. Nevertheless, when used with care, `scanf()` can be used to simplify input processing for the programmer, and more importantly, can make input-file preparation easier for humans.

Input programs should be designed for user, rather than programmer, convenience. Our discussion of the Fortran `NAMELIST` facility showed that with little programming effort, input files can contain assignments of values to variable names in any order that proves convenient for the user, and can have blank lines and comments sprinkled through the input stream.

A Ada interface

AUGUSTA ADA (1815–1852),
COUNTESS OF LOVELACE AND DAUGHTER OF LORD BYRON,
ASSISTANT AND BIOGRAPHER TO CHARLES BABBAGE,
AND THE WORLD'S FIRST COMPUTER PROGRAMMER.

The Ada programming language [Ada95] defines a standard interface to routines written in other languages, and the GNU Ada translator, `gnat`, supports that interface.

The standard Ada floating-point data types `Float`, `Long_Float`, and `Long_Long_Float` correspond directly to the C data types `float`, `double`, and `long double`. The Ada integer data types `Integer`, `Long_Integer`, and `Long_Long_Integer` are defined to be 32-bit, 32-bit, and 64-bit signed values, respectively, by the `gnat` compiler.

At the time of writing this, this author has access only to one Ada compiler family, `gnat`, running on IA-32, IA-64, and AMD64 systems, so interplatform portability of the Ada interface to the mathcw library cannot yet be extensively checked. Nevertheless, any changes needed for other platforms are expected to be relatively small, and related to the mapping of integer types.

Ada is a complex and strongly typed language, with a large collection of packages, several of which must be imported into Ada programs in order to use library functions. It is traditional in Ada to separate the *specification* of a package from its *implementation*, and with the `gnat` compiler, the two parts are stored in separate files with extensions `.ads` (Ada specification) and `.adb` (Ada body). For example, commercial software vendors can supply the specification source files and implementation object library files to customers, without revealing the implementation source code. Ada specification files therefore are similar in purpose to C header files.

The interface to the mathcw library is defined as an Ada package specification in the file `mathcw.ads`. Because the implementation is in the C language, there is no need for a separate body file.

A.1 Building the Ada interface

Before looking at the source code of the Ada interface, it is useful to see how it is built and used in a small test program.

Initially, we have just three hand-coded files in the `ada` subdirectory:

```
% cd ada
% ls
Makefile  mathcw.ads  test.adb
```

The `Makefile` contains the rules for the build process, the second file contains the Ada specification, and the last file contains the test program.

Ada is unusual in that it is a *requirement* of the language that the compiler manage source-file dependencies, so that it is impossible to compile and link a program with interface inconsistencies. The GNU `gnatmake` tool satisfies that mandate, managing compilation of files in the correct order, and then linking them into an executable program. That tool requires a lengthy list of options that are recorded in the `Makefile`. They include options that supply the installed location of the mathcw library, which can be either a static library, or a shared library.

The `make` utility manages the build, which looks like this on a GNU/LINUX system:

```
% make test

gnatmake -gnaty test.adb -L/usr/local/lib -aO/usr/local/lib -largs -lm -Xlinker -R/usr/local/lib -lmcw

gcc -c -gnaty test.adb
```

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2

```
gcc -c -gnaty mathcw.ads

gnatbind -aO./ -aO/usr/local/lib -I- -x test.ali

gnatlink -L/usr/local/lib -lm -Xlinker -R/usr/local/lib -lmcw test.ali
```

Only the `gnatmake` command is specified in the `Makefile`. That program examines the source-code file, `test.adb`, and determines that both that file and the `mathcw.ads` file must be compiled by `gcc`, which it then automatically invokes. Next, it runs `gnatbind` to perform various required consistency checks. Finally, it runs `gnatlink` to link the object files and libraries recorded in the `test.ali` file (output by `gcc`) into the target executable program, `test`.

After the `make` steps, the `ls` utility displays a list of all of the files in the directory:

```
% ls
Makefile    mathcw.ali  test      test.ali
mathcw.ads  mathcw.o    test.adb  test.o
```

Sites that use Ada extensively are likely to have a standard set of directories in which to store interface files, such as `mathcw.ads`, and in such a case, that file would be installed in one of those directories. However, there are no widely used conventions for the names of directories for locally installed Ada packages, so the `Makefile` cannot provide a system-independent `install` target. For simplicity, here we just store the interface and the test program in the same directory.

The final step is to run the test program:

```
% ./test

Test of Ada interface to MathCW library

x            MathCW.erff ()  MathCW.erfcf ()
-4.00E+00    -1.00000E+00     2.00000E+00
-3.75E+00    -1.00000E+00     2.00000E+00
...
 3.75E+00     1.00000E+00     1.13727E-07
 4.00E+00     1.00000E+00     1.54173E-08

x         MathCW.erf ()         MathCW.erfc ()
-4.00E+00 -9.99999984582742E-01  1.99999998458274E+00
-3.75E+00 -9.99999886272743E-01  1.99999988627274E+00
...
 3.75E+00  9.99999886272743E-01  1.13727256569797E-07
 4.00E+00  9.99999984582742E-01  1.54172579002800E-08

x         MathCW.erfl ()            MathCW.erfcl ()
-4.00E+00 -9.99999984582742100E-01  1.99999998458274210E+00
-3.75E+00 -9.99999886272743430E-01  1.99999988627274343E+00
...
 3.75E+00  9.99999886272743430E-01  1.13727256569796653E-07
 4.00E+00  9.99999984582742100E-01  1.54172579002800189E-08
```

Here, we trimmed trailing zeros from the reported `x` values to make the output fit the page.

A.2 Programming the Ada interface

The public part of the interface specification in `mathcw.ads` begins like this, introducing four mathematical constants that are also present in the standard Ada numerics package:

```
with Interfaces.C;

package MathCW is
```

```
PI       : constant := 3.14159_26535_89793_23846_26433_83279_50288_41971;

PI_2     : constant := PI / 2.0;

Sqrt_Two : constant := 1.41421_35623_73095_04880_16887_24209_69807_85696;

Log_Two  : constant := 0.69314_71805_59945_30941_72321_21458_17656_80755;
```

The constants are specified with sufficient precision for 128-bit floating-point arithmetic, and illustrate the useful Ada practice of separating digit groups with underscores to improve readability. That feature is sadly lacking from almost all other programming languages, and is trivial to implement in a compiler. The constants have no declared type; instead, they are converted as needed with a type wrapper, such as `Float (PI)`.

The constants are followed by a large block of function declarations that define the names, argument types, and return values of all of the functions in the mathcw library that can be accessed from Ada programs:

```
-- ================ Standard MathCW names ================

function Acosf  (X : Float)                        return Float;
function Acoshf (X : Float)                        return Float;
function Adxf   (X : Float; N : Integer)           return Float;
...
function Urcw3f                                    return Float;
function Urcw4f                                    return Float;
function Urcwf                                     return Float;
...
function Acos   (X : Long_Float)                   return Long_Float;
function Acosh  (X : Long_Float)                   return Long_Float;
function Adx    (X : Long_Float; N : Integer)      return Long_Float;
...
function Urcw3                                     return Long_Float;
function Urcw4                                     return Long_Float;
function Urcw                                      return Long_Float;

function Acosl  (X : Long_Long_Float)              return Long_Long_Float;
function Acoshl (X : Long_Long_Float)              return Long_Long_Float;
function Adxl   (X : Long_Long_Float; N : Integer) return Long_Long_Float;
...
function Urcw3l                                    return Long_Long_Float;
function Urcw4l                                    return Long_Long_Float;
function Urcwl                                     return Long_Long_Float;
```

Notice that Ada functions without arguments lack the empty parentheses that are required in C.

Next, the interface defines synonyms with the traditional names used in the Ada numerics library:

```
-- ======= Standard Ada names with C-like suffixes =======

function Arccosf (X : Float)                       return Float;
function Arcsinf (X : Float)                       return Float;
function Arctanf (X : Float)                       return Float;
...
```

Unfortunately, the match between Ada and C is imperfect in several areas:

- The source code of Ada programs is *case insensitive*. The Ada convention for spelling is that keywords are lowercase, function names are capitalized, and variables are uppercase. The `gnatmake` option `-gnaty` that we use requests fastidious style checking so that deviations from those conventions can be identified and corrected. One possibly surprising rule is that a function name must be separated from its parenthesized argument list by at least one space, contrary to centuries-old mathematical practice.

- Ada has a power operator, `**`, but C does not.
- Ada treats the ceiling, floor, and round operations as type properties, rather than as functions, although their use is quite similar. For example, the Ada code `Float'Ceiling (X)` corresponds to the C code `ceilf(x)`.

  Similarly, `Float'Rounding (X)` corresponds to `roundf(x)`: both round halfway cases away from zero, independent of the current rounding direction.

  Ada's `Float'Unbiased_Rounding (X)` rounds halfway cases to the nearest even integer, like C's `rintf(x)` when the rounding direction is set to the IEEE 754 default.
- The Ada absolute-value operation uses a function-like operator, `Abs (x)`.
- Ada provides generic names for the elementary functions. For example, as long as the appropriate library packages are included, the call `Sqrt (x)` selects a version of the square-root function that is appropriate for the type of the argument expression.
- Functions that return additional values by storing values into locations given by pointer arguments, such as C's `frexpf(x,&n)`, have no direct analogue in Ada. However, they are easily handled with arguments declared with the `in out` keyword pair, indicating that the arguments are assigned values in the function, and those new values are visible to the caller on return from the function.
- Ada provides no access to the IEEE 754 exception flags, precision control, or rounding modes. Indeed, there is no mention of IEEE 754, and only three brief references to the IEC 60559:1989 Floating-Point Standard, in the *Ada 95 Reference Manual*, along with one remark about a possible future Ada binding to that standard.

It is unclear whether it is possible to offer a generic interface to the `mathcw` library, so the current interface in `mathcw.ads` does not attempt to do so.

The private part of the interface specification in `mathcw.ads` comes next:

```
private
   pragma Import (C, Acosf,   "acosf");
   pragma Import (C, Acoshf,  "acoshf");
   ...
   pragma Import (C, Urcw4f,  "urcw4f");
   pragma Import (C, Urcwf,   "urcwf");

   pragma Import (C, Acos,    "acos");
   pragma Import (C, Acosh,   "acosh");
   ...
   pragma Import (C, Urcw4,   "urcw4");
   pragma Import (C, Urcw,    "urcw");

   pragma Import (C, Acoshl,  "acoshl");
   pragma Import (C, Acosl,   "acosl");
   ...
   pragma Import (C, Urcw4l,  "urcw4l");
   pragma Import (C, Urcwl,   "urcwl");

   pragma Import (C, Arccosf, "acosf");
   pragma Import (C, Arcsinf, "asinf");
   ...
```

The `pragma` statements tell the Ada compiler what language the external routines are written in, and give their names in Ada and in the other language.

The remainder of the interface specification in `mathcw.ads` provides hints to the compiler about inlining, and tells which functions are known to be *pure*, that is, produce the same result on successive calls with the same argument:

```
   pragma Inline (Acosf);
   pragma Inline (Acoshf);
   ...
   pragma Pure_Function (Acosf);
   pragma Pure_Function (Acoshf);
   ...
end MathCW;
```

Because they produce different results on each call, the random-number generator routines are excluded from the declarations of pure functions.

A.3 Using the Ada interface

The test program, `test.adb`, used on page 912 demonstrates the use of the library interface, mathcw, for Ada. As usual in Ada programs, it begins with a list of required packages:

```
with Ada.Text_IO;
use  Ada.Text_IO;

with Ada.Float_Text_IO;
use  Ada.Float_Text_IO;

with Ada.Long_Float_Text_IO;
use  Ada.Long_Float_Text_IO;

with Ada.Long_Long_Float_Text_IO;
use  Ada.Long_Long_Float_Text_IO;

with MathCW;
use MathCW;
```

The `with` statement imports the package, and the `use` statement tells the compiler that the long qualifying package names can be omitted from invocations of the package routines.

Four standard Ada packages are needed for output statements in the test program, and the `gnat` system knows how to find them all. By `gnat` convention, the mathcw package is known to be found in a file with a lowercase name matching the package name, followed by an extension for an Ada specification file: `mathcw.ads`.

Next comes the body of the test program, although here we omit code from the two sections that handle the double- and extended-precision tests, because they are similar to the single-precision code:

```
procedure Test is
   X : Float;
   Y : Long_Float;
   Z : Long_Long_Float;

begin
   New_Line;
   Put ("Test of Ada interface to MathCW library");
   New_Line;
   New_Line;

   Put ("x");
   Put (ASCII.HT);
   Put (ASCII.HT);
   Put ("MathCW.erff ()");
   Put (ASCII.HT);
   Put ("MathCW.erfcf ()");
   New_Line;

   X := -4.0;
   while X <= 4.0 loop

      Put (X);
      Put (ASCII.HT);
      Put (Erff (X));
      Put (ASCII.HT);
      Put (erfcf (X));
      New_Line;

      X := X + 0.25;
   end loop;

   -- ... code omitted ...
end Test;
```

The verbose output code is typical of Ada, which overloads a single function, `Put()`, for text output. Consequently, a separate implementation of that function is needed for each data type. That in turn accounts for the sequence of `with` statements at the beginning of the file.

A second test program, `perf.adb`, in the mathcw package carries out performance comparisons between the Ada and C versions of some the library functions. It too can be built and run under control of the UNIX `make` utility:

```
% make check-perf
...
Performance test of Ada interface to MathCW library

Average wall-clock time for 2000000 calls to Ada.Abs()     =   9 nsec
Average wall-clock time for 2000000 calls to MathCW.Fabs() =  21 nsec
Performance: (MathCW.Fabs() time) / (Ada.Abs() time) =  2.49372E+00
...
Average wall-clock time for 2000000 calls to Ada.Tanh()    = 151 nsec
Average wall-clock time for 2000000 calls to MathCW.Tanh() = 186 nsec
Performance: (MathCW.Tanh() time) / (Ada.Tanh() time) =  1.22695E+00

Cumulative performance summary

Average wall-clock time for 2000000 calls to Ada.ANY()    = 148 nsec
Average wall-clock time for 2000000 calls to MathCW.ANY() = 112 nsec
Performance: (MathCW.ANY() time) / (Ada.ANY() time) =  7.59194E-01
```

The relative performance is, of course, expected to vary significantly across platforms, particularly because the functions have quite different implementations in the two languages, but it is evident that there is no significant overhead from the interlanguage communication. The full benchmark report shows that the mathcw functions are usually faster than the Ada library ones, and the summary shows a 25% average improvement over the Ada library on that system.

B C# interface

THE C# NAME WAS MUSICALLY INSPIRED. IT IS A C-STYLE
LANGUAGE THAT IS A STEP ABOVE C/C++,
WHERE SHARP (#) MEANS A SEMI-TONE ABOVE THE NOTE.

— JAMES KOVACS (2007).

The C# programming language[1] [C#03b, HWG04, ECM06a, HWG06, C#06a, JPS07, HTWG08, HTWG11] is designed to run on top of a standardized virtual machine called the *Common Language Infrastructure (CLI)* [CLI03, CLI05, CLI06, ECM06b, MR04]. Microsoft's implementation of that virtual machine is part of their .NET Framework, and several other languages also share the *Common Language Runtime (CLR)* library environment built on top of the CLI virtual machine [AHG$^+$04].

That design has several important benefits:

- Interlanguage communication is greatly facilitated. Programmers can use the language that works best for a given task, without losing access to software written in every other language that runs in the CLR.
- The CLR provides a uniform run-time library and operating system interface for all languages, eliminating the duplication that has historically plagued all programming languages.
- The virtual machine layer between most software and the underlying hardware makes migration to other, and future, hardware platforms trivial once the CLI has been ported. Long experience in the computing industry shows that software usually outlives the hardware on which it was developed, so hardware independence is often an important software design goal.
- Because a substantial part of any modern operating system is largely independent of the details of the underlying hardware, the virtual machine layer can be moved from its current position between user software and the operating system, to between the operating system and the hardware. That can make almost the entire operating system independent of hardware. Only device drivers need to have knowledge of hardware details, and once devices are themselves standardized and virtualized, that last shred of hardware dependence can also be eliminated.

Just as the Java Virtual Machine helped to make a uniform Java programming environment widely available on most popular operating systems and hardware platforms, it seems likely that many programming languages will be available on the CLI. An interface to the mathcw library from the CLR then makes the library immediately available in *every* language that can be compiled for the CLI virtual machine.

The DotGNU Project[2] and the Mono Project[3] are free-software reimplementations of the CLR and the CLI virtual machine. The C# interface described in this appendix was developed on installations of the Mono system on multiple operating systems and CPU architectures, including Microsoft WINDOWS on IA-32 systems. The C# interface was also tested in installations of the DotGNU system on several platforms.

B.1 C# on the CLI virtual machine

The *Common Language Infrastructure (CLI)* virtual machine offers support for 32-bit and 64-bit two's-complement integers, and 32-bit and 64-bit floating-point values. CLI implementations can offer either 32-bit or 64-bit byte addressing, and all binary values are stored in little-endian format.

[1] There is an extensive bibliography of publications about C# at http://www.math.utah.edu/pub/tex/bib/index-table-c.html#csharp.

[2] See http://www.gnu.org/projects/dotgnu/.

[3] See http://www.mono-project.com/.

© Springer International Publishing AG 2017
N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2

The CLI provides integer arithmetic instructions that silently wrap around on overflow, and a companion set that trap on overflow. Integer arithmetic in the C# language follows most other languages in defaulting to undetected overflow, but provision is made for checking for, and recovering from, integer overflow. Here is a code fragment that illustrates how an overflow can be caught and handled:

```
try
{
    checked { i = j + k; };
}
catch (System.OverflowException)
{
    Console.WriteLine("OverflowException: last i = " + i);
    i = 0x7fffffff;
}
```

Unfortunately, the floating-point architecture defined in the CLI specification [ECM06b, pages I-67ff] is a subset of the requirements of IEEE 754. Here are the principal features of the CLI floating-point design:

- NaN and Infinity are fully supported, and computation is always nonstop, unless use is made of the CLI instruction `ckfinite`, which generates an exception if the result is NaN or Infinity. However, C# provides no way to execute that instruction after every floating-point operation. The `try/checked/catch` approach works for integer arithmetic, but not for floating-point arithmetic.
- Whether subnormals are supported or not is unspecified. Current implementations *do* provide them. In the language of the CLI specification: *This standard does not specify the behavior of arithmetic operations on denormalized* [subnormal] *floating-point numbers, nor does it specify when or whether such representations should be created.*
- There are no provisions for trapping floating-point exceptions, just as most current hardware architectures cannot trap exceptions either.
- There is only one rounding mode, the IEEE 754 default of *round to nearest*, with ties rounded to the closest even value.
- Conversion from floating-point to integer values truncates toward zero, but the *Common Language Runtime (CLR)* library environment provides additional rounding directions.
- There are no sticky flags that record floating-point exceptions.
- The CLI is permitted to carry out intermediate floating-point computations in a precision and range that is greater than that of the operands.
- The bit patterns in NaNs, and the result of converting between `float` and `double` NaNs, are defined to be platform dependent.
- There is only one type of NaN, not the separate quiet and signaling NaNs required by the IEEE 754 Standard.

Those choices were undoubtedly made to allow the CLI to be implemented efficiently on a wide range of current CPU architectures. Regrettably, they also force the many into perpetual suffering for the sins of the few, because the CLI will long outlive all current hardware. For floating-point software, it would have been much better to rectify the design flaws of past systems with an arithmetic system in the CLI that is strictly conformant with IEEE 754, and equipped with full support for the 128-bit floating-point format as well.

B.2 Building the C# interface

Before we look at the details of the C# interface code, it is helpful to see how the interface to the `mathcw` library is built.

Initially, we have just a few files in the `csharp` directory:

```
% cd csharp

% ls
Makefile  mathcw.cs  test01.cs  test03.cs  test05.cs  test07.cs
README    okay       test02.cs  test04.cs  test06.cs  test08.cs
```

The Makefile contains the rules for the build process, and the mathcw.cs file contains the interface definitions. The test*.cs source files are simple tests of some of the library routines.

The Mono C# compilers are unusual in that they do not generate object files, but instead, compile and link directly to executable programs, called *assemblies* in C# documentation. By contrast, the DotGNU compiler, cscc, generates normal object files.

The make utility directs the build, here using one of two compilers from the Mono Project (the other, older, compiler is called mcs):

```
% make
gmcs test01.cs
gmcs test02.cs
gmcs test03.cs
gmcs test04.cs
gmcs test05.cs
gmcs test06.cs mathcw.cs
gmcs test07.cs mathcw.cs
gmcs test08.cs mathcw.cs
```

The first five tests have internal definitions of a small interface to the mathcw library, so their compilation does not require the mathcw.cs file.

Notice that none of the compilation steps specifies the name or location of the mathcw library. The interface code specifies its name, but the compiler and linker do not access the library at all. Library search rules allow the library to be located at run time, and the needed routines are then dynamically loaded from the library on the first reference to each of them.

After the build, we have these files:

```
% ls
Makefile   test01.cs   test03.cs   test05.cs   test07.cs
README     test01.exe  test03.exe  test05.exe  test07.exe
mathcw.cs  test02.cs   test04.cs   test06.cs   test08.cs
okay       test02.exe  test04.exe  test06.exe  test08.exe
```

The executable programs have the file extension .exe, even on Unix systems. They cannot be run in isolation, but instead, as with Java programs, must be run on the virtual machine.

A simple validation test runs each program, comparing its output with correct output stored in the okay subdirectory:

```
% make check

There should be no differences reported, just the test names:

============== test01.exe
...
============== test08.exe
```

The command that runs each test looks something like this:

```
env LD_LIBRARY_PATH=.. mono test01.exe
```

The CLI virtual machine is the mono program on that system.

The library path that is temporarily set for the duration of the command informs the run-time loader that the shared object library is found in the parent directory, instead of in the normal system library directories.

B.3 Programming the C# interface

The C# interface to the mathcw library is stored in the file mathcw.cs, and that file must be compiled with any code that needs it, as illustrated by the test program compilations on page 919. There no conventions yet of where such shared code could be stored so as to be available to every C# programmer on the system, so we assume that programmers will simply keep private copies.

The interface defines function prototypes with instructions of where to find them, and how to call them. To do so, it needs language features that are defined in two standard namespaces, so the file begins with using statements for them:

```
using System;
using System.Runtime.InteropServices;
```

As in the C++ and Java interfaces, the function prototypes are defined in a class whose name qualifies the function names when they are used. The functions themselves are loaded dynamically from a shared library when they are first referenced at run time, so that class begins with a definition of the name of the library:

```
public sealed class MathCW
{
    private const String LIBNAME = "libmcw";
```

The sealed modifier prevents other classes from being derived from that class; a similar restriction is used in the Java interface. Only the base name of the library is given here for the value of LIBNAME. At run time, a suitable system-dependent suffix is added to it: .dll on Microsoft WINDOWS, .so on most UNIX systems, or .dylib on MAC OS X systems.

At the time of writing this, the Mono Project on MAC OS X incorrectly uses .so as the suffix, so the shared library is installed under two names on that platform, libmcw.dylib and libmcw.so.

On UNIX and MAC OS X systems, it is conventional for shared libraries to carry version numbers, with a symbolic link from the latest version to a library filename without a number. That is important, because changes to a shared library could invalidate executables that were already linked to a previous version of the library. The inability to deal with that on current Microsoft WINDOWS systems is widely referred to as the *DLL hell*, where installation of a new software package breaks already-installed and completely unrelated packages by replacing a shared library with a new one with the same name, but incompatible contents.

With other programming languages on those systems, the linker can record in the executable program the specific library versions required. In the Mono Project, the linker does not read the shared libraries, because they are not needed until run time. However, it is possible to bind a particular library version number to a program with a small manually created configuration file. For example, to indicate that myprog.exe requires version 1.2.3 of the mathcw library, the companion file myprog.exe.config would contain these three lines:

```
<configuration>
  <dllmap dll="libmcw" target="libmcw.so.1.2.3" />
</configuration>
```

In practice, that should rarely be necessary, because the mathcw library is based on an ISO standard, and is expected to be stable. Existing function prototypes are guaranteed to be constant, but new functions might be added in future versions of the library. Such additions do not invalidate any existing programs linked against the shared library.

The mathcw.cs file continues with the definitions of two mathematical constants that are also defined in the standard C# Math class:

```
    public const double E       = 2.718281828459045235360287;
    public const double PI      = 3.141592653589793238462643;
```

Next comes a long list of function prototypes for the single-precision functions:

```
    [ DllImport(LIBNAME) ] public static extern float acosf (float x);
    [ DllImport(LIBNAME) ] public static extern float acoshf(float x);
    ...
    [ DllImport(LIBNAME) ] public static extern float frexpf (float x, ref int n);
    ...
```

```
[ DllImport(LIBNAME, CharSet=CharSet.Ansi) ] public static extern float nanf (string s);
...
[ DllImport(LIBNAME) ] public static extern float urcw3f();
[ DllImport(LIBNAME) ] public static extern float urcw4f();
```

In each prototype, the square brackets contain a C# *attribute list*, a feature rarely seen in other programming languages. The `DllImport()` function is defined in the `System.Runtime.InteropServices` namespace. In the common case, it has only a single argument that provides the library name. However, the `mathcw` library contains a few functions with arguments that are pointers to numeric values or strings, and for them, the attribute list is more complicated.

Unlike Java, the C# language has pointers, but with restrictions. The C-style `int *n` declarations and `&n` arguments are supported, but only when the construct is qualified with the `unsafe` modifier. When that feature is used, the Mono Project compilers require an additional option, `-unsafe`. The intent is to make clear to the programmer and the user that danger lurks.

However, there is another way to deal with pointer arguments that does not require the `unsafe` modifier, and that is to identify a *call-by-reference* argument with either the `ref` modifier or the `out` modifier, which must be used in the prototype, as well as in every call to the function. The two modifiers are quite similar, except that `ref` requires initialization before use. The standard C# `Math` class uses `out` modifiers, so we do as well to guarantee identical calling sequences for the functions common to both classes.

In reality, a called routine written in another language could still wreak havoc on its caller by overwriting memory before and after the passed location, but a native C# program cannot.

C# and Java both use the Unicode character set in UTF-16 format, but C on modern systems expects 8-bit characters in normal strings. The C prototype `nanf(const char *s)` thus needs some additional processing to translate the C# Unicode string, and that is what the `CharSet` modifier does.

The double-precision function prototypes make up the next big block in `mathcw.cs`:

```
[ DllImport(LIBNAME) ] public static extern double acos(double x);
[ DllImport(LIBNAME) ] public static extern double acosh(double x);
...
[ DllImport(LIBNAME) ] public static extern double urcw3();
[ DllImport(LIBNAME) ] public static extern double urcw4();
```

Because the CLI virtual machine contains only 32-bit `float` and 64-bit `double` floating-point types, C# offers only those types, so there is no way to access the `long double` section of the mathcw library, or the `float` `nexttowardf()` and `double` `nexttoward()` functions, which have a `long double` argument.

As in the Java interface, we want the C# `MathCW` class to be a superset of the standard `Math` class. All of the function names in the C# `Math` class are capitalized, so we provide native wrapper functions that just call their lowercase companions:

```
public static float Acosf (float x)   { return acosf (x); }
public static float Acoshf (float x)  { return acoshf (x); }
...
public static float Urcw3f ()         { return urcw3f (); }
public static float Urcw4f ()         { return urcw4f (); }

public static double Acos (double x)  { return acos (x); }
public static double Acosh (double x) { return acosh (x); }
...
public static double Urcw3 ()         { return urcw3 (); }
public static double Urcw4 ()         { return urcw4 (); }
```

The standard `Math` class in C# is an eclectic mixture of numerical functions for different data types. For floating-point arguments, in most cases, only the `double` version is provided, because the language allows overloaded function names; the compiler picks the right version based on the argument types.

The final block of code in the `mathcw.cs` file therefore provides replacements for each of the remaining functions in the `Math` class, so that simply by changing the class name `Math` to `MathCW` everywhere it is used in a C# program, the `mathcw` library can be used in place of the standard library:

```
    public static decimal Abs (decimal value) { return Math.Abs (value); }
    public static double Abs (double a)       { return fabs (a); }
    public static float Abs (float a)         { return fabsf (a); }
    public static int Abs (int value)         { return Math.Abs (value); }
    ...
    public static double Sign (int value)     { return Math.Sign (value);}
    public static double Sign (long value)    { return Math.Sign (value);}
    public static double Sign (sbyte value)   { return Math.Sign (value);}
    public static double Sign (short value)   { return Math.Sign (value);}
}
```

B.4 Using the C# interface

The C# interface to the mathcw library is easy to use because nothing special needs to be done, other than to compile the `mathcw.cs` file with the user program, and prefix the function names with the class name. Here is a shorter version of one of the test programs that illustrates how the `frexpf()` function can be used:

```
using System;

class short_test08
{
    static void Main()
    {
        for (int k = 0; k <= 24; ++k)
        {
            float x, fx;
            int n;

            x = (float)(1 << k);
            fx = MathCW.frexpf(x, out n);

            if ((fx != 0.5F) || (n != (k + 1)))
                Console.Write("ERROR: ");

            Console.WriteLine("MathCW frexpf( " + x + ", -> " + n + " ) = " + fx);
        }
    }
}
```

Notice that the call to `frexpf()` requires the `out` modifier on the second argument, because the exponent of the first argument is stored there on return.

C C++ interface

AFTER TWO YEARS OF C++ PROGRAMMING, IT STILL SURPRISES ME.

— YUKIHIRO MATSUMOTO
DESIGNER OF THE RUBY LANGUAGE (2003).

With the exception of a small number of disagreements that most programmers need not encounter, the C++ language is effectively a strongly typed superset of the C language.

Thus, as long as a C program has prototypes for all functions, and does not use as variables any of the new reserved words in C++, it should be compilable with both C and C++ compilers. All of the C code in the mathcw library has that property.

Similarly, a C++ program should have access to all of the facilities of the C language, including the many software libraries available for C. In particular, as long as the mathcw.h header file is included in a C++ program, the entire mathcw library is available, exactly like it is for C programs.

Nevertheless, the extra features of the C++ language offer the possibility of doing more. The ability of C++ to overload functions, so that the same function name can be used for the same operation without regard to the data type, can be exploited by a suitable interface that we present in **Section C.2** on the next page.

As with other strongly typed languages, such as Ada, C#, and Java, function overloading depends on there being a distinguishing *signature* (the number and types of the function arguments). Most of the functions in the mathcw library can be provided under the same names as used for the double versions, because the functions have arguments that differ by numeric data type.

C.1 Building the C++ interface

Before we examine the source code for the C++ interface to the mathcw library, we first show how to build and install the interface.

Initially, we have just a few files in the cpp interface subdirectory:

```
% cd cpp
% ls
Makefile  exp  mathcw.hh  okay  test01.cc  test02.cc  test03.cc
```

The Makefile contains the rules for the build process, which is simple, because nothing actually needs to be built. Instead, all that is required is for the C++ interface header file, mathcw.hh, to be visible to the compiler when other code that uses the interface is processed.

We can do a simple validation test with the usual recipe:

```
% make check

CC -g -DHAVE_LONG_DOUBLE -DHAVE_LONG_LONG -I. -I.. -c test01.cc

CC -g -DHAVE_LONG_DOUBLE -DHAVE_LONG_LONG -I. -I.. -o test01 test01.o -L.. -lmcw

...

There should be no output but the test names:

========== test01
========== test02
========== test03
```

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2

That test compiles and links the three programs, then runs them, comparing their output with output files in the okay subdirectory that are known to be correct.

Installation is equally simple:

```
% make install

/bin/rm -f /usr/local/include/mathcw.hh

cp -p mathcw.hh /usr/local/include/

chmod 664 /usr/local/include/mathcw.hh

Installed files...

-rw-rw-r-- 1 mcw 20891 Mar 31 13:47 /usr/local/include/mathcw.hh
```

C.2 Programming the C++ interface

The C++ interface to the mathcw library is stored in a single file, mathcw.hh. The first part of the interface simply retrieves the function prototypes and other definitions from the C file, mathcw.h, but it does so inside a C++ namespace, so that the function names can later be qualified with the namespace name to disambiguate them from identically named C++ wrapper functions:

```
namespace C
{
#include <mathcw.h>

// Remove macro definitions that conflict with our functions:

#undef isfinite
#undef isgreater
...
#undef isunordered
#undef signbit
}
```

The namespace wrapper provides one other useful service: it makes the names in the C header file inaccessible unless qualified with the namespace name, or under the scope of a using statement for the namespace.

The #undef directives are needed on some systems to prevent unwanted macro substitutions later in the interface.

The main part of the interface file comes next. It is just a long list of public wrappers inside a class definition, so we need only show a few of them:

```
class MathCW
{
  public:
    inline float (acos)(float x)
    {
        return (C::acosf(x));
    }
    // ... code omitted ...
    inline double (acos)(double x)
    {
        return (C::acos(x));
    }
    // ... code omitted ...
    inline long double (acos)(long double x)
    {
        return (C::acosl(x));
    }
    // ... code omitted ...
};
```

The wrappers define the overloaded C++ functions. The `C::` prefix in their `return` statement expressions is the namespace qualifier; it ensures that the C versions of the functions are called.

A few functions cannot be provided under generic names because they lack distinguishing signatures. They include the NaN-generator function families `nanf()`, `qnanf()`, and `snanf()`. They also include most of the routines in the random-number generator family: `urcwf()`, `urcw1f()`, `urcw2f()`, `urcw3f()`, `urcw4f()`, and so on. All of those functions are available under their own names.

Even though there are references to all of the functions in the `mathcw` library in the `MathCW` class, only those that are actually called are present in the executable program. In addition, most modern C++ compilers optimize the wrappers away entirely, eliminating all of the class overhead.

C.3 Using the C++ interface

The `MathCW` class provides a new data type to C++ programs that serves mainly as a prefix on the library function names. Here is a short C-like program, available in the `test01.cc` file, that shows how the class is used:

```
#include <stdio.h>
#include <stdlib.h>
#include <mathcw.hh>

int
main(void)
{
    float x;
    double y;
    MathCW mcw;

    x = 1.0F;
    (void)printf("mcw.erf(%g) = %.6g\n", (double)x, (double)mcw.erf(x));
    y = 2.0;
    (void)printf("mcw.erf(%g) = %.14g\n", y, mcw.erf(y));

#if defined(HAVE_LONG_DOUBLE)
    long double z;

    z = 3.0L;
    (void)printf("mcw.erf(%Lg) = %.16Lg\n", z, mcw.erf(z));
#endif

    return (EXIT_SUCCESS);
}
```

We use the simpler C-style output statements here to keep the test program small.

Compilation and running of the test program is handled by the `Makefile`, but we can also do it manually:

```
% CC -DHAVE_LONG_DOUBLE -I.. -I. test01.cc && ./a.out
mcw.erf(1) = 0.842701
mcw.erf(2) = 0.99532226501895
mcw.erf(3) = 0.9999779095030014
```

The companion test files, `test02.cc` and `test03.cc`, use the much more verbose native C++ output statements. The second of these two files exploits template expansion to provide another level of abstraction. However, their code is not of particular interest here, so we omit it.

D Decimal arithmetic

> THE EARLIEST DEFINITE DATE ASSIGNED FOR THE USE IN ARABIA
> OF THE DECIMAL SYSTEM OF NUMERATION IS 773 [CE].
> IN THAT YEAR SOME INDIAN ASTRONOMICAL TABLES WERE BROUGHT
> TO BAGDAD, AND IT IS ALMOST CERTAIN THAT, IN THESE,
> INDIAN NUMERALS (INCLUDING A ZERO) WERE EMPLOYED.
>
> — WALTER WILLIAM ROUSE BALL
> *A Short Account of the History of Mathematics* (1888).

The nature of electronic computers gives the binary number system special significance, because two states, corresponding to the binary digits 0 and 1, can be represented by card holes, CD and DVD spots, electrical circuits, magnetization, and so on. Although a few computers built in the 1950s and 1960s used native decimal arithmetic (see **Table H.1** on page 948, and Cowlishaw's historical survey [Cow03]), it still has to be represented at a lower level in hardware in terms of bits. That in turn means additional overhead in converting stored decimal numbers into binary numbers for computation, and the reverse conversion as numbers are moved from CPU to storage. The execution-time penalty and competitive market forces therefore combine to discourage decimal arithmetic, except in devices, such as hand-held calculators, where speed does not matter.

Speed is less important for computations where relatively few numerical operations are carried out on input data, as in business data processing. Programming languages, such as Ada, COBOL, PL/1, and C# (see **Section 4.27** on page 101), that are designed to support business applications offer fixed-decimal arithmetic. Although that is adequate for most monetary computations, it is untenable for scientific computation because of the need for frequent rescaling of fixed-point numbers to avoid digit loss. Indeed, the programmatic difficulty of handling the scaling problem led to the introduction of floating-point arithmetic in computers in the mid-1950s, and for almost all computers today, that means binary arithmetic based on the IEEE 754 Standard.

D.1 Why we need decimal floating-point arithmetic

Binary arithmetic is unfamiliar to most humans, and for some tasks, can always get the wrong answer. A popular example of that problem is sales-tax computation, such as 5% of an amount 0.70 in some arbitrary currency unit. The exact answer for the total is $1.05 \times 0.70 = 0.7350$, and that value rounded to the nearest even is 0.74. In binary arithmetic, however, the simple factor 1.05 cannot be represented exactly: its value has an infinite fraction represented in hexadecimal form as `0x1.0ccccc...p0`. The required product, `+0x1.7851eb851eb85...p-1`, produces the decimal number 0.7349999 . . . , *no matter how large the numeric precision*. Under both *round-to-nearest* and *round-to-zero* (chopping) rules, the final result is 0.73, which is off by one in the last digit. Although that seems like a trivial amount, it produces substantial errors in the accounts of a business with large numbers of small transactions, such as a grocery store or telephone company. Also, it violates rounding rules for business and tax computations in some countries, which may require one additional rule beyond the four provided in IEEE 754 binary arithmetic: *round-half-up* (0.735 → 0.74). Our sales-tax example is still computed incorrectly in binary arithmetic with the *round-half-up* rule. For symmetry, decimal arithmetic may also provide the rule *round-half-down* (0.735 → 0.73).

By about 2000, the dramatic computer performance and storage improvements over several decades made it desirable to reexamine decimal floating-point arithmetic. IBM led the way by implementing decimal floating-point arithmetic in firmware in 2006 in the z9 mainframe CPU [IBM06, DDZ$^+$07]. It followed that with hardware implementations in the PowerPC version 6 CPU in 2007 [TSSK07], and in the z10 CPU in 2008 [Web08, SKC09]. However, unlike historical machines, that arithmetic is *in addition* to conventional IEEE 754 binary floating-point arithmetic, and for the z9 and z10, also hexadecimal floating-point arithmetic. For a good survey of hardware-design considerations for decimal arithmetic, see [WET$^+$10]. The latest work as this book went to press reports dramatic advances on

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2

the IBM z13 mainframe in the speed of decimal floating-point arithmetic, and the sharing of much low-level circuitry for both binary and decimal operations [LCM16].

In 2005, ISO committees for the standardization of the C and C++ languages received proposals for the addition of decimal floating-point arithmetic [Kla05, C06b, C06c, C06d, C09]. In 2006, GNU developers of compilers for those languages added preliminary support on one CPU platform for the new data types. Those compilers allowed extended development and testing of the mathcw library code for decimal arithmetic, a task that previously had been possible only by using decimal types remapped to binary types.

D.2 Decimal floating-point arithmetic design issues

IBM based its decimal hardware designs partly on its long experience with the Rexx and NetRexx scripting languages [Cow85, Cow90, Cow97], which provide arbitrary-precision decimal arithmetic (up to 10^9 digits) with an exponent range of $[-999\,999\,999, +999\,999\,999]$. That range is so large that overflow and underflow are unlikely in most practical computations, and indeed, those languages treat those two conditions as errors.

IBM's other design guides were the IEEE 854 Standard [ANS87] for radix-independent floating-point arithmetic, and the existence in several programming languages of a large software base written for IEEE 754 binary floating-point arithmetic.

In particular, IBM felt it desirable for decimal formats to have approximately the same range and precision as binary formats with the same storage requirements. Also, there was considerable existing practice on some architectures and in commercially important programming languages:

- IBM mainframes have had hardware support for packed-format decimal fixed-point arithmetic with up to 31 digits since 1964 [FIS64, POP64] (see [IBM70, Chapter 5] for programming examples).
- Existing software libraries on some other platforms support 31-digit decimal arithmetic.
- Some models of the DEC PDP-10 from about 1976 had hardware support for 21-digit decimal fixed-point arithmetic, and the VAX, and models of the older PDP-11 when equipped with the commercial instruction set, had similar support for 31-digit data.
- The Intel IA-32 architecture has had hardware support for 18-digit decimal fixed-point arithmetic since the 8087 floating-point coprocessor was introduced in 1980. The AMD64, EM64T, and IA-64 architectures include the IA-32 decimal instructions.
- The Ada language provides decimal fixed-point arithmetic with at least 18 digits [Ada95], and also includes a `Packed_Decimal` type that corresponds to COBOL's `packed-decimal` type.
- Older ISO COBOL Standards require 18-digit decimal fixed-point arithmetic.
- The 2002 ISO COBOL Standard [COB02] mandates 32-digit decimal floating-point arithmetic and a power-of-ten exponent with at least three digits.
- The PL/1 language supports decimal arithmetic with at least 15 fixed-point digits, and 33 floating-point digits.
- Most databases support storage of decimal fixed-point data with at least 18 digits, and some allow up to 35 digits.

Those considerations led to improvements in the old BCD (binary-coded decimal) format, which stores one decimal digit in a four-bit nybble, and the later Chen–Ho encoding [CH75], which is more compact. IBM calls their new encoding *Densely Packed Decimal (DPD)* [Cow02], and its features are summarized in **Table D.1** on the facing page and **Figure D.1** on page 930. The significand size and exponent range in that encoding is large enough to handle almost everything in our list of existing practice. The IBM `decNumber` package [Cow07] provides an open-source reference implementation of Rexx-style decimal arithmetic with an additional software layer for converting to and from DPD arithmetic, and between native binary and DPD formats.

Other encodings of decimal arithmetic are certainly possible. Researchers at Intel developed the *Binary Integer Decimal (BID)* format [CAH$^+$07], summarized in **Table D.2** on the facing page and **Figure D.2** on page 930. The BID encoding represents the decimal coefficient as a binary integer, and computations are done in binary, allowing re-use

Table D.1: Extended IEEE-754-like decimal floating-point characteristics and limits in the IBM Densely Packed Decimal (DPD) encoding.

	single	**double**	**quadruple**	**octuple**
Format length	32	64	128	256
Stored coefficient digits	7	16	34	70
Precision (p)	7	16	34	70
Biased-exponent bits	8	10	14	22
EC bits	6	8	12	20
Minimum exponent	−95	−383	−6143	−1 572 863
Maximum exponent	96	384	6144	1 572 864
Exponent bias	101	398	6176	1 572 932
Machine epsilon (10^{-p+1})	10^{-6}	10^{-15}	10^{-33}	10^{-69}
Largest normal	$(1-10^{-7})10^{97}$	$(1-10^{-16})10^{385}$	$(1-10^{-34})10^{6145}$	$(1-10^{-70})10^{1\,572\,865}$
Smallest normal	10^{-95}	10^{-383}	10^{-6143}	$10^{-1\,572\,863}$
Smallest subnormal	10^{-101}	10^{-398}	10^{-6176}	$10^{-1\,572\,932}$

Table D.2: Extended IEEE-754-like decimal floating-point characteristics and limits in the Intel Binary Integer Decimal (BID) encoding. Values are largely identical to those for DPD encoding, but shading indicates where there are differences.

	single	**double**	**quadruple**	**octuple**
Format length	32	64	128	256
Stored coefficient bits	23 or 21	53 or 51	113 or 111	233 or 231
Coefficient digits	7	16	34	70
Precision (p)	7	16	34	70
Biased-exponent bits	8	10	14	22
Minimum exponent	−95	−383	−6143	−1 572 863
Maximum exponent	96	384	6144	1 572 864
Exponent bias	101	398	6176	1 572 932
Machine epsilon (10^{-p+1})	10^{-6}	10^{-15}	10^{-33}	10^{-69}
Largest normal	$(1-10^{-7})10^{97}$	$(1-10^{-16})10^{385}$	$(1-10^{-34})10^{6145}$	$(1-10^{-70})10^{1\,572\,865}$
Smallest normal	10^{-95}	10^{-383}	10^{-6143}	$10^{-1\,572\,863}$
Smallest subnormal	10^{-101}	10^{-398}	10^{-6176}	$10^{-1\,572\,932}$

of circuitry in existing arithmetic units. Binary arithmetic makes correct decimal rounding difficult, but hardware engineers later found satisfactory solutions that avoid the need to convert from binary to decimal, round, and convert back [Inte06, TSGN07, TGNSC11].

BID encoding supports a larger range of coefficient values than a pure decimal representation. For example, in the first layout in **Figure D.2** on the following page, with the 32-bit format there are 23 bits for the coefficient, representing integers up to $2^{23} - 1 = 8\,388\,607$. Larger numbers require the second layout, as long as the exponent field does not start with two 1 bits. The largest integer is then `0x9fffff` $= 10\,485\,759$, beyond the pure decimal limit of 9 999 999. Thus, BID can exactly represent about 4.86% more numbers than DPD can. Software that exploits that feature of BID is not portable to systems with DPD, and despite the identical exponent range, rounding rules, and machine epsilon in the two encodings, the wider coefficient range of BID effectively means a small improvement in precision that can visibly affect final results. It remains to be seen whether BID implementations constrain representations to the exact range of DPD, thereby preventing those differences.

The computing industry will likely require a few years of experience with both encodings, and their implementations in hardware, before architects can decide whether one of them has significant advantages over the other. Some view a single standard encoding of decimal data as highly desirable, because the existence of multiple encodings

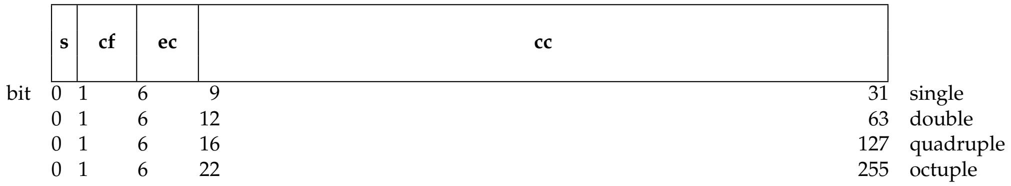


Figure D.1: Extended IEEE-754-like decimal floating-point data layout in the IBM Densely Packed Decimal (DPD) format.
s is the sign bit (0 for $+$, 1 for $-$).
cf is the combination field containing two high-order bits of the unsigned biased exponent and the leading decimal digit of the coefficient. Infinity has a *cf* of 11110, and NaN has a *cf* of 11111.
ec is the exponent continuation field with the remaining unsigned biased exponent bits. Quiet NaN has a high-order *ec* bit of 0, and signaling NaN has a high-order *ec* bit of 1. IBM zSeries documentation labels that field *bxcf* for *biased exponent continuation field*.
cc is the coefficient continuation field with the remaining exponent digits in DPD encoding with three decimal digits in ten bits. The coefficient is an integer, with the decimal point at the right, which means there are redundant representations of numbers, and also that trailing zeros can be preserved, and may be significant to the user.

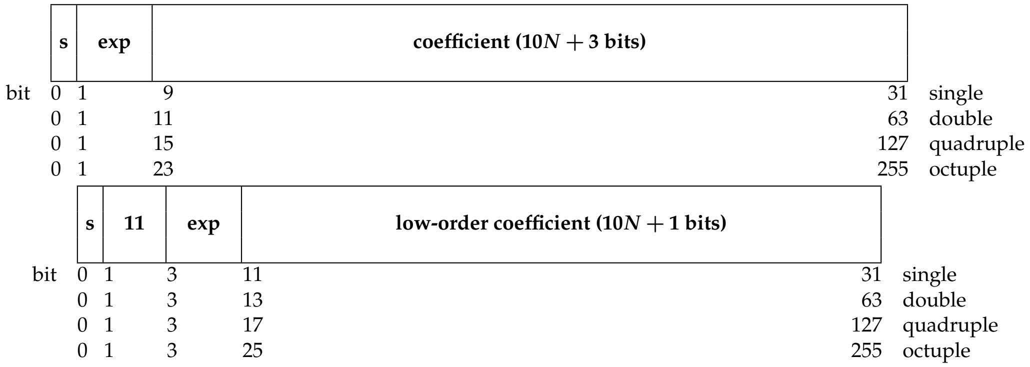


Figure D.2: Extended IEEE-754-like decimal floating-point data layout in the Intel Binary Integer Decimal (BID) format.
s is the sign bit (0 for $+$, 1 for $-$).
exp is the unsigned biased power-of-ten exponent field, with the smallest value reserved for zero and subnormal numbers. Its valid range is $[0, 3 \times 2^{w-2} - 1]$ when the exponent field is w bits wide.
Infinity normally has a zero coefficient (but nonzero is permitted too), with *exp* of 11 110 . . . . Quiet NaN has *exp* of 11 111 0 . . . . Signaling NaN has *exp* of 11 111 1 . . . . Default NaNs have zero coefficients, but nonzero values are allowed.
The remaining bits hold the coefficient when it fits in $10N + 3$ bits (first layout). If it requires $10N + 4$ bits (second layout), three high-order bits are supplied implicitly as 100 and followed by the remaining stored low-order coefficient bits, and the exponent field must then not start 11 . . . .
The coefficient is an integer, with the decimal point at the right, which means there are redundant representations of numbers, and also that trailing zeros can be preserved, and may be significant to the user.

complicates exchange of native decimal floating-point data, but so do endian addressing differences. However, a textual representation of decimal data with the correct number of digits provides an exact exchange mechanism, and compactness can be regained by a general text-compression utility.

The GNU compilers initially provided support for decimal floating-point arithmetic in only the DPD format, but later added a build-time option to choose BID encoding. However, at the time of writing this, the compilers convert data in the BID format to DPD for use with the decNumber library.

Although current compiler support provides access only to 32-bit, 64-bit and 128-bit decimal formats, it is a design goal of the mathcw library to support future 256-bit binary and decimal formats. The cited tables therefore include a final column for such arithmetic, just as we show for binary arithmetic in **Figure 4.1** on page 63 and **Table 4.2** on page 65.

The proposed exponent ranges are determined by packing constraints of the DPD format, where ten bits encode three decimal digits, and an additional high-order decimal digit is encoded in the combination field. That implies that each format encodes $3n + 1$ decimal digits. Together with the desirable feature that storage doubling from an m-decimal-digit format should provide at least $2m + 2$ decimal digits, those constraints produce a 256-bit format holding 70 decimal digits. The remaining bits determine the exponent range, and then comparison of the binary and decimal ranges for the three smaller sizes produces a clear choice for the exponent range in the 256-bit binary format.

D.3 How decimal and binary arithmetic differ

IEEE 754 binary floating-point arithmetic provides a hidden significand bit for normalized numbers in all but the 80-bit format, plus representations of Infinity, quiet and signaling NaN, and subnormal numbers. Nonzero normal numbers are represented as the product of a significand in $[1, 2)$ and an integer power of two. Thus, the binary point lies to the left of the first stored fraction bit.

With decimal arithmetic, no hidden digit is possible for normal numbers, because there are nine possibilities instead of just one.

Decimal floating-point arithmetic supports Infinity and two kinds of NaNs, but their encoding uses explicit bit patterns in the DPD *cc* and *ec* fields, or the BID *exp* field, that permit them to be recognized solely by examination of those fields. By contrast, the IEEE 754 binary encoding requires examination of all significand bits to distinguish between Infinity and NaN. Fast identification of Infinity and NaN is important because they almost always require special handling, and as significand length increases, the binary format suffers a performance hit from that task.

In the binary format, all vendors use the high-order fractional bit to identify quiet and signaling NaNs, although they do not agree on whether a 1 bit means quiet or signaling. Most choose the quiet interpretation, so that storage contents of all-bits-one is a negative quiet NaN, making it simple to initialize unset memory cells to be able to trap use before definition. Some vendor C compilers provide an option, usually called `-trapuv`, to request use of NaNs for initial values. We discuss bulk initialization in more detail in **Section D.4** on page 935.

Those differences are largely transparent to numerical programs and programmers. However, the next one is not. Decimal arithmetic places the decimal point at the *right* of the significand, so that decimal numbers are represented as the product of an integer coefficient and a power of ten. That choice was made for an important reason: *it allows floating-point arithmetic to emulate fixed-point arithmetic for financial computations.* Addition and subtraction of floating-point numbers with the same exponent are *exact* as long as overflow does not occur.

Handling fixed-point arithmetic that way requires that computed results must not be renormalized, except by explicit request. The exponent must remain constant and it must be possible to check that the property holds at the end of a computation. The `decNumberSameQuantum()` function in the `decNumber` library tests whether two values have the same exponent, and the companion function `decNumberQuantize()` forces one value to have the same decimal exponent as another. Similar capabilities are present in the proposed decimal extensions to C, and the mathcw library supplies them with these prototypes:

```
#include <mathcw.h>

decimal_float           quantizedf   (decimal_float x,           decimal_float y);
decimal_double          quantized    (decimal_double x,          decimal_double y);
decimal_long_double     quantizedl   (decimal_long_double x,     decimal_long_double y);
decimal_long_long_double quantizedll (decimal_long_long_double x, decimal_long_long_double y);

int samequantumdf                    (decimal_float x,           decimal_float y);
int samequantumd                     (decimal_double x,          decimal_double y);
int samequantumdl                    (decimal_long_double x,     decimal_long_double y);
int samequantumdll                   (decimal_long_long_double x, decimal_long_long_double y);
```

For completeness, those functions have companions for binary arithmetic with the usual suffixes `f`, *none*, `l`, and `ll`, although they are unlikely to be of much utility.

Table D.3: Behavior of the quantize() function. The third through fifth examples show that quantization can cause rounding, here with the default *round-to-nearest* rule. The last decimal and binary examples show what happens when precision is lost.

Function call	Result
decimal	
quantized(+1.DD, +1.00DD)	+1.00DD
quantized(+100.DD, +1.00DD)	+100.00DD
quantized(+0.125DD, +1.00DD)	+0.12DD
quantized(+0.135DD, +1.00DD)	+0.14DD
quantized(+0.145DD, +1.00DD)	+0.14DD
quantized(+NaN(0x1234), +1.00DD)	+NaN(0x1234)
quantized(+100.DD, +NaN(0x1234))	+NaN(0x1234)
quantized(+NaN(0x1234), +NaN(0x5678))	+NaN(0x1234)
quantized(+∞, −∞)	+∞
quantized(−∞, +∞)	−∞
quantized(+100.00DD, −∞)	+NaN
quantized(−∞, +100.00DD)	+NaN
quantized(+1234567890123456.DD, +1.DD)	+1234567890123456.DD
quantized(+1234567890123456.DD, +1.0DD)	+NaN
binary	
quantize(+1., +0.02)	+1.
quantize(+100., +0.02)	+100.
quantize(+0.125, +0.02)	+0.12
quantize(+0.135, +0.02)	+0.14000000000000001
quantize(+0.145, +0.02)	+0.14000000000000001
quantize(NaN(0x1234), +0.02)	NaN(0x1234)
quantize(+100., NaN(0x1234))	NaN(0x1234)
quantize(NaN(0x1234), NaN(0x5678))	NaN(0x1234)
quantize(+∞, −∞)	+∞
quantize(−∞, +∞)	−∞
quantize(+100.00, −∞)	NaN
quantize(−∞, +0.02)	NaN
quantize(+1234567890123456., +1.)	+1234567890123456.
quantize(+1234567890123456., +0.2)	NaN

For example, quantized(x,y) sets x to have the same *decimal* exponent as y, and if both arguments are Infinity, the result is the first argument. Otherwise, if either argument is a NaN, the result is that argument, and if just one argument is Infinity, the result is a quiet NaN. If the operation would require more digits than are available, the result is a quiet NaN. If rounding is required, it obeys the current rounding mode.

The purpose of the second argument is just to communicate a power of ten, so its particular coefficient digits are not significant. For decimal use, trailing zero digits are commonly used, so that 1.00 means *quantize to two decimal digits*. In financial computations, such operations are needed frequently, such as for rounding an amount to the nearest cent in several currency systems.

For the binary companions of quantized(), the normalized floating-point format requires that the second argument be a fractional value chosen away from an exact power of ten, to avoid off-by-one errors in the inexact determination of the power of ten. Table D.3 shows examples of how that function works.

Quantization also provides conversion to integers: quantized(x, 1.DD) is the whole number (in floating-point format) nearest to x, assuming the default rounding mode.

The function samequantumd(x,y) returns 1 if the arguments have the same *decimal* exponent, and 0 otherwise. If the arguments are both Infinity, or both NaN, the return value is also 1. Table D.4 on the next page illustrates the workings of that function.

One additional function is required for emulating fixed-point arithmetic: normalization of decimal values by removal of trailing zero digits. The decNumber library function decNumberNormalize() therefore has these counterparts in the mathcw library, even though they are absent from the proposals for standardization of decimal arithmetic in C and C++:

```
#include <mathcw.h>
```

Table D.4: Behavior of the `samequantum()` function. Notice the difference in the second of the binary and decimal examples: numbers of quite different magnitude can have the same scale in decimal, but in binary, because of normalization, they need to be within a factor of ten to have the same exponent. The third examples reflect the different normalization of binary and decimal values.

Function call	Result
decimal	
`samequantumd(+1.00DD, +9.99DD)`	1
`samequantumd(+1.00DD, +9999.99DD)`	1
`samequantumd(+1.DD, +1.00DD)`	0
`samequantumd(+100.DD, +1.00DD)`	0
`samequantumd(+NaN(0x1234), +1.00DD)`	0
`samequantumd(+100.DD, +NaN(0x1234))`	0
`samequantumd(+NaN(0x1234), +NaN(0x5678))`	1
samequantumd($+\infty$, $-\infty$)	1
samequantumd($-\infty$, $+\infty$)	1
samequantumd(+100.00DD, $-\infty$)	0
samequantumd($-\infty$, +100.00DD)	0
binary	
`samequantum(+1.00, +1.9375)`	1
`samequantum(+1.00, +9999.75)`	0
`samequantum(+1., +1.00)`	1
`samequantum(+100., +1.00)`	0
`samequantum(NaN, +1.00)`	0
`samequantum(+100., NaN)`	0
`samequantum(NaN, NaN)`	1
samequantum($+\infty$, $-\infty$)	1
samequantum($-\infty$, $+\infty$)	1
samequantum(+100.00, $-\infty$)	0
samequantum($-\infty$, +100.00)	0

```
decimal_float            normalizedf  (decimal_float x);
decimal_double           normalized   (decimal_double x);
decimal_long_double      normalizedl  (decimal_long_double x);
decimal_long_long_double normalizedll (decimal_long_long_double x);
```

Like the earlier functions, companions for binary arithmetic have the usual suffixes `f`, *none*, `l`, and `ll`, but they simply return their arguments, because binary floating-point numbers are always normalized in modern floating-point architectures. **Table D.5** on the following page shows some samples of the use of the decimal functions.

Another consequence of storing decimal floating-point values with the decimal point at the right is that there are multiple representations of any number that does not use all significand digits. Thus, `1.DF`, `1.0DF`, `1.00DF`, ..., `1.000000DF` are numerically equal for comparison, yet have different storage representations equivalent to 1×10^0, 10×10^{-1}, 100×10^{-2}, ..., 1000000×10^{-6}.

Addition and subtraction of decimal values changes scale when exponents differ, and multiplication changes scale if trailing fractional zeros are present. Thus, the result of `1.DF * 1.234DF` is `1.234DF`, preserving the original value. However, `1.000DF * 1.234DF = 1.234000DF`, an equal value of different scale with the representation 1234000×10^{-6}. Programmers who expect to use decimal floating-point arithmetic to emulate fixed-point arithmetic therefore need to be careful to avoid altering scale. When the mathcw library was first tested with decimal arithmetic, several constants with trailing zeros had to be rewritten or trimmed, like these examples: `100.DF` → `1.e2DF` and `10.0` → `1.e1DF`. Because many learn in school to write at least one digit on either side of a decimal point, and some programming languages require that practice (although the C-language family members do not), it is easy to make subtle scale-altering mistakes with decimal floating-point constants.

Multiplication and division in general change the scale, and consequently, fixed-point computations require scale readjustment after such operations. In the sales-tax computation cited earlier, we could write code like this:

```
decimal_long_double amount, rate, total;
```

Table D.5: Behavior of the decimal `normalize()` function.

Function call	Result
`normalized(+0.00100DD)`	`+0.001DD`
`normalized(+1.00DD)`	`+1.DD`
`normalized(+100.DD)`	`+1E+2DD`
`normalized(+100.00DD)`	`+1E+2DD`
`normalized(+NaN(0x1234))`	`+NaN(0x1234)`
`normalized(-NaN(0x1234))`	`-NaN(0x1234)`
`normalized(`$+\infty$`)`	$+\infty$
`normalized(`$-\infty$`)`	$-\infty$

```
amount = 0.70DL;
rate   = 1.05DL;
total  = quantizedl(amount * rate, 1.00DL);
```

Without the call to `quantizedl()`, the total would have been 0.7350 instead of 0.74.

A zero value encoded in decimal with either DPD or BID does not have all bits zero in storage, as is the case with binary floating-point formats. For example, in DPD encoding, `0.DF` is represented as `0x22500000`, whereas the all-bits-zero representation `0x00000000` corresponds to the value `0.e-101DF`. The two values are equal when compared with decimal floating-point instructions, but not when compared as bit patterns. Programmers who examine bits via `union` types or with debugger commands, and compiler writers, need to be particularly aware of that point.

The change of base from 2 to 10 has other effects on programmers who are accustomed to working with binary floating-point arithmetic:

- The larger base means larger spacing of consecutive decimal numbers (look at the machine-epsilon rows in Table 4.2 on page 65, and Table D.1 and Table D.2 on page 929), so computational precision is slightly reduced.

- The larger base lowers the frequency of alignment shifts for addition and subtraction, and when there are no shifts or digit overflow, rounding is not required and computations are *exact*. That feature improves performance of software implementations of decimal arithmetic, particularly when data have a common scale, as in financial computations.

- Multiplication by constants is only exact if they are powers of ten, or if the product is small enough to be representable without rounding. In particular, doubling by adding or by multiplying by two may no longer be an exact operation. For example, `2.DF * 9.999999DF` is 19.999 998, a value that has more digits than can be represented in the 32-bit format, so the result has to be rounded, in that case, to `20.00000DF`.

- Input of decimal values is an exact operation as long as the number of digits is not so large that rounding is required. Output of decimal values is also exact as long as the format allows sufficient output digits for the internal precision.

- Output with numeric format specifications in the `printf()` function family loses information about quantization. Preserve it by using the `ntos()` family (see Section 26.11 on page 867) to produce a string. For example, the program fragment

  ```
  decimal_float x;

  x = 123.000DF;
  (void)printf("%%He:    x = %He\n", x);
  (void)printf("%%Hf:    x = %Hf\n", x);
  (void)printf("%%Hg:    x = %Hg\n", x);
  (void)printf("ntosdf(x) = %s\n", ntosdf(x));
  ```

  produces this output:

```
%He:    x = 1.230000e+02
%Hf:    x = 123.000000
%Hg:    x = 123
ntosdf(x) = +123.000
```

D.4 Initialization of decimal floating-point storage

Important special values in the DPD encoding of decimal floating-point arithmetic can be recognized by examining just the high-order byte, and Infinity is not required to have a zero significand, as it is in IEEE 754 binary arithmetic. Those two features facilitate compile-time and fast run-time storage initialization, as demonstrated by this short test program:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <mathcw.h>

void
show(int byte)
{
    decimal_float f;
    decimal_double d;
    decimal_long_double q;
    decimal_long_long_double o;

    (void)memset((void*)&f, byte, sizeof(f));
    (void)memset((void*)&d, byte, sizeof(d));
    (void)memset((void*)&q, byte, sizeof(q));
    (void)memset((void*)&o, byte, sizeof(o));
    (void)printf("0x%02x: %+Hg %+DDg %+DLg %+DLLg\n", byte, f, d, q, o);
}

int
main(void)
{
    int k;

    for (k = 0; k < 256; ++k)
        show(k);

    return (EXIT_SUCCESS);
}
```

When compiled and run, its output looks like this, where we have elided repeated bytes in NaN payloads to fit the page, and dropped uninteresting lines:

```
0x00: +0 +0 +0 +0
...
0x78: +inf +inf +inf +inf
0x7c: +qnan(0x3c7c7c) +qnan(0xc7c...) +qnan(0x7c7c...) +qnan(0x7c7c...)
0x7d: +qnan(0x3d7d7d) +qnan(0xd7d...) +qnan(0x7d7d...) +qnan(0x7d7d...)
0x7e: +snan(0x3e7e7e) +snan(0xe7e...) +snan(0x7e7e...) +snan(0x7e7e...)
0x7f: +snan(0x3f7f7f) +snan(0xf7f...) +snan(0x7f7f...) +snan(0x7f7f...)
...
0xf8: -inf -inf -inf -inf
0xfc: -qnan(0x3cfcfc) -qnan(0xcfc...) -qnan(0x7cfc...) -qnan(0x7cf...)
```

```
0xfd: -qnan(0x3dfdfd) -qnan(0xdfd...) -qnan(0x7dfd...) -qnan(0x7df...)
0xfe: -snan(0x3efefe) -snan(0xefe...) -snan(0x7efe...) -snan(0x7ef...)
0xff: -snan(0x3fffff) -snan(0xfff...) -snan(0x7fff...) -snan(0x7ff...)
```

A straightforward modification of the test program for IEEE 754 binary arithmetic shows that only initializer bytes 0x00 and 0xff are uniformly useful for all precisions: they produce $+0$ and $-$QNaN (or $-$SNaN) values, respectively.

D.5 The <decfloat.h> header file

The proposed extensions to ISO Standard C and C++ for decimal arithmetic introduce a new header file, <decfloat.h>, as a companion to the traditional <float.h>. Those header files provide symbolic names for characteristics of the arithmetic. Each macro name proposed for <decfloat.h> contains the storage size in bits, an undesirable practice that is inflexible and short-sighted. Nevertheless, the implementation of that header file in the mathcw library includes definitions of the proposed names, with recommended alternatives that follow the naming conventions in <float.h> and hide storage size, as summarized in **Table D.6** on the facing page.

The DEC_EVAL_METHOD macro in the table requires some explanation. Its purpose is to allow implementors to provide fewer than three decimal types, with the unsupported ones masquerading as other decimal types. Its value is a compile-time constant chosen by the implementor, and suitable for use in preprocessing directives. DEC_EVAL_METHOD takes one of these values:

-1 Indeterminable.

0 Evaluate all operations and constants just to the range and precision of the type.

1 Evaluate operations and constants of type decimal_float and decimal_double to the range and precision of the decimal_double type, and decimal_long_double operations and constants to the range and precision of the decimal_long_double type.

2 Evaluate all operations and constants to the range and precision of the decimal_long_double type.

All other negative values correspond to implementation-defined behavior. In the mathcw library, DEC_EVAL_METHOD is normally -1.

D.6 Rounding in decimal arithmetic

In **Section 4.6** on page 66 we show how rounding works, and in **Section 5.6** on page 110 and **Section 5.8** on page 115, we describe how a programmer can use library calls to control rounding. IEEE 754 binary arithmetic provides four rounding directions: *to* $-\infty$, *to zero* (chopping or truncation), *to* $+\infty$, and the default, *to nearest* (or *to nearest even number* in the event of a tie).

Because of historical practice, and legal requirements on commerce and taxation, financial computations in decimal arithmetic require the additional rounding mode *round-half-up*, and possibly also its companion *round-half-down*. They set the rounding direction for halfway cases: if the computed result is exactly halfway between two adjacent representable values, round the magnitude up (away from zero) or down (toward zero). The decNumber library provides yet another choice, *round-up* (away from zero), giving a total of *seven* rounding modes. The IBM z9 and z10 offer an eighth choice, called *round-to-prepare-for-shorter-precision*, but it has no relevance for the current level of decimal arithmetic in C. Version 6 of the IBM PowerPC, introduced on 21 May 2007, supplies the same decimal rounding modes as the z9.

Unfortunately, at the time of writing this, the GNU compilers always invoke decNumber functions with the default rounding mode, so it has not yet been possible to test the mathcw library rounding-control features for decimal arithmetic.

Table D.6: The `<decfloat.h>` header file. The values match those of both DFP and BID encodings. If the 256-bit format is not supported, its macros have the same values as in the 128-bit format.

Proposed	Recommended	Value
evaluation method		
`DEC_EVAL_METHOD`	`DEC_EVAL_METHOD`	`-1, 0, 1, 2`
number of digits in significand		
`DEC32_MANT_DIG`	`DEC_FLT_MANT_DIG`	`7`
`DEC64_MANT_DIG`	`DEC_DBL_MANT_DIG`	`16`
`DEC128_MANT_DIG`	`DEC_LDBL_MANT_DIG`	`34`
`DEC256_MANT_DIG`	`DEC_LLDBL_MANT_DIG`	`70`
minimum exponent of 10		
`DEC32_MIN_EXP`	`DEC_FLT_MIN_EXP`	`-95`
`DEC64_MIN_EXP`	`DEC_DBL_MIN_EXP`	`-383`
`DEC128_MIN_EXP`	`DEC_LDBL_MIN_EXP`	`-6143`
`DEC256_MIN_EXP`	`DEC_LLDBL_MIN_EXP`	`-1572863`
maximum exponent of 10		
`DEC32_MAX_EXP`	`DEC_FLT_MAX_EXP`	`96`
`DEC64_MAX_EXP`	`DEC_DBL_MAX_EXP`	`384`
`DEC128_MAX_EXP`	`DEC_LDBL_MAX_EXP`	`6144`
`DEC256_MAX_EXP`	`DEC_LLDBL_MAX_EXP`	`1572864`
maximum representable finite decimal floating-point number [there are 6, 15, 33, and 69 9's after the decimal point]		
`DEC32_MAX`	`DEC_FLT_MAX`	`9.999999E+96DF`
`DEC64_MAX`	`DEC_DBL_MAX`	`9.999999...E+384DD`
`DEC128_MAX`	`DEC_LDBL_MAX`	`9.999999...E+6144DL`
`DEC256_MAX`	`DEC_LLDBL_MAX`	`9.999999...E+1572864DLL`
machine epsilon		
`DEC32_EPSILON`	`DEC_FLT_EPSILON`	`1E-6DF`
`DEC64_EPSILON`	`DEC_DBL_EPSILON`	`1E-15DD`
`DEC128_EPSILON`	`DEC_LDBL_EPSILON`	`1E-33DL`
`DEC256_EPSILON`	`DEC_LLDBL_EPSILON`	`1E-69DLL`
minimum normalized positive decimal floating-point number		
`DEC32_MIN`	`DEC_FLT_MIN`	`1E-95DF`
`DEC64_MIN`	`DEC_DBL_MIN`	`1E-383DD`
`DEC128_MIN`	`DEC_LDBL_MIN`	`1E-6143DL`
`DEC256_MIN`	`DEC_LLDBL_MIN`	`1E-1572863DLL`
minimum subnormal positive decimal floating-point number [there are 5, 14, 32, and 68 0's after the decimal point]		
`DEC32_DEN`	`DEC_FLT_DEN`	`0.000001E-95DF`
`DEC64_DEN`	`DEC_DBL_DEN`	`0.000000...1E-383DD`
`DEC128_DEN`	`DEC_LDBL_DEN`	`0.000000...1E-6143DL`
`DEC256_DEN`	`DEC_LLDBL_DEN`	`0.000000...1E-1572863DLL`

D.7 Exact scaling in decimal arithmetic

The `mathcw` library provides exact scaling of decimal values with obvious extensions of the venerable `ldexp()` family: if `x` has type `decimal_double`, then `ldexpd(x, 3)` returns a value with the same coefficient, and the exponent-of-ten increased by three.

The decimal members of the companion `frexp()` family decompose a number into a fraction in $[0, 1)$, and a power of ten. The decimal members of the C99 `logb()` and `scalbn()` families extract a power-of-ten exponent that can be used to find a significand in $[1, 10)$. A short test program illustrates those functions:

```
% cat dscale.c
#include <stdio.h>
#include <stdlib.h>
#include <mathcw.h>

int
```

```
main(void)
{
    decimal_double x, y;
    int n;

    x = 1234.56789e+1D;
    y = frexpd(x, &n);
    (void)printf("x                     = %s\n", ntosd(x));
    (void)printf("ldexpd(x, 3)          = %s\n", ntosd(ldexpd(x, 3)));
    (void)printf("frexpd(x, &n)         = %s\n", ntosd(y));
    (void)printf("n                     = %d\n", n);
    (void)printf("logbd(x)              = %s\n", ntosd(logbd(x)));
    (void)printf("scalbnd(x, -logbd(x)) = %s\n", ntosd(scalbnd(x, -logbd(x))));

    return (EXIT_SUCCESS);
}

% dgcc dscale.c -lmcw && ./a.out
x                     = +12345.67890000000
ldexpd(x, 3)          = +12345678.90000000
frexpd(x, &n)         = +0.1234567890000000
n                     = 5
logbd(x)              = +4
scalbnd(x, -logbd(x)) = +1.234567890000000
```

E Errata in the Cody/Waite book

> THE WORLD'S A BOOK . . .
> 'TIS FALSLY PRINTED, THOUGH DIVINELY PENN'D,
> AND ALL TH' ERRATA WILL APPEAR AT TH' END.
>
> — FRANCIS QUARLES (1632).

The Cody / Waite book [CW80] is an early example of computer-based typesetting, and because it contains hundreds of polynomial coefficients, and hand-drawn flowcharts to specify computer algorithms, one might expect that typographical errors have crept in. The exercise of implementing the mathcw library demonstrates that, compared to many more-recent books in the computing field, their book has remarkably few errors, and more than thirty years later, remains an outstanding landmark in the history and literature of numerical computation.

The choice of flowcharts and prose for algorithm descriptions is understandable, but is, in this author's view, regrettable. Programming is hard, and numerical programming is harder still, because it must deal with many small details, and variations in base, precision, rounding, and particulars of hardware instruction sets and numeric data formats, that are rarely evident until code is actually written, ported to many systems, and thoroughly tested. Flowcharts and pseudocode are *not* executable computer code, and can easily contain syntax errors, and algorithmic errors and omissions, that few humans can spot. Also, it is too easy with pseudocode to hide a complicated and subtle computer algorithm behind a vague statement of what is to be done; this book's **Chapter 9** on argument reduction was written in response to just that sort of problem, and its code and writing took more than a month to get right, despite repeated careful reading of the original research articles.

It is true that showing working computer-program code, as we do throughout this book, necessarily obscures algorithms with programming-language syntax and irregularities, and hardware details, but that is an unavoidable fact of life if computers are to work, and do so reliably, for humans. Six years after the work on this book, and its code, began, this author remains convinced that the choice of implementation language and software principles and practices that are laid out in the introductory chapter are the correct ones for this particular task.

All of the polynomial coefficients for both binary and decimal floating-point arithmetic were retyped from the book into mathcw header files. No errors attributed to those coefficients were found in numerical testing, although in a small number of cases, testing showed that polynomial degrees needed to be increased by one or two to reach the accuracy demanded of the mathcw library. The lack of machine-readable text for their book's data and algorithms, and a clear statement of what auxiliary functions the polynomials approximate, made the job much harder than it should have been, and convinced this author of the need for publication of this book, and all of its source code, so that others can build on it in the future.

Extension of the Cody / Waite algorithms to new platforms and extended floating-point types demands local control over polynomial fits, and those authors are silent on the origins of their polynomials. We remedied that serious deficiency by showing in this book how to make such fits with two popular, albeit commercial, symbolic-algebra systems, in both rational minimax form, and as Chebyshev approximations. We also showed how to convert the latter to rational form, allowing reuse of algorithm code. For every function treated in this book where a polynomial fit is required for part of the computation, our header files contain data that apply to all historical and current systems, as well as future ones with up to 70 decimal digits of precision. All of our symbolic-algebra code is part of the mathcw distribution, making it easy to adapt it for other functions not covered by the library, and to other machine precisions and number bases.

In the test program for `sqrt()` on page 32 of the Cody / Waite book, in the second line from the bottom of the page, the assignment `X1 = X` should be `X1 = Y`: it saves the test argument, and in that program, that is `Y`, not `X` as it is in the other test programs.

In the description of the computation of one of the polynomial approximations needed for `pow(x,y)` on page 100, the book incorrectly says *evaluate* `R = z * v * P(v)`. It should say instead *evaluate* `R = z * P(v)`.

In the flowcharts, there is an error on page 127 in the description of the computation of $\cos(x)$ and $\sin(x)$. In the first rectangular box for $\cos(x)$, the chart specifies an assignment $y = |x| + \pi/2$. That assignment must be

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2

omitted, because that adjustment is handled later in the second-last rectangular box, which has the assignment `XN` $=$ `XN` $- 0.5$. For $\cos(x)$ only, the computation of `N` in the middle box should be `INTRND(Y/`π` + 1/2)`, instead of the stated `INTRND(Y/`π`)`. That confusion arises because in the relation $\cos(x) = \sin(|x| + \pi/2)$, direct addition of the constant $\pi/2$ introduces unnecessary additional error because that constant is not exactly representable. Instead, it is preferable to incorporate it as the exact addend $1/2$ in the computation of `N`.

In the flowchart on page 62, and the description on page 63, the cutoff value `SMALLX` below which $\exp(x)$ is zero is defined to be *the smallest machine number greater than* $\ln(\text{XMIN})$. Although that was a good choice prior to IEEE 754 arithmetic, it has the effect of producing premature underflow to zero in the exponential function, as well as in several others that use it, instead of gradual underflow into the subnormal region. That is not strictly an error in the book, because subnormals were not yet invented when it was written, but it is an issue that modern code needs to deal with.

The most complex Cody/Waite algorithm, that for the power function that we treat in **Chapter 14** of this book, exhibits the largest number of problems:

- The inline binary search needs one more step; otherwise, it chooses the wrong endpoint, and that results in errors of a few ulps in the critical powers $(\pm 1)^n$ and β^n, which should *always* be exact.
- Separate treatment of integer powers is desirable, for speed, and for accuracy.
- Computation of n^w is not accurate enough.
- If $g = A_1[k]$ at the start of the search through that table, the computed index lies outside the table, and may cause a run-time addressing error, or a completely incorrect returned function value.
- The value u_2 is not accurate enough; it needs at least twice working precision.
- The value w is not accurate enough; it needs at least three times working precision.
- The `REDUCE()` operation needs more work; we improved it for our code.
- Cody and Waite always choose $q = 1$, but we show that larger q values improve accuracy.
- The power function for decimal arithmetic seems not to have been implemented on a real computer; had it been, several of the noted errors would have been caught.

For future work on the power function, it is likely to be better to work instead on production of higher-precision values of the exponential and logarithm, as the fdlibm library, and Markstein's book [Mar00], do to remove the nasty effects of error magnification. In all of the ELEFUNT test programs, the relative error computation contains rarely seen bugs, as described in detail in **Section 4.25.4** on page 99.

The ELEFUNT test programs are not consistent in the naming of the variables that record approximate and accurate function values: they are sometimes `Z` and `ZZ`, and other times, reversed. The legacy of keypunches and short variable names in Fortran is evident here. It would have been better to name them more mnemonically as `FAPPRX` and `FACCUR`.

All of the test programs compute the absolute value of the current relative error in a variable `W`, and track the maximum relative error in a variable `R6` initialized to zero, and the argument value for that error in a variable `X1`. Those two variables are updated when a guard condition `IF (W .GT. R6)` is satisfied. Although that usually works, it fails if the tested function is always correct. The guard is then never satisfied, and `X1` is never initialized. The cure is simple: initialize `R6` to a negative value. The guard condition is then guaranteed to be satisfied at least once, and `X1` is always properly initialized.

Finally, there is a matter of coding style that deserves mention, because it makes the test code harder to read. The test programs set the initial test region $[a, b]$ outside the region loop, and then reset the test region just before the end of the loop, which might lie 150 lines away. That requires referencing the loop index with a value one larger than the current one, using a series of `if` statements. All of the new test programs developed for the mathcw library take a different approach: they set the test regions in a C `switch` statement at the beginning of the interval loop. That makes it easy to identify the current test region, because the regions are then clearly marked with `case` labels.

Were ELEFUNT rewritten from scratch today, the test programs would be much more modular, with more shared code, many small private functions and procedures, and sensibly named variables. However, to keep the new tests similar to the old ones, the temptation to do that rewrite has so far been resisted. However, translations from the original Fortran 77 to C and Java have been prepared, and the ELEFUNT suite has been compatibly extended to test more functions.

F Fortran interface

> FORTRAN IS NOT A FLOWER, BUT A WEED.
> IT IS HARDY, OCCASIONALLY BLOOMS,
> AND GROWS IN EVERY COMPUTER.
>
> — ALAN J. PERLIS (1987)
> FIRST ACM TURING AWARD WINNER (1966).

Fortran is the oldest high-level programming language still in use, and was the first programming language to be standardized by ANSI and ISO. Despite its age, it has no standard mechanism for communicating with routines written in other languages. The Fortran 2003 Standard [FTN04b] and Fortran 2008 Standard [FTN10] introduce such a facility, but it is unlikely to be widely implemented until ten to fifteen years after their publication. Donev [Don06] discusses some of the issues in using Fortran with C.

The first compiler that supported Fortran 77 was the `f77` compiler developed on UNIX systems at AT&T Bell Laboratories. Its calling conventions were carefully defined to be easily interfaced to C, because that was the most common language at the development site. Most subsequent UNIX Fortran compilers have followed the conventions of the original `f77` compiler, so the interface that we describe here works on most UNIX systems. Nevertheless, we are careful to provide mechanisms to facilitate adapting the interface to other conventions, and other operating systems.

Here is a summary of the default calling conventions in the original `f77` compiler:

- The Fortran language requires that it be possible for the called routine to modify any of the arguments in the caller. In practice, that means that all arguments are passed *by address*. Thus, arguments passed from Fortran to C are always *pointers*. By contrast, in C, scalar arguments are normally passed *by value*, and more complicated arguments, such as arrays, are passed *by address*.

  Although some Fortran compilers support special function-like wrappers, `%REF()` and `%VAL()`, to allow the programmer to choose whether an argument is passed *by address* or *by value*, they are inconvenient, error prone, and nonportable. It is much better to design the Fortran interface so that language extensions are not required to use it.

- External names in Fortran code are converted to lowercase, because the language is case insensitive, and are suffixed with an underscore to avoid colliding with names compiled from C programs, which are preserved as written.

- The Fortran floating-point data types `REAL` (synonym: `REAL*4`), `DOUBLE PRECISION` (synonym: `REAL*8`), and `REAL*16` correspond exactly to the C data types `float`, `double`, and `long double`, respectively.

- The Fortran data type `INTEGER` corresponds to C's `int` type. Length-qualified integer types are rare in Fortran programs, but when used, `INTEGER*1` (synonym: `BYTE`) maps to `signed char`, `INTEGER*2` maps to `short int`, `INTEGER*4` to `int`, and `INTEGER*8` to `long long int`.

- The Fortran `LOGICAL` data type corresponds to the C integer type `int`. The corresponding Fortran constant `.FALSE.` is zero in C, and the constant `.TRUE.` is nonzero (usually ±1) in C.

- Old-style Fortran 66 Hollerith strings (e.g., `12Hhello, world`) are passed by the address of the first character, but are *not* NUL-terminated. The string length is not passed, unless the user supplies an explicit argument to do so, as is common practice with Hollerith strings.

- Fortran 77 `CHARACTER` strings are passed by the address of the first character, but are *not* NUL-terminated, because all characters are allowed in a Fortran string. The length is passed *by value* as an additional argument at the end of the declared argument list. Such arguments are called *hidden arguments*.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2

If there are multiple CHARACTER arguments, their lengths are passed at the end of the normal argument list in their order of occurrence. Thus, the Fortran call `f('ab','cda')` can be received in C with a function declared as `f_(char *s, char *t, int len_s, int len_t)`.

That convention means that Hollerith and quoted strings can be received the same way, easing the transition from Fortran 66 to Fortran 77.

- Arrays are passed *by address* of the first element in both Fortran and C, but Fortran array indices start at one, as long as the array dimension is not a colon-separated range, whereas C indices always start at zero.
- The Fortran language requires contiguous storage of multidimensional arrays, with the *first* subscript increasing most rapidly. Multidimensional arrays in C are also stored contiguously, but the *last* subscript increases most rapidly. If data copying is to be avoided, subscript lists must be reversed. Thus, the Fortran array reference `x(i,j,m,n)` becomes the expression `x[n-1][m-1][j-1][i-1]` in C.

  The subscript reordering and adjustment-by-one are highly error prone, but subscripting errors can be largely avoided if the array references in C are hidden in a suitable macro. That way, the C code can use `X(i,j,m,n)`, just like the Fortran code.
- Fortran supports passing array dimensions at run time, but C does not until the 1999 ISO Standard.

  Fortran dimensions may appear anywhere in the argument list, whereas C99 requires them to appear before the array argument declaration in which they are used, following its principle of *declaration-before-use*.

  That restriction may make it impossible to maintain identical argument order for routines supplied in both languages, because the common practice in many long-lived Fortran libraries is for array arguments to be immediately followed by their dimension arguments.
- Programmers must be aware of the different array storage orders in the two languages. In Fortran, the first subscript increases most rapidly. By contrast, in the C family, the last one does. Storage-access order can have a huge effect on cache use and job times, so great care is called for in mixed-language array processing.

Although Fortran 66 and Fortran 77 limit names of variables and routines to six characters, few Fortran compilers produced since the 1970s enforce that length limit. That limit was chosen because it allowed a variable name to be packed into a single machine word in the cramped memories of the early IBM mainframes on which Fortran was developed in the mid-1950s; they had six-bit characters and 36-bit words. Fortran 66 and Fortran 77 retained the six-character limit, but Fortran 90 finally raised it to 31 characters.

In the mathcw library, only a few functions have array arguments: just the `fmul()` and `qert()` families, several of the pair-precision routines, the complex primitives prefixed `cx`, the vector functions prefixed `v`, and some of the special mathematical functions. Only a small number of routines take string arguments. Thus, some of the complexity of the interlanguage communication can be avoided in our interface.

There are, however, two troublesome areas — CHARACTER arguments and external names:

- A few Fortran compilers pass CHARACTER data *by address* of a *descriptor*, which is a compiler-dependent data structure that might include at least the length, and the address of the first character.

  Other compilers may pass such data by the address of the first character, and then supply a length argument immediately following the CHARACTER argument.

  Compilers for some other languages store the string length in the first one or two bytes of the string, but this author has never encountered that practice in a Fortran compiler.
- Some Fortran compilers convert external names to uppercase, and others omit the trailing underscore, although they may offer a compile-time option to restore the underscore.

  If Fortran external names cannot be distinguished from C names, then the only alternative is to rename all of the library functions. That is most easily done by adding a unique prefix to the Fortran-visible names.

We can easily cater to the naming problems by wrapping function names in the C interface in a macro with lowercase and uppercase arguments. However, alternate CHARACTER argument conventions require a revision of our code for those few functions that have string arguments.

Our default implementation follows the original `f77` conventions, providing support for a wide range of compilers on UNIX systems.

F.1 Building the Fortran interface

Before we look at the source code for the Fortran interface to the mathcw library, we first show how to build the interface.

Each library function has an interface wrapper function, and because few UNIX linkers discard unreferenced code from object files inside libraries, each wrapper function has its own file. Otherwise, if all of the wrappers were in a single file, then each use of the library would require loading all of the functions into the executable program, wasting time and filesystem space.

A directory listing of the Fortran interface subdirectory is lengthy, so we show only part of it:

```
% cd fort
% ls
Makefile  erfl.c    ierfc.c   l101pf.c  nanf.c    sign.c
acos.c    exp.c     ierfcf.c  l101pl.c  nanl.c    signf.c
...
erfcl.c   hypotl.c  isunl.c   modfl.c   setxpf.c  urcwl.c
erff.c    ierf.c    l101p.c   nan.c     setxpl.c
```

The Makefile manages the build process, which requires compilation of all of the C files, and then creation of a static load library from the object files:

```
% make

cc -g -DHAVE_LONG_DOUBLE -DHAVE_LONG_LONG -I.. -c acosf.c

cc -g -DHAVE_LONG_DOUBLE -DHAVE_LONG_LONG -I.. -c acoshf.c

...

cc -g -DHAVE_LONG_DOUBLE -DHAVE_LONG_LONG -I.. urcw4l.c

cc -g -DHAVE_LONG_DOUBLE -DHAVE_LONG_LONG -I.. ftocs.c

ar r libfmcw.a acosf.o acoshf.o ...

ranlib libfmcw.a || true
```

The peculiar construction with the shell OR operator, `||`, in the last command is needed because some UNIX systems do not have the `ranlib` utility for generating a library index. The first part then fails, but the subsequent `true` command ensures overall success.

A companion shared load library is easily built as well (here, on SOLARIS):

```
% make shrlib

cc -G -o libfmcw.so acosf.o ... -L.. -L. -lmcw
```

A simple validation test compiles and links some Fortran files with the new library, then runs them and compares their output with saved output files that are known to be correct:

```
% make check

f77 -g  -c -o test01.o test01.f

f77 -g -o test01 test01.o -L.. -L. -lfmcw -lmcw

...

There should be no output except the test names.
test03 requires support for REAL*16 in Fortran.
```

```
========== test01
========== test02
========== test03
```

When shared libraries are used, most UNIX systems require additional options to get the linker to record in the executable program the location of the shared library. Otherwise, even though linking succeeds, the program cannot be run because of missing libraries. GNU compilers generally need an option like `-Wl,-rpath,/usr/local/lib`, whereas others may require `-R/usr/local/lib`.

Finally, we install the new libraries in a standard location for use by others:

```
% make install
...
Installed files...
-rw-rw-r-- 1 mcw 893794 Mar 31 06:40 /usr/local/lib/libfmcw.a
lrwxrwxrwx 1 mcw     16 Mar 31 06:41 /usr/local/lib/libfmcw.so -> libfmcw.so.0.0.0
-rwxrwxr-x 1 mcw 396328 Mar 31 06:40 /usr/local/lib/libfmcw.so.0.0.0
```

F.2 Programming the Fortran interface

The Fortran argument-passing conventions described on page 941 suggest a simple interface design that we record in a small header file, `fmcw.h`, for use in all of the interface code:

```
#if !defined(FMCW_H)
#define FMCW_H

#include <mathcw.h>

#define CONS(a,b)               a##b
#define FTN(lcname,ucname)      CONS(lcname,_)

extern const char * F_to_C_string(const char *, int);

#endif /* !defined(FMCW_H) */
```

The `FTN()` macro handles the mapping of function names in the C code to the form produced by the Fortran compiler. Here, we choose the lowercase form with an underscore suffix.

The interface functions that take only numeric arguments are then similar, and a single example, `frexpf.c`, serves for all of them:

```
#include "fmcw.h"

float
FTN(frexpf,FREXPF)(float *x, int *n)
{
    return (frexpf(*x, n));
}
```

Because the C library functions in the `frexp()` family expect the second argument to be a pointer to a location for storing the power-of-two exponent, the variable `n` is not dereferenced with an asterisk in the `return` statement expression.

The few functions that require string arguments are only slightly more complex, and just one example, `nanf.c`, suffices:

```
#include "fmcw.h"

float
FTN(nanf,NANF)(const char *tag, int len)
```

```
{
    return (nanf(F_to_C_string(tag, len)));
}
```

The `len` argument corresponds to the hidden argument that the Fortran compiler passed *by value*.

Because there is no terminating NUL in a Fortran string, we must copy the passed `CHARACTER` string into a temporary location. That job is handled by the `F_to_C_string()` function.

In a general Fortran-to-C interface, we would have to allocate and free memory for the string copies, just as happens with the Java interface described in **Section J.3** on page 982. However, for the mathcw library, we never need more than one string at a time, and the strings are always short, so as long as a single thread has control, we can safely use a small static buffer, avoiding the considerable overhead of dynamic memory management.

Here is the code in `ftocs.c` that does the job:

```
#include <string.h>
#include "fmcw.h"

#define MAXBUF          64
#define MAX(a,b)        (((a) > (b)) ? (a) : (b))
#define MIN(a,b)        (((a) < (b)) ? (a) : (b))

const char *
F_to_C_string(const char *s, int len)
{   /* Convert Fortran string to C string, and return it in a
       static buffer that is overwritten on the next call. */

    static char buf[MAXBUF];

    buf[0] = '\0';
    (void)strncpy(buf, s, MAX(0,MIN(MAXBUF - 1, len)));
    buf[MAXBUF-1] = '\0';

    return ((const char*)&buf[0]);
}
```

The Standard C library function, `strncpy()`, is safe: it never copies more characters than the limit set by its third argument. However, it does not supply a trailing NUL if the destination string is smaller than the source string. We therefore create an empty string in the static buffer, then copy the argument string, and finally, ensure that the last character in the buffer is a NUL.

Fortran 77 does not allow zero-length character strings, so `len` should never be zero in that environment. However, language extensions are common in some compilers, and Fortran 90 permits zero-length strings, so it is best to program defensively. If `len` is zero, `strncpy()` does not copy anything at all, so the prior assignment to `buf[0]` is essential.

The third argument to `strncpy()` is an unsigned integer, so the `MAX()` macro wrapper ensures that a negative argument is treated as zero, instead of a large unsigned value that would result in overwritten memory, and a nasty program failure.

F.3 Using the Fortran interface

Because Fortran has no standard mechanism for file inclusion, and no notion of function and subroutine prototypes, we have to explicitly declare the mathcw library functions before using them. Here is one of the simple test programs:

```
      program test01
      real ierff, ierfcf
      real x
      integer k

      write (6,'(A5, 2A15)') 'x', 'ierff(x)', 'ierfcf(x)'
      do 10 k = 1,9
          x = float(k) / 10.0
          write (6,'(F5.2, 2F15.6)') x, ierff(x), ierfcf(x)
   10 continue

      end
```

Although the `Makefile` contains instructions for building the test program, all that we need to do is invoke a suitable Fortran compiler, with additional options to identify the location and names of the mathcw libraries. In this SOLARIS example, the `-ftrap` option is needed with the Fortran 90 and Fortran 95 compilers to get IEEE 754 nonstop operation:

```
% f95 -ftrap=%none test01.f -L/usr/local/lib -lfmcw -lmcw

% ./a.out
    x      ierff(x)      ierfcf(x)
 0.10      0.088856       1.163087
 0.20      0.179143       0.906194
 0.30      0.272463       0.732869
 ...
```

Systems managers can often specify local library locations in compiler configuration files, so that most users then need not know where the libraries are installed. Alternatively, the user can provide a colon-separated list of library directories in the `LD_LIBRARY_PATH` environment variable. However, here we simply supply an explicit library path with the `-L` option.

The Fortran interface library, `-lfmcw`, must be specified before the mathcw library, `-lmcw`, because on most UNIX systems, libraries named on the command line are searched only once.

H Historical floating-point architectures

> THERE ARE LESSONS TO BE LEARNED BY LISTENING TO OTHERS.
>
> — CHINESE FORTUNE-COOKIE ADVICE (2007).

> THE PROLIFERATION OF MACHINES WITH LOUSY FLOATING-POINT HARDWARE . . . HAS DONE MUCH HARM TO THE PROFESSION.
>
> — EDSGER W. DIJKSTRA
> *Structured Programming* (1972).

> COMPUTATION WITH ANY NUMBERS BUT SMALL INTEGERS IS A TRACKLESS SWAMP IN WHICH ONLY THE FOOLISH TRAVEL WITHOUT FEAR.
>
> — MAINSAIL AND FLOATING POINT FAQ
> *Is Floating Point Really This Complicated?* (1999).

Virtually all modern computer systems adhere to the IEEE 754 binary floating-point design, at least in format, if not always in details. New software can generally safely assume that system, but one of the goals of the mathcw library is that it should also be usable on important historical systems, some of which still run in simulators. Both the 36-bit DEC PDP-10 and the 32-bit DEC VAX are available that way, as are dozens of microprocessors, and even the venerable IBM System/360 mainframe. Simulated PDP-10 and VAX systems have been used as test platforms for the mathcw library, and even though they are implemented in software, thanks to advances in processor speeds, they run faster today than the original hardware ever did.

Table H.1 on the following page summarizes the floating-point characteristics of IEEE 754 and several historical computer systems. The variation in word size and floating-point formats is perhaps surprising, considering the uniformity of formats in machines built after 1980.

On most early systems, single- and double-precision formats have the same exponent range, and on some, the wider format has a phantom exponent field in the second word. By the 1960s, most vendors adopted a floating-point format that encoded a one-bit sign, then a biased exponent, followed by the significand magnitude. However, some systems use other formats. For example, the Harris and Pr1me systems store the exponent after the fraction, the General Electric 600 series stores the sign between the exponent and the fraction, and the Illiac IV stores two 32-bit values in a 64-bit word with their signs, exponents, and fractions interleaved in the order $s_1, e_1, s_2, e_2, f_1, f_2$.

When single and double formats share the same layout in the first word, a nasty and hard-to-find software bug arises when a call passes a double that is received as a single, or vice versa. If the first case, execution is correct. In the second case, whatever follows the single in memory is treated as the second word of the double. Those bits are likely to be arbitrary and unpredictable, but because they amount to a small perturbation on the fraction, their effect is simply to lower the precision of the double argument. In a large program, the error might be unnoticed, apart from the final results being puzzlingly less precise than expected. When the two formats have different layouts in the first word, the results are likely to be so nonsensical that the bug is recognized, tracked down, and repaired. Function-declaration prototypes force automatic type conversion, preventing detection of that kind of programming error. In any event, prototypes are absent from most early programming languages.

In the following sections, we describe the floating-point architectures of a few systems that have strongly influenced modern computers, operating systems, and programming languages. Each is introduced with a brief note about its significance in the history of computing. Because the machines overlap in time, they are presented in alphabetical order. A useful online historical archive of manuals on computer architectures and programming languages[1] provides more information about them. The *Digital Computer User's Handbook* [KK67] provides a view of the early computer market up to 1967. There is also an outstanding book, *Computer Architecture: Concepts and Evolution* [BB97], by two leading computer architects who describe machine designs up to about 1980.

[1] See `http://www.bitsavers.org/`.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2

Table H.1: Arithmetic of current and historical computer systems. The digit counts *exclude* the sign.

Vendor and model or family	Integer		Floating-point			
	base	digits	base	digits	digits	digits
				single	double	quadruple
Apollo Domain	2	31	2	23	52	—
Berkeley BCC-500	2	23	2	36	84	—
Burroughs B1700	2	33	2	24	60	—
Burroughs B5700	2	39	8	13	26	—
Burroughs B6700, B7700	2	39	8	13	26	—
CDC 1604	2	47	2	36	—	—
CDC 1700	2	15	2	23	39	—
CDC 3400, 3600	2	47	2	36	84	—
CDC 6000, 7000	2	48, 59	2	48	96	—
Cray 1, 1S, 2, X-MP, Y-MP	2	63	2	48	96	—
DEC PDP-6	2	35	2	27	—	—
DEC PDP-10 KA	2	35	2	27	54	—
DEC PDP-10 KI, KL, KS, KLH10	2	35	2	27	62	—
DEC PDP-10 G-floating	2	35	2	27	59	—
DEC PDP-11	2	15, 31	2	24	56	—
DEC PDP-12	2	11, 23	2	24	60	—
DEC VAX with D-floating	2	31	2	24	56	113
DEC VAX with G-floating	2	31	2	24	53	113
Data General Eclipse S/200	2	15	16	6	14	—
English Electric KDF 9	2	39	2	39	78	—
General Electric 600	2	35, 71	2	27	63	—
Gould 9080	2	31	16	6	14	—
Harris /6 and /7	2	23	2	23	38	—
Hewlett–Packard 1000, 3000	2	16	2	23	55	—
Honeywell 600, 6000	2	35, 71	2	27	63	—
IBM 360 family	2	31	16	6	14	28
IBM 650	10	10	10	8	—	—
IBM 704, 709	2	35	2	27	—	—
IBM 1130	2	15	2	23	—	—
IBM 1130 (extended `REAL`)	2	15	2	31	—	—
IBM 7040, 7044, 7090, 7094	2	35	2	27	54	—
IBM 7030 Stretch	2	63	2	48	—	—
IEEE 754 with 80-bit format	2	31	2	24	53	64
IEEE 754 with doubled `double`	2	31	2	24	53	106
IEEE 754 with 128-bit format	2	31	2	24	53	113
Illinois Illiac I	2	39	—	—	—	—
Illinois Illiac II	2	51	4	45	—	—
Illinois Illiac III	2	31	16	6	14	—
Illinois Illiac IV	2	47	2	24	48	—
Interdata 8/32	2	31	16	6	14	—
Lawrence Livermore S-1 Mark IIA	2	35	2	13	27	57
Manchester University Atlas	2	47	8	13	—	—
Pr1me 200	2	15	2	24	47	—
Rice Institute R1	2	47	256	6	—	—
SEL Systems 85, 86	2	31	16	6	14	—
Superset PGM	2	47	2	39	—	—
Univac 1100	2	35	2	27	60	—
Xerox Sigma 5, 7, 9	2	31	16	6	14	—
Zuse Z4	2	31	2	23	—	—

For a thorough treatment of the design, analysis, and programming of historical floating-point arithmetic systems, consult the often-cited book *Floating Point Computation* [Ste74]. Until IEEE 754 arithmetic became widespread, and well-documented in textbooks, that book was the standard treatise on computer arithmetic, with coverage of number systems, overflow and underflow, error analysis, double-precision calculation, rounding, base conversion, choice of number base, floating-point hardware, and complex arithmetic.

H.1 CDC family

> The double-precision floating-point package used by 1700 FORTRAN is described in this appendix. For efficiency the package is not run-anywhere.
>
> — CDC 1700 *Fortran Version 3A/B Reference Manual* (1974).

> The one major flaw of the CDC 6000 computer is the absence of automatic traps upon overflow conditions in integer arithmetic. Such conditions are simply ignored, and there is no way to detect them. . . . This very serious deficiency makes computations using integers rather hazardous.
>
> — Niklaus Wirth
> *The Design of a Pascal Compiler* (1971).

When IBM dominated the North American computer industry, a group of competitors during the 1960s became known as the BUNCH — Burroughs, Univac, NCR, CDC, and Honeywell. Control Data Corporation, or CDC, as it is usually known, produced a line of 60-bit computers that successfully gained a share of the scientific-computing market, outperforming IBM's 7030 Stretch.

Before the 60-bit systems, CDC sold the 48-bit 1604 (1961) and 3000 (1963) family and the 16-bit 1700 (1965). The 3000 series is notable for having support in Fortran for declaring numeric variables of a user-specified type, and then having the compiler generate calls to user-provided functions for the four basic numerical operations. That simplifies programming with extended-precision data types. Cody's first published paper [Cod64] describes a fast implementation of the square-root function for the CDC 3600.

In the 60-bit product line, the CDC 6200, 6400, 6500, 6600, and 6700 mainframes were followed by a much larger system, the CDC 7600, that gained fame in national laboratories as the world's fastest computer. One of the chief architects of those systems, Seymour Cray, left CDC in 1972 and founded a company, Cray Research Inc., that in 1976 shipped the first commercially successful vector supercomputer, the Cray 1. We discuss the words *vector* and *supercomputer* later in **Appendix H.2** on page 952.

Niklaus Wirth developed the Pascal programming language on a CDC 6400 system [Wir71a, Wir71b, Amm77]. The pioneering PLATO system for computer-assisted instruction was also developed on CDC machines [Kro10]. The 60-bit machines continued to evolve in the 1980s as the Cyber series, and another vector supercomputer company, ETA, was a spinoff from the CDC work.

The lead architect of the 6600 writes this about the choice of the 60-bit word size [Tho80, page 347]:

> The selection of 60-bit word length came after a lengthy investigation into the possibility of 64 bits. Without going into it in depth, our octal background got the upper hand.

The CDC 60-bit systems [Con67, Con71] have a floating-point format with a 1-bit sign, an 11-bit biased unsigned exponent-of-2, and a 48-bit *integer* coefficient. No hidden bit is possible with that representation. The coefficient corresponds to about 14 decimal digits, and that higher precision, compared to that available on 32-bit, 36-bit, and 48-bit systems, made the CDC systems attractive for numerical computing. Although CDC Fortran compilers provide a `DOUBLE PRECISION` data type, it is implemented entirely in software, and rarely used. It represents numbers as two 60-bit `REAL` values, with the exponent of the second reduced by 48, so the second coefficient is a bitwise extension of the first coefficient. The exponent size is not increased, so the number range is effectively that of the 60-bit form, but the precision is increased to 96 bits, or about 28 decimal digits.

Integer arithmetic, and exponent arithmetic, are done in one's-complement representation (see **Appendix I.2.2** on page 972). Although integer addition and subtraction work with 60-bit values, integer multiplication and division are provided by floating-point hardware, and thus require format conversion. Results of those two operations are limited to the 48-bit coefficient size. Similar economizations are found in several models of the 64-bit Cray vector supercomputers [Cray75, Cray82]. A 1971 CDC Fortran compiler manual reports:

> Because the binary-to-decimal conversion routines use multiplication and division, the range of integer values output is limited to those which can be expressed with 48 bits.

On neither of those CDC or Cray families is integer overflow on multiplication detected. A 1972 CDC Fortran compiler manual has this ominous remark:

> The maximum value of the operands and the result of integer multiplication or division must be less than $2^{47} - 1$. High-order bits will be lost if the value is larger, *but no diagnostic is provided*.

There are multiple representations of any number with a coefficient needing fewer than 48 bits, and the results of the four basic operations are not necessarily normalized. For multiplication and division, results are normalized only if both operands are as well. A separate normalize instruction shifts a nonzero coefficient left to produce a high-order bit of 1, and reduces the exponent by the shift count. That instruction is essential, because to obtain correct results, operands for division must first be normalized.

Perhaps because of the multiple representations, there is no floating-point comparison instruction. Instead, the numbers must be subtracted, the result normalized, and finally unpacked by a single hardware instruction into an exponent and fraction, and the sign and exponent tested to determine whether the difference is negative and nonzero, zero, or positive and nonzero.

Octal notation is conventional on the CDC systems. The exponent bias is $2000_8 = 1024_{10}$, so the value 1.0 can be represented in unnormalized form with an exponent of 2000_8 and a coefficient of 1, or in normalized form by shifting the coefficient left until its most significant bit is in the high-order position, and reducing the exponent accordingly. A short hoc program illustrates these 48 equivalent encodings:

```
hoc80> k = 0
hoc80> printf("%2d  %04o_%016...4o\n", k, 1024 - k, (1 %<< k))
hoc80> for (k = 1; k < 48; ++k) \
hoc80>     printf("%2d  %04o_%016...4o\n", k, 1023 - k, (1 << k))

 0  2000_0000_0000_0000_0001
 1  1776_0000_0000_0000_0002
 2  1775_0000_0000_0000_0004
 3  1774_0000_0000_0000_0010
 4  1773_0000_0000_0000_0020
 5  1772_0000_0000_0000_0040
...
42  1725_0100_0000_0000_0000
43  1724_0200_0000_0000_0000
44  1723_0400_0000_0000_0000
45  1722_1000_0000_0000_0000
46  1721_2000_0000_0000_0000
47  1720_4000_0000_0000_0000
```

The biased exponent of 1777_8 is skipped for a reason to be explained shortly.

The exponent encoding allows a fast way to subtract the bias: simply invert the high-order bit of the biased exponent, an operation that is easy to implement in hardware. Selective bit inversion is done by an exclusive-OR operation with a mask containing 1-bits wherever inversion is needed. To see that, consider two biased exponents, 2040_8 and 1720_8, corresponding to unbiased values of 40_8 and -57_8. Performing an exclusive-OR with the mask 2000_8 on the biased values produces 0040_8 and 3720_8. The first is clearly correct, and the second is as well, because $3720_8 = 3777_8 - 0057_8$ is just the one's-complement of the negative of 0057_8.

The CDC floating-point arithmetic system supports Infinity and Indefinite bit patterns, which are the inspiration for IEEE 754 Infinity and NaN. Those features also provide the nonstop model of computation that IEEE 754 adopted. Although Zuse's pioneering machines developed in Germany in the late 1930s had both special values, the Zuse computers were not well-known elsewhere, and it is unknown to this author whether the CDC architects reinvented the special values independently, or simply borrowed Zuse's idea.

The largest stored exponent, 3777_8, represents positive Infinity, and a stored sign and exponent of 4000_8 corresponds to negative Infinity. Because a one's-complement system has both positive and negative zeros, an exponent field of 1777_8 corresponds to -0, and would be redundant with 2000_8, the encoding of $+0$. The CDC architects therefore chose the exponent value 1777_8 to indicate Indefinite. In both cases, the coefficient bits are arbitrary, but the hardware always generates a zero coefficient for those exceptional values. There are special branch instructions that recognize the two special exponents, and arithmetic units check for them early, and return a predefined result when they are encountered.

The CDC operating systems provide commands to initialize unused memory to Infinity or Indefinite, with each coefficient set to the word's own address. Machine status flags can be set to cause job termination for an Indefinite and/or Infinite operand, and the ensuing report of the special operand value holds a record of where it came from, and gives a call traceback, identifying the routine and line number last executed, thereby detecting erroneous use of an uninitialized variable. Few machines before or since have made it as easy to detect that all-too-common software bug. Some modern compilers provide a `-trapuv` option to request such initialization, but that may still not catch uninitialized-access errors in dynamically allocated memory. The real solution to the problem is in hardware: memory systems should refuse to return a value from an unset location.

The representable number range is $[2^{-1023}, (2^{47}-1) \times 2^{1022}]$, or approximately [`1.11e-308`, `1.26e+322`], for unnormalized values. However, compilers enforce normalization, reducing the range to $[2^{-976}, (2^{47}-1) \times 2^{1022}]$, or approximately [`1.56e-294`, `1.26e+322`]. The choice of normalization, an integer coefficient, and the mid-way bias of 2000_8, makes the upper limit much larger than the reciprocal of the lower limit (by a factor of $2^{94} \approx 1.98 \times 10^{28}$), which is uncommon in floating-point designs. On most other systems, the two values are close. For example, in the IEEE 754 binary and decimal formats, the reciprocal of the smallest number is smaller than the maximum representable number by a factor of β^2. On the DEC VAX, the reciprocal is about two times larger than the overflow limit, on the Cray 1 and DEC PDP-10, almost four times larger, and on IBM System/360, nearly 256 times larger.

Although the floating-point system can represent a negative zero, the hardware never generates such a value. It does recognize them, so that $-\infty / -0 \to +\infty$ and $\infty / -0 \to -\infty$, but for finite operands, the sign is lost: $+1 \times -0 \to +0$. That inconsistency makes negative zeros unreliable records of their origin as underflows of negative values.

Underflows on the CDC machines normally flush to zero silently. However, Fortran compilers set a trap so that underflows are caught and reported, then set to zero. Although it is possible to trap use of an Infinity or Indefinite operand, the normal practice is to continue execution. Infinity and Indefinite propagate through most subsequent computations, so if they arise, they are likely to be noticed in the program output. However, the much wider exponent range of the CDC number format compared to that of competing machines with smaller word sizes makes underflow and overflow less frequent in most programs.

The default floating-point instructions on the CDC systems truncate their results, but there is a companion set that produces rounded results. CDC compilers allow users to choose either set, although the default is truncated results, perhaps because rounding arithmetic is slower.

Arithmetic operations do not use a guard digit, which leads to anomalies that we discuss in **Section 4.6** on page 66. For example, consider the subtraction of two adjacent numbers, simplified to a three-digit decimal system. We find correctly that $1.01 - 1.00 = 0.01$, but if we instead subtract the next lower pair, $1.00 - 0.999$, we first have to align the decimal points and work with at most three digits, giving $1.00 - 0.99 = 0.01$. That differs from the *exactly representable* correct answer 0.001 by a factor of 10, which is just β, the base. For a similar computation in binary arithmetic, the computed answer is twice the correct value.

For more on the history of the CDC 60-bit computer family, see the book [Tho70] and article [Tho80] written by the lead designer of the 6600.

In the early 1970s, work began on a new 64-bit architecture, the CDC 8600. The project was canceled in 1974, but the design manual specifies a floating-point format consisting of a 1-bit sign, a 1-bit *out-of-range* flag, a 14-bit biased exponent in one's-complement form, and a 48-bit integer coefficient. That provides normalized numbers in the range $[2^{-8192+47}, (2^{48}-1) \times 2^{8192}]$. In C99 notation, that is [`0x1p-8145`, `0xffff_ffff_ffffp+8192`], or about [`1.29e-2452`, `3.07e+2480`]. The precision is about 14 decimal digits. The out-of-range flag is set for overflow and undefined results, merging the old Infinity and Indefinite values into a single bit. Integer arithmetic is a modified one's-complement form where the hardware replaces the all-bits-one pattern of -0 with all-bits-zero for $+0$, simplifying later tests for zero.

With the 8600 abandoned, CDC moved on to a new 64-bit vector machine, the STAR-100. That system provides both 32-bit and 64-bit floating-point arithmetic. The 32-bit format has an 8-bit biased exponent, followed by a 1-bit sign, and a 23-bit integer coefficient. However, the exponent range is restricted, to allow exponents in [`0x70`, `0x7f`] to represent Indefinite, and [`0x80`, `0x8f`] to represent zero. The 64-bit format uses the same layout, but increases the exponent field to 16 bits and the coefficient to 47 bits (about 14 decimal digits). A leading hexadecimal digit of 7 or 8 in the exponent again identifies Indefinite and zero, respectively. The number range is less obvious because of the exponent restrictions, but the Fortran manual claims the range of the 64-bit format to be about [`5.19e-8618`, `9.53e+8644`]. For reasons discussed in the next section, that machine was a commercial failure, and CDC eventually went out of business when its 60-bit product line ceased to attract new customers.

H.2 Cray family

> REVERSING THE MULTIPLIER AND MULTIPLICAND OPERANDS COULD CAUSE SLIGHTLY DIFFERENT RESULTS, THAT IS, $A \times B$ IS NOT NECESSARILY THE SAME AS $B \times A$.
>
> — *Cray 1 Computer System Hardware Reference Manual* (1977).

> VALUES y AND z MAY EXIST FOR WHICH $|y| \leq z$ BUT THE CRAY 1 CALCULATES $|y \times (1/z)| > 1$.
>
> — W. KAHAN
> *Why Do We Need a Floating-Point Arithmetic Standard?* (1981).

The Cray vector supercomputer models 1, 1S, 2, X-MP, and Y-MP are 64-bit *word-addressed* systems [Rus78].

The adjective *vector* means that the machines have instructions capable of completing a load, or a store, or a floating-point operation, on one or two one-dimensional arrays of numbers, with the result in another array. With the technology of the time, the Cray vector instructions produce one result element per clock tick, a performance rate approached by later RISC processors with superscalar pipelined operation, using multiple arithmetic units and complex, and heavily overlapped, instruction execution.

The noun *supercomputer* refers to a computer that is engineered to provide the utmost floating-point performance compared to its price competitors, even if that means taking shortcuts that compromise the design or function [EM94]. The famous LINPACK benchmark report[2] and the list of the TOP 500 supercomputer sites[3] continue to track progress in that area. The worldwide market for such machines is small, but the prices are so high that Cray had vigorous competition from at least CDC, ETA, Fujitsu, Hitachi, IBM, and NEC.

There was even a market in the 1980s and 1990s for mid-range parallel and/or vector machines, sometimes called *departmental supercomputers*, with machines from Alliant, Ardent, Convex, Cray, Cydrome, DEC, ELXSI, Encore, Floating-Point Systems, Kendall Square Research, Multiflow, Pyramid, Sequent, Stardent, Stellar, Supertek, Thinking Machines, and others. Although all but Cray failed or were bought by competitors, cheap parallel and vector processing lives on in a low-cost, but high volume, market: graphics display controllers for desktops and game systems. Advances in microprocessor technology, and possibly a 'killer application', could yet put vector processing into commodity processors on the desktop. Research projects at Stanford University and the University of California, Berkeley have produced such designs, and one large and important application is the signal processing needed in wireless telephones.

Some vector machines permit arbitrary vector lengths, as the CDC STAR-100 and ETA machines do with vectors of up to 65,536 elements stored in memory. However, the Cray systems store at most 64 elements of data in any of several fast vector registers inside the CPU. To accommodate shorter vectors, a separate count register indicates how many elements are actually to be used. Longer vectors are handled by splitting them into 64-element chunks, with a possible remaining smaller chunk, and then processing each chunk with a single vector instruction. The Cray approach proved to be superior: a descendant of the original company survives four decades later, but ETA went out of business because its performance on its customers' software could not match that of its chief competitor. Lack of adequate memory bandwidth to feed powerful CPUs continues to be a challenging problem faced by all computer vendors, and the performance gap between memory and CPUs has widened dramatically since the introduction of vector supercomputers [Bee94].

The listed Cray models provide two's-complement arithmetic for 24-bit and 64-bit integers, with instructions for integer addition and subtraction. The integer multiply instruction is available only for the 24-bit address format, so 64-bit multiplication, and integer division, must be done by conversion to and from floating-point. Like the older CDC mainframes, that means that the effective integer width is reduced if multiplication or division are needed, and integer-overflow detection is impractical. Also like the CDC systems, there is no guard digit for floating-point subtraction.

The Cray 2 does subtraction slightly differently from the others, but not enough to repair the subtraction problem, leading to anomalies that can cause a cautiously programmed Fortran statement like

```
    IF (x .NE. y) z = 1.0 / (x - y)
```

[2] See `http://www.netlib.org/benchmark/hpl/`.

[3] See `http://www.top500.org/`.

to fail with a zero-divide error on Cray systems. The guard can also fail on modern IEEE 754 systems when subnormals are not supported.

The Cray floating-point format has a 1-bit sign, a 15-bit exponent-of-2 biased by $4\_0000_8 = 4000_{16} = 16\,384_{10}$, and a 48-bit fraction. There is no hidden bit. Although it is possible to construct unnormalized values, the hardware requires normalized inputs. There is no separate normalization instruction: addition to $+0$ provides that operation, and is primarily needed for the frequent conversion from integer to floating-point form.

Just as 16-bit PDP-11 users continued with the octal notation of their older 12-bit and 18-bit systems, instead of switching to the more suitable hexadecimal form, Cray designers carried the use of octal over from their CDC past.

The exponent size suggests that the biased exponent range should be $[0, 7\_7777_8]$, but that is not the case: stored exponents are restricted to the range $[2\_0000_8, 5\_7777_8]$. Values outside that interval are considered to have underflowed or overflowed. That decision was perhaps made to simplify the vector arithmetic units so that underflow and overflow could be handled quickly. The allowed number range is thus [0x.8p-8192, 0x.ffff_ffff_ffffp+8191] in C99 notation, or about [4.54e-2467, 5.45e+2465], and the precision is about 14 decimal digits.

Although -0 is representable, the hardware never generates such a value. Underflows are silently set to zero, and represented by a word with all bits zero. Overflows produce a value with the computed coefficient, and the exponent clamped to 6_0000_8. That provides Infinity, but there is no analogue of CDC's Indefinite.

Early Cray 1 machines used a multiplication algorithm that dropped some of the lower digits in the schoolbook method of summing digit-at-a-time products, with the result that multiplication could fail to commute, as the epigraph beginning this section notes. That misfeature was sufficiently upsetting to customers that it was later repaired by an engineering change. For a description of the original multiply algorithm, see [Cray77, pages 3-24ff]. The revised algorithm still drops trailing bits, but with more care [Cray82, pages 4-26ff].

There is no divide instruction: instead, a reciprocal-approximation instruction provides a starting estimate, accurate to 30 bits, for one step of a Newton–Raphson iteration to produce a reciprocal. That introduces additional rounding errors, making division less accurate than it should be, with errors in the last three bits. It also leads to misbehavior near the underflow limit, because in the available exponent range, y may be representable although $1/y$ overflows. For example, for y near the smallest representable number, the expression $0.06/y$ on those Cray systems is about a quarter of the largest representable number, but instead, it erroneously overflows.

Similar problems can arise even on modern machines when optimizing compilers replace divisions inside a loop with multiplications by a reciprocal computed outside the loop. In IEEE 754 binary and decimal arithmetic, the reciprocal of a number near the overflow limit is subnormal, either losing significant digits, or abruptly underflowing to zero when subnormals are absent.

It is worth noting that iterative improvement of a reciprocal approximation *can* be made to produce a correctly rounded quotient. The Intel IA-64 also lacks a divide instruction, but a careful implementation that exploits fused multiply-add and an extended exponent range allows the lengthy computation to be interleaved with other work, and can be proved correct [Mar00, Chapter 8].

Double-precision arithmetic is not supported in hardware on Cray systems, but is provided in software. For performance reasons, it is rarely used.

For more on the Cray floating-point design, see the essay *How Cray's Arithmetic Hurts Scientific Computation* [Kah90].

Models of the Cray computers from the late 1990s are based on either commodity (Alpha and SPARC) or custom processors, and provide almost-conforming 32-bit and 64-bit IEEE 754 arithmetic: only subnormals are omitted. Unfortunately, for tiny numbers, that omission destroys the property that with gradual underflow, $x - (x - y)$ correctly evaluates to y within rounding error when $x - y$ underflows, whereas with flush-to-zero underflow, the result is x. On the newer Cray systems, floating-point compare instructions avoid having to do comparisons by tests on the sign and magnitude of the operand difference.

H.3 DEC PDP-10

> 21 963 283 741 IS THE ONLY NUMBER SUCH THAT IF YOU REPRESENT IT ON THE PDP-10 AS BOTH AN INTEGER AND A FLOATING-POINT NUMBER, THE BIT PATTERNS OF THE TWO REPRESENTATIONS [$243\_507\_216\_435_8$] ARE IDENTICAL.
>
> — NEW HACKER'S DICTIONARY (1993).

> CORRECT IMPLEMENTATION OF FLOATING POINT IS SUFFICIENTLY OBSCURE THAT IT IS HARD TO DESIGN A HARDWARE FLOATING-POINT UNIT CORRECTLY.

— THE TEAM THAT BUILT THREE PDP-10 CLONES AT XEROX PARC (1978).

The Digital Equipment Corporation (DEC) 36-bit PDP-10 is a descendant of the 36-bit PDP-6, which appeared in late 1964. The PDP-10, introduced in 1967, ran more than *ten* different operating systems, and was the vendor's largest computer until it was surpassed by the biggest VAX models in the mid 1980s. During its long life, a small clone market arose, with compatible machines built by Foonly, Systems Concepts, Xerox, and XKL.

The Xerox effort did not lead to a commercial product, but the architects at Xerox PARC extended their home-built PDP-10 systems with many new instructions for enhanced support of the Lisp language. That practice of tweaking the instruction set while designing a language later led to the development of the Xerox Alto and Dolphin computers, and the Cedar, Mesa, and Smalltalk languages. The Alto is credited with being the first personal workstation, with the first graphical user interface (GUI), and the first pointing device, the mouse. All desktop systems today share that legacy.

The PDP-10 is one of the earliest commercial systems to have integrated networking, and has outstanding timesharing services. Its registers overlap the first sixteen words of memory, but are implemented in faster logic, so small instruction loops run several times faster if moved into the register area. The PDP-10 has a rich instruction set, and its architecture is well-suited to the Lisp language, whose implementation makes heavy use of object pointers. A significant feature of the PDP-10 is an *indirect pointer*: a word containing an address and a flag bit that, when set, means that another pointer should be fetched from that address, and the process repeated until a direct pointer is found. Thus, a single memory address in an instruction can transparently follow a long chain of linked pointers.

The richness and versatility of the PDP-10 led to its wide adoption in universities, particularly in computer-science departments, and it became the machine of choice for most of the early Arpanet sites. That network ultimately evolved into the world-wide Internet.

Several influential programming languages, typesetting systems, and utilities were first developed on the PDP-10, including BibTeX, Bliss, `emacs`, Interlisp, `kermit`, LaTeX, MacLisp, Macsyma, MAINSAIL, Maple, METAFONT, METAPOST, PCL, PSL, PUB, REDUCE, SAIL, Scribe, SPELL, and TeX. The CP/M operating system and Microsoft BASIC were both produced with simulators for the Intel 8080 running on PDP-10 systems before the Intel microprocessors were available, and Microsoft MS-DOS borrowed many features of CP/M. Microsoft built early versions of its WINDOWS operating system on top of MS-DOS. Later, it reversed their logical order, so modern WINDOWS systems still have the old operating system available via the `command` utility. The `mathcw` library described in this book also runs on the PDP-10 in the KLH10 (2001) simulator. For further historical highlights of the PDP-10, see [Bee04a], [Bee05], and [BMM78, Chapter 21].

The PDP-10 uses two's-complement integer arithmetic (see **Appendix I.2.3** on page 972), and the larger KL10 model also provides 72-bit integer add, subtract, multiply, and divide instructions, but the Fortran compiler has no support for a double-word integer. The sign bit of the second word is made the same as that of the first word, so the precision is 70 bits. There is also a fixed-point decimal instruction set for commercial applications, and powerful string instructions that can handle any byte size from 1 bit to 36 bits.

Single-precision floating-point arithmetic has a 1-bit sign, an 8-bit power-of-2 exponent, and a 27-bit normalized fraction. The exponent bias is $200_8 = 128_{10}$, so the number range is $[2^{-129}, (1 - 2^{-27}) \times 2^{127}]$. In C99 notation, that is [`0x0.8p-128`, `0x0.ffff_ffep+127`], or approximately [`1.47e-39`, `1.70e+38`]. The precision is about 8 decimal digits. For positive numbers, the format matches that of the IBM 7090, which was a member of IBM's largest family of 36-bit machines before System/360 was announced. However, for negative numbers, the encoding on the PDP-10 was changed to store the fraction in two's-complement form, with a one's-complement exponent. That allowed integer comparison instructions to be used for single-precision floating-point numbers. To see how that works, here are some experiments with hoc on the KLH10 (a software implementation of the PDP-10) to display the native encodings of positive and negative floating-point numbers in octal:

```
hoc36> for (x = 1.0e-2; x <= 1.0e2; x *= 10) {        \
hoc36>     y = x * PI;                                 \
hoc36>     z = PI / x;                                 \
hoc36>     printf("%9.3f  %s  %9.3f  %s\n", y, ftoo(y), -z, ftoo(-z)) }
    0.031  174401267745   -314.159  566305656352
    0.314  177501545736    -31.416  572011260566
    3.142  202622077325     -3.142  575155700453
   31.416  205766517212     -0.314  600276232042
  314.159  211472121426     -0.031  603376510034
```

```
hoc36> for (k = -3; k <= 2; ++k) {                   \
hoc36>     x = PI + k * macheps(PI);                 \
hoc36>     printf("  PI%+d*eps  %s  -PI%+d*eps  %s\n",
hoc36>            k, ftoo(x), -k, ftoo(-x)) }
  PI-3*eps  202622077322  -PI+3*eps  575155700456
  PI-2*eps  202622077323  -PI+2*eps  575155700455
  PI-1*eps  202622077324  -PI+1*eps  575155700454
  PI+0*eps  202622077325  -PI+0*eps  575155700453
  PI+1*eps  202622077326  -PI-1*eps  575155700452
  PI+2*eps  202622077327  -PI-2*eps  575155700451

hoc36> x = MINNORMAL
hoc36> printf("% 14.6e  %s  % 14.6e  %s\n", x, ftoo(x), -x, ftoo(-x))
  1.469368e-39  000400000000   -1.469368e-39  777400000000

hoc36> x = MAXNORMAL
hoc36> printf("% 14.6e  %s  % 14.6e  %s\n", x, ftoo(x), -x, ftoo(-x))
  1.701412e+38  377777777777   -1.701412e+38  400000000001
```

The octal values taken as integers clearly have the same order as the floating-point numbers. Had the exponent been in two's-complement form, negative numbers near the underflow limit would have stored exponents of 000_8, 777_8, 776_8, ..., instead of 777_8, 776_8, 775_8, ..., losing the integer ordering.

On the early KA10 processor (1967), double-precision arithmetic is provided in software, using a pair of single-precision numbers, with the exponent of the second word set to 27 less than that of the first word. That gives a precision of 54 bits, or about 16 decimal digits, with the same exponent range as the single-precision format. Short sequences of three to seven instructions implement the four basic operations on software double-precision numbers [DEC76, pages 2-79–2-80].

On the later KI10 (1972), KL10 (1975), and KS10 (1978) CPUs, hardware double precision uses two words, with a 1-bit sign, 8-bit power-of-2 exponent, and 62-bit fraction; the sign bit in the second word is unused, and ignored. The number range is therefore almost identical to that for single-precision arithmetic, but the precision is extended to about 18 decimal digits. Comparisons of values in the hardware double-precision format require three successive integer compare instructions.

In the late 1970s, the PDP-10 got a third double-precision format, called G-floating, in hardware. It widens the exponent to 11 bits, reduces the fraction to 59 bits (about 17 decimal digits), and continues to ignore the sign bit of the second word. The new range is [`0x.8p-1024`, `0x.ffff_ffff_ffff_ffep+1023`], or about [`2.78e-309`, `8.98e+307`]. Compiler options select between the two hardware double-precision formats, without the need to introduce new data types into programming languages.

Floating-point instructions are available in both truncating and rounding variants, with the rounding ones being the default for most programming languages. However, rounding is peculiar, as we describe and illustrate in **Section 26.3.5** on page 848. The design manual for the planned, but later canceled, KD10 (1983) notes that its rounding instructions were to produce correct unbiased rounding, so at least the problem was recognized, and scheduled to be fixed.

The results of addition, multiplication, and subtraction are produced in a double- or triple-length register (depending on the model) and then normalized and, if requested, rounded. Division includes at least one extra bit for rounding. Those actions provide a sufficient number of guard digits to prevent the anomalies of the Cray systems.

State flags record the occurrence of underflow, overflow, and zero-divide conditions, and for the first two, the fraction is correct, but the exponent has wrapped. Setting those flags normally traps immediately to a software interrupt handler that replaces overflows by the largest representable number, and underflows by zero. Unfortunately, the two available C compilers do not supply a handler, so underflow and overflow in that language produce incorrect results with small exponents wrapped to large ones, and vice versa, as we illustrate in **Section 4.9** on page 71 and **Section 4.11** on page 77. Zero divides are suppressed, and in the absence of a handler fixup, the result is the numerator. The Fortran handler sets the result of divide-by-zero to the correct sign and magnitude of the largest representable number.

H.4 DEC PDP-11 and VAX

> I FINALLY GOT TO TRY SOME PARALLEL PROGRAMMING AS A GRADUATE STUDENT IN 1975 AT THE UNIVERSITY OF UTAH. WE HAD AN ADD-ON TO OUR PDP-11 MADE BY A COMPANY CALLED FLOATING POINT SYSTEMS THAT DID PIPELINED FLOATING ADD/MULTIPLIES. UNFORTUNATELY, IT DIED BEFORE I COULD DEBUG MY PROGRAM.
>
> — JIM BLINN (1997).

> THE SCIENTISTS WERE UPSET BECAUSE THEY DID NOT YET HAVE A VAX TO GRIND.
>
> — JOHN W. D. CONNOLLY (1978).

The 16-bit DEC PDP-11 [EM79, LE80], introduced in 1970, was one of the most commercially successful minicomputers. Early systems provided software floating-point arithmetic in a three-word format, with a signed 15-bit exponent, and a signed 31-bit fraction. Later models offered an optional floating-point processor that provided 32-bit and 64-bit arithmetic in formats described later in this section.

Although early UNIX development had begun in 1969 on the 18-bit PDP-7, and was planned to be moved to the 36-bit PDP-10, budget limitations forced a fallback to the smaller PDP-11. The limited address space and instruction-set architecture of that system had a large impact on the design of both UNIX, and the C programming language, which was developed on the PDP-11 from 1969 to 1973. Ken Thompson, the chief architect of UNIX, had worked on the complex MULTICS operating system project (see **Appendix H.5** on page 958), and the name UNIX was chosen partly as a pun, and partly to reflect the designers' philosophy of *small is beautiful* [Gan95, Gan03, Ray04, Sal94].

The first 32-bit DEC VAX (for **V**irtual **A**ddress e**X**tension), announced in 1977, has a 32-bit floating-point format called F-floating, with a 1-bit sign, 8-bit exponent with a bias of 128, and a 24-bit fraction. That total of 33 bits is possible because of the trick of supplying an implicit leading bit of 1, called a *hidden bit*, when an arithmetic operation is performed. The first VAX also has a 64-bit format, called D-floating, with the same allocation of sign and exponent bits as the 32-bit size, and 56 fraction bits, again with a hidden bit.

In response to customer demand, DEC added the 64-bit G-floating format in 1979, with three bits moved from significand to exponent, giving a format similar to IEEE 754 64-bit arithmetic, but without NaN, Infinity, or gradual underflow. In most programming languages, the choice between the two `double` formats is made by a compile-time option, rather than by a change in a data type.

Larger models of the 16-bit DEC PDP-11 systems have the F-floating and D-floating formats, and indeed, for several years, VAX models included a PDP-11 compatibility mode that allowed them to run old code from those smaller systems. During that time, the name VAX-11 was common in DEC literature.

When DEC designed the 64-bit Alpha RISC architecture in the early 1990s with floating-point arithmetic based on the IEEE 754 Standard, it also included separate instructions for the VAX F-floating, D-floating, and G-floating formats, to facilitate run-time translation and execution of VAX programs on Alpha processors. In Alpha documentation, the two IEEE 754 formats are called S-floating and T-floating. Later releases of the OSF/1 operating system for the Alpha also provide the 128-bit IEEE 754 extension, implemented in software. More recently, Gentoo GNU/LINUX distributions for Alpha and SPARC CPUs added that 128-bit arithmetic.

When DEC added the G-floating feature, the VAX architecture also got a 128-bit format, H-floating, which is usually implemented in software. The only other system offering 128-bit floating-point arithmetic at the time was the IBM System/360 family, where it had been available for more than a decade (see **Appendix H.6** on page 959).

VAX floating-point instructions never generate negative zeros. The VAX therefore treats the floating-point bit pattern for a negative zero as a special value, called a *reserved operand*, that causes a run-time fault with all instructions that access it, even load and store instructions. Unless it is caught at run time and handled, that fault causes immediate program termination. In practice, that makes reserved operands useless for anything but user-specified compile-time initialization of floating-point storage areas. The PDP-11 has the same interpretation of -0, but calls it an *undefined variable*. Unlike the CDC operating systems, there is no support from compilers or operating systems on the PDP-11 or VAX for execution-time initialization with reserved operands.

Although as shown in **Figure H.1** on the facing page, VAX floating-point values in CPU registers have the modern format of sign, biased exponent, and significand, the VAX memory layout follows that of the PDP-11, which is *little endian* [Coh81] in 16-bit words, as illustrated in **Figure H.2** through **Figure H.4** on page 958. That puts the sign and exponent between two sequences of fraction bits in the byte string in memory [DEC77, DEC79, DEC82, DEC90].

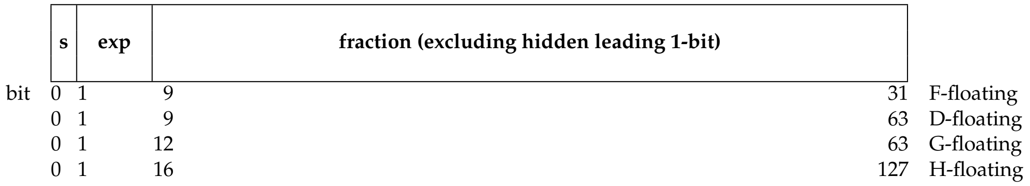


Figure H.1: VAX binary floating-point logical data layout. The sign bit is 1 for negative, and 0 for zero or positive. The exponent-of-2 is a biased unsigned integer.
For nonzero stored exponents, there is a hidden (not-stored) leading fractional bit of 1 that is implicitly prefixed to the stored fraction bits in any arithmetic operation. The binary point lies immediately to the left of the hidden bit. Thus, the fraction is always normalized, and nonzero fractions are in the range $[\frac{1}{2}, 1)$. The floating-point value is $(-1)^s \times (\text{fraction}) \times 2^{\text{exp}-\text{bias}}$.
When the stored exponent is zero, there is no hidden bit. Such a value with a sign bit of 0 represents a true zero. With a sign bit of 1, and arbitrary stored fraction bits, it is a *reserved operand* (see text for details).

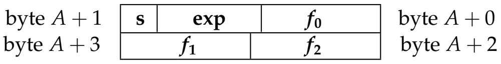


Figure H.2: VAX F-floating memory layout.
The memory address of the value is A, but data are stored in the order of 16-bit PDP-11 words, so the sign and exponent reside in the middle of a 32-bit field.
The exponent width is 8 bits, and the fraction width is 24 bits, representing about 7 decimal digits.
A hidden high-order fractional 1-bit logically precedes the stored fraction, so the significand value is the sequence $0.1 f_0 f_1 f_2$.
The exponent bias is 128, so the number range in C99 hexadecimal floating-point notation is [`0x0.8p-127`, `0x0.ffff_ffp+127`], or about [`2.94e-39`, `1.70e+38`]. A stored exponent of zero represents a numerical zero, independent of the value of the fraction.

Although that peculiar layout is transparent to most numerical software, library primitives that extract the sign, exponent, or significand need to know about it.

Zeros are recognized solely by a biased exponent of 0. When a floating-point instruction produces a zero result, the sign and fraction are always zero. Although bit manipulation and storage initialization can construct nonzero fractions with zero exponents, those values are treated as zero by the hardware.

Like many systems, the VAX includes condition-code flag bits that record the occurrence of underflow, overflow, zero divide, and a few others, and those status bits can be tested in conditional-branch instructions. The flags are sticky in the sense that floating-point operations never clear them, but might set them. The VAX calling sequence ensures that the condition codes are preserved across procedure calls, and that the call instruction sets them all to zero at procedure entry. That makes it possible, and reliable, to test them long after they have been set, even when there are intervening calls. It also allows any procedure to install a floating-point exception handler to deal with operand or result faults at that call level, or deeper, without disabling handlers established by the caller or its call-sequence ancestors. Exceptions are processed by the nearest registered handler in the call history, so most language implementations install a suitable language-dependent default handler just prior to calling the user's main program.

After 1986, the largest models of the VAX included vector instructions for 32-bit integers, and for D-, F-, and G-floating formats, and the VMS operating system supplied the VAX *Vector Instruction Emulation Facility* (VVIEF) for software support on models lacking the vector hardware. Vectors contain up to 64 elements, just as on the Cray models discussed in **Appendix H.2** on page 952.

byte $A+1$	s exp	f_0	byte $A+0$
byte $A+3$	f_1	f_2	byte $A+2$
byte $A+5$	f_3	f_4	byte $A+4$
byte $A+7$	f_5	f_6	byte $A+6$

Figure H.3: VAX D- and G-floating memory layout.
For D-floating, the exponent width is 8 bits, and the fraction width is 56 bits (1 hidden + 55 stored), giving a precision of about 16 decimal digits.
For G-floating, the exponent width is 11 bits, and the fraction width is 53 bits (1 hidden + 52 stored), with a precision of about 15 decimal digits.
The exponent bias is 128 for D-floating, and 1024 for G-floating.
The D-floating number range is [`0x0.8p-127`, `0x0.ffff_ffff_ffff_ffp+127`], barely larger than that for F-floating.
The G-floating range is [`0x0.8p-1023`, `0x0.ffff_ffff_ffff_f8p+1023`], or about [`5.56e-309`, `8.98e+307`].

byte $A+1$	s	exp	byte $A+0$
byte $A+3$	f_0	f_1	byte $A+2$
byte $A+5$	f_2	f_3	byte $A+4$
byte $A+7$	f_4	f_5	byte $A+6$
byte $A+9$	f_6	f_7	byte $A+8$
byte $A+11$	f_8	f_9	byte $A+10$
byte $A+13$	f_{10}	f_{11}	byte $A+12$
byte $A+15$	f_{12}	f_{13}	byte $A+14$

Figure H.4: VAX H-floating memory layout.
The exponent width is 15 bits and the fraction occupies 113 bits (1 hidden + 112 stored), providing a precision of about 33 decimal digits.
The exponent bias is 16384.
The range is [`0x0.8p-16383`, `0x0.ffff_ffff_ffff_ffff_ffff_ffff_ffff_8p+16383`], or about [`8.40e-4933`, `5.94e+4931`].

H.5 General Electric 600 series

> The Honeywell 6080 computer once had the property that
> a small number (approximately -10^{-39}) when divided by roughly 2 gave a
> large result (approximately 10^{38}). ... The computing world is a jungle
> of individualistic and sometimes too clever arithmetic units.
>
> – Norman L. Schryer
> *Determination of Correct Floating-Point Model Parameters*
> in *Sources and Development of Mathematical Software* (1984).

An influential early timesharing operating system, Multics, was developed on the General Electric 600 series from 1964 to 1969. The vendor's computer business was sold to Honeywell in 1970, and Multics systems operated until 2000. The 600 series is a 36-bit architecture, and provides two's-complement integer arithmetic (see Appendix I.2.3 on page 972) with 18-bit, 36-bit, and 72-bit sizes, and 36-bit and 72-bit floating-point arithmetic.

The floating-point storage layout has an 8-bit unsigned exponent (in both lengths) biased by 128, followed by a sign and a 27-bit or 63-bit fraction. Unlike the PDP-10, the exponent is stored in two's-complement form, and in double precision, the high-order bit of the second word is a data bit, rather than a sign bit.

Floating-point arithmetic is done in an extended accumulator pair containing an 8-bit exponent in register `E`, and a sign and a 71-bit fraction in register `AQ`. The two registers can be accessed separately, or jointly as `EAQ`. Loads from storage fill the extra bit positions in the accumulator with zeros. There are two 36-bit store instructions, one which truncates, and the other which first rounds by adding 1 at fraction bit 28, then stores the truncated result, approximating a *round-to-plus-infinity* operation. The 72-bit store merely truncates the accumulator value. Multiplication and division of the accumulator by a value in storage truncate the product or quotient. The extra bits in the accumulator fraction supply the important guard digits for subtraction.

If a sequence of operations reuses the accumulator value, the result is higher intermediate precision. Although that is normally beneficial, we have seen several times in this book that it can lead to surprises, and even be harmful. The General Electric and Honeywell systems are early examples of machines with arithmetic done in a precision higher than that of stored values. That practice continues in Intel and Motorola microprocessors introduced in the early 1980s, and remains widespread in desktop computers.

H.6 IBM family

> ARITHMETIC IN THE OBJECT PROGRAM WILL GENERALLY BY PERFORMED WITH SINGLE-PRECISION 704 FLOATING POINT NUMBERS. THESE NUMBERS PROVIDE 27 BINARY DIGITS (ABOUT 8 DECIMAL DIGITS) OF PRECISION, AND MAY HAVE MAGNITUDES BETWEEN APPROXIMATELY 10^{-38} AND 10^{38}, AND ZERO. FIXED POINT ARITHMETIC, BUT FOR INTEGERS ONLY, IS ALSO PROVIDED.
>
> — *FORTRAN Automatic Coding System for the IBM 704 EDPM* (OCTOBER 15, 1956).

> IT WAS POSSIBLE TO FIND VALUES x AND y FOR IBM SYSTEM/360, SUCH THAT, FOR SOME SMALL, POSITIVE ϵ, $(x + \epsilon) \times (y + \epsilon) < (x \times y)$. MULTIPLICATION HAD LOST ITS MONOTONICITY. SUCH A MULTIPLICATION IS UNRELIABLE AND POTENTIALLY DANGEROUS.
>
> — NIKLAUS WIRTH
> *Good Ideas, Through the Looking Glass*
> COMPUTER **39**(1) 28–39 (2006).

During the 1950s and early 1960s, IBM's main scientific computers were models in the 700 and 7000 series. All are 36-bit machines, with the 700 series offering only single-precision floating-point arithmetic, with a 1-bit sign, 8-bit biased exponent-of-2, and 27-bit fraction. The 7000 series added support for double-precision arithmetic, where the second word has the same layout as the first word, but with an exponent reduced by 27, providing a 54-bit fraction.

H.6.1 IBM 7030 Stretch

In late 1954, IBM researchers began work on *Project Stretch*, with the ambitious goal of building a scientific computer to be at least 100 times faster than any existing computer on the market. IBM delivered the first model, called the 7030 Stretch, to Los Alamos Laboratory in April, 1961. Although only nine machines were built, and performance did not quite reach the original goal, Project Stretch produced significant advances in hardware engineering that IBM leveraged in its next machine series. The Stretch design is well-chronicled in a book [Buc62], and a half-century retrospective [Ste11] from 2006 Turing Award winner Frances Allen, and has many innovations for its time. Here are just a few of them:

- The magnetic-core memory system contains 64-bit words, but the address space is bit-addressable. Variable-width fields of 1 to 64 bits can transparently cross word boundaries, albeit with a performance penalty.
- Hardware arithmetic supports binary and decimal fixed-point and integer, and binary floating-point, data.
- Some instructions specify an operand byte size, which may be any number in $[1, 8]$.
- The floating-point format has a 12-bit signed exponent field, and a 52-bit fraction field containing a 4-bit sign-and-flags field. The exponent field has a flag bit recording previous overflow or underflow, a 10-bit exponent, giving an approximate range of [`5.56e-308`, `1.80e+308`] (similar to that of the IEEE 754 64-bit binary format), and a final sign bit. The 48 fraction bits represent roughly 14 decimal digits, and are followed by the fraction sign bit, and three flag bits available for user-defined data tagging. The integer format also has that four-bit field.
- The 128-bit double-length format has a 96-bit fraction, with 20 unused bits at the bottom of the second word that are preserved in arithmetic, so they may contain user-defined data. The exponent range is unchanged from the 64-bit format.

- All arithmetic operations are done in a double-length accumulator, and there is no rounding until its value is stored to memory. The user can choose the truncating `store` instruction, or the slower `store rounded`, which adds one to the 49th fraction bit of the fraction magnitude before normalization and storage. That corresponds to rounding to $\pm\infty$, a mode that does not exist in the IEEE 754 design.

- The `store root` instruction produces the square root of the accumulator value and stores it to memory. The accumulator retains its original value.

- There are two kinds of zero: a *true zero*, with zero fraction, and an *order-of-magnitude-zero* (OMZ) that results from underflow in subtraction, or from subtraction of identical nonzero values. In an OMZ, the fraction is zero, but the exponent is nonzero. Most other architectures produce a true zero for the case of $x - x$.

- When the exponent flag bit is set, the value represents *Infinity* if the exponent sign is positive, and *Infinitesimal* (ϵ) if the exponent sign is negative. Arithmetic with a normal number, x, and one of those exceptional values behaves like this:

$$\begin{aligned} \infty \pm x &= \infty, & x - \infty &= -\infty, \\ x \pm \epsilon &= x & \epsilon - x &= -x, \\ \infty \times x &= \infty, & \epsilon \times x &= \epsilon, \\ \infty / x &= \infty, & \epsilon / x &= \epsilon, \\ x / \infty &= \epsilon, & x / \epsilon &= \infty. \end{aligned}$$

  Arithmetic with two exceptional values obeys these rules:

$$\begin{aligned} \infty + \infty &= \infty, & \infty - \infty &= \infty, & \infty \times \infty &= \infty, & \infty / \infty &= \infty, \\ \epsilon - \infty &= -\infty, & \epsilon / \infty &= \epsilon, & \infty \times \epsilon &= \infty, & \infty / \epsilon &= \infty, \\ \epsilon \pm \epsilon &= \epsilon, & \infty \pm \epsilon &= \infty, & \epsilon \times \epsilon &= \epsilon, & \epsilon / \epsilon &= \infty. \end{aligned}$$

  Notice that under IEEE 754 rules, the second and fourth on the first line would produce a NaN.

  Comparison rules are as expected:

$$\begin{gathered} +\infty > +x > +\epsilon > -\epsilon > -x > -\infty, \\ +\infty = +\infty, \qquad +\epsilon = +\epsilon, \\ -\infty = -\infty, \qquad -\epsilon = -\epsilon. \end{gathered}$$

- There is a run-time selectable *noisy mode* for floating-point arithmetic. When it is turned on, shifts in addition and subtraction extend one of the operands with 1-bits, instead of with 0-bits. The intent is that a user can run a program twice, once with each choice of the mode, and then compare the output to assess the impact of *significance loss* in intermediate calculations. The noisy-mode technique can be compared with those of *significance arithmetic* [AM59] and *interval arithmetic* [Gus98] for helping to judge the reliability of numeric results.

- A multiply-and-add instruction computes a double-length product and sum from single-word operands. It can be used to improve accuracy of dot products, matrix multiplication, and polynomial evaluation, but only in the 64-bit format.

- Division by zero is suppressed, and an exception flag is set. Thus, if interrupts are turned off, $x/0$ evaluates as x.

- Floating-point instructions provide for both normalized and unnormalized operation. The latter can be used for both significance arithmetic, and with the multiply-and-add instruction and exception flags, for software multiple-precision arithmetic.

- The floating-point accumulator is also used for integer arithmetic. The Fortran manual describes integer data encoded as unnormalized floating-point values with a fixed exponent of 38, a 38-bit value in the 48-bit fraction field with the lower 10 bits ignored, followed by the four-bit sign-and-flags field. Thus, integers for arithmetic have a sign-magnitude representation (see **Appendix I.2.1** on page 971), with a largest value of 274 877 906 943. However, integers used in logical operations can be any size from 1 to 64 bits, and may be either signed or unsigned.

- Fortran `LOGICAL` data are stored as floating-point values with the low-order fraction digit (bit 59) set to 1 for *true*, and 0 for *false*. The exponent, signs, and flags are then ignored.

The Stretch book has this remark on the consistency of arithmetic with exceptional values:

> For example, $\epsilon - \epsilon = \epsilon$ implies that infinitesimals are equal, but $\infty - \infty = \infty$ implies that infinities are different. This problem arises because no consistent logic applies when both operands are singular.

The two kinds of zero pose a problem too:

> Since an OMZ represents a range of indeterminacy, multiplication or division by a legitimate number simply increases or decreases the size of the range of indeterminacy appropriately. Division by an OMZ is suppressed and, when it would occur, the zero divisor indicator is turned on. Addition of an OMZ to either a legitimate operand or another OMZ produces either a legitimate result or an OMZ, depending upon the relative magnitudes of the quantities involved. (However, comparison operations call equal all OMZs whose exponents differ by less than 48.)

A *zero-multiply flag* is turned on whenever multiplication produces a zero fraction, allowing the programmer a choice of whether to fixup the result to a true zero, or to an OMZ.

Arithmetic, and its storage formats, on the Stretch architecture are unusual, and complicated, compared to most other designs produced since the early 1960s.

Here is a final historical note: in May 1971, researchers at Brigham Young University in Provo, Utah, acquired one of the Stretch machines from government surplus, restored it, and ran it successfully as the last survivor until it was retired in September, 1980.

H.6.2 IBM and Fortran

The first successful high-level programming language, Fortran, was developed from 1954 to 1956 on the IBM 704. The language was intended to be relatively independent of the underlying hardware, but when a compiler was first released in late 1956, the language contained several traces of the 704, including at least these:

- Variable names have at most six characters, because that many can fit in a 36-bit word with the 6-bit BCD character set.

- Floating-point output uses two-digit exponents, because the supported number range is only about $[10^{-39}, 10^{+38}]$. As we note in **Section 26.12.4** on page 873, that shortsighted decision about exponent width still affects software a half-century later.

- Integers are limited to the range $[-32768, 32767]$, because address arithmetic instructions are used for fixed-point arithmetic.

- Statement labels are limited to the range $[1, 32767]$, because their numeric, rather than string, values are used by the implementation.

- Arrays are limited to three subscripts, because the smallest 704 machines have only 4096 words of memory, and the maximum is 32,768 words.

- Built-in function names end in the letter `F`: `ABSF()`, `INTF()`, `MODF()`, and so on. Arrays are not permitted to have names ending in `F`, because that special letter is how the compiler distinguishes between function calls and array references without complex lookahead.

- `GO TO` statements transfer control to labeled statements, just as branch, jump, and transfer instructions do in hardware. Unfortunately, that soon produces impenetrable control flow, because *any* labeled statement can be the target of a jump from somewhere else in the same routine. That mess led to Dijkstra's famous letter, *Go To Statement Considered Harmful* [Dij68, Mis10], that spawned a vigorous debate in the programming-language community. Languages invented since that letter have better ways of managing control flow, and some even eliminate the `GO TO` statement.

- The three-way branch `IF (expr)` n_1, n_2, n_3 statement tests an expression for negative nonzero, zero, and positive nonzero, transferring control to three corresponding statement labels. It is implemented with three consecutive transfer instructions, `TMI`, `TZE`, and `TPL` (or `TRA`, for an unconditional jump).

- There are assigned and computed go-to statements for multiway branching:

```
ASSIGN label to v
GO TO v, (label1, label2, label3, ...)
k = expr
GO TO (label1, label2, label3, ...), k
```

  They are implemented by an indexed branch into a table of absolute branches, providing a two-instruction route to the desired code block.

- Statements match IBM 80-character punched cards, with a leading `C` for a comment, or else a 5-character label field, a continuation column, a 66-character statement field, and an 8-character card identification field. It was a good idea to use the latter for a card sequence number, so that a dropped deck could be put into the card sorter and restored to its correct order.

- There are several machine-specific or compiler-specific statements:

```
FREQUENCY n (i, j, k, ...)
IF (SENSE LIGHT i) n1, n2
IF (SENSE SWITCH i) n1, n2
IF (ACCUMULATOR OVERFLOW) n1, n2
IF (DIVIDE CHECK) n1, n2
IF (QUOTIENT OVERFLOW) n1, n2
PAUSE n
PRINT
PUNCH
READ
READ DRUM
READ INPUT TAPE
READ TAPE
SENSE LIGHT 3
STOP n
WRITE DRUM
WRITE OUTPUT TAPE
WRITE TAPE
```

  The `FREQUENCY` statement gives the compiler hints about how often statement label n is reached by branches. `READ` means the most common input device, the card reader, so it has a short name. `READ TAPE` means text input from tape, whereas `READ INPUT TAPE` means binary input.

- An `EQUIVALENCE` statement allows variables whose use does not overlap in time to share storage locations, conserving scarce memory.

- Arrays are declared with a `DIMENSION` statement, because there are no data type declarations yet. The `REAL` declaration came later, named to make it distinct from its a companion `INTEGER` declaration. Only after that was `DOUBLE PRECISION` added. There was insufficient foresight to have named the floating-point types `SINGLE` and `DOUBLE`. That mistake was later repeated in C with `float` and `double`.

Fortran is not the only language influenced by the IBM 704. The (CAR *list*) and (CDR *list*) functions in Lisp extract the first element of a list, and a list of the remaining elements. They are acronyms for ***C****ontents of* ***A****ddress* ***R****egister* and ***C****ontents of* ***D****ecrement* ***R****egister*, reflecting the fact that the 704 has instructions for processing fields of a word divided into a 3-bit *prefix*, a 15-bit *decrement*, a 3-bit *tag*, and a 15-bit *address*. Lisp uses the prefix and tag for data type identification.

H.6.3 IBM System/360

Until the early 1960s, computer models from most manufacturers were often unique designs, and at least a certain amount of code rewriting was needed to move software to another model in the same vendor series. Changing series or vendors required a complete rewrite. Each machine was strongly influenced by the technology of its time, and early manuals are replete with the details of punched-card devices, magnetic and paper tapes, magnetic drums, console switches and displays, the wiring of core memory and plug boards, and so on. Character encodings differed between vendors, and sometimes even between models of the same family, discouraging exchange of data. Until Fortran, COBOL, and Algol were designed and implemented, there was no possibility of exchange of software between unlike systems, because programs were written in symbolic assembly-language code, or worse, in numeric machine code.

IBM changed that situation forever when it decided to undertake the design of a family of computer systems that would guarantee upward compatibility, allowing customers to move up the IBM product line as their needs grew, without having to make a large investment in rewriting software. The family would also be the main IBM product line for years to come.

That family, announced in 1964, is the now-famous System/360, and it remains the longest surviving computer architecture by far. Its model of 8-bit bytes, byte addressing, and a 32-bit word influenced many subsequent architectures, and its addressing model has grown over four decades from a 24-bit space to a 64-bit space.

Even though IBM has since added support in that architecture for IEEE 754 binary [SK99] and decimal [IBM06] floating-point arithmetic, hexadecimal arithmetic, and all of the original CPU instructions, remain available. Its famous *Principles of Operation* manuals [POP64, POP04], which define the architecture without regard to particular implementations, are landmarks in the history of computing. Its original description [ABB64] is included in a 50th-anniversary collection of influential papers from IBM journals [ABB00].[4]

The lead author of that description, Gene Amdahl, left IBM in 1970 to found a company that developed and marketed the Amdahl 470 series of IBM-compatible mainframes. It became almost a fad in the computer industry to do so, with competing systems from Fujitsu, Hitachi, Itel, Magnuson, Nanodata, NEC, RCA, Ryad, Siemens, and Xerox. Wang produced systems similar to the System/360, but ran a different operating system.

The ultimate success of System/360 was not clear at the beginning, and the complexity of the project almost led to failure. The design and management lessons learned are the subject of a classic book, *The Mythical Man Month* [Bro82, Bro95], written by Fred Brooks, the second of the three authors of that first description. It should be read along with Tony Hoare's 1981 ACM Turing Award lecture, *The Emperor's Old Clothes* [Hoa81], about a large software project that was less fortunate.

The hexadecimal-base systems listed in **Table H.1** on page 948 all use the formats first introduced in the System/360. Data General, Denelcor, Gould, and Interdata adopted the System/360 hexadecimal floating-point architecture for their minicomputers, to ease data and software exchange with their customers' mainframes. Those systems all use *big-endian addressing* [Coh81]: the address of a value is that of its *high-order* storage byte. They also use sign-magnitude floating-point arithmetic and two's-complement integer arithmetic (see **Appendix I.2.1** on page 971 and **Appendix I.2.3** on page 972).

The Texas Instruments ASC (Advanced Scientific Computer) adopts the System/360 hexadecimal floating-point format, but reserves the largest exponent for Infinity (nonzero fraction) and Indefinite (zero fraction), concepts borrowed from the larger CDC systems. The ASC also has the concept of a *dirty zero*, a value with a zero fraction and nonzero exponent. When such a value is used in an arithmetic operation, the result is an Indefinite.

The original design choice of hexadecimal floating-point arithmetic was based on a study [Swe65] that analyzed the floating-point operands of a million addition instructions in the binary arithmetic of the 36-bit IBM 704 to find out how often shifting, normalization, underflow, and overflow occurred. For bases larger than 2, there is no possibility of having a hidden bit to increase the precision, but a larger base reduces the frequency of both shifts and normalization. Also, for a fixed number of exponent bits, it extends the representable range, making underflow and

[4] See http://www.research.ibm.com/journal/50th/.

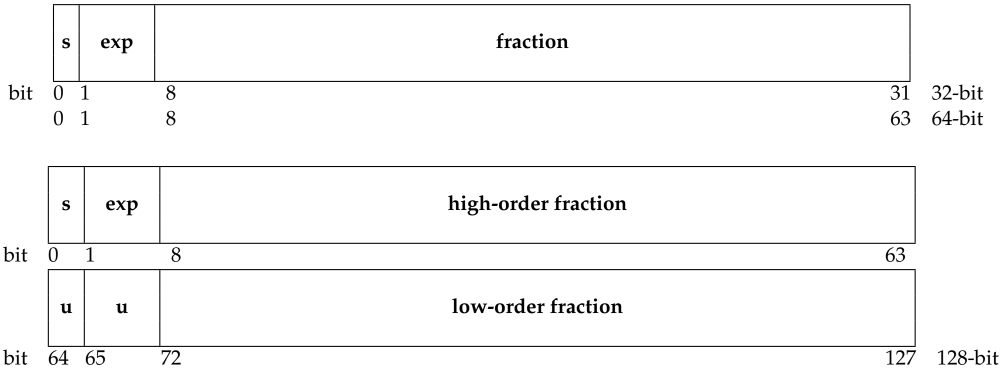


Figure H.5: System/360 hexadecimal floating-point data layout.
The sign bit is 1 for negative, and 0 for positive.
The stored unsigned exponent-of-16 has the same size (7 bits) in all formats. It is biased by 64, so the value is $(-1)^s \times (\text{fraction}) \times 16^{\text{exp}-16}$.
The sign and exponent are contained entirely in the leading byte, and the fraction has 6, 14, or 28 hexadecimal digits.
The sign and exponent fields in the high-order byte of the second part, marked as **u**, for unspecified, are ignored.
The fraction in the second part is treated as an extension of the upper fraction.
The range of normalized nonzero values with h hexadecimal digits in the fraction is $[\frac{1}{16} \times 16^{-64}, (1 - 16^{-h}) \times 16^{63}]$.
In C99 notation, that is [`0x0.1p-256`, `0x0.ff...ffp+252`], or about [`5.40e-79`, `7.23e+75`].
When the sign, exponent, and fraction are all zero, the value is called a *true zero*. Negative zeros (zero exponent and fraction) are possible, and can be generated by hardware.

overflow less likely. By default, those conditions are caught at run time and repaired by software interrupt handlers on System/360, so they are relatively costly. However, it is possible to clear a flag bit in the *Program Status Word* (PSW) to cause underflows to flush silently to zero in hardware. Choosing $\beta = 16$ gave higher speed for larger models of the System/360, and lower cost for small ones.

Unfortunately, for a base of the form $\beta = 2^K$, the normalized significand has no leading zero bits when it lies in $[1/2, 1)$, but has $K - 1$ leading zero bits in the range $[1/\beta, 1/2)$. That phenomenon, known as *wobbling precision*, is one that we deal with many times in this book. For hexadecimal arithmetic, the worst case represents a loss of three bits, or almost one decimal digit. The System/360 architects recognized that problem, and mentioned it as part of the motivation for offering a 64-bit floating-point format, when, at the time, most competing machines provided only 36-bit or 48-bit arithmetic.

Figure H.5 shows the System/360 floating-point layout.

Although the original design published in 1964 specified only 32-bit and 64-bit floating-point arithmetic, IBM added a 128-bit extension in 1967. All three formats share the same exponent range, which later numerical experience and analysis showed to be an unwise choice: increased precision should always have increased exponent range [Tur87, JP89].

Operands of any floating-point instruction may be normalized or unnormalized. Addition, subtraction, multiplication, and division produce normalized results, but there are also separate instructions for unnormalized addition and subtraction.

Addition and subtraction use a single additional hexadecimal guard digit, but on normalization, that extra digit is truncated without further effect on the remaining digits. Multiplication and division truncate their results. Thus, there is no provision for rounding other than the default *round-toward-zero*. Two decades later, the *High Accuracy Arithmetic* option for System/370 added support for IEEE-754-like rounding mode control in hexadecimal arithmetic [IBM84].

The 1964 edition of IBM's *Principles of Operation* manual [POP64] specified one hexadecimal guard digit for 32-bit arithmetic, but none for the 64-bit case. Regrettably, System/360 models produced from 1964 to 1967 lacked a guard digit, leading to the kinds of anomalies discussed earlier for the CDC and Cray systems. Under pressure from customers, a hexadecimal guard digit was added. The 1967 edition [POP67] noted that both 32-bit and 64-bit formats *may* have one hexadecimal guard digit. More recent editions of the architecture manual [POP75, POP04] specify a guard digit for 32-bit, 64-bit, and 128-bit subtraction and multiplication.

Here is an example of the misbehavior of arithmetic when there is no guard digit, from a small C program run on the Interdata 8/32, with annotation appended to mark significant output:

```
# cc guard.c && ./a.out
t = 16**(-5)           = 9.5367432e-07 = 0x3C100000
x = 16**(-6)           = 5.9604645e-08 = 0x3B100000
u = 1 + 16**(-5)       = 1.0000010e+00 = 0x41100001
v = 1 + 16**(-6) * 2   = 1.0000000e+00 = 0x41100000
w = 1 + 16**(-6)       = 1.0000000e+00 = 0x41100000
y = 1 - 16**(-6)       = 1.0000000e+00 = 0x41100000 <--- WRONG
z = 1 - (1 - 16**(-6)) = 0.0000000e+00 = 0x00000000 <--- WRONG

Patching y to expected value (number just below 1.0)
y = 1 - 16**(-6)       = 9.9999990e-01 = 0x40FFFFFF <--- OKAY!
z = 1 - (1 - 16**(-6)) = 9.5367432e-07 = 0x3C100000 <--- WRONG AGAIN!
```

The computation of $1 - 16^{-6}$ fails to produce the nearest representable number below 1.0, because of the missing guard digit. That number is then constructed manually by assignment of an integer value to the storage location of y, but the subtraction from y again fails to recover the expected value of 16^{-6}, getting 16^{-5} instead, a result that is wrong by a factor of β.

H.7 Lawrence Livermore S-1 Mark IIA

The last computer design that we consider in this appendix is the Lawrence Livermore National Laboratory S-1 Mark IIA, developed from about 1978 to 1984. The S-1 is a 36-bit vector machine, capable of emulating both the DEC PDP-10 and the Univac 1100, in addition to having its own instruction set. The design goal was about ten times the performance of the Cray 1, with an address space enlarged to 2GB. Although most commercial computers are designed privately by a small team, the S-1 is the result of extensive public scrutiny and comment.

The floating-point architecture is unusual, with ideas and features inherited from its 36-bit DEC and Univac ancestors, plus the large CDC and Cray systems, and also from IEEE 754 arithmetic, which was being designed about the same time. Here is a summary of its extraordinarily rich arithmetic:

- 18-bit, 36-bit, and 72-bit integers;
- normalized base-2 floating-point numbers;
- one's complement biased exponent;
- two's complement significand with a hidden bit in each size;
- 18-bit halfword format with 1-bit sign, 5-bit excess-16 exponent, and 13-bit significand (intended for signal-processing applications);
- 36-bit fullword format with 1-bit sign, 9-bit excess-256 exponent, and 27-bit significand;
- 72-bit doubleword format with 1-bit sign, 15-bit excess-16384 exponent, and 57-bit significand;
- Infinity (called *OVF*, for overflow);
- underflow produces the smallest nonzero magnitude, called *UNF*, instead of zero;
- negative zero is called NaN, and is an illegal operand in all floating-point instructions;
- rounding is controlled by a 5-bit field: of the 32 possible patterns, several are reserved or have no effect, but at least seven modes are possible, including *to* $-\infty$, *toward zero*, *to* $+\infty$, *away from zero*, *nearest*, *nearest with ties to* $+\infty$, and *PDP-10-style*;

- exceptions set sticky flags to record underflow, overflow, and NaN, and traps may be enabled or disabled for any of them;
- hardware instructions for the four primary operations in complex arithmetic, plus complex conjugate and complex magnitude, for both integer and floating-point operands;
- hardware instructions for square root, logarithm, exponential, sine, cosine, sine and cosine pair, arc tangent, and 2-D and 3-D Euclidean distance;
- hardware instructions for dot product, convolution, recursive filter, in-place matrix transpose, matrix multiply, normal and inverse Fast Fourier Transform (FFT), bit reversal, and quicksort partitioning.

The S-1 also has byte-field instructions for all sizes from 1 to 36 bits, and uses 9-bit characters. The PDP-10 has similar instructions, but normally packs five 7-bit characters with a final unused bit in a word, or else four 8-bit characters followed by four unused bits. The Univac 1100 series stores six 6-bit characters in a word, or with its quarterword instructions, four 9-bit characters.

H.8 Unusual floating-point systems

We discussed the arithmetic of some of the most important historical computer architectures in the preceding sections to give the reader an understanding of why they are significant, why their floating-point arithmetic designs all have limitations, and why the industry was led to a collaborative effort in the late 1970s to define the *IEEE 754 Standard for Binary Floating-Point Arithmetic* [IEEE85a], and later, its extension for radix-independent arithmetic [ANS87]. After a decade-long effort, it was further revised in 2008 to incorporate both binary and decimal arithmetic, and a fused multiply-add operation [IEEE08, ISO11].

Every computer, operating system, and computer user today enjoys, and suffers from, the legacy of those systems, so we also described some of their contributions to the industry in the interests of preserving a bit of industrial history that has radically changed human society.

Researchers have published many alternative floating-point proposals to try to overcome the problems of precision, range, significance, and exceptional conditions. The volume *Computer Arithmetic II* [Swa90b, Chapter 7] collects a half dozen important papers on that subject, and more can be found in the bibliography of floating-point arithmetic.[5] Here are some of the ideas that have been proposed:

- representation with unusual number bases, such as $i = \sqrt{-1}$ (*imaginary*), -2 (*negabinary*), and 3 (*ternary*);
- redundant number representations, such as $\{-1, 0, +1\}$;
- exponential and logarithmic representation, making multiply and divide easy, addition and subtraction hard, and greatly extending the number range to make underflow and overflow unlikely;
- interval arithmetic, which keeps strict lower and upper bounds on each operation, to provide error bars on final results;
- significance arithmetic, which uses unnormalized arithmetic to keep a record of trustworthy digits [AM59] (the Mathematica symbolic-algebra system does that with its variable-precision binary floating-point arithmetic [Jac92]);
- rational number representation, with arbitrary denominators, instead of just powers of the floating-point base;
- representation with extended, but fixed, precision (the C language tried to do that by offering only a `double` data type);
- representation with extended, and variable, precision (symbolic-algebra systems and a few scripting languages do that);
- representation with a movable boundary between exponent and significand, sacrificing precision only when a larger range is needed (sometimes called *tapered arithmetic*);

[5] See `http://www.math.utah.edu/pub/tex/bib/index-table-f.html#fparith`.

- residue number systems that represent values as sums of remainders with respect to division by prime numbers, eliminating carry from addition, making addition and multiplication easy, but division and comparison hard.

However, apart from a strong research interest in residue number systems for applications to signal processing, almost none of them has attracted sufficient commercial interest for vendors to make the large investments needed to implement them in hardware, software, and programming languages, and teach users how to employ them effectively. Also, several of them, by introducing variable numbers of digits, bring back the problem of wobbling precision, making numerical analysis and software design harder. The view of this author, based on decades of experience with many languages and many different computer systems, is that only *two* features in that list have any serious chance of adoption: longer precision, and interval arithmetic.

Longer precision is relatively easy, because it needs only a handful of new arithmetic instructions, and suitable library support. The mathcw library is designed from the start to support octuple-precision arithmetic, or about 70 decimal digits. We have shown repeatedly in this book that many computational problems can be solved much more easily if even a few more digits are available internally. The father of IEEE 754 arithmetic, William Kahan, has observed in his writings that it is unreasonable to expect that ordinary computer users will use clever numerical analysis to arrive at reliable solutions of their numerical problems. High precision, and particularly, the ability to easily rerun a computation in even higher precision, make many of the numerical difficulties disappear.

Support for interval arithmetic is one of the important reasons that IEEE 754 has extra rounding modes, although it has taken a long time to get interval arithmetic available outside its small research community. In November 2005, Sun Microsystems released free compilers that provide interval arithmetic for Fortran 90 and C++ on GNU/LINUX and SOLARIS operating systems on AMD64, IA-32, and SPARC hardware. IBM tried to do that on its mainframes with the ACRITH package [Lan87], but it is an optional, and extra cost, feature, and not well integrated into common programming languages, making programmers reluctant to adopt it. Similar problems exist for the *High Accuracy Arithmetic* option. Merely providing accurate arithmetic for parts of a computation may not be enough: the article *Anomalies in the IBM ACRITH Package* [KL85] gives numerous examples where continued access to the high-precision accumulator is needed, but ACRITH hides it inside black-box routines.

The subject of interval arithmetic is taught in only a few institutions, but it certainly deserves more attention. Researchers in engineering, the hard sciences, and medicine are expected to attach error bars to their reported physical data, to allow the reliability of that data to be assessed by others. That has practically *never* been the case with numerical results from computers, but it *should* be. One prominent numerical analyst has even called for use of interval, rather than point, arithmetic to be mandatory for certain government projects [Gus98]. Sadly, it may take government force and journal editorial requirements, rather than self-interest, for computer manufacturers and computer users to make interval arithmetic routine. Because rounding occurs only after an extended result has been generated, it should be possible to create interval instructions in hardware that produce two results simultaneously, instead of one, with little performance penalty over ordinary arithmetic. The IEEE 1788 subcommittee was formed in late 2008 to begin work on a proposed standard for interval arithmetic, but the task has proven to be difficult. They produced an official document in 2015 [IEE15], but further standardization work, and research in interval arithmetic, remain active.

H.9 Historical retrospective

The references cited in this appendix provide much additional material for readers who are interested in a better understanding of computer history. From our survey of important historical systems, we conclude that the key features to be desired of any computer-arithmetic system are at least these:

- The design must adhere to as many mathematical properties of arithmetic (of unlimited range and precision) as possible, because it is then more easily analyzed, and importantly for humans, more predictable.

- Significantly more precision and range are required than most people think, and designers of floating-point systems should not aim to support just the needs of the average user, or their currently biggest customer.

- Although many computer vendors and models have had short market lives, the IBM System/360 design reached the half-century mark after this book was largely completed. Architects are therefore advised to design for the long term, and to allow extensibility for future needs.

- Programming languages need to make access to higher precision easy, and preferably, without source-code changes. A few compilers provide an option, sometimes called `-autodbl`, that requests that floating-point constants and functions be automatically promoted to the next higher precision.

- Guard digits and correct rounding matter. The default rounding mode should always be the most unbiased, as is the case with *round to nearest with ties to even* in IEEE 754 arithmetic. However, other modes are required as well, and they must be standardly and universally available in all programming languages and on all platforms, run-time selectable, and efficient to switch among.

- The number base matters. For several reasons, use of decimal floating-point is likely to increase sharply, and may ultimately drive out binary and hexadecimal arithmetic. For more on that subject, see **Appendix D** on page 927.

- Boundary properties and undefined operations matter. Underflow and overflow need careful handling, and their incidence can be decreased by extended range and precision. IEEE 754 NaNs and Infinity seem to be a reasonable compromise in dealing with those boundaries, and also with mathematically undefined operations, like $0/0$ and $\infty - \infty$.

  The Rexx [Cow85, Cow90, REXX96] and NetRexx [Cow97, Cow00] scripting languages offer decimal arithmetic with up to 10^9 decimal digits, and exponents-of-10 with as many as 9 digits. That range is so large that those languages treat underflow and overflow as fatal errors. That is reasonable for most users, but some advanced users, and the designers of mathematical libraries for those languages, need to be cautious near those limits.

- Fast implementations of the fused multiply-add operation and efficient and exact dot products and sums can contribute significantly to the design of high-quality, reliable, and robust numerical software.

The marketing of scientific computers seems to be driven largely by instruction speed (megaflops, gigaflops, teraflops, petaflops, and perhaps even exaflops), and processor counts. Regrettably, too many customers appear to want to get a possibly incorrect answer fast, rather than to get a correct answer more slowly.

I Integer arithmetic

> For most processors, integer arithmetic is faster than floating-point arithmetic. This can be reversed in special cases such as digital signal processors.
>
> — Tutorial at `osdata.com`.

> Flight simulators are particularly vulnerable to arithmetic problems. The dynamic range of the many variables makes integer arithmetic inappropriate, yet the time costs of floating-point calculations makes them prohibitive.
>
> — Tutorial at `erasmatazz.com`.

The binary number system was extensively studied by Gottfried Leibniz who published the book *Explication de l'Arithmétique Binaire* (Explanation of Binary Arithmetic) in 1703, although there is evidence of binary arithmetic from India about 800 BCE. Leibniz may have been the first to use the binary digits 0 and 1.

At the hardware level, computers handle integers represented as strings of binary digits, called *bits*.[1] The bit string of architecture-dependent standard size is called the computer *word*. By the 1990s, almost all laptop, desktop, and larger computers used processors with a word size of 32 bits or 64 bits. Historically, however, many other word sizes have been used, as illustrated in **Table I.1** on the next page. The wide variety of sizes is surprising in comparison to modern systems.

Some architectures also support integer arithmetic on small submultiples or multiples of the word size, called *quarterwords*, *halfwords*, *doublewords* or *longwords*, and *quadwords*.

Two particular string sizes have their own special names: a 4-bit *nybble* (sometimes spelled *nibble*), and an 8-bit *byte*[2] (called an *octet* in some countries). A few historical machines have variable byte sizes. On the DEC PDP-10, a byte can be from 1 to 36 bits, and the most common byte size is 7, allowing five ASCII characters in a word.

The 60-bit CDC and 64-bit Cray systems are primarily floating-point machines. On the CDC systems, integer multiplication and division are handled by converting to floating-point values, reducing the effective word size to 48 bits, even though 60-bit integer values can be added and subtracted. There are also 18-bit integers, but they are normally used for memory addressing. The early Cray systems support both 24-bit and 64-bit integers; the Cray 2 extends the smaller size to 32 bits, reflecting its much larger address space.

Although older architectures address memory by words, the IBM System/360 family, introduced in 1964, addresses memory by 8-bit bytes, and most architectures designed since the late 1970s continue that practice. The IBM 7030 Stretch, planned to be the first supercomputer, is unusual in using bit addressing, and allowing integer word sizes from 1 to 48 (see **Section H.6.1** on page 959). The Burroughs B1700 also uses bit addressing, but its integers are always 24 bits.

[1] The convenient contraction *bit* was invented by famed Bell Labs scientist John W. Tukey about 1946 or 1947, and first used in a published paper in 1948 by his influential colleague, Claude E. Shannon [Sha48a, Sha48b]. It fortunately has replaced the alternatives *bigit* and *binit*.

In 1965, IBM research scientist James W. Cooley and Tukey discovered the *Fast Fourier Transform* (FFT) [CT65] which reduced an important $\mathcal{O}(n^2)$ computation to $\mathcal{O}(n \log n)$, revolutionizing many areas of engineering and science. It was later found that Gauss had discovered the FFT around 1805, but wrote about it only in an article in Latin published posthumously in 1866. The FFT was rediscovered by Runge in 1906, and again by Danielson and Lanczos in the field of crystallography in 1942 papers [DL42a, DL42b] that went unnoticed because World War II had most scientists busy elsewhere. Its computational significance only became evident when computers made large values of n possible, and Cooley and Tukey justly deserve the credit. The FFT has been ranked as one of the top ten algorithms of the Twentieth Century [Roc00].

Tukey also introduced the word *software* to computing in a 1959 article [Tuk58].

Shannon is regarded as the father of information theory (see earlier citations), and also made important contributions to computer science, cryptography, and mathematics [Sha45, GBC$^+$02]. The *BibNet Project* archives include bibliographies of his works, and those of Tukey.

[2] The word *byte* is coined from *bite*, but respelled to avoid confusion with *bit*. The word may have first appeared in print in a 1959 article by IBM researchers [BBB59]. It seems to be due to Werner Buchholz in an IBM memo of July 1956: see *Timeline of the IBM Stretch/Harvest era (1956–1961)*, `http://archive.computerhistory.org/resources/text/IBM/Stretch/102636400.txt`.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2

Table I.1: Integer word sizes of some historical and current computers. The model listing is representative, but not exhaustive. The ABC, operational in 1939, is credited with being the world's first operational electronic digital computer. The Z1, from 1936, predates the ABC, and had floating-point arithmetic, but is an electromechanical device. The IBM 704 is the first production computer with floating-point arithmetic (1955). The BRLESC (1962) is unusual in that it provides fixed-point binary arithmetic with 4 integer bits and 60 fractional bits; the remaining bits are used for tags, sign, and parity. The Intel 4004 is the first commercially available microprocessor (1971).

Bits	Model
1	IBM 7030 Stretch
4	Intel 4004
8	Intel 8008
9	Calcomp 900
12	CDC 160A and 6000 PPU, DEC PDP-5 and PDP-8, DSI 1000, Honeywell 1400, Univac 1
13	CDC 160G
16	Data General Nova and Eclipse, DEC PDP-11, IBM 1130 and NORC, Intel 8086, Norsk Data NORD-1, Zuse Z2
18	Cambridge EDSAC, DEC PDP-1, PDP-4, PDP-9, and PDP-15, Univac 418, Zuse Z25
20	Elliott 502, Ferranti Mark 1, General Electric GE 235
21	Advanced Scientific ASI 2100, CITAC 210B
22	Raytheon 250, Zuse Z1 and Z3
24	Burroughs B1700, CDC 924 and 3200, Datasaab D21, DEC PDP-2, Ferranti–Packard FP-6000, General Electric GE 435, Harris /6 and /7, Harvard University Mark I, Raytheon 520, Scientific Data SDS 940, SEL Systems 840A, Zuse Z26
25	Digital Electronics DIGIAC 3800, Univac III
29	CDC G-15
30	Electrologica EL X1 through EL X8, Univac 490
31	Librascope LGP-30
32	CDC G-20 and LGP-21, DEC VAX, Gould 9080, Hewlett–Packard PA-RISC 1.0, IBM System/360, POWER, and PowerPC, Intel iAPX 432 and IA-32, Interdata 7/32 and 8/32, LMI Lambda, MIPS (all Rn000 CPU models), Motorola 68000 and 88000, National Semiconductor 32016, Norsk Data NORD-5, SEL Systems System 85/86, Sun Microsystems SPARC, Xerox Sigma
33	El-tronics ALWAC III-E, Matsushita MADIC IIA, Mitsubishi Melcom 1101F, STC ZEBRA
35	Lyons LEO I
36	DEC PDP-3, PDP-6, and PDP-10, General Electric GE 635, Honeywell 600 and 6000, IBM 704, 7040, and 7090, Matsushita MADIC III, Symbolics 3600, Univac 1100
38	Zuse Z22
39	Elliott 503 and 803
40	Autometrics RECOMP II, Hitachi HITAC 3030, Los Alamos Scientific Laboratories MANIAC, Princeton University IAS, RAND JOHNNIAC, Regnecentralen DASK and GIER, University of Illinois Illiac I and ORDVAC, Zuse Z23
42	English Electric LEO-Marconi Leo 3, OKI Electric OKITAC 5090H
44	AEI 1010, University of Pennsylvania EDVAC
48	Burroughs B5000, Bull Gamma 60, CDC 1604 and 3600, English Electric KDF9, IBM 7030 Stretch, Hitachi HIPAC 103, National Physical Laboratory Pilot ACE (Automated Computing Engine), Telefunken TR440, University of Manchester Atlas
50	ABC (Atanasoff-Berry Computer)
52	University of Illinois Illiac II
54	Rice Institute R1
56	Philco 2000/213, RCA 601
60	CDC 6000 and 7000
64	Cray 1, 2, X-MP, and Y-MP, DEC Alpha, ELXSI 6400, Hewlett–Packard PA-RISC 2.0, IBM 7030 Stretch, Intel Itanium-1 and Itanium-2
72	Ballistics Research Laboratory BRLESC

Table I.2: Sign-magnitude 4-bit integers.

Bits	Value	Bits	Value
0000	0	1000	-0
0001	1	1001	-1
0010	2	1010	-2
0011	3	1011	-3
0100	4	1100	-4
0101	5	1101	-5
0110	6	1110	-6
0111	7	1111	-7

A p-bit computer word is capable of representing 2^p different bit patterns. Thus, with $p = 3$, we have these possibilities: 000_2, 001_2, 010_2, 011_2, 100_2, 101_2, 110_2, and 111_2. They represent the *unsigned* integral values 0 through 7; for example, $5_{10} = 101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$.

I.1 Memory addressing and integers

There is only one mistake that can be made in computer design that is difficult to recover from — not having enough address bits for memory addressing and memory management. The PDP-11 followed the unbroken tradition of nearly every computer.

— C. G. Bell and W. D. Strecker (1976).

A partial list of successful machines that eventually starved to death for lack of address bits includes the PDP-8, PDP-10, PDP-11, Intel 8080, Intel 8086, Intel 80186, Intel 80286, AMI 6502, Zilog Z80, CRAY-1, and CRAY X-MP.

— David A. Patterson and John L. Hennessy (1990).

Memory addresses are almost always treated as unsigned integers, but they are sometimes limited to fewer bits than the integer word size, for reasons of hardware economy. Arithmetic on addresses is often restricted to just addition and subtraction. Some machines even have separate register sets for address arithmetic.

I.2 Representations of signed integers

For most applications, negative integers are also needed, and there are four important conventions for the representation of negative numbers in binary arithmetic. Those encodings are described in the following subsections, and in each of them, positive numbers are represented identically, and the rightmost bit is the least significant.

I.2.1 Sign-magnitude representation

As the name suggests, *sign-magnitude* representation uses a sign bit together with a magnitude, that is, an unsigned value. Typically, the leftmost bit, s, is the sign bit, 0 for positive, and 1 for negative, so the sign can be written $(-1)^s$. Table I.2 shows how that works in a 4-bit system.

There are two representations of zero: -0 and $+0$. The computer hardware must be designed to treat them as equal, which complicates the implementation.

The number range of p-bit sign-magnitude arithmetic is $-(2^{p-1} - 1) \ldots -0 +0 \ldots (2^{p-1} - 1)$. The negative value $-k$ in sign-magnitude form has the same bit pattern as the *unsigned* value $2^{p-1} + k$.

Sign-magnitude arithmetic has never been widely adopted for binary arithmetic, but it is used in the IBM 704 and 7090 mainframes introduced in 1955 and 1958, respectively, as well as in the IBM 7030 Stretch from 1961. It is also used in most systems with decimal arithmetic.

Table I.3: One's-complement 4-bit integers.

Bits	Value	Bits	Value
0000	0	1000	−7
0001	1	1001	−6
0010	2	1010	−5
0011	3	1011	−4
0100	4	1100	−3
0101	5	1101	−2
0110	6	1110	−1
0111	7	1111	−0

I.2.2 One's-complement representation

In *one's-complement* arithmetic, negative integers are represented by inverting (complementing) all bits of the corresponding positive number. Table I.3 illustrates that system for 4-bit words.

As in the sign-magnitude representation, there are two zeros, $+0$ and -0, and the number range is also the same in the two systems: $-(2^{p-1}-1)\ldots -0 \; +0 \ldots (2^{p-1}-1)$.

The bit pattern for a one's-complement negative number $-k$ is the same as that for the *unsigned* value $2^p - 1 - k$.

Subtraction is easy in that system: simply add the one's-complement of the second operand to the first operand, as in this example in our 4-bit system:

$$\begin{aligned} 2 - 6 &= 2 + (-6) \\ &= 0010_2 + 1001_2 \\ &= 1011_2 \\ &= -4. \end{aligned}$$

One's-complement arithmetic is uncommon today, but it is used on CDC and Univac mainframes built in the 1960s and 1970s. When UNIX, and C, were ported to the 36-bit Univac 1100 series about 1984, the researchers remarked [BHK+84, page 1778]:

> In actual practice, problems caused by one's complement arithmetic are rare. Some of the nastiest ones are in the C compiler itself!

As a historical note, the *Pascaline*, an early mechanical calculator developed by Blaise Pascal about 1645, could only do addition, but by expressing the second operand in *nine's complement*, subtraction is also possible, albeit laborious, because the complement operation (subtracting from 999 999 with no need to borrow) must be done without the aid of the machine:

$$\begin{aligned} 987\,654 - 123\,456 &= 987\,654 + (999\,999 - 123\,456) - 999\,999 \\ &= (987\,654 + 876\,543) - 999\,999 \\ &= 1\,864\,197 - 999\,999 \\ &= 1\,864\,197 - (1\,000\,000 - 1) \\ &= (1\,864\,197 - 1\,000\,000) + 1 \\ &= 864\,197 + 1 \\ &= 864\,198. \end{aligned}$$

Notice that the procedure can be shortened by dropping the leading (overflow) digit 1 in the third step, adding it back at the low end, and omitting the complement of subtracting 999 999.

I.2.3 Two's-complement representation

Most modern computer architectures use *two's-complement* representation, because certain of its properties simplify the hardware implementation of arithmetic. It is similar to one's-complement, except that after inverting all of the

Table I.4: Two's-complement 4-bit integers.

Bits	Value	Bits	Value
0000	0	1000	−8
0001	1	1001	−7
0010	2	1010	−6
0011	3	1011	−5
0100	4	1100	−4
0101	5	1101	−3
0110	6	1110	−2
0111	7	1111	−1

bits, you then must add one to the result. Overflow is only possible in the addition when the original value is zero, and must be ignored.

Although it would appear from our description that negation requires an addition operation, that is not so. Negation can be accomplished by a fast circuit that implements this simple rule:

> *Starting at the rightmost bit, copy bits up to and including the first 1-bit, then invert the remaining bits.*

Table I.4 shows the two's-complement integers in a 4-bit system.

From the bit patterns for negative numbers, the bit pattern for $-k$ is the same as the *unsigned* result $2^p - k$.

As in the one's-complement system, subtraction can be done by adding the (two's-) complement of the second operand:

$$\begin{aligned} 2 - 6 &= 2 + (-6) \\ &= 0010_2 + 1010_2 \\ &= 1100_2 \\ &= -4. \end{aligned}$$

The low-order bit is the parity of the number: the number is *even* if that bit is 0, and otherwise, it is *odd*. That property is also enjoyed by sign-magnitude arithmetic, but *not* by one's-complement arithmetic.

The important differences between two's-complement arithmetic and the other two systems are that (a) there is only one representation of zero, and (b) there is one more negative number than positive. That is, the number range is $-2^{p-1} \ldots -1\,0\,+1 \ldots (2^{p-1} - 1)$.

For high-level language programming, point (a) is unimportant, but point (b) matters. Taking the absolute value of the most negative number in two's-complement arithmetic generates an integer overflow condition, because the result requires $p + 1$ bits.

To see what happens when we negate -8 ($= 1000_2$) in 4-bit two's-complement arithmetic, invert its bits to get 0111_2, then add 1 to get 1000_2. That is the same as -8 again, instead of $+8$. That is a general result for any number of bits in two's-complement arithmetic: $-(-2^{p-1})$ overflows to -2^{p-1}.

The following simple Fortran program demonstrates that negation of the most negative number in 32-bit two's-complement arithmetic produces that number back again, rather than a positive number:

```
N = -2**30 - 2**30
PRINT *, (N + 1), -(N + 1)
PRINT *,  N, -N
END
```

The number N is the most-negative integer in the two's-complement system, so $-(N + 1)$ should be the largest positive integer. The program's output on a Sun Microsystems workstation is

```
-2147483647   2147483647
-2147483648  -2147483648
```

and the nonsense in the last number is a consequence of the overflow.

Table I.5: Excess-n 4-bit integers. Here, the bias is $2^{4-1} - 1 = 7$.

Bits	Value	Bits	Value
0000	−7	1000	1
0001	−6	1001	2
0010	−5	1010	3
0011	−4	1011	4
0100	−3	1100	5
0101	−2	1101	6
0110	−1	1110	7
0111	0	1111	8

When digit strings are converted to integer values, the software should always accumulate a *negative result*, and then invert the sign if the number should be positive. That way, the most-negative integer that is representable in two's-complement arithmetic can be input correctly. In addition, the code works properly for the signed zeros and outer limits of one's-complement and sign-magnitude systems. Of course, overflow in the string conversion is still possible, and must be suitably handled, such as by setting a status flag and clamping the result to the nearest representable number.

I.2.4 Excess-n representation

If a p-bit string represents an unsigned binary integer, then implicit subtraction of a constant bias allows positive and negative numbers. That is called *excess-n* representation. Choosing the bias to be $n = 2^{p-1} - 1$ produces an optimum split, as illustrated in Table I.5. With that choice of n, there is only a single zero, but there is one more positive number than negative numbers. As with the two's-complement system, that means that there is a value at one end of the number range that produces itself when negated.

The excess-n representation is commonly used for the exponent field in floating-point encodings, but is rarely seen elsewhere. The reason for using a biased exponent is that integer comparison instructions can then be used for ordering of floating-point values, as long as the values are not IEEE 754 NaNs.

I.2.5 Ranges of integers

The largest integer that can be represented in most programming languages is limited by the host word size. On many machines, 32-bit integers are often the longest available. The DEC VAX, some supercomputers, and RISC architectures of the 1990s, support 64-bit integers, although computer programming languages may require use of additional integer data types to access them.

Table I.6 on the facing page shows the largest signed and unsigned integer values for typical computer word sizes. In sign-magnitude and one's-complement systems, the corresponding most-negative values are the negatives of the largest signed numbers. In two's-complement systems, they are the negatives of one more than those numbers. It is a good idea to commit to memory approximate values for $p = 8$, 16, and 32 (the shaded rows in the table), so that when you program an application that needs large integers, you can quickly determine whether your computer word size is large enough for the task.

As a small programming note, observe that you cannot compute the largest 32-bit signed integer in Fortran by the expression `2**31 - 1`, because the term `2**31` overflows. Instead, you need to write it as `2**30 + (2**30 - 1)`, where the parentheses are *essential* to avoid overflow.

Signed integers in 32-bit words are inadequate for many purposes. They are insufficient to represent

- the number of people on Earth;
- the U.S. national debt;
- the annual revenues of many large corporations;
- the number of bytes in the disk systems of many computers;

Table I.6: Largest integers in various word sizes.

Word size	Largest signed integer	Largest unsigned integer
4	7	15
8	127	255
12	2 047	4 095
16	32 767	65 535
18	131 071	262 143
24	8 388 607	16 777 215
32	2 147 483 647	4 294 967 295
36	34 359 738 367	68 719 476 735
40	549 755 813 887	1 099 511 627 775
48	140 737 488 355 327	281 474 976 710 655
60	576 460 752 303 423 487	1 152 921 504 606 846 975
64	9 223 372 036 854 775 807	18 446 744 073 709 551 615

- the estimated age (in years) of the universe;
- the number of microseconds in a day, and the number of seconds in a century, both important for time keeping on computers.

Integers of 64 bits, and 64-bit addressing, are expected to be adequate for a long time. For example, if you could afford to buy 2^{64} bytes of memory, and your computer could address it all, then if you started to write to that memory at the substantial rate of 100 MB/sec (100 million bytes per second), it would take about 5850 years to fill it just once. Equivalently, a computer incrementing an initially zero counter every nanosecond would take 585 years to reach the largest representable 64-bit number.

I.3 Parity testing

In sign-magnitude and two's-complement systems, an integer is even if the low-order bit is zero, and otherwise, is odd. The C-language family programming idiom `n & 1` is a common way to test that bit: the expression is 0 for even, and 1 for odd.

However, in one's-complement arithmetic, that test is only correct for nonnegative integers; the result must be complemented for negative n.

For the bias chosen in Table I.5 on the preceding page for excess-n arithmetic, the expression is 1 for even, and 0 for odd.

I.4 Sign testing

Even though we do not have an explicit sign bit in one's-complement and two's-complement forms, we can always determine the sign of an integer in those systems by examining the leftmost bit. That bit is 1 if the number is negative, and 0 if the number is positive, just as it is for the sign-magnitude system. Low-level assembly-code programs, and the hardware itself, make use of that fact, but in high-level languages, comparisons of entire p-bit words against zero are normally used instead for sign tests.

Full-word comparisons may not be sufficient when there are two zeros, as in one's-complement and sign-magnitude arithmetic, because -0 and $+0$ compare equal: the sign bit must be examined to distinguish between them. A similar situation exists in IEEE 754 floating-point arithmetic, where the `copysign()` and `signbit()` functions are essential library primitives.

I.5 Arithmetic exceptions

Most computer systems generate an interrupt for a division by zero in integer arithmetic, and often, unless system-dependent corrective action is taken, the program is terminated.

However, programmers must be aware that integer *overflow*, that is, the generation of integer values that take more bits to represent than the host word can hold, usually goes *undetected*, and the result may be complete nonsense. Here is a small example in the Fortran language:

```
      N = 65535
      PRINT *, 'N   = ', N
      PRINT *, 'N*N = ', N*N

      N = 65536
      PRINT *, 'N   = ', N
      PRINT *, 'N*N = ', N*N

      N = 65537
      PRINT *, 'N   = ', N
      PRINT *, 'N*N = ', N*N

      END
```

The value $65\,536_{10} = 2^{16} = 1\,0000\,0000\,0000\,0000_2$, so its square, 2^{32}, requires 34 bits to represent (33 for the magnitude, and 1 for the sign). Thus, computing `N*N` causes an overflow on systems with 32-bit integer arithmetic. However, when that program is run on DECstation (MIPS), DEC 3000/400 (Alpha), IBM RS/6000 (POWER), IBM 3090, Silicon Graphics Indy (MIPS R4000), Silicon Graphics Indigo-2 Extreme (MIPS R4400), Stardent 1520 (MIPS), Sun Microsystems 3 (Motorola 68020), Sun Microsystems 386 (Intel 80386), and Sun Microsystems 4 (SPARC) systems, the program runs *without error* and prints these peculiar results:

```
  N   =     65535
  N*N =   -131071
  N   =     65536
  N*N =         0
  N   =     65537
  N*N =    131073
```

What has happened here? Squaring N produces these exact 32- and 33-bit results:

$$65\,535 \times 65\,535 = 4\,294\,836\,225 = 0\,1111\,1111\,1111\,1110\,0000\,0000\,0000\,0001_2,$$

$$65\,536 \times 65\,536 = 4\,294\,967\,296 = 1\,0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000_2,$$

$$65\,537 \times 65\,537 = 4\,295\,098\,369 = 1\,0000\,0000\,0000\,0010\,0000\,0000\,0000\,0001_2.$$

The arithmetic hardware then *truncates* the results to 32 bits, giving the values $-131\,071$ (negative because of overflow into the sign bit), 0, and $131\,073 = 10\,0000\,0000\,0000\,0001_2$.

Interestingly, all but one of those systems forced run-time program termination for the overflow condition in a separate program for the computation of $N/0$, giving messages like those shown in **Table I.7** on the facing page. However, the Stardent 1520 produced a nonsensical value of -1 and did *not* terminate prematurely. Repeating that test on all platforms currently available to this author shows that almost all processors terminate the computation, except for the IBM PowerPC, for which $N/0$ evaluates to 0.

For programs that simply use integers as indexes into arrays of modest size, integer overflow is rarely of concern. However, applications that involve random-number generation, and hash-function computation, can require the formation of large products where overflow can happen.

A decade-old integer overflow

bug in the Java `binarySearch()` function was fixed[3] in 2004 by replacing the index-averaging calculation

```
  int middle = (low + high) >> 1;
```

by the unsigned shift

```
  int middle = (low + high) >>> 1;
```

[3] See `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5045582`.

Table I.7: Behavior of integer division by zero.

System	**Message and action**
DECstation	Trace/BPT trap (core dumped)
DEC 3000/400	forrtl: error: floating point exception IOT trap (core dumped)
Silicon Graphics Indy and Indigo-2	Trace/BPT trap (core dumped)
Sun Microsystems 3, 386i, 4	*** IOT Trap = signal 6 code 0
IBM RS/6000	Breakpoint (core dumped)
IBM 3090	AFB209I VFNTH : Program interrupt - FIXED-POINT DIVIDE exception

The bug had gone unnoticed until memories got large enough to have an array size of 2^{30}, and the resulting overflow to a negative value later resulted in an *ArrayIndexOutOfBoundsException* error, thanks to Java's mandatory bounds checks for array indexing. The same bug exists in most textbook presentations of the binary-search algorithm. The bug can be prevented in another way that avoids intermediate overflow:

```
int middle = low + (high - low) >> 1;
```

The lesson of this section is that you *cannot rely on programming-language implementations to catch integer overflow*, and sometimes, even zero divides. Your program may run to completion, but generate ridiculous results. Programs that experience uncaught integer overflow and then produce nonsensical output have fouled up accounting software, missile guidance systems, a U.S. Navy ship carrying nuclear weapons, electronic voting-system tallies, and even the New York Stock Exchange.

I.6 Notations for binary numbers

The numerical examples in the preceding sections demonstrate that it is tedious to write large numbers in binary, so bases that are larger powers of two are often used. That simplifies conversions from binary, because bit groups of fixed size can be directly converted to digits in the larger base.

In base 8 $(= 2^3)$, we have the *octal* number system, in which digits $0 \ldots 7$ represent the binary values $000_2 \ldots 111_2$. Octal notation is uncommon on modern architectures, but was traditionally used on machines for which the word size is a multiple of 3, such as 12, 18, 24, 36, 48, or 60 bits, and curiously, also on the 16-bit DEC PDP-11 systems and the Cray 64-bit vector machines. In octal, $65\,535_{10} = 177\,777_8$ and $65\,537_{10} = 200\,001_8$.

In base 16 (= 2^4), we have the *hexadecimal* number system in which the digit values $0 \ldots 9$ `a b c d e f`, represent the binary values $0000_2 \ldots 1111_2$. Hexadecimal notation is commonly used on modern systems with word sizes of 32 or 64, both multiples of 4. In hexadecimal, $65\,535_{10} =$ `ffff`$_{16}$ and $65\,537_{10} = 10001_{16}$

The Burroughs B1700 system programming language, SDL, supported binary strings in bases 2, 4, 8, and 16: the base-4 *quartal* value `@(4)3210@` is $3 \times 4^3 + 2 \times 4^2 + 1 \times 4 + 0 = 228$ in decimal.

The Ada programming language has a convenient notation that extends easily to any number base: 65 535 = `10#65535#` = `8#177777#` = `16#ffff#`.

Fortran 77 does not provide a standard way of representing numbers in octal or hexadecimal form in program source code, or of using those forms for input and output. However, most compilers provide nonstandard extensions for doing so; regrettably, vendors have not agreed on a common format. Few compilers have provided for binary representations. The forms `Zfe01`, `Z'fe01'`, `X'fe01'`, and `'fe01'X` have been used for hexadecimal constants, and `"7741`, `7741B`, `O'7741'`, and `'7741'O` for octal constants. One vendor, the now-defunct ELXSI, also supported the Ada-style notation. Use of such extensions makes code distinctly nonportable.

Fortran 90 provides the forms `B'1001'` and `B"1001"` for binary, `O'7741'` and `O"7741"` for octal, and `Z'fe01'` and `Z"fe01"` for hexadecimal.

Ada allows underscores to separate digits to make numbers more readable: 8#177777777# = 8#177_777_777#. On the 36-bit DEC PDP-10, large octal integers are normally written in two 18-bit chunks separated by double commas, 123456,,765432, but that scheme was never generalized.

In Fortran, blanks are not significant except in character strings, so they may be used to improve readability of long numbers in program source code. The following pairs of assignments are equivalent:

```
N  = 2147483647
N  = 2 147 483 647
PI = 3.14159265358979323850
PI = 3.141 592 653 589 793 238 5
```

Regrettably, few programming languages provide a simple way to read and write numbers with embedded spaces or other separators, despite the fact that in mathematical tables, spacing between digit groups is conventional to enhance readability. The mathcw library extensions to the input and output facilities of the C-language family described in **Chapter 26** on page 829 and **Chapter 27** on page 879 support digit grouping with underscore separators.

I.7 Summary

The chief lessons of this appendix on integer arithmetic are these:

- Integer division by zero is not universally detected.
- Integer overflow is almost never detected. Indeed, the specifications or standards for some programming languages require that it not be. The Java language specification [GJSB05] says that *built-in integer operators do not indicate overflow or underflow in any way*. The same is true in C#, unless the expression is qualified by the checked keyword. The C99 Standard simply declares that behavior is undefined in the event of integer overflow.

  At the Web site *Fun with Fibonacci* [Bee04b], this author solved an introductory programming problem in about 50 languages. Only *one* language, Algol 60 on the DEC PDP-10, caught integer overflow on addition.
- Programmers must ensure that integer values required in a computation do not exceed their representable range, to avoid nonsensical results, or tragic failure.
- The common programming idiom of testing the low-order bit to determine whether an integer is even or odd fails for negative values in one's-complement arithmetic.
- In the widely used two's-complement system, the most negative integer has no positive companion, so the result of the integer absolute-value function can be negative.
- Compilers for most programming languages, and the C-language family, fail to handle the most negative two's-complement integer correctly in source code, and also often in program input. That error requires subterfuges like these definitions taken from the C header file <stdint.h>:

  ```
  #define INT16_MIN (-32767 - 1)
  #define INT32_MIN (-2147483647 - 1)
  #define INT64_MIN (-9223372036854775807LL - 1LL)
  ```

Programmers usually assume that integer arithmetic is trustworthy, but the limits of finite precision sometimes violate that assumption. **Section 4.10.1** on page 74 describes functions in the mathcw library for safe integer arithmetic.

If you are interested in learning more about the low-level details of how integer arithmetic works at the electronic circuit and algorithmic level, consult books on computer arithmetic, such as [Omo94, Par00, Kor02, EL04a, KM10, MBdD^{+}10, BZ11, DSC12, Kul13]. Collections of important original papers on computer arithmetic are available in [Swa90a, Swa90b]. The book *Hacker's Delight* [War03, War13] presents many clever tricks with integer arithmetic.

The Web site http://bitsavers.org holds a large collection of manuals and books for historical computer systems. For an outstanding account of such systems up to about 1980, see *Computer Architecture: Concepts and Evolution* [BB97], written by two leading architects of the IBM System/360. For a detailed survey of computer systems up to 1967, see *Digital Computer User's Handbook* [KK67, Chapter 1.4].

We have considered only binary arithmetic in this appendix, but because numbers can always be represented as polynomials in the number base, as $n = c_0 + c_1\beta + c_2\beta^2 + \dots$, choices other than $\beta = 2$ are possible. Ternary (base-3) systems, with digits 0, 1, and 2, balanced ternary systems with digits -1, 0, and $+1$, base-10 decimal systems, and even systems with negative, imaginary, or complex bases, have all been proposed, and sometimes even implemented in experimental prototype machines. However, except for decimal machines, none of those designs has ever been commercially significant.

J Java interface

Java, *n.*: A breed of domestic hen.

— *New Century Dictionary* (1914).

java, *n.*: Coffee from Java; also (slang) any sort of coffee.
(1926) We went back to the fire and discussed breakfast.
'Nothing but Java,' said the bum that had the coffee.

— *Oxford English Dictionary*.

JAVA: Stock exchange symbol of Sun Microsystems, Inc. (2007).

The Java programming language [GJS96, GJSB00, GJSB05, GJS$^+$13, GJS$^+$14] is defined in terms of an underlying virtual machine [LY97, LY99, SSB01, LYBB13, Set13, LYBB14]. Thus, once the virtual machine is available on a new platform, all existing precompiled Java code can run on it immediately. The Java promotional slogan, *Write once, run everywhere!*, reflects the wide interest that the language has received since it was first introduced in 1995, and more than 2300 books have been written about Java, far exceeding the book counts for any other programming language.[1]

The virtual-machine layer has a drawback, however: it makes it impossible to communicate with other languages running on the underlying hardware, unless the virtual machine is augmented with a mechanism for doing so. Fortunately, the Java developers recognized the need to access code in other languages, and provided the standard *Java Native Interface (JNI)* [GM98, Lia99] for that purpose.

The Java type-modifier keyword `native` on a function definition tells the compiler that code is supplied elsewhere; consequently, the normal braced code body is replaced by a semicolon. Examples are presented on page 981.

Compilation of a source-code file with the Java compiler, `javac`, produces a file of object code, called a *class file*, for the Java Virtual Machine. The `javah` utility extracts information from the class file, producing a header file containing prototypes for the wrapper functions. Those functions must then be written in C or C++ to interface to the native code.

The C interface code is compiled and linked into a shared object library that is loaded dynamically into the virtual machine at run time when the class is instantiated.

J.1 Building the Java interface

Before looking at code samples, it is helpful to see how the Java interface to the C code in the `mathcw` library is built.

Initially, we have just three hand-coded files in the `java` subdirectory:

```
% cd java
% ls
Makefile  MathCW.c  MathCW.java
```

The `Makefile` contains the rules for the build process, and the other two files contain the C interface code and the Java class definition. The mixed lettercase is conventional in Java class definitions, and the name `MathCW` intentionally resembles that of the standard `Math` class, whose features it can entirely replace and augment.

Java requires a 64-bit `long int` data type, so the `mathcw` library must be built with a compiler that supports a corresponding type in C. The required `long long int` type in C is standardly available in C99, and many recent compilers support it as well. However, its use has to be enabled for pre-C99 compilers that can handle that type: defining the symbol `HAVE_LONG_LONG_INT` does that job.

The `make` utility manages the build, which looks like this on a Sun Microsystems Solaris system:

[1] See `http://www.math.utah.edu/pub/tex/bib/index-table-j.html#java` for extensive bibliographies.

© Springer International Publishing AG 2017
N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2

```
% make

javac -classpath . MathCW.java

javah -jni MathCW

cc -g -DHAVE_LONG_LONG_INT -I.. -I/usr/java/include -I/usr/java/include/solaris -c -o MathCW.o MathCW.c

cc -G -g -DHAVE_LONG_LONG_INT -I.. -I/usr/java/include -I/usr/java/include/solaris -o libMathCW.so \
   MathCW.o -L/usr/local/lib -lmcw
```

The first step creates `MathCW.class`, and the second step makes `MathCW.h`. The third step compiles the C interface, producing the object file `MathCW.o`. The last compilation step creates the shared object library, `libMathCW.so`, for the interface, as directed by the `-G` option.

The final directory listing shows the four new files:

```
% ls
Makefile  MathCW.c  MathCW.class  MathCW.h  MathCW.java
MathCW.o  libMathCW.so
```

The interface library file and the class file should be installed in a system-wide directory for general use, but there are as yet no widely used conventions for the names of such directories. For this demonstration, we assume that the two files are copied into another directory where a small Java test program is available:

```
% ls
Makefile  MathCW.class  libMathCW.so  test.java
```

As usual, the `make` utility directs the process:

```
% make check-test

javac -classpath . test.java

java -Djava.library.path=. test
MathCW    erf(-2.0) = -0.9953222650189527
MathCW    erf(-1.875) = -0.9919900576701199
MathCW    erf(-1.75) = -0.9866716712191824
...
MathCW   erfc(1.75) = 0.013328328780817555
MathCW   erfc(1.875) = 0.008009942329880029
MathCW   erfc(2.0) = 0.004677734981047265
```

Compilation of Java source files requires knowledge of where all referenced class files are stored. The `-classpath` option provides a list of directories to search, in the event that a suitable list has not been defined by the `CLASSPATH` environment variable. We set it to the current directory, represented in UNIX by the dot.

However, because use of the Java Native Interface is relatively uncommon, most Java implementations do not recognize a companion environment variable for the library path, although all of them have a built-in default search path that may be discoverable only by running a system-call trace on the `java` program. If the shared object library is not, or cannot be, installed in one of those standard directories, then its location must be supplied at run time, as we did here by defining the `java.library.path` variable to be the current directory.

For Java programs that are used often, it is a nuisance to have to run them by supplying them as arguments to the virtual-machine program, `java`. The conventional solution in UNIX is to hide the complexity in a shell script, and then invoke that script by name. The `javald` utility that is provided in some Java distributions makes it easy to create such wrapper scripts.

J.2 Programming the Java MathCW class

The `MathCW.java` file contains the complete specification of the Java `MathCW` class. Because none of the functions is implemented in Java, the class file is comparatively small. We present it in parts, with just enough detail to show its general structure, and with large chunks of repetitive code omitted.

Like the standard `Math` class, the `MathCW` class does not permit subclassing or instantiation. The Java keyword `final` in its declaration disallows subclasses, and the keyword `private` in the class constructor function `MathCW()` prevents other classes from creating new instances of this class:

```
public final class MathCW
{
  private MathCW() {}
```

Like the standard `Math` class, the `MathCW` class provides names for two important mathematical constants:

```
  public static final double E  = 2.71828182845904523536;
  public static final double PI = 3.14159265358979323846;
```

In some languages with strict data typing, it is possible for different functions to have identical names, provided that their signatures differ. The function names are then said to be *overloaded*. The *signature* of a function is just a list of the number and types of its arguments. The compiler can then deduce which function version is needed by examining how it is called. In practice, the function names are compiled into distinct names that include cryptic summaries of the argument types. That *name mangling* is generally invisible to programmers, except inside debuggers, or when tools, such as the Unix `nm` or `strings` utilities, are used to display names embedded inside object files.

Java allows overloading of function names, and standard classes therefore use the same function names when signatures differ but the computation is similar. Because Java supports only two floating-point types, `float` and `double`, we need two sets of function definitions that differ only in argument types:

```
  public static native float acos (float x);
  public static native float acosh (float x);
  public static native float adx (float x, int n);
  ...
  public static native float nanf (String tag);
  ...
  public static native float tanh (float x);
  public static native float tgamma (float x);
  public static native float trunc (float x);

  public static native double acos (double x);
  public static native double acosh (double x);
  public static native double adx (double x, int n);
  ...
  public static native double nan (String tag);
  ...
  public static native double tanh (double x);
  public static native double tgamma (double x);
  public static native double trunc (double x);
```

Overloading is not possible for the functions that take a single `String` argument, because their signatures are identical. As a result, the class supplies `nanf()`, `qnanf()`, and `snanf()` with `float` return values, and `nan()`, `qnan()`, and `snan()` with `double` return values. For similar reasons, many of the routines in the random-number generator family, such as `urcwf()` and `urcw()`, are available under their own names.

Java does not support pointer types, so pointer arguments in C functions are replaced by one-element arrays. For example, the C prototype

```
extern float frexpf (float x, int * pn);
```

corresponds to this Java prototype:

```
public static native float frexp (float x, int pn[/* 1 */]);
```

The function can then be used in a Java code fragment like this:

```
float f, x;
int pn[] = new int[1];

x = 1234567.0F;
f = MathCW.frexp(x, pn);
System.out.println("MathCW.frexp(" + x + ", pn) = " + f + " * 2**(" + pn[0] + ")");
```

After years of stability of the Math class, Java 1.5 (2004) and 1.6 (2006) added several new functions:

```
cbrt() cosh() expm1() hypot() log10() log1p() scalb() sinh() tanh()

copySign() getExponent() nextAfter() nextUp() signum() ulp()
```

The functions in the first group have counterparts in the C99 and mathcw libraries. Those in the second group are readily implemented in Java, sometimes with the help of other functions in the mathcw interface. All of the new functions are fully supported in the MathCW class, making it a proper superset of the standard Math class.

The final part of the class definition is a static initializer that loads the shared library when the class is first instantiated at run time:

```
    static
    {
        System.loadLibrary("MathCW");
    }
} // end of class MathCW
```

The System.loadLibrary() function maps the generic name of the library to a platform-dependent name, such as libMathCW.so on most UNIX systems, libMathCW.dylib on Apple MAC OS X, or MathCW.dll on Microsoft WINDOWS, finds it in the Java library path, and loads its code into the virtual machine for subsequent use when the MathCW class functions are called.

Apart from the need to locate additional shared object libraries at run time, Java classes that define functions with the native type-modifier keyword behave exactly like ordinary classes implemented entirely in Java. However, there may be some sacrifice of security and stability, because code implemented outside the Java environment may not be as thoroughly checked as Java requires. For example, an out-of-bounds array index in Java is caught and handled gracefully. A similar bounds violation in dynamically loaded native code could corrupt or crash the entire Java Virtual Machine. That is not a concern for the mathcw library, however, because it is carefully written, and well tested with many different compilers on numerous operating systems and CPU architectures, and it makes little use of bug-prone and dangerous features, such as pointers and run-time array indexing.

Timing measurements on several systems with the perf.java benchmark program show that the additional overhead of the JNI wrappers is not significant, except for those functions with trivial implementations, such as fabs() and fmax(), or functions like sqrt() that are replaced with inline hardware instructions at run time by some Java just-in-time compilers. Indeed, on several systems, some of the native functions from the mathcw library are considerably faster than those provided by the standard Java Math class. That strongly suggests that the substantial effort that would be required to reimplement, test, and maintain a version of the mathcw library in pure Java code is not warranted.

J.3 Programming the Java C interface

Although the Java primitive types of booleans, characters, integers, and floating-point numbers have exact counterparts in the C language, all of the more complex Java data types require special handling to access them from another language. For the mathcw library, the only such types are one-dimensional arrays of characters or numbers.

In order to support that extra special handling, and allow bidirectional communication between Java and the native language, the C interface functions each have two additional arguments. They appear first in the argument lists, and have opaque data types JNIEnv * and jobject. For the mathcw package, we only use those arguments for those few functions that have arguments of strings or numeric arrays.

The C interface in MathCW.c begins with some header-file inclusions, and a wrapper for C++ compilation that ensures that the interface will be compiled with C-style linkage, rather than the default C++ style with mangled function names encoding signatures:

```
#include <jni.h>
#include <mathcw.h>

#ifdef __cplusplus
extern "C" {
#endif
```

Next come the definitions of two private helper functions that simplify the handling of character-string arguments:

```
static void
freestr(JNIEnv *env, jstring str, const char *s)
{
#ifdef __cplusplus
    (env)->ReleaseStringUTFChars(str, s);
#else
    (*env)->ReleaseStringUTFChars(env, str, s);
#endif
}

static const char *
getstr(JNIEnv *env, jstring str)
{
#ifdef __cplusplus
    return (env->GetStringUTFChars(str, NULL));
#else
    return ((*env)->GetStringUTFChars(env, str, NULL));
#endif
}
```

These messy definitions handle the different C and C++ interface conventions of the JNI. Their purpose is simple: `getstr()` converts a Java `String` in the Unicode character set to a C string in UTF-8 format stored in a dynamically allocated memory block, and `freestr()` frees that memory when it is no longer needed. The venerable ASCII character set occupies the first 128 slots in Unicode, so for our purposes, no additional character-set code mapping needs to be done. Were that needed, there is another JNI library routine, `JNU_GetStringNativeChars()`, that could be used instead [Lia99, §10.10, p. 138].

The bulk of the `MathCW.c` file consists of straightforward wrapper functions that look like this:

```
JNIEXPORT jfloat JNICALL
Java_MathCW_acos__F(JNIEnv *env, jobject obj, jfloat x)
{
    return (acosf(x));
}

JNIEXPORT jdouble JNICALL
Java_MathCW_acos__D(JNIEnv *env, jobject obj, jdouble x)
{
    return (acos(x));
}
```

The interface types `jfloat` and `jdouble` are equivalent to the native C types `float` and `double`, so no additional conversions or type casts are necessary. The complex function headers are taken directly from the `MathCW.h` header file that is automatically generated as described in **Section J.1** on page 979. The interface types `JNIEXPORT`, `JNICALL`, `jfloat`, and so on, are defined in the system header file `<jni.h>`, provided in every Java installation.

Only the functions with character-string or vector arguments require more than a function body consisting of a single `return` statement. Those with string arguments are all quite similar, so here is what just one of them looks like:

```
JNIEXPORT jfloat JNICALL
Java_MathCW_nan__F(JNIEnv *env, jobject obj, jstring tag)
```

```
{
    float result;
    const char * s;

    s = getstr(env, tag);

    if (s == (char *)NULL)
        result = nanf("");
    else
    {
        result = nanf(s);
        freestr(env, tag, s);
    }

    return (result);
}
```

The critical point here is that `getstr()` can fail. When it does, it returns a `NULL` pointer and registers an `OutOfMemory` exception to be thrown to the Java caller. However, the exception cannot be raised in the Java program until control returns from the native routine. Therefore, on failure, we simply pass an empty string to `nanf()`. Otherwise, we pass the user-provided string, and then call `freestr()` to free the memory for the temporary copy of the argument. In either case, the code properly returns a NaN.

This interface function shows how vector arguments are handled:

```
JNIEXPORT void JNICALL
Java_MathCW_vercwf(JNIEnv * env, jobject obj, jint n, jfloatArray u)
{
    float * v;

    v = (*env)->GetFloatArrayElements(env, u, NULL);

    if (v == (float *)NULL)
    {
        float q[1];
        int k;

        q[0] = qnanf("");

        for (k = 0; k < n; ++k)
            (*env)->SetFloatArrayRegion(env, u, k, 1, q);
    }
    else
    {
        vercwf(n, v);
        (*env)->ReleaseFloatArrayElements(env, u, v, 0);
    }
}
```

The Java library function `GetFloatArrayElements()` returns the address of the first element of the array, which might require making a copy to prevent its being moved by memory-management functions running in a background thread. The check for a `NULL`-pointer return handles the rare case where storage cannot be allocated, in which case, the returned vector is set to quiet NaNs. The `ReleaseFloatArrayElements()` function transfers the contents of the C array to the original Java array, if necessary, and frees any storage needed for a copy of the array.

The final code in `MathCW.c` provides the closing brace to match the one following the `extern` declaration at the beginning of the file, and it is visible only to C++ compilers:

```
#ifdef __cplusplus
}
#endif
```

J.4 Using the Java interface

Java requires calls to functions defined in other classes to be prefixed with the class name and a dot. For example, the square-root function provided by the standard `Math` class is invoked as `Math.sqrt(x)`. All that is needed to convert an existing program to use mathematical functions from the `MathCW` class is to change the class prefix.

Here is a short program that illustrates calls to the ordinary and complementary complete elliptic integral functions, which are absent from the standard `Math` class:

```
class sample
{
    public static void main (String[] args)
    {
        double x;

        for (x = -1.0; x <= 1.0; x += 0.125)
        {
            System.out.println("MathCW elle (" + x + ") = " + MathCW.elle (x));
            System.out.println("MathCW ellec(" + x + ") = " + MathCW.ellec(x));
        }
    }
}
```

Assuming that the `MathCW` class file and library are in the current directory, the sample program can be compiled and run like this:

```
% javac sample.java

% java -Djava.library.path=. sample
MathCW elle (-1.0) = 1.0
MathCW ellec(-1.0) = 1.5707963267948966
MathCW elle (-0.875) = 1.2011106307369144
MathCW ellec(-0.875) = 1.4742582575626793
...
MathCW elle (0.875) = 1.2011106307369144
MathCW ellec(0.875) = 1.4742582575626793
MathCW elle (1.0) = 1.0
MathCW ellec(1.0) = 1.5707963267948966
```

L Letter notation

> PERHAPS YOU HAVE LEARNED IT WITHOUT BOOK:
> BUT, I PRAY, CAN YOU READ ANY THING YOU SEE?
> AY, IF I KNOW THE LETTERS AND THE LANGUAGE.
>
> — SHAKESPEARE'S *Romeo and Juliet* (1595).

The tables in this Appendix summarize common uses of Latin and Greek letters in this book.

Table L.1: Latin letters used in mathematics in this book.

Name	Description
d, D	decimal digit
x, y	real variables
z	complex variable
$f(x)$	arbitrary function of x
$g(x)$	arbitrary function of x
$h(x)$	arbitrary function of x
B_n	n-th Bernoulli number
E_n	n-th Euler number
F_n	n-th Fibonacci number
T_n	n-th tangent number
G	guard bit in floating-point arithmetic
L	last stored bit in floating-point arithmetic
R	rounding bit in floating-point arithmetic
S	sticky bit in floating-point arithmetic
$\mathcal{O}(expr)$	order of *expr*
$P(expr)$	probability that *expr* is true
$\mathcal{P}(x)$	polynomial approximation
$\mathcal{Q}(x)$	polynomial approximation
$\mathcal{R}(x)$	rational polynomial approximation ($= \mathcal{P}(x)/\mathcal{Q}(x)$)
$E(m)$	complete elliptic integral function of second kind
$K(m)$	complete elliptic integral function of first kind
$T_n(u)$	Chebyshev polynomial of degree n of the first kind
$I_\nu(z)$	modified cylindrical Bessel function of order ν of first kind
$J_\nu(z)$	cylindrical Bessel function of order ν of first kind
$K_\nu(z)$	modified cylindrical Bessel function of order ν of second kind
$Y_\nu(z)$	cylindrical Bessel function of order ν of second kind
$i_\nu(z)$	modified spherical Bessel function of order ν of first kind
$j_\nu(z)$	spherical Bessel function of order ν of first kind
$k_\nu(z)$	modified spherical Bessel function of order ν of second kind
$y_\nu(z)$	spherical Bessel function of order ν of second kind

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2

Table L.2: Greek letters in mathematics used in this book and related literature, in their order of appearance in the Greek alphabet.

Symbol	**Description**
α, A	*alpha*; (lowercase) number base; (lowercase) variable name
β, B	*beta*; (lowercase) floating-point base
γ	*gamma*; Euler–Mascheroni constant ($\approx 0.577\,215$); with two arguments, complementary incomplete gamma function
Γ	capital *gamma*; with one argument, gamma function (generalized factorial function); with two arguments, ordinary incomplete gamma function
δ	*delta*; something small
Δ	capital *delta*; difference operator; triangle operator
ϵ, E	*epsilon*; (lowercase) something small; (lowercase) floating-point big machine epsilon $= \beta^{1-t}$, for base β and t-digit significand
ε	variant *epsilon*
ζ, Z	*zeta*; (lowercase) Riemann zeta function; (lowercase) variable name; (uppercase) elliptic function
η, H	*eta*; (lowercase) variable name; (uppercase) elliptic function
θ, Θ	*theta*; (lowercase) angle of rotation; (lowercase) variable name; (uppercase) elliptic function
ϑ	variant *theta*
ι, I	*iota*
κ, K	*kappa*
λ, Λ	*lambda*
μ, M	*mu*; (lowercase) arithmetic mean; (lowercase) variable name
ν, N	*nu*; (lowercase) frequency of electromagnetic radiation; (lowercase) number of degrees of freedom; (lowercase) order of Bessel function; (lowercase) variable name
ξ, Ξ	*xi*; (lowercase) variable name
o, O	*omicron*
π	*pi*; one of the most important numbers in mathematics; ratio of circumference of circle to its diameter
ϖ	variant *pi*
Π, $\prod$	capital *pi*; product operator
ρ, P	*rho*
ϱ	variant *rho*
σ	*sigma*; nonterminal form in Greek words; standard deviation in statistics
ς	variant *sigma*; terminal form in Greek words
Σ, $\sum$	capital *sigma*; summation operator
τ, T	*tau*; (lowercase) variable name
υ, Y	*upsilon*
ϕ, Φ	*phi*; (lowercase) angle of rotation; (lowercase) golden ratio; (lowercase) variable name; (uppercase) cumulative distribution function
φ	variant *phi*
χ, X	*chi*; χ^2 (chi-square) measure in statistics
ψ, Ψ	*psi*; (lowercase) psi function
ω, Ω	*omega*; (lowercase) angular frequency

P Pascal interface

BLAISE PASCAL (1623–1662): FRENCH MATHEMATICIAN,
SCIENTIST, AND PHILOSOPHER, AND AT THE AGE OF 19,
INVENTOR OF A CALCULATING MACHINE.

The Pascal programming language [JW91], first introduced on the CDC 6400 in 1970, attracted wide interest as a clean and simple language for teaching computer programming.

Thanks to its small size, and the free availability of a relatively portable implementation of a compiler that produced *P-code*, a simple assembly language for a small virtual machine, by the mid 1980s, Pascal had been ported to many computer systems, from 8-bit microcomputers to mainframes. The TEX typesetting system used for the production of this book is written in a markup language from which Pascal code and TEX documentation can be produced.

Several restrictions of the Pascal language, however, made it difficult to use for large programming projects, and for programs that need access to the underlying operating system. Its deficiencies are well chronicled in two famous papers [WSH77, Ker81], and when C compilers became widely available by the late 1980s, Pascal use fell sharply.

Nevertheless, some popular Pascal compilers introduced language extensions that alleviated many of the unpleasant restrictions, and that practical experience led to the revised 1990 ISO Extended Pascal Standard [PAS90]. The GNU Pascal compiler, `gpc`, largely conforms to that Standard, and offers language and library extensions for interfacing to the operating system, making Pascal potentially as available to programmers on modern computing systems as the C language is.

P.1 Building the Pascal interface

Before looking at the details of the interface between Pascal and the `mathcw` library, it is useful to see how the interface is built.

Initially, we have just a few files in the `pascal` subdirectory:

```
% cd pascal
% ls
exp         RCS         test02.pas  test06.pas  test10.pas
Makefile    README      test03.pas  test07.pas
mathcw.pas  test00.pas  test04.pas  test08.pas
okay        test01.pas  test05.pas  test09.pas
```

The `Makefile` contains the rules for the build process, and the `mathcw.pas` file contains the interface definition. The other `test*.pas` source files are simple tests of some of the library routines.

The `make` utility manages the build, which looks like this on a Sun Microsystems SOLARIS system:

```
% make

gpc -c  mathcw.pas

gpc -c  test00.pas
gpc  test00.o -o test00

gpc -o test01 test01.pas -L.. -lmcw

gpc -o test02 test02.pas -L.. -lmcw
```

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2

Table P.1: Numeric data types provided by the GNU Pascal compiler, gpc. The sizes shown are typical of modern 32-bit and 64-bit computers.

C type	Pascal types	size (bits)
signed char	ByteInt	8
short int	ShortInt	16
int	Integer	32
long int	MedInt	32 or 64
long long int	LongInt	64
float	ShortReal or Single	32
double	Real or Double	64
long double	LongReal or Extended	80 or 128
char *	CString	32 or 64
const char *	protected CString	32 or 64

```
gpc -o test03 test03.pas -L.. -lmcw

gpc -o test04 test04.pas -L.. -lmcw

gpc -c  test05.pas
gpc -o test05 test05.o mathcw.o -L.. -lmcw

...

gpc -c  test10.pas
gpc -o test10 test10.o mathcw.o -L.. -lmcw
```

The first step compiles the interface, and the remaining ones build the test programs. The first four test programs contain internal directives that tell the compiler how to find the additional object files in the parent directory. The remaining test programs get their object files directly from the mathcw object library.

A simple check verifies correct operation of all of the test programs, by running them and comparing their output with correct output stored in the okay subdirectory:

```
% make check

There should be no output but the test names:

========== test00
========== test01
...
========== test10
```

P.2 Programming the Pascal MathCW module

The original Pascal language had only one integer type, Integer, and one floating-point type, Real, with implementation-dependent sizes. The extended language supported by GNU gpc offers the numeric types shown in Table P.1. Pascal is a case-insensitive language, and the lettercase shown in the table is conventional in the GNU Pascal compiler's documentation. Other Pascal compilers may offer only some of those types, and may use different names.

As in C, the Pascal type sizes are platform dependent, but GNU Pascal ensures that the type correspondence with C is maintained. Thus, on a SPARC system, long int and MedInt are both 32-bit types, whereas on an Alpha, they are both 64-bit types.

The CString type is a useful extension provided by the GNU compiler to facilitate communicating with the operating system and its many support libraries. Pascal strings are normally represented by a data structure containing a length and an array of characters. The GNU compiler allocates an additional array element that is set to the NUL character to simplify conversion of a fixed-length Pascal string to a NUL-terminated C string.

Like C, Pascal normally uses *call-by-value* argument-passing conventions, but it uses *call-by-reference* when the argument is declared with the `var` keyword. That makes all of the arguments needed for the `mathcw` library easily representable in Pascal.

Pascal is a strongly typed language designed for fast one-pass compilation. That means that all functions, procedures, and variables must be declared before use. The GNU compiler allows a function or procedure body to be replaced by `external name 'rtnname'`, to tell the compiler that the definition is provided elsewhere in another language in a routine named `rtnname`. A native Pascal implementation of the double-precision square-root function might be written like this:

```
function Sqrt (x : Real) : Real;
begin
    { implementation code omitted }
end;
```

For an external implementation in C, we can instead write this:

```
function Sqrt (x : Real) :                       Real; external name 'sqrt';
```

If only one or two functions from the `mathcw` library are required in a Pascal program, then it is not difficult to supply explicit declarations such as those that we illustrated for `Sqrt()`. We then have a choice of how to tell the compiler where to find the external routines: by linker directives embedded in comments in the source code, or by supplying either object files or object libraries at link time. The first four test programs take the first approach. For example, `test00.pas` begins like this:

```
program test00(input,output);

{$L '../adxf.o'}
{$L '../expf.o'}
{$L '../fabsf.o'}
{$L '../inftyf.o'}
{$L '../isinff.o'}
{$L '../isnanf.o'}
{$L '../psif.o'}
{$L '../psilnf.o'}
{$L '../qnanf.o'}

var
   i : Integer;
   x : ShortReal;

function Psif   (x : ShortReal) :                ShortReal; external name 'psif';
function Psilnf (x : ShortReal) :                ShortReal; external name 'psilnf';
```

If each referenced function were defined entirely in a single object file, then only two linker directives would be needed. However, that is rarely the case, and as the code fragment shows, the directive approach requires introducing a hard-to-maintain list of object-file dependencies, exposing irrelevant details that should be hidden.

The test programs `test01.pas`, `test02.pas`, and `test03.pas` provide directives for just the object files that they explicitly require, and the `Makefile` supplies the library name and location for linking.

A much better approach is to define a Pascal `module` containing all of the function prototypes, and then simply reference that module whenever it is needed. The `mathcw.pas` file that defines the *interface part* of the module begins like this:

```
module MathCW interface;
```

The `module` statement provides the name, `MathCW`, by which the module can be referenced in other source files.

Next comes the `export` statement that supplies a long list of names defined in the interface that are available to users of the module. For convenience, the list is sorted alphabetically within each precision family:

```
export MathCW = (
       { ShortReal family }
       acosf, acoshf, adxf, asinf, asinhf, atanf, atan2f,
       ...

       { Real family }
       acos, acosh, adx, asin, asinh, atan, atan2,
       ...

       { LongReal family }
       acosl, acoshl, adxl, asinl, asinhl, atanl, atan2l,
       ...
       );
```

The declarations of the single-precision functions are mostly straightforward:

```
function acosf(x : ShortReal) :                  ShortReal; external name 'acosf';
function acoshf(x : ShortReal) :                 ShortReal; external name 'acoshf';
...
function frexpf(x : ShortReal; var n : Integer) : ShortReal; external name 'frexpf';
...
function nanf(protected s : CString) :           ShortReal; external name 'nanf';
...
```

The declarations for `frexpf()` and `nanf()` show how pointer arguments are handled. The `protected` keyword prevents modification of the argument inside the called function when that function is written in Pascal. Here, it guarantees that a *copy* of the original argument is passed to the function. The effect is thus somewhat like that of the C-language `const` qualifier.

The double-precision function declarations are similar, except for the special case of the power function, which must be renamed to avoid a collision with the Pascal `pow` operator:

```
function acos(x : Real) :                        Real; external name 'acos';
function acosh(x : Real) :                       Real; external name 'acosh';
...
function powd(x : Real; y : Real) :              Real; external name 'pow';
...
```

The interface part of the module ends with the extended-precision function declarations:

```
function acosl(x : LongReal) :                   LongReal; external name 'acosl';
function acoshl(x : LongReal) :                  LongReal; external name 'acoshl';
...

end.
```

The remainder of the `mathcw.pas` file defines the *implementation part* of the module:

```
module MathCW implementation;

  to begin do
  begin
    { module initialization code here: none currently needed }
  end;

  to end do
  begin
    { module termination code here: none currently needed }
  end;

end.
```

Any native Pascal code required for the module would be provided in the implementation part, along with the initialization and termination code blocks. We do not require any real implementation code here, because the mathcw library provides it elsewhere, so the implementation block could have been left empty, or even omitted. We prefer to keep a small template in place, however, in case an implementation block is needed in future extensions of the library or its interface from Pascal.

Compilation of the `mathcw.pas` file produces two output files:

```
% ls mathcw.*
mathcw.pas

% make mathcw.o
gpc -c  mathcw.pas

% ls -lo mathcw.*
-rw-rw-r-- 1 mcw 107769 2006-04-19 09:23 mathcw.gpi
-rw-rw-r-- 1 mcw    872 2006-04-19 09:23 mathcw.o
-rw-rw-r-- 1 mcw  22105 2006-04-19 05:57 mathcw.pas
```

The large interface file, `mathcw.gpi` needs to be available to the compiler for use by other programs. Only the small object file, `mathcw.o`, needs to be provided to the linker.

P.3 Using the Pascal module interface

There are no standard conventions about where GNU Pascal module files are stored for system-wide use, so, for simplicity, we assume that the `mathcw.pas` module file is in the same directory as the code that requires it.

All that is needed to make the entire mathcw library available to a Pascal program is a single `import` statement following the `program` statement. Here is a fragment of `test05.pas` that shows how:

```
program test05(input,output);

import MathCW;

var
   i : Integer;
   x : ShortReal;

begin
   writeln(’Test of Pascal interface to MathCW library:’);
   writeln(’’);

   x := -2.0;

   for i := 0 to 32 do
   begin
      writeln(’MathCW    erff(’, x:6:3, ’) = ’, erff(x):10:6);
      x := x + 0.125;
   end;
...
end.
```

The only additional thing that needs to be done is to supply the interface object file and the library name and location at link time, like this:

```
% gpc -o test05 test05.pas mathcw.o -L.. -lmcw
```

Given the severe limitations of historical Pascal implementations, it is clear that external function declarations and modules provide a powerful extension mechanism for the language, and they make it easy to interface Pascal code to the huge collection of libraries written in C on modern operating systems. What is lacking, compared to Ada, C++, C#, and Java, is the additional notion of a module or namespace qualifier on function names to disambiguate name conflicts.

P.4 Pascal and numeric programming

The numeric function repertoire of early Pascal was a small subset of that of Fortran, but the 1990 revision improved that situation, and also added a complex type with constructor functions cmplx(r, i) and polar(r,theta), component extractor functions re(z) and im(z), and elementary functions with the same names as their real counterparts. The 1990 extensions include based integers for any base from 2 to 36 (e.g., 16#cafefeed), but alas, no hexadecimal representation of exact floating-point constants, and no predefined names for the various numeric limits and parameters that C supplies in the <float.h> and <limits.h> header files. The GNU gpc compiler adds interfaces to assembly language, and to the GMP multiple-precision arithmetic library. The limited control of numeric output formatting remains a weak spot in the Pascal language.

The test program in the file numtest.pas exercises many features of IEEE 754 64-bit arithmetic, and, at least with the GNU gpc compiler, demonstrates that the IEEE nonstop model of computation is fully supported, and Infinity, NaN, and signed zero are handled correctly. A store() function provides a workaround for the lack of a volatile qualifier, and prevents higher-precision intermediate computation when that is injurious to the job. The needed helper functions are easily expressed in Pascal:

```
function IsInf(x : Real) : Boolean;
    begin IsInf := (abs(1.0 / x) = 0.0) and (abs(x) > 1.0) end;

function IsNaN(x : Real) : Boolean;
    begin IsNaN := (x <> x) end;

function Store(x : Real) : Real;
    begin Store := x end;
```

With a substantial external numeric function library interfaced to the language in a similar way to that described earlier in this Appendix, Pascal programmers can enjoy a comfortable numeric programming environment akin to that available in C and Fortran 77.

Bibliography

> (1870) The annals of bibliography afford many examples
> of the delirious extent to which book-fancying can go.
>
> — *Oxford English Dictionary*.

Entries are followed by a braced list of page numbers where the entry is cited. Personal names are indexed in the separate author/editor index on page 1039. Some bibliography entries cite other entries, so the braced lists may include pages within the bibliography.

Entries include CODEN (*Chemical Abstracts Periodical Number*), DOI (*Digital Object Identifier*), ISBN (*International Standard Book Number*), ISSN (*International Standard Serial Number*), LCCN (*US Library of Congress Call Number*), and Web URL (*Uniform Resource Locator*) data, where available.

Prefix http://doi.org/ to any DOI value to convert it to a valid Web address.

Should you find that a URL recorded here is no longer accessible, you may be able to locate a historical copy on the *Internet Archive WayBack Machine* at http://www.archive.org/.

[AAR99] George E. Andrews, Richard Askey, and Ranjan Roy. *Special Functions*, volume 71 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, UK, 1999. ISBN 0-521-62321-9 (hardcover), 0-521-78988-5 (paperback); 978-0-521-62321-6 (hardcover), 978-0-521-78988-2 (paperback). xvi + 664 pp. LCCN QA351 .A74 1999. {**521, 619, 630, 827**}

[ABB64] Gene M. Amdahl, Gerrit A. Blaauw, and Frederick P. Brooks, Jr. Architecture of the IBM System/360. *IBM Journal of Research and Development*, 8(2):87–102, April 1964. CODEN IBMJAE. ISSN 0018-8646 (print), 2151-8556 (electronic). URL http://www.research.ibm.com/journal/50th/architectures/amdahl.html; http://www.research.ibm.com/journal/rd/082/ibmrd0802C.pdf. DOI 10.1147/rd.82.0087. {**963**}

[ABB00] Gene M. Amdahl, Gerrit A. Blaauw, and Frederick P. Brooks, Jr. Architecture of the IBM System/360. *IBM Journal of Research and Development*, 44(1/2):21–36, January/March 2000. CODEN IBMJAE. ISSN 0018-8646 (print), 2151-8556 (electronic). URL http://www.research.ibm.com/journal/rd/441/amdahl.pdf. DOI 10.1147/rd.441.0021. Special issue: reprints on Evolution of information technology 1957–1999. {**963**}

[ABC^{+}99] Paul H. Abbott, David G. Brush, Clarence W. Clark III, Chris J. Crone, John R. Ehrman, Graham W. Ewart, Clark A. Goodrich, Michel Hack, John S. Kapernick, Brian J. Minchau, William C. Shepard, Ronald M. Smith, Sr., Richard Tallman, Steven Walkowiak, Akio Watanabe, and W. Romney White. Architecture and software support in IBM S/390 Parallel Enterprise Servers for IEEE floating-point arithmetic. *IBM Journal of Research and Development*, 43(5/6):723–760, September/November 1999. CODEN IBMJAE. ISSN 0018-8646 (print), 2151-8556 (electronic). URL http://www.research.ibm.com/journal/rd/435/abbott.html. DOI 10.1147/rd.435.0723. Besides important history of the development of the S/360 floating-point architecture, this paper has a good description of IBM's algorithm for exact decimal-to-binary conversion, complementing earlier ones [Cli90, Knu90, SW90, BD96, SW04]. {**895, 998, 1004, 1034**}

[ABM^{+}97] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener, editors. *Fortran 95 Handbook: Complete ISO/ANSI Reference*. MIT Press, Cambridge, MA, USA, November 1997. ISBN 0-262-51096-0; 978-0-262-51096-7. xii + 711 pp. LCCN QA76.73.F25 F6 1997. URL http://www.cbooks.com/sqlnut/SP/search/gtsumt?source=&isbn=0262510960. {**106**}

[Ada69] Arthur G. Adams. Remark on Algorithm 304 [S15]: Normal curve integral. *Communications of the Association for Computing Machinery*, 12(10):565–566, October 1969. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI 10.1145/363235.363253. {**618**}

[Ada83] *American National Standard Reference Manual for the Ada Programming Language: ANSI/MIL-STD-1815A*. American National Standards Institute, New York, NY, USA, 1983. {**vii**}

[Ada95] *ISO/IEC 8652:1995: Information technology — Programming languages — Ada*. International Organization for Standardization, Geneva, Switzerland, 1995. 511 pp. URL http://www.adaic.org/standards/05rm/RM-Final.pdf. Available in English only. {**vii, 911, 928**}

[Ada12] *ISO/IEC 8652:2012 Information technology — Programming languages — Ada*. International Organization for Standardization, Geneva, Switzerland, 2012. 832 (est.) pp. URL http://www.ada-auth.org/standards/ada12.html; http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=61507. {**vii**}

[ADW77a] Donald E. Amos, S. L. Daniel, and M. K. Weston. Algorithm 511: CDC 6600 subroutines IBESS and JBESS for Bessel functions $I_\nu(x)$ and $J_\nu(x)$, $x \geq 0, \nu \geq 0$ [S18]. *ACM Transactions on Mathematical Software*, 3(1):93–95, March 1977. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI 10.1145/355719.355727. See erratum [Amo78]. {**693, 996**}

[ADW77b] Donald E. Amos, S. L. Daniel, and M. K. Weston. CDC 6600 subroutines IBESS and JBESS for Bessel functions $I_\nu(x)$ and $J_\nu(x)$, $x \geq 0, \nu \geq 0$. *ACM Transactions on Mathematical Software*, 3(1):76–92, March 1977. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI 10.1145/355719.355726. {**693**}

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2

[AE06] J. V. Armitage and W. F. Eberlein. *Elliptic Functions*. London Mathematical Society student texts. Cambridge University Press, Cambridge, UK, 2006. ISBN 0-521-78078-0 (hardcover), 0-521-78563-4 (paperback); 978-0-521-78078-0 (hardcover), 978-0-521-78563-1 (paperback). xiii + 387 pp. LCCN QA343 .A95 2006. {**627**}

[AF09] Alan Agresti and Christine A. Franklin. *Statistics: The Art and Science of Learning from Data*. Pearson Prentice Hall, Upper Saddle River, NJ 07458, USA, second edition, 2009. ISBN 0-13-513199-5 (student edition), 0-13-513240-1 (instructor edition); 978-0-13-513199-2 (student edition), 978-0-13-513240-1 (instructor edition). xxviii + 769 + 47 pp. LCCN QA276.12 .A35 2009. {**196**}

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, USA, May 1, 1996. ISBN 0-201-63455-4; 978-0-201-63455-6. xviii + 333 pp. LCCN QA76.73.J38A76 1996. {**vii**}

[AG98] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, USA, second edition, 1998. ISBN 0-201-31006-6; 978-0-201-31006-1. xix + 442 pp. LCCN QA76.73.J38A76 1998. {**vii**}

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Reading, MA, USA, third edition, 2000. ISBN 0-201-70433-1; 978-0-201-70433-4. xxiv + 595 pp. LCCN QA76.73.J38 A76 2000. {**vii**}

[AH01] Jörg Arndt and Christoph Haenel. *Pi — Unleashed*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2001. ISBN 3-540-66572-2; 978-3-540-66572-4. xii + 270 pp. LCCN QA484.A7513 2001. DOI `10.1007/978-3-642-56735-3`. Includes CD-ROM. Translated by Catriona and David Lischka from the 1998 German original, *Pi: Algorithmen, Computer, Arithmetik*. {**14, 623, 632**}

[AHG$^+$04] Brad Abrams, Anders Hejlsberg, Brian Grunkemeyer, Joel Marcey, Kit George, Krzysztof Cwalina, and Jeffrey Richter. *.NET Framework Standard Library Annotated Reference. Volume 1: Base Class Library and Extended Numerics Library*. Microsoft .NET development series. Addison-Wesley, Reading, MA, USA, 2004. ISBN 0-321-15489-4 (hardcover); 978-0-321-15489-7 (hardcover). xxvi + 528 pp. LCCN QA76.76.M52 A27 2004. URL `http://www.aw-bc.com/catalog/academic/product/0,1144,0321154894,00.html`. Foreword by Joel Marcey. {**917**}

[Ale10] Andrei Alexandrescu. *The D programming language*. Addison-Wesley, Reading, MA, USA, 2010. ISBN 0-321-65953-8 (hardcover), 0-321-63536-1 (paperback); 978-0-321-65953-8 (hardcover), 978-0-321-63536-5 (paperback). xxvii + 463 pp. LCCN QA76.73.D138 A44 2010; QA76.73.D138. {**830**}

[AM59] Robert L. Ashenhurst and Nicholas Metropolis. Unnormalized floating point arithmetic. *Journal of the Association for Computing Machinery*, 6(3):415–428, July 1959. CODEN JACOAH. ISSN 0004-5411 (print), 1557-735x (electronic). DOI `10.1145/320986.320996`. {**960, 966**}

[Ami62] D. Amit. Algorithm 147 [S14]: PSIF. *Communications of the Association for Computing Machinery*, 5(12):605, December 1962. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/355580.369107`. See certifications [Tha63, Par69]. {**521, 1027, 1035**}

[Amm77] Urs Ammann. On code generation in a PASCAL compiler. *Software — Practice and Experience*, 7(3):391–423, May/June 1977. CODEN SPEXBL. ISSN 0038-0644 (print), 1097-024X (electronic). DOI `10.1002/spe.4380070311`. {**949**}

[Amo78] Donald E. Amos. Erratum: "Algorithm 511: CDC 6600 subroutines IBESS and JBESS for Bessel functions $I_\nu(x)$ and $J_\nu(x)$, $x \geq 0, \nu \geq 0$ [S18]". *ACM Transactions on Mathematical Software*, 4(4):411, December 1978. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/356502.356501`. See [ADW77a]. {**995**}

[Amo83] Donald E. Amos. Algorithm 610: A portable FORTRAN subroutine for derivatives of the psi function. *ACM Transactions on Mathematical Software*, 9(4):494–502, December 1983. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/356056.356065`. {**521**}

[Amo86] Donald E. Amos. Algorithm 644: A portable package for Bessel functions of a complex argument and nonnegative order. *ACM Transactions on Mathematical Software*, 12(3):265–273, September 1986. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.acm.org/pubs/citations/journals/toms/1986-12-3/p265-amos/`. DOI `10.1145/7921.214331`. See remarks [Amo90, Amo95, Kod07]. {**693, 996, 1020**}

[Amo90] Donald E. Amos. Remark on "Algorithm 644: A portable package for Bessel functions of a complex argument and nonnegative order". *ACM Transactions on Mathematical Software*, 16(4):404, December 1990. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.acm.org/pubs/citations/journals/toms/1990-16-4/p404-amos/`. DOI `10.1145/98267.98299`. See [Amo86, Amo95, Kod07]. {**996, 1020**}

[Amo95] Donald E. Amos. A remark on Algorithm 644: A portable package for Bessel functions of a complex argument and nonnegative order. *ACM Transactions on Mathematical Software*, 21(4):388–393, December 1995. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/212066.212078`. See [Amo86, Amo90, Kod07]. {**996, 1020**}

[And98] Larry C. Andrews. *Special Functions of Mathematics for Engineers*. Oxford University Press, Oxford, UK, second edition, 1998. ISBN 0-19-856558-5 (Oxford hardcover), 0-8194-2616-4 (SPIE Press hardcover); 978-0-19-856558-1 (Oxford hardcover), 978-0-8194-2616-1 (SPIE Press). xvii + 479 pp. LCCN QA351 .A75 1998. {**827**}

[AND15] P. Ahrens, H. D. Nguyen, and J. Demmel. Efficient reproducible floating point summation and BLAS. Report UCB/EECS-2015-229, EECS Department, University of California, Berkeley, Berkeley, CA, USA, December 8, 2015. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-229.html`. {**385**}

[AND16] P. Ahrens, H. D. Nguyen, and J. Demmel. Efficient reproducible floating point summation and BLAS. Report UCB/EECS-2016-121, EECS Department, UC Berkeley, Berkeley, CA, USA, June 18, 2016. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-121.html`. {**385**}

[Ano94] Anonymous. Corrigenda. *ACM Transactions on Mathematical Software*, 20(4):553, December 1994. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). See [HFT94]. {**1014**}

[ANS87] ANSI/IEEE. *ANSI/IEEE Std 854-1987: An American National Standard: IEEE Standard for Radix-Independent Floating-Point Arithmetic*. IEEE, New York, NY, USA, October 5, 1987. ISBN 0-7381-1167-8; 978-0-7381-1167-4. v + 14 pp. URL `http://ieeexplore.ieee.org/iel1/2502/1121/00027840.pdf`. Revised 1994. INSPEC Accession Number: 3095617. {**1, 104, 109, 827, 928, 966**}

[ANSI78] *American National Standard Programming Language FORTRAN: Approved April 3, 1978, American National Standards Institute, Inc.: ANSI X3.9-1978. Revision of ANSI X3.9-1966*. American National Standards Institute, New York, NY, USA, revised edition, 1978. 438 pp. {**vii, 341**}

[ANSI97] *ANSI/ISO/IEC 1539-1:1997: Information Technology — Programming Languages — Fortran — Part 1: Base language*. American National Standards Institute, New York, NY, USA, 1997. URL `http://www.fortran.com/fortran/iso1539.html`. {**106**}

[AS64] Milton Abramowitz and Irene A. Stegun, editors. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, volume 55 of *Applied mathematics series*. U. S. Department of Commerce, Washington, DC, USA, 1964. xiv + 1046 pp. LCCN QA47.A161 1972; QA 55 A16h 1972. Tenth printing, with corrections (December 1972). This book is also available online at `http://www.convertit.com/Go/ConvertIt/Reference/AMS55.ASP` in bitmap image format. {**6, 58, 59, 196, 269, 301, 303, 341, 498, 521, 560, 562, 587, 589, 593, 600, 619, 624, 632, 638, 643, 651, 657, 661, 666, 673, 675, 678, 681–683, 689, 693, 731, 826**}

[AT17] Jared L. Aurentz and Lloyd N. Trefethen. Chopping a Chebyshev series. *ACM Transactions on Mathematical Software*, 43(4):33:1–33:21, March 2017. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/2998442`. {**57**}

[AW88] Lothar Afflerbach and Klaus Wenzel. Normal random numbers lying on spirals and clubs. *Statistical Papers = Statistische Hefte*, 29 (1):237–244, December 1988. CODEN STPAE4. ISSN 0932-5026 (print), 1613-9798 (electronic). URL `http://www.springerlink.com/content/q7885421202m6565/`. DOI `10.1007/BF02924529`. {**193**}

[AW05] George B. Arfken and Hans-Jürgen Weber. *Mathematical Methods for Physicists*. Elsevier, Amsterdam, The Netherlands, sixth edition, 2005. ISBN 0-12-059876-0, 0-12-088584-0 (paperback); 978-0-12-059876-2, 978-0-12-088584-8 (paperback). xii + 1182 pp. LCCN QA37.3 .A74 2005. {**8, 196, 303, 582, 627, 693**}

[AWH13] George B. Arfken, Hans-Jürgen Weber, and Frank E. Harris. *Mathematical Methods for Physicists: a Comprehensive Guide*. Elsevier Academic Press, Amsterdam, The Netherlands, seventh edition, 2013. ISBN 0-12-384654-4 (hardcover), 1-4832-7782-8 (e-book); 978-0-12-384654-9 (hardcover), 978-1-4832-7782-0 (e-book). xiii + 1205 pp. LCCN QA37.3 .A74 2013. URL `http://www.sciencedirect.com/science/book/9780123846549`. {**8, 196, 303, 582, 627, 693**}

[Ayo74] Raymond Ayoub. Euler and the zeta function. *American Mathematical Monthly*, 81(10):1067–1086, December 1974. CODEN AMMYAE. ISSN 0002-9890 (print), 1930-0972 (electronic). URL `http://www.jstor.org/stable/2319041`. DOI `10.2307/2319041`. {**579, 590**}

[Bai81] B. J. R. Bailey. Alternatives to Hastings' approximation to the inverse of the normal cumulative distribution function. *Applied Statistics*, 30(3):275–276, 1981. CODEN APSTAG. ISSN 0035-9254 (print), 1467-9876 (electronic). URL `http://www.jstor.org/stable/2346351`. DOI `10.2307/2346351`. {**618**}

[Bai95] David H. Bailey. A Fortran-90 based multiprecision system. *ACM Transactions on Mathematical Software*, 21(4):379–387, December 1995. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/212066.212075`. See also extension to complex arithmetic [Smi98]. {**1032**}

[Bak61] Frank B. Baker. A method for evaluating the area of the normal function. *Communications of the Association for Computing Machinery*, 4 (5):224–225, May 1961. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/366532.366544`. {**618**}

[Bak92] Louis Baker. *C Mathematical Function Handbook*. McGraw-Hill programming tools for scientists and engineers. McGraw-Hill, New York, NY, USA, 1992. ISBN 0-07-911158-0; 978-0-07-911158-6. xviii + 757 pp. LCCN QA351.B17 1991; QA351 .B17 1992. {**521, 556, 558, 567, 583, 589, 593, 657, 682, 827**}

[Ban98] Jerry Banks, editor. *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*. Wiley, New York, NY, USA, 1998. ISBN 0-471-13403-1 (hardcover); 978-0-471-13403-9 (hardcover). xii + 849 pp. LCCN T57.62 .H37 1998. DOI `10.1002/9780470172445`. {**1003, 1021**}

[Bar92] John D. Barrow. *Pi in the Sky: Counting, Thinking, and Being*. Clarendon Press, Oxford, UK, 1992. ISBN 0-19-853956-8; 978-0-19-853956-8. ix + 317 pp. LCCN QA36 .B37 1992. {**14, 623**}

[Bar96] John D. Barrow. *Pi in the Sky: Counting, Thinking, and Being*. Little, Brown and Company, Boston, Toronto, London, 1996. ISBN 0-316-08259-7; 978-0-316-08259-4. ix + 317 pp. LCCN QA36 .B37 1994. {**14, 623**}

[Bar06] Jason Socrates Bardi. *The Calculus Wars: Newton, Leibniz, and the Greatest Mathematical Clash of all Time*. Thunder's Mouth Press, New York, NY, USA, 2006. ISBN 1-56025-992-2, 1-56025-706-7; 978-1-56025-992-3, 978-1-56025-706-6. viii + 277 pp. LCCN QA303 .B2896 2006. {**8**}

[Bay90] Carter Bays. C364. Improving a random number generator: a comparison between two shuffling methods. *Journal of Statistical Computation and Simulation*, 36(1):57–59, May 1990. CODEN JSCSAJ. ISSN 0094-9655 (print), 1026-7778 (electronic), 1563-5163. URL `http://www.tandfonline.com/doi/abs/10.1080/00949659008811264`. DOI `10.1080/00949659008811264`. See [LL73, BD76] for the two nearly identical shuffling algorithms. This paper explains why the first does not lengthen the generator period, or much reduce the lattice structure of linear congruential generators, but the second improves both dramatically. {**179, 998, 1022**}

[BB83] Jonathan M. Borwein and Peter B. Borwein. A very rapidly convergent product expansion for π [pi]. *BIT*, 23(4):538–540, December 1983. CODEN BITTEL, NBITAB. ISSN 0006-3835 (print), 1572-9125 (electronic). URL `http://www.springerlink.com/openurl.asp?genre=article&issn=0006-3835&volume=23&issue=4&spage=538`. DOI `10.1007/BF01933626`. {**623**}

[BB87a] Adam W. Bojanczyk and Richard P. Brent. A systolic algorithm for extended GCD computation. *Computers and Mathematics with Applications*, 14(4):233–238, 1987. CODEN CMAPDK. ISSN 0898-1221 (print), 1873-7668 (electronic). DOI `10.1016/0898-1221(87)90130-1`. {**186**}

[BB87b] Jonathan M. Borwein and Peter B. Borwein. *Pi and the AGM: a Study in Analytic Number Theory and Computational Complexity*. Canadian Mathematical Society series of monographs and advanced texts = Monographies et études de la Société mathématique du Canada. Wiley, New York, NY, USA, 1987. ISBN 0-471-83138-7, 0-471-31515-X (paperback); 978-0-471-83138-9, 978-0-471-31515-5 (paperback). xv + 414 pp. LCCN QA241 .B774 1987. {**623**}

[BB97] Gerrit A. Blaauw and Frederick P. Brooks, Jr. *Computer Architecture: Concepts and Evolution*. Addison-Wesley, Reading, MA, USA, 1997. ISBN 0-201-10557-8; 978-0-201-10557-5. xlviii + 1213 pp. LCCN QA76.9.A73 B57 1997. {**104, 947, 978**}

[BB00] Nelson H. F. Beebe and James S. Ball. Algorithm xxx: Quadruple-precision $\Gamma(x)$ and $\psi(x)$ functions for real arguments. Technical report, Departments of Mathematics and Physics, University of Utah, Salt Lake City, UT 84112, USA, 2000. {**521, 525**}

[BB04] Jonathan M. Borwein and David H. Bailey. *Mathematics by Experiment: Plausible Reasoning in the 21st Century*. A. K. Peters, Wellesley, MA, USA, 2004. ISBN 1-56881-211-6; 978-1-56881-211-3. x + 288 pp. LCCN QA76.95 .B67 2003. Due to an unfortunate error, some of the citations in the book point to the wrong item in the Bibliography. Here is how to find the correct citation number: [1]–[85]: Citation number is correct; [86, page 100]: [86]; [86, page 2]: [87]; [87]–[156]: Add one to citation number; [157]: [159]; [158, page 139]: [158]; [158, page 97]: [160]; [159]–[196]: Add two to citation number. {**268, 622, 624, 631, 632**}

[BBB59] Frederick P. Brooks, Jr., Gerrit A. Blaauw, and Werner Buchholz. Processing data in bits and pieces. *IRE Transactions on Electronic Computers*, EC-8(2):118–124, June 1959. CODEN IRELAO. ISSN 0367-9950. URL `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5219512`. DOI `10.1109/TEC.1959.5219512`. This paper contains on page 121 the first published reference to the term "byte". An article of the same title appears in "Information Processing, Proceedings of the International Conference on Information Processing, UNESCO, Paris, 15–20 June 1959", pp. 375–381, 1959. From [Buc62, page 40]: "Byte denotes a group of bits used to encode a character, or the number of bits transmitted in parallel to and from input-output units. A term other than character is used here because a given character may be represented in different applications by more than one code, and different codes may use different numbers of bits (i.e., different byte sizes). In input-output transmission the grouping of bits may be completely arbitrary and have no relation to actual characters. (The term is coined from bite, but respelled to avoid accidental mutation to bit.)". {**969**}

[BBB00] Lennart Berggren, Jonathan M. Borwein, and Peter B. Borwein, editors. *Pi, a Sourcebook*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., second edition, 2000. ISBN 0-387-98946-3 (hardcover); 978-0-387-98946-4 (hardcover). xix + 736 pp. LCCN QA484 .P5 2000. {**14, 59, 623**}

[BBBP97] David H. Bailey, Jonathan M. Borwein, Peter B. Borwein, and Simon Plouffe. The quest for pi. *The Mathematical Intelligencer*, 19(1):50–57, January 1997. CODEN MAINDC. ISSN 0343-6993 (print), 1866-7414 (electronic). URL `http://crd.lbl.gov/~dhbailey/dhbpapers/pi-quest.pdf; http://docserver.carma.newcastle.edu.au/164/; http://link.springer.com/article/10.1007%2FBF03024340`. DOI `10.1007/BF03024340`. {**622, 623**}

[BBC^{+}07] David H. Bailey, Jonathan M. Borwein, Neil J. Calkin, Roland Girgensohn, D. Russell Luke, and Victor Moll. *Experimental Mathematics in Action*. A. K. Peters, Wellesley, MA, USA, 2007. ISBN 1-56881-271-X; 978-1-56881-271-7. xii + 322 pp. LCCN QA8.7 .E97 2007. {**631**}

[BBG03] Jonathan M. Borwein, David H. Bailey, and Roland Girgensohn. *Experimentation in Mathematics: Computational Paths to Discovery*. A. K. Peters, Wellesley, MA, USA, 2003. ISBN 1-56881-136-5; 978-1-56881-136-9. x + 357 pp. LCCN QA12 .B67 2004. {**631**}

[BC09] Franky Backeljauw and Annie Cuyt. Algorithm 895: A continued fractions package for special functions. *ACM Transactions on Mathematical Software*, 36(3):15:1–15:20, July 2009. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/1527286.1527289`. {**776, 827**}

[BCD^{+}14] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica*, 23:1–155, 2014. CODEN ANUMFU. ISSN 0962-4929 (print), 1474-0508 (electronic). DOI `10.1017/S0962492914000038`. {**385**}

[BCDH09] Javier D. Bruguera, Marius Cornea, Debjit DasSarma, and John Harrison, editors. *Proceedings of the 19th IEEE Symposium on Computer Arithmetic, June 8–10, 2009, Portland, Oregon, USA*. IEEE Computer Society Press, Silver Spring, MD, USA, 2009. ISBN 0-7695-3670-0; 978-0-7695-3670-5. ISSN 1063-6889. LCCN QA76.6. URL `http://www.ac.usc.es/arith19/`. {**1014**}

[BD76] Carter Bays and S. D. Durham. Improving a poor random number generator. *ACM Transactions on Mathematical Software*, 2(1):59–64, March 1976. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/355666.355670`. See also [LL73] for a slightly different, but inferior, shuffling algorithm, and [Bay90] for a comparison, both mathematical, and graphical, of the two algorithms. Reference [3] for IBM Report GC20-8011-0 is incorrectly given year 1969; the correct year is 1959. {**178, 997, 1022**}

[BD96] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. *ACM SIGPLAN Notices*, 31(5):108–116, May 1996. CODEN SINODQ. ISSN 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). URL `http://www.acm.org:80/pubs/citations/proceedings/pldi/231379/p108-burger/`. DOI `10.1145/231379.231397`. This paper offers a significantly faster algorithm than that of [SW90], together with a correctness proof and an implementation in Scheme. See also [Cli90, ABC^{+}99, SW04, Cli04]. {**895, 995, 1004, 1034**}

[BD03a] Sylvie Boldo and Marc Daumas. Representable correcting terms for possibly underflowing floating point operations. In Bajard and Schulte [BS03], pages 79–86. ISBN 0-7695-1894-X; 978-0-7695-1894-7. ISSN 1063-6889. LCCN QA76.6 .S919 2003. URL `http://www.dec.usc.es/arith16/papers/paper-156.pdf`. DOI `10.1109/ARITH.2003.1207663`. {**366, 1002**}

[BD03b] Sylvie Boldo and Marc Daumas. A simple test qualifying the accuracy of Horner's rule for polynomials. Research Report 2003-01, École Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France, January 2003. 41 pp. URL `ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-4707.pdf`. {**89**}

[BD04] Sylvie Boldo and Marc Daumas. A simple test qualifying the accuracy of Horner's rule for polynomials. *Numerical Algorithms*, 37(1–4): 45–60, December 2004. CODEN NUALEG. ISSN 1017-1398 (print), 1572-9265 (electronic). URL `http://link.springer.com/article/10.1023/B%3ANUMA.0000049487.98618.61`. DOI `10.1023/B:NUMA.0000049487.98618.61`. SCAN2002 International Conference (Guest Editors: Rene Alt and Jean-Luc Lamotte). {**89**}

[BD09] Jonathan M. Borwein and Keith J. Devlin. *The Computer as Crucible: an Introduction to Experimental Mathematics*. A. K. Peters, Wellesley, MA, USA, 2009. ISBN 1-56881-343-0; 978-1-56881-343-1. xi + 158 pp. LCCN QA8.7 .B67 2009. {**631**}

[BDS07] Robert E. Bradley, Lawrence A. D'Antonio, and Charles Edward Sandifer, editors. *Euler at 300: an Appreciation*, volume 5 of *The MAA tercentenary Euler celebration; Spectrum series*. Mathematical Association of America, Washington, DC, USA, 2007. ISBN 0-88385-565-8; 978-0-88385-565-2. xvi + 298 pp. LCCN QA29.E8 E95 2007. {**591**}

[BE92] Lee J. Bain and Max Engelhardt. *Introduction to Probability and Mathematical Statistics*. The Duxbury advanced series in statistics and decision sciences. PWS-Kent Publishers, Boston, MA, USA, second edition, 1992. ISBN 0-534-92930-3, 0-534-98563-7 (international student edition); 978-0-534-92930-5, 978-0-534-98563-9 (international student edition). xii + 644 pp. LCCN QA273 .B2546 1991. {**196**}

[Bec73] Petr Beckmann. *Orthogonal Polynomials for Engineers and Physicists*. Golem Press, Boulder, CO, USA, 1973. ISBN 0-911762-14-0; 978-0-911762-14-3. 280 pp. LCCN QA404.5 .B35. {**59**}

[Bec93] Petr Beckmann. *A History of π [pi]*. Barnes and Noble, New York, NY, USA, 1993. ISBN 0-88029-418-3; 978-0-88029-418-8. 200 pp. LCCN QA484 .B4 1971. Reprint of the third edition of 1971. {**14, 59, 623**}

[Bee94] Nelson H. F. Beebe. The impact of memory and architecture on computer performance. Technical report, Department of Mathematics, University of Utah, Salt Lake City, UT, USA, February 23, 1994. viii + 62 pp. URL `http://www.math.utah.edu/~beebe/memperf.pdf`. Supplemental class notes prepared for Mathematics 118 and 119. {**952**}

[Bee04a] Nelson H. F. Beebe. 25 years of TeX and METAFONT: Looking back and looking forward: TUG 2003 keynote address. *TUGboat*, 25(1): 7–30, 2004. ISSN 0896-3207. URL `http://www.math.utah.edu/~beebe/talks/tug2003/`; `http://www.tug.org/TUGboat/tb25-1/beebe-2003keynote.pdf`. {**954**}

[Bee04b] Nelson H. F. Beebe. Java programming: Fun with Fibonacci. World-Wide Web document, March 2004. URL `http://www.math.utah.edu/~beebe/software/java/fibonacci/`. This report summarizes the origin of the Fibonacci sequence, giving the full Latin text from the original book written in 1202 (not previously available on the Web). Computation of the Fibonacci sequence, and its term ratios, is implemented in about 50 different programming languages. The report comments on the relative difficulty of the task in some of those languages, and on their suitability for numerical computation. It also provides a complete floating-point formatted output package for Java. {**15, 73, 978**}

[Bee05] Nelson H. F. Beebe. Keynote address: The design of TeX and METAFONT: A retrospective. *TUGboat*, 26(1):33–51, 2005. ISSN 0896-3207. URL `http://www.tug.org/TUGboat/tb26-1/beebe.pdf`. Proceedings of the Practical TeX 2005 conference, Chapel Hill, NC, June 14–17, 2005. {**954**}

[BEJ76] J. M. Blair, C. A. Edwards, and J. H. Johnson. Rational Chebyshev approximations for the inverse of the error function. *Mathematics of Computation*, 30(136):827–830, October 1976. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL `http://www.jstor.org/stable/2005402`. DOI `10.2307/2005402`. {**600**}

[Bel37] Eric Temple Bell. *Men of mathematics: The Lives and Achievements of the Great Mathematicians from Zeno to Poincaré*. Simon and Schuster, New York, NY, USA, 1937. ISBN 0-671-62818-6; 978-0-671-62818-5. xxi + 592 pp. LCCN QA28 .B4. {**59**}

[Bel68] W. W. (William Wallace) Bell. *Special Functions for Scientists and Engineers*. Van Nostrand, London, UK, 1968. xiv + 247 pp. LCCN QA351 .B4. {**827**}

[Bel04] W. W. (William Wallace) Bell. *Special Functions for Scientists and Engineers*. Dover books on mathematics. Dover, New York, NY, USA, 2004. ISBN 0-486-43521-0; 978-0-486-43521-3. xiv + 247 pp. LCCN QA351 .B4 2004. {**827**}

[Ber68] A. Bergson. Certification of and remark on Algorithm 304 [S15]: Normal curve integral. *Communications of the Association for Computing Machinery*, 11(4):271, April 1968. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/362991.363048`. See [HJ67a, HJ67b]. {**618, 1015**}

[BF71] Paul F. Byrd and Morris D. Friedman. *Handbook of Elliptic Integrals for Engineers and Scientists*, volume 67 of *Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., second edition, 1971. ISBN 0-387-05318-2 (New York); 978-0-387-05318-9 (New York). xvi + 358 pp. LCCN QA343 .B95 1971. DOI `10.1007/978-3-642-65138-0`. {**58, 619, 630, 659, 664, 666, 682, 683, 686**}

[BFN94] Paul Bratley, Bennett L. Fox, and Harald Niederreiter. Algorithm 738: Programs to generate Niederreiter's low-discrepancy sequences. *ACM Transactions on Mathematical Software*, 20(4):494–495, December 1994. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.acm.org/pubs/citations/journals/toms/1994-20-4/p494-bratley/`. DOI `10.1145/198429.198436`. {**203**}

[BFSG74] A. R. Barnett, D. H. Feng, J. W. Steed, and L. J. B. Goldfarb. Coulomb wave functions for all real η [eta] and ρ [rho]. *Computer Physics Communications*, 8(5):377–395, December 1974. CODEN CPHCBZ. ISSN 0010-4655 (print), 1879-2944 (electronic). DOI `10.1016/0010-4655(74)90013-7`. {**17**}

[BGA90] Walter S. Brainerd, Charles H. Goldberg, and Jeanne C. Adams. *Programmer's Guide to Fortran 90*. McGraw-Hill, New York, NY, USA, 1990. ISBN 0-07-000248-7; 978-0-07-000248-7. vii + 410 pp. LCCN QA76.73.F25 B735 1990. {**106**}

[BGM96] George A. Baker, Jr. and Peter Graves-Morris. *Padé Approximants*, volume 59 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, UK, second edition, 1996. ISBN 0-521-45007-1 (hardcover); 978-0-521-45007-2 (hardcover). xiv + 746 pp. LCCN QC20.7.P3 B35 1996. {**589**}

[BGVHN99] Adhemar Bultheel, Pablo Gonzales-Vera, Erik Hendriksen, and Olav Njastad, editors. *Orthogonal Rational Functions*, volume 5 of *Cambridge Monographs on Applied and Computational Mathematics*. Cambridge University Press, Cambridge, UK, 1999. ISBN 0-521-65006-2 (hardcover); 978-0-521-65006-9 (hardcover). xiv + 407 pp. LCCN QA404.5 .O75 1999. {**59**}

[BH07] Nicolas Brisebarre and Guillaume Hanrot. Floating-point L^2-approximations to functions. In Kornerup and Muller [KM07], pages 177–186. ISBN 0-7695-2854-6; 978-0-7695-2854-0. ISSN 1063-6889. LCCN QA76.9.C62. URL `http://www.lirmm.fr/arith18/`. DOI `10.1109/ARITH.2007.38`. {**42, 89**}

[BHK$^+$84] D. E. Bodenstab, Thomas F. Houghton, Keith A. Kelleman, George Ronkin, and Edward P. Schan. UNIX operating system porting experiences. *AT&T Bell Laboratories Technical Journal*, 63(8 part 2):1769–1790, October 1984. CODEN ABLJER. ISSN 0748-612X (print), 2376-7162 (electronic). DOI `10.1002/j.1538-7305.1984.tb00064.x`. {**74, 972**}

[BHY80] Richard P. Brent, Judith A. Hooper, and J. Michael Yohe. An AUGMENT interface for Brent's multiple precision arithmetic package. *ACM Transactions on Mathematical Software*, 6(2):146–149, June 1980. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/355887.355889`. See [Bre78b, Bre79, Smi98]. {**1001, 1032**}

[BK15] David Biancolin and Jack Koenig. Hardware accelerator for exact dot product. Report, ASPIRE Laboratory, University of California, Berkeley, Berkeley, CA, USA, June 19, 2015. {**385**}

[BKMM07] J. Baik, T. Kriecherbauer, K. T.-R. McLaughlin, and P. D. Miller, editors. *Discrete Orthogonal Polynomials: Asymptotics and Applications*, volume 164 of *Annals of mathematics studies*. Princeton University Press, Princeton, NJ, USA, 2007. ISBN 0-691-12733-6 (hardcover), 0-691-12734-4 (paperback); 978-0-691-12733-0 (hardcover), 978-0-691-12734-7 (paperback). vi + 170 pp. LCCN QA404.5 .D57 2007. URL `http://press.princeton.edu/titles/8450.html`. {**59**}

[Bla64] G. Blanch. Numerical evaluation of continued fractions. *SIAM Review*, 6(4):383–421, October 1964. CODEN SIREAD. ISSN 0036-1445 (print), 1095-7200 (electronic). DOI `10.1137/1006092`. {**13**}

[Bla97] David Blatner. *The Joy of π [pi]*. Walker and Co., New York, NY, USA, 1997. ISBN 0-8027-1332-7 (hardcover), 0-8027-7562-4 (paperback); 978-0-8027-1332-2 (hardcover), 978-0-8027-7562-7 (paperback). xiii + 129 pp. LCCN QA484 .B55 1997. URL `http://www.walkerbooks.com/books/catalog.php?key=4`. {**14, 59, 623**}

[Blu78] James L. Blue. A portable Fortran program to find the Euclidean norm of a vector. *ACM Transactions on Mathematical Software*, 4(1): 15–23, March 1978. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/355769.355771`. {**223**}

[BM58] G. E. P. Box and Mervin E. Muller. A note on the generation of random normal deviates. *Annals of Mathematical Statistics*, 29(2): 610–611, June 1958. CODEN AASTAD. ISSN 0003-4851. URL `http://projecteuclid.org/euclid.aoms/1177706645; http://www.jstor.org/stable/2237361`. DOI `10.1214/aoms/1177706645`. {**193**}

[BM04] Sylvie Boldo and Guillaume Melquiond. When double rounding is odd. Research Report RR2004-48, École Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France, November 2004. 2 + 7 pp. URL `http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2004/RR2004-48.pdf`. {**403**}

[BM05] Sylvie Boldo and Jean-Michel Muller. Some functions computable with a fused-mac. In Montuschi and Schwarz [MS05], pages 52–58. ISBN 0-7695-2366-8; 978-0-7695-2366-8. LCCN QA76.9.C62 .S95 2005. URL `http://arith17.polito.it/final/paper-106.pdf`. DOI `10.1109/ARITH.2005.39`. {**397, 406**}

[BM08] Sylvie Boldo and Guillaume Melquiond. Emulation of a FMA and correctly rounded sums: Proved algorithms using rounding to odd. *IEEE Transactions on Computers*, 57(4):462–471, April 2008. CODEN ITCOB4. ISSN 0018-9340 (print), 1557-9956 (electronic). DOI `10.1109/TC.2007.70819`. {**403**}

[BM11] Sylvie Boldo and Jean-Michel Muller. Exact and approximated error of the FMA. *IEEE Transactions on Computers*, 60(2):157–164, February 2011. CODEN ITCOB4. ISSN 0018-9340 (print), 1557-9956 (electronic). DOI `10.1109/TC.2010.139`. {**397, 403, 406**}

[BMM78] C. Gordon Bell, J. Craig Mudge, and John E. McNamara, editors. *Computer Engineering: A DEC View of Hardware Systems Design*. Digital Press, Bedford, MA, USA, 1978. ISBN 0-932376-00-2; 978-0-932376-00-8. xxii + 585 pp. LCCN TK7885 .C64. URL `http://www.bitsavers.org/pdf/dec/Bell-ComputerEngineering.pdf`. {**954**}

[BMY07] N. N. Bogolyubov, G. K. Mikhaĭlov, and A. P. Yushkevich, editors. *Euler and Modern Science*, volume 4 of *MAA tercentenary Euler celebration: Spectrum series*. Mathematical Association of America, Washington, DC, USA, English edition, 2007. ISBN 0-88385-564-X; 978-0-88385-564-5. xiv + 425 pp. LCCN Q143.E84 R3913 2007. Translated by Robert Burns from the 1988 Russian original, *Razvitie ideĭ Leonarda Eulera i sovremennaya nauka*. {**591**}

[Bol09] Sylvie Boldo. Kahan's algorithm for a correct discriminant computation at last formally proven. *IEEE Transactions on Computers*, 58(2): 220–225, February 2009. CODEN ITCOB4. ISSN 0018-9340 (print), 1557-9956 (electronic). DOI `10.1109/TC.2008.200`. See [Kah04b] for the original algorithm. {**472, 1018**}

[Boy89] Joan Boyar. Inferring sequences produced by pseudo-random number generators. *Journal of the Association for Computing Machinery*, 36(1):129–141, January 1989. CODEN JACOAH. ISSN 0004-5411 (print), 1557-735x (electronic). DOI `10.1145/58562.59305`. {**207**}

[BPZ07] Richard P. Brent, Colin Percival, and Paul Zimmermann. Error bounds on complex floating-point multiplication. *Mathematics of Computation*, 76(259):1469–1481, July 2007. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL `http://www.ams.org/mcom/2007-76-259/S0025-5718-07-01931-X/`. DOI `10.1090/S0025-5718-07-01931-X`. {**458, 476**}

[BR91] Claude Brezinski and Michela Redivo Zaglia. *Extrapolation Methods: Theory and Practice*, volume 2 of *Studies in Computational Mathematics*. North-Holland, Amsterdam, The Netherlands, 1991. ISBN 0-444-88814-4; 978-0-444-88814-3. ix + 464 pp. LCCN QA281 .B74 1991. {**589**}

[Bre76a] Richard P. Brent. Analysis of the binary Euclidean algorithm. In J. F. Traub, editor, *Algorithms and Complexity: Recent Results and New Directions: [Proceedings of a Symposium on New Directions and Recent Results in Algorithms and Complexity held by the Computer Science Department, Carnegie-Mellon University, April 7–9, 1976]*, pages 321–355. Academic Press, New York, NY, USA, 1976. ISBN 0-12-697540-X; 978-0-12-697540-6. LCCN QA76.6 .S9195 1976. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.122.7959`. The complexity of the binary Euclidean algorithm for the greatest common denominator is shown to be $O(0.705 \lg N)$ for large $N = \max(|u|, |v|)$. See [Bre00] for an update, and a repair to an incorrect conjecture in this paper. See also [Bre99], where the worst case complexity is shown to be $O(\lg N)$, and the number of right shifts at most $2 \lg(N)$. {**184, 1001**}

[Bre76b] Richard P. Brent. Fast multiple-precision evaluation of elementary functions. *Journal of the Association for Computing Machinery*, 23(2): 242–251, April 1976. CODEN JACOAH. ISSN 0004-5411 (print), 1557-735x (electronic). DOI `10.1145/321941.321944`. {**623**}

[Bre78a] Richard P. Brent. A Fortran multiple-precision arithmetic package. *ACM Transactions on Mathematical Software*, 4(1):57–70, March 1978. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/355769.355775`. {**28**}

[Bre78b] Richard P. Brent. Algorithm 524: MP, A Fortran multiple-precision arithmetic package [A1]. *ACM Transactions on Mathematical Software*, 4(1):71–81, March 1978. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/355769.355776`. See also [Bre79, BHY80, Smi98]. {**28, 1000, 1001, 1032**}

[Bre79] Richard P. Brent. Remark on "Algorithm 524: MP, A Fortran multiple-precision arithmetic package [A1]". *ACM Transactions on Mathematical Software*, 5(4):518–519, December 1979. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/355853.355868`. See [Bre78b, BHY80, Smi98]. {**1000, 1001, 1032**}

[Bre91] Claude Brezinski. *History of Continued Fractions and Padé Approximants*, volume 12 of *Springer series in computational mathematics*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1991. ISBN 3-540-15286-5 (Berlin), 0-387-15286-5 (New York); 978-3-540-15286-6 (Berlin), 978-0-387-15286-8 (New York). 551 pp. LCCN QA295 .B79 1990. DOI `10.1007/978-3-642-58169-4`. {**19, 59**}

[Bre99] Richard P. Brent. Further analysis of the binary Euclidean algorithm. Technical Report TR-7-99, Programming Research Group, Oxford University, Oxford, UK, November 4, 1999. 18 pp. URL `http://arxiv.org/pdf/1303.2772.pdf`. See also earlier work [Bre76a]. {**184, 1000**}

[Bre00] Richard P. Brent. Twenty years' analysis of the binary Euclidean algorithm. In Jim Davies, A. W. Roscoe, and Jim Woodcock, editors, *Millennial perspectives in computer science: proceedings of the 1999 Oxford–Microsoft Symposium in honour of Professor Sir Antony Hoare*, pages 41–52. Palgrave, Basingstoke, UK, 2000. ISBN 0-333-92230-1; 978-0-333-92230-9. LCCN QA75.5 .O8 2000. URL `http://www.cs.ox.ac.uk/people/richard.brent/pd/rpb183pr.pdf`. {**184, 1000**}

[Bre04] Richard P. Brent. Note on Marsaglia's xorshift random number generators. *Journal of Statistical Software*, 11(5):1–5, 2004. CODEN JSSOBK. ISSN 1548-7660. URL `http://www.jstatsoft.org/counter.php?id=101&url=v11/i05/v11i05.pdf&ct=1`. DOI `10.18637/jss.v011.i05`. See [Mar03b, PL05, Vig16]. This article shows the equivalence of xorshift generators and the well-understood linear feedback shift register generators. {**1024, 1028**}

[Bro73] R. Broucke. ACM Algorithm 446: Ten subroutines for the manipulation of Chebyshev series [C1]. *Communications of the Association for Computing Machinery*, 16(4):254–256, April 1973. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/362003.362037`. See remark and certification [PM75]. {**48, 1028**}

[Bro82] Frederick P. Brooks, Jr. *The Mythical Man-Month — Essays on Software Engineering*. Addison-Wesley, Reading, MA, USA, 1982. ISBN 0-201-00650-2; 978-0-201-00650-6. xi + 195 pp. LCCN QA 76.6 B75 1982. {**963**}

[Bro95] Frederick P. Brooks, Jr. *The Mythical Man-Month — Essays on Software Engineering*. Addison-Wesley, Reading, MA, USA, anniversary edition, 1995. ISBN 0-201-83595-9; 978-0-201-83595-3. xiii + 322 pp. LCCN QA76.758 .B75 1995. {**963**}

[Bry08] Yury Aleksandrovich Brychkov. *Handbook of Special Functions: Derivatives, Integrals, Series and other Formulas*. CRC Press, Boca Raton, FL, USA, 2008. ISBN 1-58488-956-X; 978-1-58488-956-4. xix + 680 pp. LCCN QA351 .B79 2008. {**58, 521, 556, 827**}

[BS77] J. D. Beasley and S. G. Springer. Statistical algorithms: Algorithm AS 111: The percentage points of the normal distribution. *Applied Statistics*, 26(1):118–121, March 1977. CODEN APSTAG. ISSN 0035-9254 (print), 1467-9876 (electronic). URL `http://lib.stat.cmu.edu/apstat/111`. DOI `10.2307/2346889`. {**618**}

[BS80] Jon Louis Bentley and James B. Saxe. Generating sorted lists of random numbers. *ACM Transactions on Mathematical Software*, 6(3): 359–364, September 1980. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/355900.355907`. {**168**}

[BS92] Ronald F. Boisvert and Bonita V. Saunders. Portable vectorized software for Bessel function evaluation. *ACM Transactions on Mathematical Software*, 18(4):456–469, December 1992. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.acm.org/pubs/citations/journals/toms/1992-18-4/p456-boisvert/`. DOI `10.1145/138351.138370`. See correction [BS93]. {**693, 823, 1001**}

[BS93] Ronald F. Boisvert and Bonita V. Saunders. Corrigendum: "Algorithm 713: Portable vectorized software for Bessel function evaluation". *ACM Transactions on Mathematical Software*, 19(1):131, March 1993. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). See [BS92]. {**823, 1001**}

[BS03] Jean Claude Bajard and Michael J. Schulte, editors. *16th IEEE Symposium on Computer Arithmetic: ARITH-16 2003: proceedings: Santiago de Compostela, Spain, June 15–18, 2003*. IEEE Computer Society Press, Silver Spring, MD, USA, 2003. ISBN 0-7695-1894-X; 978-0-7695-1894-7. ISSN 1063-6889. LCCN QA76.6 .S919 2003. URL `http://www.dec.usc.es/arith16/`. {**998, 1005, 1032**}

[BS07] Robert E. Bradley and Charles Edward Sandifer, editors. *Leonhard Euler: Life, Work, and Legacy*, volume 5 of *Studies in the History and Philosophy of Mathematics*. Elsevier, Amsterdam, The Netherlands, 2007. ISBN 0-444-52728-1; 978-0-444-52728-8. viii + 534 pp. LCCN QA29.E8 L465 2007. URL `http://www.sciencedirect.com/science/book/9780444527288`. {**591**}

[BS12] Michael Baudin and Robert L. Smith. A robust complex division in Scilab. *CoRR*, abs/1210.4539, 2012. URL `http://arxiv.org/abs/1210.4539`. {**463**}

[BSI03a] *The C Standard: Incorporating Technical Corrigendum 1*. Wiley, New York, NY, USA, 2003. ISBN 0-470-84573-2; 978-0-470-84573-8. 538 pp. LCCN QA76.73.C15C185 2003. BS ISO/IEC 9899:1999. {**4**}

[BSI03b] *The C++ Standard: Incorporating Technical Corrigendum 1: BS ISO*. Wiley, New York, NY, USA, second edition, 2003. ISBN 0-470-84674-7; 978-0-470-84674-2. xxxiv + 782 pp. LCCN QA76.73.C153C16 2003. {**vii**}

[Buc62] Werner Buchholz, editor. *Planning a computer system: Project Stretch*. McGraw-Hill, New York, NY, USA, 1962. xvii + 322 pp. LCCN 1876. URL `http://ed-thelen.org/comp-hist/IBM-7030-Planning-McJones.pdf`. This important book is the primary description of the influential IBM 7030 Stretch computer, written by its architects. {**959, 998**}

[Bur07] David M. Burton. *The History of Mathematics: An Introduction*. McGraw-Hill, New York, NY, USA, sixth edition, 2007. ISBN 0-07-305189-6; 978-0-07-305189-5. xii + 788 pp. LCCN QA21 .B96 2007. {**59**}

[BW10] Richard Beals and R. (Roderick) Wong. *Special Functions: a Graduate Text*, volume 126 of *Cambridge studies in advanced mathematics*. Cambridge University Press, Cambridge, UK, 2010. ISBN 0-521-19797-X; 978-0-521-19797-7. ix + 456 pp. LCCN QA351 .B34 2010; QA351 BEA 2010. {**827**}

[BWKM91] Gerd Bohlender, W. Walter, Peter Kornerup, and David W. Matula. Semantics for exact floating point operations. In Kornerup and Matula [KM91], pages 22–26. ISBN 0-8186-9151-4 (case), 0-8186-6151-8 (microfiche), 0-7803-0187-0 (library binding); 978-0-8186-9151-5 (case), 978-0-8186-6151-8 (microfiche), 978-0-7803-0187-0 (library binding). LCCN QA76.9.C62 S95 1991. DOI `10.1109/ARITH.1991.145529`. See [BD03a] for some special cases that this paper may have overlooked. {**366**}

[BZ11] Richard P. Brent and Paul Zimmermann. *Modern Computer Arithmetic*, volume 18 of *Cambridge monographs on applied and computational mathematics*. Cambridge University Press, Cambridge, UK, 2011. ISBN 0-521-19469-5 (hardcover); 978-0-521-19469-3 (hardcover). xvi + 221 pp. LCCN QA76.9.C62 BRE 2011. URL `http://www.loria.fr/~zimmerma/mca/pub226.html`. {**104, 407, 574, 978**}

[C90] *ISO/IEC 9899:1990: Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, 1990. URL `http://www.iso.ch/cate/d17782.html`. {**vii, 1, 4, 827**}

[C++98] *ISO/IEC 14882:1998: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, September 1, 1998. 732 pp. URL `http://www.iso.ch/cate/d25845.html`. Available in electronic form for online purchase at `http://webstore.ansi.org/` and `http://www.cssinfo.com/`. {**vii, 1, 57, 106**}

[C99] *ISO/IEC 9899:1999: Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland, December 16, 1999. 538 pp. URL `http://anubis.dkuug.dk/JTC1/SC22/open/n2620/n2620.pdf`; `http://anubis.dkuug.dk/JTC1/SC22/WG14/www/docs/n897.pdf`; `http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+9899%3A1999`; `http://www.iso.ch/cate/d29237.html`. Available in electronic form for online purchase at `http://webstore.ansi.org/` and `http://www.cssinfo.com/`. {**vii, 1, 4, 106, 441, 449, 455, 456, 460, 482, 490, 496, 507, 513, 514, 518, 525, 534**}

[C++03a] *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, 2003. 757 pp. URL `http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38110`. {**vii, 1, 57, 106**}

[C#03b] *ISO/IEC 23270:2003: Information technology — C# Language Specification*. International Organization for Standardization, Geneva, Switzerland, 2003. xiii + 471 pp. URL `http://standards.iso.org/ittf/PubliclyAvailableStandards/c036768_ISO_IEC_23270_2003(E).zip`. {**80, 917**}

[C#06a] *ISO/IEC 23270:2006: Information technology — Programming languages — C#*. Technical report. International Organization for Standardization, Geneva, Switzerland, 2006. URL `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=42926`. {**vii, 917**}

[C06b] ISO/IEC JTC1 SC22 WG14 N1154: Extension for the programming language C to support decimal floating-point arithmetic. World-Wide Web document, February 27, 2006. URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1154.pdf`. {**875, 928**}

[C06c] ISO/IEC JTC1 SC22 WG14 N1161: Rationale for TR 24732: Extension to the programming language C: Decimal floating-point arithmetic. World-Wide Web document, February 27, 2006. URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1161.pdf`. {**875, 928**}

[C06d] ISO/IEC JTC1 SC22 WG14 N1176: Extension for the programming language C to support decimal floating-point arithmetic. World-Wide Web document, May 24, 2006. URL `http://open-std.org/jtc1/sc22/wg14/www/docs/n1176.pdf`. {**875, 928**}

[C09] *ISO/IEC TR 24732:2009 Information technology — Programming languages, their environments and system software interfaces — Extension for the programming language C to support decimal floating-point arithmetic*. Technical report. International Organization for Standardization, Geneva, Switzerland, 2009. URL `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=38842`. {**928**}

[C++10] *ISO/IEC 29124:2010: Information technology — Programming languages, their environments and system software interfaces — Extensions to the C++ Library to support mathematical special functions*. Technical report. International Organization for Standardization, Geneva, Switzerland, 2010. URL `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50511`. {**57**}

[C++11a] *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, third edition, September 1, 2011. URL `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372`. {**vii, 1**}

[C11b] *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 8, 2011. 683 (est.) pp. URL `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853`. {**vii, 1**}

[C++14] *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, fourth edition, December 15, 2014. URL `http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=64029`. {**1**}

[CAH^{+}07] Marius Cornea, Cristina Anderson, John Harrison, Ping Tak Peter Tang, Eric Schneider, and Charles Tsen. A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format. In Kornerup and Muller [KM07], pages 29–37. ISBN 0-7695-2854-6; 978-0-7695-2854-0. ISSN 1063-6889. LCCN QA76.9.C62. URL `http://www.lirmm.fr/arith18/papers/CorneaM_Decimal_ARITH18.pdf`. DOI `10.1109/ARITH.2007.7`. {**928**}

[Cai11] Liang-Wu Cai. On the computation of spherical Bessel functions of complex arguments. *Computer Physics Communications*, 182(3):663–668, March 2011. CODEN CPHCBZ. ISSN 0010-4655 (print), 1879-2944 (electronic). URL http://www.sciencedirect.com/science/article/pii/S0010465510004650. DOI 10.1016/j.cpc.2010.11.019. {**693**}

[Caj91] Florian Cajori. *A History of Mathematics*. Chelsea Publishing Company, New York, NY, USA, fifth edition, 1991. ISBN 0-8284-2303-6; 978-0-8284-2303-8. xi + 524 pp. LCCN QA21 .C15 1991. {**59**}

[Cal95] Ronald Calinger, editor. *Classics of Mathematics*. Prentice-Hall, Upper Saddle River, NJ, USA, 1995. ISBN 0-02-318342-X; 978-0-02-318342-3. xxi + 793 pp. LCCN QA21 .C55 1995. {**7**}

[Cam80] J. B. Campbell. On Temme's algorithm for the modified Bessel function of the third kind. *ACM Transactions on Mathematical Software*, 6(4):581–586, December 1980. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI 10.1145/355921.355928. {**693**}

[Car63] L. Carlitz. The inverse of the error function. *Pacific Journal of Mathematics*, 13(2):459–470, 1963. CODEN PJMAAI. ISSN 0030-8730 (print), 1945-5844 (electronic). URL http://projecteuclid.org/euclid.pjm/1103035736. {**600**}

[Car70] Bille Chandler Carlson. Hidden symmetries of special functions. *SIAM Review*, 12(3):332–345, July 1970. CODEN SIREAD. ISSN 0036-1445 (print), 1095-7200 (electronic). URL http://www.jstor.org/stable/2028552. DOI 10.1137/1012078. {**646**}

[Car71] Bille Chandler Carlson. Algorithms involving arithmetic and geometric means. *American Mathematical Monthly*, 78(5):496–505, May 1971. CODEN AMMYAE. ISSN 0002-9890 (print), 1930-0972 (electronic). URL http://www.jstor.org/stable/2317754. DOI 10.2307/2317754. {**623**}

[Car77] Bille Chandler Carlson. *Special Functions of Applied Mathematics*. Academic Press, New York, NY, USA, 1977. ISBN 0-12-160150-1; 978-0-12-160150-8. xv + 335 pp. LCCN QA351 .C32. {**644–646, 653, 682, 827**}

[Car79] Bille Chandler Carlson. Computing elliptic integrals by duplication. *Numerische Mathematik*, 33(1):1–16, March 1979. CODEN NUMMA7. ISSN 0029-599X (print), 0945-3245 (electronic). DOI 10.1007/BF01396491. {**646, 649, 651**}

[Car95] Bille Chandler Carlson. Numerical computation of real or complex elliptic integrals. *Numerical Algorithms*, 10(1–2):13–26, July 1995. CODEN NUALEG. ISSN 1017-1398 (print), 1572-9265 (electronic). DOI 10.1007/BF02198293. Special functions (Torino, 1993). {**646, 648**}

[Cat03] Don Catlin. *The Lottery Book: The Truth behind the Numbers*. Bonus Books, Chicago, IL, USA, 2003. ISBN 1-56625-193-1; 978-1-56625-193-8. xvii + 181 pp. LCCN HG6126 .C38 2003. This book describes US lotteries, and how their odds and payouts are determined. {**297**}

[CBB$^+$99] Jean-Luc Chabert, E. Barbin, Jacques Borowczyk, Michel Guillemot, Anne Michel-Pajus, Ahmed Djebbar, and Jean-Claude Martzloff, editors. *Histoire d'Algorithmes. A History of Algorithms: from the Pebble to the Microchip*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1999. ISBN 3-540-63369-3 (softcover); 978-3-540-63369-3 (softcover). ix + 524 pp. LCCN QA58 .H5813 1998. DOI 10.1007/978-3-642-18192-4. Translated by Chris Weeks from the 1994 French original, *Histoire d'algorithmes. Du caillou à la puce*. {**59**}

[CBGK13] Mathew A. Cleveland, Thomas A. Brunner, Nicholas A. Gentile, and Jeffrey A. Keasler. Obtaining identical results with double precision global accuracy on different numbers of processors in parallel particle Monte Carlo simulations. *Journal of Computational Physics*, 251:223–236, October 15, 2013. CODEN JCTPAH. ISSN 0021-9991 (print), 1090-2716 (electronic). URL http://www.sciencedirect.com/science/article/pii/S0021999113004075. DOI 10.1016/j.jcp.2013.05.041. {**385**}

[CCG$^+$84] William J. Cody, Jr., Jerome T. Coonen, David M. Gay, K. Hanson, David G. Hough, William M. Kahan, Richard Karpinski, John F. Palmer, Frederic N. Ris, and David Stevenson. A proposed radix- and word-length-independent standard for floating-point arithmetic. *IEEE Micro*, 4(4):86–100, August 1984. CODEN IEMIDZ. ISSN 0272-1732 (print), 1937-4143 (electronic). DOI 10.1109/MM.1984.291224. {**104, 152**}

[CDGI15] Sylvain Collange, David Defour, Stef Graillat, and Roman Iakymchuk. Numerical reproducibility for the parallel reduction on multi- and many-core architectures. *Parallel Computing*, 49:83–97, November 2015. CODEN PACOEJ. ISSN 0167-8191 (print), 1872-7336 (electronic). URL http://www.sciencedirect.com/science/article/pii/S0167819115001155. DOI 10.1016/j.parco.2015.09.001. {**385**}

[CGL90] R. Coquereaux, A. Grossmann, and B. E. Lautrup. Iterative method for calculation of the Weierstrass elliptic function. *IMA Journal of Numerical Analysis*, 10(1):119–128, January 1990. CODEN IJNADH. ISSN 0272-4979 (print), 1464-3642 (electronic). DOI 10.1093/imanum/10.1.119. {**689**}

[CH67] William J. Cody, Jr. and K. E. Hillstrom. Chebyshev approximations for the natural logarithm of the gamma function. *Mathematics of Computation*, 21(98):198–203, April 1967. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL http://www.jstor.org/stable/2004160. DOI 10.2307/2004160. {**521**}

[CH75] Tien Chi Chen and Irving T. Ho. Storage-efficient representation of decimal data. *Communications of the Association for Computing Machinery*, 18(1):49–52, January 1975. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). URL http://www2.hursley.ibm.com/decimal/chen-ho.html. DOI 10.1145/360569.360660. Collection of articles honoring Alston S. Householder. See comment [Smi75]. {**928, 1032**}

[Che98] Russell C. H. Cheng. Random variate generation. In Banks [Ban98], pages 138–172. ISBN 0-471-13403-1 (hardcover); 978-0-471-13403-9 (hardcover). LCCN T57.62 .H37 1998. DOI 10.1002/9780470172445.ch5. {**196**}

[Chi78] Theodore Seio Chihara. *An Introduction to Orthogonal Polynomials*, volume 13 of *Mathematics and its applications*. Gordon and Breach, New York, NY, USA, 1978. ISBN 0-677-04150-0; 978-0-677-04150-6. xii + 249 pp. LCCN QA404.5 .C44. {**59**}

[CHT02] Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific computing on Itanium-based systems*. Intel Corporation, Santa Clara, CA, USA, 2002. ISBN 0-9712887-7-1; 978-0-9712887-7-5. xvii + 406 pp. LCCN QA76.8.I83 C67 2002. URL http://www.intel.com/intelpress/sum_scientific.htm. {**299**}

[CJW06] James A. Carlson, Arthur Jaffe, and Andrew Wiles, editors. *The Millennium Prize Problems*. American Mathematical Society, Providence, RI, USA, 2006. ISBN 0-8218-3679-X; 978-0-8218-3679-8. viii + 165 pp. LCCN QA43 .M493 2006. URL `http://www.claymath.org/publications/Millennium_Problems/`. {**60, 303, 521, 579, 590**}

[CKCFR03] Martin Campbell-Kelly, Mary Croarken, Raymond Flood, and Eleanor Robson, editors. *The History of Mathematical Tables: From Sumer to Spreadsheets*. Oxford University Press, Oxford, UK, 2003. ISBN 0-19-850841-7; 978-0-19-850841-0. viii + 361 pp. LCCN QA47 .H57 2003. {**59**}

[CKP+79] Jerome T. Coonen, William M. Kahan, John F. Palmer, Tom Pittman, and David Stevenson. A proposed standard for binary floating point arithmetic: Draft 5.11. *ACM SIGNUM Newsletter*, 14(3S):4–12, October 1979. CODEN SNEWD6. ISSN 0163-5778 (print), 1558-0237 (electronic). DOI `10.1145/1057520.1057521`. {**104**}

[CKT07] Kalyan Chakraborty, Shigeru Kanemitsu, and Haruo Tsukada. *Vistas of Special Functions II*. World Scientific Publishing, Singapore, 2007. ISBN 981-270-774-3; 978-981-270-774-1. xii + 215 pp. LCCN QA351 .K35 2007. {**827**}

[Clay09] Clay Mathematics Institute. Web site., 2009. URL `http://www.claymath.org/`. This institute sponsors research in advanced mathematics, and offers large monetary prizes for solutions of selected famous unsolved problems in mathematics. {**303, 521, 579, 590**}

[Cle03] Brian Clegg. *A Brief History of Infinity: The Quest to Think the Unthinkable*. Constable and Robinson, London, UK, 2003. ISBN 1-84119-650-9; 978-1-84119-650-3. 255 pp. LCCN BD411. {**59**}

[Cli90] William D. Clinger. How to read floating point numbers accurately. *ACM SIGPLAN Notices*, 25(6):92–101, June 1990. CODEN SINODQ. ISBN 0-89791-364-7; 978-0-89791-364-5. ISSN 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). URL `http://www.acm.org:80/pubs/citations/proceedings/pldi/93542/p92-clinger/`. DOI `10.1145/93548.93557`. See also output algorithms in [Knu90, SW90, BD96, ABC+99, SW04] and retrospective [Cli04]. {**895, 896, 995, 998, 1004, 1020, 1034**}

[CLI03] *ISO/IEC 23271:2003: Information technology — Common Language Infrastructure*. International Organization for Standardization, Geneva, Switzerland, 2003. xi + 99 (Part I), ix + 164 (Part II), vi + 125 (Part III), iii + 16 (Part IV), iv + 79 (Part V) pp. URL `http://standards.iso.org/ittf/PubliclyAvailableStandards/c036769_ISO_IEC_23271_2003(E).zip`; `http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=36769`. {**917**}

[Cli04] William D. Clinger. Retrospective: How to read floating point numbers accurately. *ACM SIGPLAN Notices*, 39(4):360–371, April 2004. CODEN SINODQ. ISSN 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI `10.1145/989393.989430`. Best of PLDI 1979–1999. Reprint of, and retrospective on, [Cli90]. {**895, 998, 1004, 1020, 1034**}

[CLI05] *ISO/IEC TR 23272: Information technology — Common Language Infrastructure — Profiles and Libraries*. International Organization for Standardization, Geneva, Switzerland, 2005. 6 pp. URL `http://standards.iso.org/ittf/PubliclyAvailableStandards/c036770_ISO_IEC_TR_23272_2003(E).zip`. {**vii, 80, 917**}

[CLI06] *ISO/IEC 23271:2006: Information technology: Common Language Infrastructure (CLI) Partitions I to VI*. International standard. International Organization for Standardization, Geneva, Switzerland, second edition, 2006. {**vii, 917**}

[CLK99] Patrick Chan, Rosanna Lee, and Doug Kramer. *The Java Class Libraries:* `java.io`, `java.lang`, `java.math`, `java.net`, `java.text`, `java.util`, volume 1. Addison-Wesley, Reading, MA, USA, second edition, 1999. ISBN 0-201-31002-3; 978-0-201-31002-3. xxvi + 2050 pp. LCCN QA76.73.J38 C47 1998. {**vii**}

[CMF77] William J. Cody, Jr., Rose M. Motley, and L. Wayne Fullerton. The computation of real fractional order Bessel functions of the second kind. *ACM Transactions on Mathematical Software*, 3(3):232–239, September 1977. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/355744.355747`. {**693**}

[CN81] Bille Chandler Carlson and Elaine M. Notis. Algorithm 577: Algorithms for incomplete elliptic integrals [S21]. *ACM Transactions on Mathematical Software*, 7(3):398–403, September 1981. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.netlib.org/toms/577`. DOI `10.1145/355958.355970`. {**646, 648, 650**}

[COB02] *ISO/IEC 1989:2002: Information technology — Programming languages — COBOL*. International Organization for Standardization, Geneva, Switzerland, 2002. 859 pp. URL `http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=28805`. {**928**}

[Cod64] William J. Cody, Jr. Double-precision square root for the CDC-3600. *Communications of the Association for Computing Machinery*, 7(12): 715–718, December 1964. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/355588.365122`. {**949**}

[Cod65a] William J. Cody, Jr. Chebyshev approximations for the complete elliptic integrals *K* and *E*. *Mathematics of Computation*, 19(89–92):105–112, April 1965. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL `http://www.jstor.org/stable/2004103`. DOI `10.2307/2004103`. See corrigenda [Cod66]. {**644, 1004**}

[Cod65b] William J. Cody, Jr. Chebyshev polynomial expansions of complete elliptic integrals. *Mathematics of Computation*, 19(89–92):249–259, April 1965. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL `http://www.jstor.org/stable/2003350`. DOI `10.2307/2003350`. {**644**}

[Cod66] William J. Cody, Jr. Corrigenda: "Chebyshev approximations for the complete elliptic integrals *K* and *E*". *Mathematics of Computation*, 20(93):207, January 1966. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL `http://www.jstor.org/stable/2004329`. DOI `10.2307/2004329`. See [Cod65a]. {**1004**}

[Cod69] William J. Cody, Jr. Rational Chebyshev approximations for the error function. *Mathematics of Computation*, 23(107):631–637, July 1969. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL `http://www.jstor.org/stable/2004390`. DOI `10.2307/2004390`. {**593**}

[Cod80] William J. Cody, Jr. Preliminary report on software for modified Bessel functions of the first kind. Technical Memo AMD TM-357, Argonne National Laboratory, Argonne, IL, USA, 1980. {**693**}

[Cod81] William J. Cody, Jr. Analysis of proposals for the floating-point standard. *Computer*, 14(3):63–69, March 1981. CODEN CPTRB4. ISSN 0018-9162 (print), 1558-0814 (electronic). URL `http://ieeexplore.ieee.org/document/1667286/`. DOI `10.1109/C-M.1981.220379`. See [IEEE85a]. {**63, 1016**}

[Cod83] William J. Cody, Jr. Algorithm 597: Sequence of modified Bessel functions of the first kind. *ACM Transactions on Mathematical Software*, 9(2):242–245, June 1983. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/357456.357462`. {**693**}

[Cod88a] William J. Cody, Jr. Performance evaluation of programs for the error and complementary error functions. Mathematics and Computer Science Preprint MCS-P13-0988, Argonne National Laboratory, Argonne, IL, USA, September 1988. Published in [Cod90]. {**593**}

[Cod88b] William J. Cody, Jr. Performance evaluation of programs related to the real gamma function. Mathematics and Computer Science Preprint MCS-P12-0988, Argonne National Laboratory, Argonne, IL, USA, September 1988. Published in [Cod91]. {**521, 1005**}

[Cod90] William J. Cody, Jr. Performance evaluation of programs for the error and complementary error functions. *ACM Transactions on Mathematical Software*, 16(1):29–37, March 1990. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.acm.org/pubs/toc/Abstracts/0098-3500/77628.html`. DOI `10.1145/77626.77628`. {**593, 1005**}

[Cod91] William J. Cody, Jr. Performance evaluation of programs related to the real gamma function. *ACM Transactions on Mathematical Software*, 17(1):46–54, March 1991. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.acm.org/pubs/toc/Abstracts/0098-3500/103153.html`. DOI `10.1145/103147.103153`. Preprint in [Cod88b]. {**521, 525, 1005**}

[Cod93a] William J. Cody, Jr. Algorithm 714: CELEFUNT: A portable test package for complex elementary functions. *ACM Transactions on Mathematical Software*, 19(1):1–21, March 1993. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/151271.151272`. {**476**}

[Cod93b] William J. Cody, Jr. Algorithm 715: SPECFUN: A portable FORTRAN package of special function routines and test drivers. *ACM Transactions on Mathematical Software*, 19(1):22–32, March 1993. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.acm.org/pubs/toc/Abstracts/0098-3500/151273.html`. DOI `10.1145/151271.151273`. {**521, 525, 593, 693**}

[Coh81] Danny Cohen. On Holy Wars and a plea for peace. *Computer*, 14(10):48–54, October 1981. CODEN CPTRB4. ISSN 0018-9162 (print), 1558-0814 (electronic). DOI `10.1109/C-M.1981.220208`. This is an entertaining account of the Big-Endian and Little-Endian problems of bit and byte ordering. {**956, 963**}

[Con67] Control Data Corporation, St. Paul, MN, USA. *Control Data 6400/6500/6600 Computer Systems Reference Manual*, 1967. vi + 153 pp. URL `http://www.bitsavers.org/pdf/cdc/6x00/60100000D_6600refMan_Feb67.pdf`. {**949**}

[Con71] Control Data Corporation, St. Paul, MN, USA. *Control Data 7600 Computer Systems Reference Manual*, February 1971. v + 194 pp. URL `http://www.bitsavers.org/pdf/cdc/7600/60258200C_7600_RefMan_Feb71.pdf`. {**949**}

[Coo80] Jerome T. Coonen. An implementation guide to a proposed standard for floating-point arithmetic. *Computer*, 13(1):68–79, January 1980. CODEN CPTRB4. ISSN 0018-9162 (print), 1558-0814 (electronic). DOI `10.1109/MC.1980.1653344`. See [IEEE85a], and errata in [Coo81a]. {**63, 104, 1005, 1016**}

[Coo81a] Jerome T. Coonen. Errata: An implementation guide to a proposed standard for floating point arithmetic. *Computer*, 14(3):62, March 1981. CODEN CPTRB4. ISSN 0018-9162 (print), 1558-0814 (electronic). DOI `10.1109/C-M.1981.220378`. See [Coo80, IEEE85a]. {**63, 1005, 1016**}

[Coo81b] Jerome T. Coonen. Underflow and the denormalized numbers. *Computer*, 14(3):75–87, March 1981. CODEN CPTRB4. ISSN 0018-9162 (print), 1558-0814 (electronic). URL `http://ieeexplore.ieee.org/document/1667289/`. DOI `10.1109/C-M.1981.220382`. See [IEEE85a]. {**63, 79, 104, 1016**}

[Coo84] Jerome T. Coonen. *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic*. Thesis (Ph.D. in mathematics), Department of Mathematics, University of California at Berkeley, Berkeley, CA, USA, December 18, 1984. 320 pp. {**856**}

[Cow84] Wayne R. Cowell, editor. *Sources and Development of Mathematical Software*. Prentice-Hall Series in Computational Mathematics, Cleve Moler, Advisor. Prentice-Hall, Upper Saddle River, NJ, USA, 1984. ISBN 0-13-823501-5; 978-0-13-823501-7. xii + 404 pp. LCCN QA76.95 .S68 1984. {**823, 826**}

[Cow85] Michael F. Cowlishaw. *The REXX Language: a Practical Approach to Programming*. Prentice-Hall, Upper Saddle River, NJ, USA, 1985. ISBN 0-13-780735-X (paperback); 978-0-13-780735-2 (paperback). xi + 176 pp. LCCN QA76.73.R24 C69 1985. {**viii, 928, 968**}

[Cow90] Michael F. Cowlishaw. *The REXX Language: a Practical Approach to Programming*. Prentice-Hall, Upper Saddle River, NJ, USA, second edition, 1990. ISBN 0-13-780651-5; 978-0-13-780651-5. xii + 203 pp. LCCN QA76.73.R24 C69 1990. URL `http://vig.prenhall.com/catalog/academic/product/0,1144,0137806515,00.html`. {**viii, 928, 968**}

[Cow97] Michael F. Cowlishaw. *The NetRexx Language*. Prentice-Hall, Upper Saddle River, NJ, USA, 1997. ISBN 0-13-806332-X; 978-0-13-806332-0. viii + 197 pp. LCCN QA76.73.N47 C68 1997. See also supplement [Cow00]. {**viii, 928, 968, 1005**}

[Cow00] Michael F. Cowlishaw. *NetRexx Language Supplement*. IBM UK Laboratories, Hursley Park, Winchester, England, August 23, 2000. iii + 45 pp. URL `http://www-306.ibm.com/software/awdtools/netrexx/nrlsupp.pdf`. Version 2.00. This document is a supplement to [Cow97]. {**968, 1005**}

[Cow02] Michael F. Cowlishaw. Densely packed decimal encoding. *IEE Proceedings. Computers and Digital Techniques*, 149(3):102–104, 2002. CODEN ICDTEA. ISSN 1350-2387 (print), 1359-7027 (electronic). DOI `10.1049/ip-cdt:20020407`. {**928**}

[Cow03] Michael F. Cowlishaw. Decimal floating-point: algorism for computers. In Bajard and Schulte [BS03], pages 104–111. ISBN 0-7695-1894-X; 978-0-7695-1894-7. ISSN 1063-6889. LCCN QA76.6 .S919 2003. URL `http://www.dec.usc.es/arith16/papers/paper-107.pdf`. DOI `10.1109/ARITH.2003.1207666`. {**927**}

[Cow05] Michael F. Cowlishaw. General decimal arithmetic specification. Report Version 1.50, IBM UK Laboratories, Hursley, UK, December 9, 2005. iii + 63 pp. URL `http://www2.hursley.ibm.com/decimal/decarith.pdf`. {**109**}

[Cow07] Michael F. Cowlishaw. *The decNumber C library*. IBM Corporation, San Jose, CA, USA, April 18, 2007. URL http://download.icu-project.org/ex/files/decNumber/decNumber-icu-340.zip. Version 3.40. {**387, 433, 897, 928**}

[CPV$^+$08] Annie Cuyt, Vigdis B. Petersen, Brigitte Verdonk, Haakon Waadeland, and William B. Jones. *Handbook of Continued Fractions for Special Functions*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2008. ISBN 1-4020-6948-0; 978-1-4020-6948-2. xx + 440 pp. LCCN QA295 .H275 2008. DOI 10.1007/978-1-4020-6949-9. {**19, 58, 776, 827**}

[Cray75] Cray Research, Inc., Mendota Heights, MN, USA. *The Cray 1 Computer Preliminary Reference Manual*, June 1975. vi + 75 pp. URL http://www.bitsavers.org/pdf/cray/CRAY-1_PrelimRefRevA_Jun75.pdf. {**949**}

[Cray77] Cray Research, Inc., Minneapolis, MN, USA. *CRAY-1 Hardware Reference Manual*, November 4, 1977. URL http://www.bitsavers.org/pdf/cray/2240004C-1977-Cray1.pdf. Publication number 2240004, revision C. {**953**}

[Cray82] Cray Research, Inc., Mendota Heights, MN, USA. *Cray X-MP Computer Systems Mainframe Reference Manual*, November 1982. x + 206 pp. URL http://www.bitsavers.org/pdf/cray/HR-0032_X-MP_MainframeRef_Nov82.pdf. {**949, 953**}

[CS89] William J. Cody, Jr. and L. Stoltz. Performance evaluation of programs for certain Bessel functions. *ACM Transactions on Mathematical Software*, 15(1):41–48, March 1989. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL http://www.acm.org/pubs/toc/Abstracts/0098-3500/62039.html. DOI 10.1145/62038.62039. Also published as Technical Report MCS-P14-0988, Argonne National Laboratory, Argonne, IL, USA. {**693, 769**}

[CS91] William J. Cody, Jr. and L. Stoltz. The use of Taylor series to test accuracy of function programs. *ACM Transactions on Mathematical Software*, 17(1):55–63, March 1991. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL http://www.acm.org/pubs/toc/Abstracts/0098-3500/103154.html. DOI 10.1145/103147.103154. {**521, 593, 693, 769**}

[CST73] William J. Cody, Jr., Anthony J. Strecok, and Henry C. Thacher, Jr. Chebyshev approximations for the psi function. *Mathematics of Computation*, 27(21):123–127, January 1973. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL http://www.jstor.org/stable/2005253. DOI 10.2307/2005253. {**521**}

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, April 1965. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL http://www.jstor.org/stable/2003354. DOI 10.2307/2003354. {**969**}

[CVZ00] Henri Cohen, Fernando Rodriguez Villegas, and Don Zagier. Convergence acceleration of alternating series. *Experimental Mathematics*, 9(1):3–12, 2000. ISSN 1058-6458 (print), 1944-950x (electronic). URL http://projecteuclid.org/euclid.em/1046889587; http://www.math.u-bordeaux.fr/~cohen/sumalt2new.ps. DOI 10.1080/10586458.2000.10504632. {**589**}

[CW80] William J. Cody, Jr. and William Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Upper Saddle River, NJ, USA, 1980. ISBN 0-13-822064-6; 978-0-13-822064-8. x + 269 pp. LCCN QA331 .C635 1980. {**viii, 1, 270, 344, 411, 763, 823, 939**}

[CW08] D. Cavagnino and A. E. Werbrouck. Efficient algorithms for integer division by constants using multiplication. *The Computer Journal*, 51(4):470–480, July 2008. CODEN CMPJA6. ISSN 0010-4620 (print), 1460-2067 (electronic). URL http://comjnl.oxfordjournals.org/cgi/content/abstract/51/4/470; http://comjnl.oxfordjournals.org/cgi/content/full/51/4/470; http://comjnl.oxfordjournals.org/cgi/reprint/51/4/470. DOI 10.1093/comjnl/bxm082. {**176**}

[CW11] D. Cavagnino and A. E. Werbrouck. An analysis of associated dividends in the DBM algorithm for division by constants using multiplication. *The Computer Journal*, 54(1):148–156, January 2011. CODEN CMPJA6. ISSN 0010-4620 (print), 1460-2067 (electronic). URL http://comjnl.oxfordjournals.org/content/54/1/148.full.pdf+html. DOI 10.1093/comjnl/bxp117. [DBM = Division by Multiplication]. {**176**}

[Cyv64] S. J. Cyvin. Algorithm 226: Normal distribution function. *Communications of the Association for Computing Machinery*, 7(5):295, May 1964. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI 10.1145/364099.364315. See remarks [HJ67b]. {**618, 1015**}

[dDDL04] Florent de Dinechin, David Defour, and Christoph Lauter. Fast correct rounding of elementary functions in double precision using double-extended arithmetic. Research Report RR2004-10, École Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France, March 2004. 2 + 12 pp. URL http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2004/RR2004-10.pdf. {**28**}

[dDG04] Florent de Dinechin and Nicolas Gast. Towards the post-ultimate libm. Research Report RR2004-47, École Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France, November 2004. URL http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2004/RR2004-47.pdf. {**28**}

[DDZ$^+$07] A. Y. Duale, M. H. Decker, H.-G. Zipperer, M. Aharoni, and T. J. Bohizic. Decimal floating-point in z9: An implementation and testing perspective. *IBM Journal of Research and Development*, 51(1/2):217–227, January /March 2007. CODEN IBMJAE. ISSN 0018-8646 (print), 2151-8556 (electronic). URL http://www.research.ibm.com/journal/rd/511/duale.html. DOI 10.1147/rd.511.0217. {**927**}

[DEC76] Digital Equipment Corporation, Maynard, MA, USA. *DECsystem-10/20 Hardware Manual*, fourth edition, November 1976. Publication DEC-10-XSRMA-A-D. Also co-published with the FAIL Assembly Language Manual as Stanford Artificial Intelligence Laboratory Operating Note 75 and LOTS Computer Facility Operating Note 2. {**848, 955**}

[DEC77] Digital Equipment Corporation, Maynard, MA, USA. *Digital VAX 11/780 Architecture Handbook*, 1977. x + 324 pp. URL http://www.bitsavers.org/pdf/dec/vax/VAX_archHbkVol1_1977.pdf. {**956**}

[DEC79] Digital Equipment Corporation, Maynard, MA, USA. *Digital VAX 11/780 Hardware Handbook*, 1979. x + 324 pp. URL http://www.bitsavers.org/pdf/dec/vax/VAX_archHbkVol1_1977.pdf. {**956**}

[DEC82] Digital Equipment Corporation, Maynard, MA, USA. *VAX-11 Architecture Reference Manual*, May 20, 1982. URL http://www.bitsavers.org/pdf/dec/vax/archSpec/EL-00032-00-decStd32_Jan90.pdf. Revision 6.1. {**956**}

[DEC90] Digital Equipment Corporation, Bedford, MA, USA. *DEC STD 032 VAX Architecture Standard*, January 15, 1990. vi + 486 pp. URL `http://www.bitsavers.org/pdf/dec/vax/archSpec/EL-00032-00-decStd32_Jan90.pdf`. {**956**}

[Dek71] Theodorus J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, June 1971. CODEN NUMMA7. ISSN 0029-599X (print), 0945-3245 (electronic). URL `http://www-gdz.sub.uni-goettingen.de/cgi-bin/digbib.cgi?PPN362160546_0018`. DOI `10.1007/BF01397083`. {**353, 361, 365, 379**}

[Dem81] James Demmel. Effects of underflow on solving linear systems. In *Proceedings: 5th Symposium on Computer Arithmetic, May 18–19, 1981, University of Michigan, Ann Arbor, Michigan*, pages 113–119. IEEE Computer Society Press, Silver Spring, MD, USA, 1981. LCCN QA 76.6 S985t 1981. URL `http://www.acsel-lab.com/arithmetic/arith5/papers/ARITH5_Demmel.pdf`. IEEE catalog number 81CH1630-C. {**79**}

[Den05] Lih-Yuan Deng. Efficient and portable multiple recursive generators of large order. *ACM Transactions on Modeling and Computer Simulation*, 15(1):1–13, January 2005. CODEN ATMCEZ. ISSN 1049-3301 (print), 1558-1195 (electronic). DOI `10.1145/1044322.1044323`. {**1023**}

[Der03] John Derbyshire. *Prime Obsession: Bernhard Riemann and the Greatest Unsolved Problem in Mathematics*. Joseph Henry Press, Washington, DC, USA, 2003. ISBN 0-309-08549-7; 978-0-309-08549-6. xv + 422 pp. LCCN QA246 .D47 2003. {**59, 60, 303, 521, 579, 590**}

[Dev86] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1986. ISBN 0-387-96305-7; 978-0-387-96305-1. xvi + 843 pp. LCCN QA274 .D48 1986. DOI `10.1007/978-1-4613-8643-8`. {**196**}

[Dev02] Keith J. Devlin. *The Millennium Problems: the Seven Greatest Unsolved Mathematical Puzzles of our Time*. Basic Books, New York, NY, USA, 2002. ISBN 0-465-01729-0; 978-0-465-01729-4. x + 237 pp. LCCN QA93 .D485 2002. {**303, 521**}

[Dev08a] Keith J. Devlin. *The Unfinished Game: Pascal, Fermat, and the Seventeenth-Century Letter that Made the World Modern: a Tale of How Mathematics is Really Done*. Basic ideas. Basic Books, New York, NY, USA, 2008. ISBN 0-465-00910-7; 978-0-465-00910-7. x + 191 pp. LCCN QA273 .D455 2008. {**60**}

[Dev08b] Jay L. Devore. *Probability and Statistics for Engineering and the Sciences*. Thomson/Brooks/Cole, Belmont, CA, USA, seventh edition, 2008. ISBN 0-495-38217-5; 978-0-495-38217-1. xvi + 720 pp. LCCN QA273 .D46 2008. {**196, 610**}

[Dev11] Keith J. Devlin. *The man of numbers: Fibonacci's arithmetic revolution*. Walker and Company, New York, NY, USA, 2011. ISBN 0-8027-7812-7 (hardcover); 978-0-8027-7812-3 (hardcover). viii + 183 + 8 pp. {**15, 59**}

[DGD04] Guy Waldo Dunnington, Jeremy Gray, and Fritz-Egbert Dohse. *Carl Friedrich Gauss: titan of science*. Mathematical Association of America, Washington, DC, USA, 2004. ISBN 0-88385-547-X; 978-0-88385-547-8. xxix + 537 + 16 pp. LCCN QA29.G3 D8 2004. {**59**}

[DH97a] Iosif G. Dyadkin and Kenneth G. Hamilton. A family of enhanced Lehmer random number generators, with hyperplane suppression, and direct support for certain physical applications. *Computer Physics Communications*, 107(1–3):258–280, December 22, 1997. CODEN CPHCBZ. ISSN 0010-4655 (print), 1879-2944 (electronic). DOI `10.1016/S0010-4655(97)00101-X`. {**170, 189**}

[DH97b] Iosif G. Dyadkin and Kenneth G. Hamilton. A study of 64-bit multipliers for Lehmer pseudorandom number generators. *Computer Physics Communications*, 103(2–3):103–130, July 1997. CODEN CPHCBZ. ISSN 0010-4655 (print), 1879-2944 (electronic). URL `http://www.sciencedirect.com/science/article/pii/S0010465597000520`. DOI `10.1016/S0010-4655(97)00052-0`. {**170**}

[DH00] Iosif G. Dyadkin and Kenneth G. Hamilton. A study of 128-bit multipliers for congruential pseudorandom number generators. *Computer Physics Communications*, 125(1–3):239–258, March 2000. CODEN CPHCBZ. ISSN 0010-4655 (print), 1879-2944 (electronic). URL `http://cpc.cs.qub.ac.uk/summaries/ADLK`; `http://www.elsevier.com/gej-ng//10/15/40/55/25/42/abstract.html`; `http://www.sciencedirect.com/science/article/pii/S0010465599004671`. DOI `10.1016/S0010-4655(99)00467-1`. {**170**}

[DH04] James Demmel and Yozo Hida. Fast and accurate floating point summation with application to computational geometry. *Numerical Algorithms*, 37(1–4):101–112, December 2004. CODEN NUALEG. ISSN 1017-1398 (print), 1572-9265 (electronic). URL `http://ipsapp009.kluweronline.com/IPS/content/ext/x/J/5058/I/58/A/6/abstract.htm`. DOI `10.1023/B:NUMA.0000049458.99541.38`. {**385**}

[Dij68] Edsger W. Dijkstra. Letter to the Editor: Go to statement considered harmful. *Communications of the Association for Computing Machinery*, 11(3):147–148, March 1968. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/362929.362947`. This letter in support of structured programming, and in favor of eliminating control-flow disruption caused by *go to* statements, inspired scores of others, published mainly in SIGPLAN Notices up to the mid-1980s. The best-known is [Knu74]. {**962, 1020**}

[DKKM87] Cipher A. Deavours, David Kahn, Louis Kruh, and Greg Mellen, editors. *Cryptology Yesterday, Today, and Tomorrow*. The Artech House communication and electronic defense library. Artech House Inc., Norwood, MA, USA, 1987. ISBN 0-89006-253-6; 978-0-89006-253-1. xi + 519 pp. LCCN Z103.C76 1987. First volume of selected papers from issues of Cryptologia. {**1029**}

[DL42a] Gordon C. Danielson and Cornelius Lanczos. Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids. *Journal of The Franklin Institute*, 233(4):365–380, April 1942. CODEN JFINAB. ISSN 0016-0032 (print), 1879-2693 (electronic). DOI `10.1016/S0016-0032(42)90767-1`. {**969**}

[DL42b] Gordon C. Danielson and Cornelius Lanczos. Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids. *Journal of The Franklin Institute*, 233(5):435–452, May 1942. CODEN JFINAB. ISSN 0016-0032 (print), 1879-2693 (electronic). DOI `10.1016/S0016-0032(42)90624-0`. {**969**}

[DLS09] Lih-Yuan Deng, Huajiang Li, and Jyh-Jen Horng Shiau. Scalable parallel multiple recursive generators of large order. *Parallel Computing*, 35(1):29–37, January 2009. CODEN PACOEJ. ISSN 0167-8191 (print), 1872-7336 (electronic). URL `http://www.sciencedirect.com/science/article/pii/S0167819108001099`. DOI `10.1016/j.parco.2008.09.012`. {**1023**}

[Dom03] Diego Dominici. Nested derivatives: a simple method for computing series expansions of inverse functions. *International Journal of Mathematics and Mathematical Sciences*, 58:3699–3715, 2003. ISSN 0161-1712 (print), 1687-0425 (electronic). URL `http://www.hindawi.com/journals/ijmms/2003/457271/abs/`. DOI `10.1155/S0161171203303291`. {**600**}

[Don06] Aleksander Donev. Interoperability with C in Fortran 2003. *ACM Fortran Forum*, 25(1):8–12, April 2006. ISSN 1061-7264 (print), 1931-1311 (electronic). DOI `10.1145/1124708.1124710`. {**941**}

[DR84] Philip J. Davis and Philip Rabinowitz. *Methods of Numerical Integration*. Academic Press, New York, NY, USA, second edition, 1984. ISBN 0-12-206360-0; 978-0-12-206360-2. xiv + 612 pp. LCCN QA299.3 .D28 1984. {**560**}

[dRHG$^+$99] Theo de Raadt, Niklas Hallqvist, Artur Grabowski, Angelos D. Keromytis, and Niels Provos. Cryptography in OpenBSD: An overview. In USENIX, editor, *Usenix Annual Technical Conference. June 6–11, 1999. Monterey, California, USA*, pages 93–101. USENIX, Berkeley, CA, USA, 1999. ISBN 1-880446-33-2; 978-1-880446-33-1. LCCN A76.8.U65 U843 1999. URL `http://www.openbsd.org/papers/crypt-paper.ps`. {**207**}

[dS03] Marcus du Sautoy. *The Music of the Primes: Searching to Solve the Greatest Mystery in Mathematics*. HarperCollins College Publishers, New York, NY, USA, 2003. ISBN 0-06-621070-4; 978-0-06-621070-4. 335 pp. LCCN QA246 .D8 2003. {**60, 303, 521**}

[DSC12] Jean-Pierre Deschamps, Gustavo D. Sutter, and Enrique Cantó. *Guide to FPGA implementation of arithmetic functions*, volume 95 of *Lecture Notes in Electrical Engineering*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2012. ISBN 94-007-2986-3 (hardcover), 94-007-2987-1 (e-book); 978-94-007-2986-5 (hardcover), 978-94-007-2987-2 (e-book). xv + 469 pp. LCCN TK7895.G36. DOI `10.1007/978-94-007-2987-2`. {**978**}

[DSL12a] Lih-Yuan Deng, Jyh-Jen H. Shiau, and Henry Horng-Shing Lu. Efficient computer search of large-order multiple recursive pseudo-random number generators. *Journal of Computational and Applied Mathematics*, 236(13):3228–3237, July 2012. CODEN JCAMDI. ISSN 0377-0427 (print), 1879-1778 (electronic). DOI `10.1016/j.cam.2012.02.023`. {**176, 1023**}

[DSL12b] Lih-Yuan Deng, Jyh-Jen Horng Shiau, and Henry Horng-Shing Lu. Large-order multiple recursive generators with modulus $2^{31} - 1$. *INFORMS Journal on Computing*, 24(4):636–647, Fall 2012. ISSN 1091-9856 (print), 1526-5528 (electronic). URL `http://joc.journal.informs.org/content/24/4/636`. DOI `10.1287/ijoc.1110.0477`. {**1023**}

[Dub83] Augustin A. Dubrulle. Class of numerical methods for the computation of Pythagorean sums. *IBM Journal of Research and Development*, 27(6):582–589, November 1983. CODEN IBMJAE. ISSN 0018-8646 (print), 2151-8556 (electronic). DOI `10.1147/rd.276.0582`. See [MM83] and generalization [Jam89]. {**228, 1017, 1024**}

[Dun55] Guy Waldo Dunnington. *Carl Friedrich Gauss, titan of science: a study of his life and work*. Exposition-university book. Exposition Press, New York, NY, USA, 1955. xi + 479 pp. LCCN QA29.G38 D85 1955. {**59**}

[Dun91] William Dunham. *Journey through Genius: The Great Theorems of Mathematics*. Penguin, New York, NY, USA, 1991. ISBN 0-14-014739-X; 978-0-14-014739-1. xiii + 300 pp. {**541, 591**}

[Dun92] Charles B. Dunham. Surveyor's Forum: "What every computer scientist should know about floating-point arithmetic". *ACM Computing Surveys*, 24(3):319, September 1992. CODEN CMSVAN. ISSN 0360-0300 (print), 1557-7341 (electronic). See [Gol91a, Gol91b, Wic92]. {**1013, 1037**}

[Dun99] William Dunham. *Euler: The Master of Us All*, volume 22 of *The Dolciani mathematical expositions*. Mathematical Association of America, Washington, DC, USA, 1999. ISBN 0-88385-328-0; 978-0-88385-328-3. xxviii + 185 pp. LCCN QA29.E8 D86 1999. {**59, 591**}

[Dun07] William Dunham, editor. *The Genius of Euler: Reflections on his Life and Work*, volume 2 of *Spectrum series; MAA tercentenary Euler celebration*. Mathematical Association of America, Washington, DC, USA, 2007. ISBN 0-88385-558-5; 978-0-88385-558-4. xvi + 309 pp. LCCN QA29.E8 G46 2007. {**591**}

[DX03] Lih-Yuan Deng and Hongquan Xu. A system of high-dimensional, efficient, long-cycle and portable uniform random number generators. *ACM Transactions on Modeling and Computer Simulation*, 13(4):299–309, October 2003. CODEN ATMCEZ. ISSN 1049-3301 (print), 1558-1195 (electronic). DOI `10.1145/945511.945513`. {**1023**}

[ECM05] ECMA. *ECMA-367: Eiffel analysis, design and programming language*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, June 2005. URL `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-367.pdf`; `http://www.ecma-international.org/publications/standards/Ecma-367.htm`. {**830**}

[ECM06a] ECMA. *ECMA-334: C# Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, fourth edition, June 2006. xix + 531 pp. URL `http://www.ecma-international.org/publications/files/ecma-st/ECMA-334.pdf`; `http://www.ecma-international.org/publications/standards/Ecma-334.htm`. {**vii, 917**}

[ECM06b] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, fourth edition, June 2006. vii + 104 (Part I), viii + 191 (Part II), iv + 138 (Part III), ii + 20 (Part IV), i + 4 (Part V), ii + 57 (Part VI) pp. URL `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf`; `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.zip`; `http://www.ecma-international.org/publications/standards/Ecma-335.htm`. {**917, 918**}

[EH92] Jürgen Eichenauer-Herrmann. Inversive congruential pseudorandom numbers: a tutorial. *International Statistical Review = Revue Internationale de Statistique*, 60(2):167–176, August 1992. CODEN ISTRDP. ISSN 0306-7734 (print), 1751-5823 (electronic). URL `http://www.jstor.org/stable/1403647`. DOI `10.2307/1403647`. {**180**}

[EH95] Jürgen Eichenauer-Herrmann. Pseudorandom number generation by nonlinear methods. *International Statistical Review = Revue Internationale de Statistique*, 63(2):247–255, August 1995. CODEN ISTRDP. ISSN 0306-7734 (print), 1751-5823 (electronic). URL `http://www.jstor.org/stable/1403620`. DOI `10.2307/1403620`. {**177, 1026, 1035**}

[EHHW98] Jürgen Eichenauer-Herrmann, Eva Herrmann, and Stefan Wegenkittl. A survey of quadratic and inversive congruential pseudorandom numbers. In Harald Niederreiter, Peter Hellekalek, Gerhard Larcher, and Peter Zinterhof, editors, *Monte Carlo and Quasi-Monte Carlo methods 1996: proceedings of a conference at the University of Salzburg, Austria, July 9–12, 1996*, volume 127 of *Lecture Notes in Statistics*, pages 66–97. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1998. ISBN 0-387-98335-X (softcover); 978-0-387-98335-6 (softcover). LCCN Q183.9 .M67 1998. DOI 10.1007/978-1-4612-1690-2_4. {**180**}

[El 06] Refaat A. El Attar. *Special Functions and Orthogonal Polynomials*, volume 3 of *Mathematical series*. Lulu Press, Morrisville, NC, USA, 2006. ISBN 1-4116-6690-9 (paperback); 978-1-4116-6690-0 (paperback). vi + 302 pp. LCCN QA404.5 .E5 2006; QA351 .E5 2006. {**59**}

[EL86] Jürgen Eichenauer and Jürgen Lehn. A non-linear congruential pseudo random number generator. *Statistical Papers = Statistische Hefte*, 27(1):315–326, September 1986. CODEN STPAE4. ISSN 0932-5026 (print), 1613-9798 (electronic). DOI 10.1007/BF02932576. {**180**}

[EL04a] Miloš Dragutin Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2004. ISBN 1-55860-798-6; 978-1-55860-798-9. xxv + 709 pp. LCCN QA76.9.C62 E72 2004. {**104, 407, 881, 978**}

[EL04b] Pierre Eymard and Jean-Pierre Lafon. *The Number π [pi]*. American Mathematical Society, Providence, RI, USA, 2004. ISBN 0-8218-3246-8; 978-0-8218-3246-2. x + 322 pp. LCCN QA484 .E9613 2004. URL http://www.ams.org/bookpages/tnp/. Translated by Stephen S. Wilson from the 1999 French original, *Autour du nombre π [pi]*. {**14, 59, 623**}

[EM79] Richard H. Eckhouse, Jr. and L. Robert Morris. *Minicomputer Systems: Organization, Programming, and Applications (PDP-11)*. Prentice-Hall, Upper Saddle River, NJ, USA, 1979. ISBN 0-13-583914-9; 978-0-13-583914-0. xix + 491 pp. LCCN QA76.8.P2E26 1979. {**956**}

[EM94] Boelie Elzen and Donald MacKenzie. The social limits of speed: The development and use of supercomputers. *IEEE Annals of the History of Computing*, 16(1):46–61, Spring 1994. CODEN IAHCEX. ISSN 1058-6180 (print), 1934-1547 (electronic). URL http://dlib.computer.org/an/books/an1994/pdf/a1046.pdf; http://www.computer.org/annals/an1994/a1046abs.htm. DOI 10.1109/85.251854. {**952**}

[Ent98] Karl Entacher. Bad subsequences of well-known linear congruential pseudorandom number generators. *ACM Transactions on Modeling and Computer Simulation*, 8(1):61–70, January 1998. CODEN ATMCEZ. ISSN 1049-3301 (print), 1558-1195 (electronic). DOI 10.1145/272991.273009. {**172**}

[ESC05] D. Eastlake, 3rd, J. Schiller, and S. Crocker. RFC 4086: Randomness recommendations for security, June 2005. URL ftp://ftp.internic.net/rfc/rfc4086.txt. {**214**}

[ESU01] Karl Entacher, Thomas Schell, and Andreas Uhl. Optimization of random number generators: efficient search for high-quality LCGs. *Probabilistic Engineering Mechanics*, 16(4):289–293, October 2001. CODEN PEMEEX. ISSN 0266-8920 (print), 1878-4275 (electronic). URL http://www.sciencedirect.com/science/article/pii/S0266892001000212. DOI 10.1016/S0266-8920(01)00021-2. {**170**}

[ETZ09] Andreas Enge, Philippe Théveny, and Paul Zimmermann. *mpc — A library for multiprecision complex arithmetic with exact rounding*. INRIA, France, 0.8.1 edition, December 2009. URL http://mpc.multiprecision.org/. {**825**}

[Eul92] Leonhard Euler. *Leonhardi Euleri Opera Omnia: Series Prima: Opera Mathematica*. Birkhäuser, Cambridge, MA, USA; Berlin, Germany; Basel, Switzerland, 1992. ISBN 3-7643-1474-5; 978-3-7643-1474-3. 29 volumes in first series alone. {**591**}

[Eve83] Howard Eves. *An Introduction to the History of Mathematics*. Saunders College Publishing, Philadelphia, PA, USA, 1983. ISBN 0-03-062064-3; 978-0-03-062064-5. xviii + 593 pp. LCCN QA21.E8. {**59**}

[FC64] M. A. Fisherkeller and William J. Cody, Jr. Tables of the complete elliptic integrals K, K', E, and E'. Technical Memo ANL AMD 71, Argonne National Laboratory, Argonne, IL, USA, 1964. 14 pp. See review by John W. Wrench in Mathematics of Computation, **19**(89–92), 342, 1965. {**644**}

[Fel07] Emil Alfred Fellmann. *Leonhard Euler*. Birkhäuser, Cambridge, MA, USA; Berlin, Germany; Basel, Switzerland, 2007. ISBN 3-7643-7538-8; 978-3-7643-7538-6. xv + 179 pp. LCCN QA29.E8 F452 2007. Translated by E. Gautschi and W. Gautschi from the 1995 German original of the same title. {**591**}

[Fer95] Warren E. Ferguson, Jr. Exact computation of a sum or difference with applications to argument reduction. In Knowles and McAllister [KM95], pages 216–221. ISBN 0-8186-7089-4 (paperback), 0-8186-7089-4 (case), 0-8186-7149-1 (microfiche), 0-8186-7089-4 (softbound), 0-7803-2949-X (casebound); 978-0-8186-7089-3 (paperback), 978-0-8186-7089-3 (case), 978-0-8186-7149-4 (microfiche), 978-0-8186-7089-3 (softbound), 978-0-7803-2949-2 (casebound). LCCN QA 76.9 C62 S95 1995. URL http://www.acsel-lab.com/arithmetic/arith12/papers/ARITH12_Ferguson.pdf. DOI 10.1109/ARITH.1995.465355. {**271**}

[Fet74] Henry E. Fettis. A stable algorithm for computing the inverse error function in the 'tail-end' region. *Mathematics of Computation*, 28(126):585–587, April 1974. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL http://www.jstor.org/stable/2005933. DOI 10.2307/2005933. {**600**}

[FF01] Sarah Flannery and David Flannery. *In Code: A [Young Women's] Mathematical Journey*. Algonquin Books of Chapel Hill, Chapel Hill, NC, USA, 2001. ISBN 1-56512-377-8; 978-1-56512-377-9. ix + 341 pp. LCCN QA29.F6 A3 2003. {**208, 591**}

[FHL$^+$07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):1–15, June 2007. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI 10.1145/1236463.1236468. {**401, 407**}

[FHS78a] Phyllis A. Fox, A. D. Hall, and Norman L. Schryer. Algorithm 528: Framework for a portable library [Z]. *ACM Transactions on Mathematical Software*, 4(2):177–188, June 1978. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI 10.1145/355780.355789. See remarks [Fox79, GG99]. {**823, 1010, 1012**}

[FHS78b] Phyllis A. Fox, A. D. Hall, and Norman L. Schryer. The PORT mathematical subroutine library. *ACM Transactions on Mathematical Software*, 4(2):104–126, June 1978. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI 10.1145/355780.355783. {**341, 352**}

[FI94] Toshio Fukushima and Hideharu Ishizaki. Numerical computation of incomplete elliptic integrals of a general form. *Celestial Mechanics and Dynamical Astronomy*, 59(3):237–251, July 1994. CODEN CLMCAV. ISSN 0923-2958 (print), 1572-9478 (electronic). URL `http://www.springerlink.com/content/0923-2958/`. DOI `10.1007/BF00692874`. {**645, 690**}

[Fin97] B. F. Finkel. Biography: Leonhard Euler. *American Mathematical Monthly*, 4(12):297–302, December 1897. CODEN AMMYAE. ISSN 0002-9890 (print), 1930-0972 (electronic). URL `http://www.jstor.org/stable/2968971`. DOI `10.2307/2968971`. {**591**}

[Fin03] Steven R. Finch. *Mathematical Constants*, volume 94 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, UK, 2003. ISBN 0-521-81805-2; 978-0-521-81805-6. xix + 602 pp. LCCN QA41 .F54 2003. URL `http://numbers.computation.free.fr/Constants/constants.html`. {**59**}

[FIS64] A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth. A formal description of SYSTEM/360. *IBM Systems Journal*, 3(2):198–261, 1964. CODEN IBMSA7. ISSN 0018-8670. URL `http://www.research.ibm.com/journal/sj/032/falkoff.pdf`. DOI `10.1147/sj.32.0198`. {**928**}

[FM82] George S. Fishman and Louis R. Moore III. A statistical evaluation of multiplicative congruential random number generators with modulus $2^{31} - 1$. *Journal of the American Statistical Association*, 77(377):129–136, March 1982. CODEN JSTNAL. ISSN 0162-1459 (print), 1537-274x (electronic). URL `http://links.jstor.org/sici?sici=0162-1459%28198203%2977%3A377%3C129%3AASEOMC%3E2.0.CO%3B2-Q`. DOI `10.2307/2287778`. {**170**}

[FM86a] George S. Fishman and Louis R. Moore III. An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$. *SIAM Journal on Scientific and Statistical Computing*, 7(1):24–45, January 1986. CODEN SIJCD4. ISSN 0196-5204. URL `http://link.aip.org/link/?SCE/7/24/1`. DOI `10.1137/0907002`. See erratum [FM86b]. {**170, 1010**}

[FM86b] George S. Fishman and Louis R. Moore III. Erratum: "An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$". *SIAM Journal on Scientific and Statistical Computing*, 7(3):1058, July 1986. CODEN SIJCD4. ISSN 0196-5204. URL `http://epubs.siam.org/sisc/resource/1/sjoce3/v7/i3/p1058_s1`; `http://link.aip.org/link/?SCE/7/1058/1`. DOI `10.1137/0907072`. See [FM86a]. {**1010**}

[FO01] Michael J. Flynn and Stuart F. Oberman. *Advanced Computer Arithmetic Design*. Wiley, New York, NY, USA, 2001. ISBN 0-471-41209-0; 978-0-471-41209-0. xv + 325 pp. LCCN TK7895.A65 F59 2001. {**17**}

[For69a] George E. Forsythe. Solving a quadratic equation on a computer. In George A. W. Boehm, editor, *The Mathematical Sciences: a Collection of Essays*, pages 138–152. MIT Press, Cambridge, MA, USA, 1969. LCCN QA11 .M39. Edited by the National Research Council's Committee on Support of Research in the Mathematical Sciences (COSRIMS) with the collaboration of George A. W. Boehm. {**472, 474**}

[For69b] George E. Forsythe. What is a satisfactory quadratic equation solver? In Bruno Dejon and Peter Henrici, editors, *Constructive aspects of the fundamental theorem of algebra: Proceedings of a symposium conducted at the IBM Research Laboratory, Zürich-Rüschlikon, Switzerland, June 5–7, 1967*, pages 53–61. Wiley-Interscience, New York, NY, USA, 1969. ISBN 0-471-20300-9; 978-0-471-20300-1. LCCN QA212 .C65. URL `http://www.dtic.mil/dtic/tr/fulltext/u2/657639.pdf`. {**472**}

[Fox79] Phyllis A. Fox. Remark on "Algorithm 528: Framework for a portable library [Z]". *ACM Transactions on Mathematical Software*, 5(4):524, December 1979. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/355853.355871`. See [FHS78a, GG99]. {**1009, 1012**}

[FP68] L. Fox and I. B. Parker. *Chebyshev Polynomials in Numerical Analysis*. Oxford mathematical handbooks. Oxford University Press, Oxford, UK, 1968. ix + 205 pp. LCCN QA297 .F65. {**58**}

[Fra99] Donald R. Franceschetti, editor. *Biographical Encyclopedia of Mathematicians*. Marshall Cavendish, New York, NY, USA, 1999. ISBN 0-7614-7069-7 (set), 0-7614-7070-0 (vol. 1), 0-7614-7071-9 (vol. 2); 978-0-7614-7069-4 (set), 978-0-7614-7070-0 (vol. 1), 978-0-7614-7071-7 (vol. 2). xiv + 585 + xix pp. LCCN QA28 .B544 1999. {**59**}

[Fri67] Paul Friedland. Algorithm 312: Absolute value and square root of a complex number. *Communications of the Association for Computing Machinery*, 10(10):665, October 1967. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/363717.363780`. {**481**}

[FS03] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. Wiley, New York, NY, USA, 2003. ISBN 0-471-22894-X (hardcover), 0-471-22357-3 (paperback); 978-0-471-22894-3 (hardcover), 978-0-471-22357-3 (paperback). xx + 410 pp. LCCN QA76.9.A25 F466 2003. URL `http://www.counterpane.com/book-practical.html`. {**168, 206, 214, 591**}

[FSK10] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley, New York, NY, USA, 2010. ISBN 0-470-47424-6 (paperback); 978-0-470-47424-2 (paperback). xxix + 353 pp. LCCN QA76.9.A25 F466 2010. {**214**}

[FTN91] *International Standard: Information, Technology, Programming Languages, Fortran*. International Organization for Standardization, Geneva, Switzerland, second edition, 1991. xvii + 369 pp. URL `http://www.iso.ch/cate/d26933.html`; `http://www.iso.ch/cate/d26934.html`; `http://www.iso.ch/cate/d29926.html`. {**vii, 106**}

[FTN97] *ISO/IEC 1539-1:1997: Information technology — Programming languages — Fortran — Part 1: Base language*. International Organization for Standardization, Geneva, Switzerland, 1997. URL `http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+1539%2D1%3A1997`. {**vii**}

[FTN04a] *Draft International Standard ISO/IEC 1539-1:2004(E): Information technology — Programming languages — Fortran Part 1: Base Language*. International Organization for Standardization, Geneva, Switzerland, May 2004. xiv + 569 pp. URL `ftp://ftp.nag.co.uk/sc22wg5/N1601-N1650/N1601.pdf.gz`. {**341**}

[FTN04b] *ISO/IEC 1539-1:2004 Information technology — Programming languages — Fortran – Part 1: Base language*. International Organization for Standardization, Geneva, Switzerland, 2004. xiv + 569 pp. URL `http://www.dkuug.dk/jtc1/sc22/open/n3661.pdf`. {**941**}

[FTN10] *ISO/IEC 1539-1:2010 Information technology — Programming languages — Fortran — Part 1: Base language*. International Organization for Standardization, Geneva, Switzerland, June 2010. xviii + 603 pp. URL ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1830.pdf. {**vii, 223, 591, 593, 694, 941**}

[Fuk09a] Toshio Fukushima. Fast computation of complete elliptic integrals and Jacobian elliptic functions. *Celestial Mechanics and Dynamical Astronomy*, 105(4):305–328, December 2009. CODEN CLMCAV. ISSN 0923-2958 (print), 1572-9478 (electronic). URL http://www.springerlink.com/content/0923-2958/. DOI 10.1007/s10569-009-9228-z. {**627, 645, 690**}

[Fuk09b] Toshio Fukushima. Fast computation of Jacobian elliptic functions and incomplete elliptic integrals for constant values of elliptic parameter and elliptic characteristic. *Celestial Mechanics and Dynamical Astronomy*, 105(1–3):245–260, 2009. CODEN CLMCAV. ISSN 0923-2958 (print), 1572-9478 (electronic). URL http://www.springerlink.com/content/0923-2958/. DOI 10.1007/s10569-008-9177-y. {**627, 645, 690**}

[Fuk10] Toshio Fukushima. Fast computation of incomplete elliptic integral of first kind by half argument transformation. *Numerische Mathematik*, 116(4):687–719, October 2010. CODEN NUMMA7. ISSN 0029-599X (print), 0945-3245 (electronic). URL http://www.springerlink.com/openurl.asp?genre=article&issn=0029-599X&volume=116&issue=4&spage=687. DOI 10.1007/s00211-010-0321-8. {**645, 690**}

[Fuk11] Toshio Fukushima. Precise and fast computation of the general complete elliptic integral of the second kind. *Mathematics of Computation*, 80(275):1725–1743, July 2011. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL http://www.ams.org/journals/mcom/2011-80-275/S0025-5718-2011-02455-5/; http://www.ams.org/journals/mcom/2011-80-275/S0025-5718-2011-02455-5/S0025-5718-2011-02455-5.pdf. DOI 10.1090/S0025-5718-2011-02455-5. {**690**}

[Fuk12] Toshio Fukushima. Series expansions of symmetric elliptic integrals. *Mathematics of Computation*, 81(278):957–990, April 2012. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL http://www.ams.org/journals/mcom/2012-81-278/S0025-5718-2011-02531-7; http://www.ams.org/journals/mcom/2012-81-278/S0025-5718-2011-02531-7/S0025-5718-2011-02531-7.pdf. DOI 10.1090/S0025-5718-2011-02531-7. {**690**}

[Fuk13a] Toshio Fukushima. Precise and fast computation of Jacobian elliptic functions by conditional duplication. *Numerische Mathematik*, 123(4):585–605, April 2013. CODEN NUMMA7. ISSN 0029-599X (print), 0945-3245 (electronic). URL http://link.springer.com/article/10.1007/s00211-012-0498-0. DOI 10.1007/s00211-012-0498-0. {**690**}

[Fuk13b] Toshio Fukushima. Recursive computation of derivatives of elliptic functions and of incomplete elliptic integrals. *Applied Mathematics and Computation*, 221:21–31, September 15, 2013. CODEN AMHCBQ. ISSN 0096-3003 (print), 1873-5649 (electronic). URL http://www.sciencedirect.com/science/article/pii/S0096300313006152. DOI 10.1016/j.amc.2013.06.008. {**690**}

[Ful81a] L. Wayne Fullerton. FNLIB user's manual. Technical report CSTR 95, Bell Telephone Laboratories, Murray Hill, NJ, USA, March 1981. {**341, 352**}

[Ful81b] L. Wayne Fullerton. FNLIB user's manual explanatory table of contents. Technical report CSTR 92, Bell Telephone Laboratories, Murray Hill, NJ, USA, March 1981. {**341, 352**}

[FvGM90] W. H. J. Feijen, A. J. M. van Gasteren, David Gries, and J. Misra, editors. *Beauty is our Business: a Birthday Salute to Edsger W. Dijkstra*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1990. ISBN 0-387-97299-4; 978-0-387-97299-2. xix + 453 pp. LCCN QA76 .B326 1990. DOI 10.1007/978-1-4612-4476-9. Contains important treatment of accurate binary-to-decimal conversion [Gri90, Knu90]. {**1013, 1020**}

[Gam88] George Gamow. *One, Two, Three, . . . , Infinity: Facts and Speculations of Science*. Dover, New York, NY, USA, 1988. ISBN 0-486-25664-2 (paperback); 978-0-486-25664-1 (paperback). xii + 340 pp. LCCN Q162 .G23 1988. {**59**}

[Gan95] Mike Gancarz. *The UNIX philosophy*. Digital Press, Bedford, MA, USA, 1995. ISBN 1-55558-123-4; 978-1-55558-123-7. xix + 151 pp. LCCN QA76.76.O63G365 1995. {**956**}

[Gan03] Mike Gancarz. *Linux and the Unix Philosophy*. Digital Press, Bedford, MA, USA, 2003. ISBN 1-55558-273-7; 978-1-55558-273-9. xxvii + 220 pp. LCCN QA76.76.O63G364 2003. {**956**}

[Gau64] Walter Gautschi. ACM Algorithm 236: Bessel functions of the first kind [S17]. *Communications of the Association for Computing Machinery*, 7(8):479–480, August 1964. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI 10.1145/355586.355587. See remark [Sko75]. {**693, 1032**}

[Gau67] Walter Gautschi. Computational aspects of three-term recurrence relations. *SIAM Review*, 9(1):24–82, January 1967. CODEN SIREAD. ISSN 0036-1445 (print), 1095-7200 (electronic). URL http://link.aip.org/link/?SIR/9/24/1. DOI 10.1137/1009002. {**705**}

[Gau79a] Walter Gautschi. Algorithm 542: Incomplete gamma functions [S14]. *ACM Transactions on Mathematical Software*, 5(4):482–489, December 1979. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI 10.1145/355853.355864. {**562**}

[Gau79b] Walter Gautschi. A computational procedure for incomplete gamma functions. *ACM Transactions on Mathematical Software*, 5(4):466–481, December 1979. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI 10.1145/355853.355863. {**562**}

[Gau04] Walter Gautschi. *Orthogonal Polynomials: Computation and Approximation*. Oxford University Press, Oxford, UK, 2004. ISBN 0-19-850672-4; 978-0-19-850672-0. viii + 301 pp. LCCN QA404.5 .G356 2004. {**58, 59**}

[Gau08] Walter Gautschi. Leonhard Euler: His life, the man, and his works. *SIAM Review*, 50(1):3–33, 2008. CODEN SIREAD. ISSN 0036-1445 (print), 1095-7200 (electronic). URL http://link.aip.org/link/?SIR/50/3/1. DOI 10.1137/070702710. {**59, 591**}

[Gay90] David M. Gay. Correctly rounded binary-decimal and decimal-binary conversions. Numerical Analysis Manuscript 90-10, AT&T Bell Laboratories, November 30 1990. 16 pp. URL http://cm.bell-labs.com/cm/cs/doc/90/4-10.ps.gz; http://www.ampl.com/ampl/REFS/rounding.ps.gz; http://www.netlib.org/fp/dtoa.c; http://www.netlib.org/fp/g_fmt.c; http://www.netlib.org/fp/gdtoa.tgz; http://www.netlib.org/fp/rnd_prod.s. {**895**}

[GB91] Shmuel Gal and Boris Bachelis. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Transactions on Mathematical Software*, 17(1):26–45, March 1991. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.acm.org/pubs/citations/journals/toms/1991-17-1/p26-gal/`. DOI `10.1145/103147.103151`. {**827**}

[GBC$^+$02] Solomon W. Golomb, Elwyn Berlekamp, Thomas M. Cover, Robert G. Gallager, James L. Massey, and Andrew J. Viterbi. Claude Elwood Shannon (1916–2001). *Notices of the American Mathematical Society*, 49(1):8–16, January 2002. CODEN AMNOAN. ISSN 0002-9920 (print), 1088-9477 (electronic). URL `http://www.ams.org/notices/200201/fea-shannon.pdf`. {**969**}

[GBGL08] Timothy Gowers, June Barrow-Green, and Imre Leader, editors. *The Princeton Companion to Mathematics*. Princeton University Press, Princeton, NJ, USA, 2008. ISBN 0-691-11880-9; 978-0-691-11880-2. xx + 1034 pp. LCCN QA11.2 .P745 2008. {**59, 60**}

[GDT$^+$05] Mark Galassi, Jim Davies, James Theiler, Brian Gough, Gerard Jungman, Michael Booth, and Fabrice Rossi. *GNU Scientific Library: Reference Manual*. Network Theory Ltd., Bristol, UK, second revised edition, 2005. ISBN 0-9541617-3-4; 978-0-9541617-3-6. xvi + 601 pp. LCCN QA76.73.C15. URL `http://www.network-theory.co.uk/gsl/manual/`. {**567, 583, 693, 694, 825**}

[Gen03] James E. Gentle. *Random Number Generation and Monte Carlo Methods*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., second edition, 2003. ISBN 0-387-00178-6; 978-0-387-00178-4. xv + 381 pp. LCCN QA298 .G46 2003. URL `http://www.science.gmu.edu/~jgentle/rngbk/`. DOI `10.1007/b97336`. {**214**}

[GG98] I. Grattan-Guinness. *The Norton History of the Mathematical Sciences: the Rainbow of Mathematics*. Norton history of science. W. W. Norton & Co., New York, NY, USA, 1998. ISBN 0-393-04650-8; 978-0-393-04650-2. 817 pp. LCCN QA21 .G695 1998. {**59**}

[GG99] David M. Gay and Eric Grosse. Self-adapting Fortran 77 machine constants: Comment on Algorithm 528. *ACM Transactions on Mathematical Software*, 25(1):123–126, March 1999. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). URL `http://cm.bell-labs.com/who/ehg/mach/d1mach.ps`. DOI `10.1145/305658.305711`. See [FHS78a, Fox79]. {**1009, 1010**}

[GH85] Paul Griffiths and Ian David Hill, editors. *Applied Statistics Algorithms*. Ellis Horwood series in mathematics and its applications. Ellis Horwood, New York, NY, USA, 1985. ISBN 0-85312-772-7 (UK), 0-470-20184-3 (US); 978-0-85312-772-7 (UK), 978-0-470-20184-8 (US). 307 pp. LCCN QA276.4 .A57 1985. Published for the Royal Statistical Society. {**1037**}

[Gil51] S. Gill. A process for the step-by-step integration of differential equations in an automatic digital computing machine. *Proceedings of the Cambridge Philosophical Society. Mathematical and physical sciences*, 47(1):96–108, January 1951. CODEN PCPSA4. ISSN 0008-1981. DOI `10.1017/S0305004100026414`. {**353**}

[GJS96] James Gosling, Bill Joy, and Guy L. Steele Jr. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996. ISBN 0-201-63451-1; 978-0-201-63451-8. xxv + 825 pp. LCCN QA76.73.J38G68 1996. URL `http://www.aw.com/cp/javaseries.html`; `http://www.aw.com/cseng/titles/0-201-63451-1/`. {**979**}

[GJS$^+$13] James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Addison-Wesley, Reading, MA, USA, Java SE 7 edition, 2013. ISBN 0-13-326022-4 (paperback); 978-0-13-326022-9 (paperback). xxvii + 644 pp. LCCN QA76.73.J38 G68 2013. {**vii, 979**}

[GJS$^+$14] James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Addison-Wesley, Addison-Wesley, Java SE 8 edition, 2014. ISBN 0-13-390069-X (paperback); 978-0-13-390069-9 (paperback). xxii + 758 pp. LCCN QA76.73.J38 G68 2014. {**vii, 979**}

[GJSB00] James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *The Java language specification*. Java series. Addison-Wesley, Reading, MA, USA, second edition, 2000. ISBN 0-201-31008-2; 978-0-201-31008-5. xxv + 505 pp. LCCN QA76.73.J38 G68 2000. URL `http://java.sun.com/people/jag/`. {**vii, 979**}

[GJSB05] James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *The Java Language Specification*. The Java series. Addison-Wesley, Reading, MA, USA, third edition, 2005. ISBN 0-321-24678-0 (paperback); 978-0-321-24678-3 (paperback). xxxii + 651 pp. {**vii, 978, 979**}

[GLL05] Stef Graillat, Philippe Langlois, and Nicolas Louvet. Compensated Horner scheme. Research Report RR2005-04, Équipe de Recherche DALI, Laboratoire LP2A, Université de Perpignan, Via Domitia, Perpignan, France, July 24, 2005. ii + 25 pp. URL `http://gala.univ-perp.fr/~graillat/papers/rr2005-04.pdf`. {**89**}

[GLL06] Stef Graillat, Philippe Langlois, and Nicolas Louvet. Improving the compensated Horner scheme with a fused multiply and add. In Hisham M. Haddad, editor, *Applied computing 2006: proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France, April 23–27, 2006*, pages 1323–1327. ACM Press, New York, NY 10036, USA, 2006. ISBN 1-59593-108-2; 978-1-59593-108-5. LCCN QA76.76.A65 S95 2006. URL `http://portal.acm.org/toc.cfm?id=1141277`. DOI `10.1145/1141277.1141585`. {**89**}

[GM74] M. W. Gentleman and S. B. Marovich. More on algorithms that reveal properties of floating point arithmetic units. *Communications of the Association for Computing Machinery*, 17(5):276–277, May 1974. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/360980.361003`. See [Mal72]. {**1023**}

[GM98] Robert Gordon and Alan McClellan. *Essential JNI: Java Native Interface*. Prentice-Hall, Upper Saddle River, NJ, USA, 1998. ISBN 0-13-679895-0; 978-0-13-679895-8. xxvii + 499 pp. LCCN QA76.73.J38 G665 1998. URL `http://www.prenhall.com/ptrbooks/ptr_0136798950.html`. {**979**}

[GME99] Fred G. Gustavson, José E. Moreira, and Robert F. Enenkel. The fused multiply-add instruction leads to algorithms for extended-precision floating point: applications to Java and high-performance computing. In Stephen A. MacKay and J. Howard Johnson, editors, *CASCON '99: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. November 8–11, 1999, Mississauga, Ontario, Canada*, page [4]. IBM Corporation, San Jose, CA, USA, 1999. URL `http://tinyurl.com/z9dzuxf`. Dedicated to Cleve Moler on his 60th birthday. {**354**}

[Gol67] I. Bennett Goldberg. 27 bits are not enough for 8-digit accuracy. *Communications of the Association for Computing Machinery*, 10(2):105–106, February 1967. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/363067.363112`. {**840, 844, 851**}

[Gol91a] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991. CODEN CMSVAN. ISSN 0360-0300 (print), 1557-7341 (electronic). URL `http://www.acm.org/pubs/toc/Abstracts/0360-0300/103163.html`. DOI `10.1145/103162.103163`. This widely cited article is an outstanding presentation of floating-point arithmetic. See also correction [Gol91b] and remarks [Dun92, Wic92]. {**1008, 1013, 1037**}

[Gol91b] David Goldberg. Corrigendum: "What every computer scientist should know about floating-point arithmetic". *ACM Computing Surveys*, 23(3):413, September 1991. CODEN CMSVAN. ISSN 0360-0300 (print), 1557-7341 (electronic). See [Gol91a, Dun92, Wic92]. {**103, 1008, 1013, 1037**}

[Gol02] David Goldberg. Computer arithmetic. In *Computer Architecture — A Quantitative Approach* [PH02], chapter H, pages H–1–H–74. ISBN 1-55860-596-7; 978-1-55860-596-1. LCCN QA76.9.A73 P377 2003. URL `http://books.elsevier.com/companions/1558605967/appendices/1558605967-appendix-h.pdf`. The complete Appendix H is not in the printed book; it is available only at the book's Web site: `http://www.mkp.com/CA3`. {**103**}

[Goo69] I. J. Good. How random are random numbers? *The American Statistician*, 23(4):42–45, October 1969. CODEN ASTAAJ. ISSN 0003-1305 (print), 1537-2731 (electronic). URL `http://www.jstor.org/stable/2681742`. DOI `10.2307/2681742`. {**169**}

[Gra00] Jeremy Gray. *The Hilbert Challenge*. Oxford University Press, Oxford, UK, 2000. ISBN 0-19-850651-1; 978-0-19-850651-5. xii + 315 pp. LCCN QA29.H5 G739 2000. {**579, 590**}

[Gra09] Stef Graillat. Accurate floating-point product and exponentiation. *IEEE Transactions on Computers*, 58(7):994–1000, July 2009. CODEN ITCOB4. ISSN 0018-9340 (print), 1557-9956 (electronic). URL `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4711041`. DOI `10.1109/TC.2008.215`. {**529**}

[GRAST16] Amparo Gil, Diego Ruiz-Antolín, Javier Segura, and Nico M. Temme. Algorithm 969: Computation of the incomplete gamma function for negative values of the argument. *ACM Transactions on Mathematical Software*, 43(3):26:1–26:9, November 2016. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://dl.acm.org/citation.cfm?id=2972951`. DOI `10.1145/2972951`. {**567**}

[Gri90] David Gries. Binary to decimal, one more time. In Feijen et al. [FvGM90], chapter 16, pages 141–148. ISBN 0-387-97299-4; 978-0-387-97299-2. LCCN QA76 .B326 1990. DOI `10.1007/978-1-4612-4476-9_17`. This paper presents an alternate proof of Knuth's algorithm [Knu90] for conversion between decimal and fixed-point binary numbers. {**895, 1011, 1020**}

[GRJZ07] I. S. Gradshteyn, I. M. Ryzhik, Alan Jeffrey, and Daniel Zwillinger. *Table of Integrals, Series and Products*. Academic Press, New York, NY, USA, seventh edition, 2007. ISBN 0-12-373637-4 (hardcover); 978-0-12-373637-6 (hardcover). xlv + 1171 pp. LCCN QA55 .G6613 2007. {**58, 619, 682, 687**}

[GSR^{+}04] Torbjörn Granlund, Gunnar Sjödin, Hans Riesel, Richard Stallman, Brian Beuning, Doug Lea, John Amanatides, Paul Zimmermann, Ken Weber, Per Bothner, Joachim Hollman, Bennet Yee, Andreas Schwab, Robert Harley, David Seal, Robert Harley, Torsten Ekedahl, Paul Zimmermann, Linus Nordberg, Kent Boortz, Kevin Ryde, Steve Root, Gerardo Ballabio, and Hans Thorsen. *GNU MP: The GNU Multiple Precision Arithmetic Library*. Free Software Foundation, Boston, MA, USA, version 4.1.4 edition, September 21, 2004. iv + 127 pp. URL `ftp://ftp.gnu.org/gnu/gmp/gmp-4.1.4.tar.gz; http://www.swox.se/gmp/`. GNU MP development began in 1991. Earler versions are 1.0 (8-Aug-1991), 2.0 (24-Apr-1996), 3.0 (17-Apr-2000), and 4.0 (1-Dec-2001). {**401, 407, 825**}

[GST02] Amparo Gil, Javier Segura, and Nico M. Temme. Evaluation of the modified Bessel function of the third kind of imaginary orders. *Journal of Computational Physics*, 175(2):398–411, January 20, 2002. CODEN JCTPAH. ISSN 0021-9991 (print), 1090-2716 (electronic). DOI `10.1006/jcph.2001.6894`. {**693**}

[GST04] Amparo Gil, Javier Segura, and Nico M. Temme. Computing solutions of the modified Bessel differential equation for imaginary orders and positive arguments. *ACM Transactions on Mathematical Software*, 30(2):145–158, June 2004. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/992200.992203`. {**693**}

[GST07] Amparo Gil, Javier Segura, and Nico M. Temme. *Numerical Methods for Special Functions*. SIAM (Society for Industrial and Applied Mathematics), Philadelphia, PA, USA, 2007. ISBN 0-89871-634-9; 978-0-89871-634-4. xvi + 415 pp. LCCN QA351 .G455 2007. {**13, 17, 18, 58, 644, 693, 827**}

[Gus98] John Gustafson. Computational verifiability and feasibility of the ASCI program. *IEEE Computational Science & Engineering*, 5(1):36–45, January/March 1998. CODEN ISCEE4. ISSN 1070-9924 (print), 1558-190x (electronic). DOI `10.1109/99.660304`. Discusses recent progress in interval arithmetic and its relevance to error estimation in very large computations of the type envisioned for the US ASCI (Advanced Strategic Computing Initiative) project. {**960, 967**}

[Gut04] Peter Gutmann. *Cryptographic Security Architecture: Design and Verification*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2004. ISBN 0-387-95387-6; 978-0-387-95387-8. xviii + 320 pp. LCCN QA76.9.A25 G88 2002. DOI `10.1007/b97264`. {**214**}

[HA85] T. E. Hull and A. Abrham. Properly rounded variable precision square root. *ACM Transactions on Mathematical Software*, 11(3):229–237, September 1985. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.acm.org/pubs/citations/journals/toms/1985-11-3/p229-hull/`. DOI `10.1145/214408.214413`. {**216, 827**}

[HA86] T. E. Hull and A. Abrham. Variable precision exponential function. *ACM Transactions on Mathematical Software*, 12(2):79–91, June 1986. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.acm.org/pubs/citations/journals/toms/1986-12-2/p79-hull/`. DOI `10.1145/6497.6498`. {**827**}

[Hac84] Ian Hacking. Trial by number: Karl Pearson's chi-square test measured the fit between theory and reality, ushering in a new sort of decision making. *Science 84*, 5(9):69–70, November 1984. This issue is entitled *Century of the Sciences: 20 Discoveries That Changed Our Lives*. {**197**}

[Ham78] Hugo C. Hamaker. Miscellanea: Approximating the cumulative normal distribution and its inverse. *Applied Statistics*, 27(1):76–77, 1978. CODEN APSTAG. ISSN 0035-9254 (print), 1467-9876 (electronic). URL http://www.jstor.org/stable/2346231. DOI 10.2307/2346231. {**618**}

[Har70] V. C. Harris. An algorithm for finding the greatest common divisor. *Fibonacci Quarterly*, 8(1):102–103, February 1970. CODEN FIBQAU. ISSN 0015-0517. URL http://www.fq.math.ca/Scanned/8-1/harris1.pdf. {**184**}

[Har06] Laszlo Hars. Modular inverse algorithms without multiplications for cryptographic applications. *EURASIP Journal on Embedded Systems*, 2006:1–13, 2006. ISSN 1687-3955 (print), 1687-3963 (electronic). URL http://downloads.hindawi.com/journals/es/2006/032192.pdf. DOI 10.1155/ES/2006/32192. Article ID 32192. {**184**}

[Har09a] John Harrison. Decimal transcendentals via binary. In Bruguera et al. [BCDH09], pages 187–194. ISBN 0-7695-3670-0; 978-0-7695-3670-5. ISSN 1063-6889. LCCN QA76.6. URL http://www.ac.usc.es/arith19/. DOI 10.1109/ARITH.2009.31. {**826**}

[Har09b] John Harrison. Fast and accurate Bessel function computation. In Bruguera et al. [BCDH09], pages 104–113. ISBN 0-7695-3670-0; 978-0-7695-3670-5. ISSN 1063-6889. LCCN QA76.6. URL http://www.ac.usc.es/arith19/. DOI 10.1109/ARITH.2009.32. {**693, 716**}

[Has55] Cecil Hastings, Jr. *Approximations for Digital Computers*. The Rand series. Princeton University Press, Princeton, NJ, USA, 1955. viii + 201 pp. LCCN QA76 .H37. Assisted by Jeanne T. Hayward and James P. Wong, Jr. {**269, 600, 643, 827**}

[Hav03] Julian Havil. *Gamma: Exploring Euler's Constant*. Princeton University Press, Princeton, NJ, USA, 2003. ISBN 0-691-09983-9; 978-0-691-09983-5. xxiii + 266 pp. LCCN QA41 .H23 2003. {**59, 591**}

[Haw05] Stephen Hawking. *God Created the Integers: the Mathematical Breakthroughs that Changed History*. Running Press Book Publishers, Philadelphia, PA; London, UK, 2005. ISBN 0-7624-1922-9 (hardcover); 978-0-7624-1922-7 (hardcover). xiii + 1160 pp. URL http://www.perseusbooksgroup.com/runningpress/book_detail.jsp?isbn=0762419229. {**59, 299, 521**}

[HBF09] Victor Henner, Tatyana Belozerova, and Kyle Forinash. *Mathematical Methods in Physics: Partial Differential Equations, Fourier Series, and Special Functions*. A. K. Peters, Wellesley, MA, USA, 2009. ISBN 1-56881-335-X, 1-4398-6516-7 (e-book); 978-1-56881-335-6, 978-1-4398-6516-3 (e-book). xviii + 841 pp. LCCN QC20 .H487 2009. URL http://www.crcnetbase.com/isbn/9781568813356;. {**827**}

[HCGE17] Miguel Herrero-Collantes and Juan Carlos Garcia-Escartin. Quantum random number generators. *Reviews of Modern Physics*, 89 (1):015004:1–015004:48, January 2017. CODEN RMPHAT. ISSN 0034-6861 (print), 1538-4527 (electronic), 1539-0756. URL http://journals.aps.org/rmp/abstract/10.1103/RevModPhys.89.015004. DOI 10.1103/RevModPhys.89.015004. {**178**}

[HCL$^+$68] John F. Hart, E. W. Cheney, Charles L. Lawson, Hans J. Maehly, Charles K. Mesztenyi, John R. Rice, Henry G. Thatcher, Jr., and Christoph Witzgall. *Computer Approximations*. Robert E. Krieger Publishing Company, Huntington, NY, USA, 1968. ISBN 0-88275-642-7; 978-0-88275-642-4. x + 343 pp. LCCN QA 297 C64 1978. Reprinted 1978 with corrections. {**1, 270, 306, 521, 589, 593, 644, 693, 768, 827**}

[Hea05] Anthony C. Hearn. REDUCE: The first forty years. In Andreas Dolzmann, Andreas Seidl, and Thomas Sturm, editors, *Algorithmic Algebra and Logic: Proceedings of the A3L 2005, April 3–6, Passau, Germany, Conference in Honor of the 60th Birthday of Volker Weispfenning*, pages 19–24. Herstellung und Verlag: Books on Demand GmhH, Norderstedt, Germany, 2005. ISBN 3-8334-2669-1; 978-3-8334-2669-8. LCCN A155.7.E4 A39 2005. URL http://reduce-algebra.com/reduce40.pdf. {**28**}

[Hea09] Anthony C. Hearn. REDUCE is free software as of January 2009. *ACM Communications in Computer Algebra*, 43(1–2):15–16, March/June 2009. ISSN 1932-2232 (print), 1932-2240 (electronic). DOI 10.1145/1610296.1610299. {**28**}

[Hel06] Hal Hellman. *Great Feuds in Mathematics: Ten of the Liveliest Disputes Ever*. Wiley, New York, NY, USA, 2006. ISBN 0-471-64877-9 (cloth); 978-0-471-64877-2 (cloth). vi + 250 pp. LCCN QA21 .H45 2006. {**59**}

[Hen06] Doug Hensley. *Continued Fractions*. World Scientific Publishing, Singapore, 2006. ISBN 981-256-477-2; 978-981-256-477-1. xiii + 245 pp. LCCN QA295 .H377 2006. {**19**}

[HF98] Roger Herz-Fischler. *A Mathematical History of Golden Number*. Dover, New York, NY, USA, 1998. ISBN 0-486-40007-7; 978-0-486-40007-5. xxii + 195 pp. LCCN QA481.H47 1998. The *golden number*, or *golden ratio*, is $\phi = \frac{1}{2}(\sqrt{5}+1) \approx 1.618$. It is the last of the *big five* mathematical constants: e, i, π, γ, and ϕ. {**8, 14, 59, 577**}

[HF09] Frank E. Harris and J. G. Fripiat. Methods for incomplete Bessel function evaluation. *International Journal of Quantum Chemistry*, 109 (8):1728–1740, February 4, 2009. CODEN IJQCB2. ISSN 0020-7608 (print), 1097-461X (electronic). DOI 10.1002/qua.21972. {**693**}

[HFT94] T. E. Hull, Thomas F. Fairgrieve, and Ping Tak Peter Tang. Implementing complex elementary functions using exception handling. *ACM Transactions on Mathematical Software*, 20(2):215–244, June 1994. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL http://www.acm.org/pubs/citations/journals/toms/1994-20-2/p215-hull/. DOI 10.1145/178365.178404. See corrigenda [Ano94]. {**476, 996**}

[HFT97] T. E. Hull, Thomas F. Fairgrieve, and Ping Tak Peter Tang. Implementing the complex arcsine and arccosine functions using exception handling. *ACM Transactions on Mathematical Software*, 23(3):299–335, September 1997. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL http://www.acm.org/pubs/citations/journals/toms/1997-23-3/p299-hull/. DOI 10.1145/275323.275324. {**476**}

[HH80] Velma R. Huskey and Harry D. Huskey. Lady Lovelace and Charles Babbage. *Annals of the History of Computing*, 2(4):299–329, October/December 1980. CODEN AHCOE5. ISSN 0164-1239. URL http://dlib.computer.org/an/books/an1980/pdf/a4299.pdf; http://www.computer.org/annals/an1980/a4299abs.htm. DOI 10.1109/MAHC.1980.10042. {**568**}

[HHPM07] Andreas K. Heyne, Alice K. Heyne, Elena S. Pini, and Tahu Matheson. *Leonhard Euler: a Man to be Reckoned with*. Birkhäuser, Cambridge, MA, USA; Berlin, Germany; Basel, Switzerland, 2007. ISBN 3-7643-8332-1; 978-3-7643-8332-9. 45 pp. LCCN QA29.E8 H49 2007. {**59, 591**}

[Hil77] Geoffrey W. Hill. Algorithm 518: Incomplete Bessel function I_0. The von Mises distribution [S14]. *ACM Transactions on Mathematical Software*, 3(3):279–284, September 1977. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI 10.1145/355744.355753. {**693**}

[Hil81] Geoffrey W. Hill. Evaluation and inversion of the ratios of modified Bessel functions, $I_1(x)/I_0(x)$ and $I_{1.5}(x)/I_{0.5}(x)$. *ACM Transactions on Mathematical Software*, 7(2):199–208, June 1981. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI 10.1145/355945.355949. {**693**}

[HJ67a] Ian David Hill and S. A. Joyce. Algorithm 304: Normal curve integral. *Communications of the Association for Computing Machinery*, 10 (6):374–375, June 1967. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI 10.1145/363332.363411. See remarks [HJ67b, Ber68]. {**618, 999, 1015**}

[HJ67b] Ian David Hill and S. A. Joyce. Remarks on Algorithm 123 [S15]: Real error function, `ERF(x)`; Algorithm 180 [S15]: Error function — large *X*; Algorithm 181 [S15]: Complementary error function — large *X*; Algorithm 209 [S15]: Gauss; Algorithm 226 [S15]: Normal distribution function; Algorithm 272 [S15]: Procedure for the normal distribution functions; Algorithm 304 [S15]: Normal curve integral. *Communications of the Association for Computing Machinery*, 10(6):377–378, June 1967. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI 10.1145/363332.365433. See [Cyv64, Mac65, HJ67a]. {**618, 999, 1006, 1015, 1023**}

[HLB00] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Quad-double arithmetic: Algorithms, implementation, and application. Technical report 46996, Lawrence Berkeley National Laboratory, 1 Cycloton Rd, Berkeley, CA 94720, October 30, 2000. 28 pp. URL http://www.cs.berkeley.edu/~yozo/papers/LBNL-46996.ps.gz. {**366, 407, 777, 781**}

[HLD04] Wolfgang Hörmann, Josef Leydold, and Gerhard Derflinger. *Automatic Nonuniform Random Variate Generation*. Statistics and computing, 1431-8784. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2004. ISBN 3-540-40652-2; 978-3-540-40652-5. x + 441 pp. LCCN QA273 .H777 2004. DOI 10.1007/978-3-662-05946-3. {**196**}

[HLSZ07] Guillaume Hanrot, Vincent Lefèvre, Damien Stehlé, and Paul Zimmermann. Worst cases of a periodic function for large arguments. In Kornerup and Muller [KM07], pages 133–140. ISBN 0-7695-2854-6; 978-0-7695-2854-0. ISSN 1063-6889. LCCN QA76.9.C62. URL http://www.lirmm.fr/arith18/. DOI 10.1109/ARITH.2007.37. {**28**}

[Hoa81] C. A. R. Hoare. The Emperor's old clothes. *Communications of the Association for Computing Machinery*, 24(2):75–83, 1981. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI 10.1145/358549.358561. This is the 1980 ACM Turing Award Lecture, delivered at ACM'80, Nashville, Tennessee, October 27, 1980. {**963**}

[Hou81] David G. Hough. Applications of the proposed IEEE-754 standard for floating point arithmetic. *Computer*, 14(3):70–74, March 1981. CODEN CPTRB4. ISSN 0018-9162 (print), 1558-0814 (electronic). URL http://ieeexplore.ieee.org/document/1667288/. DOI 10.1109/C-M.1981.220381. See [IEEE85a]. {**63, 1016**}

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1990. ISBN 1-55860-069-8; 978-1-55860-069-0. xxviii + 594 pp. LCCN QA76.9.A73 P377 1990. {**103**}

[HP91] David G. Hough and Vern Paxson. Testbase: base conversion test program. World-Wide Web document, July 20, 1991. URL http://www.netlib.org/fp/testbase. See [PK91]. {**851**}

[HP94] John L. Hennessy and David A. Patterson. *Computer Organization and Design — The Hardware/Software Interface*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1994. ISBN 1-55860-281-X; 978-1-55860-281-6. xxiv + 648 pp. LCCN QA76.9 .C643 P37 1994. {**103**}

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1996. ISBN 1-55860-329-8; 978-1-55860-329-5. xxiii + 760 + A-77 + B-47 + C-26 + D-26 + E-13 + R-16 + I-14 pp. LCCN QA76.9.A73P377 1995. {**103**}

[HP97] John L. Hennessy and David A. Patterson. *Computer Organization: The Hardware/Software Interface*. Morgan Kaufmann Publishers, San Mateo, CA, USA, second edition, 1997. ISBN 1-55860-428-6 (hardcover), 1-55860-491-X (softcover); 978-1-55860-428-5 (hardcover), 978-1-55860-491-9 (softcover). 1000 pp. LCCN QA76.9.C643H46 1997. {**103**}

[HP03] John L. Hennessy and David A. Patterson. *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, third edition, 2003. ISBN 1-55860-596-7; 978-1-55860-596-1. xxi + 883 + A-87 + B-42 + C-1 + D-1 + E-1 + F-1 + G-1 + H-1 + I-1 + R-22 + I-44 pp. LCCN QA76.9.A73 P377 2003. URL http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-596-7; http://www.mkp.com/CA3. {**103**}

[HP04] John L. Hennessy and David A. Patterson. *Computer Organization: The Hardware/Software Interface*. Morgan Kaufmann Publishers, San Mateo, CA, USA, third edition, 2004. ISBN 1-55860-604-1; 978-1-55860-604-3. xvii + 621 pp. LCCN QA76.9.C643 H46 2004. {**103**}

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann/Elsevier, Waltham, MA, USA, fifth edition, 2012. ISBN 0-12-383872-X (paperback); 978-0-12-383872-8 (paperback). xxvii + 493 + 325 pp. LCCN QA76.9.A73 P377 2012. URL http://store.elsevier.com/product.jsp?isbn=9780123838728. With contributions by Krste Asanović, Jason D. Kabos, Robert P. Colwell, Thomas M. Conte, José Duato, Diana Franklin, David Goldberg, Norman P. Jouppi, Sheng Li, Naveen Muralimanohar, Gregory D. Peterson, Timothy M. Pinkston, Parthasarathy Ranganthan, David A. Wood, and Amr Zaky. {**103**}

[HPW90] Eldon R. Hansen, Merrell L. Patrick, and Richard L. C. Wang. Polynomial evaluation with scaling. *ACM Transactions on Mathematical Software*, 16(1):86–93, March 1990. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL http://www.acm.org/pubs/citations/journals/toms/1990-16-1/p86-hansen/. DOI 10.1145/77626.77633. {**89**}

[HPWW94] Brad Lee Holian, Ora E. Percus, Tony T. Warnock, and Paula A. Whitlock. Pseudorandom number generator for massively parallel molecular-dynamics simulations. *Physical Review E (Statistical physics, plasmas, fluids, and related interdisciplinary topics)*, 50(2):1607–1615, August 1994. CODEN PLEEE8. ISSN 1539-3755 (print), 1550-2376 (electronic). URL http://link.aps.org/doi/10.1103/PhysRevE.50.1607. DOI 10.1103/PhysRevE.50.1607. {**177**}

[HSH+09] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the Association for Computing Machinery*, 52(5):91–98, May 2009. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI 10.1145/1506409.1506429. {**208**}

[HTWG08] Anders Hejlsberg, Mads Togersen, Scott Wiltamuth, and Peter Golde, editors. *The C# Programming Language*. Addison-Wesley, Reading, MA, USA, third edition, 2008. ISBN 0-321-56299-2; 978-0-321-56299-9. xviii + 754 pp. LCCN QA76.73.C154 H45 2008. {**917**}

[HTWG11] Anders Hejlsberg, Mads Togersen, Scott Wiltamuth, and Peter Golde, editors. *The C# Programming Language*. Addison-Wesley, Reading, MA, USA, fourth edition, 2011. ISBN 0-321-74176-5; 978-0-321-74176-9. xviii + 844 pp. LCCN QA76.73.C154. URL http://www.pearsonhighered.com/program/Hejlsberg-C-Programming-Language-Covering-C-4-0-The-4th-Edition/PGM269050.html. {**917**}

[Hub11] Raymond Hubbard. The widespread misinterpretation of p-values as error probabilities. *Journal of Applied Statistics*, 38(11):2617–2626, November 2011. ISSN 0266-4763 (print), 1360-0532 (electronic). DOI 10.1080/02664763.2011.567245. {**196**}

[HW03] Peter Hellekalek and Stefan Wegenkittl. Empirical evidence concerning AES. *ACM Transactions on Modeling and Computer Simulation*, 13(4):322–333, October 2003. CODEN ATMCEZ. ISSN 1049-3301 (print), 1558-1195 (electronic). URL http://random.mat.sbg.ac.at/ftp/pub/publications/peter/aes_sub.ps; http://random.mat.sbg.ac.at/~peter/slides_YACC04.pdf. DOI 10.1145/945511.945515. {**178**}

[HWG04] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# programming language*. Addison-Wesley, Reading, MA, USA, 2004. ISBN 0-321-15491-6; 978-0-321-15491-0. xiv + 644 pp. LCCN QA76.76.C154 H45 2004. {**vii, 80, 917**}

[HWG06] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# programming language*. Addison-Wesley, Reading, MA, USA, second edition, 2006. ISBN 0-321-33443-4 (hardback); 978-0-321-33443-5 (hardback). xiv + 704 pp. LCCN QA76.73.C154 H45 2006. {**917**}

[IA6406] Intel Corporation, Santa Clara, CA, USA. *Intel Itanium Architecture Software Developer's Manual, Volume 1: Application Architecture*, January 2006. 250 pp. URL http://download.intel.com/design/Itanium/manuals/24531705.pdf. Order number 245317-005. {**241**}

[IBM70] IBM Corporation, San Jose, CA, USA. *A Programmer's Introduction to IBM System/360 Assembler Language*, August 1970. vii + 148 pp. URL http://bitsavers.org/pdf/ibm/360/asm/SC20-1646-6_int360asm_Aug70.pdf. Form number SC20-1646-6. {**928**}

[IBM84] IBM Corporation, Poughkeepsie, NY, USA. *High Accuracy Arithmetic*, January 1984. iii + 22 pp. Publication number SA22-7093-0, File number S370-01. {**964**}

[IBM06] IBM Corporation, San Jose, CA, USA. *Preliminary Decimal-Floating-Point Architecture*, November 2006. viii + 52 pp. URL http://publibz.boulder.ibm.com/epubs/pdf/a2322320.pdf. Form number SA23-2232-00. {**927, 963**}

[IBM07] IBM Corporation, San Jose, CA, USA. *Power Instruction Set Architecture: Preliminary Decimal Floating-Point Architecture*, July 2007. 52 pp. URL http://www.power.org/resources/downloads/PowerDFP.pdf. {**109**}

[IEE13] IEEE, editor. *Proceedings of the 21st IEEE Symposium on Computer Arithmetic, Austin, Texas, USA, 8–10 April 2013*. IEEE Computer Society Press, Silver Spring, MD, USA, 2013. ISBN 0-7695-4957-8; 978-0-7695-4957-6. ISSN 1063-6889. LCCN QA76.9.C62 S95 2013. {**1017, 1019**}

[IEE15] IEEE. *1788-2015 — IEEE Standard for Interval Arithmetic*. IEEE, New York, NY, USA, June 30, 2015. ISBN 0-7381-9721-1 (PDF), 0-7381-9720-3 (electronic); 978-0-7381-9721-0 (PDF), 978-0-7381-9720-3 (electronic). xiv + 79 pp. URL http://ieeexplore.ieee.org/servlet/opac?punumber=7140719. DOI 10.1109/IEEESTD.2015.7140721. Approved 11 June 2015 by IEEE-SA Standards Board. {**967**}

[IEEE85a] *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, NY, USA, August 12, 1985. ISBN 1-55937-653-8; 978-1-55937-653-2. 20 pp. URL http://standards.ieee.org/reading/ieee/std/busarch/754-1985.pdf. Revised 1990. A preliminary draft was published in the January 1980 issue of IEEE Computer, together with several companion articles [Cod81, Coo81b, Coo80, Coo81a, Hou81, Ste81a, Ste81b]. The final version was republished in [IEEE87, IEEE85b]. See also [WF82]. Also standardized as *IEC 60559 (1989-01) Binary floating-point arithmetic for microprocessor systems*. {**1, 63, 104, 827, 966, 1005, 1015, 1016, 1033, 1036**}

[IEEE85b] *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Std 754-1985. IEEE Computer Society Press, Silver Spring, MD, USA, 1985. 18 pp. See [IEEE85a]. {**63, 1016**}

[IEEE87] ANSI/IEEE Std 754-1985. an American National Standard: IEEE Standard for Binary Floating-Point Arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, February 1987. CODEN SINODQ. ISSN 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). URL http://portalparts.acm.org/30000/24686/fm/frontmatter.pdf. See [IEEE85a]. {**63, 104, 1016**}

[IEEE01] *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) System Interfaces, Issue 6*. IEEE, New York, NY, USA, 2001. ISBN 1-85912-247-7 (UK), 1-931624-07-0 (US), 0-7381-3094-4 (print), 0-7381-3010-9 (PDF), 0-7381-3129-6 (CD-ROM); 978-1-85912-247-1 (UK), 978-1-931624-07-7 (US), 978-0-7381-3094-1 (print), 978-0-7381-3010-1 (PDF), 978-0-7381-3129-0 (CD-ROM). xxx + 1690 pp. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992, Open Group Technical Standard Base Specifications, Issue 6. {**441**}

[IEEE08] *IEEE 754-2008, Standard for Floating-Point Arithmetic*. IEEE, New York, NY, USA, August 29, 2008. ISBN 0-7381-5753-8 (paper), 0-7381-5752-X (electronic); 978-0-7381-5753-5 (paper), 978-0-7381-5752-8 (electronic). 58 pp. URL http://en.wikipedia.org/wiki/IEEE_754-2008; http://ieeexplore.ieee.org/servlet/opac?punumber=4610933. DOI 10.1109/IEEESTD.2008.4610935. {**vii, 1, 104, 825, 966**}

[Ifr00] Georges Ifrah. *The Universal History of Numbers from Prehistory to the Invention of the Computer*. Wiley, New York, NY, USA, 2000. ISBN 0-471-37568-3; 978-0-471-37568-5. xxii + 633 pp. LCCN QA141.I3713 2000. Translated by David Bellos, E. F. Harding, Sophie Wood, and Ian Monk from the 1994 French original, *Histoire universelle des chiffres*. {**59**}

[Int85] Intel. *The iAPX 286 Programmer's Reference Manual*. Intel Corporation, Santa Clara, CA, USA, 1985. {**1028**}

[Inte06] Reference software implementation of the IEEE 754R decimal floating-point arithmetic. World-Wide Web document, 2006. URL `http://cache-www.intel.com/cd/00/00/29/43/294339_294339.pdf`. {**929**}

[Ioa05] John P. A. Ioannidis. Why most published research findings are false. *PLoS Medicine*, 2(8):696–701, August 2005. CODEN PMLEAC. ISSN 1549-1277 (print), 1549-1676 (electronic). URL `http://www.plosmedicine.org/article/info:doi/10.1371/journal.pmed.0020124`. DOI `10.1371/journal.pmed.0020124`. {**196**}

[ISO11] ISO. *ISO/IEC/IEEE 60559:2011 Information technology — Microprocessor Systems — Floating-Point arithmetic*. International Organization for Standardization, Geneva, Switzerland, 2011. 58 pp. URL `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57469`. {**vii, 1, 104, 825, 966**}

[Jab94] Aleksander Jablonski. Numerical evaluation of spherical Bessel functions of the first kind. *Journal of Computational Physics*, 111(2): 256–259, April 1994. CODEN JCTPAH. ISSN 0021-9991 (print), 1090-2716 (electronic). DOI `10.1006/jcph.1994.1060`. {**693**}

[Jac75] John David Jackson. *Classical Electrodynamics*. Wiley, New York, NY, USA, second edition, 1975. ISBN 0-471-43132-X; 978-0-471-43132-9. xxii + 848 pp. LCCN QC631 .J3 1975. {**627, 693**}

[Jac92] David Jacobson. Engineer's toolbox: Floating point in Mathematica. *Mathematica Journal*, 2(3):42–46, Summer 1992. ISSN 1047-5974 (print), 1097-1610 (electronic). URL `http://www.mathematica-journal.com/issue/v2i3/tutorials/toolbox/index.html`. This article describes the *significance arithmetic* used in Mathematica's software arbitrary-precision floating-point arithmetic. {**966**}

[Jam89] M. J. Jamieson. Rapidly converging iterative formulae for finding square roots and their computational efficiencies. *The Computer Journal*, 32(1):93–94, February 1989. CODEN CMPJA6. ISSN 0010-4620 (print), 1460-2067 (electronic). URL `http://www3.oup.co.uk/computer_journal/hdb/Volume_32/Issue_01/tiff/93.tif`; `http://www3.oup.co.uk/computer_journal/hdb/Volume_32/Issue_01/tiff/94.tif`. DOI `10.1093/comjnl/32.1.93`. This work generalizes the Pythagorean sums in [Dub83, MM83]. {**228, 1008, 1024**}

[Jam90] F. James. A review of pseudorandom number generators. *Computer Physics Communications*, 60(3):329–344, October 1990. CODEN CPHCBZ. ISSN 0010-4655 (print), 1879-2944 (electronic). URL `http://www.sciencedirect.com/science/article/pii/001046559090032V`. DOI `10.1016/0010-4655(90)90032-V`. {**177**}

[JD08] Alan Jeffrey and Hui-Hui Dai. *Handbook of Mathematical Formulas and Integrals*. Elsevier Academic Press, Amsterdam, The Netherlands, fourth edition, 2008. ISBN 0-12-374288-9; 978-0-12-374288-9. xlv + 541 pp. LCCN QA47 .J38 2008. {**58, 619**}

[Jea16] Claude-Pierre Jeannerod. A radix-independent error analysis of the Cornea–Harrison–Tang method. *ACM Transactions on Mathematical Software*, 42(3):19:1–19:20, May 2016. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/2824252`. {**463**}

[JEL68] E. Jahnke, Fritz Emde, and Friedrich Lösch. *Tafeln höherer Funktionen. (German) [Tables of higher functions]*. B. G. Teubner, Stuttgart, Germany, seventh edition, 1968. xii + 322 pp. LCCN QA55 .J32 1966. {**58**}

[JK77] Norman Lloyd Johnson and Samuel Kotz. *Urn Models and Their Application: An Approach to Modern Discrete Probability Theory*. Wiley series in probability and mathematical statistics. Wiley, New York, NY, USA, 1977. ISBN 0-471-44630-0; 978-0-471-44630-9. xiii + 402 pp. LCCN QA273 .J623. {**196**}

[JKB94] Norman Lloyd Johnson, Samuel Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Wiley series in probability and mathematical statistics. Wiley, New York, NY, USA, second edition, 1994. ISBN 0-471-58495-9 (vol. 1), 0-471-58494-0 (vol. 2); 978-0-471-58495-7 (vol. 1), 978-0-471-58494-0 (vol. 2). 761 (vol 1., est.), 752 (vol. 2, est.) pp. LCCN QA273.6 .J6 1994. Two volumes. {**196**}

[JKB97] Norman Lloyd Johnson, Samuel Kotz, and N. Balakrishnan. *Discrete Multivariate Distributions*. Wiley series in probability and statistics. Applied probability and statistics. Wiley, New York, NY, USA, 1997. ISBN 0-471-12844-9 (cloth); 978-0-471-12844-1 (cloth). xxii + 299 pp. LCCN QA273.6 .J617 1997. {**196**}

[JKK05] Norman Lloyd Johnson, Adrienne W. Kemp, and Samuel Kotz. *Univariate Discrete Distributions*. Wiley, New York, NY, USA, third edition, 2005. ISBN 0-471-27246-9; 978-0-471-27246-5. xix + 646 pp. LCCN QA273.6 .J64 2005. {**196**}

[JKLM17] Claude-Pierre Jeannerod, Peter Kornerup, Nicolas Louvet, and Jean-Michel Muller. Error bounds on complex floating-point multiplication with an FMA. *Mathematics of Computation*, 86(304):881–898, 2017. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). DOI `10.1090/mcom/3123`. {**458, 463**}

[JL12] U. D. Jentschura and E. Lötstedt. Numerical calculation of Bessel, Hankel and Airy functions. *Computer Physics Communications*, 183 (3):506–519, March 2012. CODEN CPHCBZ. ISSN 0010-4655 (print), 1879-2944 (electronic). URL `http://www.sciencedirect.com/science/article/pii/S0010465511003729`. DOI `10.1016/j.cpc.2011.11.010`. {**693**}

[JLM13a] Claude-Pierre Jeannerod, Nicolas Louvet, and Jean-Michel Muller. Further analysis of Kahan's algorithm for the accurate computation of 2×2 determinants. *Mathematics of Computation*, 82(284):2245–2264, 2013. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL `http://www.ams.org/journals/mcom/2013-82-284/S0025-5718-2013-02679-8`; `http://www.ams.org/journals/mcom/2013-82-284/S0025-5718-2013-02679-8/S0025-5718-2013-02679-8.pdf`. DOI `10.1090/S0025-5718-2013-02679-8`. {**463**}

[JLM13b] Claude-Pierre Jeannerod, Nicolas Louvet, and Jean-Michel Muller. On the componentwise accuracy of complex floating-point division with an FMA. In IEEE [IEE13], pages 83–90. ISBN 0-7695-4957-8; 978-0-7695-4957-6. ISSN 1063-6889. LCCN QA76.9.C62 S95 2013. DOI `10.1109/ARITH.2013.8`. {**463**}

[JLMP11] Claude-Pierre Jeannerod, Nicolas Louvet, Jean-Michel Muller, and Adrien Panhaleux. Midpoints and exact points of some algebraic functions in floating-point arithmetic. *IEEE Transactions on Computers*, 60(2):228–241, February 2011. CODEN ITCOB4. ISSN 0018-9340 (print), 1557-9956 (electronic). DOI `10.1109/TC.2010.144`. {**28**}

[JLMP16] Claude-Pierre Jeannerod, Nicolas Louvet, Jean-Michel Muller, and Antoine Plet. Sharp error bounds for complex floating-point inversion. *Numerical Algorithms*, 73(3):735–760, November 2016. CODEN NUALEG. ISSN 1017-1398 (print), 1572-9265 (electronic). URL `http://link.springer.com/article/10.1007/s11075-016-0115-x`. DOI `10.1007/s11075-016-0115-x`. {**463**}

[JP89] P. Johnstone and F. E. Petry. Higher radix floating point representations. In Miloš D. Ercegovac and Earl E. Swartzlander, Jr., editors, *Proceedings: 9th Symposium on Computer Arithmetic: September 6–8, 1989, Santa Monica, California, USA*, pages 128–135. IEEE Computer Society Press, Silver Spring, MD, USA, 1989. ISBN 0-8186-8963-3 (case), 0-8186-5963-7 (microfiche); 978-0-8186-8963-5 (case), 978-0-8186-5963-8 (microfiche). LCCN QA 76.9 C62 S95 1989. DOI 10.1109/ARITH.1989.72818. IEEE catalog number 89CH2757-3. {**964**}

[JPS07] Jon Jagger, Nigel Perry, and Peter Sestoft. *Annotated C# standard*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2007. ISBN 0-12-372511-9; 978-0-12-372511-0. xxiii + 825 pp. LCCN QA76.73.C154 J35 2007. {**102, 103, 917**}

[JT74] William B. Jones and Wolfgang J. Thron. Numerical stability in evaluating continued fractions. *Mathematics of Computation*, 28(127): 795–810, July 1974. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL http://www.jstor.org/stable/2005701. DOI 10.2307/2005701. {**13**}

[JT84] William B. Jones and Wolfgang J. Thron. *Continued Fractions: Analytic Theory and Applications*, volume 11 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, UK, 1984. ISBN 0-521-30231-5; 978-0-521-30231-9. xxviii + 428 pp. LCCN QA295 .J64 1984. {**19**}

[JW91] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report: ISO Pascal Standard*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., fourth edition, 1991. ISBN 0-387-97649-3, 3-540-97649-3; 978-0-387-97649-5, 978-3-540-97649-3. xvi + 266 pp. LCCN QA76.73.P2 J46 1991. DOI 10.1007/978-1-4612-4450-9. Revised by Andrew B. Mickel and James F. Miner. {**vii, 989**}

[Kah65] William M. Kahan. Further remarks on reducing truncation errors. *Communications of the Association for Computing Machinery*, 8(1):40, January 1965. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI 10.1145/363707.363723. {**353**}

[Kah83] William M. Kahan. Minimizing $q{*}m - n$. Technical report, Department of Mathematics and Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, CA, USA, March 1983. URL http://www.cs.berkeley.edu/~wkahan/testpi/nearpi.c. This important report discusses the use of continued fractions for finding worst cases for argument reduction. {**251, 265**}

[Kah87] William M. Kahan. Branch cuts for complex elementary functions or much ado about nothing's sign bit. In A. Iserles and M. J. D. Powell, editors, *The State of the Art in Numerical Analysis: Proceedings of the Joint IMA/SIAM Conference on the State of the Art in Numerical Analysis held at the University of Birmingham, 14–18 April 1986*, volume 9 of *Inst. Math. Appl. Conf. Ser. New Ser.*, pages 165–211. Oxford University Press, Oxford, UK, 1987. ISBN 0-19-853614-3; 978-0-19-853614-7. LCCN QA297 .S781 1987. {**69, 476, 482, 514**}

[Kah90] William M. Kahan. How Cray's arithmetic hurts scientific computation (and what might be done about it), June 14, 1990. URL http://754r.ucbtest.org/issues/cray-hurts.pdf. Manuscript prepared for the Cray User Group meeting in Toronto, Canada, April 10, 1990. {**953**}

[Kah04a] William M. Kahan. A logarithm too clever by half. World-Wide Web document, August 9, 2004. URL http://www.cs.berkeley.edu/~wkahan/LOG10HAF.TXT. {**81, 259**}

[Kah04b] William M. Kahan. On the cost of floating-point computation without extra-precise arithmetic. World-Wide Web document, November 20, 2004. URL http://www.cs.berkeley.edu/~wkahan/Qdrtcs.pdf. See [Bol09] for a proof of this algorithm for accurate computation of the discriminant needed for the solution of quadratic equations. {**472, 1000**}

[Kap99] Robert Kaplan. *The Nothing That Is: A Natural History of Zero*. Oxford University Press, Oxford, UK, 1999. ISBN 0-19-512842-7; 978-0-19-512842-0. xii + 225 pp. LCCN QA141 .K36 1999. {**59**}

[Kar85] Richard Karpinski. Paranoia: a floating-point benchmark. *Byte Magazine*, 10(2):223–235, February 1985. CODEN BYTEDJ. ISSN 0360-5280 (print), 1082-7838 (electronic). {**773**}

[Kat09] Victor J. Katz. *A History of Mathematics: An Introduction*. Addison-Wesley, Reading, MA, USA, third edition, 2009. ISBN 0-321-38700-7; 978-0-321-38700-4. xvi + 976 pp. LCCN QA21 .K33 2009. {**59**}

[KB67] Donald E. Knuth and Thomas J. Buckholtz. Computation of tangent, Euler, and Bernoulli numbers. *Mathematics of Computation*, 21 (100):663–688, October 1967. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL http://www.jstor.org/stable/2005010. DOI 10.2307/2005010. {**574**}

[KBJ00] Samuel Kotz, N. Balakrishnan, and Norman Lloyd Johnson. *Continuous Multivariate Distributions: Volume 1: Models and Applications*. Wiley series in probability and statistics. Wiley, New York, NY, USA, second edition, 2000. ISBN 0-471-18387-3 (cloth); 978-0-471-18387-7 (cloth). xxii + 722 pp. LCCN QA273.6 .K68 2000. {**196**}

[KD98] William M. Kahan and Joseph D. Darcy. How Java's floating-point hurts everyone everywhere. Technical report, Department of Mathematics and Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, CA, USA, June 18, 1998. 80 pp. URL http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf. {**4, 64, 105**}

[KE97] Aleksandr Yakovlevich Khinchin and Herbert Eagle. *Continued Fractions*. Dover, New York, NY, USA, 1997. ISBN 0-486-69630-8 (paperback); 978-0-486-69630-0 (paperback). xi + 95 pp. LCCN QA295 .K513 1997. {**19**}

[Ker81] Brian W. Kernighan. Why Pascal is not my favorite programming language. Computer Science Report 100, AT&T Bell Laboratories, Murray Hill, NJ, USA, July 1981. URL http://cm.bell-labs.com/cm/cs/cstr/100.ps.gz. Published in [Ker84]. See also [WSH77]. {**989, 1018, 1037**}

[Ker84] Brian W. Kernighan. Why Pascal is not my favorite programming language. In Alan R. Feuer and Narain Gehani, editors, *Comparing and Assessing Programming Languages: Ada, C, and Pascal*, Prentice-Hall software series, pages 170–186. Prentice-Hall, Upper Saddle River, NJ, USA, 1984. ISBN 0-13-154840-9 (paperback), 0-13-154857-3 (hardcover); 978-0-13-154840-4 (paperback), 978-0-13-154857-2 (hardcover). LCCN QA76.73.A35 C66 1984. See also [WSH77, Ker81]. {**1018, 1037**}

[KGD13] Edin Kadric, Paul Gurniak, and André DeHon. Accurate parallel floating-point accumulation. In IEEE [IEE13], pages 153–162. ISBN 0-7695-4957-8; 978-0-7695-4957-6. ISSN 1063-6889. LCCN QA76.9.C62 S95 2013. DOI `10.1109/ARITH.2013.19`. {**385**}

[KGD16] Edin Kadric, Paul Gurniak, and André DeHon. Accurate parallel floating-point accumulation. *IEEE Transactions on Computers*, 65(11): 3224–3238, November 2016. CODEN ITCOB4. ISSN 0018-9340 (print), 1557-9956 (electronic). DOI `10.1109/TC.2016.2532874`. {**385**}

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Upper Saddle River, NJ, USA, 1992. ISBN 0-13-590472-2; 978-0-13-590472-5. LCCN QA76.8.M52 K37 1992. {**73, 146**}

[Khr08] Sergey V. Khrushchev. *Orthogonal Polynomials and Continued Fractions: from Euler's Point of View*, volume 122 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, UK, 2008. ISBN 0-521-85419-9 (hardcover); 978-0-521-85419-1 (hardcover). xvi + 478 pp. LCCN QA404.5 .K47 2008. {**19, 59**}

[Kin21] Louis Vessot King. On some new formulae for the numerical calculation of the mutual induction of coaxial circles. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 100(702):60–66, October 4, 1921. ISSN 0950-1207 (print), 2053-9150 (electronic). URL `http://www.jstor.org/stable/93861`. DOI `10.1098/rspa.1921.0070`. This is the first known publication of the AGM method, discovered by the author in 1913, for computing Jacobian elliptic functions. See also [Kin24, Kin07]. {**663**}

[Kin24] Louis Vessot King. *On the Direct Numerical Calculation of Elliptic Functions and Integrals*. Cambridge University Press, Cambridge, UK, 1924. viii + 42 pp. LCCN QA343. {**663, 1019**}

[Kin07] Louis Vessot King. *On the Direct Numerical Calculation of Elliptic Functions and Integrals*. Mellon Press, 2007. ISBN 1-4067-4226-0; 978-1-4067-4226-8. 56 pp. {**663, 1019**}

[KIR$^+$07] Victor J. Katz, Annette Imhausen, Eleanor Robson, Joseph W. Dauben, Kim Plofker, and J. Lennart Berggren, editors. *The Mathematics of Egypt, Mesopotamia, China, India, and Islam: a Sourcebook*. Princeton University Press, Princeton, NJ, USA, 2007. ISBN 0-691-11485-4 (hardcover); 978-0-691-11485-9 (hardcover). xiv + 685 pp. LCCN QA22 .M3735 2007. {**59**}

[KK67] Melvin Klerer and Granino A. Korn, editors. *Digital Computer User's Handbook*. McGraw-Hill, New York, NY, USA, 1967. LCCN QA76.5 .K524. {**947, 978**}

[KL85] William M. Kahan and E. LeBlanc. Anomalies in the IBM ACRITH package. In Kai Hwang, editor, *Proceedings: 7th Symposium on Computer Arithmetic, June 4–6, 1985, University of Illinois, Urbana, Illinois*, pages 322–331. IEEE Computer Society Press, Silver Spring, MD, USA, 1985. ISBN 0-8186-0632-0 (paperback), 0-8186-8632-4 (hard), 0-8186-4632-2 (microfiche); 978-0-8186-0632-8 (paperback), 978-0-8186-8632-0 (hard), 978-0-8186-4632-4 (microfiche). LCCN QA76.9.C62 S95 1985. DOI `10.1109/ARITH.1985.6158956`. IEEE catalog number 85CH2146-9. IEEE Computer Society order number 632. {**967**}

[Kla05] Robert Klarer. Decimal types for C++: Second draft. Report C22/WG21/N1839 J16/05-0099, IBM Canada, Ltd., Toronto, ON, Canada, June 24, 2005. URL `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1839.html`. {**109, 875, 928**}

[KLL$^+$10] Peter Kornerup, Christoph Lauter, Vincent Lefèvre, Nicolas Louvet, and Jean-Michel Muller. Computing correctly rounded integer powers in floating-point arithmetic. *ACM Transactions on Mathematical Software*, 37(1):4:1–4:23, January 2010. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/1644001.1644005`. {**28, 420**}

[KLLM12] Peter Kornerup, Vincent Lefèvre, Nicolas Louvet, and Jean-Michel Muller. On the computation of correctly rounded sums. *IEEE Transactions on Computers*, 61(3):289–298, March 2012. CODEN ITCOB4. ISSN 0018-9340 (print), 1557-9956 (electronic). DOI `10.1109/TC.2011.27`. {**385**}

[KM91] Peter Kornerup and David W. Matula, editors. *Proceedings: 10th IEEE Symposium on Computer Arithmetic: June 26–28, 1991, Grenoble, France*. IEEE Computer Society Press, Silver Spring, MD, USA, 1991. ISBN 0-8186-9151-4 (case), 0-8186-6151-8 (microfiche), 0-7803-0187-0 (library binding); 978-0-8186-9151-5 (case), 978-0-8186-6151-8 (microfiche), 978-0-7803-0187-0 (library binding). LCCN QA76.9.C62 S95 1991. IEEE catalog number 91CH3015-5. {**1002, 1019, 1025, 1028**}

[KM95] Simon Knowles and William H. McAllister, editors. *Proceedings of the 12th Symposium on Computer Arithmetic, July 19–21, 1995, Bath, England*. IEEE Computer Society Press, Silver Spring, MD, USA, 1995. ISBN 0-8186-7089-4 (paperback), 0-8186-7089-4 (case), 0-8186-7149-1 (microfiche), 0-8186-7089-4 (softbound), 0-7803-2949-X (casebound); 978-0-8186-7089-3 (paperback), 978-0-8186-7089-3 (case), 978-0-8186-7149-4 (microfiche), 978-0-8186-7089-3 (softbound), 978-0-7803-2949-2 (casebound). LCCN QA 76.9 C62 S95 1995. {**1009, 1021**}

[KM07] Peter Kornerup and Jean-Michel Muller, editors. *Proceedings of the 18th IEEE Symposium on Computer Arithmetic, June 25–27, 2007, Montpellier, France*. IEEE Computer Society Press, Silver Spring, MD, USA, 2007. ISBN 0-7695-2854-6; 978-0-7695-2854-0. ISSN 1063-6889. LCCN QA76.9.C62. URL `http://www.lirmm.fr/arith18/`. {**1000, 1002, 1015, 1022, 1035**}

[KM10] Peter Kornerup and David W. Matula. *Finite Precision Number Systems and Arithmetic*, volume 133 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, UK, 2010. ISBN 0-521-76135-2 (hardcover); 978-0-521-76135-2 (hardcover). xv + 699 pp. LCCN QA248 .K627 2010. {**978**}

[KMN89] David Kahaner, Cleve B. Moler, and Stephen Nash. *Numerical Methods and Software*. Prentice-Hall, Upper Saddle River, NJ, USA, 1989. ISBN 0-13-627258-4; 978-0-13-627258-8. xii + 495 pp. LCCN TA345 .K341 1989. {**202, 299**}

[Kna92] Anthony W. Knapp. *Elliptic Curves*, volume 40 of *Mathematical notes*. Princeton University Press, Princeton, NJ, USA, 1992. ISBN 0-691-08559-5 (paperback); 978-0-691-08559-3 (paperback). xv + 427 pp. LCCN QA567.2.E44 K53 1992. {**222**}

[Knö91] A. Knöfel. Fast hardware units for the computation of accurate dot products. In Kornerup and Matula [KM91], pages 70–74. ISBN 0-8186-9151-4 (case), 0-8186-6151-8 (microfiche), 0-7803-0187-0 (library binding); 978-0-8186-9151-5 (case), 978-0-8186-6151-8 (microfiche), 978-0-7803-0187-0 (library binding). LCCN QA76.9.C62 S95 1991. DOI `10.1109/ARITH.1991.145536`. IEEE catalog number 91CH3015-5. {**385**}

[Knu74] Donald E. Knuth. Structured programming with **go to** statements. *ACM Computing Surveys*, 6(4):261–301, December 1974. CODEN CMSVAN. ISSN 0360-0300 (print), 1557-7341 (electronic). DOI `10.1145/356635.356640`. Reprinted with revisions in *Current Trends in Programming Methodology*, Raymond T. Yeh, ed., **1** (Englewood Cliffs, NJ: Prentice-Hall, 1977), 140–194; *Classics in Software Engineering*, Edward Nash Yourdon, ed. (New York: Yourdon Press, 1979), 259–321. Reprinted with "final" revisions in [Knu92, pp. 17–89]. This paper is a response to [Dij68]. {**1007**}

[Knu85] Donald E. Knuth. Deciphering a linear congruential encryption. *IEEE Transactions on Information Theory*, IT-31(1):49–52, January 1985. CODEN IETTAW. ISSN 0018-9448 (print), 1557-9654 (electronic). DOI `10.1109/TIT.1985.1056997`. Russian translation, to appear. {**207**}

[Knu90] Donald E. Knuth. A simple program whose proof isn't. In Feijen et al. [FvGM90], chapter 27, pages 233–242. ISBN 0-387-97299-4; 978-0-387-97299-2. LCCN QA76 .B326 1990. DOI `10.1007/978-1-4612-4476-9_28`. This paper discusses the algorithm used in TeX for converting between decimal and scaled fixed-point binary values, and for guaranteeing a minimum number of digits in the decimal representation. See also [Cli90, Cli04] for decimal to binary conversion, [SW90, SW04] for binary to decimal conversion, and [Gri90] for an alternate proof of Knuth's algorithm. {**895, 995, 1004, 1011, 1013, 1034**}

[Knu92] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992. ISBN 0-937073-80-6 (paperback), 0-937073-81-4 (hardcover); 978-0-937073-80-3 (paperback), 978-0-937073-81-0 (hardcover). xiii + 368 pp. LCCN QA76.6 .K644 1992. {**1020**}

[Knu97] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, third edition, 1997. ISBN 0-201-89684-2; 978-0-201-89684-8. xiii + 762 pp. LCCN QA76.6 .K64 1997. {**89, 170, 182, 184, 186, 188, 200, 214, 366, 416**}

[Knu99] Donald E. Knuth. *MMIXware: a RISC computer for the third millennium*, volume 1750 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1999. ISBN 3-540-66938-8 (softcover); 978-3-540-66938-8 (softcover). ISSN 0302-9743 (print), 1611-3349 (electronic). viii + 550 pp. LCCN QA76.9.A73 K62 1999. DOI `10.1007/3-540-46611-8`. {**104**}

[Kod07] Masao Kodama. Remark on Algorithm 644. *ACM Transactions on Mathematical Software*, 33(4):28:1–28:3, August 2007. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/1268776.1268783`. See [Amo86, Amo90, Amo95]. {**693, 996**}

[Kod08] Masao Kodama. Algorithm 877: A subroutine package for cylindrical functions of complex order and nonnegative argument. *ACM Transactions on Mathematical Software*, 34(4):22:1–22:21, July 2008. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/1377596.1377602`. {**693**}

[Kor02] Israel Koren. *Computer Arithmetic Algorithms*. A. K. Peters, Wellesley, MA, USA, second edition, 2002. ISBN 1-56881-160-8; 978-1-56881-160-4. xv + 281 pp. LCCN QA76.9.C62 K67. {**68, 881, 978**}

[KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, Reading, MA, USA, 1999. ISBN 0-201-61586-X; 978-0-201-61586-9. xii + 267 pp. LCCN QA76.6 .K48 1999. URL `http://cm.bell-labs.com/cm/cs/tpop/code.html; http://tpop.awl.com`. {**6**}

[Kra14] Ilia Krasikov. Approximations for the Bessel and Airy functions with an explicit error term. *LMS Journal of Computation and Mathematics*, 17(1):209–225, 2014. ISSN 1461-1570. DOI `10.1112/S1461157013000351`. {**693**}

[Kro10] Kirk L. Kroeker. News: Celebrating the legacy of PLATO. *Communications of the Association for Computing Machinery*, 53(8):19–20, August 2010. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/1787234.1787261`. The PLATO system on time-shared CDC mainframes was the first major effort at developing online courseware. {**949**}

[KSS95] Marek A. Kowalski, Krzysztof A. Sikorski, and Frank Stenger. *Selected Topics in Approximation and Computation*. Oxford University Press, Oxford, UK, 1995. ISBN 0-19-508059-9; 978-0-19-508059-9. xiv + 349 pp. URL `http://site.ebrary.com/lib/utah/Doc?id=10087215`. {**733**}

[KT99] Eugene Eric Kim and Betty Alexandra Toole. Ada and the first computer: The collaboration between ada, countess of lovelace, and computer pioneer Charles Babbage resulted in a landmark publication that described how to program the world's first computer. *Scientific American*, 280(5):76–81, May 1999. CODEN SCAMAC. ISSN 0036-8733 (print), 1946-7087 (electronic). URL `http://www.nature.com/scientificamerican/journal/v280/n5/pdf/scientificamerican0599-76.pdf`. DOI `10.1038/scientificamerican0599-76`. {**568**}

[Kul13] Ulrich Kulisch. *Computer Arithmetic and Validity*, volume 33 of *De Gruyter studies in mathematics*. Walter de Gruyter, Berlin, Germany, second edition, 2013. ISBN 3-11-030173-3, 3-11-030179-2 (e-book), 3-11-030180-6 (set); 978-3-11-030173-1, 978-3-11-030179-3 (e-book), 978-3-11-030180-9 (set). ISSN 0179-0986. xxii + 434 pp. LCCN QA76.9.C62 K853 2013. DOI `10.1515/9783110301793`. {**978**}

[KW96] Chiang Kao and J. Y. Wong. An exhaustive analysis of prime modulus multiplicative congruential random number generators with modulus smaller than 2^{15}. *Journal of Statistical Computation and Simulation*, 54(1–3):29–35, 1996. CODEN JSCSAJ. ISSN 0094-9655 (print), 1026-7778 (electronic), 1563-5163. URL `http://www.tandfonline.com/doi/abs/10.1080/00949659608811717`. DOI `10.1080/00949659608811717`. {**170**}

[Lan64] Cornelius Lanczos. A precision approximation of the gamma function. *Journal of the Society for Industrial and Applied Mathematics: Series B, Numerical Analysis*, 1(1):86–96, 1964. ISSN 0887-459X (print), 1095-7170 (electronic). URL `http://www.jstor.org/stable/2949767`. DOI `10.1137/0701008`. {**521, 536**}

[Lan87] Eberhard Lange. Implementation and test of the ACRITH facility in a System/370. *IEEE Transactions on Computers*, C-36(9):1088–1096, September 1987. CODEN ITCOB4. ISSN 0018-9340 (print), 1557-9956 (electronic). URL `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5009539`. DOI `10.1109/TC.1987.5009539`. {**967**}

[Lap08] Michel L. Lapidus. *In Search of the Riemann Zeros: Strings, Fractal Membranes and Noncommutative Spacetimes*. American Mathematical Society, Providence, RI, USA, 2008. ISBN 0-8218-4222-6; 978-0-8218-4222-5. xxix + 558 pp. LCCN QA333 .L37 2008. {**303, 521, 579**}

[LAS$^+$95] Tom Lynch, Ahmed Ahmed, Michael J. Schulte, Tom Callaway, and Robert Tisdale. The K5 transcendental functions. In Knowles and McAllister [KM95], pages 163–171. ISBN 0-8186-7089-4 (paperback), 0-8186-7089-4 (case), 0-8186-7149-1 (microfiche), 0-8186-7089-4 (softbound), 0-7803-2949-X (casebound); 978-0-8186-7089-3 (paperback), 978-0-8186-7089-3 (case), 978-0-8186-7149-4 (microfiche), 978-0-8186-7089-3 (softbound), 978-0-7803-2949-2 (casebound). LCCN QA 76.9 C62 S95 1995. URL `http://mesa.ece.wisc.edu/publications/cp_1995-04.pdf; http://www.acsel-lab.com/arithmetic/arith12/papers/ARITH12_Lynch.pdf`. DOI `10.1109/ARITH.1995.465368`. The K5 is one of AMD's IA-32-compatible microprocessors. {**293**}

[Lau08] Detleff Laugwitz. *Bernhard Riemann 1826–1866: Turning Points in the Conception of Mathematics*. Modern Birkhäuser classics. Birkhäuser Boston Inc., Cambridge, MA, USA, 2008. ISBN 0-8176-4776-7 (paperback), 0-8176-4777-5; 978-0-8176-4776-6 (paperback), 978-0-8176-4777-3. xvi + 357 pp. LCCN QA29 R425 L3813 2008. URL `http://www.gbv.de/dms/bowker/toc/9780817647766`. DOI `10.1007/978-0-8176-4777-3`. Translated by Abe Shenitzer from the 1996 German original, *Bernhard Riemann 1826–1866: Wendepunkte in der Auffassung der Mathematik*. {**579, 590**}

[Law89] Derek F. Lawden. *Elliptic Functions and Applications*, volume 80 of *Applied mathematical sciences*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1989. ISBN 0-387-96965-9; 978-0-387-96965-7. xiv + 334 pp. LCCN QA1 .A647 vol. 80; QA343. DOI `10.1007/978-1-4757-3980-0`. {**619, 627, 654, 682, 688**}

[Law06] Averill M. Law. *Simulation Modeling and Analysis*. McGraw-Hill series in industrial engineering and management science. McGraw-Hill, New York, NY, USA, fourth edition, 2006. ISBN 0-07-298843-6 (hardcover), 0-07-125519-2 (paperback), 0-07-329441-1, 0-07-110336-8, 0-07-110051-2; 978-0-07-298843-7 (hardcover), 978-0-07-125519-6 (paperback), 978-0-07-329441-4, 978-0-07-110336-7, 978-0-07-110051-9. xix + 768 pp. LCCN QA76.9.C65 L38 2005. {**196**}

[LB92] J. Lund and K. L. Bowers. *Sinc Methods for Quadrature and Differential Equations*. SIAM (Society for Industrial and Applied Mathematics), Philadelphia, PA, USA, 1992. ISBN 0-89871-298-X; 978-0-89871-298-8. x + 304 pp. LCCN QA372 .L86 1992. {**733**}

[LBC93] Pierre L'Ecuyer, François Blouin, and Raymond Couture. A search for good multiple recursive random number generators. *ACM Transactions on Modeling and Computer Simulation*, 3(2):87–98, April 1993. CODEN ATMCEZ. ISSN 1049-3301 (print), 1558-1195 (electronic). DOI `10.1145/169702.169698`. {**170**}

[LCM16] Cedric Lichtenau, Steven Carlough, and Silvia Melitta Mueller. Quad precision floating point on the IBM z13. In Montuschi et al. [MSH$^+$16], pages 87–94. ISBN 1-5090-1615-5; 978-1-5090-1615-0. ISSN 1063-6889. LCCN QA76.9.C62 S95 2016. URL `http://ieeexplore.ieee.org/servlet/opac?punumber=7562813`. DOI `10.1109/ARITH.2016.26`. {**928**}

[LDB$^+$00] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, J. Iskandar, William M. Kahan, Anil Kapur, M. C. Martin, T. Tung, and D. J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. LAPACK Working Note 149, Department of Computer Science, University of Tennessee, Knoxville, Knoxville, TN 37996, USA, October 2000. URL `http://www.netlib.org/lapack/lawnspdf/lawn149.pdf`. Appendix B contains an improved version of Smith's algorithm for complex division [Smi62], using scaling to avoid premature overflow and underflow. {**452, 1032**}

[LE80] Henry M. Levy and Richard H. Eckhouse, Jr. *Computer Programming and Architecture—the VAX-11*. Digital Press, Bedford, MA, USA, 1980. ISBN 0-932376-07-X; 978-0-932376-07-7. xxi + 407 pp. LCCN QA76.8 .V37 L48 1980. {**956**}

[L'E96] Pierre L'Ecuyer. Maximally equidistributed combined Tausworthe generators. *Mathematics of Computation*, 65(213):203–213, January 1996. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL `http://www.ams.org/jourcgi/jour-pbprocess?fn=110&arg1=S0025-5718-96-00696-5&u=/mcom/1996-65-213/`. DOI `10.1090/S0025-5718-96-00696-5`. {**177**}

[L'E98] Pierre L'Ecuyer. Random number generation. In Banks [Ban98], chapter 4, pages 93–137. ISBN 0-471-13403-1 (hardcover); 978-0-471-13403-9 (hardcover). LCCN T57.62 .H37 1998. DOI `10.1002/9780470172445.ch4`. {**196**}

[Lef05] Vincent Lefèvre. New results on the distance between a segment and Z^2. Application to the exact rounding. In Montuschi and Schwarz [MS05], pages 68–75. ISBN 0-7695-2366-8; 978-0-7695-2366-8. LCCN QA76.9.C62 .S95 2005. URL `http://arith17.polito.it/final/paper-147.pdf`. DOI `10.1109/ARITH.2005.32`. {**28**}

[Lef16] Vincent Lefèvre. Correctly rounded arbitrary-precision floating-point summation. In Montuschi et al. [MSH$^+$16], pages 71–78. ISBN 1-5090-1615-5; 978-1-5090-1615-0. ISSN 1063-6889. LCCN QA76.9.C62 S95 2016. URL `http://ieeexplore.ieee.org/servlet/opac?punumber=7562813`. DOI `10.1109/ARITH.2016.9`. {**385**}

[Leh51] D. H. Lehmer. Mathematical methods in large-scale computing units. In Anonymous, editor, *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery, Harvard University Computation Laboratory, 13–16 September 1949*, volume 26 of *Annals of the Computation Laboratory of Harvard University*, pages 141–146. Harvard University Press, Cambridge, MA, USA, 1951. LCCN QA75 .S9 1949. URL `http://archive.org/stream/proceedings_of_a_second_symposium_on_large-scale_/Proceedings_of_a_Second_Symposium_on_Large-Scale_Digital_Calculating_Machinery_Sep49_djvu.txt`. {**169**}

[Lev09] Thomas Levenson. *Newton and the Counterfeiter: the Unknown Detective Career of the World's Greatest Scientist*. Houghton Mifflin Harcourt, Boston, MA, USA, 2009. ISBN 0-15-101278-4; 978-0-15-101278-7. xii + 318 pp. LCCN Q143.N495 L48 2009. {**8**}

[Lew75] John Gregg Lewis. Certification of "Algorithm 349: Polygamma functions with arbitrary precision". *ACM Transactions on Mathematical Software*, 1(4):380–381, December 1975. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/355656.355664`. See [TS69]. {**1035**}

[LHKK79] Charles L. Lawson, Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/355841.355847`. {**223**}

[Lia99] Sheng Liang. *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, Reading, MA, USA, 1999. ISBN 0-201-32577-2; 978-0-201-32577-5. xiv + 303 pp. LCCN QA76.38 .L53 1999. {**979, 983**}

[Lin81] Seppo Linnainmaa. Software for doubled-precision floating-point computations. *ACM Transactions on Mathematical Software*, 7(3): 272–283, September 1981. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/355958.355960`. {**370**}

[Lin89] Jinn Tyan Lin. Approximating the normal tail probability and its inverse for use on a pocket calculator. *Applied Statistics*, 38(1): 69–70, 1989. CODEN APSTAG. ISSN 0035-9254 (print), 1467-9876 (electronic). URL `http://www.jstor.org/stable/2347681`. DOI `10.2307/2347681`. {**618**}

[Lin90] Jinn Tyan Lin. Miscellanea: A simpler logistic approximation to the normal tail probability and its inverse. *Applied Statistics*, 39(2): 255–257, 1990. CODEN APSTAG. ISSN 0035-9254 (print), 1467-9876 (electronic). URL `http://www.jstor.org/stable/2347764`. DOI `10.2307/2347764`. {**618**}

[Lio96] John Lions. *Lions' Commentary on UNIX 6th Edition, with Source Code*. Computer classics revisited. Peer-to-Peer Communications, San Jose, CA 95164-0218, USA, 1996. ISBN 1-57398-013-7; 978-1-57398-013-5. 254 pp. URL `http://www.peer-to-peer.com/catalog/opsrc/lions.html`. With forewords by Dennis M. Ritchie and Ken Thompson. Prefatory notes by Peter H. Salus and Michael Tilson; a historical note by Peter H. Salus; and appreciations by Greg Rose, Mike O'Dell, Berny Goodheart, Peter Collinson, and Peter Reintjes. Originally circulated as two restricted-release volumes: "UNIX Operating System Source Code Level Six", and "A Commentary on the UNIX Operating System". This document was the first published detailed analysis of the entire source code of an operating-system kernel. {**850**}

[Liu87] Zhi-Shun Alex Liu. Berkeley elementary function test suite: Research project. Master of Science, Plan II, Computer Science Division, Department of Electrical Engineering and Computer Science, Univerity of California at Berkeley, Berkeley, CA, USA, December 1987. {**774**}

[Liu88] Zhi-Shun Alex Liu. Berkeley elementary function test suite. Technical report, Computer Science Division, Department of Electrical Engineering and Computer Science, Univerity of California at Berkeley, Berkeley, CA, USA, December 30, 1988. ii + 59 pp. URL `http://www.netlib.org/fp/ucbtest.tgz`; `http://www.ucbtest.org/zaliu-papers/zaliu-beef-doc.pdf`. {**774**}

[Liv02] Mario Livio. *The Golden Ratio: The Story of Phi, the World's Most Astonishing Number*. Broadway Books, New York, NY, USA, 2002. ISBN 0-7679-0815-5; 978-0-7679-0815-3. viii + 294 pp. LCCN QA466 .L58 2002. The *golden ratio* is $\phi = \frac{1}{2}(\sqrt{5}+1) \approx 1.618$. It is the last of the *big five* mathematical constants: e, i, π, γ, and ϕ. {**8, 14, 59, 577**}

[Liv05] Mario Livio. *The Equation that Couldn't be Solved: How Mathematical Genius Discovered the Language of Symmetry*. Simon and Schuster, New York, NY, USA, 2005. ISBN 0-7432-5820-7; 978-0-7432-5820-3. x + 353 pp. LCCN QA174.2 .L58 2005. {**7**}

[LL73] G. P. Learmonth and P. A. W. Lewis. Naval Postgraduate School random number generator package LLRANDOM. Report NP555LW73061A, Naval Postgraduate School, Monterey, CA, USA, 1973. The shuffling algorithm proposed in this report does *not* lengthen the period, and only marginally reduces the lattice structure of linear congruential generators, despite the apparently tiny difference with the [BD76] algorithm: see [Bay90] for a comparison, both mathematical, and graphical. {**179, 997, 998**}

[LL01] Eli Levin and Doran S. Lubinsky. *Orthogonal Polynomials for Exponential Weights*, volume 4 of *CMS books in mathematics*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2001. ISBN 0-387-98941-2 (hardcover); 978-0-387-98941-9 (hardcover). xi + 476 pp. LCCN QA404.5 .L48 2001. DOI `10.1007/978-1-4613-0201-8`. {**59**}

[LL07] Philippe Langlois and Nicolas Louvet. How to ensure a faithful polynomial evaluation with the compensated Horner algorithm. In Kornerup and Muller [KM07], pages 141–149. ISBN 0-7695-2854-6; 978-0-7695-2854-0. ISSN 1063-6889. LCCN QA76.9.C62. URL `http://www.lirmm.fr/arith18/`. DOI `10.1109/ARITH.2007.21`. {**89**}

[LM01] Vincent Lefèvre and Jean-Michel Muller. Worst cases for correct rounding of the elementary functions in double precision. In N. Burgess and L. Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic: ARITH-15 2001: proceedings: Vail, Colorado, 11–13 June, 2001*, pages 111–118. IEEE Computer Society Press, Silver Spring, MD, USA, 2001. ISBN 0-7695-1150-3; 0-7695-1152-X; 978-0-7695-1150-4; 978-0-7695-1152-8. ISSN 1063-6889. LCCN QA76.9.C62 S95 2001. DOI `10.1109/ARITH.2001.930110`. IEEE order number PR01150. {**28**}

[LM03a] Vincent Lefèvre and Jean-Michel Muller. The Table Maker's Dilemma: our search for worst cases. World-Wide Web software project archive, October 28, 2003. URL `http://perso.ens-lyon.fr/jean-michel.muller/Intro-to-TMD.htm`. {**28**}

[LM03b] Vincent Lefèvre and Jean-Michel Muller. Worst cases for correct rounding for the elementary functions in double precision. Technical report, INRIA, Projet Spaces, LORIA, Campus Scientifique, B.P. 239, 54506 Vandoeuvre-lès-Nancy Cedex, France, August 14, 2003. URL `http://perso.ens-lyon.fr/jean-michel.muller/TMDworstcases.pdf`. {**28**}

[LM08] Lawrence M. Leemis and Jacquelyn T. McQueston. Univariate distribution relationships. *The American Statistician*, 62(1):45–53, February 2008. CODEN ASTAAJ. ISSN 0003-1305 (print), 1537-2731 (electronic). URL `http://www.ingentaconnect.com/content/asa/tas/2008/00000062/00000001/art00008`. DOI `10.1198/000313008X270448`. {**196**}

[Loi10] Florian Loitsch. Printing floating-point numbers quickly and accurately with integers. *ACM SIGPLAN Notices*, 45(6):233–243, June 2010. CODEN SINODQ. ISSN 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI `10.1145/1809028.1806623`. {**895**}

[Loz03] Daniel W. Lozier. NIST Digital Library of Mathematical Functions. *Annals of Mathematics and Artificial Intelligence*, 38(1–3):105–119, May 2003. CODEN AMAIEC. ISSN 1012-2443 (print), 1573-7470 (electronic). URL `http://math.nist.gov/acmd/Staff/DLozier/publications/Linz01.ps`. DOI `10.1023/A:1022915830921`. {**826**}

[LS02] Pierre L'Ecuyer and Richard Simard. TestU01: A software library in ANSI C for empirical testing of random number generators: Software user's guide. Web report, Départment d'Informatique et de Recherche Opérationelle, Université de Montréal, Montréal, Québec, Canada, 2002. URL `http://www.iro.umontreal.ca/~simardr/TestU01.zip`. {**200**}

[LS07] Pierre L'Ecuyer and Richard Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4):22:1–22:40, August 2007. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/1268776.1268777`. {**170, 178, 200, 214**}

[LS14] Pierre L'Ecuyer and Richard Simard. On the lattice structure of a special class of multiple recursive random number generators. *INFORMS Journal on Computing*, 26(3):449–460, 2014. ISSN 1091-9856 (print), 1526-5528 (electronic). DOI `10.1287/ijoc.2013.0576`. Analysis and exposure of serious lattice structure in earlier work on fast multiple recursive generators [DX03, Den05, DLS09, DSL12a, DSL12b]. {**176**}

[LSZ06] Vincent Lefèvre, Damien Stehlé, and Paul Zimmermann. Worst cases for the exponential function in the IEEE 754r `decimal64` format. Technical report, LORIA/INRIA Lorraine, Villers-lès-Nancy Cedex, France, September 2006. 14 pp. URL `http://www.loria.fr/~zimmerma/papers/decimalexp-lncs-final.pdf`. {**28**}

[Luk69a] Yudell L. Luke. *The Special Functions and Their Approximations*, volume 53-I of *Mathematics in Science and Engineering*. Academic Press, New York, NY, USA, 1969. ISBN 0-12-459901-X; 978-0-12-459901-7. xx + 349 pp. LCCN QA351 .L94 1969. URL `http://www.sciencedirect.com/science/book/9780124599017`. DOI `10.1016/S0076-5392(08)62628-4`. {**521, 693, 827**}

[Luk69b] Yudell L. Luke. *The Special Functions and Their Approximations*, volume 53-II of *Mathematics in Science and Engineering*. Academic Press, New York, NY, USA, 1969. ISBN 0-12-459902-8; 978-0-12-459902-4. xx + 485 pp. LCCN QA351 .L797. URL `http://www.sciencedirect.com/science/bookseries/00765392/53/part/P2`. DOI `10.1016/S0076-5392(09)60064-3`. {**521, 827**}

[Luk77] Yudell L. Luke. *Algorithms for the Computation of Mathematical Functions*. Academic Press, New York, NY, USA, 1977. ISBN 0-12-459940-0; 978-0-12-459940-6. xiii + 284 pp. LCCN QA351 .L7961. {**693, 827**}

[LW92] Lisa Lorentzen and Haakon Waadeland. *Continued Fractions with Applications*, volume 3 of *Studies in Computational Mathematics*. North-Holland, Amsterdam, The Netherlands, 1992. ISBN 0-444-89265-6; 978-0-444-89265-2. xvi + 606 pp. LCCN QA295 .L64 1992. {**19**}

[LW08] Lisa Lorentzen and Haakon Waadeland. *Continued Fractions: Convergence Theory*, volume 1 of *Atlantis Studies in Mathematics for Engineering and Science*. Atlantis Press, Amsterdam, The Netherlands, second edition, 2008. ISBN 90-78677-07-4; 978-90-78677-07-9. ISSN 1875-7642. xii + 308 pp. LCCN QA295 .L64 2008. {**19**}

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1997. ISBN 0-201-63452-X; 978-0-201-63452-5. xvi + 475 pp. LCCN QA76.73.J38L56 1997. URL `http://www.aw.com/cp/javaseries.html`. {**viii, 80, 979**}

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999. ISBN 0-201-43294-3; 978-0-201-43294-7. xv + 473 pp. LCCN QA76.73.J38L56 1999. {**viii, 80, 979**}

[LYBB13] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Addison-Wesley, Addison-Wesley, Java SE 7 edition, 2013. ISBN 0-13-326049-6, 0-13-326044-5; 978-0-13-326049-6, 978-0-13-326044-1. xvii + 587 (est.) pp. LCCN QA76.73.J38 L56 1999. URL `http://proquest.tech.safaribooksonline.de/9780133260496`. {**viii, 979**}

[LYBB14] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification: Java SE 8 edition*. Addison-Wesley, Addison-Wesley, 2014. ISBN 0-13-390590-X (paperback), 0-13-392274-X (e-book); 978-0-13-390590-8 (paperback), 978-0-13-392274-5 (e-book). xvi + 584 pp. LCCN QA76.73.J38 L56 2014. {**viii, 979**}

[Mac65] M. Donald MacLaren. Algorithm 272: Procedure for the normal distribution functions [S15]. *Communications of the Association for Computing Machinery*, 8(12):789–790, December 1965. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/365691.365957`. See remarks [HJ67b, Mac68]. {**618, 1015, 1023**}

[Mac68] M. Donald MacLaren. Remark on Algorithm 272: Procedure for the normal distribution functions. *Communications of the Association for Computing Machinery*, 11(7):498, July 1968. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/363397.363553`. See [Mac65]. {**618, 1023**}

[Mac92] N. M. Maclaren. A limit on the usable length of a pseudorandom sequence. *Journal of Statistical Computation and Simulation*, 42(1–2): 47–54, 1992. CODEN JSCSAJ. ISSN 0094-9655 (print), 1026-7778 (electronic), 1563-5163. URL `http://www.tandfonline.com/doi/abs/10.1080/00949659208811409`. DOI `10.1080/00949659208811409`. {**180**}

[Mac98] I. G. Macdonald. *Symmetric Functions and Orthogonal Polynomials*, volume 12 of *University lecture series*. American Mathematical Society, Providence, RI, USA, 1998. ISBN 0-8218-0770-6 (softcover); 978-0-8218-0770-5 (softcover). ISSN 1047-3998. xv + 53 pp. LCCN QA212 .M33 1998. {**59**}

[Mal72] M. A. Malcolm. Algorithms to reveal properties of floating-point arithmetic. *Communications of the Association for Computing Machinery*, 15(11):949–951, November 1972. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/355606.361870`. See also [GM74]. {**103, 1012**}

[Mao91] Eli Maor. *To Infinity and Beyond: A Cultural History of the Infinite*. Princeton paperbacks. Princeton University Press, Princeton, NJ, USA, 1991. ISBN 0-691-02511-8 (paperback); 978-0-691-02511-7 (paperback). xvi + 284 pp. LCCN QA9 .M316 1991. {**59**}

[Mao94] Eli Maor. *e: The Story of a Number*. Princeton University Press, Princeton, NJ, USA, 1994. ISBN 0-691-03390-0; 978-0-691-03390-7. xiv + 223 pp. LCCN QA247.5.M33 1994. URL `http://www-gap.dcs.st-and.ac.uk/~history/HistTopics/e.html`. The number $e \approx 2.718$ is the base of the natural logarithms, and is one of the *big five* mathematical constants: e, i, π, γ, and ϕ. {**59, 269**}

[Mao07] Eli Maor. *The Pythagorean Theorem: a 4,000-year History*. Princeton University Press, Princeton, NJ, USA, 2007. ISBN 0-691-12526-0; 978-0-691-12526-8. xvi + 259 pp. LCCN QA460.P8 M36 2007. {**59**}

[Mar68] George Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences of the United States of America*, 61(1):25–28, September 15, 1968. CODEN PNASA6. ISSN 0027-8424 (print), 1091-6490 (electronic). URL `http://www.pnas.org/content/61/1/25`. This important, and frequently cited, paper was the first to point out the serious problem of correlations in random numbers produced by *all* congruential generators. {**170**}

[Mar00] Peter Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard professional books. Prentice-Hall, Upper Saddle River, NJ, USA, 2000. ISBN 0-13-018348-2; 978-0-13-018348-4. xix + 298 pp. LCCN QA76.9.A73 M365 2000. URL `http://www.markstein.org/`. {**86, 87, 101, 241, 242, 410, 412, 824, 827, 940, 953**}

[Mar03a] George Marsaglia. Random number generators. *Journal of Modern Applied Statistical Methods*, 2(1):2–13, May 2003. ISSN 1538-9472. URL http://stat.fsu.edu/pub/diehard/; http://tbf.coe.wayne.edu/jmasm/; http://www.csis.hku.hk/~diehard/. DOI 10.22237/jmasm/1051747320. {**207**}

[Mar03b] George Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003. CODEN JSSOBK. ISSN 1548-7660. URL http://www.jstatsoft.org/v08/i14; http://www.jstatsoft.org/v08/i14/xorshift.pdf. DOI 10.18637/jss.v008.i14. See [Bre04] for corrections and the equivalence of xorshift generators and the well-understood linear feedback shift register generators. See also [SMDS11, SM12, SLF14] for the failure of Marsaglia's xorwow() generator from this paper. See [PL05, Vig16] for detailed analysis. {**176, 1001, 1028**}

[Mat68a] David W. Matula. The base conversion theorem. *Proceedings of the American Mathematical Society*, 19(3):716–723, June 1968. CODEN PAMYAR. ISSN 0002-9939 (print), 1088-6826 (electronic). URL http://www.ams.org/journals/proc/1968-019-03/S0002-9939-1968-0234908-9/S0002-9939-1968-0234908-9.pdf. DOI 10.1090/S0002-9939-1968-0234908-9. {**840, 851**}

[Mat68b] David W. Matula. In-and-out conversions. *Communications of the Association for Computing Machinery*, 11(1):47–50, January 1968. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI 10.1145/362851.362887. {**840, 851**}

[MBdD^{+}10] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston Inc., Cambridge, MA, USA, 2010. ISBN 0-8176-4704-X; 978-0-8176-4704-9. xxiii + 572 pp. LCCN QA76.9.C62 H36 2010. DOI 10.1007/978-0-8176-4704-9. {**42, 104, 407, 881, 978**}

[McC81] Peter McCullagh. A rapidly convergent series for computing $\psi(z)$ and its derivatives. *Mathematics of Computation*, 36(153):247–248, January 1981. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL http://www.jstor.org/stable/2007741. DOI 10.2307/2007741. The psi function, $\psi(z)$, is the logarithmic derivative of the gamma function, $\Gamma(z)$. The higher derivatives are called *polygamma* functions. {**543**}

[McL85] A. Ian McLeod. Statistical algorithms: Remark AS R58: a remark on Algorithm AS 183. an efficient and portable pseudo-random number generator. *Applied Statistics*, 34(2):198–200, 1985. CODEN APSTAG. ISSN 0035-9254 (print), 1467-9876 (electronic). URL http://lib.stat.cmu.edu/apstat/183. DOI 10.2307/2347378. See [WH82, Zei86]. {**1037**}

[McL91] John McLeish. *The Story of Numbers: How Mathematics has Shaped Civilization*. Fawcett Columbine, New York, NY, USA, 1991. ISBN 0-449-90938-7; 978-0-449-90938-6. 266 + 8 pp. LCCN QA21.M38 1991. {**59**}

[MF53] Philip McCord Morse and Herman Feshbach. *Methods of Theoretical Physics*. International series in pure and applied physics. McGraw-Hill, New York, NY, USA, 1953. ISBN 0-07-043316-X (vol. 1), 0-07-043317-8 (vol. 2); 978-0-07-043316-8 (vol. 1), 978-0-07-043317-5 (vol. 2). xxii + 997 (vol. 1), xviii + 1978 (vol. 2) pp. LCCN QC20 .M6 1999. {**693**}

[MH72] Tohru Morita and Tsuyoshi Horiguchi. Convergence of the arithmetic-geometric mean procedure for the complex variables and the calculation of the complete elliptic integrals with complex modulus. *Numerische Mathematik*, 20(5):425–430, October 1972. CODEN NUMMA7. ISSN 0029-599X (print), 0945-3245 (electronic). URL http://www.springerlink.com/openurl.asp?genre=article&issn=0029-599X&volume=20&issue=5&spage=425. DOI 10.1007/BF01402565. {**632**}

[MH80] Jerry B. Marion and Mark A. Heald. *Classical Electromagnetic Radiation*. Academic Press, New York, NY, USA, second edition, 1980. ISBN 0-12-472257-1; 978-0-12-472257-6. xvii + 488 pp. LCCN QC661 .M38 1980. {**693**}

[MH03] J. C. Mason and D. C. Handscomb. *Chebyshev Polynomials*. Chapman and Hall/CRC, Boca Raton, FL, USA, 2003. ISBN 0-8493-0355-9; 978-0-8493-0355-5. xiii + 341 pp. LCCN QA404.5 .M37 2003. {**58**}

[MH08] A. M. Mathai and H. J. Haubold. *Special Functions for Applied Scientists*. Springer Science + Business Media, New York, NY, USA, 2008. ISBN 0-387-75893-3; 978-0-387-75893-0. xxv + 464 pp. LCCN QA351.M37; QA351.M37 2008. DOI 10.1007/978-0-387-75894-7. {**827**}

[MH16] Jamshaid Sarwar Malik and Ahmed Hemani. Gaussian random number generation: a survey on hardware architectures. *ACM Computing Surveys*, 49(3):53:1–53:37, November 2016. CODEN CMSVAN. ISSN 0360-0300 (print), 1557-7341 (electronic). DOI 10.1145/2980052. {**196**}

[Mis10] Thomas J. Misa. An interview with Edsger W. Dijkstra. *Communications of the Association for Computing Machinery*, 53(8):41–47, August 2010. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI 10.1145/1787234.1787249. {**962**}

[ML15] George Michelogiannakis and Xiaoye S. Li. Extending summation precision for network reduction operations. *International Journal of Parallel Programming*, 43(6):1218–1243, December 2015. CODEN IJPPE5. ISSN 0885-7458 (print), 1573-7640 (electronic). URL http://link.springer.com/article/10.1007/s10766-014-0326-5. DOI 10.1007/s10766-014-0326-5. {**385**}

[MM65] M. Donald MacLaren and George Marsaglia. Uniform random number generators. *Journal of the Association for Computing Machinery*, 12(1):83–89, January 1965. CODEN JACOAH. ISSN 0004-5411 (print), 1557-735X (electronic). DOI 10.1145/321250.321257. {**179**}

[MM83] Cleve B. Moler and Donald Morrison. Replacing square roots by Pythagorean sums. *IBM Journal of Research and Development*, 27 (6):577–581, November 1983. CODEN IBMJAE. ISSN 0018-8646 (print), 2151-8556 (electronic). URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5390405. DOI 10.1147/rd.276.0577. See [Dub83] and generalization [Jam89]. {**227, 1008, 1017**}

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998. CODEN ATMCEZ. ISSN 1049-3301 (print), 1558-1195 (electronic). DOI 10.1145/272991.272995. {**177**}

[MNZ90] George Marsaglia, B. Narasimhan, and Arif Zaman. A random number generator for PC's. *Computer Physics Communications*, 60(3): 345–349, October 1990. CODEN CPHCBZ. ISSN 0010-4655 (print), 1879-2944 (electronic). URL http://www.sciencedirect.com/science/article/pii/001046559090033W. DOI 10.1016/0010-4655(90)90033-W. {**177**}

[Møl65a] Ole Møller. Note on quasi double-precision. *Nordisk Tidskrift for Informationsbehandling*, 5(4):251–255, December 1965. CODEN BITTEL, NBITAB. ISSN 0006-3835 (print), 1572-9125 (electronic). URL `http://www.springerlink.com/openurl.asp?genre=article&issn=0006-3835&volume=5&issue=4&spage=251`. DOI `10.1007/BF01937505`. See [Møl65b]. {**1025**}

[Møl65b] Ole Møller. Quasi double-precision in floating point addition. *Nordisk Tidskrift for Informationsbehandling*, 5(1):37–50, March 1965. CODEN BITTEL, NBITAB. ISSN 0006-3835 (print), 1572-9125 (electronic). URL `http://www.springerlink.com/openurl.asp?genre=article&issn=0006-3835&volume=5&issue=1&spage=37`. DOI `10.1007/BF01975722`. See also [Møl65a]. {**353, 1025**}

[Mos89] Stephen L. B. Moshier. *Methods and Programs for Mathematical Functions*. Ellis Horwood, New York, NY, USA, 1989. ISBN 0-7458-0289-3; 978-0-7458-0289-3. vii + 415 pp. LCCN QA331 .M84 1989. URL `http://www.netlib.org/cephes`. {**133, 270, 521, 556, 558, 567, 583, 593, 600, 617, 644, 657, 693, 708, 823**}

[MP98] Charlene Morrow and Teri Perl, editors. *Notable Women in Mathematics: A Biographical Dictionary*. Greenwood Press, Westport, CT, USA, 1998. ISBN 0-313-29131-4; 978-0-313-29131-9. xv + 302 pp. LCCN QA28 .N68 1998. {**59**}

[MPFR04] *MPFR: The Multiple Precision Floating-Point Reliable Library: Edition 2.1.0: November 2004*, 2004. ii + 35 pp. URL `http://www.mpfr.org/mpfr-current/mpfr.pdf`. {**401, 407, 825**}

[MR90] Michael Metcalf and John Ker Reid. *Fortran 90 Explained*. Oxford science publications. Oxford University Press, Oxford, UK, 1990. ISBN 0-19-853772-7 (paperback); 978-0-19-853772-4 (paperback). xiv + 294 pp. LCCN QA76.73.F28 M48 1990. {**106**}

[MR04] James S. Miller and Susann Ragsdale. *The Common Language Infrastructure Annotated Standard*. Addison-Wesley, Reading, MA, USA, 2004. ISBN 0-321-15493-2; 978-0-321-15493-4. xxxii + 891 pp. LCCN QA76.7 .M52 2003. {**917**}

[MRR91] Michael Müller, Christine Rüb, and Wolfgang Rülling. Exact accumulation of floating-point numbers. In Kornerup and Matula [KM91], pages 64–69. ISBN 0-8186-9151-4 (case), 0-8186-6151-8 (microfiche), 0-7803-0187-0 (library binding); 978-0-8186-9151-5 (case), 978-0-8186-6151-8 (microfiche), 978-0-7803-0187-0 (library binding). LCCN QA76.9.C62 S95 1991. DOI `10.1109/ARITH.1991.145535`. IEEE catalog number 91CH3015-5. {**385**}

[MRR96] Michael Müller, Christine Rüb, and Wolfgang Rülling. A circuit for exact summation of floating-point numbers. *Information Processing Letters*, 57(3):159–163, February 12, 1996. CODEN IFPLAT. ISSN 0020-0190 (print), 1872-6119 (electronic). URL `http://www.sciencedirect.com/science/article/pii/0020019095002057`. DOI `10.1016/0020-0190(95)00205-7`. {**385**}

[MS00a] Michael Mascagni and Ashok Srinivasan. Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software*, 26(3):436–461, September 2000. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/358407.358427`. See correction [MS00b]. {**158, 1025**}

[MS00b] Michael Mascagni and Ashok Srinivasan. Corrigendum: Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software*, 26(4):618–619, December 2000. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/365723.365738`. See [MS00a]. {**1025**}

[MS05] Paolo Montuschi and Eric M. Schwarz, editors. *Proceedings of the 17th IEEE Symposium on Computer Arithmetic, ARITH-17 2005, June 27–29, 2005, Cape Cod, Massachusetts, USA*. IEEE Computer Society Press, Silver Spring, MD, USA, 2005. ISBN 0-7695-2366-8; 978-0-7695-2366-8. LCCN QA76.9.C62 .S95 2005. {**1000, 1021, 1034**}

[MSH^{+}16] Paolo Montuschi, Michael Schulte, Javier Hormigo, Stuart Oberman, and Nathalie Revol, editors. *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH 2016), Santa Clara, California, USA, 10–13 July 2016*. IEEE Computer Society Press, Silver Spring, MD, USA, 2016. ISBN 1-5090-1615-5; 978-1-5090-1615-0. ISSN 1063-6889. LCCN QA76.9.C62 S95 2016. URL `http://ieeexplore.ieee.org/servlet/opac?punumber=7562813`. {**104, 1021**}

[MT02] George Marsaglia and Wai Wan Tsang. Some difficult-to-pass tests of randomness. *Journal of Statistical Software*, 7(3):1–8, 2002. CODEN JSSOBK. ISSN 1548-7660. URL `http://www.jstatsoft.org/v07/i03`; `http://www.jstatsoft.org/v07/i03/tuftests.c`; `http://www.jstatsoft.org/v07/i03/tuftests.pdf`; `http://www.jstatsoft.org/v07/i03/updates`. DOI `10.18637/jss.v007.i03`. {**200**}

[MU49] Nicholas Metropolis and Stanisław Ulam. The Monte Carlo method. *Journal of the American Statistical Association*, 44(247):335–341, September 1949. CODEN JSTNAL. ISSN 0162-1459 (print), 1537-274X (electronic). URL `http://www.jstor.org/stable/2280232`. DOI `10.2307/2280232`. This may be the earliest published article on the Monte Carlo method after the algorithm was declassified following World War II. {**203**}

[Mul97] Jean-Michel Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser, Cambridge, MA, USA; Berlin, Germany; Basel, Switzerland, 1997. ISBN 0-8176-3990-X; 978-0-8176-3990-7. xv + 204 pp. LCCN QA331.M866 1997. URL `http://perso.ens-lyon.fr/jean-michel.muller/`; `http://www.birkhauser.com/cgi-win/ISBN/0-8176-3990-X`. {**251, 827**}

[Mul05] Jean-Michel Muller. On the definition of `ulp(x)`. Rapport de recherche LIP RR2005-09, INRIA RR-5504, Laboratoire de l'Informatique du Parallélisme, Lyon, France, February 2005. 19 pp. URL `ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-5504.pdf`. The function `ulp(x)` returns the *unit in the last place* of `x`. {**82**}

[Mul06] Jean-Michel Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser, Cambridge, MA, USA; Berlin, Germany; Basel, Switzerland, second edition, 2006. ISBN 0-8176-4372-9; 978-0-8176-4372-0. xxii + 266 pp. LCCN QA331 .M866 2006. URL `http://perso.ens-lyon.fr/jean-michel.muller/SecondEdition.html`; `http://www.springer.com/sgw/cda/frontpage/0,,4-40109-22-72377986-0,00.html`. {**55, 827**}

[Mul15] Jean-Michel Muller. On the error of computing $ab + cd$ using Cornea, Harrison and Tang's method. *ACM Transactions on Mathematical Software*, 41(2):7:1–7:8, January 2015. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/2629615`. {**463**}

[Mul16] Jean-Michel Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser Boston Inc., Cambridge, MA, USA, third edition, 2016. ISBN 1-4899-7981-6 (print), 1-4899-7983-2 (e-book); 978-1-4899-7981-0 (print), 978-1-4899-7983-4 (e-book). xxv + 283 pp. LCCN QA331 .M866 2016. DOI `10.1007/978-1-4899-7983-4`. {**827**}

[MvA06] Francisco Marcellán and Walter van Assche, editors. *Orthogonal Polynomials and Special Functions: Computation and Applications*, volume 1883 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2006. ISBN 3-540-31062-2; 978-3-540-31062-4. ISSN 0075-8434 (print), 1617-9692 (electronic). xiv + 418 pp. LCCN QA3 .L28 no. 1883; QA404.5 .O735 2006. DOI `10.1007/b128597`. {**59, 827**}

[MWKA07] Makoto Matsumoto, Isaku Wada, Ai Kuramoto, and Hyo Ashihara. Common defects in initialization of pseudorandom number generators. *ACM Transactions on Modeling and Computer Simulation*, 17(4):15:1–15:20, September 2007. CODEN ATMCEZ. ISSN 1049-3301 (print), 1558-1195 (electronic). DOI `10.1145/1276927.1276928`. {**176**}

[MXJ04] D. N. Prabhakar Murthy, Min Xie, and Renyan Jiang. *Weibull Models*. Wiley series in probability and statistics. Wiley, New York, NY, USA, 2004. ISBN 0-471-36092-9 (cloth); 978-0-471-36092-6 (cloth). xvi + 383 pp. LCCN QA273.6 .M87 2004. {**196**}

[MZ91] George Marsaglia and Arif Zaman. A new class of random number generators. *Annals of Applied Probability*, 1(3):462–480, August 1991. ISSN 1050-5164. URL `http://projecteuclid.org/euclid.aoap/1177005878`. DOI `10.1214/aoap/1177005878`. See remarks in [EH95, TLC93] about the extremely bad lattice structure in high dimensions of the generators proposed in this paper. {**177, 1035**}

[MZ93] George Marsaglia and Arif Zaman. Monkey tests for random number generators. *Computers and Mathematics with Applications*, 26(9): 1–10, November 1993. CODEN CMAPDK. ISSN 0898-1221 (print), 1873-7668 (electronic). DOI `10.1016/0898-1221(93)90001-C`. See also [PW95]. {**200, 1028**}

[MZM94] George Marsaglia, Arif Zaman, and John C. W. Marsaglia. Rapid evaluation of the inverse of the normal distribution function. *Statistics & Probability Letters*, 19(4):259–266, March 15, 1994. CODEN SPLTDC. ISSN 0167-7152 (print), 1879-2103 (electronic). DOI `10.1016/0167-7152(94)90174-0`. {**618**}

[Nah06] Paul J. Nahin. *Dr. Euler's Fabulous Formula: Cures Many Mathematical Ills*. Princeton University Press, Princeton, NJ, USA, 2006. ISBN 0-691-11822-1 (hardcover); 978-0-691-11822-2 (hardcover). xx + 380 pp. LCCN QA255 .N339 2006. The Euler formula, $e^{i\pi} + 1 = 0$, relates five important mathematical constants, and the digits of the binary number system. {**14, 59, 591, 623**}

[Nea73] Henry R. Neave. Miscellanea: On using the Box–Muller transformation with multiplicative congruential pseudo-random number generators. *Applied Statistics*, 22(1):92–97, 1973. CODEN APSTAG. ISSN 0035-9254 (print), 1467-9876 (electronic). URL `http://www.jstor.org/stable/2346308`. DOI `10.2307/2346308`. {**193**}

[Nea15] Radford M. Neal. Fast exact summation using small and large superaccumulators. Report, Department of Statistical Sciences and Department of Computer Science, University of Toronto, Toronto, ON, Canada, 2015. 22 pp. URL `http://www.cs.toronto.edu/~radford/ftp/xsum.pdf`. {**385**}

[Neh07] Markus Neher. Complex standard functions and their implementation in the CoStLy library. *ACM Transactions on Mathematical Software*, 33(1):2:1–2:27, March 2007. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/1206040.1206042`. {**476**}

[Nev44] Eric Harold Neville. *Jacobian Elliptic Functions*. Clarendon Press, Oxford, UK, 1944. xiii + 2 + 331 + 1 pp. LCCN QA343 .N5. {**678**}

[Nev51] Eric Harold Neville. *Jacobian Elliptic Functions*. Clarendon Press, Oxford, UK, second edition, 1951. xiv + 345 pp. LCCN QA343 .N5 1951. {**678**}

[New05] M. E. J. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary physics*, 46(5):323–351, September 2005. CODEN CTPHAF. ISSN 0010-7514 (print), 1366-5812 (electronic). DOI `10.1080/00107510500052444`. {**196**}

[Ng92] K. C. Ng. Argument reduction for huge arguments: Good to the last bit. *SunPro*, July 13, 1992. URL `http://www.validlab.com/arg.pdf`. {**250**}

[Nie78] Harald Niederreiter. Quasi-Monte Carlo methods and pseudo-random numbers. *Bulletin of the American Mathematical Society*, 84(6): 957–1041, November 1978. CODEN BAMOAD. ISSN 0002-9904 (print), 1936-881X (electronic). URL `http://www.ams.org/bull/1978-84-06/S0002-9904-1978-14532-7/S0002-9904-1978-14532-7.pdf`. DOI `10.1090/S0002-9904-1978-14532-7`. {**203**}

[Nie92] Harald Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*, volume 63. SIAM (Society for Industrial and Applied Mathematics), Philadelphia, PA, USA, 1992. ISBN 0-89871-295-5; 978-0-89871-295-7. vi + 241 pp. LCCN QA298 .N54 1992. {**203**}

[Nie03] Yves Nievergelt. Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit. *ACM Transactions on Mathematical Software*, 29(1):27–48, March 2003. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/641876.641878`. {**87**}

[NIS15] NIST. SHA-3 standard: Permutation-based hash and extendable-output functions. FIPS PUB 202, National Institute for Standards and Technology, Gaithersburg, MD, USA, 2015. viii + 29 pp. DOI `10.6028/NIST.FIPS.202`. {**178**}

[NW97] Roger M. Needham and David J. Wheeler. TEA extensions. Report, Cambridge University, Cambridge, UK, October 1997. URL `http://www.movable-type.co.uk/scripts/xtea.pdf`. See also original TEA [WN95] and extension XXTEA [WN98]. {**178, 1037**}

[OE74] R. E. Odeh and J. O. Evans. Statistical algorithms: Algorithm AS 70: The percentage points of the normal distribution. *Applied Statistics*, 23(1):96–97, March 1974. CODEN APSTAG. ISSN 0035-9254 (print), 1467-9876 (electronic). URL `http://lib.stat.cmu.edu/apstat/70`. DOI `10.2307/2347061`. {**618**}

[OLBC10] Frank W. J. Olver, Daniel W. Lozier, Ronald F. Boisvert, and Charles W. Clark, editors. *NIST Handbook of Mathematical Functions*. Cambridge University Press, Cambridge, UK, 2010. ISBN 0-521-19225-0; 978-0-521-19225-5. xv + 951 pp. LCCN QA331 .N57 2010. URL `http://dlmf.nist.gov/`; `http://www.cambridge.org/catalogue/catalogue.asp?isbn=9780521192255`. {**6, 58, 269, 301, 303, 341, 498, 521, 560, 562, 574, 589, 593, 600, 619, 624, 627, 632, 653, 657, 661, 666, 673, 675, 678, 682, 689, 693, 826**}

[Olv74] Frank W. J. Olver. *Asymptotics and Special Functions*. Academic Press, New York, NY, USA, 1974. ISBN 0-12-525850-X; 978-0-12-525850-0. xvi + 572 pp. LCCN QA351 .O481 1974. {**19, 521, 693, 827**}

[Omo94] Amos R. Omondi. *Computer Arithmetic Systems: Algorithms, Architecture, and Implementation*. Prentice-Hall, Upper Saddle River, NJ, USA, 1994. ISBN 0-13-334301-4; 978-0-13-334301-4. xvi + 520 pp. LCCN QA76.9.C62 O46 1994. {**407, 881, 978**}

[OMS09] Keith B. Oldham, Jan Myland, and Jerome Spanier, editors. *An Atlas of Functions*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., second edition, 2009. ISBN 0-387-48807-3 (softcover), 0-387-48806-5 (hardcover); 978-0-387-48807-3 (softcover), 978-0-387-48806-6 (hardcover). xi + 748 pp. LCCN QA331 .S685 2009. DOI `10.1007/978-0-387-48807-3`. {**1032**}

[OOO16] Katsuhisa Ozaki, Takeshi Ogita, and Shin'ichi Oishi. Error-free transformation of matrix multiplication with a posteriori validation. *Numerical Linear Algebra with Applications*, 23(5):931–946, October 2016. CODEN NLAAEM. ISSN 1070-5325 (print), 1099-1506 (electronic). DOI `10.1002/nla.2061`. {**385**}

[ORO05] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6): 1955–1988, November 2005. CODEN SJOCE3. ISSN 1064-8275 (print), 1095-7197 (electronic). URL `http://epubs.siam.org/sam-bin/dbq/article/60181`. DOI `10.1137/030601818`. {**385**}

[O'S07] Donal O'Shea. *The Poincaré Conjecture: in Search of the Shape of the Universe*. Walker and Company, New York, NY, USA, 2007. ISBN 0-8027-1532-X; 978-0-8027-1532-6. ix + 293 pp. LCCN QA612 .O83 2007. {**521, 579**}

[Ove01] Michael L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic, Including One Theorem, One Rule of Thumb, and One Hundred and One Exercises*. SIAM (Society for Industrial and Applied Mathematics), Philadelphia, PA, USA, 2001. ISBN 0-89871-482-6; 978-0-89871-482-1. xiv + 104 pp. LCCN QA76.9.M35 O94 2001. URL `http://www.cs.nyu.edu/cs/faculty/overton/book/; http://www.siam.org/catalog/mcc07/ot76.htm`. {**67, 103**}

[Pag77] E. Page. Miscellanea: Approximations to the cumulative normal function and its inverse for use on a pocket calculator. *Applied Statistics*, 26(1):75–76, 1977. CODEN APSTAG. ISSN 0035-9254 (print), 1467-9876 (electronic). URL `http://www.jstor.org/stable/2346872`. DOI `10.2307/2346872`. {**618**}

[Pai19] Eleanor Pairman. *Tables of the Digamma and Trigamma Functions*, volume I of *Tracts for computers*. Cambridge University Press, Cambridge, UK, 1919. 9 + 11 pp. LCCN QA47 .T7 no.1. {**537**}

[Par69] Ronald G. Parson. Certification of Algorithm 147 [S14]: PSIF. *Communications of the Association for Computing Machinery*, 12(12):691–692, December 1969. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/363626.363651`. See [Ami62, Tha63]. {**996, 1035**}

[Par00] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, Oxford, UK, 2000. ISBN 0-19-512583-5; 978-0-19-512583-2. xx + 490 pp. LCCN QA76.9.C62P37 1999. {**68, 881, 978**}

[PAS82] *Specification for Computer Programming Language Pascal, ISO 7185-1982*. International Organization for Standardization, Geneva, Switzerland, 1982. URL `http://www.iso.ch/cate/d13802.html`. {**70**}

[PAS90] *Extended Pascal ISO 10206:1990*. International Organization for Standardization, Geneva, Switzerland, 1990. xii + 218 pp. URL `http://www.iso.ch/cate/d18237.html`. Available in PostScript for personal use only at http://pascal.miningco.com/msub1.htm. {**vii, 989**}

[Pat88] S. J. Patterson. *An Introduction to the Theory of the Riemann Zeta-function*, volume 14 of *Cambridge studies in advanced mathematics*. Cambridge University Press, Cambridge, UK, 1988. ISBN 0-521-33535-3; 978-0-521-33535-5. xiii + 156 pp. LCCN QA246 .P28 1988. {**579**}

[Pea00] Karl Pearson. On a criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen in random sampling. *Philosophical Magazine*, 50(302):157–175, July/December 1900. CODEN PHMAA4. ISSN 0031-8086. URL `http://www.tandfonline.com/doi/pdf/10.1080/14786440009463897`. DOI `10.1080/14786440009463897`. {**197**}

[PGPB90] E. S. Pearson, William Sealy Gosset, R. L. Plackett, and George A. Barnard, editors. *Student: a Statistical Biography of William Sealy Gosset*. Clarendon Press, Oxford, UK, 1990. ISBN 0-19-852227-4; 978-0-19-852227-0. viii + 142 pp. LCCN QA276.157.G67 P43 1990. Gosset is the originator of the widely used *Student t-test* in the statistics of small sample sizes. {**196**}

[PH83a] M. H. Payne and R. N. Hanek. Degree reduction for trigonometric functions. *ACM SIGNUM Newsletter*, 18(2):18–19, April 1983. CODEN SNEWD6. ISSN 0163-5778 (print), 1558-0237 (electronic). DOI `10.1145/1057605.1057606`. {**250, 253**}

[PH83b] M. H. Payne and R. N. Hanek. Radian reduction for trigonometric functions. *ACM SIGNUM Newsletter*, 18(1):19–24, January 1983. CODEN SNEWD6. ISSN 0163-5778 (print), 1558-0237 (electronic). DOI `10.1145/1057600.1057602`. {**250, 253**}

[PH02] David A. Patterson and John L. Hennessy. *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, third edition, 2002. ISBN 1-55860-596-7; 978-1-55860-596-1. xxi + 883 + A-87 + B-42 + C-1 + D-1 + E-1 + F-1 + G-1 + H-1 + I-1 + R-22 + I-44 pp. LCCN QA76.9.A73 P377 2003. URL `http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-596-7; http://www.mkp.com/CA3`. {**103, 1013**}

[PH08] David A. Patterson and John L. Hennessy. *Computer Organization and Design: the Hardware/Software Interface*. Elsevier/Morgan Kaufmann, San Francisco, CA, USA, fourth edition, 2008. ISBN 0-12-374493-8; 978-0-12-374493-7. xxv + 703 + A-77 + B-83 + I-26 pp. LCCN QA76.9.C643. {**103**}

[PH12] David A. Patterson and John L. Hennessy. *Computer Organization and Design: the Hardware/Software Interface*. The Morgan Kaufmann series in computer architecture and design. Morgan Kaufmann/Elsevier, Waltham, MA, USA, fourth edition, 2012. ISBN 0-12-374750-3 (paperback); 978-0-12-374750-1 (paperback). xxv + 703 pp. LCCN QA76.9.C643 H46 2012. URL `http://store.elsevier.com/product.jsp?isbn=9780080922812`. With contributions by Perry Alexander, Peter J. Ashenden, Javier Bruguera, Jichuan Chang, Matthew Farrens, David Kaeli, Nicole Kaiyan, David Kirk, James R. Larus, Jacob Leverich, Kevin Lim, John Nickolls, John Oliver, Milos Prvulovic, and Parta Ranganthan. {**103**}

[Phi60] J. R. Philip. The function inverfc θ. *Australian Journal of Physics*, 13(1):13–20, March 1960. CODEN AUJPAS. ISSN 0004-9506 (print), 1446-5582 (electronic). DOI 10.1071/PH600013. {**600**}

[Phi86] Jen Phillips. *The NAG Library*. Clarendon Press, Oxford, UK, 1986. ISBN 0-19-853263-6; 978-0-19-853263-7. viii + 245 pp. LCCN QA297.P53 1986. {**826**}

[PK91] Vern Paxson and William M. Kahan. A program for testing IEEE binary–decimal conversion. World-Wide Web document, May 1991. URL ftp://ftp.ee.lbl.gov/testbase-report.ps.Z; ftp://ftp.ee.lbl.gov/testbase.tar.Z. {**1015**}

[PL05] François Panneton and Pierre L'Ecuyer. On the xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation*, 15(4):346–361, October 2005. CODEN ATMCEZ. ISSN 1049-3301 (print), 1558-1195 (electronic). DOI 10.1145/1113316.1113319. See [Mar03b, Bre04, Vig16]. {**1001, 1024**}

[PL07] Alfred S. Posamentier and Ingmar Lehmann. *The Fabulous Fibonacci Numbers*. Prometheus Books, Amherst, NY, USA, 2007. ISBN 1-59102-475-7; 978-1-59102-475-0. 385 pp. LCCN QA241 .P665 2007. {**15**}

[Pla92] P. J. Plauger. *The Standard C Library*. Prentice-Hall, Upper Saddle River, NJ, USA, 1992. ISBN 0-13-838012-0; 978-0-13-838012-0. xiv + 498 pp. LCCN QA76.73.C15 P563 1991. {**133, 827**}

[PM75] Robert Piessens and Irene Mertens. Remark and certification on "Algorithm 446: Ten subroutines for the manipulation of Chebyshev series". *Communications of the Association for Computing Machinery*, 18(5):276, 1975. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI 10.1145/360762.360782. See [Bro73]. {**1001**}

[PM84] John F. Palmer and Stephen P. Morse. *The 8087 Primer*. Wiley, New York, NY, USA, 1984. ISBN 0-471-87569-4; 978-0-471-87569-7. viii + 182 pp. LCCN QA76.8.I2923 P34 1984. Excellent coverage of the 8087 numeric coprocessor by the chief architects of the Intel 8087 (Palmer) and 8086 (Morse). Contains many candid statements about design decisions in these processors. A must for serious assembly language coding of the 8087 and 80287 chips. See also [Int85]. {**63, 104**}

[PM88] Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the Association for Computing Machinery*, 31(10):1192–1201, October 1988. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). URL http://www.acm.org/pubs/toc/Abstracts/0001-0782/63042.html. DOI 10.1145/63039.63042. {**170, 214**}

[Pól49] George Pólya. Remarks on computing the probability integral in one and two dimensions. In *Proceedings of the [First] Berkeley Symposium on Mathematical Statistics and Probability: held at the Statistical Laboratory, Department of Mathematics, University of California, August 13–18, 1945, January 27–29, 1946*, pages 63–78. University of California Press, Berkeley, CA, USA, 1949. LCCN QA276 .B4. URL http://projecteuclid.org/euclid.bsmsp/1166219199. {**618**}

[POP64] IBM Corporation, San Jose, CA, USA. *IBM System/360 Principles of Operation*, 1964. 168 pp. URL http://bitsavers.org/pdf/ibm/360/poo/A22-6821-0_360PrincOps.pdf. File number S360-01. Form A22-6821-5. {**928, 963, 964**}

[POP67] IBM Corporation, San Jose, CA, USA. *IBM System/360 Principles of Operation*, seventh edition, January 13, 1967. 175 pp. URL http://www.bitsavers.org/pdf/ibm/360/poo/A22-6821-6_360PrincOpsJan67.pdf. File number S360-01. Form A22-6821-6. {**964**}

[POP75] IBM Corporation, San Jose, CA, USA. *IBM System/370 Principles of Operation*, September 1, 1975. viii + 9–326 pp. URL http://bitsavers.org/pdf/ibm/360/poo/A22-6821-0_360PrincOps.pdf. File number S/360-01. Form GA22-7000-4. {**965**}

[Pop00] Bogdan A. Popov. Optimal starting approximation and iterative algorithm for inverse error function. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 34(1):25–26, March 2000. CODEN SIGSBZ. ISSN 0163-5824 (print), 1557-9492 (electronic). DOI 10.1145/373500.373510. {**600, 604**}

[POP04] IBM Corporation, Department 55JA Mail Station P384, 2455 South Road Poughkeepsie, NY, 12601-5400, USA. *z/Architecture Principles of Operation*, fourth edition, May 2004. xxvi + 1124 pp. URL http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/download/DZ9ZR003.pdf. IBM order number SA22-7832-03. {**963, 965**}

[PP05] Wolfgang K. H. Panofsky and Melba Phillips. *Classical Electricity and Magnetism*. Dover books on physics. Dover, New York, NY, USA, second edition, 2005. ISBN 0-486-43924-0; 978-0-486-43924-2. xvi + 494 pp. LCCN QC518 .P337 2005. {**693**}

[Pri91] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In Kornerup and Matula [KM91], pages 132–143. ISBN 0-8186-9151-4 (case), 0-8186-6151-8 (microfiche), 0-7803-0187-0 (library binding); 978-0-8186-9151-5 (case), 978-0-8186-6151-8 (microfiche), 978-0-7803-0187-0 (library binding). LCCN QA76.9.C62 S95 1991. DOI 10.1109/ARITH.1991.145549. IEEE catalog number 91CH3015-5. {**385, 407, 476**}

[Pri92] Douglas M. Priest. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. Thesis (Ph.D. in mathematics), Department of Computer Science, University of California, Berkeley, Berkeley, CA, USA, December 1992. iv + 136 pp. URL ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z. {**89**}

[Pri04] Douglas M. Priest. Efficient scaling for complex division. *ACM Transactions on Mathematical Software*, 30(4):389–401, December 2004. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI 10.1145/1039813.1039814. {**453–455, 1033**}

[PSLM00] P. J. Plauger, Alexander A. Stepanov, Meng Lee, and David R. Musser. *The C++ Standard Template Library*. Prentice-Hall, Upper Saddle River, NJ, USA, 2000. ISBN 0-13-437633-1; 978-0-13-437633-2. xii + 485 pp. LCCN QA76.73.C153 C17 2000. {**827**}

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes — The Art of Scientific Computing*. Cambridge University Press, Cambridge, UK, third edition, 2007. ISBN 0-521-88068-8 (hardcover), 0-521-88407-1 (with source code CD ROM), 0-521-70685-8 (source code CD ROM); 978-0-521-88068-8 (hardcover), 978-0-521-88407-5 (with source code CD ROM), 978-0-521-70685-8 (source code CD ROM). xxi + 1235 pp. LCCN QA297 .N866 2007. URL http://www.cambridge.org/numericalrecipes. {**19, 196, 567, 589, 657**}

[PW95] Ora E. Percus and Paula A. Whitlock. Theory and application of Marsaglia's monkey test for pseudorandom number generators. *ACM Transactions on Modeling and Computer Simulation*, 5(2):87–100, April 1995. CODEN ATMCEZ. ISSN 1049-3301 (print), 1558-1195 (electronic). DOI 10.1145/210330.210331. See [MZ93]. {**200, 1026**}

[Rai60] Earl David Rainville. *Special Functions*. Chelsea Publishing Company, New York, NY, USA, 1960. ISBN 0-8284-0258-2; 978-0-8284-0258-3. xii + 365 pp. LCCN QA351 .R3 1971. Reprinted in 1971. {**521, 827**}

[Ran82] Brian Randell, editor. *The Origins of Digital Computers: Selected Papers*. Texts and monographs in computer science. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., third edition, 1982. ISBN 0-387-11319-3, 3-540-11319-3; 978-0-387-11319-7, 978-3-540-11319-5. xvi + 580 pp. LCCN TK7885.A5 O741 1982. DOI `10.1007/978-3-642-61812-3`. This book collects many important early papers on computers from 1837 to 1949. {**104**}

[Ray04] Eric Steven Raymond. *The Art of UNIX Programming*. Addison-Wesley, Reading, MA, USA, 2004. ISBN 0-13-124085-4, 0-13-142901-9; 978-0-13-124085-8, 978-0-13-142901-7. xxxii + 525 pp. LCCN QA76.76.O63 R395 2003. {**956**}

[RB05a] Arnold Robbins and Nelson H. F. Beebe. *Classic Shell Scripting*. O'Reilly Media, Sebastopol, CA, USA, 2005. ISBN 0-596-00595-4; 978-0-596-00595-5. xxii + 534 pp. LCCN QA76.76.O63 R633 2005. URL `http://www.oreilly.com/catalog/shellsrptg/`. Also available in Chinese [RB08], French [RB05b], German [RB06a], Japanese [RB06c], and Polish [RB06b] translations. {**ix, 873, 1029**}

[RB05b] Arnold Robbins and Nelson H. F. Beebe. *Introduction aux Scripts Shell*. O'Reilly & Associates, Sebastopol, CA, USA, and Cambridge, MA, USA, 2005. ISBN 2-84177-375-2; 978-2-84177-375-6. xxii + 558 pp. URL `http://www.silicon.fr/getarticle.asp?id=14015`. French translation of [RB05a] by Eric Jacoboni. {**1029**}

[RB06a] Arnold Robbins and Nelson H. F. Beebe. *Klassische Shell-Programmierung: [automatisieren Sie Ihre Unix/Linux-Tasks]*. O'Reilly & Associates, Sebastopol, CA, USA, and Cambridge, MA, USA, 2006. ISBN 3-89721-441-5; 978-3-89721-441-5. xxiii + 572 pp. LCCN QA76.76.O63 R563 2005. URL `http://www.gbv.de/dms/hebis-darmstadt/toc/17645067X.pdf`. German translation of [RB05a] by Kathrin Lichtenberg. {**1029**}

[RB06b] Arnold Robbins and Nelson H. F. Beebe. *Programowanie Skryptów Powłoki*. Helion, Gliwice, Poland, 2006. ISBN 83-246-0131-7; 978-83-246-0131-8. 557 + 2 pp. URL `http://www.empik.com/b/o/19/f1/19f16b85e0d75ae1d3a1e7062569fbb0.jpg; http://www.empik.com/programowanie-skryptow-powloki-ksiazka,360529,p`. Polish translation of [RB05a] by Przemysław Szeremiota. {**1029**}

[RB06c] Arnold Robbins and Nelson H. F. Beebe. *Shōkai shieru sukuriputo*. Orairī Japan, Tōkyō, Japan, 2006. ISBN 4-87311-267-2; 978-4-87311-267-1. 345 pp. Japanese translation of [RB05a] by Aoi Hyūga. {**1029**}

[RB08] Arnold Robbins and Nelson H. F. Beebe. *Shell Jiao Ben Xue Xi Zhi Nan = Shell Script Study Guide*. O'Reilly Media, Sebastopol, CA, USA, 2008. ISBN 7-111-25504-6; 978-7-111-25504-8. vi + 494 pp. Simplified Chinese translation of [RB05a]. {**1029**}

[RBJ16] Siegfried M. Rump, Florian Bünger, and Claude-Pierre Jeannerod. Improved error bounds for floating-point products and Horner's scheme. *BIT Numerical Mathematics*, 56(1):293–307, March 2016. CODEN BITTEL, NBITAB. ISSN 0006-3835 (print), 1572-9125 (electronic). URL `http://link.springer.com/article/10.1007/s10543-015-0555-z`. DOI `10.1007/s10543-015-0555-z`. {**89**}

[Ree77] James A. Reeds. "Cracking" a random number generator. *Cryptologia*, 1(1):20–26, January 1977. CODEN CRYPE6. ISSN 0161-1194 (print), 1558-1586 (electronic). URL `http://alumni.cs.ucr.edu/~jsun/random-number.pdf; http://www.dean.usma.edu/math/pubs/cryptologia/ClassicArticleReprints/V01N1PP20-26JamesReeds.pdf; http://www.informaworld.com/smpp/content~content=a748865252~db=all~order=page`. DOI `10.1080/0161-117791832760`. Reprinted in [DKKM87, pp. 509–515]. {**207**}

[Ree79] James A. Reeds. Cracking a multiplicative congruential encryption algorithm. In *Information linkage between applied mathematics and industry (Proc. First Annual Workshop, Naval Postgraduate School, Monterey, Calif., 1978)*, pages 467–472. Academic Press, New York, NY, USA, 1979. DOI `10.1016/B978-0-12-734250-4.50037-0`. {**207**}

[Rei06] Constance Reid. *From Zero to Infinity: What Makes Numbers Interesting*. A. K. Peters, Wellesley, MA, USA, fifth edition, 2006. ISBN 1-56881-273-6; 978-1-56881-273-1. xvii + 188 pp. LCCN QA93 .R42 2006. {**59**}

[REXX96] *American National Standard for information technology: programming language REXX: ANSI X3.274-1996*. American National Standards Institute, New York, NY, USA, 1996. iv + 167 pp. {**viii, 968**}

[Rib91] Paulo Ribenboim. *The Little Book of Big Primes*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1991. ISBN 0-387-97508-X (New York), 3-540-97508-X (Berlin); 978-0-387-97508-5 (New York), 978-3-540-97508-3 (Berlin). xvii + 237 pp. LCCN QA246 .R472 1991. DOI `10.1007/978-1-4757-4330-2`. {**590**}

[Rib96] Paulo Ribenboim. *The New Book of Prime Number Records*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., third edition, 1996. ISBN 0-387-94457-5 (hardcover); 978-0-387-94457-9 (hardcover). xxiv + 541 pp. LCCN QA246 .R47 1996. DOI `10.1007/978-1-4612-0759-7`. {**590**}

[Rib04] Paulo Ribenboim. *The Little Book of Bigger Primes*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., second edition, 2004. ISBN 0-387-20169-6; 978-0-387-20169-6. xxiii + 356 pp. LCCN QA246 .R473 2004. DOI `10.1007/b97621`. {**590**}

[Ric64] John R. Rice. *The Approximation of Functions*, volume 1. Addison-Wesley, Reading, MA, USA, 1964. LCCN QA221 .R5 V.1-2. {**31**}

[Rip90] B. D. Ripley. Thoughts on pseudorandom number generators. *Journal of Computational and Applied Mathematics*, 31(1):153–163, July 24, 1990. CODEN JCAMDI. ISSN 0377-0427 (print), 1879-1778 (electronic). URL `http://www.sciencedirect.com/science/article/pii/0377042790903462`. DOI `10.1016/0377-0427(90)90346-2`. {**178**}

[Riv74] Theodore J. Rivlin. *The Chebyshev Polynomials*. Pure and applied mathematics. Wiley, New York, NY, USA, 1974. ISBN 0-471-72470-X; 978-0-471-72470-4. vi + 186 pp. LCCN QA404.5 .R58 1974. {**58**}

[Riv90] Theodore J. Rivlin. *Chebyshev Polynomials: From Approximation Theory to Algebra and Number Theory*. Pure and applied mathematics. Wiley, New York, NY, USA, second edition, 1990. ISBN 0-471-62896-4; 978-0-471-62896-5. xiii + 249 pp. LCCN QA404.5 .R58 1990. {**58**}

[Rob82] C. S. Roberts. Implementing and testing new versions of a good, 48-bit, pseudo-random number generator. *The Bell System Technical Journal*, 61(8):2053–2063, October 1982. CODEN BSTJAN. ISSN 0005-8580 (print), 2376-7154 (electronic). URL `http://bstj.bell-labs.com/BSTJ/images/Vol61/bstj61-8-2053.pdf`; `http://www.alcatel-lucent.com/bstj/vol61-1982/articles/bstj61-8-2053.pdf`. DOI `10.1002/j.1538-7305.1982.tb03099.x`. {**162**}

[Roc00] Daniel N. Rockmore. The FFT: An algorithm the whole family can use. *Computing in Science and Engineering*, 2(1):60–64, January/February 2000. CODEN CSENFA. ISSN 1521-9615 (print), 1558-366X (electronic). URL `http://dlib.computer.org/cs/books/cs2000/pdf/c1060.pdf`; `http://www.computer.org/cse/cs1999/c1060abs.htm`; `http://www.cs.dartmouth.edu/~rockmore/cse-fft.pdf`. DOI `10.1109/5992.814659`. [FFT = Fast Fourier Transform]. {**969**}

[Roc06] Daniel N. Rockmore. *Stalking the Riemann Hypothesis: the Quest to Find the Hidden Law of Prime Numbers*. Vintage Books, New York, NY, USA, 2006. ISBN 0-375-72772-8 (paperback); 978-0-375-72772-6 (paperback). x + 292 pp. LCCN QA246 .R63 2006. {**60, 303, 521, 579**}

[ROO08a] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate floating-point summation. Part I: Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224, 2008. CODEN SJOCE3. ISSN 1064-8275 (print), 1095-7197 (electronic). DOI `10.1137/050645671`. {**385**}

[ROO08b] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate floating-point summation. Part II: Sign, *K*-fold faithful and rounding to nearest. *SIAM Journal on Scientific Computing*, 31(2):1269–1302, 2008. CODEN SJOCE3. ISSN 1064-8275 (print), 1095-7197 (electronic). DOI `10.1137/07068816X`. {**385**}

[Ros11] Greg Rose. KISS: A bit too simple. Report, Qualcomm Inc., San Diego, CA, USA, April 18, 2011. URL `http://eprint.iacr.org/2011/007.pdf`. {**177**}

[RS92] Andrew Mansfield Rockett and Peter Szüsz. *Continued Fractions*. World Scientific Publishing, Singapore, 1992. ISBN 981-02-1047-7; 978-981-02-1047-2. ix + 188 pp. LCCN QA295.R6 1992; QA295 .R6 1992. {**19**}

[RSN^{+}01] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. *A Statistical Test Suite For Random and Pseudorandom Number Generators for Cryptographic Applications*. National Institute for Standards and Technology, Gaithersburg, MD, USA, May 15, 2001. 162 pp. URL `http://csrc.nist.gov/rng/rng2.html`; `http://csrc.nist.gov/rng/SP800-22b.pdf`; `http://csrc.nist.gov/rng/sts-1.5.tar`; `http://csrc.nist.gov/rng/sts.data.tar`; `http://csrc.nist.gov/rng/StsGui.zip`. NIST Special Publication 800-22. {**200**}

[Rud07] Peter Strom Rudman. *How Mathematics Happened: the First 50,000 Years*. Prometheus Books, Amherst, NY, USA, 2007. ISBN 1-59102-477-3; 978-1-59102-477-4. 314 pp. LCCN QA22 .R86 2007. {**59**}

[Rum09] Siegfried M. Rump. Ultimately fast accurate summation. *SIAM Journal on Scientific Computing*, 31(5):3466–3502, 2009. CODEN SJOCE3. ISSN 1064-8275 (print), 1095-7197 (electronic). DOI `10.1137/080738490`. {**385**}

[Rum12] Siegfried M. Rump. Error estimation of floating-point summation and dot product. *BIT Numerical Mathematics*, 52(1):201–220, March 2012. CODEN BITTEL, NBITAB. ISSN 0006-3835 (print), 1572-9125 (electronic). URL `http://www.springerlink.com/openurl.asp?genre=article&issn=0006-3835&volume=52&issue=1&spage=201`. DOI `10.1007/s10543-011-0342-4`. {**385**}

[Rus78] Richard M. Russell. The Cray-1 computer system. *Communications of the Association for Computing Machinery*, 21(1):63–72, January 1978. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/359327.359336`. {**952**}

[Ryd74] Barbara G. Ryder. The PFORT verifier. *Software — Practice and Experience*, 4(4):359–377, October/December 1974. CODEN SPEXBL. ISSN 0038-0644 (print), 1097-024X (electronic). DOI `10.1002/spe.4380040405`. {**823**}

[Sab03] Karl Sabbagh. *The Riemann Hypothesis: the Greatest Unsolved Problem in Mathematics*. Farrar, Straus and Giroux, New York, NY, USA, 2003. ISBN 0-374-25007-3; 978-0-374-25007-2. viii + 340 pp. LCCN QA241 .S23 2003. Originally published in 2002 as *Dr. Riemann's zeroes* by Grove Atlantic, London, UK. {**60, 303, 521, 579, 590**}

[SAI^{+}90] Herbert Schildt, American National Standards Institute, International Organization for Standardization, International Electrotechnical Commission, and ISO/IEC JTC 1. *The Annotated ANSI C Standard: American National Standard for Programming Languages C: ANSI/ISO 9899-1990*. Osborne/McGraw-Hill, Berkeley, CA, USA, 1990. ISBN 0-07-881952-0; 978-0-07-881952-0. xvi + 219 pp. LCCN QA76.73.C15S356 1990. URL `http://www.iso.ch/cate/d29237.html`. {**4**}

[Sal76] Eugene Salamin. Computation of π [pi] using arithmetic-geometric mean. *Mathematics of Computation*, 30(135):565–570, July 1976. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL `http://www.jstor.org/stable/2005327`. DOI `10.2307/2005327`. {**623**}

[Sal94] Peter H. Salus. *A Quarter Century of UNIX*. Addison-Wesley, Reading, MA, USA, 1994. ISBN 0-201-54777-5; 978-0-201-54777-1. xii + 256 pp. LCCN QA76.76.O63 S342 1994. {**956**}

[San07a] Charles Edward Sandifer. *The Early Mathematics of Leonhard Euler*, volume 1 of *Spectrum series; MAA tercentenary Euler celebration*. Mathematical Association of America, Washington, DC, USA, 2007. ISBN 0-88385-559-3; 978-0-88385-559-1. xix + 391 pp. LCCN QA29.E8 S26 2007. {**591**}

[San07b] Charles Edward Sandifer. *How Euler did it*, volume 3 of *The MAA tercentenary Euler celebration; Spectrum series*. Mathematical Association of America, Washington, DC, USA, 2007. ISBN 0-88385-563-1; 978-0-88385-563-8. xiv + 237 pp. LCCN QA29.E8. {**591**}

[SC08] Walter Schreppers and Annie Cuyt. Algorithm 871: A C/C++ precompiler for autogeneration of multiprecision programs. *ACM Transactions on Mathematical Software*, 34(1):5:1–5:20, January 2008. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/1322436.1322441`. {**410**}

[Sch78] J. L. Schonfelder. Chebyshev expansions for the error and related functions. *Mathematics of Computation*, 32(144):1232–1240, October 1978. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL `http://www.jstor.org/stable/2006347`. DOI `10.2307/2006347`. {**600**}

[Sch79a] Bruce W. Schmeiser. Miscellanea: Approximations to the inverse cumulative normal function for use on hand calculators. *Applied Statistics*, 28(2):175–176, 1979. CODEN APSTAG. ISSN 0035-9254 (print), 1467-9876 (electronic). URL `http://www.jstor.org/stable/2346737`. DOI `10.2307/2346737`. {**618**}

[Sch79b] Linus Schrage. A more portable Fortran random number generator. *ACM Transactions on Mathematical Software*, 5(2):132–138, June 1979. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/355826.355828`. {**174, 175, 214**}

[Sch81] Norman L. Schryer. A test of a computer's floating-point arithmetic unit. Technical Report Computer Science Technical Report 89, AT&T Bell Laboratories, February 1981. 66 pp. URL `http://plan9.bell-labs.com/cm/cs/cstr/89.ps.gz`. {**775**}

[Sch96] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Wiley, New York, NY, USA, second edition, 1996. ISBN 0-471-12845-7 (cloth), 0-471-11709-9 (paper); 978-0-471-12845-8 (cloth), 978-0-471-11709-4 (paper). xxiii + 758 pp. LCCN QA76.9.A25 S35 1996. {**207, 214, 591**}

[Sch00] Bruce Schneier. *Secrets and Lies: Digital Security in a Networked World*. Wiley, New York, NY, USA, 2000. ISBN 0-471-25311-1; 978-0-471-25311-2. xv + 412 pp. LCCN QA76.9.A25 S352 2000. {**214, 591**}

[Sch03] Bruce Schneier. *Beyond Fear: Thinking Sensibly about Security in an Uncertain World*. Copernicus Books, New York, NY, USA, 2003. ISBN 0-387-02620-7; 978-0-387-02620-6. 295 pp. LCCN HV6432 .S36 2003. {**591**}

[Sea05] Robert C. Seacord. *Secure Coding in C and C++*. Addison-Wesley, Reading, MA, USA, 2005. ISBN 0-321-33572-4 (paperback); 978-0-321-33572-2 (paperback). xxiv + 341 pp. LCCN QA76.9.A25 S368 2005. URL `http://www.cert.org/books/secure-coding/`. {**870**}

[Sei00] Charles Seife. *Zero: The Biography of a Dangerous Idea*. Viking, New York, NY, USA, 2000. ISBN 0-670-88457-X, 0-14-029647-6 (paperback); 978-0-670-88457-5, 978-0-14-029647-1 (paperback). vi + 248 pp. LCCN QA141 .S45 2000. {**59**}

[Set13] Sachin Seth. *Understanding Java Virtual Machine*. Alpha Science International, Oxford, UK, 2013. ISBN 1-84265-815-8; 978-1-84265-815-4. 318 pp. LCCN QA76.73.J38 S437 2013. {**viii, 979**}

[Sev98a] Charles Severance. An interview with the old man of floating-point. Reminiscences elicited from William Kahan. World-Wide Web document, February 1998. URL `http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`. A shortened version appears in [Sev98b]. {**63**}

[Sev98b] Charles Severance. Standards: IEEE 754: An interview with William Kahan. *Computer*, 31(3):114–115, March 1998. CODEN CPTRB4. ISSN 0018-9162 (print), 1558-0814 (electronic). URL `http://pdf.computer.org/co/books/co1998/pdf/r3114.pdf`. DOI `10.1109/MC.1998.10038`. {**63, 1031**}

[SF16] Wafaa S. Sayed and Hossam A. H. Fahmy. What are the correct results for the special values of the operands of the power operation? *ACM Transactions on Mathematical Software*, 42(2):14:1–14:17, June 2016. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/2809783`. {**413**}

[Sha45] Claude E. Shannon. A mathematical theory of cryptography. Memorandum MM 45-110-02, Bell Laboratories, Murray Hill, NJ, USA, September 1, 1945. 114 + 25 pp. Classified report. Superseded by [Sha49]. {**969, 1031**}

[Sha48a] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, July 1948. CODEN BSTJAN. ISSN 0005-8580 (print), 2376-7154 (electronic). DOI `10.1002/j.1538-7305.1948.tb01338.x`. From the first page: "If the base 2 is used the resulting units may be called binary digits, or more briefly, *bits*, a word suggested by J. W. Tukey.". This is the first known printed instance of the word 'bit' with the meaning of binary digit. {**969**}

[Sha48b] Claude E. Shannon. A mathematical theory of communication (continued). *The Bell System Technical Journal*, 27(4):623–656, October 1948. CODEN BSTJAN. ISSN 0005-8580 (print), 2376-7154 (electronic). DOI `10.1002/j.1538-7305.1948.tb00917.x`. {**969**}

[Sha49] Claude E. Shannon. Communication theory of secrecy systems. *The Bell System Technical Journal*, 28(4):656–715, October 1949. CODEN BSTJAN. ISSN 0005-8580 (print), 2376-7154 (electronic). URL `http://bstj.bell-labs.com/BSTJ/images/Vol28/bstj28-4-656.pdf`; `http://en.wikipedia.org/wiki/Communication_Theory_of_Secrecy_Systems`; `http://www.cs.ucla.edu/~jkong/research/security/shannon1949.pdf`. DOI `10.1002/j.1538-7305.1949.tb00928.x`. A footnote on the initial page says: "The material in this paper appeared in a confidential report, 'A Mathematical Theory of Cryptography', dated Sept. 1, 1945 ([Sha45]), which has now been declassified.". This paper is sometimes cited as the foundation of modern cryptography. {**1031**}

[Sho82] Haim Shore. Simple approximations for the inverse cumulative function, the density function and the loss integral of the normal distribution. *Applied Statistics*, 31(2):108–114, 1982. CODEN APSTAG. ISSN 0035-9254 (print), 1467-9876 (electronic). URL `http://www.jstor.org/stable/2347972`. DOI `10.2307/2347972`. {**618**}

[Sid03] Avram Sidi. *Practical Extrapolation Methods: Theory and Applications*, volume 10 of *Cambridge monographs on applied and computational mathematics*. Cambridge University Press, Cambridge, UK, 2003. ISBN 0-521-66159-5; 978-0-521-66159-1. xxii + 519 pp. LCCN QA281 .S555 2003. {**589**}

[Sig02] L. E. Sigler. *Fibonacci's Liber Abaci: A Translation into Modern English of Leonardo Pisano's Book of Calculation*. Sources and studies in the history of mathematics and physical sciences. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2002. ISBN 0-387-95419-8; 978-0-387-95419-6. viii + 636 pp. LCCN QA32 .F4713 2002. DOI `10.1007/978-1-4613-0079-3`. This historically important book is the first English translation of the original Latin edition of 1202, on the 800th anniversary of the book that introduced to Europe the Hindu numerals 0 through 9, the word zero, the notion of an algorithm, and the subject of algebra. {**15, 59, 575**}

[Sil06] Joseph H. Silverman. *A Friendly Introduction to Number Theory*. Pearson Prentice Hall, Upper Saddle River, NJ 07458, USA, third edition, 2006. ISBN 0-13-186137-9; 978-0-13-186137-4. vii + 434 pp. LCCN QA241 .S497 2006. {**186**}

[Sin97] Simon Singh. *Fermat's Enigma: The Epic Quest to Solve the World's Greatest Mathematical Problem*. Walker and Company, New York, NY, USA, 1997. ISBN 0-8027-1331-9; 978-0-8027-1331-5. xiii + 315 pp. LCCN QA244.S55 1997. {**60**}

[Sin99] Simon Singh. *The Code Book: the Evolution of Secrecy from Mary, Queen of Scots, to Quantum Cryptography*. Doubleday, New York, NY, USA, 1999. ISBN 0-385-49531-5; 978-0-385-49531-8. xiii + 402 pp. LCCN Z103 .S56 1999. {**214, 591**}

[SK99] Eric M. Schwarz and C. A. Krygowski. The S/390 G5 floating-point unit. *IBM Journal of Research and Development*, 43(5/6):707–721, September/November 1999. CODEN IBMJAE. ISSN 0018-8646 (print), 2151-8556 (electronic). URL `http://www.research.ibm.com/journal/rd/435/schwarz.html`. DOI `10.1147/rd.435.0707`. {**963**}

[SKC09] Eric M. Schwarz, John S. Kapernick, and Michael F. Cowlishaw. Decimal floating-point support on the IBM System z10 processor. *IBM Journal of Research and Development*, 53(1):4:1–4:10, January/February 2009. CODEN IBMJAE. ISSN 0018-8646 (print), 2151-8556 (electronic). URL `http://www.research.ibm.com/journal/rd/531/schwarz.pdf`. DOI `10.1147/JRD.2009.5388585`. {**927**}

[Sko75] Ove Skovgaard. Remark on "Algorithm 236: Bessel functions of the first kind [S17]". *ACM Transactions on Mathematical Software*, 1 (3):282–284, September 1975. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/355644.355653`. See [Gau64]. {**693, 1011**}

[SLF14] Guy L. Steele Jr., Doug Lea, and Christine H. Flood. Fast splittable pseudorandom number generators. *ACM SIGPLAN Notices*, 49(10):453–472, October 2014. CODEN SINODQ. ISSN 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI `10.1145/2714064.2660195`. {**1024**}

[Slo07] Neil J. A. Sloane. The on-line encyclopedia of integer sequences. Web database, 2007. URL `http://oeis.org/`. See also [SP95]. {**627, 629, 1033**}

[SLZ02] Damien Stehlé, Vincent Lefèvre, and Paul Zimmermann. Worst cases and lattice reduction. Research report, LORIA/INRIA Lorraine, Villers-lès-Nancy Cedex, France, October 15, 2002. 10 pp. URL `http://www.loria.fr/~zimmerma/papers/wclr.ps.gz`. {**28**}

[SLZ03] Damien Stehlé, Vincent Lefèvre, and Paul Zimmermann. Worst cases and lattice reduction. In Bajard and Schulte [BS03], pages 142–147. ISBN 0-7695-1894-X; 978-0-7695-1894-7. ISSN 1063-6889. LCCN QA76.6 .S919 2003. URL `http://www.dec.usc.es/arith16/`. DOI `10.1109/ARITH.2003.1207672`. {**28**}

[SLZ05] Damien Stehlé, Vincent Lefèvre, and Paul Zimmermann. Searching worst cases of a one-variable function using lattice reduction. *IEEE Transactions on Computers*, 54(3):340–346, March 2005. CODEN ITCOB4. ISSN 0018-9340 (print), 1557-9956 (electronic). URL `http://csdl.computer.org/dl/trans/tc/2005/03/t0340.pdf`. DOI `10.1109/TC.2005.55`. {**28**}

[SM12] Mutsuo Saito and Makoto Matsumoto. A deviation of CURAND: Standard pseudorandom number generator in CUDA for GPGPU. Slides presented at the Tenth International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing, February 2012. URL `http://www.mcqmc2012.unsw.edu.au/slides/MCQMC2012_Matsumoto.pdf`. {**1024**}

[SMDS11] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: as easy as $1, 2, 3$. In Scott Lathrop, Jim Costa, and William Kramer, editors, *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, Seattle, WA, November 12–18 2011*, pages 16:1–16:12. ACM Press and IEEE Computer Society Press, New York, NY 10036, USA and Silver Spring, MD, USA, 2011. ISBN 1-4503-0771-X; 978-1-4503-0771-0. LCCN QA76.5 .S96 2011. DOI `10.1145/2063384.2063405`. {**1024**}

[Smi58] David Eugene Smith. *History of Mathematics*. Dover histories, biographies and classics of mathematics and the physical sciences. Dover, New York, NY, USA, 1958. ISBN 0-486-20430-8, 0-486-20429-4; 978-0-486-20430-7, 978-0-486-20429-1. xii + 725 pp. LCCN QA21 .S62. {**59**}

[Smi62] Robert L. Smith. Algorithm 116: Complex division. *Communications of the Association for Computing Machinery*, 5(8):435, August 1962. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/368637.368661`. See also an improved version-with-scaling of this algorithm [LDB^{+}00]. {**451–453, 455, 1021**}

[Smi75] Alan Jay Smith. Comments on a paper by T. C. Chen and I. T. Ho. *Communications of the Association for Computing Machinery*, 18(8):463, August 1975. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/360933.360986`. See [CH75]. {**1003**}

[Smi91] David M. Smith. Algorithm 693: A FORTRAN package for floating-point multiple-precision arithmetic. *ACM Transactions on Mathematical Software*, 17(2):273–283, June 1991. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.acm.org/pubs/citations/journals/toms/1991-17-2/p273-smith/`. DOI `10.1145/108556.108585`. {**827**}

[Smi95] Roger Alan Smith. A continued-fraction analysis of trigonometric argument reduction. *IEEE Transactions on Computers*, 44(11):1348–1351, November 1995. CODEN ITCOB4. ISSN 0018-9340 (print), 1557-9956 (electronic). DOI `10.1109/12.475133`. {**251, 253**}

[Smi98] David M. Smith. Algorithm 786: Multiple-precision complex arithmetic and functions. *ACM Transactions on Mathematical Software*, 24 (4):359–367, December 1998. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/293686.293687`. See also [Bai95, Bre78b, Bre79, BHY80]. {**476, 509, 827, 997, 1000, 1001**}

[SN05] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2005. ISBN 1-55860-910-5; 978-1-55860-910-5. xxii + 638 pp. LCCN QA76.9.V5 S54 2005. URL `http://www.elsevierdirect.com/product.jsp?isbn=9781558609105`. {**viii**}

[SO87] Jerome Spanier and Keith B. Oldham. *An Atlas of Functions*. Hemisphere Publishing Corporation, Washington, DC, USA, 1987. ISBN 0-89116-573-8, 3-540-17395-1; 978-0-89116-573-6, 978-3-540-17395-3. ix + 700 pp. LCCN QA331 .S685 1987. See also the second edition [OMS09]. {**521, 556, 558, 593, 657**}

[Sor95] Jonathan Sorenson. An analysis of Lehmer's Euclidean GCD algorithm. In A. H. M. Levelt, editor, *ISSAC '95: Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation: July 10–12, 1995, Montréal, Canada*, ISSAC -PROCEEDINGS- 1995, pages 254–258. ACM Press, New York, NY 10036, USA, 1995. ISBN 0-89791-699-9; 978-0-89791-699-8. LCCN QA 76.95 I59 1995. URL `http://www.acm.org:80/pubs/citations/proceedings/issac/220346/p254-sorenson/`. DOI `10.1145/220346.220378`. ACM order number 505950. {**184**}

[SP95] Neil J. A. Sloane and Simon Plouffe. *The Encyclopedia of Integer Sequences*. Academic Press, New York, NY, USA, 1995. ISBN 0-12-558630-2; 978-0-12-558630-6. xiii + 587 pp. LCCN QA246.5 .S66 1995. URL `http://oeis.org/`. See also the more-recent online resource [Slo07]. {**524, 568, 572, 576, 1032**}

[SR05] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the Unix Environment*. Addison-Wesley, Reading, MA, USA, second edition, 2005. ISBN 0-201-43307-9 (hardcover); 978-0-201-43307-4 (hardcover). xxviii + 927 pp. LCCN QA76.76.O63 S754 2005. {**91**}

[SS66] A. H. Stroud and Don Secrest. *Gaussian Quadrature Formulas*. Prentice-Hall, Upper Saddle River, NJ, USA, 1966. ix + 374 pp. LCCN QA299.4.G3 S7 1966. {**560**}

[SS94] Jeffrey Shallit and Jonathan Sorenson. Analysis of a left-shift binary GCD algorithm. *Journal of Symbolic Computation*, 17(6):473–486, June 1994. CODEN JSYCEH. ISSN 0747-7171 (print), 1095-855X (electronic). URL `http://www.sciencedirect.com/science/article/pii/S0747717184710303`. DOI `10.1006/jsco.1994.1030`. {**184**}

[SSB01] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: definition, verification, validation*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2001. ISBN 3-540-42088-6; 978-3-540-42088-0. x + 381 pp. LCCN QA76.73.J38 S785 2001. DOI `10.1007/978-3-642-59495-3`. Includes CD-ROM with the entire text of the book and numerous examples and exercises. {**viii, 979**}

[Ste67] Josef Stein. Computational problems associated with Racah algebra. *Journal of Computational Physics*, 1(3):397–405, February 1967. CODEN JCTPAH. ISSN 0021-9991 (print), 1090-2716 (electronic). URL `http://www.sciencedirect.com/science/article/pii/0021999167900472`. DOI `10.1016/0021-9991(67)90047-2`. {**184**}

[Ste74] Pat H. Sterbenz. *Floating Point Computation*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Upper Saddle River, NJ, USA, 1974. ISBN 0-13-322495-3; 978-0-13-322495-5. xiv + 316 pp. LCCN QA76.8.I12 S771 1974. {**948**}

[Ste81a] David Stevenson. A proposed standard for binary floating-point arithmetic. *Computer*, 14(3):51–62, March 1981. CODEN CPTRB4. ISSN 0018-9162 (print), 1558-0814 (electronic). URL `http://ieeexplore.ieee.org/document/1667284/`. DOI `10.1109/C-M.1981.220377`. See [IEEE85a]. {**63, 1016**}

[Ste81b] David Stevenson. *A Proposed Standard for Binary Floating-Point Arithmetic: Draft 8.0 of IEEE Task P754*. IEEE Computer Society Press, Silver Spring, MD, USA, 1981. 36 pp. See [IEEE85a]. {**63, 1016**}

[Ste84] R. G. Stewart. P854 working group completes radix-independent floating-point draft. *IEEE Micro*, 4(1):82–83, February 1984. CODEN IEMIDZ. ISSN 0272-1732 (print), 1937-4143 (electronic). DOI `10.1109/MM.1984.291326`. {**104**}

[Ste85] G. W. Stewart. A note on complex division. *ACM Transactions on Mathematical Software*, 11(3):238–241, September 1985. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.acm.org/pubs/citations/journals/toms/1985-11-3/p238-stewart/`. DOI `10.1145/214408.214414`. See corrigendum [Ste86] and the faster and more robust algorithm in [Pri04]. {**452, 453, 455, 476, 1033**}

[Ste86] G. W. Stewart. Corrigendum: "A note on complex division". *ACM Transactions on Mathematical Software*, 12(3):285, September 1986. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/7921.356182`. See [Ste85]. {**1033**}

[Ste90] Guy L. Steele Jr. *Common Lisp — The Language*. Digital Press, Bedford, MA, USA, second edition, 1990. ISBN 1-55558-041-6 (paperback), 1-55558-042-4 (hardcover), 0-13-152414-3 (Prentice-Hall); 978-1-55558-041-4 (paperback), 978-1-55558-042-1 (hardcover), 978-0-13-152414-9 (Prentice-Hall). xxiii + 1029 pp. LCCN QA76.73.L23 S73 1990. URL `http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html`. {**341, 476**}

[Ste93] Frank Stenger. *Numerical Methods Based on Sinc and Analytic Functions*, volume 20 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1993. ISBN 0-387-94008-1 (New York), 3-540-94008-1 (Berlin); 978-0-387-94008-3 (New York), 978-3-540-94008-1 (Berlin). xv + 565 pp. LCCN QA372 .S82 1993. DOI `10.1007/978-1-4612-2706-9`. {**733**}

[Ste11] Guy L. Steele Jr. An interview with Frances E. Allen. *Communications of the Association for Computing Machinery*, 54(1):39–45, January 2011. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/1866739.1866752`. This article contains an important half-century retrospective on the IBM 7030 Stretch project, and its impact on subsequent computer designs. {**959**}

[Sti80] George R. Stibitz. Early computers. In Nicholas Metropolis, Jack Howlett, and Gian-Carlo Rota, editors, *A History of Computing in the Twentieth Century: A Collection of Essays*, pages 479–483. Academic Press, New York, NY, USA, 1980. ISBN 0-12-491650-3; 978-0-12-491650-0. LCCN QA75.5 .I63 1976. DOI `10.1016/B978-0-12-491650-0.50034-4`. Original versions of these papers were presented at the International Research Conference on the History of Computing, held at the Los Alamos Scientific Laboratory, 10–15 June 1976. {**463**}

[Sti02] John Stillwell. *Mathematics and its History*. Undergraduate texts in mathematics. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., second edition, 2002. ISBN 0-387-95336-1; 978-0-387-95336-6. xviii + 542 pp. LCCN QA21 .S84 2002. DOI `10.1007/978-1-4684-9281-1`. {**59, 541**}

[Str59] C. Strachey. On taking the square root of a complex number. *The Computer Journal*, 2(2):89, July 1959. CODEN CMPJA6. ISSN 0010-4620 (print), 1460-2067 (electronic). URL `http://www3.oup.co.uk/computer_journal/hdb/Volume_02/Issue_02/020089.sgm.abs.html`; `http://www3.oup.co.uk/computer_journal/hdb/Volume_02/Issue_02/tiff/89.tif`. DOI 10.1093/comjnl/2.2.89. {**481**}

[Str68] Anthony J. Strecok. On the calculation of the inverse of the error function. *Mathematics of Computation*, 22(101):144–158, January 1968. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL `http://www.jstor.org/stable/2004772`. DOI `10.2307/2004772`. {**600, 603**}

[Stu95] Students of Prof.William M.Kahan. UCBTEST: a suite of programs for testing certain difficult cases of IEEE 754 floating-point arithmetic. World-Wide Web document, March 12, 1995. URL `http://www.netlib.org/fp/ucbtest.tgz`. From the source code, students and authors credited are (in alphabetical order) M. Alemi, D. Feenberg, Warren Ferguson, David G. Hough, David Gay, William J. Cody, Jr., R. Karkinski, Zhi-Shun Alex Liu, S. Ma, Stephen Moshier, M. Mueller, K. C. Ng, Douglas M. Priest, T. Quarles, T. Sumner, G. Taylor, B. Toy, William Waite, and Brian A. Wichmann. {**774**}

[Suz02] Jeff Suzuki. *A History of Mathematics*. Prentice-Hall, Upper Saddle River, NJ, USA, 2002. ISBN 0-13-019074-8; 978-0-13-019074-1. xiii + 815 pp. LCCN QA21 .S975 2002. {**59**}

[SvG12] Stefan Siegel and Jürgen Wolff von Gudenberg. A long accumulator like a carry-save adder. *Computing*, 94(2–4):203–213, March 2012. CODEN CMPTA2. ISSN 0010-485X (print), 1436-5057 (electronic). URL `http://www.springerlink.com/openurl.asp?genre=article&issn=0010-485X&volume=94&issue=2&spage=203`. DOI `10.1007/s00607-011-0164-x`. {**385**}

[SW90] Guy L. Steele Jr. and Jon L. White. How to print floating-point numbers accurately. *ACM SIGPLAN Notices*, 25(6):112–126, June 1990. CODEN SINODQ. ISBN 0-89791-364-7; 978-0-89791-364-5. ISSN 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). URL `http://www.acm.org:80/pubs/citations/proceedings/pldi/93542/p112-steele/`. DOI `10.1145/93548.93559`. See also input algorithm in [Cli90, Cli04], and a faster output algorithm in [BD96] and [Knu90], IBM S/360 algorithms in [ABC$^+$99] for both IEEE 754 and S/360 formats, and a twenty-year retrospective in [SW04]. In electronic mail dated Wed, 27 Jun 1990 11:55:36 EDT, Guy Steele reported that an intrepid pre-SIGPLAN 90 conference implementation of what is stated in the paper revealed 3 mistakes:

1. Table 5 (page 124):
   insert `k <- 0` after assertion, and also delete `k <- 0` from Table 6.
2. Table 9 (page 125):
   for `-1:USER!("");`
   substitute `-1:USER!("0");`
   and delete the comment.
3. Table 10 (page 125):
   for `fill(-k, "0")`
   substitute `fill(-k-1, "0")`

{**895, 896, 995, 998, 1004, 1020, 1034**}

[SW95] Richard L. Sites and Richard L. Witek. *Alpha AXP Architecture Reference Manual*. Digital Press, Bedford, MA, USA, second edition, 1995. ISBN 1-55558-145-5; 978-1-55558-145-9. LCCN QA76.9.A73A46 1995. {**107**}

[SW04] Guy L. Steele Jr. and Jon L. White. Retrospective: How to print floating-point numbers accurately. *ACM SIGPLAN Notices*, 39(4):372–389, April 2004. CODEN SINODQ. ISSN 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI `10.1145/989393.989431`. Best of PLDI 1979–1999. Reprint of, and retrospective on, [SW90]. {**895, 995, 998, 1004, 1020, 1034**}

[SW05] Alicja Smoktunowicz and Iwona Wróbel. On improving the accuracy of Horner's and Goertzel's algorithms. *Numerical Algorithms*, 38 (4):243–258, April 2005. CODEN NUALEG. ISSN 1017-1398 (print), 1572-9265 (electronic). DOI `10.1007/s11075-004-4570-4`. {**89**}

[Swa90a] Earl E. Swartzlander, Jr. *Computer Arithmetic*, volume 1. IEEE Computer Society Press, Silver Spring, MD, USA, 1990. ISBN 0-8186-8931-5; 978-0-8186-8931-4. xiii + 378 pp. LCCN QA76.6 .C633 1990. This is part of a two-volume collection of influential papers on the design of computer arithmetic. See also [Swa90b]. {**104, 978, 1034**}

[Swa90b] Earl E. Swartzlander, Jr. *Computer Arithmetic*, volume 2. IEEE Computer Society Press, Silver Spring, MD, USA, 1990. ISBN 0-8186-8945-5; 978-0-8186-8945-1. ix + 396 pp. LCCN QA76.9 .C62C66 1990. This is part of a two-volume collection of influential papers on the design of computer arithmetic. See also [Swa90a]. {**104, 966, 978, 1034**}

[Swe65] D. W. Sweeney. An analysis of floating-point addition. *IBM Systems Journal*, 4(1):31–42, 1965. CODEN IBMSA7. ISSN 0018-8670. URL `http://www.research.ibm.com/journal/sj/041/ibmsjIVRID.pdf`. DOI `10.1147/sj.41.0031`. This important paper describes the analysis that led to the adoption of a hexadecimal base for floating-point arithmetic in IBM System/360, an unfortunate decision that numerical analysts later came to deplore. Reprinted in [Swa90a, pp. 317–328]. {**963**}

[SZ49] Herbert E. Salzer and Ruth Zucker. Tables of the zeros and weight factors of the first fifteen Laguerre polynomials. *Bulletin of the American Mathematical Society*, 55(10):1004–1012, October 1949. CODEN BAMOAD. ISSN 0002-9904 (print), 1936-881x (electronic). URL `http://projecteuclid.org/euclid.bams/1183514167`. {**560**}

[SZ04] Damien Stehlé and Paul Zimmermann. Gal's accurate tables method revisited. World-Wide Web document, 2004. URL `http://www.loria.fr/~stehle/downloads/2x-double.txt`; `http://www.loria.fr/~stehle/downloads/sincos-double.txt`; `http://www.loria.fr/~stehle/IMPROVEDGAL.html`. {**28**}

[SZ05] Damien Stehlé and Paul Zimmermann. Gal's accurate tables method revisited. In Montuschi and Schwarz [MS05], pages 275–264. ISBN 0-7695-2366-8; 978-0-7695-2366-8. LCCN QA76.9.C62 .S95 2005. URL `http://arith17.polito.it/final/paper-152.pdf`. DOI `10.1109/ARITH.2005.24`. {**28**}

[Szp03] George Szpiro, editor. *Kepler's Conjecture: How Some of the Greatest Minds in History Helped Solve One of the Oldest Math Problems in the World*. Wiley, New York, NY, USA, 2003. ISBN 0-471-08601-0; 978-0-471-08601-7. viii + 296 pp. LCCN QA93 .S97 2003. {**60**}

[Szp07] George Szpiro. *Poincaré's Prize: The Hundred-Year Quest to Solve One of Math's Greatest Puzzles*. Dutton, New York, NY, USA, 2007. ISBN 0-525-95024-9; 978-0-525-95024-0. ix + 309 pp. LCCN QA43 .S985 2007. {**60**}

[Tan89] Ping Tak Peter Tang. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 15(2):144–157, June 1989. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/63522.214389`. {**271**}

[Tan06] Hui-Chin Tang. An exhaustive analysis of two-term multiple recursive random number generators with efficient multipliers. *Journal of Computational and Applied Mathematics*, 192(2):411–416, August 2006. CODEN JCAMDI. ISSN 0377-0427 (print), 1879-1778 (electronic). URL `http://dl.acm.org/citation.cfm?id=1148032.1148047`. DOI `10.1016/j.cam.2005.06.001`. {**170, 176**}

[Tau63] A. H. Taub, editor. *John von Neumann: Collected Works. Volume V: Design of Computers, Theory of Automata and Numerical Analysis*. Pergamon, New York, NY, USA, 1963. ix + 784 pp. {**1036**}

[TB86] I. J. Thompson and A. R. Barnett. Coulomb and Bessel functions of complex arguments and order. *Journal of Computational Physics*, 64 (2):490–509, June 1986. CODEN JCTPAH. ISSN 0021-9991 (print), 1090-2716 (electronic). DOI `10.1016/0021-9991(86)90046-X`. {**18**}

[TC11] Hui-Chin Tang and Hwapeng Chang. An exhaustive search for good 64-bit linear congruential random number generators with restricted multiplier. *Computer Physics Communications*, 182(11):2326–2330, November 2011. CODEN CPHCBZ. ISSN 0010-4655 (print), 1879-2944 (electronic). URL `http://www.sciencedirect.com/science/article/pii/S0010465511002360`. DOI `10.1016/j.cpc.2011.06.013`. {**170**}

[Tem96] Nico M. Temme. *Special Functions: An Introduction to the Classical Functions of Mathematical Physics*. Wiley, New York, NY, USA, 1996. ISBN 0-471-11313-1; 978-0-471-11313-3. xii + 374 pp. LCCN QC20.7.F87 T46 1996. {**521, 627, 827**}

[Ten06] M. B. W. Tent. *The Prince of Mathematics: Carl Friedrich Gauss*. A. K. Peters, Wellesley, MA, USA, 2006. ISBN 1-56881-261-2; 978-1-56881-261-8. xviii + 245 pp. LCCN QA29.G3 T46 2006. {**59**}

[Ten09] M. B. W. Tent. *Leonhard Euler and the Bernoullis: Mathematicians from Basel*. A. K. Peters, Wellesley, MA, USA, 2009. ISBN 1-56881-464-X; 978-1-56881-464-3. xix + 276 pp. LCCN QA28 .T46 2009. {**591**}

[TGNSC11] Charles Tsen, Sonia Gonzalez-Navarro, Michael J. Schulte, and Katherine Compton. Hardware designs for binary integer decimal-based rounding. *IEEE Transactions on Computers*, 60(5):614–627, May 2011. CODEN ITCOB4. ISSN 0018-9340 (print), 1557-9956 (electronic). DOI `10.1109/TC.2010.268`. {**929**}

[Tha63] Henry C. Thacher, Jr. Certification of Algorithm 147 [S14]: PSIF. *Communications of the Association for Computing Machinery*, 6(4):168, April 1963. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/366349.366537`. See [Ami62, Par69]. {**996, 1027**}

[Tho70] James E. Thornton. *Design of a Computer: the Control Data 6600*. Scott, Foresman, Glenview, IL, USA, 1970. 181 pp. LCCN TK7889.C6 T5 1970; TK7889.C2 T5. {**951**}

[Tho80] James E. Thornton. The CDC 6600 project. *Annals of the History of Computing*, 2(4):338–348, October/December 1980. CODEN AHCOE5. ISSN 0164-1239. URL `http://dlib.computer.org/an/books/an1980/pdf/a4338.pdf`. DOI `10.1109/MAHC.1980.10044`. {**949, 951**}

[Tho97] William J. Thompson. *Atlas for Computing Mathematical Functions: an Illustrated Guide for Practitioners with Programs in Fortran 90 and Mathematica*. Wiley, New York, NY, USA, 1997. ISBN 0-471-18171-4 (cloth); 978-0-471-18171-2 (cloth). xiv + 888 pp. LCCN QA331.T386 1997. Includes CD-ROM. {**520, 521, 556, 558, 567, 583, 593, 595, 644, 657, 693, 827**}

[TLC93] Shu Tezuka, Pierre L'Ecuyer, and Raymond Couture. On the lattice structure of the add-with-carry and subtract-with-borrow random number generators. *ACM Transactions on Modeling and Computer Simulation*, 3(4):315–331, October 1993. CODEN ATMCEZ. ISSN 1049-3301 (print), 1558-1195 (electronic). DOI `10.1145/159737.159749`. See remark in [EH95, page 248], and [MZ91] for the original work analyzed in this paper. {**177, 1026**}

[TS69] Adilson Tadeu de Medeiros and Georges Schwachheim. Algorithm 349: Polygamma functions with arbitrary precision. *Communications of the Association for Computing Machinery*, 12(4):213–214, April 1969. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). DOI `10.1145/362912.362928`. See certification [Lew75]. {**521, 552, 555, 558, 1021**}

[TSGN07] Charles Tsen, Michael J. Schulte, and Sonia Gonzalez-Navarro. Hardware design of a binary integer decimal-based IEEE P754 rounding unit. In IEEE, editor, *ASAP 07: conference proceedings: IEEE 18th International Conference on Application-Specific Systems, Architectures, and Processors: Montréal, Canada: July 8–11, 2007*, pages 115–121. IEEE Computer Society Press, Silver Spring, MD, USA, 2007. ISBN 1-4244-1027-4; 978-1-4244-1027-9. LCCN TK7874.6 .I57a 2007. URL `http://ieeexplore.ieee.org/servlet/opac?punumber=4429947`. DOI `10.1109/ASAP.2007.4429967`. {**929**}

[TSSK07] Son Dao Trong, Martin Schmookler, Eric M. Schwarz, and Michael Kroener. P6 binary floating-point unit. In Kornerup and Muller [KM07], pages 77–86. ISBN 0-7695-2854-6; 978-0-7695-2854-0. ISSN 1063-6889. LCCN QA76.9.C62. URL `http://www.lirmm.fr/arith18/`. DOI `10.1109/ARITH.2007.26`. The P6 processor is the sixth generation of the IBM POWER and PowerPC architecture. {**927**}

[Tuk58] John W. Tukey. The teaching of concrete mathematics. *American Mathematical Monthly*, 65(1):1–9, January 1958. CODEN AMMYAE. ISSN 0002-9890 (print), 1930-0972 (electronic). DOI `10.2307/2310294`. This article is believed to contain the first published instance of the word 'software' in the meaning of instructions to a computer: "Today the 'software' comprising the carefully planned interpretive routines, compilers, and other aspects of automative programming are at least as important to the modern electronic calculator as its 'hardware' of tubes, transistors, wires, tapes and the like." [page 2]. {**969**}

[Tur87] Peter R. Turner. The distribution of l.s.d. and its implications for computer design. *The Mathematical Gazette*, 71(455):26–31, March 1987. CODEN MAGAAS. ISSN 0025-5572 (print), 2056-6328 (electronic). DOI `10.2307/3616283`. [l.s.d. = least significant digits]. The topic is variously known as Benford's Law, the Law of Anomalous Numbers, and Zipf's Law. {**964**}

[TW99] K. V. Tretiakov and K. W. Wojciechowski. Efficient Monte Carlo simulations using a shuffled nested Weyl sequence random number generator. *Physical Review E (Statistical physics, plasmas, fluids, and related interdisciplinary topics)*, 60(6):7626–7628, December 1999. CODEN PLEEE8. ISSN 1539-3755 (print), 1550-2376 (electronic). URL `http://link.aps.org/doi/10.1103/PhysRevE.60.7626`. DOI `10.1103/PhysRevE.60.7626`. {**177**}

[Ueb97] Christoph W. Ueberhuber. *Numerical Computation: Methods, Software, and Analysis*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1997. ISBN 3-540-62058-3 (vol. 1: softcover), 3-540-62057-5 (vol. 2: softcover); 978-3-540-62058-7 (vol. 1: softcover), 978-3-540-62057-0 (vol. 2: softcover). xvi + 474 (vol. 1), xvi + 495 (vol. 2) pp. LCCN QA297 .U2413 1997. DOI `10.1007/978-3-642-59118-1`. {**627**}

[vA87] Walter van Assche. *Asymptotics for Orthogonal Polynomials*, volume 1265 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1987. ISBN 0-387-18023-0 (paperback); 978-0-387-18023-6 (paperback). vi + 201 pp. LCCN QA3 .L28 no. 1265; QA404.5. DOI `10.1007/BFb0081880`. {**59**}

[Van92] Charles F. Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM (Society for Industrial and Applied Mathematics), Philadelphia, PA, USA, 1992. ISBN 0-89871-285-8; 978-0-89871-285-8. xiii + 273 pp. LCCN QA403.5 .V35 1992. {**299**}

[VC06] Joris Van Deun and Ronald Cools. Algorithm 858: Computing infinite range integrals of an arbitrary product of Bessel functions. *ACM Transactions on Mathematical Software*, 32(4):580–596, December 2006. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/1186785.1186790`. {**693**}

[VCV01a] Brigitte Verdonk, Annie Cuyt, and Dennis Verschaeren. A precision- and range-independent tool for testing floating-point arithmetic I: Basic operations, square root, and remainder. *ACM Transactions on Mathematical Software*, 27(1):92–118, March 2001. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.win.ua.ac.be/~cant/ieeecc754.html`. DOI `10.1145/382043.382404`. {**776, 827**}

[VCV01b] Brigitte Verdonk, Annie Cuyt, and Dennis Verschaeren. A precision- and range-independent tool for testing floating-point arithmetic II: Conversions. *ACM Transactions on Mathematical Software*, 27(1):119–140, March 2001. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.win.ua.ac.be/~cant/ieeecc754.html`. DOI `10.1145/382043.382405`. {**776, 827**}

[Vig16] Sebastiano Vigna. An experimental exploration of Marsaglia's `xorshift` generators, scrambled. *ACM Transactions on Mathematical Software*, 42(4):30:1–30:23, July 2016. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://dl.acm.org/citation.cfm?id=2845077`. DOI `10.1145/2845077`. {**1001, 1024, 1028**}

[vN51] John von Neumann. 13. Various techniques used in connection with random digits. In Alston S. Householder, George E. Forsythe, and Hallett-Hunt Germond, editors, *Monte Carlo method. Proceedings of a Symposium Held June 29, 30 and July 1, 1949 in Los Angeles, California*, volume 12 of *Applied Mathematics Series / National Bureau of Standards*, pages 36–38. United States Government Printing Office, Washington, DC, USA, 1951. URL `http://dornsifecms.usc.edu/assets/sites/520/docs/VonNeumann-ams12p36-38.pdf`. Summary written by G. E. Forsythe. Reprinted in [Tau63, Paper 23, pp. 768–770]. {**190**}

[VS04] P. W. J. Van Eetvelt and S. J. Shepherd. Accurate, computable approximations to the error function. *Mathematics Today*, 40(1):25–27, February 2004. ISSN 1361-2042. {**600, 606**}

[Wal96] P. L. Walker. *Elliptic Functions: a Constructive Approach*. Wiley, New York, NY, USA, 1996. ISBN 0-471-96531-6; 978-0-471-96531-2. xv + 214 pp. LCCN QA343 .W3 1996. {**619**}

[Wal00] H. S. Wall. *Analytic Theory of Continued Fractions*. American Mathematical Society, Providence, RI, USA, 2000. ISBN 0-8218-2106-7; 978-0-8218-2106-0. xiii + 433 pp. LCCN QA295 .W28 2000. This is a reprint of the definitive, and widely cited, treatise first published in 1948. {**19**}

[War03] Henry S. Warren. *Hacker's Delight*. Addison-Wesley, Reading, MA, USA, 2003. ISBN 0-201-91465-4; 978-0-201-91465-8. xiv + 306 pp. LCCN QA76.6 .W375 2003. URL `http://www.hackersdelight.org/`. While this book does not specifically address computational aspects of floating-point arithmetic (apart from the nine-page Chapter 15), it has extensive coverage of, and clever algorithms for, integer arithmetic operations that are fundamental for implementing hardware floating-arithmetic and software multiple-precision arithmetic. The book's Web site contains supplementary material in preparation for the second edition [War13]. {**166, 176, 978**}

[War13] Henry S. Warren. *Hacker's Delight*. Addison-Wesley, Reading, MA, USA, second edition, 2013. ISBN 0-321-84268-5 (hardcover); 978-0-321-84268-8 (hardcover). xvi + 494 pp. LCCN QA76.6 .W375 2013. URL `http://www.pearsonhighered.com/educator/product/Hackers-Delight/9780321842688.page`. {**166, 176, 978, 1036**}

[Wat95] G. N. Watson. *A Treatise on the Theory of Bessel Functions*. Cambridge mathematical library. Cambridge University Press, Cambridge, UK, second edition, 1995. ISBN 0-521-48391-3 (paperback), 0-521-06743-X (hardcover); 978-0-521-48391-9 (paperback), 978-0-521-06743-0 (hardcover). vi + 804 pp. LCCN QA408 .W2 1995. {**693**}

[WE12] Dong Wang and Miloš D. Ercegovac. A radix-16 combined complex division/square root unit with operand prescaling. *IEEE Transactions on Computers*, 61(9):1243–1255, September 2012. CODEN ITCOB4. ISSN 0018-9340 (print), 1557-9956 (electronic). DOI `10.1109/TC.2011.143`. {**463**}

[Web08] Charles F. Webb. IBM z10: The next-generation mainframe microprocessor. *IEEE Micro*, 28(2):19–29, March/April 2008. CODEN IEMIDZ. ISSN 0272-1732 (print), 1937-4143 (electronic). DOI `10.1109/MM.2008.26`. {**927**}

[Wei99] Eric W. Weisstein. *The CRC Concise Encyclopedia of Mathematics*. CRC Press, Boca Raton, FL, USA, 1999. ISBN 0-8493-9640-9; 978-0-8493-9640-3. 1969 pp. LCCN QA5.W45 1999. URL `http://mathworld.wolfram.com/`. {**619**}

[Wei09] Eric W. Weisstein. *CRC Encyclopedia of Mathematics*. CRC Press/Taylor and Francis, Boca Raton, FL, third edition, 2009. ISBN 1-4200-7221-8; 978-1-4200-7221-1. 4307 pp. LCCN QA5 .W45 2009. Three hardcover volumes. {**58, 569, 572, 576, 579, 587**}

[WET$^+$10] Liang-Kai Wang, Mark A. Erle, Charles Tsen, Eric M. Schwarz, and Michael J. Schulte. A survey of hardware designs for decimal arithmetic. *IBM Journal of Research and Development*, 54(2):8:1–8:15, 2010. CODEN IBMJAE. ISSN 0018-8646 (print), 2151-8556 (electronic). URL `http://www.research.ibm.com/journal/abstracts/rd/542/wang-schwarz.html`. DOI `10.1147/JRD.2010.2040930`. {**927**}

[WF82] Shlomo Waser and Michael J. Flynn. *Introduction to Arithmetic for Digital Systems Designers*. Holt, Reinhart, and Winston, New York, NY, USA, 1982. ISBN 0-03-060571-7; 978-0-03-060571-0. xvii + 308 pp. LCCN TK7895 A65 W37 1982. This book went to press while the IEEE 754 Floating-Point Standard was still in development; consequently, some of the material on that system was invalidated by the final Standard (1985) [IEEE85a]. {**1016**}

[WG89] Z. X. Wang and D. R. Guo. *Special Functions*. World Scientific Publishing, Singapore, 1989. ISBN 9971-5-0659-9; 978-9971-5-0659-9. xiii + 422 pp. LCCN QA331 .W296 1989. {**521, 827**}

[WH82] Brian A. Wichmann and Ian David Hill. Statistical algorithms: Algorithm AS 183: An efficient and portable pseudo-random number generator. *Applied Statistics*, 31(2):188–190, June 1982. CODEN APSTAG. ISSN 0035-9254 (print), 1467-9876 (electronic). URL `http://lib.stat.cmu.edu/apstat/183`. DOI `10.2307/2347988`. See correction [WH84] and remarks [McL85, Zei86]. Reprinted in [GH85, pages 238–242]. See also the extended 32-bit generator in [WH06]. {**177, 1024, 1037**}

[WH84] Brian A. Wichmann and Ian David Hill. Statistical algorithms: Correction: Algorithm AS 183: An efficient and portable pseudo-random number generator. *Applied Statistics*, 33(1):123, 1984. CODEN APSTAG. ISSN 0035-9254 (print), 1467-9876 (electronic). URL `http://www.jstor.org/stable/2347676`. DOI `10.2307/2347676`. {**1037**}

[WH06] Brian A. Wichmann and Ian David Hill. Generating good pseudo-random numbers. *Computational Statistics & Data Analysis*, 51(3):1614–1622, December 1, 2006. CODEN CSDADW. ISSN 0167-9473 (print), 1872-7352 (electronic). URL `http://www.sciencedirect.com/science/article/pii/S0167947306001836`. DOI `10.1016/j.csda.2006.05.019`. This work extends a widely used generator [WH82] developed for 16-bit arithmetic to a new four-part combination generator for 32-bit arithmetic with a period of $2^{121} \approx 10^{36}$. {**177, 1037**}

[Wic88] Michael J. Wichura. Statistical algorithms: Algorithm AS 241: The percentage points of the normal distribution. *Applied Statistics*, 37(3):477–484, September 1988. CODEN APSTAG. ISSN 0035-9254 (print), 1467-9876 (electronic). URL `http://lib.stat.cmu.edu/apstat/241`. DOI `10.2307/2347330`. {**600**}

[Wic92] Brian A. Wichmann. Surveyor's Forum: "What every computer scientist should know about floating-point arithmetic". *ACM Computing Surveys*, 24(3):319, September 1992. CODEN CMSVAN. ISSN 0360-0300 (print), 1557-7341 (electronic). See [Gol91a, Gol91b, Dun92]. {**1008, 1013**}

[Wie99] Thomas Wieder. Algorithm 794: Numerical Hankel transform by the Fortran program HANKEL. *ACM Transactions on Mathematical Software*, 25(2):240–250, June 1999. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.netlib.org/toms/794`. DOI `10.1145/317275.317284`. {**693**}

[Wil02] Robin J. Wilson. *Four Colors Suffice: How the Map Problem Was Solved*. Princeton University Press, Princeton, NJ, USA, 2002. ISBN 0-691-11533-8; 978-0-691-11533-7. xii + 262 pp. LCCN QA612.19 .W56 2002. {**60**}

[Wir71a] Niklaus Wirth. The design of a PASCAL compiler. *Software — Practice and Experience*, 1(4):309–333, October/December 1971. CODEN SPEXBL. ISSN 0038-0644 (print), 1097-024X (electronic). DOI `10.1002/spe.4380010403`. {**949**}

[Wir71b] Niklaus Wirth. The programming language Pascal. *Acta Informatica*, 1(1):35–63, January 1971. CODEN AINFA2. ISSN 0001-5903 (print), 1432-0525 (electronic). URL `http://link.springer.com/article/10.1007/BF00264291`. DOI `10.1007/BF00264291`. {**949**}

[Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Upper Saddle River, NJ, USA, 1976. ISBN 0-13-022418-9; 978-0-13-022418-7. xvii + 366 pp. LCCN QA76.6 .W561. {**3**}

[WN95] David J. Wheeler and Roger M. Needham. TEA, a tiny encryption algorithm. *Lecture Notes in Computer Science*, 1008:363–366, 1995. CODEN LNCSD9. ISSN 0302-9743 (print), 1611-3349 (electronic). URL `http://www.springerlink.com/content/p16916lx735m2562/`. DOI `10.1007/3-540-60590-8_29`. {**178, 1026, 1037**}

[WN98] David J. Wheeler and Roger M. Needham. Correction to XTEA. Report, Cambridge University, Cambridge, UK, October 1998. URL `http://www.movable-type.co.uk/scripts/xxtea.pdf`. See also original TEA [WN95] and first extension XTEA [NW97]. {**178, 1026**}

[WSH77] J. Welsh, W. J. Sneeringer, and C. A. R. Hoare. Ambiguities and insecurities in Pascal. *Software — Practice and Experience*, 7(6):685–696, November/December 1977. CODEN SPEXBL. ISSN 0038-0644 (print), 1097-024X (electronic). DOI `10.1002/spe.4380070604`. See also [Ker81, Ker84]. {**989, 1018**}

[Wu97] Pei-Chi Wu. Multiplicative, congruential random-number generators with multiplier $\pm 2^{k_1} \pm 2^{k_2}$ and modulus $2^p - 1$. *ACM Transactions on Mathematical Software*, 23(2):255–265, June 1997. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.acm.org/pubs/citations/journals/toms/1997-23-2/p255-wu/`. DOI `10.1145/264029.264056`. {**170**}

[YM98] Robyn V. Young and Zoran Minderović, editors. *Notable Mathematicians: From Ancient Times to the Present*. Gale, Detroit, MI, USA, 1998. ISBN 0-7876-3071-3; 978-0-7876-3071-3. xxi + 612 pp. LCCN QA28 .N66 1998. {**59**}

[Ypm95] Tjalling J. Ypma. Historical development of the Newton–Raphson method. *SIAM Review*, 37(4):531–551, December 1995. CODEN SIREAD. ISSN 0036-1445 (print), 1095-7200 (electronic). URL `http://epubs.siam.org/23425.htm`; `http://link.aip.org/link/?SIR/37/531/1`. DOI `10.1137/1037125`. {**8**}

[ZC70] D. G. Zill and Bille Chandler Carlson. Symmetric elliptic integrals of the third kind. *Mathematics of Computation*, 24(109):199–214, January 1970. CODEN MCMPAF. ISSN 0025-5718 (print), 1088-6842 (electronic). URL `http://www.jstor.org/stable/2004890`. DOI `10.2307/2004890`. {**646, 651**}

[Zei86] H. Zeisel. Statistical algorithms: Remark ASR 61: a remark on Algorithm AS 183. an efficient and portable pseudo-random number generator. *Applied Statistics*, 35(1):89, 1986. CODEN APSTAG. ISSN 0035-9254 (print), 1467-9876 (electronic). URL `http://www.jstor.org/stable/2347876`. See [WH82, McL85]. {**1024, 1037**}

[ZH10] Yong-Kang Zhu and Wayne B. Hayes. Algorithm 908: Online exact summation of floating-point streams. *ACM Transactions on Mathematical Software*, 37(3):37:1–37:13, 2010. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). DOI `10.1145/1824801.1824815`. {**385**}

[Zim06] Paul Zimmermann. Worst cases for sin(BIG). World-Wide Web slides, November 2, 2006. URL `http://www.loria.fr/~zimmerma/talks/sinbig.pdf`. {**28**}

[Ziv91] Abraham Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, September 1991. CODEN ACMSCU. ISSN 0098-3500 (print), 1557-7295 (electronic). URL `http://www.acm.org/pubs/citations/journals/toms/1991-17-3/p410-ziv/`. DOI `10.1145/114697.116813`. {**827**}

[ZJ96] Shanjie Zhang and Jianming Jin. *Computation of Special Functions*. Wiley, New York, NY, USA, 1996. ISBN 0-471-11963-6; 978-0-471-11963-0. xxvi + 717 pp. LCCN QA351.C45 1996. {**521, 556, 567, 593, 644, 657, 693, 827**}

[ZM08] Stephen Thomas Ziliak and Deirdre N. McCloskey. *The Cult of Statistical Significance: How the Standard Error Costs Us Jobs, Justice, and Lives*. Economics, cognition, and society. University of Michigan Press, Ann Arbor, MI, USA, 2008. ISBN 0-472-07007-X (cloth), 0-472-05007-9 (paperback); 978-0-472-07007-7 (cloth), 978-0-472-05007-9 (paperback). xxiii + 321 pp. LCCN HB137 .Z55 2008. This book about the use and abuse of statistics, and the historical background of the *t*-test, is recommended for all producers and consumers of statistical analyses. {**196**}

[ZOHR01] Abraham Ziv, Moshe Olshansky, Ealan Henis, and Anna Reitman. Accurate portable mathematical library (IBM APMathLib). World-Wide Web document, December 20, 2001. URL `ftp://www-126.ibm.com/pub/mathlib/mathlib12.20.2001.tar.gz`; `http://oss.software.ibm.com/mathlib/`. {**827**}

[ZRP05] Paul Zimmermann, Nathalie Revol, and Patrick Pélissier. `mpcheck`: a program to test the accuracy of elementary functions. World-Wide Web software archive, 2005. URL `http://www.loria.fr/~zimmerma/free/mpcheck-1.1.0.tar.gz`; `http://www.loria.fr/~zimmerma/mpcheck/`. {**825**}

[Zwi92] Daniel Zwillinger. *Handbook of Integration*. Jones and Bartlett, Boston, MA, USA, 1992. ISBN 0-86720-293-9; 978-0-86720-293-9. xv + 367 pp. LCCN QA299.3 .Z85 1992. {**58, 560**}

[Zwi03] Daniel Zwillinger, editor. *CRC Standard Mathematical Tables and Formulae*. Chapman and Hall/CRC, Boca Raton, FL, USA, 31st edition, 2003. ISBN 1-58488-291-3, 1-4200-3534-7 (electronic); 978-1-58488-291-6, 978-1-4200-3534-6 (electronic). xiv + 910 pp. LCCN QA47 .M315 2003. {**58**}

[Zwi12] Daniel Zwillinger, editor. *CRC Standard Mathematical Tables and Formulae*. CRC Press, Boca Raton, FL, USA, 32nd edition, 2012. ISBN 1-4398-3548-9; 978-1-4398-3548-7. xiii + 819 pp. LCCN QA47 C73 2012; QA47 .M315 2012. URL `http://www.crcpress.com/CRC-Standard-Mathematical-Tables-and-Formulae-32nd-Edition/Zwillinger/p/book/9781439835487`. {**58**}

Author/editor index

If authors did not write,
what would editors do?
— Anonymous.

Primary authors are shown in Small Caps, and secondary authors in roman. Page numbers of citations of primary authors are in **bold**.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2

Function and macro index

FUNCTION: A MATHEMATICAL QUANTITY WHOSE CHANGES OF VALUE
DEPEND ON THOSE OF OTHER QUANTITIES CALLED ITS VARIABLES.

— *New Century Dictionary* (1914).

MACRO: A NAME (POSSIBLY FOLLOWED BY A FORMAL *arg* LIST)
THAT IS EQUATED TO A TEXT OR SYMBOLIC EXPRESSION TO WHICH IT IS TO BE EXPANDED
(POSSIBLY WITH THE SUBSTITUTION OF ACTUAL ARGUMENTS) BY A MACRO EXPANDER.

— *The New Hacker's Dictionary* (1996).

Page numbers of defining and primary entries are shown in **bold**. The entries for *algorithm*, *function*, *macro*, and *mathematical function* each contain a lengthy list of subentries. Uppercase macro names followed by empty parentheses are often wrappers for integer, binary floating-point, and decimal floating-point functions of all supported lengths. Greek letters are indexed by their English names.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2

Subject index

> IN THE BAD OLD DAYS, THE INDEX WAS A LIST OF PROHIBITED BOOKS; MAY WE NOW, IN A MORE ENLIGHTENED AGE, BAN BOOKS WITHOUT INDEXES?
>
> — STEPHEN JAY GOULD
> *An Urchin in the Storm* (1987).

Page numbers of biographical, defining, and primary entries are in **bold**. Publication titles explicitly cited in the text are in *italic*. Names of people are indexed if they are explicitly mentioned prior to the bibliography. Personal names in the bibliography are recorded in the separate author/editor index on page 1039.

Index entries for words inside verbatim code blocks are generated at end-of-block. Thus, blocks that cross page boundaries may produce off-by-one page numbers in this index.

The *Greek letter* entry has subentries for each such letter used in this book, but ordered by their names in the Latin, rather than Greek, alphabet.

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2

Colophon

This book was typeset with $\LaTeX\,2_{\varepsilon}$ using the extended extbook document class, with limited customization of the layout. The book pages use Palatino text fonts, Palatino Italic and Computer Modern mathematical fonts, and DejaVuSansMono typewriter fonts, with 9.7pt type on 11.64pt leading.

The typographic features exploited in the design of this book were provided by several standard $\LaTeX\,2_{\varepsilon}$ packages, including at least these:

amsfonts
amsmath
array
calligra
color
colortbl
fontenc
graphicx
luximono
makeidx
mathpazo
multicol
pifont
url
varioref

These private packages supplied the remaining $\LaTeX\,2_{\varepsilon}$ extensions:

authidx
hhmm
mathcw
mod-mathpazo
namelist
resize-mathcw
rgb
widecenter

Document and software development and production were managed on UNIX-like systems on a score of vendor/hardware platforms, on the `SIMH` VAX and `Hercules` System/370, ESA/390, and z/Architecture simulators, on DEC TOPS-20 on the KLH10 PDP-10 simulator, and more than 100 operating systems running on `QEMU` and VMware virtual machines for IA-32 and AMD64 processor families, with the help of at least these external programs:

```
acroread
authidx
bc
bibtex
build-all
chkdelim
chktex
col
deroff
detex
distill
dv2dt
dvips
dw
egrep
emacs
gnuplot
gp
groff
gs
gv
gzip
hoc32
hoc36
hoc64
hoc72
hoc80
hoc128
hocd32
hocd64
hocd128
ispell
lacheck
latex
makeindex
makeinfo
make
man2html
man2texi
maple
math
matlab
maxima
mupad
myspell
nawk
pdflatex
pr
ps2pdf
reduce
R
sage
sed
sh
sort
spell
Splus
xdvi
```

N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2