Robert M. Corless
Nicolas Fillion

A Graduate Introduction to Numerical Methods

From the Viewpoint of Backward Error Analysis

Springer

A Graduate Introduction to Numerical Methods

Robert M. Corless • Nicolas Fillion

A Graduate Introduction to Numerical Methods

From the Viewpoint of Backward Error Analysis

Springer

Robert M. Corless
Applied Mathematics
University of Western Ontario
London, ON, Canada

Nicolas Fillion
Applied Mathematics
University of Western Ontario
London, ON, Canada

ISBN 978-1-4614-8452-3 ISBN 978-1-4614-8453-0 (eBook)
DOI 10.1007/978-1-4614-8453-0
Springer New York Heidelberg Dordrecht London

Library of Congress Control Number: 2013955042

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

“At its highest level, numerical analysis is a mixture of science, art, and bar-room brawl.”

T. W. Körner
The Pleasures of Counting, CUP, 1996, p. 505.

Dedicated to
our mentors

Foreword

It is a great privilege to be able to write these few words of introduction to this fine book. Computational mathematics is now recognised as a central tool in all aspects of applied mathematics. Scientific modelling falls well short of the mark if it attempts to describe problems and predict outcomes, without numerical computations. Thus, an understanding and appreciation of numerical methods are vital components in the training of scientists and engineers.

What are numerical methods? Clearly, they are methods for obtaining numerical results. But what numerical results are we looking for? This depends on whom you ask, but a general point of view is to look for common ideas and systematic structures. Thus, linear algebra is central to much of numerical analysis because many scientific problems we need to solve are nothing more than linear equation solutions and eigenvalue calculations. But more than this, many other problem types are capable of being expressed in linear algebra terms, and other calculations require efficient linear algebra computations within their core. Many years ago I was told that it had been estimated that if a random computer was stopped at a random time, there would be more than even chances that it would be caught in the middle of an LU factorization. Even if this were true once, it might no longer be true, but it is no more than an exaggeration of the undoubtedly true statement that computational linear algebra is very important and fundamental to science.

Numerical linear algebra occupies Part II of this four-part book and covers familiar topics as well as many topics that deserve to be familiar. If all the reader wants are the algorithms, then these are there, but the authors are scholars and the reader is not let off so easily. You are dragged gently but firmly to a higher world in which the algorithms are presented in the context of a deductive science. You learn judgment and understanding, and you benefit from the authors' combined experience and knowledge.

But if this is Part II, what of Part I? Even more fundamental issues are needed before linear algebra can be properly presented, such as the fundamental ideas of computer arithmetic, and the limitations of practical computation in a finite-word computer. Questions about the roots of equations, about the evaluation of series and about partial fractions are presented in the entertaining, but at the same time informative, style that characterizes the work as a whole.

If the key ideas in Parts I and II are algebraic, the last two parts are calculus-based. In terms of complexity, the first half of the book deals mainly with problems whose solutions in principle are exact, but the second half is about problems for which there is an intrinsic approximation in what is being evaluated. Central to Part III is interpolation, where $f(x)$ is estimated from values of $f(x_i)$ based on a set $\{x_1, x_2, \ldots, x_n\}$, with an error usually expressed in terms of the behavior of $f^{(n)}$. The four chapters that comprise this part represent areas in which the authors have made many of their own original contributions. These chapters represent a high point of this very high book.

Part IV deals with differential equations and related problems. There are detailed studies of both initial value and boundary value ordinary differential equation problems. Finally, there is a chapter each on delay differential equations and on various types of partial differential equations.

The book is rounded out with three useful appendix chapters, presented at the end of this book.

I love this book.

Auckland, New Zealand John Butcher

Preface

About This Book

This book is designed to be used by mathematicians, engineers, and computer scientists as a graduate-level introduction to numerical analysis and its methods. Readers are expected to have had courses or experience in calculus, linear algebra, complex variables, differential equations, and programming. Of course, many students will be missing some of that material, and we encourage generalized review, especially of linear algebra.

The book is intended to be suitable both for one-semester and for two-semester courses. It gathers important and recent material from floating-point arithmetic, numerical linear algebra, polynomials, interpolation, numerical differentiation and integration, and numerical solutions of differential equations. Our guiding principle for the selection of material and the choice of perspective is that numerical methods should be discussed as a part of a more general practice of mathematical modeling as is found in applied mathematics and engineering. Once mostly absent from texts on numerical methods, this *desideratum* has become an integral part of much of the active research in various fields of numerical analysis (see, e.g., Enright 2006a). However, because the intended audience is so broad that we cannot really presume a common background in application material, while we focus on applicable computational mathematics, we will not present many actual applications. We believe that the best-compromise approach is to use a perspective on the quality of numerical solution known as *backward error analysis*, together with the theory of *conditioning* or *sensitivity of a problem*, already known to Turing and widely practiced and written on by J. H. Wilkinson, W. Kahan, and others.[1] These ideas, very important although not a panacea, will be introduced progressively. The basic underpinning of the backward error idea, that a numerical method's errors should be analyzable in

[1] The first explicit use of backward error analysis is credited by Wilkinson (1971) to Wallace Givens, and indeed, it is already present in Von Neumann and Goldstine (1947) (see also Grcar 2011), but it is broadly agreed that it was Wilkinson himself who began the systematic exploitation of the idea in a broad collection of contexts.

the same terms as whatever physical (or chemical or biological or social or what-have-you) modeling errors, is readily understandable across all fields of application. As Wilkinson (1971 p. 554) pointed out, backward error analysis

> has the advantage that rounding errors are put on the same footing as errors in the original data and the effect of these has usually to be considered in any case.

The notion of the sensitivity of the problem to changes in its data is also one that is easy to get across to application-oriented students. As Chap. 1 explains, this means that we favor a residual-based a posteriori type of backward error analysis that provides numerical solutions that are readily interpretable in the broader context of mathematical modeling.

The pedagogical problem that (we hope!) justifies the existence of this book is that even though many excellent numerical analysis books exist, no single one of them that we know of is suitable for such a broad introductory graduate course—at least, not one that provides a unifying perspective based on the concept of backward error analysis, which we think is the most valuable aspect of this present book. Some older books *do* hold this perspective, most notably Henrici (1982), but that book is dated in other respects nowadays.

Other differences between this book and the general numerical analysis literature is that it uses the Lagrange and Hermite interpolational bases heavily, with a complex-variable focus, both because of the recent recognition of the superiority of this approach, and in order to introduce topics in an example-based format. Our objective is to provide the reader with a perspective on scientific computing that provides a systematic method for thinking about numerical solutions and about their interpretation and assessment.

The closest existing texts to our book in this outlook might be Quarteroni et al. (2007), or perhaps the pair of books Deuflhard and Bornemann (2002) and Deuflhard and Hohmann (2003), but our book differs from those in several other respects. We believe, for one, that our relatively informal treatment is less demanding on the mathematical and analytical prerequisites of the students; our students in particular have a very wide range of backgrounds. The topics we cover are also slightly different from those in the aforementioned books—for example, we cover delay differential equations and they do not, whereas their coverage of the numerical solution of PDEs is more complete than ours. But for us, the most important thing about a graduate-level introduction is to show the essential unity of the subject, and we feel that aim of this present work is worth pursuing.

Thus, our objective is to present a unified view of numerical computation, insofar as that is possible. The book cannot, therefore, be self-contained, or anything like complete; it can only hit some of the highlights and point to more extensive discussions of specific points. This is, unfortunately, a necessary tradeoff for such a book, and in partial compensation the list of references is substantial. Consequently, the book is not a “standard” numerical analysis text, in several respects. The topic selection is intended to introduce the reader to important components of a graduate students’ toolbox, but more on the *analysis* side than the *methods* side. It is not intended to be a book of recipes.

This brings up the "elephant in the room," the massively popular and useful book (Press et al. 1986), which has been cited more than 33,000 times according to Google Scholar, as we write this. That book is intended for "quick use," we believe. If you have a numerical problem to solve, and only a weekend to do something about it, that book should probably be your first choice of reference. One thing we certainly do not criticize that book for is its attempt at comprehensive coverage. However, it is not a textbook and does not serve the purpose of a course in numerical analysis, which we believe includes a unified theoretical view of all numerical methods. Hence, this present book attempts a complementary view to that of Press et al. (1986).

Finally, even though a unified view is attempted here, many important topics in numerical analysis had to be left out altogether. This includes optimization, integral equations, parallel and high-performance computing, among others. We regret that,[2] but we make no promises to remedy this deficit any time soon. Instead, it is our hope that the reader of this book will have acquired a framework for assessing and understanding numerical methods generally.

Another difference in perspective of this book is the following. As the reader might know (or will know very soon!), there tends to be a tension between computation time, on the one hand, and accuracy and reliability, on the other hand. There are two points of view in scientific computing nowadays, which are paraphrased below:

1. I don't care how correct your answer is if it takes 100 years to get it.
2. I don't care how quickly you give me the wrong answer.

Of these two blunders, we tend to think the first is worse: Hence, this book concentrates on reliability. Therefore, we will not focus on cost very much, nor will we discuss vectorization of algorithms and related issues.

There are more schemes for computation than just IEEE standard fixed-precision floating-point arithmetic, which is the main tool used in this book (and, without much doubt, the main tool used in scientific and engineering computing). There is also *arbitrary*-precision floating-point arithmetic, which is used in computer algebra systems such as MAPLE. This is comparatively slow but occasionally of great interest; some examples will be given in this book. There is also *interval* arithmetic, which is discussed concisely, with references, on the Wikipedia page of the same name: The principle of interval arithmetic is to compute not just answers, but also bounds for the errors in the answers. Again, this is slower than standard fixed-precision floating-point arithmetic, but not solely for the reason that more computation is done with the bounds, but also for the somewhat surprising reason that for many algorithms of practical interest as implemented in floating-point, the *rounding errors usually cancel,* leaving an accurate answer but with overly wide error bounds in interval arithmetic. As a consequence, other algorithms (usually iterative) have to be developed specifically for use with intervals, and while this has been done, particularly for many problems in optimization, and is valuable especially in cases where

[2] In particular, we regret not covering the finite-element method; or multigrid; or ...; you get the idea.

the consequences of rounding errors are disastrously expensive, interval arithmetic is not as widely used as floating-point arithmetic is.

A prominent computer algebra researcher asks, “Why not compute the answer exactly?” This researcher knows full well that in the vast majority of cases, exact computation is either impossible outright or impossibly expensive. However, for *some* problems, particularly some linear algebra problems, the data are indeed known exactly and the algorithms for computing the exact rational answer have now been developed to a high degree of efficiency, making it possible nowadays to really get the exact answer (what we will call the *reference* answer in this book). We do not discuss such algorithms here, in part because they are specialized, but really because this course is about numerical methods with approximate arithmetic.

There are yet other arithmetics that have been proposed: significance arithmetic (which is similar to interval arithmetic but less rigorous), and “rounded rational” arithmetic, and others. Some of these are discussed in Knuth (1981). A recent discussion of computational arithmetic can be found in Brent and Zimmermann (2011).

Finally, we underline the fact that the background theoretical ideas from analysis and algebra used in this book are explained in a rather informal way, focusing more on helping visualization and intuition than precise theoretical understanding. This is justified by the fact that if the reader knows the material already, then it serves as a good refresher and also introduces the perspective on it that is relevant to the matter at hand. If the reader does not know the necessary material, or does not know it well, then it should provide just enough guidance to have a feel of what is going on, while at the same time give precise indications as to what and where to look to acquire the required concepts. Just pointing at a book wouldn’t do if we can’t say what to look for. In this way, we expect to be able to reach the vastly different kinds of reader who need the course this book was designed to support.

On Programming

Computations in the book are carried out almost exclusively in MATLAB (but we also use MAPLE on some occasions). Readers not familiar with MATLAB are encouraged to acquire the wonderful book Higham and Higham (2005) to help them to learn MATLAB. We have no commercial commitment to MATLAB, and if the reader wishes to use SCILAB or OCTAVE instead, then other than some of the advanced techniques available in MATLAB but not in those two for the numerical solution of sparse matrices or ordinary differential equations, the substitution might be all right (but we have not tried). Similarly, the reader may wish to use SAGE or some other freely available computer algebra package to help get through the more formulaic aspects of this book.

This book is *not* a book that teaches programming skills, even by example (our programs are all short and intended to illustrate one or two numerical techniques only). The programs in this book are not even intended as good examples of programming style, although we hope they meet minimal goals in that respect, with an emphasis on readability over efficiency. The elegant little book Johnson (2010) is a useful guide to a consistent Matlab style. The style of the programs in this present book differs slightly from that advocated there, in part because our aesthetic tastes differ slightly and in part because the purpose of numerical computing, being more limited than computing that includes, for example, data management, can bear a simpler style without loss of readability or maintainability. However, we emphatically agree with Johnson that a consistent style is a great help, both to the readers of the code (which might include the writer, three months later) and to the users of the code. We also agree that attention to stylistic issues while writing code is a great help in minimizing the number and severity of bugs.

In this book, Matlab commands will be typeset in the `lstlisting` style and are intended to be typed as shown (with the exception of the line numbers to the left, when any, which are added for pedagogical purposes). For example, the commands

```
x = linspace( -1, 1, 21 );
y = sin( pi*x );
plot( x, y, 'k--' )
```

produce a black dashed-line plot of $\sin(\pi x)$ on the interval $-1 \le x \le 1$. One difference to the style advocated in Johnson (2010) is that spaces are introduced after each opening parenthesis and before each closing parenthesis; similarly, spaces are used after commas. These spaces have no significance to Matlab but they significantly improve readability for humans (especially in the small font in which this book is typeset). The programs written for this book are all intended to be made available over the web, so longer bits of code need not be typed. The code repository can be accessed at `http://www.nfillion.com/coderepository`. Similarly, Maple commands will also be typeset in the `lstlisting` style; since the syntaxes for the two languages are similar but *not* identical, this has a risk of causing confusion, for which we apologize in advance. However, there are not that many pieces of Maple code in the book, and each of them is marked in the text surrounding it, so any confusion will not last long. For example, a similar plot to that created above can be done in Maple by the single command

```
plot( sin( Pi*x ), x=-1..1, linestyle=3, color=BLACK );
```

Moreover, we *request* the reader to minimize the use of `sym` in Matlab. If you are going to do symbolic computation, fire up a computer algebra system (Maple, Sage, MuPAD, whatever you like) and use it and its features separately. Yes, the Symbolic Toolbox (which uses MuPAD or Maple), if you have it, can be helpful and professionals often do use it for small symbolic computations. In a numerical course, however, `sym` can be very confusing and requires more care in handling than we want to discuss here. This book will not use it at all, and the problems and exercises have been designed so that you need not use it. If you *do* choose to use it, do so on your own recognizance.

Scientific programming is, in our view, seriously underrated as a discipline and given nowhere near the attention in the curriculum that it deserves or needs. Many people view the course that this book is intended to support, namely, an introductory course in numerical analysis for graduate students, as “the” course that a graduate student takes in order to learn how to program. *This is a serious mistake.* If this is the only course that you take that has programming in it, you are in trouble. It takes more than a few weekends to learn how to program (and given the amount of material here, you won't have many weekends available, even).

However, you can make a *start* on programming at the same time as you read this book if you are willing to really put in some effort. Both MATLAB and MAPLE are easier to learn than many scientific programming languages, at least for people with a high level of mathematical maturity and background knowledge. You will need substantial guidance, though, in addition to this book. The aforementioned book Higham and Higham (2005) is highly recommended. The older book Corless (2002), while dated in some respects, was intended to teach MAPLE to numerical analysts, and since the *programming language* for MAPLE has not changed much since then (although the GUI has), it remains potentially useful. Our colleague Dhavide Aruliah also recommends the Software Carpentry project by Greg Wilson `http://software-carpentry.org/`, which we were delighted to learn about—there seems to be a wealth of useful material there, including a section on MATLAB. See also Aruliah et al. (2012).

Large scientific programs require a serious level of discipline and mathematical thought; this is the discipline nowadays called software engineering. This book does not teach software engineering at all. For those wishing to have a glimpse, we highly recommend the (ancient, in computer terms) books by Leo J. Brodie, which use the curiously lovely computer language Forth.[3] In some sense, Forth is natural to teach those concepts: It is possible to write arbitrarily unreadable code in Forth entirely by accident, and you *need* to learn some discipline to write it well; it's harder to write unreadable code in MAPLE (although for sure it can be done!).

Writing software that is robust, readable, maintainable, usable, and efficient, and overall does what it was intended to do is a humbling activity. The first thing that one learns is—a true scientific lesson—that one's thought processes are not as reliable as one had believed. Numerical analysis and scientific computing (along with computer programming generally) have overturned many things that were thought mathematically to be true, and computer programs have had a profound influence on how we view the world and how we think about it. Indeed, one of us has coined the term "computer-mediated thinking" to cover some aspects of that profound change (see Corless 2004, for a discussion of this in a pedagogical context). Put simply, there is *no other way* to think about some complex systems than to combine the power of the mind with the power of the computer. We will see an example due to Turing, shortly.

[3] The book *Starting Forth* is now available free online at `http://www.forth.com/starting-forth/`, and although Forth has very little in common with MATLAB or MAPLE, the programming concepts and discipline begun in that book will transfer easily. The second book, *Thinking Forth*, is also available online at `http://thinking-forth.sourceforge.net/` and is one of the most useful introductions to software engineering, even though it, like its predecessor, is focused on Forth.

How to Use This Book

We believe, with Trefethen (2008b p. 606), that

> the main business of numerical analysis is designing algorithms that converge quickly; rounding-error analysis, while often a part of the discussion, is rarely the central issue.

Why, then, are the first chapter and the first appendix of this book so heavy on floating-point arithmetic? The answer is that the material is logically first, not that it is of the first importance didactically. In fact, when RMC teaches this course, he begins with Chap. 4 and looks back on the logically prior material when needed: His approach is “leap ahead, back fill.” But the students’ needs may vary considerably, and there are those who decidedly prefer an abstract presentation first, filled in with examples later: For them, they may begin with Chap. 1 and proceed in the order of Fig. 1a.

The instructor should find that the book can be used in many ways. Following the linear order is an option, provided you have enough time (a one-semester course certainly isn’t enough time). With the time constraint in mind, Fig. 1a follows the same theoretical order, but it shows what should be considered optional. As stated before, at Western we start with Chap. 4. That way the course starts with the material on the QR and SVD factoring, culminating in a definition of condition number. This approach brings the student to immediately engage a problem that stimulated the development of numerical analysis in the first place. Then we come

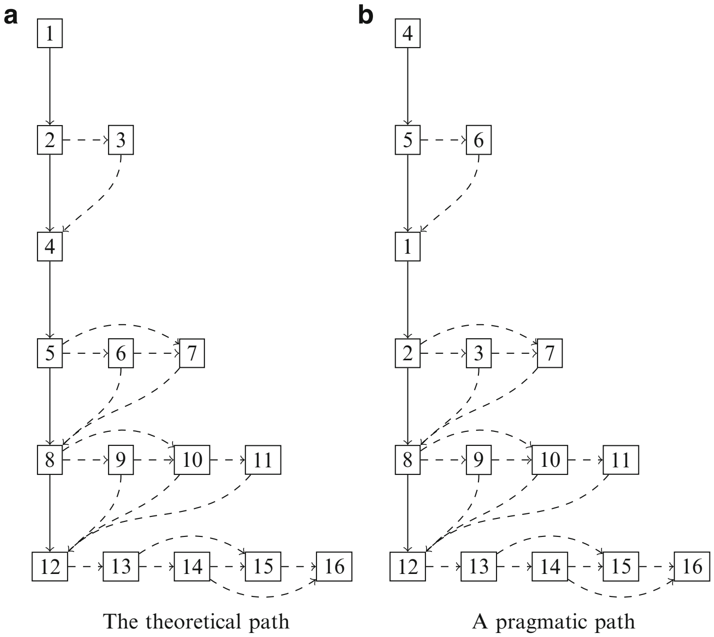


Fig. 1 Suggested teaching paths for this book, where *dashed lines* denote options. (**a**) The theoretical path. (**b**) A pragmatic path

back to Chap. 1 (and Appendix A) for a necessary examination of theoretical issues in finite-precision computation and approximation. We then return to Part II to discuss eigenvalue problems, sparse systems, and structured systems. We then proceed to polynomials, function evaluation, and then rootfinding. The course closes with material covering numerical integration and numerical solution of differential equations, followed by delay differential equations or partial differential equations, as the tastes of the students indicate and as time permits. A curriculum closely related to this pragmatic orientation is in Fig. 1b.

Experience has shown that the material in Chap. 8 is used heavily in almost all later chapters. Experience has also shown that the later chapters *always* get short-changed in a one-semester course: Probably at most one of Chaps. 14, 15, or 16 can be covered, and Chap. 9, though short and important, is in some danger of being omitted too. In any case, perhaps that is because RMC is personally focused more on Chap. 12 and its sequels, not because of the students' needs. In any case, the linear algebra topics can (and should!) always be covered.

Some of the chapters may be used for reading only. Good candidates are Chap. 1, Chap. 3 on the evaluation of functions, and Chap. 7 on iterative methods. Chapter 14, on delay DE, seems quite popular and goes quickly after the work on IVP and on interpolation.

Exercises

This book contains many exercises. They are identified as belonging to one of these categories:

1. Theory and Practice;
2. Investigations and Projects.

The first type of problem will include simple tasks that amount to "getting the go of it" or to make sure that one understands the basic notions that are assumed in the various manipulations. This includes practice with basic MATLAB and MAPLE tricks. It may also involve proofs—either from scratch, sometimes with hints, or completing proof sketches, including some error analyses (although not too many). The final type of problem—namely, investigations and projects—typically involves more time and effort from the students. Typically, these problems will involve exploring various numerical methods in depth, that is, doing analytic work and then implementing it, usually in MATLAB. That is, this type of problem is to some extent a programming assignment although the course we teach is not intended to teach programming skill.

The instructor may find it convenient to combine problems of different categories as well as different degrees of difficulty following this scheme. Students thus have the chance to feel the pleasant breeze of review, get their hands dirty with the tedious but very important practical work, and feel the ecstatic frustration of working on a problem of some *envergure*. For the more challenging projects, the student should refrain from being frustrated, remembering the words of J. S. Mill (1873 p. 45):

> A pupil from whom nothing is ever demanded which [s]he cannot do, never does all [s]he can.

One of the authors (NF), upon whom the teaching method used in this book was tested, has to agree that some of the difficult problems in this book are among those from which he learned most. We hope the reader will feel the same.

London, ON

Robert M. Corless
Nicolas Fillion

Acknowledgments

This project was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC). Many of the research papers that helped form the point of view of this book were also supported by NSERC and also by MITACS. Some of the writing was done while RMC was on sabbatical at Australian National University working with Trevor Lamb and with Markus Hegland, at the University of Newcastle Australia working with Jon Borwein, and finally at the University of Otago working with Gerrard Liddell. Many thanks to those people and institutions for their support.

Many more people have helped with this project. Perhaps most important are the students from the course Applied Math 9561, who encountered ("suffered" in the old sense is maybe the right word) the early versions of these chapters as lectures. As a whole, the Department of Applied Mathematics at the University of Western Ontario has been very supportive of this effort. The Ontario Research Centre for Computer Algebra and the Rotman Institute of Philosophy at UWO have, in particular, been very stimulating environments in which to work and think. The staff, including Audrey Kager, Pat Malone, and Karen Foullong at Applied Math, Elisa Kilbourne at Philosophy, and Rob Read at the Rotman Institute, have all been very helpful. The Rotman Institute itself deserves thanks for many kinds of support to both authors.

Acknowledgments from RMC:

My thanks also go to my best friend, David J. Jeffrey (who incidentally taught AM9561 before I did, and then assigned it to me when he became chair). Some of my UWO colleagues have now passed on, so they won't read these words, but I remember them and the lessons they taught me: M.A.H. (Paddy) Nerenberg, Peter Fraser, Vic Elias, Henning Rasmussen, and Karin Gatermann. My Ph.D. advisor, Geoffrey V. Parkinson of UBC, has also passed on, but again I remember. There's some of each of them in this book. I hope they would have liked it.

I have used several libraries in the course of working on this book. First and foremost is that at the University of Western Ontario, and I would like to say a special thank-you to my friend David J. Fiander, who has acted as an informal interface to many of the library's features; although he is indeed a librarian at UWO, the main

reason I talked to him instead of to the official Applied Math resource person as I was supposed to is that he's my friend. He has patiently endured my pestering and been very helpful. Thanks, David.

More generally, the other UWO library staff have themselves been helpful and the online version of the library has been available to me wherever I was in the world. This has been essential. While on sabbatical, I was granted full access to the libraries at Australian National University, the University of Newcastle, and the University of Otago, and these were also very helpful, but time and again I found what I wanted online through the off-campus access to the UWO library, which is as good as any I have seen. Thanks to all those library staff who made that possible.

Jim Varah and Uri Ascher first taught me numerical analysis at the University of British Columbia; R. Bruce Simpson and Keith Geddes continued the lessons at Waterloo. My mentors since then have included Christina Christara, Wayne Enright, Gaston Gonnet, Ken Jackson, Cleve Moler, Sanzheng Qiao, Bob Russell, Larry Shampine, Gustav Söderlind, Jim Verner, and to a certain extent William (Velvel) Kahan, who always had time to chat at conferences, and Donald E. Knuth. It's fair to say that I learned a lot from each of them. George F. Corliss, John C. Butcher (about whom more shortly), and David Stoutemyer were especially good friends for a young numerical analyst to make.

Peter Borwein gets a special thank-you, not least for his accidental invention of the "end-of-proof" symbol (♮) used in this book, pronounced "naturally."

My colleagues at the Ontario Research Center for Computer Algebra, including Dhavide Aruliah, Keith Geddes, Mark Giesbrecht, David Jeffrey, Hiroshi Kai, George Labahn, Marc Moreno Maza, Greg Reid, Éric Schost, Arne Storjohann, Lihong Zhi, and especially Ilias Kotsireas, have consistently provided a stimulating environment for research in scientific and symbolic computing.

I thank John C. Butcher, whose ANODE conferences remain among my fondest memories. Some of John's ideas have had a *huge* influence on this present book: All of the (many!) contour integrals used in this book are because of him. Larry Shampine took quite a thorough look at an early draft of this book, and his thoughtful critiques were very helpful indeed. While I am acknowledging Larry's help with this specific project, I would also like to point out that he is responsible in a *very* big way for much of the technology that makes the ideas discussed in this book possible and practical. It is all too easy to overlook such a contribution, because the codes he (and his students and co-workers) produced do their job so quietly and efficiently. Nobody notices when there aren't any problems and often fail to realize just how hard the problem was that is being solved with such apparent ease! Not only that, Larry was a pioneer in the use of backward error for the solution of differential equations and was an active contributor to much of its development. Both Nic and I are very grateful that he took the time to do all this.

Paul Sullivan, here at Western, taught me a very important lesson about backward error analysis, maybe the most important one: It's about data error, not rounding error. Of course, others had taught me that, too, but it was Paul's words—just a single sentence as I recall—that made the lesson sink in.

I have had many stimulating conversations with Nick Trefethen over the years, most recently at the MFUN13 conference in Manchester organized by Nick Higham and Françoise Tisseur; needless to say, I have learned an enormous amount from each of them, as well as from their beautiful books and papers.

There are other colleagues and friends not listed here who have also been helpful and more than helpful. Thank you all.

My past and present graduate students, including Anne-Marie E. Allison, Amir Amiraslani, Silvana Ilie, Piers W. Lawrence, Robert H. C. Moir, Xianping Liu, Azar Shakoori, Nargol Rezvani, Yiming Zhang, and Jichao Zhao, have together had a strong influence on this book.

Many thanks also go to Steven Thornton, who helped with the figures and reproducibility.

My final thanks go to my highly energetic co-author. Thank you, Nic—without you this book might not have begun, and certainly would not have been finished.

Acknowledgments from NF:

In addition to those already thanked by Rob, I would like to thank a number of exceptional people without whom I would never have had the opportunity to work on this book. To begin with, I want to thank my family, especially my parents, for their incessant support of my academic endeavors, however incomprehensible or silly they might have appeared. Merci du fond du cœur! Secondly, I must thank my first academic mentor, François Tournier, who turned me into a decent academic by instilling in me the virtues of hard but effective work, interdisciplinarity, and intellectual discipline and rigor, and preached all of this by exceptional example.

I would also like to thank Bob Batterman, who encouraged me to pursue work in applied mathematics. Without your support, this whole adventure would certainly not have been possible, and if counterfactually possible, then not successful. It is after Bob's recommendation that I went on to take my first course in applied mathematics, and it is what ultimately led to this book. I also want to thank Rob Moir for the countless hours that we spent thinking about backward error analysis and its ramifications and implications, and also Piers Lawrence and Walid Mnif for many helpful suggestions and ideas.

Finally, I want to thank Rob Corless for his inspiring mentoring and for this exciting opportunity of collaboration. This has been quite an adventure, and I want to thank you in particular for putting up with my stubbornness and with my ill-conditioned schedule. It has been an immense (as immense as the book!) pleasure.

Contents

Part I Preliminaries

Part II Numerical Linear Algebra

Part III Interpolation, Differentiation, and Quadrature

Part IV Differential Equations

Part V Afterword

Part VI Appendices

List of Figures

List of Tables

Part I
Preliminaries

Computational mathematics, even without computers, is enormously powerful. Mathematical models of physical, biological, environmental, and social phenomena greatly increase our understanding of the world in which we live, and offer opportunities to achieve many desirable outcomes in many situations. This use of mathematical thinking is *old*: Imhotep (the earliest architect and engineer whose name is known to us, who worked in the time of Zhoser, about 2700 BCE) likely used mathematics in designing the first pyramids. Archimedes was famous for his intellectual help in the defence of Syracuse and of course his mechanical inventions survive in use to this day. Analog (not digital) computation is also very old—consider the Antikythera mechanism, which dates to about 100 BCE and had a tradition of similar instruments, now all lost, possibly lasting a 1,000 years.[4]

However useful mathematics is when only hand or analog computation is available, it seems obvious that mathematical models that are detailed enough to explain—and allow means of control over—even moderately complicated systems need significant computer help in order to provide useful accounts of their predictions. The main difficulty is the complexity of interactions of subsystems in each model.

To complicate matters even further, already in the nineteenth century certain impossibility results were being obtained: Abel and Galois showed that it is not possible to solve general polynomial equations of degree five or more in radicals (although there is a less-well-known algorithm using elliptic functions for the quintic itself). Liouville showed that many important integrals could not be expressed in terms of elementary functions (and provided a basic theory to decide just when this could in fact be done). Lindemann showed that π was transcendental. More such impossibility results arrived in the twentieth century. Yet in order to provide scientific and engineering answers, when the phenomenon of interest is being modelled by, say, a differential equation or partial differential equation, something has to be done.

The answer that is the foundation of this book is equally old: approximation. For example, Archimedes famously used an approximation method based on polygons to compute lower and upper bounds on π. In general, the basic idea of approximation is to give up on an exact answer, and to settle for "something close enough."

> The applications of mathematics are everywhere, not just in the traditional sciences of physics and chemistry, but in biology, medicine, agriculture and many more areas. Traditionally, mathematicians tried to give an exact solution to scientific problems or, where this was impossible, to give exact solutions to modified or simplified problems. With the birth of the computer age, the emphasis started to shift towards trying to build exact models but resorting to numerical approximations. (Butcher 2008a)

In one sense, the idea of a solution being "close enough" gave birth to the whole of mathematical analysis. In another, closer to the spirit of this book, it gave birth to scientific and engineering computation.

[4] For details of this, and more, see the lovely book by de Camp (1960). Concerning the Antikythera mechanism, see Freeth et al. (2008) and `http://www.antikythera-mechanism.gr/`. For a more modern 'classic' take on mathematical modeling, see Wan (1989).

We take a specific example, from the classic paper *The Chemical Basis of Morphogenesis* by Alan Turing (1952), one of the pioneers of modern scientific computing. Turing considered various mathematical models of chemical reactions involving an interaction between *reaction* of the chemical agents (or 'morphogens') and *diffusion* of these agents in the tissues of a cell. These models have the following form (in two space dimensions x and y, for time t):

$$\begin{aligned}\frac{\partial u}{\partial t} &= f(u,v) + \varepsilon_1 \Delta u \\ \frac{\partial v}{\partial t} &= g(u,v) + \varepsilon_2 \Delta v \qquad \text{(I.1)}\end{aligned}$$

where $\Delta u = {}^{\partial^2 u}/_{\partial x^2} + {}^{\partial^2 u}/_{\partial y^2} = u_{xx} + u_{yy}$ is the Laplacian. The functions $f(u,v)$ and $g(u,v)$ vary, depending on what exactly is being modelled, and the constants ε_1 and ε_2 model the possibly different rates of diffusion of the 'morphogens' u and v.

Turing pointed out that systems behaving in a way described by Eq. (I.1) could, in theory, start from a homogeneous and boring state and then, by the introduction of tiny imperfections in that homogeneity, more or less spontaneously evolve under this dynamic to a pronounced pattern. These patterns could be used as models for a great many phenomena, including in particular the patterns of spots on a leopard's coat. Turing's paper started an entire field of investigations of the properties of such models, which can potentially explain such diverse things as how cell networks grow (in particular how neurons connect to one another) and animal or robotic locomotion.[5]

Turing made great strides in detailed understanding of these model equations for systems in limited configurations (such as a ring of cells), and fundamental understanding of what *might* happen in quite general configurations starting from a homogeneous initial configuration. However, as to further progress, perhaps it is best to give Turing's own words:

> Most of an organism, most of the time, is developing from one pattern into another, rather than from homogeneity into a pattern. One would like to be able to follow this more general process mathematically also. The difficulties are, however, such that one *cannot hope* [emphasis added] to have any very embracing *theory* of such processes, beyond the statement of the equations. It might be possible, however, to treat a few particular cases in detail with the aid of a digital computer. This method has the advantage that it is not so necessary to make simplifying assumptions as it is when doing a more theoretical type of analysis. (Turing 1952, pp. 71–72)

Turing also points out that such a computational approach has a disadvantage, too, namely that it only gets results for particular cases; he then claims that this disadvantage is "probably of comparatively little importance." We agree, and so do the large number of people doing simulations of specific Turing systems in order to achieve understanding (and in some cases control) of certain real systems of interest. In particular, if one is interested in *dynamic* Turing patterns, as Turing himself alludes to above, there seems to be no recourse other than computation.

[5] See, e.g., Arena and Fortuna (2002). Some recent work can be found at Leppänen et al. (2002), and this is an active area.

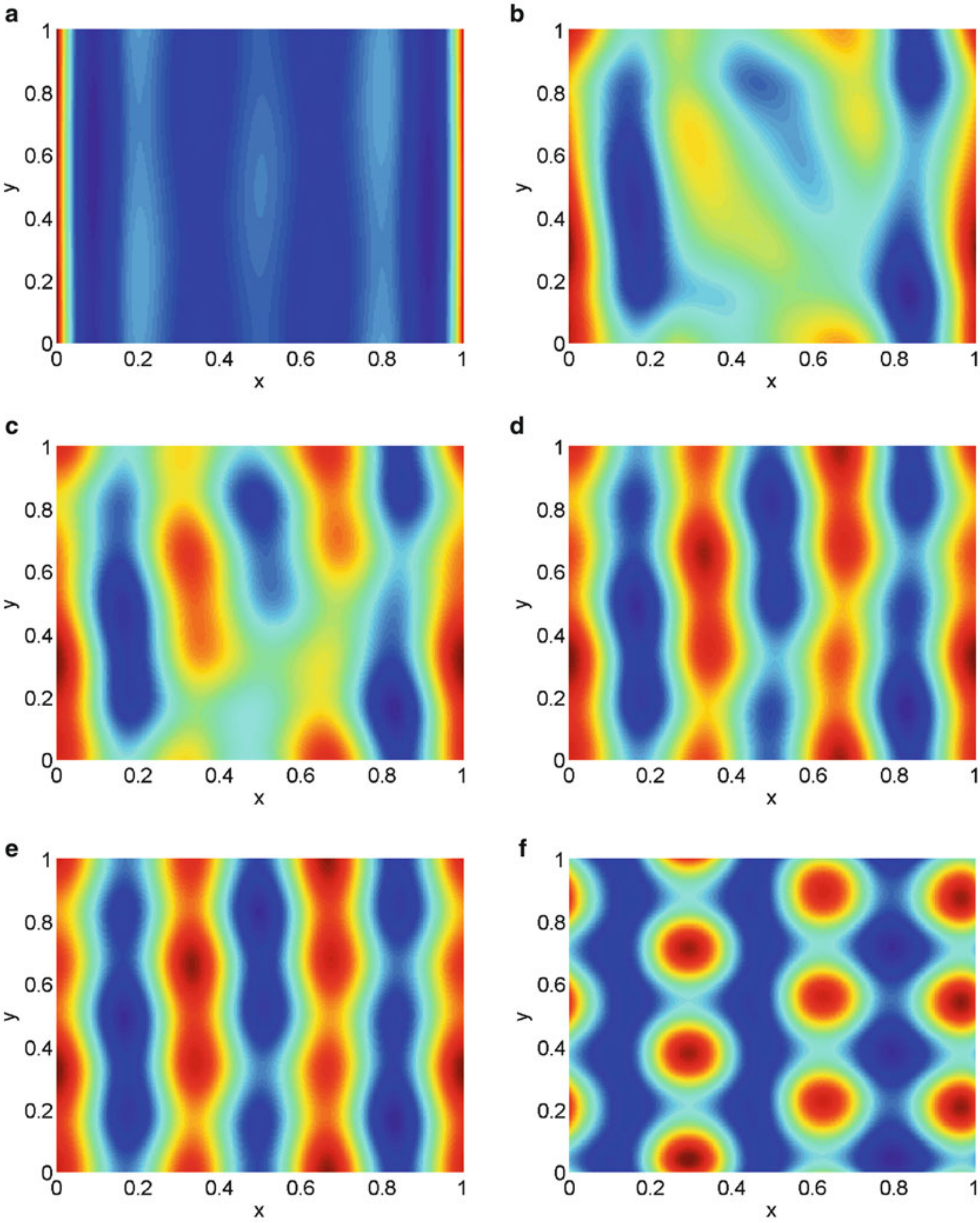


Fig. I.1 Snapshots of the solution to the Schnakenberg equations at various times. A static Turing pattern has evolved by $t = 2$. (**a**) $t = 0.01$. (**b**) $t = 0.02$. (**c**) $t = 0.03$. (**d**) $t = 0.05$. (**e**) $t = 0.06$. (**f**) $t = 2.00$

As a definite example[6] (of an ultimately static pattern), consider the Schnakenberg equations:

$$\begin{aligned} u_t &= \lambda(0.126779 - u + u^2 v) + \Delta u \\ v_t &= \lambda(0.792366 - u^2 v) + 10\Delta v\,, \end{aligned} \tag{I.2}$$

where $\lambda = 1000$. We will revisit these equations in Chap. 16. For now, note that these equations, unlike the ones studied by Turing himself, are *nonlinear* because of the u^2v term. It is extremely unlikely that we will find an exact, closed-form solution. Yet, using approximate computation, we can simulate the solution and observe changes as pictured in Fig. I.1.

To understand in what sense these solutions are approximate, and to understand how to produce controlled solutions of your own to problems like these, we recommend that you continue to read…

To solve problems stemming from real applications, it is essential to gain an understanding of the methods and strategies involved in numerical solutions. Moreover, even if a problem is formulated in terms of partial differential equations, the numerical strategies used essentially depend on more basic methods that efficiently and accurately solve problems of arithmetic, function evaluation, finding zeros, linear algebra, interpolation, differentiation and integration, *etc*. Step by step, chapter by chapter, this book will introduce you to elementary and more advanced methods that are necessary to solve those problems that arise in applications.

[6] We take this example from Ruuth (1995).

Chapter 1
Computer Arithmetic and Fundamental Concepts of Computation

Abstract This chapter introduces the main concepts of error analysis used in this book. The chapter defines *reference problems* and *modified problems* and notation to distinguish them. Two kinds of modified problems are shown to be particularly important in numerical analysis, namely, *engineered* and *reverse-engineered* problems. The reader is introduced to three concepts of error: (*forward error*, *backward error*, and *residual*), to the concept of *conditioning*, and to residual-based backward error analysis—which is the method favored in this book. We define numerical properties of algorithms, including *stability* and *cost*. Finally, we apply those concepts to *floating-point arithmetic*. ◁

As we have explained in the preface, there are two main paths that one can follow with this book: a theoretical path that starts with this chapter, and a pragmatic path that starts with Chap. 4 (see Fig. 1). If you are following the theoretical path—thus reading this chapter first, before you read any other chapter—please be aware that it is among the most abstract: It provides logical and conceptual grounding for the rest of the book. We believe that the readers who prefer to consider concrete examples before encountering the general ideas of which they are instances will be better off on the first reading to start somewhere else, for example, with Chap. 4, and return to this chapter only after seeing the examples there. But if you are a theory-minded learner, then by all means this is the place to start.

1.1 Mathematical Problems and Computability of Solutions

We begin by introducing a few foundational concepts that we will use to discuss computation in the context of numerical methods, adding a few parenthetical remarks meant to contrast our perspective from others. We represent a mathematical problem by an operator φ, which has an *input* (data) space $\mathscr{I}$ as its domain and an *output* (result, solution) space $\mathscr{O}$ as its codomain:

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_1,

$$\varphi : \mathscr{I} \to \mathscr{O} ,$$

and we write $y = \varphi(x)$. In many cases, the input and output spaces will be $\mathbb{R}^n$ or $\mathbb{C}^n$, in which case we will use the function symbols $f, g, \ldots$ and accordingly write

$$y = f(z_1, z_2, \ldots, z_n) = f(\mathbf{z}) .$$

Here, y is the (exact) solution to the problem f for the input data $\mathbf{z}$.[1] But φ need not be a function; for instance, we will study problems involving differential and integral operators. That is, in other cases, both x and y will themselves be functions.

We can delineate two general classes of computational problems related to the mathematical objects x, y, and φ:

C1. *verifying* whether a certain output y is actually the value of φ for a given input x, that is, verifying whether $y = \varphi(x)$;

C2. *finding* the output y determined by applying the map φ to a given input x, that is, finding the y such that $y = \varphi(x)$.[2]

In this classification, we consider "inverse problems," that is, trying to find an input x such that $\varphi(x)$ is a desired (known) value y, to be instances of C2 in that this corresponds to computation of the possibly many-valued inverse function $\varphi^{-1}(y)$.

The computation required by each type of problem is normally determined by an *algorithm*, that is, by a procedure performing a sequence of primitive operations leading to the solution in a finite number of steps. Numerical analysis is a mathematical reflection on the complexity and numerical properties of algorithms in contexts that involve *data error* and *computational error*.

In the study of numerical methods, as in many other branches of mathematical sciences, the reflection involves a subtle concept of *computation*. With a precise model of computation at hand, we can refine our views on what's computationally achievable, and if it turns out to be, how much effort is required.

The classical model of computation used in most textbooks on logic, computability, and algorithm analysis stems from metamathematical problems addressed in the 1930s; specifically, while trying to solve Hilbert's *Entscheidungsproblem*, Turing developed a model of primitive mathematical operations that could be performed by some type of machine affording finite but unlimited time and memory. This model, which turned out to be equivalent to other models developed independently by Gödel, Church, and others, resulted in a notion of computation based on *effective computability*. From there, we can form an idea of what is "truly feasible" by further adding constraints on time and memory.

Nonetheless, scientific computation requires an alternative, *complementary* notion of computation, because the methods and the objectives are quite different from those of metamathematics. A first important difference is the following:

[1] We use boldface font for vectors and matrices.

[2] It is normally computationally simpler to verify whether a certain value satisfies an equation than finding a value that satisfies it.

> [. . .] The Turing model (we call it “classical”), with its dependence on 0s and 1s, is fundamentally inadequate for giving such a foundation to the modern scientific computation, where most of the algorithms—with origins in Newton, Euler, Gauss, et al.—are *real number algorithms*. (Blum et al. 1998 3)

Blum et al. (1998) generalize the ideas found in the classical model to include operations on elements of arbitrary rings and fields. But the difference goes even deeper:

> [R]ounding errors and instability are important, and numerical analysts will always be experts in the subjects and at pains to ensure that the unwary are not tripped up by them. But our central mission is to compute quantities that are typically uncomputable, from an analytic point of view, and to do it with lightning speed. (Trefethen 1992)

Even with an improved picture of effective computability, it remains that the concept that matters for a large part of applied mathematics (including engineering) is the different idea of *mathematical tractability*, understood in a context where there are error in the data and error in computation, and where approximate answers can be entirely satisfactory. Trefethen’s seemingly contradictory phrase “compute quantities that are typically uncomputable” underlines the complementarity of the two notions of computation.

This second notion of computability addresses the proper computational difficulties posed by the application of mathematics to the solution of practical problems from the outset. Certainly, both pure and applied mathematics heavily use the concepts of real and complex analysis. From real analysis, we know that every real number can be represented by a nonterminating fraction:

$$x = \lfloor x \rfloor .d_1 d_2 d_3 d_4 d_5 d_6 d_7 \cdots .$$

However, in contexts involving applications, only a finite number of digits is ever dealt with. For instance, in order to compute $\sqrt{2}$, one could use an iterative method (e.g., Newton’s method, which we cover in Chap. 3) in which the number of accurate digits in the expansion will depend upon the number of iterations. A similar situation would hold if we used the first few terms of a series expansion for the evaluation of a function.

However, one must also consider another source of error due to the fact that, within each iteration (or each term), only finite-precision numbers and arithmetic operations are being used. We will find the same situation in numerical linear algebra, interpolation, numerical integration, numerical differentiation, and so forth.

Understanding the effect of limited-precision arithmetic is important in computation for problems of continuous mathematics. Since computers only store and operate on finite expressions, the arithmetic operations they process necessarily incur an error that may, in some cases, propagate and/or accumulate in alarming ways.[3] In

[3] But let’s not panic: “These risks are very real, but the message was communicated all too successfully, leading to the current widespread impression that the main business of numerical analysis is coping with rounding errors (Trefethen 2008b).

this first chapter, we focus on the kind of error that arises in the context of computer arithmetic, namely, representation and arithmetic error. In fact, we will limit ourselves to the case of floating-point arithmetic, which is by far the most widely used. Thus, the two errors we will concern ourselves with are the error that results from representing a real number by a floating-point number and the error that results from computing using floating-point operations instead of real operations. For a brief review of floating-point number systems, the reader is invited to consult Appendix A.

Remark 1.1. The objective of this chapter is not so much an in-depth study of error in floating-point arithmetic as an occasion to introduce some of the most important concepts of error analysis in a context that should not pose important technical difficulty to the reader. In particular, we will introduce the concepts of residual, backward and forward error, and condition number, which will be the central concepts around which this book revolves. Together, these concepts will give solid conceptual grounds to the main theme of this book: *A good numerical method gives you nearly the right solution to nearly the right problem.* $\triangleleft$

1.2 Representation and Computation Error

Floating-point arithmetic does not operate on real numbers, but rather on floating-point numbers. This generates two types of *roundoff* errors: representation error and arithmetic error. The first type of error we encounter, *representation error*, comes from the replacement of real numbers by floating-point numbers. If we let $x \in \mathbb{R}$ and $\bigcirc : \mathbb{R} \to \mathbb{F}$ be an operator for the standard rounding procedure to the nearest floating-point number[4] (see Appendix A), then the *absolute representation error* Δx is

$$\Delta x = \bigcirc x - x = \hat{x} - x\,. \tag{1.1}$$

(We will usually write $\hat{x}$ for $x + \Delta x$.) If $x \neq 0$, the *relative representation error* δx is given by

$$\delta x = \frac{\Delta x}{x} = \frac{\hat{x} - x}{x}\,. \tag{1.2}$$

From those two definitions, we obtain the following useful equality if $x \neq 0$:

$$\hat{x} = x + \Delta x = x(1 + \delta x)\,. \tag{1.3}$$

The IEEE standard described in Appendix A guarantees that $|\delta x| < \mu_M$, where μ_M is half the machine epsilon ε_M. In this book, when no specification of which IEEE

[4] In this chapter, we will always assume that x and the other real numbers are within the range of $\mathbb{F}$ for the sake of simplicity. See Appendix A for an explanation of what happens outside this domain (i.e., overflow and underflow).

standard is given, it will by default be the IEEE-754 standard described in Appendix A. In a numerical computing environment such as MATLAB, $\varepsilon_M = 2^{-52} \approx 2.2 \cdot 10^{-16}$, so that $\mu_M \approx 10^{-16}$.

The IEEE standard also guarantees that the floating-point sum of two floating-point numbers, written $\hat{z} = \hat{x} \oplus \hat{y}$,[5] is the floating-point number nearest the real sum $z = \hat{x} + \hat{y}$ of the floating-point numbers; that is, it is guaranteed that

$$\hat{x} \oplus \hat{y} = \bigcirc(\hat{x} + \hat{y})\,. \tag{1.4}$$

In other words, the floating-point sum of two floating-point numbers is the correctly rounded real sum. As explained in Appendix A, similar guarantees are given for $\ominus, \otimes$, and $\oslash$. Paralleling the definitions of Eqs. (1.1) and (1.2), we define the absolute and relative *computation errors* (for addition) by

$$\Delta z = \hat{z} - z = (\hat{x} \oplus \hat{y}) - (\hat{x} + \hat{y}) \tag{1.5}$$

$$\delta z = \frac{\Delta z}{z} = \frac{(\hat{x} \oplus \hat{y}) - (\hat{x} + \hat{y})}{\hat{x} + \hat{y}}\,. \tag{1.6}$$

As in Eq. (1.3), we obtain

$$\hat{x} \oplus \hat{y} = \hat{z} = z + \Delta z = z(1 + \delta z) \tag{1.7}$$

with $|\delta z| < \mu_M$. Moreover, the same relations hold for multiplication, subtraction, and division. These facts give us an automatic way to transform expressions containing elementary floating-point operations into expressions containing only real quantities and operations.

Remark 1.2. Similar but not identical relationships hold for floating-point complex number operations. If $z = x + iy$, then a complex floating-point number is a pair of real floating-point numbers, and the rules of arithmetic are inherited as usual. The IEEE real floating-point guarantees discussed above translate into the following:

$$\begin{aligned} fl(z_1 \pm z_2) &= (z_1 \pm z_2)(1 + \delta) \qquad & |\delta| &\le \mu_M \\ fl(z_1 z_2) &= (z_1 z_2)(1 + \delta) \qquad & |\delta| &\le \sqrt{2}\gamma_2 \\ fl({}^{z_1}/_{z_2}) &= ({}^{z_1}/_{z_2})(1 + \delta) \qquad & |\delta| &\le \sqrt{2}\gamma_7\,, \end{aligned} \tag{1.8}$$

where the γ_k notation [in which $\gamma_k = {}^{k\mu_M}/_{(1-k\mu_M)}$] is as defined in Eq. (1.18) below. Division is done by a method that avoids unnecessary overflow but is slightly more complicated than the usual method (see Example 4.15). Proofs of these are given in Higham (2002). The bounds on the error are thus slightly larger for complex operations but of essentially the same character. ◁

[5] A note on notation: To make it clear that we are dealing with a floating-point counterpart of one of the elementary arithmetical operation $+, -, \times$, and $\div$, we will circle them. When we will discuss the floating-point counterparts of other operations, we will simply add "*fl*," such as $fl(\mathbf{x} \cdot \mathbf{y})$ for an inner product.

We can usually assume that $\sqrt{x}$ also provides the correctly rounded result, but it is not generally the case for other operations, such as e^x, $\ln x$, and the trigonometric functions (see Muller et al. 2009).

To understand floating-point arithmetic better, it is important to verify whether the standard axioms of fields are satisfied, or at least nearly satisfied. As it turns out, many standard axioms do not hold, not even nearly, and neither do their more direct consequences. Consider the following statements (for $\hat{x}, \hat{y}, \hat{z} \in \mathbb{F}$), *which are not always true in floating-point arithmetic*:

1. Associative law of $\oplus$:

$$\hat{x} \oplus (\hat{y} \oplus \hat{z}) = (\hat{x} \oplus \hat{y}) \oplus \hat{z} \tag{1.9}$$

2. Associative law of $\otimes$:

$$\hat{x} \otimes (\hat{y} \otimes \hat{z}) = (\hat{x} \otimes \hat{y}) \otimes \hat{z} \tag{1.10}$$

3. Cancellation law (for $\hat{x} \neq 0$):

$$\hat{x} \otimes \hat{y} = \hat{x} \otimes \hat{z} \ \Rightarrow \ \hat{y} = \hat{z} \tag{1.11}$$

4. Distributive law:

$$\hat{x} \otimes (\hat{y} \oplus \hat{z}) = (\hat{x} \otimes \hat{y}) \oplus (\hat{x} \otimes \hat{z}) \tag{1.12}$$

5. Multiplication cancelling division:

$$\hat{x} \otimes (\hat{y} \oslash \hat{x}) = \hat{y}. \tag{1.13}$$

In general, the associative and distributive laws fail, but commutativity still holds, as you will prove in Problem 1.15. As a result of these failures, mathematicians find it very difficult to work directly in floating-point arithmetic—its algebraic structure is weak and unfamiliar. However, thanks to the discussion above, we know how to translate a problem involving floating-point operations into a problem involving only real arithmetic on real quantities $(x, \Delta x, \delta x, \ldots)$. This approach allows us to use the mathematical structures that we are familiar with in algebra and analysis. So, instead of making our error analysis directly in floating-point arithmetic, we try to work on a problem that is *exactly* (or nearly exactly) equivalent to the original floating-point problem, by means of the study of perturbations of real (and eventually complex) quantities. This insight was first exploited systematically by J. H. Wilkinson.

1.3 Error Accumulation and Catastrophic Cancellation

In applications, it is usually the case that a large number of operations have to be done sequentially before results are obtained. In sequences of floating-point operations, arithmetic error may accumulate. The magnitude of the accumulating error will often be negligible for well-tested algorithms.[6] Nonetheless, it is important to be aware of the possibility of massive *accumulating rounding error* in some cases. For instance, even if the IEEE standard guarantees that, for $x, y \in \mathbb{F}$, $x \oplus y = \bigcirc(x+y)$,[7] it does not guarantee that equations of the form

$$\bigoplus_{i=1}^{k} x_i = \bigcirc \sum_{i=1}^{k} x_i\,, \qquad k > 2 \tag{1.14}$$

hold true. This can potentially cause problems for the computation of sums, for instance, for the computation of an inner product $\mathbf{x}\cdot\mathbf{y} = \sum_{i=1}^{k} x_i y_i$. In this case, the direct floating-point computation would be

$$\bigoplus_{i=1}^{k} (x_i \otimes y_i)\,, \tag{1.15}$$

summed from left to right following the indices. How big can the error be? Let us use our results from the last section in the case $n = 3$:

$$\begin{aligned} fl(\mathbf{x}\cdot\mathbf{y}) =& ((x_1 \otimes y_1) \oplus (x_2 \otimes y_2)) \oplus (x_3 \otimes y_3) \\ =& \Big(\big(x_1 y_1 (1+\delta_1) + x_2 y_2 (1+\delta_2)\big)(1+\delta_3) + x_3 y_3 (1+\delta_4)\Big)(1+\delta_5) \\ =& x_1 y_1 (1+\delta_1)(1+\delta_3)(1+\delta_5) \\ & + x_2 y_2 (1+\delta_2)(1+\delta_3)(1+\delta_5) \\ & + x_3 y_3 (1+\delta_4)(1+\delta_5). \end{aligned} \tag{1.16}$$

Note that the δ_is will not, in general, be identical; however, we need not pay attention to their particular values, since we are primarily interested in the fact that for real arithmetic $|\delta_i| \le \gamma_3$ for all of them, and for complex arithmetic $|\delta_i| \le \gamma_4$ in the θ-γ notation of Higham (2002) that we introduce below in order to clean up the presentation.

Theorem 1.1. *Consider a real floating-point system satisfying the IEEE standards, so that $|\delta_i| < \mu_M$. Moreover, let $e_i = \pm 1$ and suppose that $n\mu_M < 1$. Then*

$$\prod_{i=1}^{n} (1+\delta_i)^{e_i} = 1 + \theta_n\,, \tag{1.17}$$

where

[6] In fact, as explained by Higham (2002 chap. 1), errors can in some cases cancel each other out to give surprisingly accurate results.

[7] We are often only concerned with the *arithmetic* error resulting from implementing a given algorithm in floating-point arithmetic. In this case, we will drop the "ˆ" symbol when it does not result in confusion.

$$|\theta_n| \leq \frac{n\mu_M}{1-n\mu_M} =: \gamma_n\,. \tag{1.18}$$

Notice that, for double-precision floating-point arithmetic, the supposition $n\mu_M < 1$ will almost always be satisfied. Then we can rewrite Eq. (1.16) in the real case as

$$fl(\mathbf{x}\cdot\mathbf{y}) = x_1y_1(1+\theta_3) + x_2y_2(1+\theta_3') + x_3y_3(1+\theta_2), \tag{1.19}$$

where each $|\theta_j| \leq \gamma_j$, (and where θ_3 and θ_3' each represent three different rounding errors) so that the computation error satisfies

$$|\mathbf{x}\cdot\mathbf{y} - fl(\mathbf{x}\cdot\mathbf{y})| \leq \gamma_3 \sum_{i=1}^{3} |x_iy_i| = \gamma_3 |\mathbf{x}|^T |\mathbf{y}|\,. \tag{1.20}$$

This analysis obviously generalizes to the case of n-vectors, and a similar formula can be deduced for complex vectors; as explained in the solution to (Higham 2002 Problem 3.7), all that needs to be done is to replace γ_n in the above with γ_{n+2}. However, note that this is a worst-case analysis, which returns the maximum error that can result from the mere satisfaction of the IEEE standard. In practice, it will often be much better. In fact, if you use a built-in routine for inner products, the accumulating error will be well below that (see, e.g., Problem 1.50).

Example 1.1. Another typical case in which the potential difficulty with sums poses a problem is in the computation of the value of a function using a convergent series expansion and floating-point arithmetic. Consider the simple case of the exponential function (from Forsythe 1970), $f(x) = e^x$, which can be represented by the uniformly convergent series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots\,. \tag{1.21}$$

If we work in a floating-point system with a five-digit precision, we obtain the sum

$$\begin{aligned} e^{-5.5} &\approx 1.0000 - 5.5000 + 15.125 - 27.730 + 38.129 - 41.942 + 38.446 \\ &\quad - 30.208 + 20.768 - 12.692 + 6.9803 - 3.4902 + 1.5997 + \cdots \\ &= 0.0026363\,. \end{aligned}$$

This is the sum of the first 25 terms, following which the first few digits do not change, perhaps leading us to believe (incorrectly) that we have reached an accurate result. But, in fact, $e^{-5.5} \approx 0.00408677$, so that $\Delta y = \hat{y} - y \approx 0.0015$. This might not seem very much, when posed in absolute terms, but it corresponds to $\delta y = 35\%$, an enormous relative error! Note, however, that it would be within what would be guaranteed by the IEEE standard for this number system. To decrease the magnitude of the maximum rounding error, we would need to add precision to the number system, thereby decreasing the magnitude of the machine epsilon. But as we will see below, this would not save us either. We are better off to use a more accurate formula for e^{-x}, and it turns out that reciprocating the series for e^x works well for this example. See Problem 1.7. ◁

There usually are excellent built-in algorithms for the exponential function. But a similar situation could occur with the computation of values of some transcendental function for which no built-in algorithm is provided, such as the Airy function. The Airy functions (see Fig. 1.1) are solutions of the differential equation $\ddot{x} - tx = 0$ with certain standard initial conditions. The first Airy function can be defined by the integral

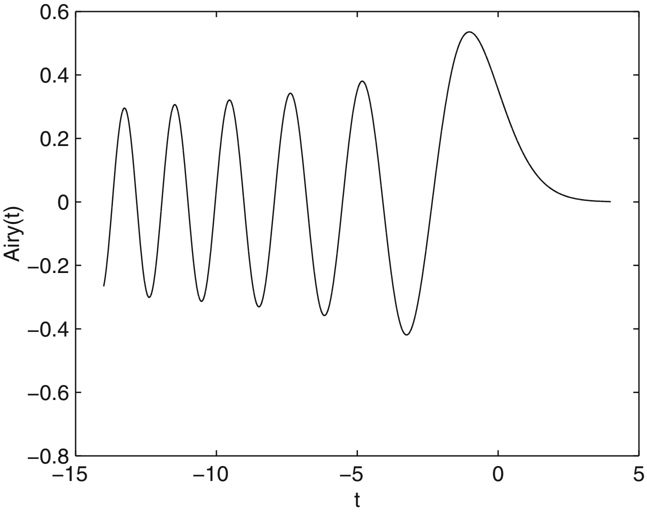


Fig. 1.1 The Airy function

$$\mathrm{Ai}(t) = \frac{1}{\pi}\int_0^\infty \cos\left(\frac{1}{3}\zeta^3 + t\zeta\right) d\zeta\,. \tag{1.22}$$

This function occurs often in physics. For instance, if we study the undamped motion of a weight attached to a Hookean spring that becomes linearly stiffer with time, we get the equation of motion $\ddot{x} + tx = 0$, and so the motion is described by $\mathrm{Ai}(-t)$ (Nagle et al. 2000). Similarly, the zeros of the Airy function play an important geometric role for the optics of the rainbow (Batterman 2002). And there are many more physical contexts in which it arises. So, how are we to evaluate it? The Taylor series for this function (which converges for all x) can be written as

$$\mathrm{Ai}(t) = 3^{-2/3}\sum_{n=0}^\infty \frac{t^{3n}}{9^n n!\,\Gamma(n+2/3)} - 3^{-4/3}\sum_{n=0}^\infty \frac{t^{3n+1}}{9^n n!\,\Gamma(n+4/3)} \tag{1.23}$$

(see Bender and Orszag (1978) and Chap. 3 of this book). As above, we might consider naively adding the first few terms of the Taylor series using floating-point operations, until apparent convergence (i.e., until adding new terms does not change the solution anymore because they are too small).

Of course, true convergence would require that, for every $\varepsilon > 0$, there existed an N such that $\left|\sum_{k\geq N+1}^{M} a_k\right| < \varepsilon$ for any $M > N$, that is, that the sequence of partial sums was a Cauchy sequence. There are many tests for convergence. Indeed, for this Taylor series, we can easily use the Lagrange form of the remainder and an accurate plot of the 31st derivative of the Airy function on this interval to establish that 30 terms in the series has an error less than 10^{-16} on the interval $-12 \leq z \leq 4$. Such analysis is not always easy, though, and it is often tempting to let the machine decide when to quit adding terms; and if the terms omitted could make no difference in floating-point, then we may as well stop anyway. Of course, examples exist where this approach fails, and some of them are explored in the exercises, but when the convergence is rapid enough, as it is for this example, then this device should be harmless though a bit inefficient.

We implement this in MATLAB in the routine below:

```
function [ Ai ] = AiTaylor( z )
%AiTaylor. Try to use (naively) the explicitly-known Taylor
% series about z=0 to evaluate Ai(z). Ignore rounding errors,
% overflow/underflow, NaN. The input z may be a vector of
% complex numbers.
%
%   y = AiTaylor( z );
%
THREETWOTH  = 3.0^(-2/3);
THREEFOURTH = 3.0^(-4/3);

Ai = zeros(size(z));
zsq = z.*z;
n = 0;
zpow = ones(size(z));  % zpow = z^(3n)

term = THREETWOTH*ones(size(z))/gamma(2/3);
% recall n! = gamma(n+1)
nxtAi = Ai + term;

% Convergence is deemed to occur when adding new terms makes no
    difference numerically.
while any( nxtAi ~= Ai ),
    Ai = nxtAi;
    zpow = zpow.*z;  % zpow = z^(3n+1)
    term = THREEFOURTH*zpow/9^n/factorial(n)/gamma(n+4/3);
    nxtAi = Ai - term;
    if all( nxtAi == Ai ), break, end;
    Ai = nxtAi;
    n = n + 1;
    zpow = zpow.*zsq;  % zpow = z^(3n)
    term = THREETWOTH*zpow/9^n/factorial(n)/gamma(n+2/3);
    nxtAi = Ai + term;
end

    % We are done.  If the loop exits, Ai = AiTaylor(z).
```

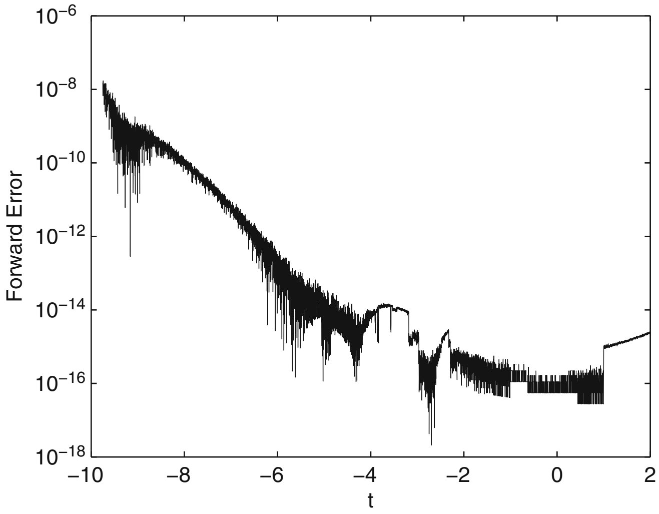


Fig. 1.2 Error in a naive MATLAB implementation of the Taylor series computation of Ai

```
end
```

Using this algorithm, can one expect to have a high accuracy, with error close to ε_M? Figure 1.2 displays the difference between the correct result (as computed with MATLAB's function `airy`) and the naive Taylor series approach. So, suppose we want to use this algorithm to compute $f(-12.82)$, a value near the 10th zero (counting from the origin toward $-\infty$); the absolute error is

$$\Delta y = |\mathrm{Ai}(x) - \mathrm{AiTaylor}(x)| = 0.002593213070374\,, \tag{1.24}$$

resulting in a relative error $\delta y \approx 0.277$. The solution is only accurate to two digits! Even though the series converges for all x, it is of little practical use. We examine this example in more detail in Chap. 2 when discussing the evaluation of polynomial functions.

The underlying phenomenon in the former examples, sometimes known as "the hump phenomenon," could also occur in a floating-point number system with higher precision. What happened exactly? If we consider the magnitude of some of the terms in the sum, we find out that they are much larger than the returned value (and the real value). We observe that this series is an alternating series in which the terms of large magnitude mostly cancel each other out. When such a phenomenon occurs—a phenomenon that Lehmer coined *catastrophic cancellation*—we are more likely to encounter erratic solutions. After all, how can we expect that numbers such as 38.129, a number with only five significant figures, could be used to accurately obtain the sixth or seventh figure in the answer? This explains why one must be careful in cases involving catastrophic cancellation.

Another famous example of catastrophic cancellation involves finding the roots of a degree-2 polynomial ax^2+bx+c using the quadratic equation (Forsythe 1966):

$$x_{\pm}^{*} = \frac{-b \pm \sqrt{b^2-4ac}}{2a}.$$

If we take an example for which $b^2 \gg 4ac$, catastrophic cancellation can occur. Consider this example:

$$a = 1\cdot 10^{-2} \qquad b = 1\cdot 10^{7} \qquad c = 1\cdot 10^{-2}.$$

Such numbers could easily arise in practice. Now, a MATLAB computation returns $x_{+}^{*} = 0$, which is obviously not a root of the polynomial. In this case, the answer returned is 100% wrong, in relative terms. Further exploration of this example will be made in Problem 1.18.

1.4 Perspectives on Error Analysis: Forward, Backward, and Residual-Based

The problematic cases can provoke a feeling of insecurity. When are the results provided by actual computation satisfactory? Sometimes, it is quite difficult to know intuitively whether it is the case. And how exactly should satisfaction be understood and measured? Here, we provide the concepts that will warrant confidence or non-confidence in some results based on an error analysis of the computational processes involved.

Our starting point is that problems arising in scientific computation are such that we typically do not compute the exact value $y = \varphi(x)$, for the *reference problem* φ, but instead some other more convenient value $\hat{y}$. The value $\hat{y}$ is not an exact solution of the reference problem, so that many authors regard it as an approximate solution, that is, $\hat{y} \approx \varphi(x)$. However, we will regard the quantity $\hat{y}$ as the *exact* solution of a modified problem, that is, $\hat{y} = \hat{\varphi}(x)$, where $\hat{\varphi}$ denotes the modified problem. For reasons that will become clearer later, we also call *some* modified problems *engineered problems*, because they arise on deliberately modifying φ in a way that makes computation easier or at least possible. We thus get this general picture:

$$\begin{tikzcd} x \arrow[r, "\varphi"] \arrow[dr, "\hat{\varphi}"'] & y \arrow[d, dash, "\Delta y"] \\ & \hat{y} \end{tikzcd} \tag{1.25}$$

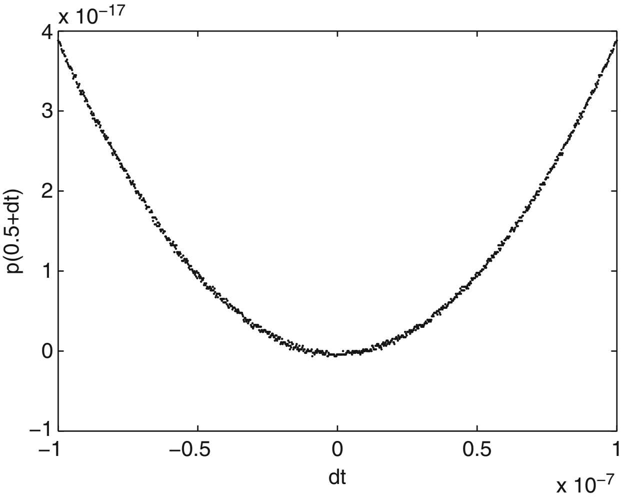


Fig. 1.3 Zooming in near a polynomial that we expect to have a double zero at $z = {}^{1}\!/_{2}$, we see the curve getting "fuzzy" as we get closer because of computational error in the evaluation of the polynomial

Example 1.2. Let us consider a simple case. If we have a simple problem of addition of real numbers to do, instead of computing $y = f(x_1, x_2) = x_1 + x_2$, we might compute $\hat{y} = \hat{f}(\hat{x}_1, \hat{x}_2) = \hat{x}_1 \oplus \hat{x}_2$. Here, we regard the computation of the floating-point sum as an engineered problem. In this case, we have

$$\begin{aligned} \hat{y} = \hat{x}_1 \oplus \hat{x}_2 &= x_1(1+\delta x_1) \oplus x_2(1+\delta x_2) \\ &= \big(x_1(1+\delta x_1) + x_2(1+\delta x_2)\big)(1+\delta x_3) \\ &= (x_1+x_2)\left(1 + \frac{x_1\delta x_1 + x_2\delta x_2}{x_1+x_2}\right)(1+\delta x_3), \end{aligned} \tag{1.26}$$

and so we regard $\hat{y}$ as the exact computation of the modified formula (1.26). ◁

Similarly, if the problem is to find the zeros of a polynomial, we can use various methods that will give us so-called pseudozeros, which are usually not zeros. Instead of regarding the pseudozeros as approximate solutions of the reference problem "find the zeros," we regard those pseudozeros as the *exact* solution to the modified problem "find some zeros of nearby polynomials," which is what we mean by pseudozeros (see Chap. 2). We point out that evaluation near multiple zeros is especially sensitive to computational error; see Figs. 1.3 and 1.4.

If the problem is to find a vector **x** such that **Ax** = **b**, given a matrix **A** and a vector **b**, we can use various methods that will give us a vector that almost satisfies the equation, but not quite. Then we can regard this vector as the solution for a matrix with slightly modified entries (see Chap. 4). The whole book is about cases of this sort arising from all branches of mathematics.

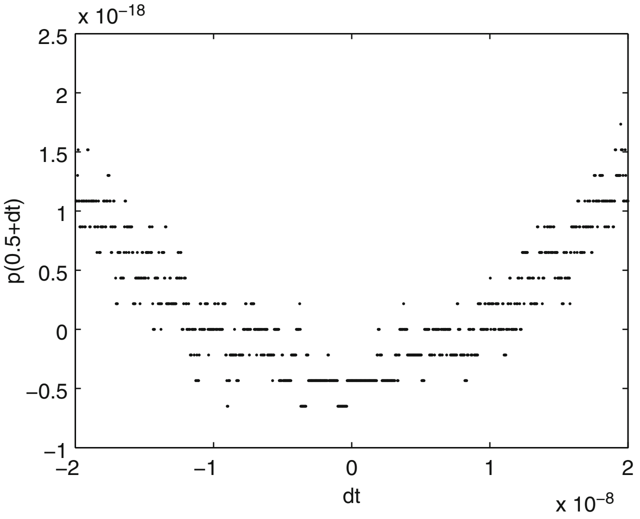


Fig. 1.4 Zooming in even closer, we see the curve broken up into discrete samples because of representation error of the computed values of the polynomial. It has also become apparent that the double zero has split to become two nearby simple zeros, each about $\sqrt{\mu_M}$ away from the reference zero $z = {}^1\!/\!_2$. Exactly which simple zeros best represent the zeros of "the" computational polynomial is not clear-cut

What is so fruitful about this seemingly trivial change in the way the problems and solutions are discussed? Once this change of perspective is adopted, we do not focus so much on the question, "How far is the computed solution from the exact one?" (i.e., in diagram 1.25, how big is Δy?), but rather on the question, "How closely related are the original problem and the engineered problem?" (i.e., in diagram 1.25, how closely related are φ and $\hat{\varphi}$?). If the modified problem behaves closely like the reference problem, we will say it is a *nearby problem*.

The quantity labeled Δy in diagram 1.25 is called the *forward error*, which is defined by

$$\Delta y = y - \hat{y} = \varphi(x) - \hat{\varphi}(x)\,. \tag{1.27}$$

We can, of course, also introduce the *relative* forward error by dividing by y, provided $y \neq 0$. In certain contexts, the forward error is in some sense the key quantity that we want to control when designing algorithms to solve a problem. Then, a very important task is to carry a forward error analysis; the task of such an analysis is to put an upper bound on $\|\Delta y\| = \|\varphi(x) - \hat{\varphi}(x)\|$. However, as we will see, there are also many contexts in which the control of the forward error is not so crucial.

Even in contexts requiring a control of the forward error, direct forward error analysis will play a very limited role in our analyses, for a very simple reason. We engineer problems and algorithms because we don't know or don't have efficient means of computing the solution of the reference problem. But directly computing the forward error involves solving a computational problem of type C2 (as defined on p. 8), which is often unrealistic. As a result, scientific computation presents us situations in which we usually don't know or don't have efficient ways of computing the forward error. Somehow, we need a more manageable concept that will also reveal if our computed solutions are good. Fortunately, there's another type of a priori error analysis—that is, antecedent to actual computation—one can carry out, namely, *backward error analysis*. We explain the perspective it provides in the next subsection. Then, in Sects. 1.4.2 and 1.4.3, we show how to supplement a backward error analysis with the notions of condition and residual in order to obtain an informative assessment of the forward error. Finally, in the next section, we will provide definitions for the stability of algorithms in these terms.

1.4.1 Backward Error Analysis

Let us generalize our concept of error to include any type of error, whether it comes from data error, measurement error, rounding error, truncation error, discretization error, and so forth. In effect, the success of backward error analysis comes from the fact that it treats all types of errors (physical, experimental, representational, and computational) on an equal footing. Thus, $\hat{x}$ will be some approximation of x, and Δx will be some absolute error that may be or may not be the rounding error. Similarly, in what follows, δx will be the relative error, that may or may not be the relative rounding error. The error terms will accordingly be understood as *perturbations of the initially specified data*. So, in a backward error analysis, if we consider the problem $y = \varphi(x)$, we will in general consider all the values of the data $\hat{x} = x(1+\delta x)$ satisfying a condition $|\delta x| < \varepsilon$, for some ε prescribed by the modeling context,[8] and not only the rounding errors determined by the real number x and the floating-point system. In effect, this change of perspective shifts our interest from particular values of the input data to sets of input data satisfying certain inequalities.

Now, if we consider diagram 1.25 again, we could ask: Can we find a perturbation of x that would have effects on φ comparable to the effect of changing the reference problem φ by the engineered problem $\hat{\varphi}$? Formally, we are asking: Can we find a Δx such that $\varphi(x+\Delta x) = \hat{\varphi}(x)$? The smallest such Δx is what is called the *backward error*. For input spaces whose elements are numbers, vectors, matrices, functions, and the like, we use norms as usual to determine what Δx is the backward error.[9] For other types of mixed inputs, we might have to use a set of norms for each component of the input. In case the reader needs it, Appendix C reviews basic facts about norms. The resulting general picture is illustrated in Fig. 1.5 (see, e.g., Higham 2002), and we see that this analysis amounts to *reflecting* the forward error *back* into the backward error. In effect, the question that is central to backward error analysis is, *when we modified the reference problem φ to get the engineered problem $\hat{\varphi}$, for what set of data have we actually solved the problem φ?* If solving the problem $\hat{\varphi}(x)$ amounts to having solved the problem $\varphi(x+\Delta x)$ for a Δx smaller

[8] Note that, since modeling contexts usually include the proper choice of scale, the value of ε will usually be given in relative rather than absolute terms.

[9] The choice of norm may be a delicate issue, but we will leave it aside for the moment.

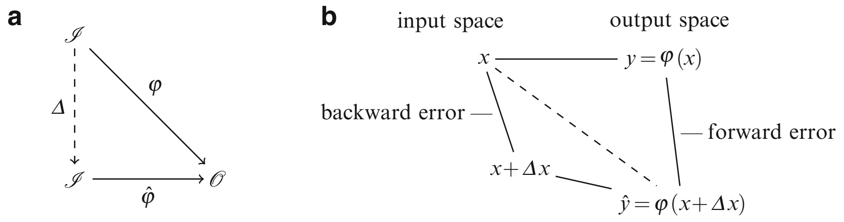


Fig. 1.5 Backward error analysis: the general picture. (**a**) Reflecting back the backward error: finding maps Δ. (**b**) Input and output space in a backward error analysis

than the perturbations inherent in the modeling context, then our solution $\hat{y}$ must be considered completely satisfactory.[10]

Adopting this approach, we benefit from the possibility of using well-known perturbation methods to talk about different problems and functions:

> The effects of errors in the data are generally easier to understand than the effects of rounding errors committed during a computation, because data errors can be analyzed using perturbation theory for the problem at hand, while intermediate rounding errors require an analysis specific to the given method. (Higham 2002 6)
>
> [T]he process of bounding the backward error of a computed solution is called *backward error analysis*, and its motivation is twofold. First, it interprets rounding errors as being equivalent to perturbations in the data. The data frequently contain uncertainties due to previous computations or errors committed in storing numbers on the computer. If the backward error is no larger than these uncertainties, then the computed solution can hardly be criticized—it may be the solution we are seeking, for all we know. The second attraction of backward error analysis is that it reduces the question of bounding or estimating the forward error to perturbation theory, which for many problems is well understood (and only to be developed once, for the given problem, and not for each method). (Higham 2002 7–8)

One can examine the effect of perturbations of the data using basic methods we know from calculus, various orders of perturbation theory, and the general methods used for the study of dynamical systems.

Example 1.3. Consider this (almost trivial!) example using only first-year calculus. Take the polynomial $p(x) = 17x^3 + 11x^2 + 2$; if there is a measurement uncertainty or a perturbation of the argument x, then how big will the effect be? One finds that

$$\Delta y = p(x + \Delta x) - p(x) = 51x^2\Delta x + 51x(\Delta x)^2 + 17(\Delta x)^3 + 22x\Delta x + 11(\Delta x)^2.$$

Now, since typically $|\Delta x| \ll 1$, we can ignore the higher degrees of Δx, so that

$$\Delta y \doteq 51x^2\Delta x\,.$$

Consequently, if $x = 1 \pm 0.1$, we get $y \doteq 35 \pm 5.1$; the perturbation in the input data has been magnified by about 50, and that would get worse if x were bigger. Also,

[10] There are cases, however, where finding such a Δx will not be possible. See Higham (2002 p. 71).

we can see from this analysis that if we want to know y to 5 decimal places, we will in general need an input accurate to 7 decimal places. ◁

Let us consider an example showing concretely how to reflect back the forward error into the backward error, in the context of floating-point arithmetic.

Example 1.4. Suppose we want to compute $y = f(x_1,x_2) = x_1^3 - x_2^3$ for the input $\mathbf{x} = [12.5, 0.333]$. For the sake of the example, suppose we have to use a computer working with a floating-point arithmetic with three-digit precision. So we will really compute $\hat{y} = ((x_1 \otimes x_1) \otimes x_1) \ominus ((x_2 \otimes x_2) \otimes x_2)$. We assume that $\mathbf{x}$ is a pair of floating-point numbers, so there is no representation error. The result of the computation is $\hat{y} = 1950$, and the exact answer is $y = 1953.014111$, leaving us with a forward error $\Delta y = 3.014111$ (or, in relative terms, $\delta y = {}^{3.014111}/_{1953.014111} \approx 1.5\%$). In a backward error analysis, we want to reflect the arithmetic (forward) error back in the data; that is, we need to find some Δx_1 and Δx_2 such that

$$\hat{y} = (12.5 + \delta x_1)^3 - (0.333 + \delta x_2)^3$$

A solution is $\Delta\mathbf{x} \approx [0.0064, 0]$ (whereby $\delta x_1 = 0.05\%$). But as one sees, the condition determines an infinite set of real solutions S, with real and complex elements. In such cases, where the entire set of solutions can be characterized, it is possible to find particular solutions, such as the solution that would minimize the 2-norm of the vector $\Delta\mathbf{x}$. See the discussions in Chaps. 4 and 6. ◁

Most of the time, we will want to use Theorem 1.1 to express the results of our backward error analyses. Consider again the case of the inner product from Eq. (1.19). The analysis we did for the three-dimensional case can be interpreted as showing that we have exactly evaluated the product $\big(\mathbf{x} + \Delta\mathbf{x}\big) \cdot \mathbf{y}$, where each perturbation is componentwise relatively small given by some θ_n (we could also have reflected back the error in $\mathbf{y}$). Specifically we have $\Delta x_1 = \theta_3 x_1$, $\Delta x_2 = \theta_3 x_2$, and $\Delta x_3 = \theta_2 x_3$. Thus, we have

$$fl(\mathbf{x} \cdot \mathbf{y}) = \big(\mathbf{x} + \Delta\mathbf{x}\big) \cdot \mathbf{y},$$

with $|\Delta\mathbf{x}| \le \gamma_n |\mathbf{x}|$. Thus, the floating-point inner product exactly solves the reference problem for slightly perturbed data (slightly more in the case of complex data). As a result:

Theorem 1.2. *The floating-point inner product of two n-vectors is backward stable.*

Note that the order of summation does not matter for this result to obtain. However, carefully choosing the order of summation will have an impact on the forward error.

1.4.2 Condition of Problems

We have seen how we can reflect back the forward error in the backward error. Now the question we ask is: *What is the relationship between the forward and the back-*

ward error? In fact, in modeling contexts, we are not really after an expression or a value for the forward error *per se*. The only reason for which we want to estimate the forward error is to ascertain whether it is smaller than a certain user-defined "tolerance," prescribed by the modeling context. To do so, all you need is to find how the perturbations of the input data (the so-called backward error we discussed) are magnified by the reference problem. Thus, the relationship we seek lies in a problem-specific coefficient of magnification, namely, the sensitivity of the solution to perturbations in the data, which we call the *conditioning of the problem*. The conditioning of a problem is measured by the *condition number*. As for the errors, the condition number can be defined in relative and absolute terms, and it can be measured normwise or componentwise.

The *normwise relative condition number* κ_{rel} is the maximum of the ratio of the relative change in the solution to the relative change in input, which is expressed by

$$\kappa_{rel} = \sup_x \frac{\|\delta y\|}{\|\delta x\|} = \sup_x \frac{\|{}^{\Delta y}/_{y}\|}{\|{}^{\Delta x}/_{x}\|} = \sup_x \frac{\|{}^{(\varphi(\hat{x})-\varphi(x))}/_{\varphi(x)}\|}{\|{}^{\hat{x}-x}/_{x}\|}$$

for some norm $\|\cdot\|$. As a result, we obtain the relation

$$\|\delta y\| \leq \kappa_{rel}\|\delta x\| \tag{1.28}$$

between the forward and the backward error. Knowing the backward error and the conditioning thus gives us an upper bound on the forward error.

In the same way, we can define the *normwise absolute condition number* κ_{abs} as $\sup_x {}^{\|\Delta y\|}/_{\|\Delta x\|}$, thus obtaining the relation

$$\|\Delta y\| \leq \kappa_{abs}\|\Delta x\|\,. \tag{1.29}$$

If κ has a moderate size, we say that the problem is *well-conditioned*. Otherwise, we say that the problem is *ill-conditioned*.[11] Consequently, even for a very good algorithm, the approximate solution to an ill-conditioned problem may have a large forward error.[12] It is important to observe that this fact is totally independent of any method used to compute φ. What matters is the existence of κ and what its size is.

Suppose that our problem is a scalar function. It is convenient to observe immediately that, for a sufficiently differentiable problem f, we can get an approximation of κ in terms of derivatives. Since

$$\lim_{\Delta x \to 0} \frac{\delta y}{\delta x} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} \cdot \frac{x}{y} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} \frac{x}{f(x)} = \frac{xf'(x)}{f(x)}\,,$$

the approximation of the condition number

$$\kappa_{rel} \approx \frac{|x||f'(x)|}{|f(x)|} \tag{1.30}$$

[11] When κ is unbounded, the problem is sometimes said to be *ill-posed*.

[12] Note the "may," which means that backward error analysis often provides pessimistic upper bounds on the forward error.

will provide a sufficiently good measure of the conditioning of a problem for small Δx. In the absolute case, we have $\kappa_{abs} \approx |f'(x)|$. This approximation will become useful in later chapters, and it will be one of our main tools in Chap. 3. If f is a multivariable function, the derivative $f'(x)$ will be the Jacobian matrix

$$\mathbf{J_f}(x_1,x_2,\ldots,x_n) = \left[{}^{\partial f}/_{\partial x_1} \; {}^{\partial f}/_{\partial x_2} \; \cdots \; {}^{\partial f}/_{\partial x_n}\right] ,$$

and the norm used for the computation of the condition number will be the induced matrix norm $\|\mathbf{J}\| = \max_{\|\mathbf{x}\|=1} \|\mathbf{Jx}\|$. In effect, this approximation amounts to ignoring the terms $O(\Delta x^2)$ in the Taylor expansion of $f(x+\Delta x) - f(x)$; using this approximation will thus result in a *linear error analysis*.

Though normwise condition numbers are convenient in many cases, it is often important to look at the internal structure of the arguments of the problem, for example, the dependencies between the entries of a matrix or between the components of a function vector. In such cases, it is better to use a componentwise analysis of conditioning. The relative componentwise condition number of the problem φ is the smallest number $\kappa_{rel} \geq 0$ such that

$$\max_i \frac{|f_i(\hat{x}) - f_i(x)|}{|f_i(x)|} \overset{\bullet}{\leq} k_{rel} \max_i \frac{|\hat{x}_i - x_i|}{|x_i|}, \quad \hat{x} \to x ,$$

where $\overset{\bullet}{\leq}$ indicate that the inequality holds in the limit $\Delta x \to 0$ (so, again, it holds for a linear error analysis). If the condition number is in this last form, we get a convenient theorem:

Theorem 1.3 (Deuflhard and Hohmann (2003)). *The condition number is submultiplicative; that is,*

$$\kappa_{rel}(g \circ h, x) \leq \kappa_{rel}(g, h(x)) \cdot \kappa_{rel}(h, x) .$$

In other words, the condition number of a composed problem $g \circ h$ evaluated near x is smaller than or equal to the product of the condition number of the problem h evaluated at x by the condition number of the problem g evaluated at $h(x)$. □

Consider three simple examples of condition number.

Example 1.5. Let us take the identity function $f(x) = x$ near $x = a$ (this is, of course, a trivial example). As one would expect, we get the absolute condition number

$$\kappa_{abs} = \sup \frac{|f(a+\Delta a) - f(a)|}{|\Delta a|} = \frac{|a + \Delta a - a|}{|\Delta a|} = 1 . \tag{1.31}$$

As a result, we get the relation $|\Delta y| \leq |\Delta x|$ between the forward and the backward error. κ_{abs} surely has moderate size in any context, since it does not amplify the input error. ◁

Example 1.6. Now, consider addition, $f(a,b) = a + b$. The derivative of f is

$$f'(a,b) = \left[{}^{\partial f}/_{\partial a} \quad {}^{\partial f}/_{\partial b}\right] = \begin{bmatrix} 1 & 1 \end{bmatrix} .$$

Suppose we use the 1-norm on the Jacobian matrix. Then the condition numbers are $\kappa_{abs} = \|f'(a,b)\|_1 = \|\begin{bmatrix}1 & 1\end{bmatrix}\|_1 = 2$ and

$$\kappa_{rel} = \frac{\left\|\begin{bmatrix}a\\b\end{bmatrix}\right\|_1}{\|a+b\|_1}\left\|\begin{bmatrix}1 & 1\end{bmatrix}\right\|_1 = 2\frac{|a|+|b|}{|a+b|}\,. \tag{1.32}$$

(Since the function is linear, the approximation of the definitions is an equality.) Accordingly, if $|a+b| \ll |a|+|b|$, we consider the problem to be ill-conditioned. ◁

Example 1.7. Consider the problem

$$a \xrightarrow{\varphi} \{x \mid x^2 - a = 0\}\,;$$

that is, evaluate x, where $x^2 - a = 0$. Take the positive root. Now here $x = \sqrt{a}$, so

$$|\delta x| = \left|\frac{f(a+\Delta a) - f(a)}{f(a)}\right| \overset{\bullet}{\leq} \left|\frac{af'(a)}{f(a)}\right| \frac{\Delta a}{a} = \frac{1}{2}\frac{\delta a}{a} \tag{1.33}$$

Thus, $\kappa = \frac{1}{2}$ is of moderate size, in a relative sense. However, note that in the absolute sense, the condition number is $\left(\sqrt{a+\Delta a}+\sqrt{a}\right)^{-1}$, which can be arbitrarily large as $a \to 0$. ◁

We will see many more examples throughout the book. Moreover, many other examples are to be found in Deuflhard and Hohmann (2003).

1.4.3 Residual-Based A Posteriori Error Analysis

The key concept we exploit in this book is the *residual*. For a given problem φ, the image y can have many forms. For example, if the reference problem φ consists in finding the roots of the equation $\xi^2 + x\xi + 2 = 0$, then for each value of x, the object y will be a set containing two numbers satisfying $\xi^2 + x\xi + 2 = 0$; that is,

$$y = \left\{\xi \;\middle|\; \xi^2 + x\xi + 2 = 0\right\}\,. \tag{1.34}$$

In general, we can then define a problem to be a map

$$x \xrightarrow{\quad\varphi\quad} \left\{\xi \mid \phi(x,\xi) = 0\right\}\,, \tag{1.35}$$

where $\phi(x,\xi)$ is some function of the input x and the output ξ. The function $\phi(x,\xi)$ is called the *defining function* and the equation $\phi(x,\xi) = 0$ is called the *defining equation* of the problem. On that basis, we can introduce the very important concept of *residual*: Given the reference problem φ—whose value at x is a y such that the defining equation $\phi(x,y) = 0$ is satisfied—and an engineered problem $\hat{\varphi}$, the residual r is defined by

$$r = \phi(x, \hat{y}) \,. \tag{1.36}$$

As we see, we obtain the residual by substituting the computed value $\hat{y}$ (i.e., the exact solution of the engineered problem) for y as the second argument of the defining function.

Let us consider some examples in which we apply our concept of residual to various kinds of problems.

Example 1.8. The reference problem consists in finding the roots of $a_2x^2 + a_1x + a_0 = 0$. The corresponding map is $\varphi(\mathbf{a}) = \{x \,|\, \phi(\mathbf{a}, x) = 0\}$, where the defining equation is $\phi(\mathbf{a}, x) = a_2x^2 + a_1x + a_0 = 0$. Our engineered problem $\hat{\varphi}$ could consist in computing the roots to three correct places. With the resulting "pseudozeros" $\hat{x}$, we can then easily compute the residual $r = a_2\hat{x}^2 + a_1\hat{x} + a_0$. We revisit this problem in Chap. 3. ◁

Example 1.9. The reference problem consists in finding a vector $\mathbf{x}$ such that $\mathbf{Ax} = \mathbf{b}$, for a nonsingular matrix $\mathbf{A}$. The corresponding map is $\varphi(\mathbf{A}, \mathbf{b}) = \{\mathbf{x} \,|\, \phi(\mathbf{A}, \mathbf{b}, \mathbf{x}) = \mathbf{0}\}$, where the defining equation is $\phi(\mathbf{A}, \mathbf{b}, \mathbf{x}) = \mathbf{b} - \mathbf{Ax} = \mathbf{0}$. In this case, the set is a singleton since there's only one such $\mathbf{x}$. Our engineered problem could consist in using Gaussian elimination in five-digit floating-point arithmetic. With the resulting solution $\hat{\mathbf{x}}$, we can compute the residual $\mathbf{r} = \mathbf{b} - \mathbf{A}\hat{\mathbf{x}}$. We revisit this problem in Chap. 4. ◁

Example 1.10. The reference problem consists in finding a function $x(t)$ on the interval $0 < t \leq 1$ such that

$$\dot{x}(t) = f(t, x(t)) = t^2 + x(t) - \frac{1}{10}x^4(t) \tag{1.37}$$

and $x(0) = 0$. The corresponding map is

$$\varphi\big(x(0), f(t,x)\big) = \{x(t) \,|\, \phi(x(0), f(t,x), x(t)) = 0\}, \tag{1.38}$$

where the defining equation is

$$\phi\big(x(0), f(t,x), x(t)\big) = \dot{x} - f(t,x) = 0, \tag{1.39}$$

together with $x(0) = 0$ (on the given interval). In this case, if the solution exists and is unique (as happens when f is Lipschitz), the set is a singleton since there's only one such $x(t)$. Our engineered problem could consist in using, say, a continuous Runge–Kutta method. With the resulting computed solution $\hat{z}(t)$, we can compute the residual $r = \dot{\hat{z}} - f(t, \hat{z})$. We revisit this theme in Chaps. 12 and 13. ◁

Many more examples of different kinds could be included, but this should sufficiently illustrate the idea for now.

In cases similar to Example 1.10, we can rearrange the equation $r = \dot{\hat{x}} - f(t, \hat{x})$ to have $\dot{\hat{x}} = f(t, \hat{x}) + r$, so that the residual is itself a perturbation (or a backward error) of the function defining the integral operator for our initial value problem. The new "perturbed" problem is

$$\tilde{\varphi}(x(0), f(t,x)+r(t,x)) = \{x(t) \mid \tilde{\phi}(x(0), f(t,x)+r(t,x), x(t)) = 0\}, \qquad (1.40)$$

and we observe that our computed solution $\hat{x}(t)$ is an exact solution of this problem. When such a construction is possible, we say that $\tilde{\varphi}$ is a *reverse-engineered problem*.

The remarkable usefulness of the residual comes from the fact that in scientific computation we normally choose $\hat{\varphi}$ so that we can compute it efficiently. Consequently, even if finding the solution of $\hat{\varphi}$ is a problem of type C2 (as defined on p. 8), it is normally not too computationally difficult because we engineered the problem specifically to guarantee it is so. All that remains to do to compute the residual is the evaluation of $\phi(x,\hat{y})$, a simpler problem of type C1. Thus, the computational difficulty of computing the residual is much less than that of the forward error. Accordingly, we can usually compute the residual efficiently, thereby getting a measure of the quality of our solution. Consequently, it is simpler to reverse-engineer a problem by reflecting back the residual into the backward error than by reflecting back the forward error.

Thus, the efficient computation of the residual allows us to gain important information concerning the reliability of a method on the grounds of what we have managed to compute with this method. In this context, we do not need to know as much about the intrinsic properties of a problem; we can use our computation method a posteriori to replace an a priori analysis of the reliability of the method. This allows us to use a feedback-control method to develop an adaptive procedure that controls the quality of our solution "as we go." This shows why a posteriori error estimation is tremendously advantageous in practice.

The residual-based a posteriori error analysis that we emphasize in this book thus proceeds as follows:

1. For the problem φ, use an engineered version of the problem to compute the value $\hat{y} = \hat{\varphi}(x)$.
2. Compute the residual $r = \phi(x,\hat{y})$.
3. Use the defining equation and the computed value of the residual to obtain an estimate of the backward error. In effect, this amounts to (sometimes only approximately) reflecting back the residual as a perturbation of the input data.
4. Draw conclusions about the satisfactoriness of the solution in one of two ways:

   a. If you do not require an assessment of the forward error, but only need to know that you have solved the problem for small enough perturbation Δx, conclude that your solution is satisfactory if the backward error (reflected back from the residual) is small enough.
   b. If you require an assessment of the forward error, examine the condition of the problem. If the problem is well-conditioned and the computed solution amounts to a small backward error, then conclude that your solution is satisfactory.

We still have to add some more concepts regarding the stability of algorithms, and we will do so in the next section.

But before, it is important not to mislead the reader into thinking that this type of error analysis solves *all* the problems of computational applied mathematics! There are cases involving a complex interplay of quantitative and qualitative properties that prove to be challenging. This reminds us of the following:

> A useful backward error-analysis is an explanation, not an excuse, for what may turn out to be an extremely incorrect result. The explanation seems at times merely a way to blame a bad result upon the data regardless of whether the data deserves a good result. (Kahan 2009)

Thus, even if the perspective on backward error analysis presented here is extremely fruitful, it does not cure all evils. Moreover, there are cases in which it will not even be possible to use the backward analysis framework. Here is a simple example:

Example 1.11. The outer product $\mathbf{A} = \mathbf{x}\mathbf{y}^T$ multiplies a column vector by a row vector to produce a rank-1 matrix. In floating-point arithmetic, the entries of the computed matrix $\hat{\mathbf{A}}$ will be $\hat{a}_{ij} = x_i \otimes y_j = x_i y_j (1+\delta)$ such that $|\delta| \leq \mu_M$. However, it is not possible to find perturbations $\Delta\mathbf{x}$ and $\Delta\mathbf{y}$ such that

$$\hat{\mathbf{A}} = (\mathbf{x}+\Delta\mathbf{x})(\mathbf{y}+\Delta\mathbf{y})^T .$$

See Problem 1.19. Consequently, it certainly cannot hold for small perturbations! But then, we cannot use backward error analysis to analyze this problem. ◁

1.5 Numerical Properties of Algorithms

An algorithm to solve a problem is a complete specification of how, exactly, to solve it: each step must be unambiguously defined in terms of known operations, and there must only be a finite number of steps. Algorithms to solve a problem φ correspond to the engineered problems $\hat{\varphi}$. There are many variants on the definition of an algorithm in the literature, and we will use the term loosely here. As opposed to the more restrictive definitions, we will count as algorithms methods that may fail to return the correct answer, or perhaps fail to return at all, and sometimes the method may be designed to use random numbers, thus failing to be deterministic. The key point for us is that the algorithms allow us to do computation with satisfactory results, this being understood from the point of view of mathematical tractability discussed before.

Whether $\hat{\varphi}(x)$ is satisfactory can be understood in different ways. In the literature, the algorithm-specific aspect of satisfaction is developed in terms of the numerical properties known as *numerical stability*, or just stability for short. Unfortunately "stability" is perhaps the most overused word in applied mathematics, and there is a particularly unfortunate clash with the use of the word in the theory of dynamical systems. In the terms introduced here, the concept of stability used in dynamical systems—which is a property of problems, not numerical algorithms—correspond to "well-conditioning." For algorithms, "stability" refers to the fact that an algorithm returns results that are about as accurate as the problem and the resources available allow.

Remark 1.3. The takeaway message is that, following our terminology, well-conditioning and ill-conditioning are properties of problems, while stability and instability are properties of algorithms. ◁

The first sense of numerical stability corresponds to the forward analysis point of view: an algorithm $\hat{\varphi}$ is *forward stable* if it returns a solution $y = \hat{\varphi}(x)$ with a small forward error Δy. Note that, if a problem is ill-conditioned, there will typically not be any forward stable algorithm to solve it. Nonetheless, as we explained earlier, the solution can still be satisfactory from the backward error point of view. This leads us to define *backward stability*:

Definition 1.1. An algorithm $\hat{\varphi}$ engineered to compute $y = \varphi(x)$ is *backward stable* if, for any x, there is a sufficiently small Δx such that

$$\hat{y} = f(x + \Delta x), \qquad \|\Delta x\| \leq \varepsilon .$$

As mentioned before, what is considered "small," that is, how big ε is, is prescribed by the modeling context and, accordingly, is context-dependent. □

For example, the IEEE standard guarantees that $x \oplus y = x(1+\delta x) + y(1+\delta y)$, with $|\delta x|, |\delta y| \leq \mu_M$. Hence, the IEEE standard in effect guarantees that the algorithms for basic floating-point operations are backward stable.

Note that an algorithm returning values with large forward errors can be backward stable. This happens particularly when we are dealing with ill-conditioned problems. As Higham (2002 p. 35) puts it:

> From our algorithm we cannot expect to accomplish more than from the problem itself. Therefore we are happy when its error $\hat{f}(x) - f(x)$ lies within reasonable bounds of the error $f(\hat{x}) - f(x)$ caused by the input error.

On that basis, we can introduce the concept of stability that we will use the most. It guarantees that we obtain theoretically informative solutions, while at the same time being very convenient in practice. Often, we only establish that $\hat{y} + \Delta y = f(x + \Delta x)$ for some small Δx and Δy. We do so either for convenience of proof, or because of theoretical limitations, or because we are implementing an adaptive algorithm as we described in Sect. 1.4.3. Nonetheless, this is often sufficient from the point of view of error analysis. This leads us to the following definition (de Jong 1977; Higham 2002):

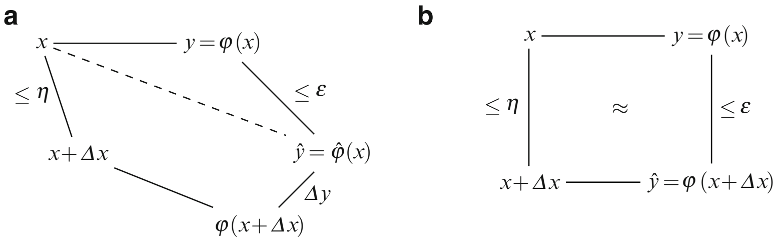


Fig. 1.6 Stability in the mixed forward–backward sense. (**a**) Representation as a commutative diagram (Higham 2002). (**b**) Representation as an "approximately" commuting diagram (Robidoux 2002). We can replace '≈' by the order to which the approximation holds

Definition 1.2. An algorithm $\hat{\varphi}$ engineered to compute $y = \varphi(x)$ is *stable in the mixed forward–backward sense* if, for any x, there are sufficiently small Δx and Δy such that

$$\hat{y} + \Delta y = f(x + \Delta x)\,, \quad \|\Delta y\| \leq \varepsilon \|y\|\,, \quad \|\Delta x\| \leq \eta \|x\|\,. \tag{1.41}$$

See Fig. 1.6. If this case, Eq. (1.41) is interpreted as saying that $\hat{y}$ is almost the right answer for almost the right data or, alternatively, that the algorithm $\hat{\varphi}$ nearly solves the right problem for nearly the right data. □

In most cases, when we will say that an algorithm is *numerically stable* (or just stable for short), we will mean it in the mixed forward–backward sense of (1.41).

The solution to a problem $\varphi(x)$ is often obtained by replacing φ by a finite sequence of simpler problems $\varphi_1, \varphi_2, \ldots, \varphi_n$. In effect, given that the domains and codomains of the simpler subproblems match, this amount to saying that

$$\varphi(x) = \varphi_n \circ \varphi_{n-1} \circ \cdots \circ \varphi_2 \circ \varphi_1(x)\,. \tag{1.42}$$

As we see, this is just composition of maps. For example, if the problem $\varphi(\mathbf{A}, \mathbf{b})$ is to solve the linear equation $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$, we might use the LU factoring (i.e., $\mathbf{A} = \mathbf{LU}$ for a lower-triangular matrix $\mathbf{L}$ and an upper-triangular matrix $\mathbf{U}$) factorization to obtain the two equations

$$\mathbf{Ly} = \mathbf{Pb} \tag{1.43}$$

$$\mathbf{Ux} = \mathbf{y}\,. \tag{1.44}$$

We have then decomposed $\mathbf{x} = \varphi(\mathbf{A}, \mathbf{b})$ into two problems; the first problem $\mathbf{y} = \varphi_1(\mathbf{L}, \mathbf{P}, \mathbf{b})$ consists in the simple task of solving a lower-triangular system and the second problem $\mathbf{x} = \varphi_2(\mathbf{U}, \mathbf{y})$ consists in the simple task of solving an upper-triangular system (see Chap. 4).

Remark 1.4. Such decompositions are hardly unique. A good choice of $\varphi_1, \varphi_2, \ldots, \varphi_n$ may lead to a good algorithm for solving φ in this way: Solve $\varphi_1(x)$ using its stable algorithm to get $\hat{y}_1$, then solve $\varphi_2(\hat{y}_1)$ using its stable algorithm to get $\hat{y}_2$, and so on. If the subproblems φ_1 and φ_2 are also well-conditioned, by Theorem 1.3, it follows that the resulting composed numerical algorithm for φ is numerically stable. (The same principle can be use as a very accurate rule of thumb for the formulations of the condition number not covered by Theorem 1.3). ◁

The *converse* statement is also very useful: Decomposing a *well-conditioned* φ into two *ill-conditioned* subproblems $\varphi = \varphi_2 \circ \varphi_1$ will usually result in an *unstable* algorithm for φ, even if stable algorithms are available for each of the subproblems (unless, as seems unlikely, the errors in $\hat{\varphi}_1$ and $\hat{\varphi}_2$ cancel each other out).

To a large extent, any numerical methods book is about decomposing problems into subproblems, and examining the correct numerical strategies to solve the subproblems. In fact, if you take any problem in applied mathematics, chances are that it will involve as subproblems things such as evaluating functions, finding roots of polynomials, solving linear systems, finding eigenvalues, interpolating function values, and so on. Thus, in each chapter, a small number of "simple" problems will be examined, so that you can construct the composed algorithm that is appropriate for your own composed problems.

1.6 Complexity and Cost of Algorithms

So far, we have focused on the accuracy and stability of numerical methods. In fact, most of the content of this book will focus more on accuracy and stability than on cost of algorithms and complexity of problems. Nonetheless, we will at times need to address issues of complexity. To evaluate the cost of some method, we need two elements: (1) a count of the number of elementary operations required by its execution and (2) a measure of the amount of resources required by each type of elementary operation, or group of operations. Following the traditional approach, we will only include the first element in our discussion.[13] Thus, when we will discuss the cost of algorithms, we will really be discussing the number of floating-point operations (*flops*[14]) required for the termination of an algorithm. Moreover, following a common convention, we will consider one flop to be one addition, one multiplication, and one comparison.

Example 1.12. If we take two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, the inner product

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

requires n flops. Thus, the multiplication of two arbitrary $n \times n$ matrices requires n^3 flops, since each entry is computed by an inner product.

Note that the order of operations may affect the flop count. If we also take $\mathbf{z} \in \mathbb{R}^n$, there will be a difference between $(\mathbf{x}\mathbf{y}^T)\mathbf{z}$ and $\mathbf{x}(\mathbf{y}^T\mathbf{z})$. In the former case, the first operation is an outer product forming an $n \times n$ matrix, which require n^2 flops. It is followed by a matrix–vector multiplication; this is equivalent to n inner products, each requiring n flops. Thus, the cost is $n^2 + n^2 = 2n^2$. However, if we instead compute $\mathbf{x}(\mathbf{y}^T\mathbf{z})$, the first operation is a scalar product (n flops) and the second operation is a multiplication of a vector by a scalar (n flops), which together require $2n$ flops. ◁

Note that sometimes the vectors, matrices, or other objects on which we operate will have a particular structure that we will be able to exploit to produce more efficient algorithms. The *computational complexity* of a problem is the cost of the algorithm

[13] The second element, particularly memory resources, is very relevant in practice today; in fact, possibly *more* relevant than the cost of floating-point, since one can demonstrate that computation time can sometimes be accurately be accurately estimated from memory requirements alone.

[14] In computer science, the acronym "flops" is sometimes used to denote flop/s, or floating-point operations per second. Here, the "s" only marks the plural of "flop."

solving this problem with the least cost, that is, what it would require to solve the problem using the cheapest method.

Typically, we will not be too concerned with the exact flop count. Rather, we will only provide an order of magnitude determined by the highest-order terms of the expressions for the flop count. Thus, if an algorithm taking an input of size n requires $n^2/2 + n + 2$ flops, we will simply say that its cost is $n^2/2 + O(n)$ flops, or even just $O(n^2)$ flops. This way of describing cost is achieved by means of *asymptotic notation*. The asymptotic notation uses the symbols Θ, O, Ω, o and ω to describe the comparative rate of growth of functions of n as n becomes large. In this book, however, we will only use the big-O and small-o notation, which are defined as follows:

$$\begin{aligned} f(n) = O(g(n)) \quad &\text{iff} \quad \exists c > 0 \exists n_0 \forall n \geq n_0 \quad \text{such that} \quad 0 \leq f(n) \leq c \cdot g(n) \\ f(n) = o(g(n)) \quad &\text{iff} \quad \forall c > 0 \exists n_0 \forall n \geq n_0 \quad \text{such that} \quad 0 \leq f(n) \leq c \cdot g(n). \end{aligned} \tag{1.45}$$

Intuitively, a function $f(n)$ is $O(g(n))$ when its rate of growth with respect to n is the same or less than the rate of growth of $g(n)$, as depicted in Fig. 1.7 (in other words, $\lim_{n\to\infty} f(n)/g(n)$ is bounded). A function $f(n)$ is $o(g(n))$ in the same circumstances, except that the rate of growth of $f(n)$ must be strictly less than $g(n)$'s (in other words, $\lim_{n\to\infty} f(n)/g(n)$ is zero). Thus, $g(n)$ is an asymptotic upper bound for $f(n)$. However, with the small-o notation, the bound is not tight.

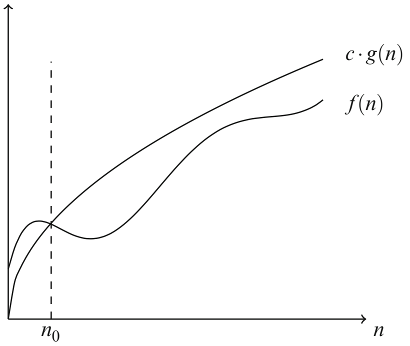


Fig. 1.7 Asymptotic notation: $f(n) = O(g(n))$ if, for some c, $cg(n)$ asymptotically bounds $f(n)$ above as $n \to \infty$

In our context, if we say that the cost of a method is $O(g(n))$, we mean that as n becomes large, the number of flops required will be at worst $g(n)$ times a constant. Some standard terminology to qualify cost growth, from smaller to larger growth rate, in introduced in Table 1.1. We will also use this notation when writing sums. See Sect. 2.8.

This notation is also used to discuss *accuracy*, and work-accuracy relationships. We will often want to analyze the cost of an algorithm as a function of a parameter, typically a dimension, say n, or a grid size, say h. The interesting limits are as the dimension goes to infinity or as the grid size goes to zero. The residual or backward error will typically go to zero as some power of h or inverse power of n (sometimes faster, in which case we say the convergence is *spectral*). If we have the error behaving as $\|\Delta\| = O(h^p)$ as $h \to 0$, we say the method has *order* p, and similarly if $\|\Delta\| = O(n^{-p})$. The asymptotic O-symbol hides a constant that may or may not be important.

Table 1.1 Common growth rates

The cost $f(n)$ is	The growth rate if the cost is
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n\log n)$	Quasilinear
$O(n^2)$	Quadratic
$O(n^k)$, $k = 2, 3, \ldots$	Polynomial
$O(c^n)$	Exponential

One useful trick for *measuring* the rate of convergence of a problem is to use a Fibonacci sequence[15] of dimension parameters, measure the errors for each dimension (this is typically easy if the error is a backward error), and plot the results on a log–log graph. This is called a work-accuracy diagram because the work increases as n increases (usually as a power of n itself) and the slope of the line of best fit then estimates p. We do this at several places in the book.

1.7 Notes and References

For a presentation of the classical model of computation, see, for instance, Davis (1982), Brassard and Bratley (1996), Pour-El and Richards (1989), and for a specific discussion of what is "truly feasible," see Immerman (1999).

Brent and Zimmermann (2011) provides a recent extensive discussion of algorithms and models of computer arithmetic, including floating-point arithmetic.

For an alternative, more formal presentation of the concepts presented here to systematically articulate backward error analysis, see Deuflhard and Hohmann (2003 chap. 2). The "reflecting back" terminology goes back to Wilkinson (1963). For a good historical essay on backward error analysis, see Grcar (2011).

Many other examples of numerical surprises can be found in the paper "Numerical Monsters," by Essex et al. (2000). The experience of W. Kahan in constructing floating-point systems to minimize the impact on computation has been

[15] Why use a Fibonacci sequence or something like it? Because they grow exponentially, but not as quickly as doubling the dimension does, and this often produces a more pleasing density of results on the graph.

presented in a systematic way in the entertaining and informative talk (Kahan and Darcy 1998). Many of his other papers are available on his website at `http://www.cs.berkeley.edu/~wkahan`.

Problems

Theory and Practice

1.1. Suppose you're an investor who will get interest daily (for an annual rate of, say 5%) on \$1,000,000. Your interest can be calculated in one of two ways: (a) The sum is calculated every day, and rounded to the nearest cent. This new amount will be used to calculate your sum on the next day. (b) Your sum is calculated only once at the end of the year with the formula $M_f = M_i(1+i_d)^d$, and then rounded to the nearest cent.

1. Which method should you choose? How big is the difference? How much smaller is it than the worst-case scenario obtained from mere satisfaction of the IEEE standard? Explain in terms of floating-point error.
2. If the rounding procedure used for the floating-point arithmetic was "round toward zero," would you make the same decision?

Explain the correspondence between computational error and real-world operations.

1.2. An important value to determine in the analysis of alternating current circuits is the capacitive reactance X_C, which is given by

$$X_C = -\frac{1}{2\pi f C},$$

where f is the frequency of the signal (in Hertz) and C is the capacitance (in Farads). It is common to encounter the values $f = 60$ Hz while C is the range of picofarads (i.e., $10^{-12}F$). Given this, could we expect MATLAB to accurately compute the reactive capacitance in common situations? Also, look up common values for the tolerance in the value of C provided by manufacturers. Would the rounding error be smaller than the error due to the tolerance? In at most a few sentences, discuss the significance of your last answer for assessing the quality of computed solutions.

1.3. Suppose you want to use MATLAB to help you with some calculations involved in special relativity. A common quantity to compute is the Lorentz factor γ defined by

$$\gamma = \frac{1}{\sqrt{1-\frac{v^2}{c^2}}},$$

where v is the relative velocity between two inertial frames in m/s and c is the speed of light, which is nearly equal to $299,792,458$ m/s. Will MATLAB provide results sufficiently precise to identify the relativistic effect of a vehicle moving at $v = 100.000$ km/h? Given the significant figures of v, is MATLAB's numerical result satisfactory? Compare your results with what you obtain from

$$(1-x^2)^{-1/2} = 1 + x^2/2 + O(x^4)\,. \tag{1.46}$$

1.4. Computing powers z^n for integers n and floating-point z can be done by simple repeated multiplication, or by a more efficient method known as *binary powering*. If $n = 2k+1$ is odd, replace the problem with that of computing $z \cdot z^{2k}$. If $n = 2k$ is even, replace the problem with that of computing $z^k \cdot z^k$. Recursively descend until $k = 1$. This can be done efficiently by looking at the bit pattern of the original n. Estimate the maximum number of multiplications are performed.

1.5. Suppose a, b are real but not machine-representable numbers. Compare the accuracy of computing $(a+b)^2$ as written and computing instead using the expanded form $a^2 + 2ab + b^2$. Are both methods backward stable? Mixed forward–backward stable? Would the difference between the methods, if any, become more important for $(a+b)^n$, $n > 2$? Give examples supporting your theoretical conclusions. You may use Problem 1.4.

1.6. Show that, for $a \neq 0$ and $b \neq 0$,

1. $25n^3 + n^2 + n - 4 = O(n^3)$;
2. any linear function $f(n) = an + b$ is $O(n^k)$ and $o(n^k)$ for integers $k \geq 2$;
3. no quasilinear function $an\log(bn)$ is $o(n\log(n))$.

1.7. Rework Example 1.1 using five-digit precision as before but compute instead $\exp(5.5)$ and then take the reciprocal. This uses the same numbers printed in the text, just all with positive signs. Is your final answer more accurate?

1.8. Euler was the first to discover[16] that

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}\,. \tag{1.47}$$

Write a program in MATLAB to sum the terms of this series in order (i.e., start with $k = 1$, then $k = 2$, *etc.*) until the double-precision sum is unaffected by adding another term. Record the number of terms taken (we found nearly 10^8). Compare the answer to $pi^2/6$ and record the relative accuracy. Write another program to evaluate the same sum in decreasing order of the values of k. What is the relative forward error in this case? Is it different? Is it significantly different? That is, is the accumulation of error reduced for a sum of positive numbers if we add the numbers from smallest to largest? (Higham 2002 1.12.3). Use the "integral test" from first-year calculus to estimate the true error in stopping the sum where you did, and estimate the number of terms you would have to take to get $\pi^2/6$ to as much accuracy as you could in double precision simply by summing terms.

[16] For a historical discussion of this, see the beautiful book Hairer and Wanner (1996), if you like, though it is not necessary for this problem.

1.9. The value of the Riemann zeta-function at 3 is

$$\zeta(3) = \sum_{k \geq 1} \frac{1}{k^3}\,. \tag{1.48}$$

Quite a lot is known about this number, but all you are asked to do here is to compute its value by simple summation as in the `AiTaylor` program and as in the previous problem, by simply adding terms until the next term is so small it has no effect after rounding. Use the integral test to estimate the actual error of your sum, and to estimate how many terms you would really need to sum to get double-precision accuracy. If you summed in reverse order, would you get an accurate answer?

1.10. Testing for convergence in floating-point arithmetic is tricky due to computational error. Discuss foreseeable difficulties and workarounds. In particular, you may wish to address the “method” used in the function `AiTaylor` of this chapter, namely to assume “convergence” of a series if adding a term t to a sum s produces $\hat{s} = s \oplus t$ that, after rounding, exactly equals s. Consider in particular what happens if you use this method on a *divergent* sum such as the harmonic series $H = 1 + 1/2 + 1/3 + 1/4 + \cdots$. (This is the source of many Internet arguments, by the way, but there is a clear and unambiguously correct way of looking at it.)

1.11. Show that computing the sum $\sum_{i=1}^{n} x_i$ naively term by term (a process called *recursive summation*) produces the result

$$\bigoplus_{i=1}^{n} x_i = \sum_{i=1}^{n} x_i(1+\delta_i), \tag{1.49}$$

where each $|\delta_i| \leq \gamma_{n+1-i}$ if $i \geq 2$ and $|\delta_1| \leq \gamma_{n-1}$ if $i = 1$.

There are a surprising number of different ways to sum n real numbers, as discussed in Higham (2002). Using Kahan’s algorithm for *compensated summation* as described below instead returns the computed sum

$$\sum_{i=1}^{n} x_i(1+\delta_i), \tag{1.50}$$

where now each $|\delta_i| < 2\mu_M + O(n\mu_M)$, according to Higham (2002) (you do not have to prove this). That is, compensated summation gains a factor of n in backward accuracy.

The algorithm in question is the following:

```
Require: A vector x with n components.
  $s := x_1$
  $c := 0$
  for $i$ from 2 to $n$ do
    $y := x_i - c$
    $t := s + y$
    $c := (t - s) - y$ % the order is important, and the parentheses too!
```

```
        $s := t$
    end for
    return $s$, the sum of the components of $\mathbf{x}$
```

Using some examples, compare the accuracy of naive recursive summation and of Kahan's sum. If you can, show that Eq. (1.50) really holds for your examples (Goldberg 1991).

1.12. For this problem, we work with a four-digit precision floating-point system. Note that $1+1=2$ gives no error since $1 \in \mathbb{F}$. In exact arithmetic, ${}^1\!/_3+{}^1\!/_3={}^2\!/_3$, but floating-point operations imply that $\frac{1}{3}(1+\delta_1)+\frac{1}{3}(1+\delta_2)=0.667$, from which we find that $\delta_1+\delta_2=(3\cdot 0.6667-2)=0.0001$. Show that $\max(|\delta_1|,|\delta_2|)$ is minimized if $|\delta_1|=|\delta_2|=5\cdot 10^{-5}$.

1.13. The following expressions are theoretically equivalent:

$$\begin{aligned}
s_1 &= 10^{20}+17-10+130-10^{20}\\
s_2 &= 10^{20}-10+130-10^{20}+17\\
s_3 &= 10^{20}+17-10^{20}-10+130\\
s_4 &= 10^{20}-10-10^{20}+130+17\\
s_5 &= 10^{20}-10^{20}+17-10+130\\
s_6 &= 10^{20}+17+130-10^{20}-10.
\end{aligned}$$

Nonetheless, a standard computer returns the values $0,17,120,147,137,-10$ (see, e.g.,Kulisch 2002 [8]). These errors stem from the fact that catastrophic cancellation takes place due to very different orders of magnitude. For each expression, find some values of δx_i, $1 \le i \le 5$, such that

$$s = x_1(1+\delta x_1)+x_2(1+\delta x_2)+x_3(1+\delta x_3)+x_4(1+\delta x_4)+x_5(1+\delta x_5)$$

with $|\delta x_i| < \mu_M$. In each case, find $\min\|\delta\mathbf{x}\|$.

1.14. Show that Eqs. (1.9), (1.10), (1.11), (1.12), and (1.13) do not generally hold for floating-point numbers.

1.15. Other laws of algebra for inequalities fail in floating-point arithmetic. Let $a,b,c,d \in \mathbb{F}$ (Parhami 2000 325):

1. Show that if $a<b$, then $a \oplus c \le b \oplus c$ holds for all c; that is, adding the same value to both sides of a strict inequality cannot affect its direction but may change the strict "$<$" relationship to "$\le$."
2. Show that if $a<b$ and $c<d$, then $a \oplus c \le b \oplus d$.
3. Show that if $c>0$ and $a<b$, then $a \otimes c \le b \otimes c$.

Assume that none of a, b, c, and d are `NaN`.

1.16. Higham (2002 1.12.2) considers what happens in floating-point computation when one first takes square roots repeatedly, and then squares the result repeatedly.

We here look at a slight variation, which (surprisingly for such an innocuous-looking computation) has something to do with an ancient but effective algorithm known as *Briggs' method* (Higham 2004 chapter 11). Here, write a MATLAB function that accepts a vector **x** as input, takes the square root 52 times, and then squares the result 52 times: theoretically achieving nothing. Call your function `Higham`. The algorithm is indicated below.

```
Require: A vector x
  for i from 1 to 52 do
    $x := \sqrt{x}$
  end for
  for i from 1 to 52 do
    $x := x^2$
  end for
  return a vector x, surprisingly different to the input
```

Then run

```
x = logspace( 0, 1, 2013 );
y = Higham( x );
plot( x, y, 'k.', x, x, '--' )
```

Explain the graph (see Fig. 1.8). (Hint: Identify the points where $y = x$ after all.)

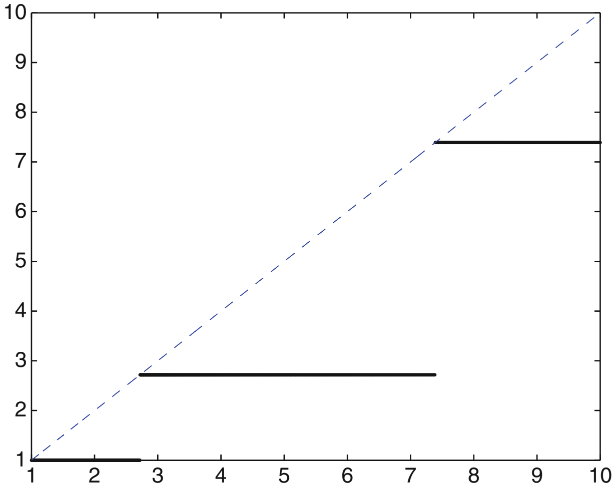


Fig. 1.8 The results of the code in Problem 1.16

1.17. We now know that unfortunate subtractions bring loss of significant figures. In fact, the subtraction per se does not introduce much error, but it reveals earlier error. On that basis, compare the following two methods to find the two roots of a second-degree polynomial:

1. Use the two cases of the quadratic formula;
2. Using the fact that $x_+x_- = c$ (where $x^2+bx+c=0$, i.e., $a=1$), keep the root among the two obtained with the quadratic formula that has the largest absolute value, and find the other one using the equation $x_+x_- = c$.

Which method is more accurate? Explain.

Investigations and Projects

1.18. Consider the quadratic equation $x^2+2bx+1=0$.

1. Show by the quadratic formula or otherwise that $x = -b \pm \sqrt{b^2-1}$ and that the product of the two roots is 1.
2. Plot $(-b+\sqrt{b^2-1})(-b-\sqrt{b^2-1})$, which is supposed to be 1, on a logarithmic scale in MATLAB as follows:

```
b = logspace( 6, 7.5, 1001 );
one = (-b-sqrt(b.^2-1) ).*(-b+sqrt(b.^2-1));
plot( b, one, '.' )
```

3. Using no more than one page of handwritten text (about a paragraph of typed text), partly explain why the plot looks the way it does.
4. If $b \gg 1$, which is more accurately evaluated in floating-point arithmetic, $-b-\sqrt{b^2-1}$ or $-b+\sqrt{b^2-1}$? Why?

1.19. Consider the outer product of two vectors $\mathbf{x} \in \mathbb{C}^m$ and $\mathbf{y} \in \mathbb{C}^n$: $\mathbf{P} = \mathbf{x}\mathbf{y}^H \in \mathbb{C}^{m\times n}$ with $p_{ij} = x_i\bar{y}_j$. Show that if $mn > m+n$, then rounding errors in computing this object cannot be modeled as a backward error; in other words, show that $\hat{\mathbf{P}}$ is not the exact outer product of any two perturbations $\mathbf{x}+\Delta\mathbf{x}$ and $\mathbf{y}+\Delta\mathbf{y}$.

1.20. Let $p = 1/2$. Consider the mathematically equivalent sums

$$1 = \sum_{k\geq 1} \frac{1}{k^p} - \frac{1}{(k+1)^p} \tag{1.51}$$

$$= \sum_{k\geq 1} \frac{(k+1)^p - k^p}{k^p(k+1)^p} \tag{1.52}$$

$$= \sum_{k\geq 1} \frac{1}{k^p(k+1)^p((k+1)^p+k^p)} . \tag{1.53}$$

Which of these is the most accurate to evaluate in floating-point using naive recursive summation? Why?

1.21 (Zeno's paradox: The dichotomy). One of the classical paradoxes of Zeno runs (more or less) as follows: A pair of dance partners are two units apart and wish to move together, each moving one unit. But for that to happen, they must first each move half a unit. After they have done that, then they must move half of the distance remaining. After *that*, they must move half the distance yet remaining, and so on. Since there are an infinite number of steps involved, logical difficulties seem to arise and indeed there is puzzlement in the first-year calculus class regarding things like this, although in modern models of analysis this paradox has long since been resolved. Roughly speaking, the applied mathematics view is that after a finite number of steps, the dancers are close enough for all practical purposes!

In MATLAB, we might phrase the paradox as follows. By symmetry, replace one partner with a mirror. Then start the remaining dancer off at $s_0 = 0$. The mirror is thus at $s = 1$. The first move is to $s_1 = s_0 + (1 - s_0)/2$. The second move is to $s_2 = s_1 + (1 - s_1)/2$. The third move is to $s_3 = s_2 + (1 - s_2)/2$, and so on. This suggests the following loop.

```
s = 0
i = 0
while s < 1,
    i = i+1;
    s = s + (1-s)/2;
end
disp( sprintf( 'Dancer␣reached␣the␣mirror␣in␣%d␣steps', i) )
```

Does this loop terminate? If so, how many iterations does it take?

Chapter 2
Polynomials and Series

Abstract This chapter introduces the reader to the numerical aspects of polynomials. In particular, we examine different polynomial bases such as the monomial, the Chebyshev, and the Lagrange basis; we provide *algorithms to evaluate polynomials* in many of those bases and examine the *different condition numbers in different bases*. We give a first look at the important problem of numerically finding *zeros* and *pseudozeros* of polynomials. We give an algorithmic overview of the *numerical computation of truncated power series* including Taylor series. Finally, we give a brief discussion of *asymptotics*. ◁

Computation with polynomials is one of the pillars on which numerical analysis stands. This book makes extensive use of polynomials, as do all numerical analysis texts, but it takes advantage of several recent theoretical and practical advances in this foundational discipline. It is perhaps somewhat surprising that there were advances to be made in so venerable and well-studied an area, but there were, and almost certainly there still are. This chapter introduces our notations, reviews the basic ideas of the theory and practice of univariate polynomial computation, and gives several facts and algorithms. Some of these algorithms and theorems may be surprising even to people who have some numerical analysis background, and so we recommend that everyone at least skim this chapter, for notation if nothing else.

The related topic of series algebra is also one of the pillars of numerical analysis; indeed, numerical analysis has often been dubbed nothing but "a huge collection of applications of Taylor's theorem." We believe that it isn't quite true (even when the assertion is modified to include "—and, of course, linear algebra"). More properly, the theory of Taylor series provides an interesting and common way of generating polynomial approximations to functions. While Taylor series are of more than just marginal value in this book, they aren't central; but they are useful, and so a section on how to compute them (which will most likely differ from the way the reader was taught to compute them, in their first-year calculus class!) is included.

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_2,

2.1 Polynomials, Their Bases, and Their Roots

Let us begin with a definition of the main object of this chapter.

Definition 2.1 (Polynomial). A polynomial is a function $f : \mathbb{C} \to \mathbb{C}$ such that, for some nonnegative integer n and for some $a_k \in \mathbb{C}$, $0 \le k \le n$, with $a_n \ne 0$,

$$f(z) = \sum_{k=0}^{n} a_k z^k \tag{2.1}$$

for all $z \in \mathbb{C}$. The functions $1, z, z^2, \ldots, z^n$ are called *monomials*, and the a_k are called the coefficients of the monomials for $f(z)$. By convention, the identically zero function $f(z) \equiv 0$ is also called a polynomial, and in this case alone there is no n with $a_n \ne 0$. The *degree* of $f(z)$, written $\deg f$ or $\deg_z f$, is the number n of Eq. (2.1). Moreover, by convention, the degree of the identically zero polynomial is $-\infty$. ◁

The set of all polynomials of degree at most n forms a finite-dimensional vector space. As we can see from their definitions, polynomials are linear combinations of $1, z, z^2, z^3, \ldots, z^m$, for $m \le n$. Moreover, the following fact is obtained:

Theorem 2.1. *If a polynomial $p(z)$ is identically zero, that is, if*

$$a_0 + a_1 z + a_2 z^2 + \cdots + a_n z^n \equiv 0\,,$$

then $a_k = 0$ for all k such that $0 \le k \le n$.

The proof is left as Exercise 2.1. As a result, the functions $1, z, z^2, z^3, \ldots, z^n$ are linearly independent in $\mathbb{C}$. Also, the functions $1, z, z^2, z^3, \ldots, z^n$ span the vector space of polynomials of degrees at most n. Consequently, the monomials form an $(n+1)$-dimensional basis. This basis is known as the *monomial basis*.

There are many other possible bases that can be used to represent spaces of polynomials and, as we will see, what basis we use has important consequences in numerical contexts. The most common bases will be discussed in Sect. 2.2. We can define bases generally as follows.

Definition 2.2 (Basis). A *basis* for the space of polynomials of degree at most n is a set of polynomials $\{\phi_k(z)\}_{k=0}^{n}$ that may be written as

$$\begin{bmatrix} \phi_0(z) \\ \phi_1(z) \\ \vdots \\ \phi_n(z) \end{bmatrix} = \mathbf{B} \begin{bmatrix} 1 \\ z \\ \vdots \\ z^n \end{bmatrix} \tag{2.2}$$

for some *nonsingular* $(n+1) \times (n+1)$ matrix $\mathbf{B}$. In this case, $\boldsymbol{\phi}(z)$ will denote the vector $[\phi_0(z), \ldots, \phi_n(z)]^T$ and $\mathbf{z^k}$ will denote the vector $[1, z, \ldots, z^n]^T$, and we will simply write

$$\boldsymbol{\phi}(z) = \mathbf{B}\mathbf{z}^{\mathbf{k}}\,. \tag{2.3}$$

When the degree of each polynomial in the basis is such that $\deg \phi_k(z) = k$, we say that the basis is *degree-graded*. ◁

Moreover, polynomial bases have the properties we expect from bases, most notably uniqueness of representation.

Theorem 2.2. *The coefficients of $f(z)$ in the basis $\{\phi_k(z)\}_{k=0}^{n}$ are unique. That is, if*

$$f(z) = \sum_{k=0}^{n} c_k \phi_k(z) \qquad \text{and} \qquad f(z) = \sum_{k=0}^{n} b_k \phi_k(z) \tag{2.4}$$

for all $z \in \mathbb{C}$, then $c_k = b_k$ for $0 \le k \le n$.

The proof is left as Exercise 2.2.

The role of polynomials in scientific computation is such that we often want to find their roots. Because of that, we now turn to some important facts about roots of polynomials that will be used in what follows.

Definition 2.3 (Root, or Zero). A complex number r is called a *root* (or *zero*) of $f(z)$ if $f(r) = 0$. The *multiplicity* of r is the least number m such that $f^{(m)}(r) \ne 0$. It is guaranteed that $m \le n$ unless $f(z) \equiv 0$. A root is called *simple* if $m = 1$. ◁

One of the most important properties of polynomials is revealed by this theorem, first proved by Gauss in 1797:

Theorem 2.3 (Fundamental theorem of algebra). *If $f(z)$ is a polynomial not equal to a nonzero constant, that is, if $\deg f \ne 0$ (remember that $\deg f = -\infty$ if $f = 0$ identically), then f has a root.*

As Wilkinson (1984) notices,

> [t]he Fundamental Theorem of Algebra asserts that every polynomial equation over the complex field has a root. It is almost beneath such a majestic theorem to mention that in fact it has precisely n roots.

Remark 2.1. The problem of *finding* all roots of a polynomial, and in particular finding multiple roots when the data are ambiguous, is quite difficult[1]; we shall discuss this material later. A good place for the impatient to start some extra reading is Zeng (2004). ◁

We end this subsection with two important theorems that will be used later:

[1] With some definitions of “finding,” it is impossible for generic polynomials $p(z)$ of degree 5 or more. Degree-5 polynomials can be solved using elliptic functions, though, and there are other tricks. Here, by “finding,” we mean finding a good approximation.

Theorem 2.4 (Factor theorem). *If $f(z)$ has ℓ distinct roots r_k, $1 \le k \le \ell$, each with multiplicity m_k (so $n = \sum_{k=1}^{\ell} m_k$), then*

$$f(z) = a_n \prod_{k=1}^{\ell} (z - r_k)^{m_k} \,. \tag{2.5}$$

Theorem 2.5 (Continuity). *(Ostrowski 1940, 1973) The roots of a polynomial are continuous functions of the coefficients a_k (in any fixed basis). Simple roots are continuously differentiable functions of the coefficients.*

2.1.1 Change of Polynomial Bases

One sometimes wants to change a representation of a polynomial p from one basis to another. In other words, given two bases $\{\phi_k(z)\}_{k=0}^{n}$ and $\{\psi_k(z)\}_{k=0}^{n}$, what is the relation between the coefficients a_k and b_k in the expression

$$p(z) = \sum_{k=0}^{n} a_k \phi_k(z) = \sum_{k=0}^{n} b_k \psi_k(z)\,?$$

In theory, the answer straightforwardly follows from the definition of a basis: If we are given a basis $\{\phi_k(z)\}_{k=0}^{n}$, then it can be expressed as the product of a nonsingular matrix $\mathbf{B}$ and the vector of monomials $\mathbf{z^k}$. The same is true of another basis $\{\psi_k(z)\}_{k=0}^{n}$. Thus, if we let $\boldsymbol{\phi}(z) = \mathbf{B}_1\mathbf{z^k}$ and $\boldsymbol{\psi}(z) = \mathbf{B}_2\mathbf{z^k}$, the relation between the bases is given by

$$\boldsymbol{\phi}(z) = \mathbf{B}_1\mathbf{B}_2^{-1}\boldsymbol{\psi}(z)\,. \tag{2.6}$$

If we let $\boldsymbol{\Phi} = \mathbf{B}_1\mathbf{B}_2^{-1}$ denote the *change-of-basis matrix*,[2] we see that change of basis is the following simple linear transformation:

$$\begin{bmatrix} \phi_0(z) \\ \phi_1(z) \\ \vdots \\ \phi_n(z) \end{bmatrix} = \begin{bmatrix} \phi_{00} & \phi_{01} & \cdots & \phi_{0n} \\ \phi_{10} & \phi_{11} & \cdots & \phi_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{n0} & \phi_{n1} & \cdots & \phi_{nn} \end{bmatrix} \begin{bmatrix} \psi_0(z) \\ \psi_1(z) \\ \vdots \\ \psi_n(z) \end{bmatrix} . \tag{2.7}$$

When the basis is degree-graded, the change-of-basis matrix is triangular.

Finally, observe that the relation between the coefficients of the polynomial bases is as follows. Since

[2] Note that, depending on the author and conventions being used, $\boldsymbol{\Phi}$ or its transpose may refer to the change-of-basis matrix.

$$p(z) = \begin{bmatrix} b_0 \, b_1 \, \cdots \, b_n \end{bmatrix} \begin{bmatrix} \psi_0(z) \\ \psi_1(z) \\ \vdots \\ \psi_n(z) \end{bmatrix} = \begin{bmatrix} a_0 \, a_1 \, \cdots \, a_n \end{bmatrix} \begin{bmatrix} \phi_0(z) \\ \phi_1(z) \\ \vdots \\ \phi_n(z) \end{bmatrix}$$

$$= \begin{bmatrix} a_0 \, a_1 \, \cdots \, a_n \end{bmatrix} \begin{bmatrix} \phi_{00} & \phi_{01} & \cdots & \phi_{0n} \\ \phi_{10} & \phi_{11} & \cdots & \phi_{1n} \\ \vdots & \vdots & \ddots & \\ \phi_{n0} & \phi_{n1} & \cdots & \phi_{nn} \end{bmatrix} \begin{bmatrix} \psi_0(z) \\ \psi_1(z) \\ \vdots \\ \psi_n(z) \end{bmatrix} , \tag{2.8}$$

the relation between the coefficients of $p(z)$ in the bases $\{\phi_k(z)\}_{k=0}^{n}$ and $\{\psi_k(z)\}_{k=0}^{n}$ is given by

$$\begin{bmatrix} b_0 \, b_1 \, \cdots \, b_n \end{bmatrix} = \begin{bmatrix} a_0 \, a_1 \, \cdots \, a_n \end{bmatrix} \boldsymbol{\Phi} \, . \tag{2.9}$$

Thus, the same matrix $\boldsymbol{\Phi}$ relates the bases vectors $\boldsymbol{\phi}$ and $\boldsymbol{\psi}$ and their coefficients.

Remark 2.2. Changing the expression of a polynomial from one basis to another is a *mathematically* valid operation, but we remark right now that it is not always (or even often) a good thing to do *numerically*. This is why Wilkinson (1959a) claims that if

> the explicit polynomial [in monomial basis] has been derived by expanding some other expression, then we may well question the wisdom of this step.

As we will see in Sect. 8.6, changing polynomial bases can amplify numerical errors dramatically: even in the normwise sense, error bounds can grow exponentially with the degree of the polynomial, and componentwise the relative errors can be infinitely larger in one basis than in another. Changing the basis must be done with caution, if at all. ◁

2.1.2 Operations on Polynomials

The following operations can be performed in any polynomial basis. To begin with, the sum of two polynomials (say f and g, of degrees n and m) is a polynomial (just add the coefficients), the negation of a polynomial is a polynomial (just negate the coefficients), and the product of two polynomials is again a polynomial (in this case, the coefficients of the product are bilinear functions of the coefficients of the multiplicands, and the particular function depends on the basis, as we will see).

Polynomial division is a bit more complicated, but not that much. If $f(z) = Q(z)g(z) + R(z)$ and $\deg R < \deg g$, we say that $R(z)$ is the *remainder* on division of $f(z)$ by $g(z)$; if $R(z)$ is identically zero, then we say that $g(z)$ *divides* (or divides evenly into) $f(z)$. This cannot happen if $g(z)$ is identically zero. If g does divide f, then we write $g \mid f$ (which is read as "g divides f"). The polynomial $Q(z)$ in $f = Qg + R$ is called the *quotient*. It is easy to prove that, given $f(z)$ and $g(z)$, the quotient and remainder are unique.[3] Polynomial division is merely mentioned in this book, but is occasionally needed in applications. Again the details of the division process depend on the basis being used, but note that it amounts to solving a linear system of equations for the unknown coefficients of $Q(z)$ and $R(z)$, once the bilinear functions of multiplication in that basis are known.

We also occasionally need the notion of *relatively prime polynomials*, and for that we need the notion of greatest common divisor, or GCD. A polynomial $d(z)$ is a common divisor of f and g if both $d \mid f$ and $d \mid g$. If d has the maximum possible degree of all common divisors of f and g, we say that it is a GCD of f and g. Every constant multiple of a common divisor is a common divisor, and so GCDs are unique only up to multiplication by a constant.

The *composition* $f(g(z))$ is also a polynomial, of degree nm. It is sometimes worthwhile to seek to rewrite a large polynomial $F(z)$ as a composition $F(z) = f(g(z))$; finding such f and g is called polynomial decomposition. We will not pursue this further in this book, but it also finds use in some applications.

2.2 Examples of Polynomial Bases

Several polynomial bases are commonly encountered in applications. We have already encountered the monomial basis, and we will soon see why it should sometimes be avoided in applications. Before that, we examine some of the most common bases that arise, and indicate some of their advantages and disadvantages.

2.2.1 Shifted Monomials

Shifted monomials (shifted by a constant $a \in \mathbb{C}$) are polynomials having the form

$$\phi_k(z) = (z-a)^k\,, \tag{2.10}$$

and the set $\{(z-a)^k\}_{k=0}^n$ forms a basis. The expansion of a polynomial $f(z)$ in this basis is just its Taylor series:

$$f(z) = f(a) + f'(a)(z-a) + \cdots + \frac{f^{(n)}(a)}{n!}(z-a)^n\,. \tag{2.11}$$

If $a = 0$, this is just the standard monomial basis, also called the power basis. The change-of-basis matrix from the monomials to the shifted monomials is simple. For $n = 3$, this is

[3] This is more generally true than we need here: the coefficients of our polynomials are complex numbers or real numbers and this statement is true for more general objects as well.

$$\begin{bmatrix}\phi_0(z)\\ \phi_1(z)\\ \phi_2(z)\\ \phi_3(z)\end{bmatrix} = \begin{bmatrix}1\\ z-a\\ (z-a)^2\\ (z-a)^3\end{bmatrix} = \begin{bmatrix}1 & 0 & 0 & 0\\ -a & 1 & 0 & 0\\ a^2 & -2a & 1 & 0\\ -a^3 & 3a^2 & -3a & 1\end{bmatrix}\begin{bmatrix}1\\ z\\ z^2\\ z^3\end{bmatrix}. \qquad (2.12)$$

The change-of-basis matrix that goes from the shifted monomials to the monomials is just the inverse of this matrix (and that it exists and is nonsingular for any a means that the shifted monomials are indeed a basis).

Remark 2.3. Multiplication of polynomials expressed in the monomial basis is a familiar operation. Multiplication of two polynomials expressed in an arbitrary (but common to the two polynomials) shifted monomial basis may be done by embedding them in Taylor series and using the methods of Sect. 2.6. This can also be done rapidly by use of the fast Fourier transform (FFT) (see Chap. 9). ◁

As we have seen in Chap. 1, it is important to compute sums in a stable and predictable way when we use computer arithmetic. For polynomials expressed in the shifted monomial basis, we can use *Horner's method*, which can be written as

$$f(z) = f(a) + (z-a)\left(f'(a) + (z-a)\left(\frac{f''(a)}{2}\right.\right.$$
$$\left.\left. + (z-a)\left(\cdots + (z-a)\left(\frac{f^{(n)}(a)}{n!}\right)\cdots\right)\right)\right). \qquad (2.13)$$

The key difference with the use of Eq. (2.11) is that we associate terms in a way that does not require us to compute higher powers of $z-a$. In addition, as in this formula, it is generally preferable to include the factorials in the Taylor coefficients. For a polynomial of degree n, this formula requires $O(n)$ flops, where explicitly forming each terms $(z-a)^k$ requires more.

Assuming that the coefficients of f in this basis are stored in a vector `c` indexed from 1 to $n+1$, so that `c(1)` $= f(a)$, `c(2)` $= f'(a)$, `c(3)` $= f^{(2)}(a)/2!$, and so on, one can use a simple MATLAB program to carry out the computation of $f(z)$ based on Horner's method:

```
p = c(n+1)*ones(size(z));
za = z - a;
for i=n:-1:1,
    p = za.*p + c(i);
end;
```

Note that, in this code, the coefficient of the power-n term is the last component of the vector of coefficients, as opposed to other commands such as MATLAB's built-in command `polyval`, where the order is reversed. Because `polyval` can be simply adapted to use a shifted monomial basis, we show how to use it in an example.

Example 2.1. A monomial basis polynomial is entered as a vector of coefficients (in decreasing order of exponent, and zero coefficients must be explicitly included).

Consider the polynomial $p(z) = z^4 - 4z^3 + 3z^2 - 2z + 5$ on, say, $0 \le z \le 2$.[4] Thus, we simply need to execute

```
p = [ 1, -4, 3, -2, 5 ];
z = linspace( 0, 2, 101 );
pz = polyval( p, z );
plot( z, pz, 'k' )
```

This code generates a graph in which we see, by eye, a zero of $p(z)$ near $z = 1.5$. ◁

Instead of just evaluating a polynomial, one can change a polynomial from the monomial basis to a shifted monomial basis (that is, a Taylor series) by using an extension of Horner's method called *synthetic division*. This method, which is widely discussed in the literature, is described by Algorithm 2.1. We will use this algorithm occasionally, and so we will discuss its accuracy later, in Sect. 2.2.1.2.

Algorithm 2.1 Synthetic division of a polynomial $f(z) = \sum_{j=0}^{n} c_j (z-a)^j$ expressed in a shifted monomial basis, evaluating $f(z)$ and its first k derivatives at $z = b$, returning $f_k = {f^{(k)}(b)}/{k!}$

```
Require: The expansion point $a \in \mathbb{C}$, a vector of monomial coefficients $\mathbf{c} \in \mathbb{C}^{n+1}$ (indexed from 0 to $n$) such that $f(z) = \sum_{j=0}^{n} c_j (z-a)^j$, a new expansion point $b$ and a desired number $k \ge 0$ of Taylor coefficients of $f(z)$ at $z = b$.
  $f_0 := c_n$
  $f_{(1:k)} := 0$
  for $j$=$n-1$:$-1$:0 do
    for $i$=min($k, n-j$):$-1$:1 do
      $f_i = (b-a) f_i + f_{i-1}$
    end for
    $f_0 = (b-a) f_0 + f_j$
  end for
  return The $(k+1)$-vector $\mathbf{f}$ such that $f(z) = \sum_{j=0}^{k} f_j (z-b)^j + O(z-b)^{k+1}$. That is, $f_j = {f^{(j)}(b)}/{j!}$.
```

Example 2.2. Consider Example 2.1 again, and let us expand this polynomial about $z = 1.5$, where we saw our approximate zero. We used MAPLE and its `series` command to effect Algorithm 2.1, and thus found that

$$p(z) = 0.3125 - 6.5(z-1.5) - 1.5(z-1.5)^2 + 2(z-1.5)^3 + (z-1.5)^4 \,. \qquad (2.14)$$

This is a new expression for the polynomial, this time expanded about $z = 1.5$. ◁

2.2.1.1 Newton's Method for Polynomials

As an aside, we briefly introduce Newton's method for finding zeros of polynomials. This will be taken up in greater detail and generality in the next chapter. Newton

[4] This example is drawn from Henrici (1964). In Exercise 2.5, you will be asked to consider instead Newton's example, $p(z) = z^3 - 2z - 5$.

suggested that we could use the first two coefficients of the shifted polynomial as a linear approximation to the polynomial and could be used in an attempt to find a root: In this example, setting the linear approximation $0.3125-6.5\,(z-1.5)=0$ yields $z-1.5={}^{0.3125}/_{6.5}\approx 0.048076923$, suggesting that we should shift our expansion point again, this time to $z=1.5+0.048076923\approx 1.548$. When we do so, we find

$$p(z)=-0.002730-6.630\,(z-1.548)-1.198\,(z-1.548)^2 \\ +2.192\,(z-1.548)^3+1.0\,(z-1.548)^4$$

and since now $p(1.548)$ is smaller than before, we begin to see that the process might work in an iterative fashion.

In general, suppose that we have expanded the polynomial about $z=r_k$, where r_k is our current approximation of the root:

$$p(z)=p(r_k)+p'(r_k)(z-r_k)+O(z-r_k)^2\,. \tag{2.15}$$

Using Newton's idea, we solve this linear approximation (which is possible if $p'(r_k)\neq 0$) to find

$$z\doteq r_k-\frac{p(r_k)}{p'(r_k)}, \tag{2.16}$$

and it makes sense to name this approximation r_{k+1}:

$$r_{k+1}=r_k-\frac{p(r_k)}{p'(r_k)}\,. \tag{2.17}$$

This is, of course, Newton's method, which we will take up further in Chap. 3. For now, note two things: First, each approximation r_k is the *exact* root of

$$p(z)-p(r_k)=0\,, \tag{2.18}$$

and so if $p(r_k)$ (which we call the *residual*) is small, then we have found the exact solution of a nearby polynomial, and, second, we have found that this process is apt to fail near multiple roots because if both $p(z^*)$ and $p'(z^*)=0$, then since $r_k\to z^*$, both $p(r_k)\to 0$ and $p'(r_k)\to 0$, making the solving step problematic.

Continuing our example[5] just one more iteration, with $r_k=1.548$ in Eq. (2.15), we have $r_{k+1}=1.548-\frac{(-0.0027295)}{(-6.62973)}$, that is, $r_{k+1}\approx 1.5475883$. Shifting to the basis centered here using synthetic division, we have

$$p(z)=-0.00000024948774-6.6287459395492\,(z-1.5475883)-\cdots\,, \tag{2.19}$$

[5] If you try to reproduce these computations, your results may differ because we were somewhat cavalier in rounding intermediate results. Keeping all figures—as one should—will make the results slightly different.

and we notice that 1.5475883 is an exact root of the nearby polynomial $p(z) + 2.494877\ldots \times 10^{-7}$.

2.2.1.2 Errors in Synthetic Division

We refer to Higham (2002) for a complete accuracy analysis of synthetic division, but we state a result here connecting rounding errors and the forward error via a condition number. Let $B^{(j)}(z)$ be defined as

$$B^{(j)}(z) := \sum_{k=j}^{n} k^{\underline{j}} \left| \frac{f^{(k)}(a)}{k!} \right| \left| (z-a)^{k-j} \right| , \tag{2.20}$$

where $k^{\underline{j}}$, read as "k to the j falling," is defined as

$$k^{\underline{j}} = \frac{k!}{(k-j)!} = k(k-1)(k-2)\cdots(k-j+1)$$

(see Graham et al. 1994). Then the difference between the reference value of the derivative $f^{(j)}(\alpha)$ and the value computed by synthetic division, say $\hat{r}_j$, is bounded by

$$\left| f^{(j)}(\alpha) - \hat{r}_j \right| \le O(n\mu_M) B^{(j)}(\alpha) + O(\mu_M^2) \,. \tag{2.21}$$

We will see later in this chapter many more examples of such $B(z)$ functions, which are called *condition numbers for evaluation of polynomials*. In some sense, the above theorem, which (to first order) bounds the *forward error* $|f^{(j)} - \hat{r}_j|$ by the product of a condition number and a backward error (here $O(n\mu_M)$), is as important to numerical analysis as $F = ma$ is to physics.

Changing from bases other than the monomial basis to shifted monomials is sometimes useful (again, numerically this has to be done with caution, as we will see). We pursue this in the exercises.

2.2.2 *The Newton Basis*

We have seen that the shifted monomial basis is defined in reference to a given data point a. Similarly, the Newton basis is defined in reference to a set of points, which we call *nodes*. The Newton basis on the $n+1$ nodes τ_0, τ_1, τ_2, …, τ_n is given by

$$\{\phi_k(z)\}_{k=0}^{n} = \{1, z-\tau_0, (z-\tau_0)(z-\tau_1), \ldots, (z-\tau_0)(z-\tau_{\cdot})\cdots(z-\tau_{n-1})\} \,,$$

or, more compactly,

$$\{\phi_k(z)\}_{k=0}^{n} = \left\{ \prod_{i=0}^{k-1} (z - \tau_i) \right\}_{k=0}^{n} . \qquad (2.22)$$

Note that, by convention, if $m > n$, a product $\prod_{i=m}^{n}$ is just 1. Note also and especially that one node, namely, τ_n, is omitted from any mention in this basis. We remark that this permits choice: One speaks of “a” Newton basis, not of “the” Newton basis. There is a further choice involved, namely, the ordering of the nodes; once one of the $n+1$ nodes has been omitted, there is a further $n!$ different orderings possible if the nodes are distinct. Some of them are numerically better than others, as we will see.

Newton bases are typically used with what are called *divided differences* (see Problem 8.13). In fact, de Boor (2005) defines divided differences as the coefficients of $f(z)$ expressed in a Newton basis. Though divided differences and Newton bases have a rich theory and practice, they will only rarely be used in this book because there are better choices available. Trefethen (2013 p. 33) takes a similar stance:

> Many textbooks claim that it is important to use this approach for reasons of numerical stability, but this is not true, and we shall not discuss the Newton approach here.

They are the preferred basis in de Boor (1978), because they are convenient, inexpensive, and, for low degrees, accurate. However, as we will see, the barycentric Lagrange basis that we prefer is much better conditioned for larger degrees on good sets of nodes. After introducing the Lagrange basis, we will return to this point.

2.2.3 Chebyshev Polynomials

The Chebyshev polynomials can be defined by

$$\phi_k(z) = T_k(z) = \cos(k \cos^{-1} z) \qquad (2.23)$$

for $k = 0, 1, 2, \ldots$. It is easy to see that, for $k = 0$ and $k = 1$, these are indeed polynomials:

$$T_0(z) = \cos 0 = 1 \qquad (2.24)$$

$$T_1(z) = \cos\cos^{-1} z = z . \qquad (2.25)$$

Moreover, by applying the angle sum and angle difference formulæ for cosines to $\cos((k+1)\cos^{-1} z)$ and $\cos((k-1)\cos^{-1} z)$, it follows that, for $k > 1$,

$$T_{k+1}(z) = 2zT_k(z) - T_{k-1}(z) . \qquad (2.26)$$

Hence, all $\phi_k(z)$ are polynomials. Figure 2.1 displays the first nine Chebyshev polynomials.

A well-known algorithm to compute the values of polynomials expressed in this basis is provided in Rivlin (1990 156–158). It turns out that this algorithm is called

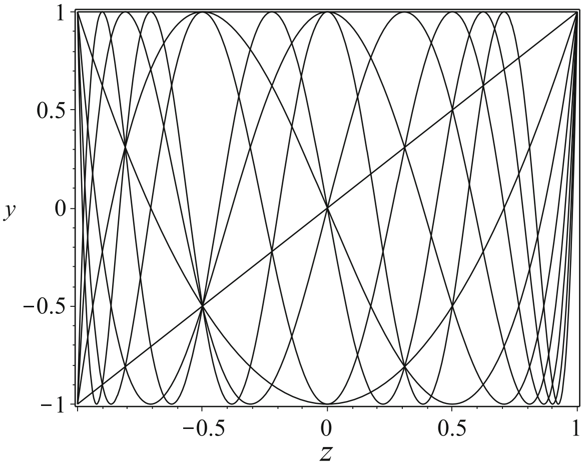


Fig. 2.1 The first nine Chebyshev polynomials $T_0(z) = 1$, $T_1(z) = z$, and $T_{n+1}(z) = 2zT_n(z) - T_{n-1}(z)$. See Exercise 2.33

the Clenshaw algorithm, which we take up after mentioning other polynomials belonging to an important class to which Chebyshev polynomials belong, namely, *orthogonal polynomials*. In the real case, Chebyshev polynomials can be shown to be orthogonal with respect to the inner product

$$\langle f, g\rangle := \int_{-1}^{1} \frac{f(x)g(x)}{\sqrt{1-x^2}}\,dx\,. \tag{2.27}$$

In the complex case, they are also orthogonal. The zeros of $T_n(z)$ are

$$\xi_k^{(n)} := \cos\left(\frac{\pi(k-1/2)}{n}\right) \tag{2.28}$$

for $k = 1, 2, \ldots, n$. The proof is left as Exercise 2.4. Chebyshev polynomials are *also* orthogonal with respect to the following *discrete* inner product:

$$\langle f, g\rangle := \sum_{j=1}^{n} f(\xi_j^{(n)})g(\xi_j^{(n)})\,. \tag{2.29}$$

See Rivlin (1990 Exercise 1.5.26, p. 53) for a complete enumeration of all cases of $\langle T_k, T_p\rangle$. You will be asked to prove in Exercise 2.6 that this discrete orthogonality relation allows easy computation of the coefficients of the expansion of a degree-$(n-1)$ polynomial $p(z)$ if you can evaluate it on the zeros of $T_n(z)$:

$$p(z) = \frac{A_0}{2}T_0(z) + A_1T_1(z) + \cdots + A_{n-1}T_{n-1}(z), \tag{2.30}$$

where[6]

$$A_m = \frac{2}{n}\sum_{j=1}^{n} p(\xi_j^{(n)})T_m(\xi_j^{(n)})\,, \tag{2.31}$$

for $m = 0,\, 1,\, \ldots,\, n-1$, can be computed with $O(n^2)$ floating-point operations.

Another and in some sense more interesting set of discrete points is called the *Chebyshev–Lobatto* points or *Chebyshev extreme* points, which are the places where $T_n(z)$ achieves its maximum and minimum values on $-1 \le z \le 1$. The endpoints are special, and are always included, because $T_n(1) = 1$ and $T_n(-1) = (-1)^n$, as you may easily prove by the definition $T_n(z) = \cos(n\cos^{-1} z)$. The interior (relative) extrema are the zeros of $T_n'(z)$ and there can be at most $n-1$ of them. Since $\cos(n\theta) = \pm 1$ at these extrema, we can verify that

$$\eta_k = \eta_k^{(n)} = \cos\frac{k\pi}{n} \tag{2.32}$$

for $k = 0,\, 1,\, \ldots,\, n$. This gives $n+1$ extrema on the interval, including the endpoints with $k = 0$ and $k = n$, and thus it must include all possible extrema. Note that both ξ_k and η_k run "backward" across the interval, which is sometimes inconvenient but only trivially so.

Chebyshev polynomials have a large collection of interesting and useful properties, some of which will be discussed when they come up naturally in the book. Chebyshev polynomials are the favorite of many numerical analysts. In particular, the Chebfun package is founded on the properties of Chebyshev polynomials (it uses the η_k, not the ξ_k). We will see several examples of its use in this book. Chebfun uses the syntax `chebpoly(n)` to pick out a Chebyshev polynomial. See Exercise 2.7.

2.2.4 *Other Orthogonal Polynomials*

There are a great many other examples of orthogonal polynomials. The orthogonal polynomials implemented in MAPLE include the Chebyshev polynomials, where the name `ChebyshevT` is used, with the syntax `ChebyshevT(n,z)`. Other orthogonal polynomials implemented include the Gegenbauer polynomials (`GegenbauerC`), the Hermite orthogonal polynomials[7] (`HermiteH`), and the Jacobi polynomials (`JacobiP(n,a,b,z)`). The latter include as a special case (`JacobiP(n,0,0,z)`), more usually called the *Legendre* polynomials; these will be used in Chap. 10 for Gaussian quadrature. Maple has another package for the ma-

[6] Note that we use $A_0/2$ in equation (2.30) so that formula (2.31) can be the same for $m = 0, 1, 2, \ldots$

[7] They will almost never be used in this book and are not to be confused with the Hermite interpolational basis polynomials, which will be used.

nipulation of orthogonal series, namely, the `OrthogonalSeries` package, which is quite extensive.

A common characteristic of orthogonal polynomials is that they generally satisfy a three-term recurrence relation for $n \geq 2$, which we write here as

$$\alpha_{n-1}\phi_n(z) = (z - \beta_{n-1})\phi_{n-1}(z) - \gamma_{n-1}\phi_{n-2}(z)\,. \tag{2.33}$$

As we saw above, the recurrence for the Chebyshev polynomials has $\alpha_{n-1} = \gamma_{n-1} = {}^1\!/\!{}_2$ and $\beta_{n-1} = 0$ for all n. However, for other classes of polynomials, there is a dependence on n. For instance, the recurrence relation for the Jacobi polynomials starts with $P_0(z) = 1$, $P_1(z) = {}^{(a-b)}\!/\!{}_2 + (1 + {}^{(a+b)}\!/\!{}_2)z$, and thereafter

$$\begin{aligned}
\alpha_{n-1} &= \frac{2n\,(n+a+b)}{(2n+a+b-1)\,(2n+a+b)} \\
\beta_{n-1} &= \frac{(b-a)\,(a+b)}{(2n+a+b-2)\,(2n+a+b)} \\
\gamma_{n-1} &= \frac{2\,(n+a-1)\,(n+b-1)}{(2n+a+b-1)\,(2n+a+b-2)}\,.
\end{aligned} \tag{2.34}$$

In the special case $a = b = 0$, for the Legendre polynomials, we have $P_0(z) = 1$, $P_1(z) = z$, and

$$\alpha_{n-1} = \frac{n}{2n-1} \qquad \beta_{n-1} = 0 \qquad \text{and} \qquad \gamma_{n-1} = \frac{n-1}{2n-1}\,. \tag{2.35}$$

The first 10 Legendre polynomials are plotted in Fig. 10.4 of Chap. 10.

Recent uses in mathematical handwriting recognition of generalizations of orthogonal polynomials—namely, the Legendre–Sobolev polynomials, which include derivatives in their inner product—can be seen in Golubitsky and Watt (2009) and in Golubitsky and Watt (2010). See Exercise 4.28.

2.2.5 The Clenshaw Algorithm for Evaluating Polynomials Expressed in Orthogonal Bases

The Clenshaw algorithm generalizes the idea used in Horner's method to certain orthogonal polynomial bases. If the elements of the polynomial basis $\phi_k(z)$ are related by a three-term recurrence relation

$$\phi_k(z) = \alpha_k(z)\phi_{k-1}(z) - \beta_k\phi_{k-2}(z) \tag{2.36}$$

(the notation has changed from the previous section, to match the paper Smoktunowicz (2002), which we reference ahead) and $\phi_0(z) = 1$ and $\phi_1(z) = \alpha_1(z)$, where, for all the examples we are concerned with, the $\alpha_k(z)$ are linear polynomials in z and the β_k are constants, then a polynomial $p(z)$ expressed in this orthogonal basis as

$$p(z) = \sum_{k=0}^{n} b_k \phi_k(z) \qquad (2.37)$$

can be evaluated in $O(n)$ flops by the *Clenshaw algorithm*, as follows.

Algorithm 2.2 The Clenshaw algorithm

```
Require: A value $z$, a nonnegative integer $n$, and a sequence $b_0, b_1, \ldots, b_n$ of coefficients
Require: The functions $\alpha_k(z)$ and the constants $\beta_k$
  $y_{n+1} := 0$
  $y_n := b_n$
  for $k$ from $n-1$ by $-1$ to 1 do
    $y_k := b_k + \alpha_{k+1}(z)y_{k+1} - \beta_{k+2}y_{k+2}$
  end for
  $p := (y_0 - \beta_2 y_2)\phi_0(z) + y_1\phi_1(z)$
  return The value of $p(z) = \sum_{k=0}^{n} b_k\phi_k(z)$
```

To see that this algorithm is correct, note that a *loop invariant* for the algorithm is the sum

$$p(z) = -\beta_{k+1}y_{k+1}\phi_{k-1}(z) + y_k\phi_k(z) + \sum_{j=0}^{k-1} b_j\phi_j(z)\,. \qquad (2.38)$$

That is, before the start of the loop i.e. when $k = n$, this statement is trivially true because $y_{k+1} = 0$, and the update step changes the value of what will be the next y_k and replaces $y_{k+1}\phi_{k+1}(z)$ with $y_{k+1}(\phi_{k+1}(z) - \alpha_{k+1}\phi_k(z))$ or $-\beta_{k+1}y_{k+1}\phi_{k-1}(z)$. The process finishes when there are only two terms left, which sum to $p(z) = (b_0 - \beta_2 y_2)\phi_0(z) + y_1\phi_1(z)$ by the loop invariant.

Now, let us address the numerical stability of this method for the evaluation of polynomials:

Theorem 2.6 (Backward Stability of the Clenshaw Algorithm). *Under natural assumptions, evaluation of this algorithm is backward stable: that is, for a given z, the algorithm gives the exact evaluation of $p + \Delta p$, where the coefficients of $p + \Delta p$ are only slightly perturbed: $b_k + \Delta b_k$, where, with a modestly growing function L of n,*

$$|\Delta b_k| \le \mu_M L |b_k| \qquad (2.39)$$

in the best scenario (this holds only for some bases and polynomials with nonincreasing coefficients b_k), and

$$\|\Delta b\|_\infty \le \mu_M L \|b\|_\infty \qquad (2.40)$$

in the usual case.

In particular, for the Chebyshev polynomials, we have $L = O(n^2)$ in Eq. (2.39), showing that the Chebyshev basis evaluated by the Clenshaw algorithm has excellent backward-stability properties. The proof of this theorem is given in Smoktunowicz (2002).

2.2.6 Lagrange Polynomials

We now look at a very important *non*orthogonal basis family, the Lagrange bases. These are different to the previously discussed examples in that they are not *degree-graded*: Each element of a particular Lagrange basis has full degree, here n. Given $n+1$ distinct nodes τ_k, $0 \le k \le n$, define the numbers β_k by the partial fraction expansion

$$\frac{1}{w(z)} = \frac{1}{\prod_{k=0}^{n}(z-\tau_k)} = \sum_{k=0}^{n} \frac{\beta_k}{z-\tau_k}. \tag{2.41}$$

Then, solving for the numbers β_k, we obtain

$$\beta_k = \prod_{\substack{j=0\\ j\neq k}}^{n} (\tau_k - \tau_j)^{-1}. \tag{2.42}$$

Definition 2.4 (Lagrange polynomials). Given a set of nodes $\{\tau_k\}_{k=0}^{n}$ and the resulting numbers β_k,

$$\phi_k(z) = L_k(z) = \beta_k \prod_{\substack{j=0\\ j\neq k}}^{n} (z-\tau_j) \tag{2.43}$$

is the kth Lagrange polynomial. ◁

Note that, using the Kronecker delta, we can write

$$L_k(\tau_j) = \delta_j^k = \begin{cases} 0 & j \neq k \\ 1 & j = k \end{cases}, \tag{2.44}$$

and so for any polynomial $f(z)$ of degree at most n,

$$f(z) = \sum_{j=0}^{n} f(\tau_j) L_j(z). \tag{2.45}$$

Theorem 2.7. *The set of polynomials $L_j(z)$, for $0 \le j \le n$, forms a basis if the nodes τ_k, $0 \le k \le n$, are distinct.*

Proof. The theorem is equivalent (by definition) to the statement that the change-of-basis matrix $\mathbf{A}$ in $[L_0(z), L_1(z), \ldots, L_n(z)] = [1, z, z^2, \ldots, z^n]\mathbf{A}$ is nonsingular. That in turn is equivalent to the statement that the change-of-basis matrix in the other direction $[1, z, z^2, \ldots, z^n] = \mathbf{LB}$ is nonsingular, and this is easier. By the above formula, the entries of $\mathbf{B}$ are $B_{k,j} = \tau_k^j$. It is an (interesting) exercise to show that $\det\mathbf{B} = \prod_{j>k}(\tau_j - \tau_k)$, which is nonzero when the nodes are distinct. See Exercise 4.14. ♮

Corollary 2.1. *The only polynomial of degree at most n that satisfies $f(\tau_i) = 0$ for $n+1$ distinct nodes τ_i, $0 \le i \le n$, is the identically zero polynomial.*

Proof. Since the $L_j(z)$ form a basis, we may express an arbitrary polynomial of degree at most n as $\sum_{j=0}^{n} a_j L_j(z)$. Evaluating this polynomial at each of τ_k in turn gives $a_k = 0$, uniquely resulting in the identically zero polynomial. ♮

Remark 2.4. This corollary is part of the normal proof that interpolants are unique; we here see, doing things in a different order, that it is a corollary of the theorem we proved directly above. That is, this proof is done in a different order than usual but is equivalent. ◁

We will see shortly another notation for Eq. (2.45): With $\rho_i := f(\tau_i)$,

$$f(z) = \sum_{i=0}^{n} \rho_i L_j(z) = \sum_{i=0}^{n} \rho_i \frac{w(z)}{z-\tau_i}\beta_i = w(z)\sum_{i=0}^{n} \frac{\rho_i \beta_i}{z-\tau_i} \,. \tag{2.46}$$

This is the *first barycentric form* of a polynomial expressed in the Lagrange basis. We will see the second barycentric form in the exercises in this chapter and again in Chap. 8. The coefficients in the expansion of $f(z)$ in the Lagrange basis on $\tau_0, \ldots, \tau_n$ are simply the values $f(\tau_j)$.

The Lagrange polynomials are wonderfully useful, and we will use them every chance we get. An algorithm to compute polynomials in this basis is provided by Berrut and Trefethen (2004) (see Algorithm 2.3).

Algorithm 2.3 First barycentric form for evaluation of a Lagrange interpolating polynomial

```
Require: A value $z$, an integer $n > 0$, a vector of coefficients $\rho_k$, a vector of nodes $\tau_k$, and a
  precomputed vector of barycentric weights $\beta_k$
  if $z$ is identical to any $\tau_k$ then
    return $\rho_k$
  end if
  $w = 1$
  for $j$=0:$n$ do
    $w = w \cdot (z - \tau_j)$
  end for
  $p = 0$
  for j=0:n do
    $p = p + \beta_j \rho_j / (z - \tau_j)$
  end for
  return $w \cdot p$
```

2.2.6.1 Numerical Stability of the Barycentric Form

The numerical stability for Algorithm 2.3 is interesting. The paper (Higham 2004) shows that evaluation of this (first) barycentric form is nearly perfectly backward stable: The computed $\hat{p}(z)$ satisfies

$$\hat{p}(z) = \prod_{j=0}^{n}(z-\tau_j)\sum_{j=0}^{n}\frac{\beta_j\hat{\rho}_j}{z-\tau_j} = w(z)\sum_{j=0}^{n}\frac{\beta_j\hat{\rho}_j}{z-\tau_j}\,, \tag{2.47}$$

where each perturbed $\hat{\rho}_j$ satisfies

$$\hat{\rho}_j = \rho_j(1+\delta_j), \tag{2.48}$$

such that $|\delta_j| < \gamma_{5(n+1)}$. That is, provided n isn't so large that it is $O(1/\mu_M)$, the computed sum is the *exact* value of a polynomial passing through only slightly different data values.

Remark 2.5. This result is one of the most important *backward-stability* results presented in this book. What the paper (Higham 2004) provides is a *guarantee* that evaluating the first barycentric form will *always* produce the exact value of a polynomial of the same form as the one we started with, with at most only slightly perturbed data. This result should be compared with the similar result quoted from Smoktunowicz (2002) for orthogonal polynomials, and contrasted with the results of the *forward* error analysis for Horner's method presented in Sect. 2.2.1. ◁

2.2.6.2 Change-of-Basis from a Lagrange Basis

The change-of-basis matrices are deceptively simple *from* a Lagrange basis. We say "deceptively" because the change-of-basis itself may be ill advised because of difficulties related to the conditioning of the matrix, as we will see. However, if it is desired (in spite of misgivings) to perform the change of basis, it is, in theory, simple to carry out. Because any polynomial can be written using Eq. (2.45), each element of a different basis, say $\phi_k(z)$, may be written as

$$\phi_k(z) = \sum_{j=0}^{n}\phi_k(\tau_j)L_j(z)\,. \tag{2.49}$$

That is, the change-of-basis matrix *from* a Lagrange basis is just (in the four-by-four case for simplicity)

$$\begin{bmatrix}\phi_0(z)\\ \phi_1(z)\\ \phi_2(z)\\ \phi_3(z)\end{bmatrix} = \begin{bmatrix}\phi_0(\tau_0) & \phi_0(\tau_1) & \phi_0(\tau_2) & \phi_0(\tau_3)\\ \phi_1(\tau_0) & \phi_1(\tau_1) & \phi_1(\tau_2) & \phi_1(\tau_3)\\ \phi_2(\tau_0) & \phi_2(\tau_1) & \phi_2(\tau_2) & \phi_2(\tau_3)\\ \phi_3(\tau_0) & \phi_3(\tau_1) & \phi_3(\tau_2) & \phi_3(\tau_3)\end{bmatrix}\begin{bmatrix}L_0(z)\\ L_1(z)\\ L_2(z)\\ L_3(z)\end{bmatrix}\,, \tag{2.50}$$

or, more compactly,

$$\boldsymbol{\varphi}(z) = \mathbf{VL}(z)\,. \tag{2.51}$$

In the particular case when $\phi_k(z) = z^k$, $\mathbf{V}$ is called a *Vandermonde* matrix, and it is nonsingular precisely when the nodes τ_k are distinct. For other $\phi_k(z)$, $\mathbf{V}$ is called a *generalized Vandermonde matrix* (see, for example, Problems 8.36 and 8.37).

The Vandermonde matrices occurs often enough that, for emphasis, we will display one here:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ \tau_0 & \tau_1 & \tau_2 & \tau_3 \\ \tau_0^2 & \tau_1^2 & \tau_2^2 & \tau_3^2 \\ \tau_0^3 & \tau_1^3 & \tau_2^3 & \tau_3^3 \end{bmatrix} . \tag{2.52}$$

That matrix can be generated by the MAPLE command

```
V := Matrix( 4, 4,
     shape = Vandermonde[ [tau[0], tau[1], tau[2], tau[3]] ]  );
latex( LinearAlgebra:-Transpose(V) );
```

The convention of needing the transpose to get a "Vandermonde" matrix in this notation agrees with the use in Higham (2002); however, in Chap. 8, we often use the alternative.

In Chap. 8, we will extend the Lagrange basis to the *Hermite interpolational basis*, which allows some of the nodes τ_k to coalesce or "flow together," in which case we talk about *confluency*. This basis is quite distinct from the Hermite orthogonal basis mentioned briefly earlier, and is not to be confused with it. For change of bases to other bases, one talks about *confluent* Vandermonde matrices. This will be taken up later.

Multiplication and division of polynomials expressed in a Lagrange basis are not yet widely encountered in practice[8]; but multiplication is simple enough, provided there are enough data to represent the product (one needs $nm+1$ points if the degrees of the multiplicands are n and m). The entries are simply $f(\tau_i)g(\tau_i)$.[9]

2.2.6.3 The Degree of Difficulty

Given a polynomial expressed in a Lagrange basis, what is its degree? Clearly, if we have enough points to capture the polynomial (say $n+1$), then the degree is *at most* n. But it may very well be less than that, and knowledge of the actual degree can be quite important. We return to this problem in Sect. 11.8, but for now we note a lemma that you are asked to prove in Exercise 2.12.

Lemma 2.1. *If a polynomial $f(z)$ of degree at most n has the values ρ_k on the $n+1$ distinct nodes τ_k, for $0 \le k \le n$, then the degree of $f(z)$ is exactly n if*

$$\sum_{k=0}^{n} \beta_k \rho_k \neq 0 , \tag{2.53}$$

where the β_k are the barycentric weights of the nodes.

[8] We believe this will change, as the realization that working directly in a Lagrange basis is a good idea gradually percolates through the communities.

[9] Division with remainder, on the other hand, requires solving an overspecified linear system, in order to enforce the degree constraints; see Amiraslani (2004).

Proof. Left as Exercise 2.12.

What happens if this is not zero, but small relative to $\|\boldsymbol{\rho}\|$, the norm of the vector of values of $f(z)$ on the nodes? Does this mean that $f(z)$ is "nearly" of lower degree? We do not give a complete answer to this, but rather note only that the generic case with values ρ_i on nodes τ_i is that the degree is exactly n; if the values of a low-degree polynomial are perturbed by arbitrarily small amounts, then almost certainly the perturbed values are the exact values of a degree-n polynomial.

But how close are the given values, then, to the values of a lower-degree polynomial? This question has been addressed (using the lemma above) in Rezvani and Corless (2005), and using the witness vector in Hölder's inequality, we can find an analytic solution in the case the nearby polynomial is of degree *one* less than n, in a manner similar to what we use later in Theorem 2.9. For still-lower degrees, a computational procedure is available. We take a different tack here and give an alternative characterization of the degree of a polynomial.

In exact arithmetic, a polynomial of degree n has exactly n complex roots, counting multiplicity. But numerically, the condition number $B(z)$ grows as $|z|$ grows, so the location of *large* roots is often very sensitive to perturbations. If a polynomial $p(z)$ can be perturbed by a small amount in such a way that some large roots go out to infinity, then the original polynomial is somehow close to a lower-degree polynomial. More precisely, we define the *ε-degree* of a polynomial $p(z) = \sum_{k=0}^{n} c_k \phi_k(z)$ expressed in a basis $\boldsymbol{\phi}$ with weights $w_k \geq 0$ not all zero as

$$\deg_\varepsilon(p) = \min \left\{ \deg(p + \Delta p) \,\middle|\, |\Delta c_k| \leq w_k \varepsilon \right\} . \tag{2.54}$$

In a degree-graded basis, the computation is easy; for the Lagrange and Hermite interpolational basis, it is not quite so easy. For the Bernstein–Bézier basis (to be introduced below), quite a bit of attention has been paid to this issue in the CAGD literature, and we refer you, for instance, to Farin (1996). The choice of norm for nearness they use is not a coefficientwise norm as we use here, but rather a function norm. Nonetheless, the ideas are similar.

2.2.7 Bernstein–Bézier Polynomials

The following family of polynomials is a basis for polynomials of degree n and is positive on the interval $a < z < b$. Like the Lagrange and Hermite interpolational bases, these are not degree-graded: Each element of the basis is degree n.

$$\phi_k(z) = (b-a)^{-n} \binom{n}{k} (z-a)^k (b-z)^{n-k} . \tag{2.55}$$

The Bernstein–Bézier polynomials of degree 8 or less are displayed in Fig. 2.2. These are extremely useful in computer-aided geometric design. To evaluate poly-

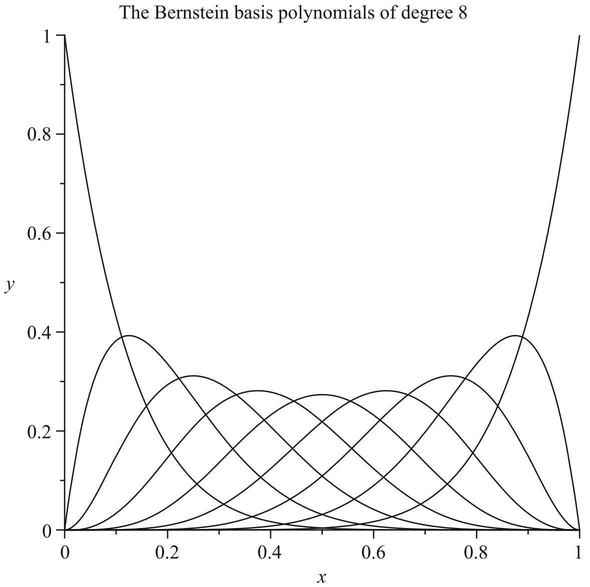


Fig. 2.2 Bernstein–Bézier basis polynomials of degree at most 8

nomials expressed in this basis, you may use de Casteljau's algorithm, as discussed, for example, in Tsai and Farouki (2001). We do not pursue that algorithm further here, except to note that it is partially implemented in MAPLE.[10] This basis has a number of interesting properties, including an *optimal conditioning* property, that we discuss in Chap. 8.

2.3 Condition Number for the Evaluation of Polynomials

Now, let us look at the *condition number* for evaluation of polynomials. This is studied in many works (for example, in de Boor (1978)), but we take the following formulation from Farouki and Rajan (1987).

Theorem 2.8. *If we consider a polynomial*

$$f(z) = \sum_{k=0}^{n} c_k \phi_k(z) \tag{2.56}$$

with coefficients c_k *in the base* $\{\phi_k(z)\}_{k=0}^{n} = 0^n$*, and a perturbed polynomial*

[10] The Bernstein–Bézier basis is not yet well supported in MAPLE: For serious use, we recommend instead the package described in Tsai and Farouki (2001).

$$(f+\Delta f)(z)=\sum_{k=0}^{n} c_k(1+\delta_k)\phi_k(z) \tag{2.57}$$

with perturbed coefficients $c_k(1+\delta_k)$*, then*

$$|\Delta f(z)| \leq \left(\sum_{k=0}^{n} |c_k||\phi_k(z)|\right)\cdot \max_{0\leq k\leq n} |\delta_k|\,. \tag{2.58}$$

If we let $B(z)=\sum_{k=0}^{n}|c_k||\phi_k(z)|$*, we have the simple inequality*

$$|\Delta f(z)| \leq B(z) \max_{0\leq k\leq n} |\delta_k|\,. \tag{2.59}$$

Here is a compact proof of this very important theorem, which we will use repeatedly in this book:

Proof. For the error term Δf, we have

$$\Delta f(z)=\sum_{k=0}^{n} c_k\delta_k\phi_k(z)=\left[c_0\phi_0(z),c_1\phi_1(z),\cdots,c_n\phi_n(z)\right]\begin{bmatrix}\delta_0\\ \delta_1\\ \vdots\\ \delta_n\end{bmatrix}\,. \tag{2.60}$$

In this form, we can use Hölder's inequality (Steele 2004): If $1/p+1/q=1$, then

$$|\mathbf{a}\cdot\mathbf{b}| \leq \|\mathbf{a}\|_p\|\mathbf{b}\|_q\,. \tag{2.61}$$

The result follows directly if we take $\mathbf{a}=[c_0\phi_0(z),\ldots,c_n\phi_n(z)]$, $\mathbf{b}=[\delta_0,\delta_1,\ldots,\delta_n]$, $p=1$, and $q=\infty$. ♮

The number $B(z)$ (and for fixed z, it is indeed just a number) thus serves as an absolute *condition number* for evaluation of the polynomial f at the point z: If we change the coefficients c_k by a relative amount $|\delta_k|\leq\varepsilon$, this means that the value of f might change by as much as $\varepsilon B(z)$. Higham (2004) uses instead $B(z)/|f(z)|$, which is a *relative* condition number, and indeed this may be more informative in many situations.

Remark 2.6. This is the first derivation of an explicit general formula in this book for a *condition number*, which was defined for general computation in Sect. 1.4.2 and used earlier to express the error results for Horner's method. This notion is usually introduced in numerical analysis texts not with polynomial evaluation as we have done here, but rather with the solution of linear systems of equations (which we begin in Chap. 4). The notion is perhaps the most important in the book, and the reader will see it in every single chapter. The reader is urged to make a note of this usage here, and later in Chap. 3, and again in Chap. 4; after that, return and reread Sect. 1.4.2 before going on. ◁

Example 2.3. As an example, we take a single polynomial, $f(t)$, and plot its condition number (2.59) in several different bases. Consider the polynomial $f(t)$ that takes on the values $\boldsymbol{\rho} = [1, -1, 1, -1]$ on $\boldsymbol{\tau} = [-1, -1/3, 1/3, 1]$. Its Lagrange form is

$$\begin{aligned} f(t) = &-\frac{9}{16}\left(t+\frac{1}{3}\right)\left(t-\frac{1}{3}\right)(t-1) - \frac{27}{16}(t+1)\left(t-\frac{1}{3}\right)(t-1) \\ &-\frac{27}{16}(t+1)\left(t+\frac{1}{3}\right)(t-1) - \frac{9}{16}(t+1)\left(t+\frac{1}{3}\right)\left(t-\frac{1}{3}\right), \end{aligned} \tag{2.62}$$

while its monomial form is

$$f(t) = -\frac{9}{2}t^3 + \frac{7}{2}t, \tag{2.63}$$

and its Newton form, if the nodes are taken in the left-to-right order in which they are given, is

$$f(t) = -2 - 3t + \frac{9}{2}(t+1)\left(t+\frac{1}{3}\right) - \frac{9}{2}(t+1)\left(t+\frac{1}{3}\right)\left(t-\frac{1}{3}\right). \tag{2.64}$$

If instead we use the Leja ordering of the nodes (see Chap. 8, Exercise 4), namely, $[-1, 1, -1/3, 1/3]$, the form is

$$f(t) = -t + \frac{3}{2}(t+1)(t-1) - \frac{9}{2}(t+1)\left(t+\frac{1}{3}\right)(t-1). \tag{2.65}$$

For each of these, $B(t)$ is simply the sum of the absolute values of the terms. The results are displayed in Fig. 2.3. ◁

Remark 2.7. In Fig. 2.3, we see $B(t)$ plotted for all but that for Eq. (2.64), which is so large (going up to 25) that it would compress the graph. This example is hardly unique: The Newton basis is often poorly conditioned and, moreover, depends on the ordering of the nodes. We will pursue this in great detail in the exercises, and again in Chap. 8. This book differs from many numerical analysis texts in that it avoids use of the Newton basis for this reason and uses the Lagrange and Hermite interpolational bases instead, which are often better conditioned. This understanding in a broad popular sense is a relatively recent development and is due to the papers Berrut and Trefethen (2004) and Higham (2004), although in a more limited sense, it was known previously. ◁

Example 2.4. Let us continue Example 2.3 with another basis, namely, the Bernstein–Bézier basis, given by

$$\phi_k(z) = (b-a)^{-n}\binom{n}{k}(z-a)^k(b-z)^{n-k}. \tag{2.66}$$

For polynomials of degree 3 on the interval $-1 \le z \le 1$, we just let $a = -1$, $b = 1$, and $n = 3$, so that the basis elements are easily computed. We then find that if

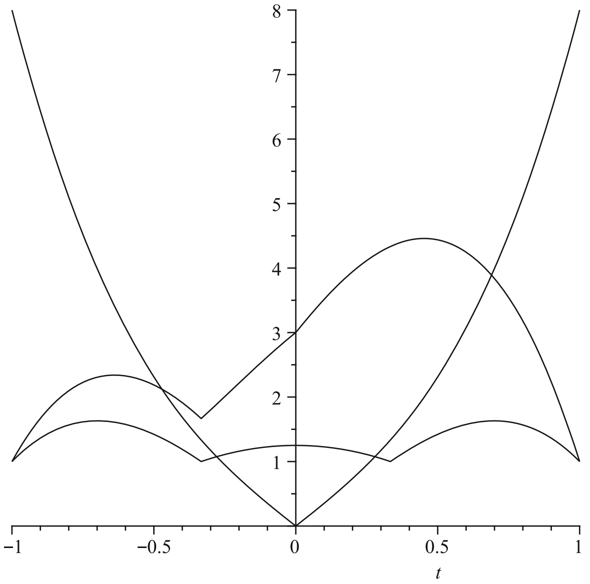


Fig. 2.3 Condition number of evaluation of a particular degree-3 polynomial in three different bases: Lagrange (which is best), Newton with the Leja ordering (which is next-best), and standard monomials (which is worst in this example)

$$f(z) = \sum_{k=0}^{3} c_k \phi_k(z) \,,$$

$c_0 = 1$, $c_1 = {}^{-17}\!/3$, $c_2 = {}^{17}\!/3$, and $c_3 = -1$. Then, the condition number

$$\sum_{k=0}^{3} |\, c_k \,| \cdot |\, \phi_k(z) \,|$$

is displayed in Fig. 2.4, where it is shown with the condition numbers from Fig. 2.3. ◁

Remark 2.8. After this discussion, it is clear that the same polynomial will have different condition numbers in different bases. It is shown in Farouki and Goodman (1996) that among all polynomial bases that are nonnegative on an interval, the Bernstein–Bézier basis has optimal condition numbers in a generic sense. Taken as a whole, one can expect a polynomial to have a smaller condition number in the Bernstein–Bézier basis than in any other nonnegative basis. The Farouki–Goodman theorem thus guarantees, for example, that the Bernstein–Bézier basis is better than the monomial basis in general. For a *particular* polynomial, however, this need not be true, as we have seen in the previous example. In Chap. 8, we will show,

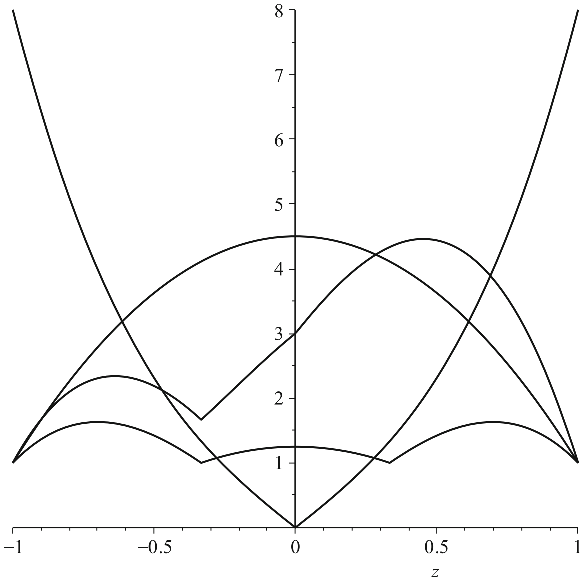


Fig. 2.4 Condition numbers from Fig. 2.3, together with the condition number of the same polynomial expressed in the Bernstein–Bézier basis. This graph shows that, for this example, the Bernstein–Bézier basis is worse than the Lagrange basis and comparable to the Newton basis with Leja ordering

using the same techniques as Farouki and Goodman (1996) used, that if we relax the nonnegativity condition, then the Lagrange basis has the same optimality property. This in some sense explains why the Lagrange basis did so well in Example 2.3. ◁

Remark 2.9. The question of "which basis is best overall?" is somewhat vexed. The answer is, "It really depends on the problem, and what information you want." In the simple example above, it is clear that the Lagrange basis has a lower condition number than either of the Newton bases or the Bernstein–Bézier bases, over the entire interval. The monomial basis, however, is better than the Lagrange basis, for all z "near enough" to the origin. For this problem, the conditioning of the Lagrange basis expression is better for "most" of the z in this restricted interval.

The picture would change if we took a different interval, or if we considered instead the complex disk $|z| < 1$ (where, in fact, the monomial basis would show itself to good advantage). It is the position of this book that the Lagrange basis is *generally* to be preferred over other bases, on the principle that you probably have sampled your polynomial where you know it best; while the Bernstein–Bézier basis is provably the best on an interval for generic polynomials (and is widely used in CAGD in part because of that); and that the monomial basis is likely overused—that is, often used where it shouldn't be—but can be the best tool for the job.

The relative condition number ${}^{|B(z)|}\!/\!{}_{|f(z)|}$ is also of interest (perhaps of more interest). Since this example polynomial has a zero at $t = 0$, the monomial basis condition number shows itself to be best there—$B_{\text{monomial}}(0) = 0$ as well, whereas all the other absolute condition numbers are nonzero at zero, and so the relative condition of the monomial basis is the only finite one there.

Finally, the flexibility and uniform approximation properties of discrete Chebyshev bases—that is, Lagrange interpolation on nodes that are the zeros of Chebyshev polynomials—make them extremely interesting to the computational scientist. See the Chebfun package as described in Battles and Trefethen (2004) and subsequent papers. ◁

Example 2.5. As we have seen in Chap. 1, the Airy function has a Taylor series that converges for all z; it can be written as

$$\mathrm{Ai}(z) = 3^{-2/3} \sum_{n=0}^{\infty} \frac{z^{3n}}{9^n n! (n - 1/3)!} - 3^{-4/3} \sum_{n=0}^{\infty} \frac{z^{3n+1}}{9^n n! (n + 1/3)!} \,. \tag{2.67}$$

Consider using the degree-127 truncation of this Taylor series as a way of approximating $\mathrm{Ai}(z)$ for various z. In applications, for instance, the geometric optics of the rainbow, the zeros of $\mathrm{Ai}(z)$ are often important, so we want to accurately assess it there. So let us focus on values near $z = -7.94$, which is somewhat close to a zero of $\mathrm{Ai}(z)$. A preliminary analysis shows that, in theory, the degree-127 truncation has *more* than enough terms for convergence—here, we ought to be able to get about 25 significant figures of $\mathrm{Ai}(-7.94)$ if we want. We are using so many terms here in part to show that the mathematical theory of convergence is not at issue for this example. Write the degree-127 polynomial in Horner form (Exercise 2.29 contains two programs that implement an efficient, specific variation of Horner form for this particular polynomial).

If we use only 8 digits in our computation, because we only want 8 digits in our answer, we get $\mathrm{Ai}(-7.94) = 0.00359469$. Here is how the (general) Horner form begins, with 8-digit coefficients:

$$0.35502806 + z(-0.25881940 + z(0.059171345 + \cdots)) \,. \tag{2.68}$$

If we use 16 digits, we get a different answer, beginning with

$$\mathrm{Ai}(-7.94) = 0.0039158060872139 \,.$$

Only a single significant digit was right the first time! If we don't use Horner's form, the answer is worse, by the way. In order to understand what has gone wrong with the 8-digit computation, we need to plot the $B(z)$ function, which is the same Taylor series but with all positive signs and with powers of $|z|$, not z. This is plotted in Fig. 2.5. We see that the $B(z)$ function becomes very large, for large z: We say that the (monomial basis) polynomial approximation to the Airy function that we derived from Taylor series is ill-conditioned to evaluate for large $|z|$. This point deserves emphasis: Taylor series about the origin are often *impractical to use* for large $|z|$ because the resulting polynomial expression, although adequate in theory to deliver accuracy, is far too *ill-conditioned to use*. The condition number we see at the right is about 10^{15}—10 rounding errors, and we cannot count on any accuracy in the result (and since we are adding up hundreds of terms, we will make many more than 10 rounding errors). Near $z = -7.94$, the condition number is about 10^9. This phenomenon is sometimes known as "the hump" (see Exercise 2.16).

There is a bit more to say, for this example, though: If we take each separate series in Eq. (2.67), the one multiplied by $3^{-2/3}$ (call it $f_1(z)$) and the other multiplied by $3^{-4/3}$ (call it $f_2(z)$), and plot *their* condition numbers, we see that each of them has condition number 1 for $z > 0$ (because all terms are positive). So we can say that each of them is accurately evaluated for $z > 0$. (This is how the programs in Exercise 2.29 do it, by the way.) Yet the condition number for $\mathrm{Ai}(z) = 3^{-2/3} f_1(z) - 3^{-4/3} f_2(z)$ grows very large, even for positive z. This is because each of $f_1(z)$ and $f_2(z)$ grows very large for large positive z, while $\mathrm{Ai}(z)$ gets very small indeed—that is, we are computing $\mathrm{Ai}(z)$ as the tiny difference of two large numbers. This is a recipe for catastrophic cancellation. Notice that this shows up automatically in the condition number analysis: we have discovered directly that the condition number of this polynomial is very large. We defer analysis of the condition number of $\mathrm{Ai}(z)$ itself to Chap. 3 (Exercise 3.6). ◁

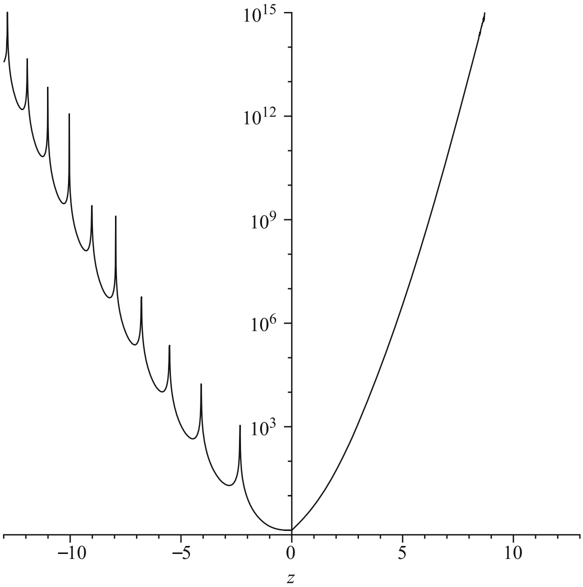


Fig. 2.5 The condition number for evaluation of the degree-127 Taylor polynomial for the Airy function $\mathrm{Ai}(z)$ on $-13 \le z \le 13$. Note that it goes to infinity in very narrow spikes (the graph only shows a portion of each spike since the singularity is so narrow) around each zero, but even away from zeros, the condition number grows very rapidly with $|z|$

2.4 Pseudozeros

We now look at the relationship between the condition number for *evaluation* of polynomials and the condition number for *rootfinding* for polynomials. In mathematical terms, given $\varepsilon > 0$ and weights $w_i \geq 0$, $0 \leq i \leq n$, not all zero, define the set of *pseudozeros*

$$\Lambda_\varepsilon(f) = \left\{ z \,\middle|\, (f + \Delta f)(z) = 0, \text{ where } \Delta f = \sum_{i=0}^{n} \Delta c_i \phi_i(z) \text{ and each } |\Delta c_i| \leq \varepsilon w_i \right\}.$$

These are the roots of the polynomials that are near f. Studying this set will help us to understand what happens if the coefficients of our polynomials are changed somehow (perhaps due to measurement error, or to numerical errors in computation). To do so, we make use of the following theorem:

Theorem 2.9. *Let $\Lambda_\varepsilon(f)$ be defined as above. Then*

$$\Lambda_\varepsilon(f) = \left\{ z \,\middle|\, \left|(f(z))^{-1}\right| \geq (\varepsilon B(z))^{-1} \right\}, \tag{2.69}$$

where $B(z) = \sum_{i=0}^{n} w_i |\phi_i(z)|$ or, equivalently for scalar polynomials,

$$\Lambda_\varepsilon(f) = \left\{ z \,\middle|\, |f(z)| \leq \varepsilon B(z) \right\}. \tag{2.70}$$

Proof. First, suppose $z \in \Lambda_\varepsilon(f)$. Moreover, if $|\Delta c_i| \leq \varepsilon w_i$, and $\Delta f(z) = \sum_{i=0}^{n} \Delta c_i \phi_i(z)$, then

$$|\Delta f(z)| \leq \sum_{i=0}^{n} |\Delta c_i||\phi_i(z)| \leq \sum_{i=0}^{n} \varepsilon w_i |\phi_i(z)| = \varepsilon B(z),$$

so that $\Lambda_\varepsilon(f) \subseteq \left\{ z \,\middle|\, |f(z)| \leq \varepsilon B(z) \right\}$. Now, suppose $|f(z)| \leq \varepsilon B(z)$. Take

$$\Delta c_i = -\mathrm{signum}\left(\overline{\phi_i(z)}\right) w_k \frac{f(z)}{B(z)}.$$

Then it follows that

$$|\Delta c_i| = \left| w_i \frac{f(z)}{B(z)} \right| \leq \varepsilon w_i \frac{B(z)}{B(z)} = \varepsilon w_i.$$

Also, observe that

$$f(z) + \sum_{i=0}^{n} \Delta c_i \phi(z) = f(z) + \sum_{i=0}^{n} \frac{-w_i |\phi_i(z)| f(z)}{B(z)}$$

$$
\begin{aligned}
&= f(z) - \frac{f(z)}{B(z)} \sum_{i=0}^{n} w_i |\phi_i(z)| \\
&= f(z) - f(z) = 0\,.
\end{aligned}
$$

Thus, the set identity is obtained. ♮

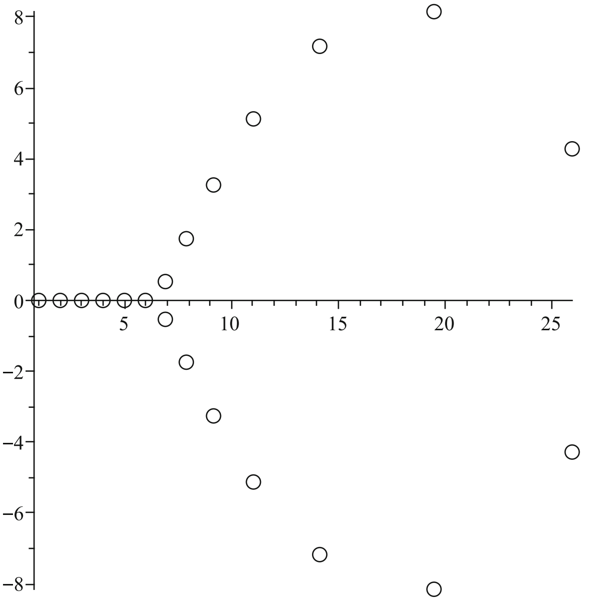


Fig. 2.6 Zeros of a small perturbation of the Wilkinson polynomial $W(z) = \prod_{k=1}^{20}(z-k)$, after first having been expanded into the monomial basis $W_k(z) = z^{20} - 210z^{19} + \cdots$

Remark 2.10. It is no coincidence that the condition number of Theorem 2.8 appears as the expansion factor in equations in the proof of Theorem 2.9. An ill-conditioned polynomial, with large B, will have its roots spread widely when the coefficients are changed.

This is a very useful and important result: It says that if $|f(z)|$ is *small*, then z is the exact root of a *nearby* polynomial $f(z) + \Delta f(z)$. Note that this works for only one root at a time. ◁

Example 2.6. The Wilkinson polynomial[11] can be written as

$$
W(z) = \prod_{k=1}^{20}(z-k) = z^{20} - 210z^{19} + \cdots + 20! \qquad (2.71)
$$

[11] See, for instance, Wilkinson (1984)

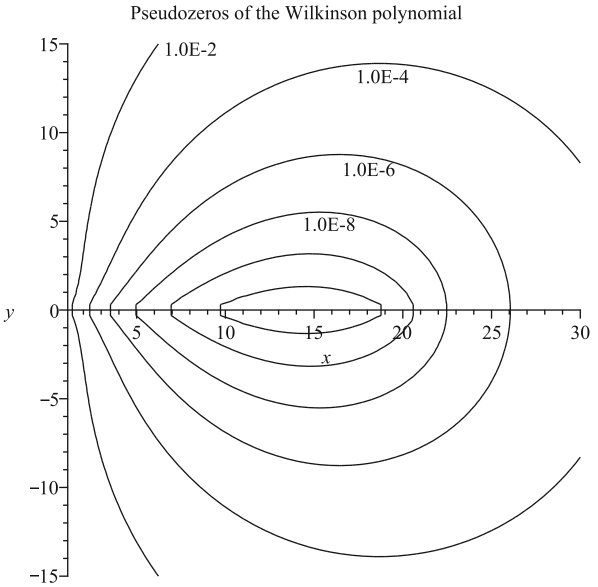


Fig. 2.7 Pseudozeros of the Wilkinson polynomial expressed in the monomial basis: a plot of the contours of $\frac{|f(x+iy)|}{B(x+iy)}$

when expanded into the monomial basis. Compare Figs. 2.6 and 2.7. In the latter case, we see contours bounding the sets of zeros of *all* perturbations of the Wilkinson polynomial expressed in the monomial basis—as we see, quite small perturbations can have dramatic effects on the location of the zeros. In the former, we have an explicit perturbation of one coefficient (again in the monomial basis), and the perturbed zeros lie pretty much along one of the contours of the latter. In contrast, if we *don't* expand it, then the natural basis to recognize in which it is written is the Lagrange basis on the nodes 1, 2, ..., 20, and (say) 0, so $W(z) = 20!\, L_0(z)$ has only one nonzero coefficient. The condition number in *this* basis is, remarkably, 0 at all the roots, in an absolute sense; in a relative sense, the condition number is just 1. Of course, this is unsurprising, and uninformative: If we know the roots, then they are easy to find. However, there is something more useful in this observation than just that, and we return to it later. ◁

2.5 Partial Fractions

Every reader of this book will have encountered the partial fraction decomposition. However, it is all too common to encounter only the hand practice, and not the theory or a practical algorithm. We give the theory here, and later we give a practical algorithm for the easy case that we need for interpolation. The basic object under study here is the class of rational functions and a simple representation for them.

Definition 2.5 (Rational function). A function $f : \mathbb{C} \to \mathbb{C}$ is called a *rational function* if there exist polynomials $p(z)$ and $q(z)$ such that

$$f(z) = \frac{p(z)}{q(z)} \qquad \forall z \in \mathbb{C}\,, \tag{2.72}$$

except possibly at the zeros of $q(z)$. If the degree of $p(z)$ is n and the degree of $q(z)$ is m, then we say that $f(z)$ is here represented as an $[n,m]$-degree rational function. By convention, the identically zero function $f(z) \equiv 0$ is also called a rational function, for example, taking $q(z) = 1$. $\triangleleft$

Rational functions often arise in approximation theory. One class of these are called Padé approximants:

Definition 2.6 (Padé approximant). A *Padé approximant* to a function $f(z)$ is a rational function whose coefficients are wholly determined by matching the Taylor series of $f(z)$. $\triangleleft$

We can find a unique representative $p(z)/q(z)$ for a rational function $f(z)$ by insisting that $p(z)$ and $q(z)$ have no common factors (dividing out the GCD) and normalizing one of $f(z)$ and $q(z)$ in some way—often by making $f(z)$ monic by dividing the numerator and the denominator by the leading coefficient of $q(z)$, or by making the norm of the vector of coefficients of $q(z)$ equal to 1 and insisting that one particular coefficient be positive, or simply by insisting that $\mathbf{a} \cdot \mathbf{q} = 1$ for some given nonzero vector $\mathbf{a}$, where $\mathbf{q}$ means the vector of coefficients of the polynomial $q(z)$ expanded in the monomial basis. Moreover, since the polynomials are linear in their coefficients, we may divide the numerator and denominator by a constant in order to make this dot product 1.

We also usually insist that $\deg p < \deg q$, by first doing polynomial division if necessary: $p = Qq + R$ and so $p/q = Q + R/q$, separating out a polynomial part $Q(z)$ and leaving a rational part $R(z)/q(z)$ with the desired "proper form."

We now state and prove the key theorem of this section:

Theorem 2.10 (Partial fraction decomposition). *Suppose we have already found machine number representations of all the roots of the denominator,*[12] *and that*

$$q(z) = \prod_{k=0}^{n} (z - \tau_k)^{s_k} \tag{2.73}$$

is the unique factorization of monic $q(z)$ into distinct factors over $\mathbb{C}$. That is, $\tau_i = \tau_j \Leftrightarrow i = j$, and each integer $s_k \geq 1$. Let $m = \sum_{k=0}^{n} s_k$. Note that the degree of $q(z)$ is exactly m and that $q(z)$ is not identically zero (if the product is empty with $n < 0$, then $q(z) = 1$ by convention). Suppose that $m \geq 1$. Suppose $p(z)$ is a polynomial

[12] Of course, this avoids the hard questions of how to do this if we start with $q(z)$ expressed in some other basis, and also the hard question of what are the consequences of approximating polynomial roots by machine numbers. But for the interpolation applications that we need in this book, this assumption suffices.

with $\deg p < m$ *having no common factor with* $q(z)$*. Then there exist* m *numbers* $\alpha_{i,j}$ *(with* $0 \le i \le n$ *and* $0 \le j \le s_i - 1$*) such that* $\forall z \notin \{\tau_0, \tau_1, \dots \tau_n\}$*,*

$$\frac{p(z)}{q(z)} = \sum_{i=0}^{n} \sum_{j=0}^{s_i-1} \frac{\alpha_{i,j}}{(z-\tau_i)^{j+1}} . \tag{2.74}$$

The numbers $\alpha_{i,j}$ *provide the decomposition.*

Proof. We proceed by induction on the degree $m \ge 1$, which gives a perfectly satisfactory algorithm to use in hand computation. The base of the induction, $m = 1$, is trivial: $\alpha_{0,0} = p_0 = p(z)$, because $\deg p = 0$ and there is nothing to prove. Suppose now that the theorem is true for all polynomials with degrees $m-1$ or less. Let

$$\alpha_{0,s_0-1} = p(\tau_0) \prod_{k=1}^{n} (\tau_0 - \tau_k)^{-s_k}$$

and consider

$$\frac{p(z)}{q(z)} - \frac{\alpha_{0,s_0-1}}{(z-\tau_0)^{s_0}} = \frac{p(z) - \alpha_{0,s_0-1} \prod_{k=1}^{n} (z-\tau_k)^{s_k}}{q(z)} .$$

We claim that the numerator and denominator on the right have a nontrivial common divisor, $z - \tau_0$. Since $s_0 \ge 1$, it is clear that $(z-\tau_0) \mid q(z)$. It only remains to show that this factor divides the numerator. It is equivalent to show that τ_0 is a zero of the numerator. But this is obvious, because the polynomial at $z = \tau_0$ has the value $p(\tau_0) - p(\tau_0) \prod_{k=1}^{n} (\tau_0 - \tau_k)^{-s_k} \prod_{k=1}^{n} (\tau_0 - \tau_k)^{s_k} = 0$.

On dividing the numerator and denominator on the right by $z - \tau_0$ (as many times as necessary but certainly at least once), we are left with a proper rational function on the right with denominator of the form (2.73) and having degree strictly less than m. By the induction hypothesis, this can be expressed uniquely in partial fraction form, thus completing the proof of the theorem. ♮

Remark 2.11. This proof provides an excellent hand algorithm: see Scott and Peeples (1988). It is also "self-checking": If exact division does *not* occur at the second step, we know that we have made an arithmetic blunder. ◁

Example 2.7. Consider

$$\frac{1}{z^2(z-1)^2} = \frac{1}{z^2} + \mathrm{Rest}(z)$$

on taking out the leading term in z as $z \to 0$. Rearranging as in the proof of the theorem, we get

$$\begin{aligned} \mathrm{Rest}(z) &= \frac{1}{z^2(z-1)^2} - \frac{1}{z^2} = \frac{1-(z-1)^2}{z^2(z-1)^2} = \frac{1-z^2+2z-1}{z^2(z-1)^2} \\ &= \frac{z(2-z)}{z^2(z-1)^2} \qquad (2.75) \\ &= \frac{2-z}{z(z-1)^2} \qquad (2.76) \end{aligned}$$

and this is a proper rational function with a degree-3 denominator, one less than we started with. As stated previously, the exact cancellation from Eq. (2.75) to Eq. (2.76) is necessary, and if it doesn't happen, then we know that we have made an arithmetic blunder.

The process can be continued, to get

$$\frac{1}{z^2(z-1)^2} = \frac{1}{z^2} + \frac{2}{z} - \frac{2}{z-1} + \frac{1}{(z-1)^2}\,. \tag{2.77}$$

The numerators on the right-hand side are the $\alpha_{i,j}$ desired. ◁

Remark 2.12. Later we will need this particular partial fraction decomposition many times: It is the foundation for cubic Hermite interpolation. The reader is urged to complete the computation above and confirm Eq. (2.77). ◁

This algorithm can be implemented recursively, once an algorithm for division of polynomials by linear factors $z-\tau_k$ has been made available (and, of course, this can be done in any polynomial basis). For our purposes in this book, however, there is a more practical algorithm for partial fractions, based on *local Taylor series*. In order to develop that algorithm (and indeed for many other numerical purposes), we need to learn to manipulate formal power series, and so we turn to this in the next section.

Remark 2.13. MAPLE has several commands to compute partial fraction decompositions. Using exact arithmetic, the command

```
convert( R, parfrac, z, true );
```

does the trick (the `true` flag means that the rational function R has already had its denominator factored). However, at the time of this writing, for floating-point arithmetic, this command is not always satisfactory, because it converts internally to a monomial basis centered at 0, and this can induce numerical instability in the algorithm because the intermediate monomial basis representations are ill-conditioned. See Exercise 2.28. ◁

2.6 Formal Power Series Algebra

Numerical methods rely heavily on Taylor series. In this section we give a short generalized reminder[13] of how to operate on them. Suppose that, instead of being given a function, we are directly given the series and that we want to do standard operations with it, such as adding it, multiplying it, or dividing it by another series, differentiating or integrating it, exponentiating it, and so on. We will examine how to do so in this section. To begin with, suppose we have two series, given by

$$u = \sum_{k=0}^{N} u_k(x-a)^k + O(x-a)^{N+1} \tag{2.78}$$

[13] This generalization includes $O(n^2)$ algorithms for computation.

$$v = \sum_{k=0}^{N} v_k (x-a)^k + O(x-a)^{N+1} \,. \tag{2.79}$$

The scalar linear combination is defined to be

$$\alpha u + \beta v = \sum_{k=0}^{N} (\alpha u_k + \beta v_k)(x-a)^k + O(x-a)^{N+1} \,. \tag{2.80}$$

In other words, to add, subtract, and scalar-multiply series, we simply add, subtract, and scalar-multiply the corresponding coefficients. We examine the other operations in the following subsections.

2.6.1 Multiplication of Series and the Cauchy Convolution

The product $w = uv$ of two series u and v, as in Eqs. (2.78) and (2.79), can be written

$$w = uv = \sum_{k=0}^{N} w_k (x-a)^k + O(x-a)^{N+1} \tag{2.81}$$

as any other series. The problem, then, is to express the coefficients w_k in terms of the coefficients of u and v. The relationship in question is simply

$$w_k = \sum_{j=0}^{k} u_{k-j} v_j = \sum_{j=0}^{k} u_j v_{k-j} \,. \tag{2.82}$$

This is the Cauchy product formula or *convolution product*. It is crucial in what follows. It can be done faster than the direct sum formula above, by using the fast Fourier transform: See Henrici (1979b). See also Chap. 9 in this book.

Example 2.8. If we are given the series

$$u = 1 + 2(x-a) + 3(x-a)^2 + 4(x-a)^3 + 5(x-a)^4 + 6(x-a)^5 + O\left((x-a)^6\right)$$

and

$$v = 2 - 3(x-a) + 4(x-a)^2 - 5(x-a)^3 + 6(x-a)^4 - 7(x-a)^5 + O\left((x-a)^6\right) ,$$

then their product $w = uv$ has the series starting

$$uv = 2 + (x-a) + 4(x-a)^2 + 2(x-a)^3 + 6(x-a)^4 + 3(x-a)^5 + O\left((x-a)^6\right) .$$

These series were computed using the `series` command in MAPLE, which, among other things, implements the Cauchy convolution product. ◁

Is the computation of the Cauchy convolution numerically stable? If $N = 0$, so that we are multiplying constants, then (obviously) Cauchy convolution is numerically stable: $u_0 v_0 (1+\delta)$ can be interpreted as the exact product of $u_0(1+{}^{\delta}\!/_2)$ and $v_0(1+\delta)(1+{}^{\delta}\!/_2)^{-1} \approx v_0(1+{}^{\delta}\!/_2)$ by the IEEE standard. Cauchy convolution is *normwise* forward stable, as we will see, for any fixed N; but it is not componentwise stable for $N > 1$, as we will also see. But it is stable for $N = 1$.

Theorem 2.11. *Cauchy convolution is componentwise stable if* $N = 1$.

Proof. Suppose $N = 1$, so that $u = u_0 + u_1 z + O(z^2)$, and $v = v_0 + v_1 z + O(z^2)$. Then $uv = u_0 v_0 + (u_0 v_1 + u_1 v_0)z + O(z^2)$ in exact arithmetic. If we are using floating-point arithmetic instead, then as we just saw, we may choose perturbations in u_0 and v_0 that allow us to interpret the first term as the exact product of perturbed u_0 and v_0. Suppose that we have done so, with $\hat{u}_0 = u_0(1+\delta_1)$ and $\hat{v}_0 = v_0(1+\delta_2)$, where δ_1 and δ_2 are such that $(1+\delta_1)(1+\delta_2) = (1+\delta_0)$ with $\hat{u}_0\hat{v}_0 = u_0 v_0(1+\delta_0)$ and $|\delta_0| \le \mu_M$. As shown above, we may choose δ_1 and δ_2 so that each is also smaller than μ_M in magnitude. Now we wish to interpret the floating-point value $u_0 \otimes v_1 \oplus u_1 \otimes v_0$ as $\hat{u}_0\hat{v}_1 + \hat{u}_1\hat{v}_0$ with $\hat{u}_1 = u_1(1+\delta_3)$ and $\hat{v}_1 = v_1(1+\delta_4)$, with each of δ_3 and δ_4 small. We break the proof up into cases.

In the first case, suppose that $v_1 = 0$. Then we are multiplying u by a constant, and obviously each term in the product is the exact product of v_0 with a relatively minor change in u_0 and u_1: We have $\hat{u}_0 = u_0(1+\delta_0)$ and can take $\hat{u}_1 = u_1(1+\delta_3)$ directly, and leave $\hat{v}_0 = v_0$, and similarly if $u_1 = 0$.

Now suppose we are in the second case, where neither u_1 nor v_1 is zero. Then

$$\begin{aligned} u_0 \otimes v_1 \oplus u_1 \otimes v_0 &= (u_0 v_1(1+\delta_5) + u_1 v_0(1+\delta_6))(1+\delta_7) \\ &= (\hat{u}_0 v_1 (1+\delta_0)^{-1}(1+\delta_5) + u_1 \hat{v}_0(1+\delta_6))(1+\delta_7) \\ &= \hat{u}_0\hat{v}_1 + \hat{u}_1\hat{v}_0, \end{aligned} \tag{2.83}$$

where $\hat{v}_1 = v_1(1+\delta_0)^{-1}(1+\delta_5)(1+\delta_7)$ is only three rounding errors different to v_1 and $\hat{u}_1 = u_1(1+\delta_6)(1+\delta_7)$ is only two rounding errors different to u_1. That is, the computed Cauchy convolution if $N = 1$ is the exact product of two series that differ only minutely in a relative sense to each multiplicand series. ♮

However, for $N = 2$, this kind of analysis cannot succeed.

Theorem 2.12. *For $N = 2$, Cauchy convolution can be componentwise unstable: That is, the computed product of two series of order $O(z^3)$ is not necessarily the exact product of any two nearby series, where "nearby" means each coefficient is relatively close to the original.*

Proof. Using a more systematic notation to help with the bookkeeping, suppose to the contrary that we may choose relative perturbations $\hat{u}_k = u_k(1+\delta_k^u)$ and $\hat{v}_k = v_k(1+\delta_k^v)$ in order to match the rounding errors in the computation, which we will denote by ε_k. We would then have

$$u_0 v_0 (1+\delta_0^u)(1+\delta_0^v) = u_0 v_0 (1+\varepsilon_0)$$

$$
\begin{aligned}
u_0v_1(1+\delta_0^u)(1+\delta_1^v)+u_1v_0(1+\delta_1^u)(1+\delta_0^v) &= u_0v_1(1+\varepsilon_1)(1+\varepsilon_3)\\
&\quad +u_1v_0(1+\varepsilon_2)(1+\varepsilon_3),
\end{aligned}
$$

where each $|\varepsilon_j| < \mu_M$, the unit roundoff. As we saw in the previous theorem, if we stop here, we may choose small δ's in order to satisfy these constraints: For $N=1$, we may interpret the rounding errors as small relative backward errors. However, we need one more equation for $N=2$:

$$
\begin{aligned}
&u_2v_0(1+\delta_2^u)(1+\delta_0^v)+u_1v_1(1+\delta_1^u)(1+\delta_1^v)+u_0v_2(1+\delta_0^u)(1+\delta_2^v)\\
&=u_2v_0(1+\varepsilon_4)(1+\varepsilon_6)(1+\varepsilon_8)+u_1v_1(1+\varepsilon_5)(1+\varepsilon_6)(1+\varepsilon_8)+u_0v_2(1+\varepsilon_7)(1+\varepsilon_8).
\end{aligned}
$$

Because the u_k and v_k are independent variables, each monomial gives an equation for the unknown perturbations, so that we have

$$(1+\delta_0^u)(1+\delta_0^v) = (1+\varepsilon_0) \tag{2.84}$$

$$(1+\delta_0^u)(1+\delta_1^v) = (1+\varepsilon_1)(1+\varepsilon_3) \tag{2.85}$$

$$(1+\delta_1^u)(1+\delta_0^v) = (1+\varepsilon_2)(1+\varepsilon_3)\,, \tag{2.86}$$

and from the $O(z^2)$ term, we will only need

$$(1+\delta_1^u)(1+\delta_1^v) = (1+\varepsilon_5)(1+\varepsilon_6)(1+\varepsilon_8) \tag{2.87}$$

to arrive at a contradiction. Multiply Eqs. (2.85) and (2.86) together and divide by Eq. (2.84) to get

$$(1+\delta_1^u)(1+\delta_1^v) = \frac{(1+\varepsilon_1)(1+\varepsilon_3)^2(1+\varepsilon_2)}{(1+\varepsilon_0)}\,. \tag{2.88}$$

For this to hold simultaneously with (2.87) requires that the rounding errors ε_5, ε_6, and ε_8 be perfectly correlated with the earlier rounding errors ε_0, ε_1, ε_2, and ε_3. In general, this does not happen. Therefore, there is no possible set of perturbations δ_k^u and δ_k^v that allows rounding errors in Cauchy convolution for $N>1$ to be interpreted as a small relative backward error. ♮

Remark 2.14. In the *forward* error sense, this computation also shows that the componentwise relative error may be infinite. Take an example where $u_0v_2+u_1v_1+u_2v_0=0$ in exact arithmetic. Then the rounding errors, which will be proportional to $|u_0v_2|+|u_1v_1|+|u_2v_0|$, will be infinitely large in comparison with the reference result of 0. ◁

However, there is a forward accuracy result for all N in the normwise sense, as follows. If $c_n=\sum_{j=0}^{n}u_jv_{n-j}$, then Eq. (3.5) in (Higham 2002 section 3.1), which gives us a general result on the forward accuracy of inner products, tells us that

$$|\hat{c}_n-c_n| \le \sqrt{2}\gamma_{n+1}\sum_{j=0}^{n}|u_j||v_{n-j}|\,. \tag{2.89}$$

If all terms u_j and v_j are positive, this is a decent relative accuracy (and the constant in front can be improved with minor modifications of how the sum is done and in which order). If, however, c_n is very small while some u_j and v_j are large in magnitude, then there must be cancellation, and the error bound will then be large to reflect this.

Remark 2.15. This difficulty may be mitigated by performing this recurrence relation in higher precision, or by using certain compensated summation techniques as described in Higham (2002). However, the inaccuracy is often of little consequence in computation with the resulting series, anyway, even if the recurrence relation is performed in a naive way. The reason is simply that the errors grow at worst in c_k like $O((k+1)\|\mathbf{u}\|\|\mathbf{v}\|\mu_M)$, and thus for the low-order terms, the error is small in any case; and the high-order terms are used only together with high powers of $(z-a)$, which is presumed small. Thus, the total error in the *computed sum* $\sum_{j=0}^{N} c_j(z-a)^j$ will be small enough: The terms with larger errors will not contribute much to the total sum. ◁

Finally, we leave aside the question of whether the Cauchy convolution is well-conditioned, which we will take it up in the exercises in Chap. 3.

2.6.2 Division of Series

Let us now consider the case of division. If we consider

$$r = \frac{u}{v} = \sum_{k=0}^{N} r_k (x-a)^k + O(x-a)^{N+1}\,, \tag{2.90}$$

then we must have $u = rv$, so that

$$u_k = \sum_{j=0}^{k} r_j v_{k-j} = r_k v_0 + \sum_{j=0}^{k-1} r_j v_{k-j}, \tag{2.91}$$

and we see that for r_k to be defined we must have $v_0 \neq 0$. Then

$$r_k = \frac{1}{v_0}\left(u_k - \sum_{j=0}^{k-1} r_j v_{k-j}\right) \tag{2.92}$$

and the base of the recurrence is (if $v_0 \neq 0$)

$$r_0 = \frac{u_0}{v_0}. \tag{2.93}$$

However, if $v_0 = 0$, then no series for r exists, unless perhaps also $u_0 = 0$ and we may cancel a factor in $^u\!/_v$.

Example 2.9. With the same u and v that we used in Example 2.8, we find that

$$\frac{u}{v} = \frac{1}{2} + \frac{7}{4}(x-a) + \frac{25}{8}(x-a)^2 + \frac{71}{16}(x-a)^3 + \frac{185}{32}(x-a)^4 + O\left((x-a)^5\right) ,$$

while

$$\frac{v}{u} = 2 - 7(x-a) + 12(x-a)^2 - 16(x-a)^3 + 20(x-a)^4 + O\left((x-a)^5\right) .$$

Again these series were computed in MAPLE, which knows how to do series algebra including division, and cancels common factors in order to avoid division by zero wherever possible. MAPLE also knows how to work with several generalizations of Taylor series, including *Laurent* series, which allow negative integer powers of $(x-a)$, and *Puiseux* series, which allow fractional powers. The algebra of these is a straightforward extension of that for Taylor series. We will occasionally have need for these generalizations. ◁

2.6.3 Differentiation and Integration

Differentiation of power series is very straightforward. If we are given a series

$$u = \sum_{k=0}^{N} u_k (x-a)^k + O(x-a)^{N+1} ,$$

then it is easy to see that its derivative is

$$\begin{aligned}\frac{du}{dx} &= \sum_{k=0}^{N} k u_k (x-a)^{k-1} + O(x-a)^N \\ &= \sum_{k=0}^{N-1} (k+1) u_{k+1} (x-a)^k + O(x-a)^N .\end{aligned}$$

Moreover, its integral is also directly seen to be

$$\begin{aligned}\int_a^x u(\xi)d\xi &= u_0(x-a) + u_1\frac{(x-a)^2}{2} + \ldots \\ &= \sum_{k=0}^{N} \frac{u_k}{k+1}(x-a)^{k+1} + O(x-a)^{N+2} .\end{aligned}$$

These two simple operations will be applied to many problems in this book.

2.6.4 The Algebra of Series

The rules we have examined already give us the series for all polynomials and rational functions. Using these rules, we see how to square a series,

$$\begin{aligned} x^2 &= \left(a+(x-a)+O(x-a)^{N+1}\right)^2 \\ &= a^2+2a(x-a)+(x-a)^2+O(x-a)^{N+1}\,, \end{aligned}$$

or to take higher powers. Moreover, we see that

$$\begin{aligned} \frac{1}{x} &= \frac{1+O(x-a)^{N+1}}{a+(x-a)+O(x-a)^{N+1}} \\ &= \sum_{k=0}^{N} \frac{(-1)^k}{a^{k+1}}(x-a)^k+O(x-a)^{N+1}\,. \end{aligned}$$

As a result, we can also find the series for $\ln(x)$ by using the integration rule. Observe that

$$\ln(x) = \int_1^x \frac{dt}{t} = \int_1^a \frac{dt}{t} + \int_a^x \frac{dt}{t}\,.$$

From this, we obtain

$$\begin{aligned} \ln(x) &= \ln(a) + \int_a^x \left(\sum_{k=0}^{N} \frac{(-1)^k}{a^{k+1}}(x-a)^k+O(x-a)^{N+1}\right) dx \\ &= \ln(a) + \sum_{k=0}^{N} \frac{(-1)^k}{(k+1)a^{k+1}}(x-a)^{k+1}+O(x-a)^{N+2}\,. \end{aligned}$$

Algebraically, the set of truncated power series (TPS) of order N forms an *integral domain*: The sum, difference, and product of TPS are TPS, but there are zero divisors, and not every element has a reciprocal—indeed, each element with a zero leading coefficient fails to have a TPS reciprocal. If we allow negative integer powers of $(x-a)$, then we have *truncated Laurent series*, which are indeed useful.

2.6.5 The Exponential of a Series

To find the series for e^x, we may introduce series reversion (see Exercise 2.24) or look at the differential equation

$$\frac{dy}{dx} = y, \qquad y(a) = e^a, \tag{2.94}$$

which is, of course, satisfied by e^x. Now, let y be given by

$$y = \sum_{k=0}^{N} y_k (x-a)^k + O(x-a)^{N+1} . \tag{2.95}$$

We have $y_0 = e^a$ and, by differentiation, we also have

$$\sum_{k=0}^{N} k y_k (x-a)^{k-1} + O(x-a)^N = \sum_{k=0}^{N} y_k (x-a)^k + O(x-a)^{N+1},$$

or, by rearranging the summation indices,

$$\sum_{k=0}^{N-1} (k+1) y_{k+1} (x-a)^k + O(x-a)^N = \sum_{k=0}^{N} y_k (x-a)^k + O(x-a)^{N+1} . \tag{2.96}$$

By the uniqueness of power series, we can identify the coefficients of corresponding powers, thereby obtaining the relation

$$(k+1) y_{k+1} = y_k , \qquad k = 0,1,2,3,\ldots,N-1 . \tag{2.97}$$

Using our initial condition and this recursive relation, we find that

$$\begin{aligned} y_1 &= y_0 = e^a \\ 2y_2 &= y_1 = e^a \\ 3y_3 &= y_2 = \frac{1}{2} e^a \\ 4y^4 &= y_3 = \frac{1}{6} e^a , \end{aligned}$$

and so on, so that the series for the exponential itself is

$$e^x = \sum_{k=0}^{N} \frac{e^a}{k!} (x-a)^k + O(x-a)^{N+1} . \tag{2.98}$$

However, that was really too easy for such a powerful trick. How about $y = e^u$, where

$$u = \sum_{k=0}^{N} u_k (x-a)^k + O(x-a)^{N+1}$$

instead? It still works! Let y be as in Eq. (2.95). Then, because

$$\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx} = y \frac{du}{dx} ,$$

we find that

$$\sum_{k=0}^{N-1}(k+1)y_{k+1}(x-a)^k+O(x-a)^N=\\ \left(\sum_{k=0}^{N}y_k(x-a)^k+O(x-a)^{N+1}\right)\left(\sum_{k=0}^{N}u_k(x-a)^k+O(x-a)^{N+1}\right).$$

Now, applying the Cauchy convolution rule to the right-hand side gives us

$$\sum_{k=0}^{N-1}(k+1)y_{k+1}(x-a)^k+O(x-a)^N=\sum_{k=0}^{N}c_k(x-a)^k+O(x-a)^{N+1}, \qquad (2.99)$$

where

$$c_k=\sum_{j=0}^{k}y_ju_{k-j}.$$

By the same method, we thus find the relation

$$(k+1)y_{k+1}=\sum_{j=0}^{k}y_ju_{k-j}. \qquad (2.100)$$

Also, it is obviously the case that the recurrence starts with $y_0=e^{u_0}$. This recurrence relation allows us to compute the exponential of any series. We will later solve differential equations with this technique.

Example 2.10. If $u(x)$ has the following series,

$$u=\frac{\sqrt{2}}{2}+\frac{\sqrt{2}}{2}\left(x-\frac{\pi}{4}\right)-\frac{\sqrt{2}}{4}\left(x-\frac{\pi}{4}\right)^2-\frac{\sqrt{2}}{12}\left(x-\frac{\pi}{4}\right)^3\\ +\frac{\sqrt{2}}{48}\left(x-\frac{\pi}{4}\right)^4+O\left(\left(x-\frac{\pi}{4}\right)^5\right),$$

then $\exp(u)$ has the series beginning

$$e^u=e^{\sqrt{2}/2}+\frac{1}{2}e^{\sqrt{2}/2}\sqrt{2}\left(x-\frac{\pi}{4}\right)+e^{\sqrt{2}/2}\left(\frac{1}{4}-\frac{\sqrt{2}}{4}\right)\left(x-\frac{\pi}{4}\right)^2\\ -e^{\sqrt{2}/2}\left(\frac{1}{4}+\frac{\sqrt{2}}{24}\right)\left(x-\frac{\pi}{4}\right)^3-e^{\sqrt{2}/2}\left(\frac{1}{96}+\frac{\sqrt{2}}{24}\right)\left(x-\frac{\pi}{4}\right)^4+O\left(\left(x-\frac{\pi}{4}\right)^5\right).$$

Again, MAPLE was used with its series command, which implements the algorithms discussed here. Specifically, once the series for u was defined, the command

```
series( exp(u), u=Pi/4 )
```

generated the above result. ◁

You may notice that *convergence* has not entered the discussion. Since we work only with truncated, finite power series, this is not a serious omission. Truncation error formulæ, on the other hand, are very useful, even if the series don't converge. You may be familiar with the Lagrange form of the remainder (that is, truncation error) for *real* Taylor series:

$$f(x) = f(a) + f'(a)(x-a) + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + R_{n+1}(x;a), \tag{2.101}$$

where

$$R_{n+1}(x;a) = \frac{f^{(n+1)}(a+\theta x)}{(n+1)!}(x-a)^{n+1}. \tag{2.102}$$

Here θ is some number between 0 and 1, which we don't know exactly. Knowledge of bounds on the $(n+1)$st derivative allows us to estimate how much accuracy we have in our real Taylor series when we truncate at n terms. This formula doesn't work over the complex numbers, however: Instead, we have (replacing x by z everywhere above)

$$R_{n+1}(z;a) = \frac{(z-a)^{n+1}}{2\pi i}\oint_C \frac{f(\zeta)}{(\zeta-a)^{n+1}(\zeta-z)}\,d\zeta. \tag{2.103}$$

Here C is a contour enclosing a and z. This integral can be interpreted as an "average value" of the $(n+1)$st derivative; in the complex plane, however, this average value is not always attained at some point $a+\theta z$. We will see a generalization of this formula to the case of interpolation error in Eq. (8.40) in Chap. 8.

We may also need to worry about whether the computed series is *well-conditioned* with respect to the data. Again this is taken up in Chap. 3.

2.7 A Partial Fraction Decomposition Algorithm Using Local Taylor Series

We return to the problem of computing the partial fraction decomposition of $^{p(z)}/_{q(z)}$, where $q(z)$ has been completely factored down to distinct linear (complex) factors $(z-\tau_k)^{s_k}$ for $0 \le k \le n$. We will need, first, the local Taylor series of p for each τ_k:

$$p(z) = p_{k,0} + p_{k,1}(z-\tau_k) + \cdots + p_{k,s_k-1}(z-\tau_k)^{s_k-1} + O(z-\tau_k)^{s_k}. \tag{2.104}$$

In other words, we need to reexpress $p(z)$ in each of the n local (i.e., shifted) monomial bases 1, $(z-\tau_k)$, $(z-\tau_k)^2$, …, $(z-\tau_k)^d$, except that we only need the first s_k coefficients in each case. This can be done using synthetic division, as discussed earlier. Assume that this has been done. Then, if

$$q(z) = \prod_{k=0}^{n} (z-\tau_k)^{s_k}\ , \tag{2.105}$$

then the rational function we wish to decompose into partial fractions, $p(z)/q(z)$, can be written as follows. We choose τ_0 as being special, for the moment, and let $w_0(z) = \prod_{k=1}^{n} (z-\tau_k)^{-s_k}$, which is analytic at τ_0 because all the τ_k are distinct by hypothesis (confluency is explicitly known). Thus, we can compute *its* local Taylor series by the methods of the previous section. In general, that is, not just for $k=0$, let

$$w_i(t) = \prod_{\substack{k=0\\k\neq i}}^{n} (t-\tau_k)^{-s_k}\,.$$

Then, we obtain the local Taylor series

$$w_i(t) = w_{i,0} + w_{i,1}(t-\tau_i) + \cdots = \sum_{\ell \geq 0} w_{i,\ell}(t-\tau_i)^{\ell}\,.$$

Then, observe that

$$\frac{p(z)}{q(z)} = \frac{p(z)}{\prod\limits_{k=0}^{n}(z-\tau_k)^{s_k}} = \frac{p(z)}{(z-\tau_i)^{s_i}\prod\limits_{\substack{k=0\\k\neq i}}^{n}(z-\tau_k)^{s_k}} = \frac{p(z)w_i(z)}{(z-\tau_i)^{s_i}}\,. \tag{2.106}$$

Now, this is exactly the form required for a partial fraction decomposition. As a result, the partial fraction decomposition we want may be obtained by Cauchy convolution with the local series for $p(z)$.

There are many ways to do this. The following is one method, and it has been implemented in MATLAB.[14] Begin by taking logarithms of $w_i(z)$:

$$\ln w_i(z) = \sum_{\substack{k=0\\k\neq i}}^{n} -s_k \ln(z-\tau_k) + \text{complex piecewise constants}\,. \tag{2.107}$$

When we take derivatives with respect to z, all the piecewise constants (multiples of $2\pi i$) disappear:

$$\frac{w_i'(z)}{w_i(z)} = \sum_{\substack{k=0\\k\neq i}} \frac{-s_k}{z-\tau_k}\,. \tag{2.108}$$

[14] The program is discussed in Chap. 8.

Note that

$$\frac{1}{z-\tau_k}=\frac{1}{\tau_i-\tau_k+z-\tau_i}$$

and so the summands in the right-hand sum of Eq. (2.108) can be expressed as follows:

$$\frac{-s_k}{z-\tau_k}=\frac{s_k}{\tau_k-\tau_i}\frac{1}{1-\frac{z-\tau_i}{\tau_k-\tau_i}}=\frac{s_k}{\tau_k-\tau_i}\sum_{\ell\geq 0}\left(\frac{z-\tau_i}{\tau_k-\tau_i}\right)^{\ell}.$$

Therefore,

$$\frac{w_i'(z)}{w_i(z)}=\sum_{\substack{k=0\\k\neq i}}^{n}\sum_{\ell\geq 0}\frac{s_k}{(\tau_k-\tau_i)^{\ell+1}}(z-\tau_i)^{\ell}=\sum_{\ell\geq 0}\left(\sum_{\substack{k=0\\k\neq i}}^{n}\frac{s_k}{(\tau_k-\tau_i)^{\ell+1}}\right)(z-\tau_i)^{\ell}. \quad (2.109)$$

If we define $u_{i,\ell}$ as follows,

$$u_{i,\ell}=\sum_{\substack{k=0\\k\neq i}}^{n}\frac{s_k}{(\tau_k-\tau_i)^{\ell+1}}, \quad (2.110)$$

then Eq. (2.109) becomes

$$\frac{w_i'(z)}{w_i(z)}=\sum_{\ell\geq 0}u_k(z-\tau_i)^{\ell}. \quad (2.111)$$

To simplify things a bit more, let $v_{i,m}={}^{w_{i,m}}\!/_{w_{i,0}}$ (so $v_{i,0}=1$) and note that

$$w_{i,0}=\prod_{\substack{k=0\\k\neq i}}^{n}(\tau_i-\tau_k)^{-s_k}.$$

Now, it follows that

$$w_i(z)=\sum_{m\geq 0}w_{i,m}(z-\tau_i)^m=w_{i,0}\sum_{m\geq 0}v_{i,m}(z-\tau_i)^m.$$

Taking derivatives (using $'$ to denote d/dz), we have

$$\begin{aligned}w_i'(z)&=w_{i,0}\sum_{m\geq 0}mv_{i,m}(z-\tau_i)^{m-1}=w_{i,0}\sum_{m\geq 1}mv_{i,m}(z-\tau_i)^{m-1}\\&=w_{i,0}\sum_{m\geq 0}(m+1)v_{i,m+1}(z-\tau_i)^m.\end{aligned}$$

Putting these in (2.111) and rearranging in order to more easily compare coefficients, we get the following:

$$\begin{aligned}\sum_{m\geq 0}(m+1)v_{i,m+1}(z-\tau_i)^m &= \left(\sum_{m\geq 0}v_{i,m}(z-\tau_i)^m\right)\left(\sum_{\ell\geq 0}u_{i,\ell}(z-\tau_i)^\ell\right)\\ &= \sum_{m\geq 0}c_m(z-\tau_i)^m,\end{aligned}$$

where

$$c_m = \sum_{\ell=0}^{m} v_{i,m-\ell}u_{i,\ell}$$

is Cauchy's convolution formula. Equating coefficients gives (remember $v_{i,0}=1$)

$$\begin{aligned}v_{i,m+1} &= \frac{1}{m+1}c_m\\ &= \frac{1}{m+1}\sum_{\ell=0}^{m} v_{i,m-\ell}u_{i,\ell}\,. \end{aligned}\tag{2.112}$$

Recall that (2.110) defines $u_{i,\ell}$. The recurrence relation (2.112) is the heart of the local Taylor series algorithm for partial fractions. Once we have the $v_{i,k}$, then we have the desired $\beta_{i,j}$.

Algorithm 2.4 Partial fraction decomposition by local Taylor series

```
Require: A positive integer $n$, a list of positive integers $s_k$, a list of $n$ distinct zeros $\tau_k$ of the
  denominator $q(z) = \prod_{k=0}^{n}(z-\tau_k)^{s_k}$, and the $n+1$ lists of local series coefficients $p_{k,j}$, $0 \leq j \leq$
  $s_k - 1$ of $p(z)$.
  for $i$=0:$n$ do
    for $j$=$i+1$:$n$ do
      $\Delta\tau_{i,j} = \tau_i - \tau_j$
    end for
    $v_{i,0} = 1$
    for $m$=0:$s_i - 1$ do
      $u_{i,m} = \sum_{k=0, k\neq i}^{n} \Delta\tau_{i,k}^{-m-1}$
      $v_{i,m+1} = \frac{1}{m+1}\sum_{k=0}^{m} u_{i,k}v_{i,m-k}$
    end for
    $\beta_i = \prod_{k=0, k\neq i}^{n}(\tau_i - \tau_k)^{-s_k}$
    for $m$=1:$s_i$ do
      $w_{i,m} = \beta_i v_{i,s_i-m}$
    end for
    for $m$=1:$s_i$ do
      $\alpha_{i,m} = \sum_{k=0}^{m} p_{i,k}w_{i,m-k}$
    end for
  end for
  return The coefficients $\alpha_{k,j}$ in the partial fraction decomposition
```

$$\frac{p(z)}{\prod_{k=0}^{n}(z-\tau_k)^{s_k}} = \sum_{k=0}^{n}\sum_{j=0}^{s_k-1}\frac{\alpha_{k,j}}{(z-\tau_k)^{j+1}}\tag{2.113}$$

Algorithm 2.4 has been implemented in the MATLAB program `genbarywts` and in the MAPLE program `BHIP`, for the case where the numerator is just 1. You will be asked to show in the exercises that this algorithm costs $O(d^2)$ flops, when proper care is taken to avoid redundancy. In the case when all $s_k = 1$, the algorithm reduces merely to the computation of β_i for $1 \le i \le n$. In that case, the computation was proved to be numerically stable by Higham (2004). If any $s_k > 1$, then the algorithm is *not* backward stable, in the case when the nodes τ_k are symmetric about zero (for example) and some of the partial fraction decomposition coefficients $\alpha_{i,j}$ are exactly zero. However, the algorithm is stable *enough* for many purposes.

Example 2.11. Suppose the nodes τ_k are the Chebyshev–Lobatto nodes $\tau_k = \cos(\pi k/n)$ for $0 \le k \le n$. Take first the case $n = 5$, and execute this code:

```
tau = cos( pi*k/n );
[w,D] = genbarywts( tau, 1 )
```

It returns the values

$$w = 1.6000, -3.2000, 3.2000, -3.2000, 3.2000, -1.6000\,.$$

By comparison with MAPLE, these answers are correct up to $O(\mu_M)$. When $n = 50$, the numbers are larger, $O(10^{13})$, but still have relative forward error only about 2×10^{-14}.

When we make each node have confluency $s = 2$, the situation changes a bit, but not much, for $n = 5$:

$$\alpha = \begin{bmatrix} -43.520 & 2.5600 \\ 23.978 & 10.240 \\ 3.4984 & 10.240 \\ -3.4984 & 10.240 \\ -23.978 & 10.240 \\ 43.520 & 2.5600 \end{bmatrix},$$

and again the forward error is $O(\mu_M)$. For $n = 50$, this is again true. We detect no instability in this example. For higher confluencies, we expect more trouble. ◁

2.8 Asymptotic Series in Scientific Computation

Niels Henrik Abel (1802–1829) wrote

> The divergent series are the invention of the devil, and it is a shame to base on them any demonstration whatsoever. By using them, one may draw any conclusion [s]he pleases and that is why these series have produced so many fallacies and paradoxes [...]. (cited in Hoffman 1998 p. 218)

Nowadays, the attitude is different, and closer to what Heaviside meant when he said

> The series is divergent; therefore we may be able to do something with it. (cited in Hardy 1949)

In fact, asymptotic series will be used a lot in this book, and we will often not care too much whether they converge. This is because, in many contexts, the first few terms contain all the numerical information one needs; there's no need to ponder on what happens in the tail end of the series.

The key to understanding asymptotic series is to realize that there are *two* limits to choose from, with a series. Suppose we have, for example,

$$f(z) = \sum_{k=0}^{N} \frac{f^{(k)}(a)}{k!}(z-a)^k \quad + \quad R_N(z)(z-a)^{N+1}\,, \tag{2.114}$$

as the usual truncated Taylor series for $f(z)$ near $z=a$. We can take the first limit, $N \to \infty$, to get the familiar mathematical object of the *infinite series*. This only makes sense if the limit exists. (There is some freedom to alter the definition of limit that we use in this case; we do not pursue this here.) If that limit exists, we say the series is *convergent*. However, there is another limit to be considered here, which often leads to very useful results. Namely, do *not* let $N \to \infty$, but rather keep it fixed (perhaps even at $N=1$ or $N=2$). Instead, consider the limit as $z \to a$. Even if the series is divergent in the first sense, this second limit often gives enormously useful information, typically because $R_N(z)$ (as it is written above) is well behaved near $z=a$, and so the term $(z-a)^{N+1}$ ensures that the remainder term vanishes more quickly than do the terms that are kept. The rest of this section explores that simple idea.

We often want to consider the behavior of a function $y(x)$ in the presence of some perturbations. Then, instead of studying the original function $y(x)$, we study the asymptotic behavior of a two-parameter function $y(x,\varepsilon)$, where ε is considered "small." An asymptotic expansion for the function $y(x,\varepsilon)$ has the form

$$y(x,\varepsilon) = y_0(x)\phi_0(\varepsilon) + y_1(x)\phi_1(\varepsilon) + y_2(x)\phi_2(\varepsilon) + \ldots = \sum_{k=0}^{\infty} y_k(x)\phi_k(\varepsilon)\,, \tag{2.115}$$

where $\phi_k(\varepsilon)$ are referred to as *gauge functions*; that is, they are a sequence of functions $\{\phi_k(\varepsilon)\}$ such that, for all k,

$$\lim_{\varepsilon \to 0} \frac{\phi_{k+1}(\varepsilon)}{\phi_k(\varepsilon)} = 0\,.$$

The type of gauge function we will use the most often is the power of the perturbation ε, namely, $\phi_k(\varepsilon) = \varepsilon^k$, in which case we simply have a formal power series:

$$y(x,\varepsilon) = y_0(x) + y_1(x)\varepsilon + y_2(x)\varepsilon^2 + \ldots = \sum_{k=0}^{\infty} y_k(x)\varepsilon^k.$$

We then have to solve for the $y_k(x)$, $k = 0, 1, \ldots, N$. To find the first coefficient $y_0(x)$, divide Eq. (2.115) by $\phi_0(\varepsilon)$, and then take the limit as $\varepsilon \to 0$:

$$\begin{aligned}
\frac{y(x,\varepsilon)}{\phi_0(\varepsilon)} &= y_0(x) + \frac{1}{\phi_0(\varepsilon)} \sum_{k=1}^{\infty} y_k(x)\phi_k(\varepsilon) \\
\lim_{\varepsilon\to 0} \frac{y(x,\varepsilon)}{\phi_0(\varepsilon)} &= y_0(x)\,.
\end{aligned}$$

All the higher-order terms vanish since $\phi_k(\varepsilon)$ is a gauge function. This gives us $y_0(x)$. Now, subtract $y_0(x)\phi_0(\varepsilon)$ from both sides in Eq. (2.115); we then divide both sides by $\phi_1(\varepsilon)$ and take the limit as $\varepsilon \to 0$:

$$\frac{y(x,\varepsilon) - y_0(x)\phi_0(\varepsilon)}{\phi_1(\varepsilon)} = y_1(x) + \frac{1}{\phi_1(\varepsilon)} \sum_{k=2}^{\infty} y_k(x)\phi_k(\varepsilon),$$

so

$$\lim_{\varepsilon\to 0} \frac{y(x,\varepsilon) - y_0(x)\phi_0(\varepsilon)}{\phi_1(\varepsilon)} = y_1(x)\,. \tag{2.116}$$

As we see, we will in general have

$$y_k(x) = \lim_{\varepsilon\to 0} \frac{1}{\phi_k(\varepsilon)} \left(y(x,\varepsilon) - \sum_{\ell=0}^{k-1} y_\ell(x)\phi_\ell(\varepsilon) \right)\,. \tag{2.117}$$

Convergence of a series is all about the tail, which requires an infinite amount of work. What we want instead is gauge functions that go to zero very fast; in other words, the speed at which they go to zero is asymptotically faster from one term to the next.

Example 2.12. Consider the (convergent) integral and the (divergent) asymptotic series

$$\int_0^{\infty} \frac{e^{-t}}{1+xt} dt = \sum_{k=0}^{n} (-1)^k k! x^k + O(x^{n+1})\,.$$

One can discover that series by replacing $1/(1+xt)$ with the finite sum $1 - xt + x^2t^2 + \cdots (xt)^n + \frac{(-xt)^{n+1}}{(1+xt)}$, giving

$$\int_0^{\infty} \frac{e^{-t}}{1+xt} dt = \sum_{k=0}^{n} (-1)^k x^k \int_0^{\infty} t^k e^{-t}\, dt + (-1)^{n+1} x^{n+1} \int_0^{\infty} \frac{t^{k+1} e^{-t}}{1+xt}\, dt\,.$$

This provides a perfectly definite meaning to each of the entries in the asymptotic series. Notice that the series diverges for any $x \neq 0$, if we take the limit as $n \to \infty$.

Nonetheless, taking (say) $n = 5$ allows us to evaluate the integral perfectly accurately for small enough x, say $x = 0.03$: summing the six terms gives 0.9716545240, whereas the exact value (found by the methods of Chap. 10) begins 0.9716549596, which differs by about $5 \cdot 10^{-7}$. ◁

Remark 2.16. In the previous example, we have used a *divergent* series to give us a good approximation to the correct answer. Heaviside was right, and asymptotic series are extremely useful in numerical analysis. The reason this works is that it is the limit as $x \to 0$ that is dominating here: If we had wanted an accurate answer for $x = 10$, we would have been out of luck. We will often be concerned with the asymptotics of the error as the *average mesh width* (call it h) goes to zero, for example, and methods will be designed to be accurate in that limit. ◁

2.9 Chebyshev Series and Chebfun

This generalized review chapter is not the right place to begin explaining the underlying methods of the Chebfun package. Here we mention only that the package does not use Taylor series, but rather *interpolation at Chebyshev points* (we expand on this in Chap. 8), which is closely related to *Chebyshev series*: One can convert back and forth using the FFT, in a stable and efficient fashion (see Chap. 9). What, then, are Chebyshev series? Just as with Taylor series, one can find convergent series for elementary functions, but where now the gauge functions are not shifted monomials but rather Chebyshev polynomials[15]; for example,

$$e^x = J_0(i)T_0(x) + 2\sum_{k=1}^{\infty} i^k J_k(-i)T_k(x)\,. \tag{2.118}$$

The coefficients are evaluations of the Bessel functions $J_k(z)$ at particular arguments (complex arguments, as it happens, although the results are real). This series is not expressed in powers of x or of $x - a$, but rather in higher- and higher-degree Chebyshev polynomials. One could do this on other intervals by the linear transformation $x = {}^{2(t-a)}\!/\!_{(b-a)} - 1$, so $a \le t \le b$; the coefficients would be different, of course. When one has evaluated $J_0(i) \approx 1.266\ldots$ and $J_k(-i)$ for several k, this series (and series like this) can provide a quite effective method for evaluating the function under consideration. See, for example, Boyd (2002) for applications to computing zeros of functions. For example, taking the first 15 terms here gives us

$$\begin{aligned} e^x &= 1.26606587775201 T_0(x) + 1.13031820798497\, T_1(x) \\ &\quad + 0.271495339534077\, T_2(x) + 0.0443368498486638\, T_3(x) \\ &\quad + 0.00547424044209373\, T_4(x) + 0.000542926311913944\, T_5(x) \\ &\quad + 0.0000449773229542951\, T_6(x) + 0.00000319843646240199\, T_7(x) \end{aligned}$$

[15] See Rivlin (1990 Chapter 3).

$$
\begin{aligned}
&+ 0.0000001992124806672 80\,T_8(x) + 0.00000001103677172551 73\,T_9(x)\\
&+ 0.000000000550589607967375\,T_{10}(x) + 2.49795661698498 \times 10^{-11}\,T_{11}(x)\\
&+ 1.03915223067857 \times 10^{-12}\,T_{12}(x) + 3.99126335641440 \times 10^{-14}\,T_{13}(x)\\
&+ 1.42375801082566 \times 10^{-15}\,T_{14}(x) \qquad (2.119)
\end{aligned}
$$

and this approximation has the relative error (on the interval $-1 \le x \le 1$) shown in Fig. 2.8. We will pursue this concept further in later chapters. For now, note that

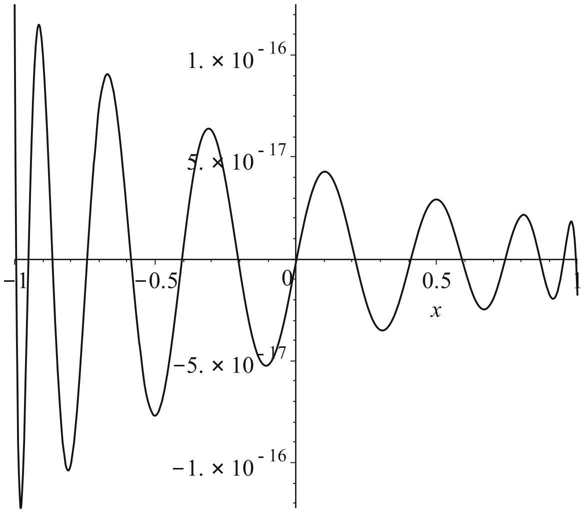


Fig. 2.8 The relative error $S \cdot \exp(-x) - 1$ in the truncated Chebyshev series (2.119), computed in high precision in MAPLE

$|T_k(x)| \le 1$, and so the size of the coefficients tells us directly how much each term contributes (at most) to the sum.

In Chebfun itself, this series can be computed as follows, assuming the Chebfun package has been installed.

```
x = chebfun('x',[-1,1]);
y = exp(x);
co = chebpoly(y);
format long e
co(end:-1:1)'
% ans =
%
%     1.266065877752008e+000
%     1.130318207984970e+000
%     2.714953395340767e-001
%     4.433684984866388e-002
%     5.474240442093829e-003
%     5.429263119140007e-004
%     4.497732295430233e-005
%     3.198436462443460e-006
%     1.992124806757106e-007
%     1.103677179109604e-008
%     5.505895456820691e-010
%     2.497954620928056e-011
%     1.039121170062377e-012
%     4.003147020219850e-014
%     1.395708945243054e-015
%
t = linspace(-1,1,3011);
reler = exp(-t).*y(t)-1;
plot( t, reler, 'k-' );
set(gca, 'fontsize', 16);
axis([-1,1,-1.5E-15,1.5E-15]);
set(gca, 'YTick', -1.5E-15:5E-16:1.5E-15);
```

As you can see, the numbers do not quite match (although the largest three do): The series at the beginning of this section was computed using MAPLE in 60 digits of precision, and then the coefficients were rounded to 15 digits. They are different from the Chebfun series coefficients printed above, but not in any important way because the differences in the smallest coefficient (which are the greatest, relatively speaking) matter the least to the sum. The source of the difference is not a numerical error, but rather a difference in type of approximation. We will return to this later, but for now, observe that Chebfun is doing what it is supposed to—something similar to the Chebyshev series above, but not exactly the same thing. As we see in Fig. 2.9, it produces a perfectly acceptable (and even somewhat similar, ignoring the discrete-level effect of being so close to unit roundoff) error curve.

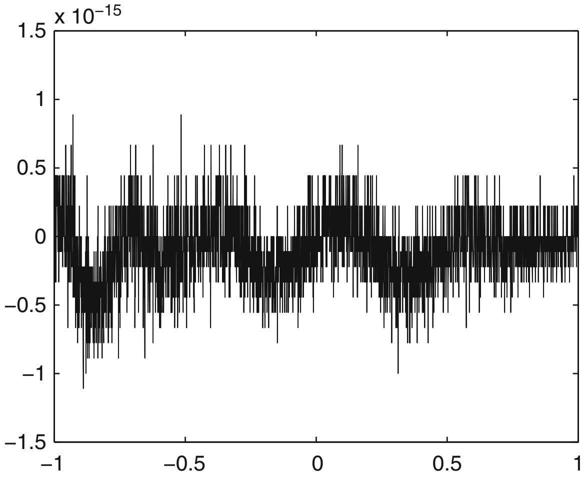


Fig. 2.9 The relative error $y \cdot \exp(-x) - 1$ in the `chebfun` for $y = \exp(x)$, as computed in normal precision in MATLAB

2.10 Notes and References

This chapter was originally intended for self-study, although the importance of the material suggests that it should be more formally included in any course using this book. A more elementary introduction to the theory of univariate polynomials can be found in Barbeau (2003). A more detailed introduction to the computation of Taylor series can be found in Henrici (1974). For a statement and discussion of the fundamental theorem of algebra, see, for instance, Levinson and Redheffer (1970).

Variations of Theorem 2.9 can be found throughout the literature, so many that Stetter (1999) says that it is misleading to cite any; the paper (Rezvani and Corless 2005) points out that it is really just an application of Hölder's inequality.

Algorithm 2.1, our version of the synthetic division algorithm, is an adaptation of Algorithm 5.2 in Higham (2002 p. 96). We modify that algorithm here to return the local Taylor coefficients, that is, $p^{(k)}(a)/k!$ instead of multiplying by factorials as done there to return values of the derivatives $p^{(k)}(a)$.

The special case $\tau_k = -k$ or $\tau_k = k$ for $0 \leq k \leq n-1$ of Newton polynomials is useful in combinatorics and is sometimes called the Pochhammer basis. We have already seen this, but called it $z^{\underline{k}}$, z to the k falling.

For a thorough treatment of Chebyshev polynomials, see Rivlin (1990). See Salzer (1972) for more discussion of useful properties of the Chebyshev–Lobatto points η_k. For a discussion of Chebfun and Chebyshev polynomials, see Battles and Trefethen (2004) and `http://www2.maths.ox.ac.uk/chebfun/`.

Some other orthogonal bases are discussed in the venerable book (Abramowitz and Stegun 1972). That book has been substantially revised to become the Digital Library of Mathematical Functions from the National Institute of Standards and Technology (`http://dlmf.nist.gov/`). A similar INRIA project, the Dynamic Dictionary of Mathematical Functions, may be found at `http://ddmf.msr-inria.inria.fr/1.6/ddmf`. More details on many orthogonal polynomials can be found in Andrews et al. (1999), and some important algorithms in Wilf (1962), available for free for educational purposes from `http://www.math.upenn.edu/~wilf/website/Mathematics_for_the_Physical_Sciences.html`. A discussion of MAPLE's methods for othogonal series can be found in Rebillard (1997) and Ronveaux and Rebillard (2002).

Faster methods of partial fraction decomposition than the one advocated here are certainly available: Kung and Tong (1977) and Chin (1977) use FFT methods, which we believe are unstable because they implicitly convert to the monomial basis; the divided-difference algorithm of Schneider and Werner (1991), which, although not asymptotically fast, is twice as fast as the algorithm given here, has an unhelpful dependence on node ordering and again can produce $\beta_{i,j}$ that do not accurately reproduce 1. Their method is much more stable than methods that convert to the monomial basis, however.

Problems

Theory and Practice

2.1. Prove Theorem 2.1 on page 44.

2.2. Prove Theorem 2.2 on page 45.

2.3. Show that the roots of $z^n - 1 = 0$ are $z_k = \exp(2\pi ik/n)$.

2.4. Show that the roots of $T_n(z)$ are

$$\xi_k = \cos\left(\frac{\pi(2k-1)}{2n}\right), \qquad 1 \le k \le n\,.$$

2.5. Consider the polynomial $p(x) = x^3 - 2x - 5$. Plot this on $0 \le x \le 4$ and see thereby that there is a root near $x = 2$. Shift the basis to 1, $(x-2)$, $(x-2)^2$, and $(x-2)^3$. By neglecting all but the first two terms, get an improved approximate root. Shift the basis again to 1, $(x-r)$, $(x-r)^2$, and $(x-r)^3$, where r is your estimated root. Neglect all but the first two terms again, and solve to get an even more improved root. Repeat the process until you have identified the root to machine accuracy. As discussed in the text, this is how Newton conceived of what we now call Newton's method.

2.6. Prove the discrete orthogonality of the Chebyshev polynomials on the zeros of $T_n(x)$, and show thereby that Eq. (2.31) gives the Chebyshev coefficients of a given $p(x)$ with degree at most $n-1$.

2.7. Download and install the Chebfun package. Execute the following commands.

```
close all
plot( chebpoly(0), 'k' );
hold on
for i=1:30,
  plot( chebpoly(i), 'k' );
end;
axis('square')
```

Explain what you see. The discussion by Rivlin (1990) is very extensive, but for this problem a simple description will suffice (this problem is more about syntax than anything).

2.8. Use the change of variables $x = \cos\theta$ to show that

$$\int T_n(x)\,dx = \frac{1}{2(n+1)}T_{n+1}(x) - \frac{1}{2(n-1)}T_{n-1}(x) + C$$

for $n \ge 1$, for some constant C. Since $\int T_0(x)\,dx = T_1(x) + C$, this gives a beautiful short formula for integrals of Chebyshev polynomials. The formula for derivatives of T_n, expressed in terms of lower-degree Chebyshev T polynomials, is not so elegant, but useful nonetheless. It is found in Rivlin (1990), and also as an MAPLE program in Corless (2002).

2.9. Show that the Chebyshev–Lobatto points $\eta_k^{(n)} = \cos({k\pi}/{n})$ are the zeros of the polynomial

$$(1-x^2)\frac{\sin n\theta}{\sin\theta}\,,$$

where $x = \cos\theta$. The polynomial $U_{n-1}(x) := {\sin n\theta}/{n\sin\theta}$ (note the extra n in the denominator) is called the Chebyshev polynomial of the second kind, and the η_k are sometimes called the Chebyshev points of the second kind.

2.10. Show that the Chebyshev–Lobatto points $\eta_k^{(n)} = \cos({k\pi}/{n})$ are (also) the zeros of the monic polynomial

$$w(x) = \prod_{k=0}^{n}\left(x-\eta_k^{(n)}\right) = 2^{-n}\left(T_{n+1}(x) - T_{n-1}(x)\right)$$

if $n \geq 1$. A discussion of this result can be found in Trefethen (2013).

2.11. Show that Horner's method recursively applied to $p(z) = p(\tau_k) + q(z)(z-\tau_k)$ gives Algorithm 2.1.

2.12. Prove Lemma 2.1 on page 61.

2.13. Show that for every pair of sets of polynomial bases $\phi_k(x)$ and $\psi_k(x)$, $0 \leq k \leq n$, there exists a nonsingular matrix $\mathbf{A}_{\psi\phi}$ for which

$$\left[\psi_0(x), \psi_1(x), \ldots, \psi_n(x)\right] = \left[\phi_0(x), \phi_1(x), \ldots, \phi_n(x)\right]\mathbf{A}_{\psi\phi}\,.$$

Show $\mathbf{A}_{\phi\psi} = \mathbf{A}_{\psi\phi}^{-1}$.

2.14. Show that Algorithm 2.4 costs $O(d^2)$ flops to execute. Discuss the varying cases when all s_k are small and the opposite case when only one or two nodes have high confluency.

2.15. Plot the condition numbers for evaluating the scaled Wilkinson polynomial

$$W_{20}(x) = \prod_{k=1}^{20}(x - \frac{k}{21})\,,$$

in each of the following bases:

1. monomial basis $\phi_k(x) = x^k$;
2. Bernstein–Bézier on $[0,1]$,

$$\phi_k(x) = \binom{20}{k}x^k(1-x)^{20-k};$$

3. Lagrange basis on $\tau_k = \frac{k}{21}$, $0 \le k \le 20$;
4. Lagrange basis on random nodes τ_k chosen from a uniform distribution on $[0,1]$.

2.16. Consider the Taylor series for the Airy function $\mathrm{Ai}(z)$. Think about each term as $a_k z^k$, for $0 \le k \le 127$. For $z = 10$, at which a 127-degree Taylor polynomial *ought* to give an accurate answer, compute all 128 of these terms separately, and plot them on a graph, with k on the horizontal axis. Verify that the largest term occurs at about $k = 30$, and has size about 10^7. This picture is why the phenomenon is known as "the hump." What does this have to do with our condition number analysis in the text?

2.17. Compute $e^x = \sum_{k=0}^{\infty} x^k/k!$ for various values of x and truncate the series at various values of N. Does this series converge, in theory? Why, then, does it compute (say) $\exp(-30)$ to such poor relative accuracy? Compare with Problems 2.16 and 1.7.

2.18. In this problem, we examine exact formulæ for finding zeros of polynomials of low degree. To begin with, the zero polynomial $f_0(z) \equiv 0$ is exceptional; with $\deg f_0(z) = -\infty$, it is zero no matter what z is. Also, degree-0 polynomials, of the form $f_0(z) = a_0$, $a_0 \neq 0$, are never zero; they have no roots. Moreover, degree-1 polynomials, of the form $f_1(z) = a_0 + a_1 z$, $a_1 \neq 0$, have one root, which is given by $z = -a_0/a_1$. Notice that, as $a_1 \to 0$, unless $a_0 = 0$, this root goes to (complex) infinity. These are very straightforward cases. Degree-2 polynomials are already more interesting:

1. Show that $f_2(z) = a_0 + a_1 z + a_2 z^2$ with $a_2 \neq 0$ may be written as
$$f_2(z) = a_2\left(z + \frac{a_1}{2a_2}\right)^2 - \frac{1}{4a_2}\left(a_1^2 - 4a_2a_0\right)$$
(because $a_2 \neq 0$), and that therefore the two roots of $f_2(z)$ are
$$z = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_2a_0}}{2a_2},$$
and that as $a_2 \to 0$, if $a_1 \neq 0$ that one root tends to $-a_0/a_1$ and the other tends to ∞. If $a_1 = 0$, then both roots tend to ∞ (remember: $a_k \in \mathbb{C}$).
2. What is the absolute condition number of the roots?

Now, let us turn to degree-3 polynomials and, in particular, Cardano's method:

1. Consider a third-degree polynomial $f_3(z) = a_0 + a_1 z + a_2 z^2 + a_3 z^3 = 0$. Show that, using $z = t - a_2/3a_3$, this is equivalent to solving $t^3 + pt + q = 0$.
2. Let $t = u + v$ and gather terms so that
$$u^3 + v^3 + q + (3uv + p)(u + v) = 0\,.$$

Conclude that if one can find u and v such that

$$\begin{aligned} 3uv + p &= 0 \\ u^3 + v^3 + q &= 0 \end{aligned}$$

simultaneously, then one can solve any cubic equation.

3. By solving $u^3v^3 = -p^3/27$ and $u^3 + v^3 = -q$ simultaneously for, say, u^3 first, and then using $uv = -p/3$ to find v unambiguously, show that you really can solve cubics.
4. The "condition numbers" $\frac{\partial x}{\partial a_k}$, $\left[\frac{\partial t}{\partial p}, \frac{\partial t}{\partial q}\right]$ and $\left[\frac{\partial u}{\partial p}, \frac{\partial u}{\partial q}, \frac{\partial v}{\partial p}, \frac{\partial v}{\partial q}\right]$ are different but related. Discuss. In particular, is the use of Cardano's formula always numerically stable?

Finally, let's have a look at degree-4 polynomials (encountered in quartic equations), and in particular Descartes' method:

1. Convert $f(z) = a_0 + a_1z + a_2z^2 + a_3z^3 + a_4z^4 = 0$, with $a_4 \neq 0$, to $F(t) = t^4 + pt^2 + qt + r = 0$.
2. Show that if

$$\begin{aligned} v + w - u^2 &= p \\ u(w - v) &= q \\ vw &= r\,, \end{aligned}$$

then $F(t) = (t^2 + ut + v)(t^2 - ut + u)$. Eliminate v and w to find a cubic equation for u^2.

3. Discuss the conditioning of the transformed problems.

2.19. Show that if

$$f(z) = \sum_{k=0}^{n} c_k \phi_k(z) \qquad \text{and} \qquad (f + \Delta f)(z) = \sum_{k=0}^{n} c_k(1 + \delta_k)\phi_k(z)$$

with $|\delta_k| \leq \varepsilon w_k$, that for each simple root $\hat{z}$ of f, when $\varepsilon > 0$ is small enough, that there is a simple root $\tilde{z}$ of $f + \Delta f$ such that $\tilde{z} = \hat{z} + \Delta z$ and

$$|\Delta z| \leq \frac{\varepsilon B(\hat{z})}{|f'(\hat{z})|} + O(\varepsilon^2)\,,$$

where $B(z) = \sum_{k=0}^{n} w_k |\phi_k(z)|$.

2.20. Find a recurrence relation for the series coefficients of $y = \ln u$ if

$$u = \sum_{k=0}^{N} u_k (x - a)^k + O(x - a)^{N+1}$$

and $u_0 \neq 0$.

2.21. Find a recurrence relation for the series coefficients of s and c, where $s = \sin(u)$ and $c = \cos(u)$, if

$$u = \sum_{k=0}^{N} u_k(x-a)^k + O(x-a)^{N+1}\,.$$

2.22. The JCP Miller formula. If $y = u^\alpha$ (for constant α), find a recurrence relation for the series coefficients of y by use of ${}^{dy}\!/\!{}_{dx}$, where

$$u = \sum_{k=0}^{N} u_k(x-a)^k + O(x-a)^{N+1}\,.$$

2.23. The Airy function $\mathrm{Ai}(z)$ satisfies the differential equation

$$\frac{d^2y}{dz^2} = zy(z) \tag{2.120}$$

with the initial conditions $y(0) = 3^{1/3}/(3\Gamma(2/3))$ and $y'(0) = -3^{1/6}\Gamma(2/3)/(2\pi)$. Use the methods of this section to generate a recurrence relation that determines the Taylor coefficients of $\mathrm{Ai}(z)$ in its series about 0. (That recurrence is used in the programs in Exercise 2.29.) As for the exponential, sine and cosine, and logarithm, this can be extended to allow you to generate recurrence relations for the series coefficients of $\mathrm{Ai}(u(z))$, where $u(z)$ is known by a truncated power series.

2.24. This problem is on series reversion. Suppose that

$$x = x_0 + x_1(y-y_0) + x_2(y-y_0)^2 + \ldots\,,$$

that we know the x_k, and that $x_1 \neq 0$. We wish to find the coefficients y_k so that

$$y = y_0 + y_1(x-x_0) + y_2(x-x_0)^2 + \ldots\,.$$

Proceed as follows. Take

$$x = x_0 + x_1(y_1(x-x_0) + y_2(x-x_0)^2 + \ldots) + x_2(y_1(x-x_0) + y_2(x-x_0)^2 + \ldots)^2 \\ + x_3(y_1(x-x_0) + \ldots)^3 + \ldots,$$

expand, and solve for y_1, y_2 and y_3 in turn. This is the brute force approach. See Henrici (1974) for a discussion of the Lagrange–Bürmann theorem, which explores an elegant connection (perhaps originally due to Lambert, if we are to believe his claims about his *Acta Helvetica* paper) to powers of the series being reverted.

2.25. Using your answer: For Problem 2.24, find the first three terms of the series for $\tan(x)$ about $x = 0$ from the series for $\arctan(x) = \int_0^x \frac{dt}{1+t^2}$.

2.26. Let $f(x,y,t) = 0$ with $x = x(t), y = y(t)$, $g(x,y,t) = 0$, and

$$f(x_0,y_0,t_0) = g(x_0,y_0,t_0) = 0\,.$$

Show that if the Jacobian determinant

$$\det\begin{bmatrix} f_x(x_0,y_0,t_0) & f_y(x_0,y_0,t_0) \\ g_x(x_0,y_0,t_0) & g_y(x_0,y_0,t_0) \end{bmatrix} \tag{2.121}$$

is not zero, and f and g are analytic in all variables, then $x(t) = x_0 + x_1(t-t_0) + \ldots$ and $y(t) = y_0 + y_1(t-t_0) + \ldots$ may be constructively developed in Taylor series to as many terms as one likes. (Hint: differentiate. This is the implicit function theorem.)

2.27. If you have access to MAPLE, solve

$$\begin{aligned} x^2 + y^2 &= t^2 \\ 25xy - 12 &= 0 \end{aligned}$$

in series for x and y near $t = 1$, when $x = 3/5$ and $y = 4/5$ (there are three other intersections also; just follow this one). (Hint: You can `dsolve/series`, which implements the ideas of this chapter, but differentiate first.)

2.28. If you have access to MAPLE, consider the command

```
convert( R, parfrac, z, true );
```

when R is a simple factored rational function, say $\prod_{i=0}^{n}(x-\tau_i)^{-1}$, with (say) Chebyshev–Lobatto nodes computed to 16 digits of precision, via commands similar to

```
Digits := 16;
n := 5;
tau := [ seq( evalf( cos(Pi*j/n) ), j = 0..n ) ];
R := 1/mul( z-tau[1+j], j = 0..n);
PF := convert( R, parfrac, z, true );
ONE := PF/R;
plots[logplot]( abs(ONE-1), z = -1 .. 1, colour = BLACK, style=
    POINT );
```

Try these commands for various n. How well does MAPLE do, in your version? At this time of writing, MAPLE 15 makes acceptable plots for n as large as 15, but it's already bad for $n = 20$.

Investigations and Projects

2.29. The following MAPLE program uses a handwritten version of Horner's method to evaluate the degree-N Taylor polynomial at $z = 0$ for the Airy function $\mathrm{Ai}(z)$.

```
#
# Horner form of Taylor polynomial approximation to AiryAi(z)
#
TaylorAi := proc( z, N )
  local Ai0, Aip0, f1, f2, k, n, z3, zsq;
  Ai0  := evalf( 3^(1/3)/(3*GAMMA(2/3)) );
  Aip0 := evalf( -3^(1/6)*GAMMA(2/3)/(2*Pi) );
  z3   := evalf( z*z*z );
  n    := max( floor( (N-2)/3 ), 0 );
  f1 := 1;
  f2 := 1;
  for k from n by -1 to 1 do
     f1 := evalf( 1 + z3*f1/((3*k)*(3*k-1)) );
     f2 := evalf( 1 + z3*f2/((3*k+1)*(3*k)) );
  end do;
  return Ai0*f1 + Aip0*z*f2
end;
```

When this is translated into MATLAB via the CodeGeneration[Matlab] feature of MAPLE, and the resulting code is polished a bit by hand, the result is

```
%
% Automatic translation of TaylorAi.mpl
% which was written by RMC 2011, using
% CodeGeneration[Matlab] in Maple
% plus fixups GAMMA --> gamma
%             vectorized multiplications
%             added "end" to function
function TaylorAireturn = TaylorAi( z, N )
  Ai0 = ((3 ^ (0.1e1 / 0.3e1) / gamma(0.2e1 / 0.3e1)) / 0.3e1);
  Aip0 = (-(3 ^ (0.1e1 / 0.6e1)) * gamma(0.2e1 / 0.3e1) / pi /
      0.2e1);
  z3 = (z .* z .* z);
  n = max( floor(N / 0.3e1 - 0.2e1 / 0.3e1), 0 );
  f1 = 1;
  f2 = 1;
  for k = n:-1:1
    f1 = (0.1e1 + z3 .* f1 / k / (3 * k - 1) / 0.3e1);
    f2 = (0.1e1 + z3 .* f2 / (3 * k + 1) / k / 0.3e1);
  end
  TaylorAireturn = Ai0 * f1 + Aip0 * z .* f2;
end
```

The automatically generated and curiously ugly `0.3e1` meaning `3.0`, and its ilk, were left as-is. Notice also that this automatically generated code does not follow the MATLAB style guidelines of Johnson (2010). The following MATLAB commands

```
z = linspace( -13, 13, 40012 );
y = TaylorAi( z, 127 );
relerr = y./airy(z) - 1;
semilogy( z, abs(relerr), 'k.' )
xlabel( 'z' ), ylabel( 'relative␣error' )
axis([-15 15 10E-21 10E14]);
set(gca, 'YTick', [10.^-20, 10.^-15, 10.^-10, 10.^-5, 10.^0,
    10.^5, 10.^10, 10.^15]);
```

produce the plot in Fig. 2.10. Explain this plot in general. Can you explain the curious horizontal line starting at about $z = 7$? If you have access to MAPLE, you might consider running the original program at varying levels of precision, say `Digits` equal to 5, 10, 15, 20, and 25, in order to help.

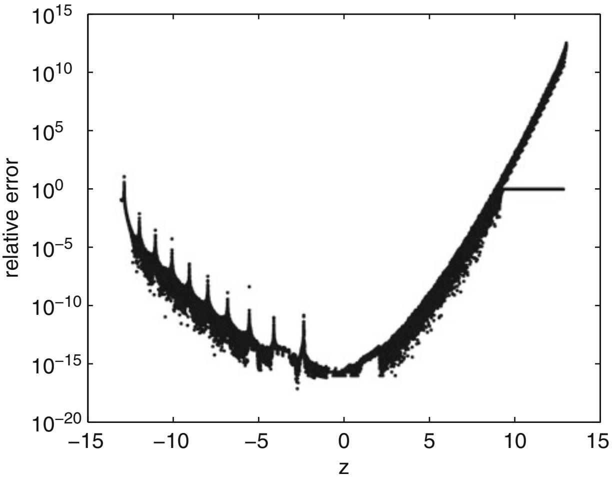


Fig. 2.10 The output of the program in Problem 2.29

2.30. Prove that you may algorithmically compute the Taylor coefficients of any function or set of functions defined by a system of polynomial or rational differential equations such as this:

$$\frac{dy_1}{dx} = f_1(x, y_1, y_2, \ldots, y_n) \tag{2.122}$$

$$\frac{dy_2}{dx} = f_1(x, y_1, y_2, \ldots, y_n) \tag{2.123}$$

$$\vdots$$

$$\frac{dy_n}{dx} = f_1(x, y_1, y_2, \ldots, y_n) \tag{2.124}$$

with $\mathbf{y}(a) = y_a$ given, and each f_i polynomial or rational in its arguments (with no poles at a, $\mathbf{y}(a)$ in any f_i). This is quite a large class of functions!

2.31. We know of one function, the Γ function (and its derivatives), that does not fall into the class of functions in Problem 2.30. Can you think of any others? Describe some.

2.32. Draw the pseudozero sets for the following polynomials as in Fig. 2.7. Choose interesting contour levels. Use weights equal to the polynomial coefficients.

1. $T_{20}(x)$. Compare with the Wilkinson polynomial of degree 20.
2. $q(x) = (x-1)^{30}(x-2)^{18}(x-3)^{12}$ (Zeng 2004).
3. $p(x) = x^{17} - (4x-1)^3$ (Bini and Mourrain 1996).
4. The Fibonacci polynomials $f_n(x) = x^n - \sum_{k=0}^{n-1} x^k$ for, say, $n = 5$ and $n = 10$.
5. For any of the Zeng (2004) test polynomials that you fancy.
6. One of the paper by Wilkinson (1959a).

2.33. Draw the first 30 Chebyshev polynomials on the same graph (like Fig. 2.1 but with more of them). You should see several smooth curves suggested by the gaps in the graph; these curves are called "ghost curves." They can be described analytically. See Rivlin (1990).

2.34. Functions containing square roots or other radicals may not have Taylor series at the branch point. A useful extension is *Puiseux* series, that is, series in terms of powers of $(z-a)^{1/p}$ for some p. Compute five terms of the Puiseux series of $\sin(\exp(\sqrt{x})-1)$ about $x = 0$.

2.35. The Mandelbrot polynomials are defined by $p_0(z) = 1$ and

$$p_{k+1}(z) = zp_k^2(z) + 1\,.$$

Expanding these polynomials in the monomial basis is a bad idea. Demonstrate this by proving that the coefficients are all positive, the leading coefficient is 1 as is the trailing coefficient, and at least one coefficient grows doubly exponentially with k (the degree is exponential in k, so the coefficients grow exponentially in the degree). Explain why this makes the condition number $B(z)$ of the monomial basis expression very large on the interval $-2 \le z \le 0$.

2.36. Implement and test the Clenshaw algorithm (see Algorithm 2.2) for the Chebyshev polynomials, which have $\alpha_k(z) = 2z$ and $\beta_k = 1$ for $k \ge 2$.

2.37. The first barycentric form is

$$p(z) = w(z) \sum_{k=0}^{n} \frac{\beta_k \rho_k}{z - \tau_k},$$

where the ρ_k are the values of $p(z)$ at $z = \tau_k$; that is, $\rho_k = p(\tau_k)$. Since this is true for all $p(z)$, it is in particular true for the constant polynomial 1:

$$1 = w(z) \sum_{k=0}^{n} \frac{\beta_k \cdot 1}{z - \tau_k}\,.$$

Dividing these two gives us the *second* barycentric form,

$$p(z) = \frac{\sum_{k=0}^{n} \frac{\beta_k \rho_k}{z-\tau_k}}{\sum_{k=0}^{n} \frac{\beta_k}{z-\tau_k}}\,,$$

about which we will learn more in Chap. 8. By cross-multiplying and using the product rule, find an expression for the derivative of a polynomial expressed in the second barycentric form of the Lagrange basis. What is the cost to evaluate this, supposing that the β_k are available?

2.38. Once one has found an approximate root r of a polynomial, one usually wants to deflate, that is, find a new polynomial $q(z) = {}^{p(z)}/_{(z-r)}$ that has the same roots as the other roots of $p(z)$ but is one degree less. Done incorrectly, this can lead to instability; Wilkinson advocated deflating roots from smallest magnitude to largest, but it has since been realized that by reversing the polynomial, that is, considering the polynomial $P(z) = z^n p({}^1/_z)$, which has as roots the reciprocals of the roots of p, one can instead deflate from largest to smallest. Use Newton's method, synthetic division, and deflation to find all roots of Newton's example polynomial $p(z) = z^3 - 2z - 5$.

2.39. Show how to reverse polynomials $P(z) = z^n p({}^1/_z)$ that are expressed in a Lagrange basis. Do not convert to the monomial basis.

2.40. Show that if the forward error $e_k = r_k - z^*$ in an approximate root to a polynomial $p(z) = 0$ is small, and $p'(z) \neq 0$ nearby, then the next iteration r_{k+1} has forward error proportional to the square of e_k. This is called quadratic convergence.

2.41. We said in the text that elements of the sequence of Newton iterates $r_{k+1} = r_k - {}^{p(r_k)}/_{p'(r_k)}$ were each *exact* solutions of the polynomials $p(z) - p(r_k) = 0$. This is trivial, in one sense, and very useful in another if $p(r_k)$ is small enough to be ignored. There is another way to look at this that is also useful. Given an approximate root r_k for $p(z)$, we can ask, "What is the *closest* polynomial $p(z) + \Delta p(z)$ for which r_k is an exact root?"

We know that the size of $\Delta p(z)$ is at most $|p(r_k)|$ by the previous "trivial" statement. But are there closer polynomials? The answer is usually yes, and there is an analytical formula for the coefficients of the optimal $\Delta p(z)$ that we can find using the Hölder inequality, as follows.

Given $p(z) = \sum_{k=0}^{n} c_k \phi_k(z)$, weights $w_k \geq 0$ for $0 \leq k \leq n$ not all zero, and a putative root r, find the minimum ε such that $(p + \Delta p)(r) = 0$ with $\Delta p(z) = \sum_{k=0}^{n} (\Delta c_k) \phi_k(z)$ such that each $|\Delta c_k| \leq w_k \varepsilon$. Then ε is the "minimal backward error" of the root r; you should find that ε is proportional to $|p(r)|$, the residual. (Hint: Reread Theorem 2.9 on page 70 and then use (C.2) in Appendix C.)

2.42. Following the discussion in Sect. 2.2.6.3 and the solution of the previous problem, find an expression for the nearest polynomial of lower degree.

Chapter 3
Rootfinding and Function Evaluation

Abstract We introduce general methods to *evaluate functions* and to *find roots* (or zeros) of functions of all kinds. We examine various approximation methods and study their respective numerical accuracy, by examining their *backward error* and the *condition numbers* for evaluation and rootfinding. ◁

If you execute the MATLAB command `y = sin( 3*pi/7 )`, you immediately get the answer `y = 0.9749`, where, as usual, you can see more figures in the answer if you execute `format long` first. Nowadays one can use a calculator, a web browser, a phone, and the like to get the same answer. This wasn't always so. Before computers, humans compiled tables of trigonometric functions by hand, beginning with geometrical methods. Analog computers to compute functions (especially the logarithm) were invented next, and when the digital computer arrived, one of the first things they were made to do was to compute mathematical functions on demand. In the early days of modern computing, quite a lot of effort was spent on the task.

However, nearing the end of the 20th century, the computation of simple mathematical functions such as the elementary functions was already old-fashioned. Now, in the 21st century, it doesn't seem to form a large part of numerical analysis either, although approximation theory is alive and well as a mathematical field.

Most of the theory of evaluation of elementary functions was developed under the *forward error* model: A subroutine for the evaluation of a mathematical function $f(x)$ was judged against the standard of requiring

$$|\hat{f}(x) - f(x)| < \mu_M |f(x)| \, ; \tag{3.1}$$

that is, if the computed function $\hat{f}(x)$ returned an answer with a relative forward error less than half ε_M ("half a Unit in the Last Place," or ulp, denoted μ_M in this book), then the result could be rounded to the machine number nearest to the correct answer—this is referred to as a *correctly rounded* result. This was, and is, the gold standard and is very hard to achieve.

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_3,

This chapter treats the issue more lightly. Instead of the gold standard, we take a backward error point of view: We think that a subroutine has done a perfect job if it evaluates

$$\hat{f}(x) = f(x(1+\delta_x)) \tag{3.2}$$

for some $|\delta_x| < \mu_M$. That is, the subroutine is doing its job if it gives you the exact value of the function at a point differing only by at most a rounding error from the argument you asked it to evaluate. This is not quite the same thing. And we may even relax our requirement a little further, too—we might be happy enough with only giving *nearly* the right value of the function at nearly the right point, as captured by the mixed forward–backward notion of stability introduced in Chap. 1.

Our reasons for taking this point of view are twofold. First, it is unlikely that the reader will be called on to write an industrial-grade piece of software for the evaluation of an elementary function (and if the reader is indeed so lucky, then reliance on just this book for a reference would be a mistake anyway, as we only have time for the general picture, and in practice the details really matter, in a nitty-gritty way). The real reason, however, is that the theory of evaluation of mathematical functions provides a good opportunity to reinforce the backward error point of view. Moreover, especially when we come to the sections on rootfinding, it's even quite a productive way to look at existing code that was designed to try to meet the "gold standard" mentioned above.

What would a backward error viewpoint do for the `y = sin( 3*pi/7 )` example? Well, the answer 0.9749 given is, in fact, the exact sine of a number slightly different to $3\pi/7$, being $\arcsin(3\pi(1-9.31\cdot 10^{-4})/7)$. If, instead, we use all the decimal places MATLAB computes, we find that MATLAB has given us (pretty nearly) the exact sine of $3\pi(1+\delta_x)/7$, where $\delta_x \approx -2.342\cdot 10^{-17}$. To find this out, we used a 30-digit computation in MAPLE. In this case, the two viewpoints don't disagree: A small backward error, such as this, entails a small forward error, too, because

$$\sin(x(1+\delta_x)) = \sin(x) + x\cos(x)\delta_x + O(\delta_x^2)\,.$$

This is, in essence, a description of the *condition number* of evaluation of the sine function. In the next section, we will see how to obtain a general understanding of such condition numbers.

3.1 Function Evaluation

As before, we will in general discuss the case of complex functions. A complex-valued function $f(z)$ can be treated directly as a univariate complex-valued function and approximated by various simpler functions over $\mathbb{C}$, or it can be split into a real part and an imaginary part. Thus, if we let $z = x+iy$ and $f(z) = u(x,y)+iv(x,y)$, then we are faced with evaluation of two real-valued *bivariate* functions $u(x,y)$ and $v(x,y)$. In principle, this is only more complicated, but not intrinsically more difficult than real-valued univariate functions (although we are now in some sense evaluating on boxes $(x,y) \in [a,b] \times [c,d]$ and not on disks $|z-a| < r$). Moreover, it is in many cases even simpler than that. For example,

$$\sin(z) = \sin(x+iy) = \sin(x)\cosh(y) + i\cos(x)\sinh(y) \ ,$$

and now to evaluate $\sin(z)$ we see that we should merely evaluate 4 *univariate* real functions. Arcane and clever tricks can be used (and are used, in MATLAB, for example) to ensure that the functions $u(x,y)$ and $v(x,y)$ that are used are accurate and efficient. Functions such as the Lambert W function (Corless et al. 1996), however, are best thought of as needing direct complex approximations. We will begin with this latter case, and then consider splitting a complex function in two real functions.

3.1.1 Condition Number for Function Evaluation

Consider a complex function $y = f(z)$ and, in particular, the value $y + \Delta y = f(z + \Delta z)$ it takes for a perturbed argument $z + \Delta z$. If $f(z)$ is an analytic function of z, then

$$\frac{\Delta y}{y} \doteq \frac{zf'(z)}{f(z)} \frac{\Delta z}{z} \, , \tag{3.3}$$

so that if we let $\kappa(f) = C = {}^{zf'(z)}/_{f(z)}$, which can be taken as a definition of the *relative condition number* for the evaluation of f, we obtain the relation

$$\delta y \doteq \kappa(f) \delta z \, .$$

The condition number of a composition of functions may be computed by the chain rule.

Example 3.1. Consider the function $f(x) = e^z$. Its relative condition number is $\kappa(f) = z$. Thus, e^z is badly conditioned only for large $|z|$. ◁

Example 3.2. Now, consider the function $f(z) = \ln z$. Its condition number is $\kappa(f) = 1/\ln z$, and so it is badly conditioned near $z = 1$. It also has a problem near the branch cut $z < 0$. The function is discontinuous across the branch cut, so that a rounding error that went from a zero imaginary part to a negative imaginary part would cause a jump in value of order 1, essentially, infinitely larger. ◁

Example 3.3. The function $g(z) = \ln(1+z)$ has the condition number

$$\kappa(g) = \frac{z}{(1+z)\ln(1+z)} \, .$$

At $z = 0$, κ is in an indeterminate form, but we can find the condition near this point by expanding κ as follows:

$$\kappa = 1 - \frac{z}{2} + \frac{5}{12}z^2 - \frac{3}{8}z^3 + O(z^4) \, .$$

From this, we see that $\kappa(g) \approx 1$ near $z = 0$. This is *good*, which is surprising given Example 3.2.[1] Here, note that these are *different functions*: one takes a small but highly precise argument z and then (conceptually) adds 1 to that, whereas the other starts with a highly precise number near 1, whose difference from 1 will not necessarily be known very well. $\triangleleft$

Example 3.4. The function $f(z) = \sqrt{z}$ has the condition number

$$\kappa(f) = \frac{z}{2\sqrt{z}\sqrt{z}} = \frac{1}{2}\,, \tag{3.4}$$

which is good everywhere (except the branch cut). $\triangleleft$

Example 3.5. The function $f(z) = \sin \omega z$ has the condition number

$$\kappa(f) = \frac{\omega z \cos \omega z}{\sin \omega z}\,, \tag{3.5}$$

which is good near $z = 0$, but very bad near $z = \pm\pi, \pm 2\pi, \ldots$. $\triangleleft$

It is important to realize that the condition number $\kappa(f)$ estimates relative changes in output versus *small* relative changes in input. That is, it is a linear in-the-limit-of-small-errors perturbation analysis. Nonetheless, it can be very useful. The reader should be wary, however. Quite a bit of information about the data being used is contained in the simple assumptions that we care about, for example, $\Delta z/z$ and $\Delta f/f$: This model makes sense only if $z \neq 0$ and $f(z) \neq 0$ and is of importance only if the values of f (or z) differ greatly from 1, either being very large, or very small (in which case the "nonzero" assumption becomes a bit delicate). The fact that $\ln z$ is ill-conditioned near $z = 1$ but the function $\ln(1+\zeta)$ is well-conditioned near $\zeta = 0$ appears to be a contradiction, but $\Delta z/z$ and $\Delta\zeta$ will not be the same thing at all, and this is the first example of such a delicate argument that most numerical analysts see.

Example 3.6. The Mandelbrot polynomials, which you met in Problem 2.35, have a natural recurrence relation definition: $p_0(z) = 1$ and $p_{k+1}(z) = zp_k^2(z) + 1$ thereafter. It is easy to differentiate this rule, to get $p_0'(z) = 0$ and $p_{k+1}'(z) = p_k^2(z) + 2zp_k(z)p_k'(z)$. This recurrence relation provides an effective method for computation of $p_k(z)$ for any reasonable k. For example, we can take $k = 7$ and compute the condition number $zp_k'(z)/p_k(z)$, and plot its contours on, say, $-3 \leq x \leq 1$, $-2 \leq y \leq 2$, as follows:

```
function mandelpic

[X,Y] = meshgrid(-3:.001:1,-2:.001:2);
Z=X+1i*Y;

[p,dp]=mandelbrot(Z,7);

```

[1] See Higham (2002) for an extended discussion of this issue.

```
contour(X,Y,abs(Z.*dp./p),2.^[-4:1:8],'k');
axis('square');
axis image, axis off;

end

function [p,dp]=mandelbrot(z,n)
  if n<=0
    p=ones(size(z));
    dp=zeros(size(z));
  else
    [p0,dp0]=mandelbrot(z,n-1);
    p=z.*p0.^2+1;
    dp=p0.^2+2.*z.*p0.*dp0;
  end
end
```

The result displayed was computed with MAPLE with color in Fig. 3.1. ◁

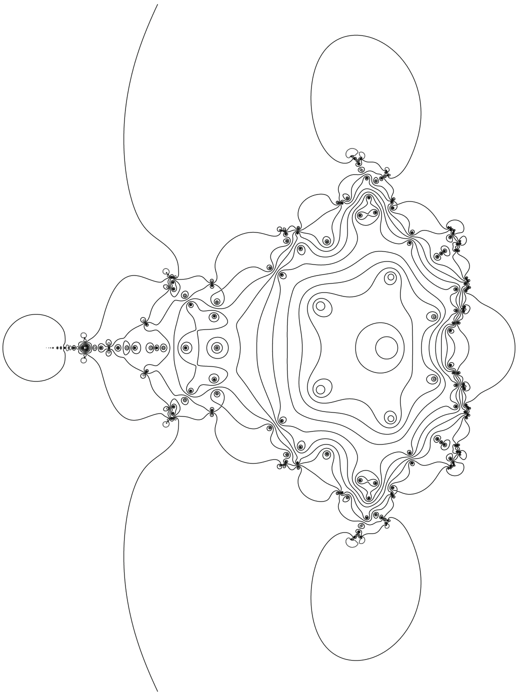


Fig. 3.1 Contours of the condition number of the Mandelbrot polynomial with $k = 7$, which is of degree $2^k - 1 = 127$. The contours are at levels 2^m for various m

It will also be important to have a condition number for the evaluation of *multivariate functions*. Thus, if we wish to know how sensitive a function such as

$$y = f(z_1, z_2, z_3)$$

is to changes in any z_k, the chain rule gives

$$\delta_y \doteq \left[{}^{x_1 f_1}/{}_f, {}^{x_2 f_2}/{}_f, {}^{x_3 f_3}/{}_f\right] \cdot \left[\delta_1, \delta_2, \delta_3\right] . \tag{3.6}$$

Thus, it makes sense to think of a (convenient) vector norm of this vector of scaled partial derivatives as "the" condition number of this scalar function of a vector of variables. Similarly, vector-valued multivariate functions give a matrix of scaled partial derivatives, and the "size" of this matrix (we will study matrix norms in some detail in upcoming chapters) will give a sense of how errors are amplified during function evaluation.

3.1.2 Conditioning of Real and Imaginary Parts Separately

If one chooses to treat the real and imaginary parts of $f(z)$ separately, then a matrix of condition numbers is needed, as we now show. We consider the effects of separate changes in the real and imaginary parts of z on the real and imaginary parts of $f(z)$. We write $z = x + iy$ and $f(z) = u(x,y) + iv(x,y)$. Then,

$$\begin{bmatrix} \Delta u \\ \Delta v \end{bmatrix} = \mathbf{C} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} ,$$

where the 2×2 matrix $\mathbf{C}$ is given by

$$\mathbf{C} = \begin{bmatrix} u_x(x,y) & u_y(x,y) \\ v_x(x,y) & v_y(x,y) \end{bmatrix} . \tag{3.7}$$

Because of the Cauchy–Riemann equations, if $f(z)$ is analytic, these quantities are related: $u_x = v_y$ and $u_y = -v_x$. So far, this is just differentiation (and, if $f(z)$ is analytic, equivalent to $\Delta f(z) = f'(z)\Delta(z)$). However, it sometimes happens that while $|{}^{zf'(z)}/{}_{f(z)}|$ does not have a problem, one of the four values xu_x/u, yu_y/u, xv_x/v, or yv_y/v *does* have a problem, as (say) $v \to 0$ (but not quite), while u stays bounded away from 0 (and hence, $f(z)$ stays bounded away from 0). In this case, we may be able to compute $f(z)$ perfectly well as a complex function, but (say) errors in its imaginary part will be large (relative to the imaginary part, although not large relative to $|f(z)|$).

Example 3.7. Consider the complex function $w = \log(x + iy)$. Its real and imaginary parts are

$$\begin{aligned} u &= \log r = \log \sqrt{x^2 + y^2} \\ v &= \arctan(y,x) . \end{aligned} \tag{3.8}$$

Computation of the separate condition numbers gives

$$x \frac{u_x}{u} = \frac{x^2}{(x^2 + y^2)\log(r)} \tag{3.9}$$

$$y\frac{u_y}{u} = \frac{y^2}{(x^2+y^2)\log(r)} \tag{3.10}$$

$$x\frac{v_x}{v} = -\frac{xy}{(x^2+y^2)\arctan(y,x)} \tag{3.11}$$

$$y\frac{v_y}{v} = \frac{xy}{(x^2+y^2)\arctan(y,x)}\,, \tag{3.12}$$

with the interesting complication that $\arctan(y,x)$ has a jump discontinuity across the negative real axis, that is, $x < 0$ and $y = 0$ (and hence, the last two derivative formulæ are not valid there).

We see from Eqs. (3.9) and (3.10) that the real part is relatively ill-conditioned on (or near) the unit circle: Small changes in either x or y will cause large relative changes in the real part of the logarithm. Moreover, we see from Eqs. (3.11) and (3.12) that there is a problem if both x and y go to zero in such a way that ${}^{xy}\!/_{r^2}$ goes to infinity; on the other hand, if just $y \to 0$, then the arctangent will balance it. An example of where this gets into trouble is if $y = \varepsilon \tan\theta$ and $x = \varepsilon$, when $\arctan(y,x) = \theta$ and the condition number is

$$\frac{\varepsilon^2}{(\varepsilon^2+\varepsilon^2\tan^2\theta)\theta} = \frac{1}{\theta\sec^2\theta} = \frac{\cos^2(\theta)}{\theta}\,.$$

This clearly has a problem if $\theta = 0$, but this is fortunately hard to achieve in practice: The difficulty shows up only when ε is small, while θ is small but not as small as ε. In practice, this problem is ignored (and in any case, the argument $\arctan(y,x)$ doesn't make much sense if both x and y are very small).

However, the difficulty along the negative real axis is quite genuine. A small change in y, from positive to negative, will change the arctangent by nearly 2π. Thus, the imaginary part is infinitely ill-conditioned along the negative real axis *if perturbations are allowed to cross the branch cut.* ◁

3.2 Rootfinding

At this point, we can already evaluate polynomial and rational functions. The next simplest kind of function is an *algebraic function*, which is defined as a root (or zero) of a bivariate polynomial function. For instance, the function $y^2 - z = 0$ defines $y = \sqrt{z}$, and by the conventional branch choice, $\mathrm{Re}(y) \geq 0$, and if $\mathrm{Re}(y) = 0$, then $\mathrm{Im}(y) \geq 0$ also. Of course, the other branch, $y = -\sqrt{z}$, also satisfies this algebraic equation. To evaluate general algebraic functions, we need techniques for *rootfinding*, and similarly for transcendental functions, of course. Can we write a reliable routine that, given y, finds x so that $y = f(x)$? That is, can we solve the “inverse function” problem and compute $x = f^{-1}(y)$? What about for complex z and w in $w = f(z)$, instead of real x and y? In this section, we examine how to approach such questions.

3.2.1 The Condition Number of a Root

When we took the equation $y = f(x)$ and considered x as the input and y as the output, and considered the effects of changes in x on the value of y, we arrived at the condition number for function evaluation $C = xf'(x)/f(x)$ and at the relation $\delta y \doteq C\delta x$. Now the situation is similar, but reversed: We are given y, which is $y = 0$ for rootfinding problems, and are asked to find x. We can now think about what happens if y is changed a bit, or alternatively if the function f is changed to $(f + \Delta f)(x)$. Considering

$$0 + \Delta y = f(x + \Delta x) = f(x) + f'(x)\Delta x + O(\Delta x)^2 ,$$

we obtain $\Delta y \doteq f'(x)\Delta x$, which can be rewritten as

$$\Delta x \doteq \frac{1}{f'(x)}\Delta y ,$$

giving us our absolute condition number. If $x \neq 0$, by considering the relative change in x, we get $\Delta x/x = \delta x \doteq (1/(xf'(x)))\,\Delta y$. Since the ratio $\Delta y/y$ does not make sense because $y = 0$, this gives us a mixed absolute-relative condition number that is related to the reciprocal of C as computed for function evaluation. Indeed, the pure absolute condition number of function evaluation is just $f'(x)$, whereas the pure absolute condition number of rootfinding is exactly its reciprocal, $1/f'(x)$. This says that, where a function is vertical, it is hard to evaluate accurately; and where a function is horizontal at a root, it is hard to locate the root accurately.

Before we examine the conditioning of roots in general in Sect. 3.2.4, consider the special case in which $f(z) = \sum_{k=0}^{n} c_k\phi_k(z)$ is a polynomial and the coefficients are perturbed to $c_k(1 + \delta_k)$, where each $|\delta_k| \leq \varepsilon$, and $f(z)$ has a simple root at $z = r$, which gets perturbed to $r(1 + \delta_r)$ when we evaluate $0 = (f + \Delta f)(r(1 + \delta_r))$, we find that

$$0 = (f + \Delta f)(r(1 + \delta_r)) = f(r) + \Delta f(r) + f'(r)r\delta_r + \cdots .$$

Since $f(r) = 0$, we have to first order that

$$\begin{aligned} |\delta_r| = \left| -\frac{\Delta f(r)}{rf'(r)} \right| &= \frac{|\sum_{k=0}^{n} c_k\delta_k\phi_k(r)|}{|rf'(r)|} \\ &\leq \frac{B(r)\varepsilon}{|rf'(r)|} . \end{aligned} \tag{3.13}$$

The last inequality follows from Hölder's inequality. Therefore, the condition number of a simple root $r \neq 0$ of a polynomial f with respect to changes in its coefficients is

$$C = \frac{B(r)}{|rf'(r)|} .$$

Here $B(z) = \sum_{k=0}^{n} |c_k||\phi_k(z)|$ is simply the condition number for *evaluation* of the polynomial, which we have seen before in Chap. 2.

3.2.2 Newton's Method

For real-valued problems with simple roots, bisection[2] is slow but reliable, since one can often then find a and b with $f(a)f(b) < 0$. In contrast, Newton's method, which we introduced in the last chapter in the context of polynomials, is fast but skittish and needs derivatives and good initial guesses; the secant method and inverse quadratic iteration (IQI) are almost as fast but don't need derivatives. We shall look at all of these, but let's begin with Newton's method.

Newton's method for solving a general transcendental equation $f(x) = 0$ is the iteration

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\,,$$

which is derived from the Taylor series at the initial guess from the supposed root x^*:

$$0 = f(x^*) = f(x_0 + (x^* - x_0)) \doteq f(x_0) + f'(x_0)(x^* - x_0) + O(x^* - x_0)^2\,.$$

The iteration usually converges quickly if it is given a good enough starting guess x_0, but it requires a derivative evaluation at each step (and thus can be a bit costly). Moreover, Newton's method will have problems if $f'(x_n) = 0$ or if $f'(x^*) = 0$ (where x^* is the root that we are looking for); and it can get caught in various kinds of oscillations. Without a good starting guess, convergence can be quite slow initially.

Example 3.8. We start with a polynomial example. Newton's method can be used effectively to find the square root of a number. Suppose we wish to find the square root of, say, 5. In this case, $f(z) = z^2 - 5$, and so $f'(z) = 2z$. We choose an initial guess, say $x_0 = 2$, and then use this iteration:

$$x_{n+1} = x_n - \frac{x_n^2 - 5}{2x_n}\,.$$

Unlike many Newton iterations, this one can be usefully rewritten as

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{5}{x_n}\right)\,; \tag{3.14}$$

[2] Bisection is explained in many numerical analysis texts, and it isn't quite trivial: If you know a function is positive somewhere (say at $x = 1$) and negative somewhere else (say $x = 0$), have a look at it halfway between: If it's zero there, you've found the root; if it's positive, then the root is between 0 and 1/2, and otherwise it's between 1/2 and 1. Repeat as necessary.

that is, divide your guess into what you want to find the square root of (here, 5), and then take the mean with what you had before. For taking square roots of positive x, this has very little trouble from rounding error. We get $x_0 = 2$, $x_1 = 2.25$, $x_2 = 2.236111111111111$, $x_3 = 2.236067977915804$, $x_4 = 2.236067977499790$, and thereafter all the x_k are constant, just the same as x_4: The iteration has converged (to the correct square root of 5 to all digits in MATLAB). ◁

Remark 3.1. Except for square roots as in that example, it's never good to rewrite Newton's method away from its usual form $x_{\text{new}} = x_{\text{old}} +$ small update. In this form, it tends to minimize the effect of rounding errors. For positive square roots, the above rewriting does no harm, but this is unusual. ◁

Theorem 3.1. *Newton's method has quadratic convergence (when it converges).*

Proof. Suppose that $f(x^*) = 0$ and, again, that the relation $x_{n+1} = x_n - {}^{f(x_n)}\!/_{f'(x_n)}$ holds. Then

$$0 = f(x^*) = f(x_n - e_n) = f(x_n) - f'(x_n)e_n + \frac{\overline{f''}}{2}e_n^2$$

for some average value $\overline{f''}$ of the second derivative,[3] and

$$e_n = x_n - x^* = x_n - x_{n+1} + x_{n+1} - x^* = (x_n - x_{n+1}) + e_{n+1}\,.$$

So, it follows that

$$\begin{aligned} 0 &= f(x_n) - f'(x_n)((x_n - x_{n+1}) + e_{n+1}) + \frac{1}{2}\overline{f''}e_n^2 \\ &= f(x_n) - f'(x_n)(x_n - x_{n+1}) - f'(x_n)e_{n+1} + \frac{1}{2}\overline{f''}e_n^2\,. \end{aligned}$$

However, by the definition of the Newton iteration, $f(x_n) - f'(x_n)(x_n - x_{n+1}) = 0$. Hence,

$$e_{n+1} = \frac{\overline{f''}}{2f'(x_n)}e_n^2\,,$$

showing that the convergence is quadratic. ♮

We remark that if $f'(x^*) \neq 0$, then $f'(x_n) \neq 0$ for values of x_n that are "close enough" to x^*, and of course, $\overline{f''} \to f''(x^*)$ as well if the iteration converges. Hence, if the iteration converges, it does so ultimately with an approximate doubling of correct digits each iteration.

[3] Phrased this way, the proof works even for complex x with an appropriate definition of "average." See the complex form of the Taylor series remainder given in Chap. 2.

This method has many applications. Schoolchild algorithms for multiplication and division of decimal fractions are likely known to every reader. A few readers might know such an algorithm for extracting square roots. As we saw, Newton's method is very effective for square root computation, which is not surprising, but it might be surprising to learn that Newton's method quite often replaces the *division* algorithm. One trick for that is to use Newton's method on $f(y) = {}^1\!/\!_y - x$ to find $y = {}^1\!/\!_x$, because then the iteration is

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)} = y_n - \frac{{}^1\!/\!_{y_n} - x}{-{}^1\!/\!_{y_n^2}} = y_n + (y_n - xy_n^2) = y_n(2 - xy_n)\,,$$

which can be carried out using only multiplication and addition.

Newton's method is surprisingly important in the computation of functions to arbitrary precision. For instance, if one has a fast method for computing a logarithm, then the cost to use Newton's method to find the exponential adds nothing significant to the asymptotic cost of the algorithm.

Remark 3.2. Notice that there is a problem with Newton's method if the root is *multiple*, in which case $f'(x^*) = 0$. Notice also that in this case the root is *ill-conditioned*: A tiny change in the function value forces a potentially large relative change in the root location. ◁

To end this subsection, we want to stress that determining when Newton's method converges is a complicated issue. Cayley had already proved in the 19th century that, for the simple equation $x^2 - a$, the *basins of attraction* for Newton's method, that is, the sets of initial points x_0 for which Newton's iteration converges to a root, were simple: If $\mathrm{Re}(x_0) < 0$, then $x_n \to -\sqrt{a}$, and if $\mathrm{Re}(x_0) > 0$, then $x_n \to +\sqrt{a}$. In a lovely paper, Strang (1991) discusses the chaotic behavior (so that the iteration does not converge at all) that results from $x_0 \in i\mathbb{R}$.

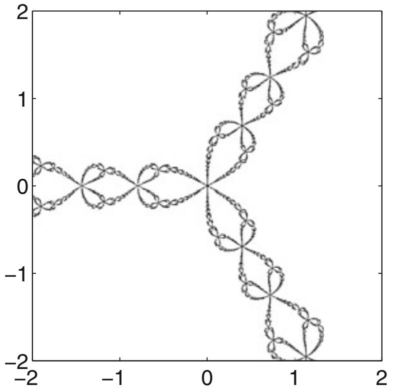


Fig. 3.2 Fractal boundary for the basins of attraction of Newton's map for $f(z) = z^3 - 1$

But it was not realized until the early 20th century that even the simple cubic

$$f(z) = z^3 - 1\,,$$

which has roots $\omega_1 = 1$, $\omega_2 = e^{2\pi i/3}$, and $\omega_3 = \overline{\omega_2}$, has very complicated convergence behavior. If we start the iteration with some $z \in \mathbb{C}$, then for almost all z, the Newton iteration converges to one of ω_1, ω_2, or ω_3. However, while each basin of attraction has an open interior, the boundary between basins is more complicated. Julia and Fatou proved (apparently without looking at a picture!) that, given a point z_b in the boundary, then in every neighborhood $|z - z_b| < \varepsilon$, there is a $z_0^{(i)}$ such that $z_{k+1}^{(i)} = z_k^{(i)} - f(z_k^{(i)})/f'(z_k^{(i)})$ has $\lim_{k\to\infty} z_k^{(i)} = \omega_i$. That is, arbitrarily near to each boundary point, there are initial points that lead under Newton's iteration to each root. This can only mean that the boundary is fractal, as shown in Fig. 3.2. The MATLAB code to produce this figure is the following:

```
%
% Draw an approximate boundary between basins of attraction
% in Newton iteration for f(x) = x^3 - 1
%
% RMC November 2010
%
f = @(x) x.^3-1;
df = @(x) 3*x.^2;
Newt = @(x) x - f(x)./df(x);
% The following takes quite a bit of memory
% but makes a nice, lacy picture
lots = 1001;
x = linspace( -2, 2, lots );
y = x';
z = ones(lots,1)*x + 1i*y*ones(1,lots);
% Simple initial guess
r = z;
% Thirty iterations is plenty
for j=1:30,
  r = Newt(r);
end;
% The only points left are the roots or the boundary.
contour(x,y,r,'k');
axis('square');
set(gca,'fontsize',16);
```

Remark 3.3. Newton's method is a workhorse for solving transcendental equations. It does have its problems: Because derivatives are needed for the iteration, one must find a way to compute or approximate them; it can be sensitive to rounding errors; it has difficulty with multiple roots; global convergence is by no means given (and indeed there can be full-measure regions of initial conditions for which convergence does not occur); even if the method converges, it may converge to the "wrong root" (perhaps one you have found already); and in any case, it only finds one root at a time. Nonetheless, in spite of all these difficulties, it remains a powerful method. ◁

3.2.3 Wilkinson's First Example Polynomial

In order to understand the notion of condition number introduced above, and how it relates to the use of Newton's particular method to find roots, we now consider in some detail Wilkinson's first (and more famous) example polynomial, namely,

$$p(z) = \prod_{j=0}^{20}(z-j) = (z-1)(z-2)\cdots(z-20)\,. \tag{3.15}$$

We have already begun to look at this polynomial in Example 2.6. As we said, the key misstep in evaluating this polynomial or finding its roots is *expanding it into the monomial basis*. Since, as written earlier, the polynomial is expressed in the Lagrange basis on the nodes 0, 1, ..., 20, this amounts to a change of basis, and this is a particularly ill-advised one. As Wilkinson himself notes, in the form above the polynomial is perfectly conditioned. This is trivial: If we know the roots, we can find them—this is not a surprise. It is a bit more of a surprise to realize that the values of $p(z)$ for any z can be computed accurately using this formula (which is already in barycentric form of course), because the *evaluation* condition number $B(z)$, namely,

$$B(z) = |\prod_{j=0}^{20}(z-j)|\,, \tag{3.16}$$

is of modest size in comparison to $|p(z)|$. In fact, the ratio is just 1.

In what follows, we use only the *evaluation* condition number because the *rootfinding* condition number is, from Eq. (3.13), just $B(r)/|rp'(r)|$, and the other two factors are independent of the basis. Thus, by comparing the size of the evaluation condition number in each basis, we are comparing the size of the rootfinding condition number in each basis as well. The condition numbers that we will compute are summarized in Fig. 3.3.

Naively converting Wilkinson's polynomial to the monomial basis results in

$$p(z) = z^{20} - 210z^{19} + 20615z^{18} + \cdots + 20!\,, \tag{3.17}$$

where the last coefficient is $20! = 2{,}432{,}902{,}008{,}176{,}640{,}000$. In *this* basis,

$$B_{\text{monomial}}(z) = |z|^{20} + 210|z|^{19} + \cdots + 20!\,, \tag{3.18}$$

which is *substantially* larger, growing to 3.3537×10^{29} at $z = 20$, compared to the maximum value in the original of $20! = 2.4329 \times 10^{18}$. That's 10^{11} times as big. The rootfinding condition number ${}^{B(r)}/_{|rp'(r)|}$ has a maximum of about 10^{14}, so in double precision we might lose all but two figures of accuracy (see Problem 3.24). For $N = 30$, this is about 10^{21}, and for $N = 40$, this is about 10^{29}. This is (apparently) exponential growth in the maximum rootfinding condition number.

Remark 3.4. Stoutemyer's rule of thumb is, in David Stoutemyer's own words, this: "In my experience, it is often wise to use more than n decimal digits of precision

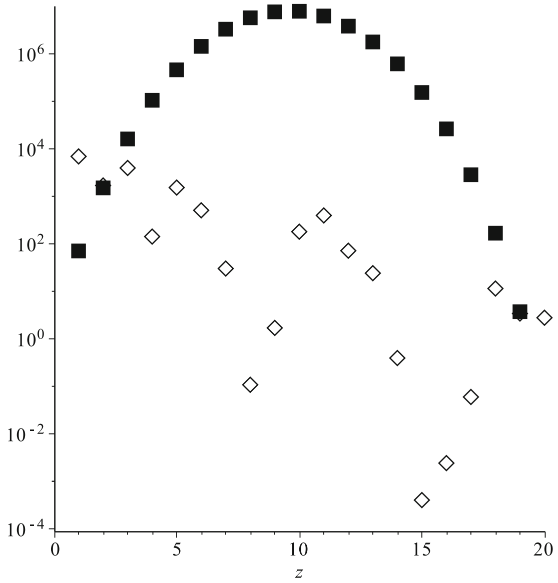


Fig. 3.3 The condition number $B(z)$ of Wilkinson's polynomial in two different bases at the roots of the polynomial: The *squares* are the "optimal" Bernstein–Béizier basis and the *diamonds* are a Lagrange basis on nodes chosen uniformly at random on the interval $[0,20]$. The monomial basis condition number is not shown, as it is too large, reaching over 10^{29} at the right-hand side, and this would compress the graph unacceptably

when summing n terms of a series, computing nth-degree regressions, computing the zeros of an nth degree polynomial, etc." The Wilkinson polynomial examples have a condition number that grows exponentially with the degree and thus needs a precision that grows *linearly* with the degree. To work with $N = 20$ needs a bit more than 16 digits; to work with $N = 30$ requires a bit more than 21 digits; to work with $N = 40$ requires a bit more than 29 digits. This provides experimental support for Stoutemyer's rule of thumb. $\triangleleft$

Farouki and Goodman (1996) show that the condition numbers are much better and indeed in a certain sense optimal if the polynomial is instead expressed in the Bernstein–Bézier basis. In this case, the polynomial is (exactly)

$$\frac{14849255421}{640000000000000000}(20-z)^{20} - \frac{617191994979}{512000000000000000}z(20-z)^{19} \\
+\frac{25953467080473}{1024000000000000000}z^2(20-z)^{18} - \frac{60042878381637}{204800000000000000}z^3(20-z)^{17} \\
+\frac{10866631664192427}{5120000000000000000}z^4(20-z)^{16} - \frac{52830616292575641}{5120000000000000000}z^5(20-z)^{15} \\
+\frac{36026164321154639}{1024000000000000000}z^6(20-z)^{14} - \frac{88585902536686877}{1024000000000000000}z^7(20-z)^{13}$$

$$
\begin{aligned}
&+\frac{1599474726069290 43}{1024000000000000000}z^{8}\left(20-z\right)^{12}-\frac{1071867924710442689}{5120000000000000000}z^{9}\left(20-z\right)^{11}\\
&+\frac{1071867924710442689}{5120000000000000000}z^{10}\left(20-z\right)^{10}-\frac{1599474726069290 43}{1024000000000000000}z^{11}\left(20-z\right)^{9}\\
&+\frac{885859025366868 77}{1024000000000000000}z^{12}\left(20-z\right)^{8}-\frac{360261643211546 39}{1024000000000000000}z^{13}\left(20-z\right)^{7}\\
&+\frac{52830616292575641}{5120000000000000000}z^{14}\left(20-z\right)^{6}-\frac{10866631664192427}{5120000000000000000}z^{15}\left(20-z\right)^{5}\\
&+\frac{60042878381637}{2048000000000000000}z^{16}\left(20-z\right)^{4}-\frac{25953467080473}{1024000000000000000}z^{17}\left(20-z\right)^{3}\\
&+\frac{617191994979}{5120000000000000000}z^{18}\left(20-z\right)^{2}-\frac{14849255421}{6400000000000000000}z^{19}\left(20-z\right)
\end{aligned}
$$

and the maximum value of the condition number in *this* basis is about 1.078×10^{20}, only about two orders of magnitude worse than the original formulation. But it *is* actually worse, which seems odd, given the (correct!) characterization of the Bernstein–Bézier basis as "optimal." The catch is that the optimality only holds over all nonnegative bases and for generic polynomials, and the Lagrange bases do indeed take on negative values—and we are looking at a *particular* polynomial. This is a concern. However, they are nonnegative on a particular *finite set*, namely, the interpolation nodes (where they take on the values 0 or 1, trivially). This is enough to extend the optimality result by weakening the conditions, as is done in Chap. 8.

We have already seen that the original Lagrange basis is better for expressing the Wilkinson polynomial. But that was somehow unfair. What about, say, a Lagrange basis on nodes chosen at random from a uniform distribution on the interval $[0,20]$? Well, sometimes this can be bad, as bad as the monomial basis; but much more frequently it is better, by as much as six orders of magnitude, than the Bernstein–Bézier basis! Let us take a particular instance in which we apply Newton's method on Wilkinson's polynomial expressed in the Lagrange basis on the following set of nodes:

$$
\begin{aligned}
[&0.448483409300000, 2.13014107314000, 3.86279632830000, 4.21872857844000,\\
&6.59689183604000, 7.72816614900000, 7.91437721068000, 7.92825446006000,\\
&8.24572571680000, 8.55104113738000, 9.09488793946000, 12.3146413849600,\\
&13.8921437853000, 14.6123258589200, 14.7320524468800, 15.0014414439800,\\
&15.4602596004600, 16.0037496891800, 16.8524536888400, 16.9403753077600,\\
&18.8982670005800, 19.9283442836000].
\end{aligned}
$$

Starting at $x_0 = 16.5$ yields the sequence

$$
\begin{aligned}
x_0 &= 16.5\\
x_1 &= 15.7773942261971\\
x_2 &= 16.1676354668247\\
x_3 &= 16.0180872226801
\end{aligned}
$$

$$\begin{aligned} x_4 &= 16.0003781413965 \\ x_5 &= 16.0000001763342 \\ x_6 &= 16.0000000000000, \end{aligned}$$

which is entirely satisfactory. Contrariwise, using the *monomial* basis, we stopped the iteration after computing x_{10} because nothing had settled down (indeed, $x_8 = 16.4999$ had returned quite close to the original starting value). The difficulty is more than just that we are starting exactly between two roots: Perturbing the monomial basis coefficients by trivial amounts changes the roots by a very large amount, because this basis is so ill-conditioned. It is almost certainly true that the floating-point Wilkinson polynomial in the monomial basis does not have 20 real roots, and the roots near 16 are particularly sensitive.

3.2.4 Backward Error Analysis Again

One can regard a general rootfinding problem as a map $\varphi : f \to \{x \mid f(x) = 0\}$, where "$f(x) = 0$" is the defining equation for this problem. For an approximately computed root $\hat{x}$ generated by some engineered version $\hat{\varphi}$ of the map φ (e.g., Newton's map iterated 5 times, in which case $\hat{x} = x_5$), the defining equation will not be exactly satisfied. Rather, we will have $f(\hat{x}) = r$ and, following the definitions from Chap. 1, we see that this value r is the residual. Moreover, if we simply let $g(x) = f(x) - r$, we see that our computed value $\hat{x}$ is the exact root of a modified function, since $g(\hat{x}) = 0$. As we see, in this case the residual and the backward error are the very same quantities. Thus, if the residual is small, we know that we have found the *exact root of a slightly perturbed equation*.

Backward error analysis for rootfinding of a univariate function can be as simple as that. This observation brings us back to the question of conditioning. That is, how sensitive is the root x to such changes? In order to find out, observe that, by Taylor expansion,

$$f(x) = 0 \doteq f(\hat{x}) + f'(\hat{x})(x - \hat{x}),$$

so that if $f(\hat{x}) = r$, then $x - \hat{x} \doteq -r/f'(\hat{x})$. Thus, the forward error is approximately

$$\Delta x = x^* - x_n \doteq -\frac{f(x_n)}{f'(x_n)}.$$

An iteration of Newton's method would use this estimate of the error to improve the root. Here, we will not use it to improve the root, but rather to estimate the error. If we divide by the root, so that

$$\delta x = \frac{x^* - x_n}{x^*} \doteq -\frac{f(x_n)}{x^* f'(x_n)} = \frac{1}{x^* f'(x^*)} \cdot (-r), \tag{3.19}$$

and if we then scale by a "typical" size of $f(x)$ on the interval in question, say $\|f\|$, we obtain the relation

$$\delta x \doteq \frac{\|f\|}{x^* f'(x^*)} \cdot \left(\frac{-r}{\|f\|}\right) = \kappa \delta y\,, \tag{3.20}$$

and so we have obtained a *relative condition number for the root*.[4]

Remark 3.5. This is distinct from the *condition number of the particular expression* used to evaluate $f(x)$ discussed in Sect. 3.2.1. The notions are related, but here only the value of the function and its derivatives are used; there is no other perturbation considered than the perturbation in the *value* of x, whereas before, the coefficients of the expression were allowed to change as well. Different condition numbers are appropriate for different situations. ◁

3.3 Transcendental Functions

How *does* a computer evaluate a transcendental function f at a given point z, that is, find the value of $f(z)$? Moreover, how does it compute the roots of such functions? In everyday mathematical life, we take the elementary functions $\sqrt{z}$, $\ln z$, e^z, $\sin z$, $\cos z$, $\arcsin z$, $\arccos z$, $\arctan z$, and others for granted.[5] But what about other functions? In application, we are very often also interested in the so-called special functions, too.[6] How are we to proceed, and what sort of guarantees can be expected for our computed numerical solutions?

As mentioned in Chap. 1, the IEEE 754 floating-point standard guarantees that addition, subtraction, multiplication, and division (and possibly square root) are *correctly rounded*. That is, they give the nearest machine number to the exact results for those operations. It would be nice if the built-in routines for the elementary functions (and some special functions) also carried this guarantee. They do *not*, because of what's known as the "table maker's dilemma," which essentially consists of not knowing beforehand how many figures to work to in order to ensure that numbers containing a long string of 0s (is the next bit a 1?) get rounded correctly.[7]

[4] Note that the role of the variables x and y are reversed if we compare it to other relation of this type derived so far. This is because, as we mentioned, rootfinding is in some sense an inverse problem.

[5] A function is *elementary* if it can be constructed in a finite tower of Liouvillian extensions of logarithmic, exponential, or algebraic type: This means that $\ln x = \int_1^x \frac{dt}{t}$, the exponential function, all rational polynomials, and all roots of polynomial functions are elementary. The trigonometric functions are just exponentials, for example, $\sin x = {}^{(e^{ix}-e^{-ix})}\!/_{2i}$. A transcendental function is a function that is not algebraic; some elementary functions are transcendental. Transcendental functions that are not elementary (such as the Gamma function) are called *special* functions. See, for example, Geddes et al. (1992) for a fuller discussion.

[6] See Gil et al. (2007) for an excellent compendium of methods for numerical evaluation of many special functions of practical interest.

[7] See Muller et al. (2009) for a discussion of recent progress toward that laudable goal.

So, how do computer subroutines for transcendental functions actually work? The full story is too complicated for this introductory book (indeed, only specialists know the full story), especially when it is done in hardware. Rather than attempting to do that in vain, we will illustrate by examples the sort of reasoning that underlies numerical methods to solve evaluation and rootfinding problems involving transcendental functions.

3.3.1 Evaluation of Transcendental Functions

There are a few basic, simple ideas that find themselves systematically exploited for the evaluation of transcendental functions: *argument reduction*, *polynomial and rational approximation*, and special algorithms such as the arithmetic–geometric mean (AGM) method.[8] Rational approximation uses any of several constructive methods in order to build a good approximant; this is possibly slow and expensive, but, once constructed, the approximant can be used cheaply thereafter. But in what follows, we focus on the interplay between the other two ideas.

The method of *argument reduction* can be used to evaluate a function $f(x)$ when two things are known: (1) an accurate way to evaluate $f(x)$ on an interval $[a,b]$, and (2) an effective way of computing $f(x)$ in terms of $f(\xi)$, with $\xi \in [a,b]$, plus possibly some other functions whose evaluation is unproblematic. A very simple example of argument reduction would arise if we knew how to evaluate $\sin\xi$ on $[-\pi,\pi]$; since $\sin(x+2\pi) = \sin x$, we can reduce the evaluation of $\sin x$ to the evaluation of $\sin\xi$ for some $\xi \in [-\pi,\pi]$.[9]

Nowadays there is not much call for a numerical analyst to write a subroutine to evaluate an elementary function. They have almost all been done, and done very well. In order to *understand* the basic strategies for evaluation just mentioned, we will consider the following imaginary scenario in which we are forced to return to the sources: Robin Crusoe is marooned on an easy-living desert island, where all material needs are easily satisfied. In order to break the mental monotony, Robin sets out to do something mentally challenging, but not too taxing. Sand is the only computing tool available, and Robin will attempt, with its aid, to design a function for computing the logarithm function. We'll just follow along in the development. Our purpose in this hypothetical is not to prepare you for marooning, but rather to illustrate some of the ideas that have gone into the making of the real subroutines and chips in use today. We are, of course, going to make full use of MAPLE for solving the equations, and delicately draw a veil on Robin's labors on such mechanical tasks, and we will test Robin's formulæ by drawing high-precision pictures of the error curves, something that Robin obviously cannot do.

[8] The AGM is used for elliptic functions and for high-precision computation. See Borwein and Borwein (1984) and Brent (1976).

[9] There is a difficulty in this example for very large x; see Problem 3.28.

To begin with, Robin experiments with Briggs' method.[10] A simple variant of Briggs' method (mentioned briefly in Chap. 1) consists in taking 12 successive square roots of a number, subtracting 1, and then multiplying the result by $2^{12} = 4096$. It is left to Exercise 3.16 to explain why this method works at all, and why $\ln(x) = 2^{12}(\varepsilon - \varepsilon^2/2 + \varepsilon^3/3 - \varepsilon^4/4 + \cdots)$ actually works rather well. The rule $\ln(\sqrt{x}) = \ln(x)/2$ isn't quite enough, all by itself, to explain this, but almost. Thus, Robin takes the square root of 5 to get 2.236067977499790, then takes the square root of this number to get 1.495348781221221, and so on, until after the 12th repetition, the answer is 1.000393006384622. Subtracting 1 results in $\varepsilon = 0.000393006384622$, and multiplying by 2^{12} gives 1.609754151411712. As it happens, the true logarithm of 5 is about 1.60943791243410, and so all that labor produced an answer accurate only to about 4 decimal places.

Robin is not happy with so little accuracy. Could argument reduction help? During this labor, Robin was reminded that iterated square roots make large numbers small rather quickly, and make small (positive) numbers larger rather quickly—driving both extremes toward 1. In fact, the maximum real number in IEEE754 arithmetic (`realmax` in MATLAB) is about $1.8 \cdot 10^{308}$, and the minimum positive real (`realmin` in MATLAB) is about $2.2 \cdot 10^{-308}$. Twelve square roots of each of these bring them into the interval $[1, \sqrt{2}]$. Even just nine square roots brings them into the interval $[1/5, 5]$. Robin then realizes that an *accurate* logarithm function on the interval $[1/5, 5]$, together with (at most) nine square roots (and doubling at most nine times afterward), would suffice to compute any IEEE754 double-precision real number. This is, of course, argument reduction. By using $\ln(x^2) = 2\ln x$, we have reduced the range of required accurate logarithm from $[\text{realmin}, \text{realmax}]$ to $[1/5, 5]$. One could reduce it even further, to $[0.707, 1.414]$ by using 11 square roots, but we leave that design choice to the exercises.

Robin also considers another common trick, to make the reduced interval symmetric. By introducing the change of variables

$$x = \frac{1 + \alpha u}{1 - \alpha u},$$

with $\alpha = 2/3$, the symmetric u-interval $-1 \le u \le 1$ corresponds uniquely to $1/5 \le x \le 5$ (note that $u = (x-1)/(\alpha(x+1))$, and that neither denominators $x+1$ or $1 - \alpha u$ is ever zero on $-1 \le u \le 1$ or $1/5 \le x \le 5$). Moreover, this symmetry produces an *odd* function of u:

$$\ln\left(\frac{1+\alpha u}{1-\alpha u}\right) = 2\alpha u + \frac{2}{3}\alpha^3 u^3 + \frac{2}{5}\alpha^5 u^5 + \frac{2}{7}\alpha^7 u^7 + O\left(u^9\right). \tag{3.21}$$

[10] Henry Briggs (1561–1630) made the first truly useful tables of the real logarithm function. The calculations were carried out, by hand, to astonishing 14-digit accuracy; the tables contained tens of thousands of entries. Nowadays this computing feat seems superhuman; we doubt Robin could duplicate this, though as we will see, better methods will occur in this hypothetical than in the actuality of Briggs' history.

Because $\ln(1/r) = -\ln r$ for real $r > 0$, Robin then saw that only $x > 1$ has to be considered, and then the negative is taken.

Now, this series already gives a fairly decent hand algorithm for computing logarithms when $-1 \le u \le 1$. But Robin still isn't particularly happy, since it doesn't seem accurate enough. We can see that this is true more easily than Robin can: We can graph the relative error of these approximations, for various degrees (see Fig. 3.4).

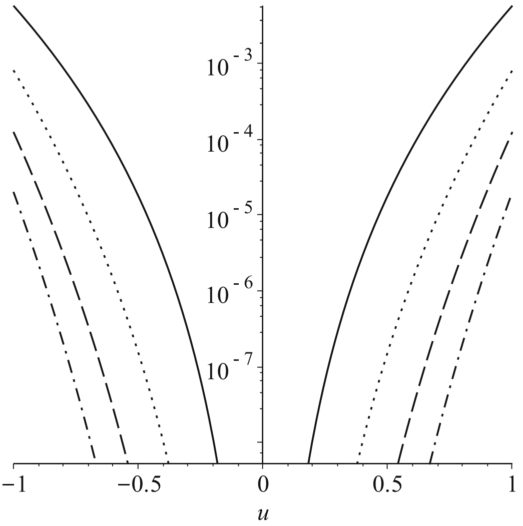


Fig. 3.4 Relative errors in the degree-k Taylor approximations (3.21) for $k = 9$, 13, 17, and 21. Higher-degree approximations have smaller error, as expected, but the errors seem to grow rather quickly near the ends of the interval $-1 \le u \le 1$

Robin is wondering: How can better results be achieved? Given the interval to which our problem has been reduced, the name "Chebyshev" arises from some dim recess of memory; indeed, the interval $[-1,1]$ is the natural domain of Chebyshev polynomials (see Chap. 2). Given that, the goal would be now to produce not a Taylor series approximation, which is good near $u = 0$ and not so good near $u = \pm 1$ (see again Fig. 3.4), but rather a Chebyshev series, with error more or less equally small across the interval. Robin also remembers reading about the *Lanczos* τ *method*, which is a truly simple idea, and well within reach of a desert island computation.[11] The idea (which Lanczos used to find an approximation to the exponential function) is to try to solve a differential equation for our unknown function, here $\ln(x)$, in Chebyshev series, by using a useful property of Chebyshev polynomials: They can be multiplied and integrated very easily.

[11] See the beautiful book Lanczos (1988). Lanczos' method is indeed alive and well in modern numerical analysis, nowadays, but mostly for the solution of partial differential equations; our use of it here is twofold, as we will see.

To implement this method, the simplest differential equation that can be used for $\ln(x)$ is, of course,

$$x\frac{dy}{dx} - 1 = 0\,,$$

subject to the initial condition $y(1) = 0$. The method begins by assigning a Chebyshev series for the derivative. As we mentioned, the natural domain for the Chebyshev polynomials is the interval $[-1,1]$, which means that we should be working in the u-variable, not the x-variable. This only necessitates a change of variable, using ${}^{dy}\!/_{dx} = ({}^{dy}\!/_{du})/({}^{dx}\!/_{du})$, from which we obtain

$$(1-\alpha^2u^2)\frac{dy}{du} - 2\alpha = 0\,. \tag{3.22}$$

This is a purely polynomial differential equation (no transcendental functions; only multiplication, addition, and differentiation). Thus, we need to determine recurrence relations for the product and integration of Chebyshev polynomials. By virtue of their definitions (see Sect. 2.2.3), the products of those polynomials are given by

$$\begin{aligned} uT_0(u) &= T_1(u) \\ uT_k(u) &= \frac{T_{k+1}(u) + T_{k-1}(u)}{2} \qquad\qquad k \geq 1\,. \end{aligned}$$

This can be iterated as follows to find u^2T_k:

$$\begin{aligned} u^2T_0(u) &= \frac{1}{2}\big(T_2(u) + T_0(0)\big) \\ u^2T_1(u) &= \frac{1}{4}T_3(u) + \frac{3}{4}T_1(u) \end{aligned}$$

and

$$u^2T_k(u) = \frac{1}{4}\big(T_{k+2}(u) + 2T_k(u) + T_{k-2}(u)\big) \qquad\qquad k \geq 2\,.$$

Moreover, using the substitution $u = \cos\theta$ (so that $du = -\sin\theta\, d\theta$) and the trigonometric identity $2\cos n\theta \sin\theta = \sin(n+1)\theta - \sin(n-1)\theta$, we find from the definition of T_k that

$$\begin{aligned} \int T_k(u)du = \int \cos(k\cos^{-1}u)\,du &= -\int \cos(k\theta)\sin\theta\, d\theta \\ &= -\frac{1}{2}\frac{\cos(k-1)\theta}{k-1} + \frac{1}{2}\frac{\cos(k+1)\theta}{k+1} + K\,. \end{aligned}$$

Choosing the constant of integration K so that the integral is zero at $u = 0$, and so that the limit as $k \to 1$ gives

$$\int T_1(u)\,du = \frac{1}{4}T_2(u) + \frac{1}{4}T_0(u)\,,$$

we find that

$$\int T_k(u)du = \frac{1}{2(k+1)}T_{k+1}(u) - \frac{1}{2(k-1)}T_{k-1}(u) + \frac{k\sin(\pi k/2)}{k^2-1}\,.$$

So, Robin was right: Products and integrals of Chebyshev polynomials are straightforward.

This being understood, Robin is able to set up and solve a *linear* system of equations for the unknown coefficients a_k in the *derivative* of $y(u)$,

$$\frac{dy}{du} = \sum_{k=0}^{N} a_k T_k(u)\,,$$

and, once identified, integrate the result to get a Chebyshev series for y as a function of u. For example, let us take $N=4$. If we substitute this series in the differential equation (3.22), and if we multiply and expand, we obtain a polynomial equation expressed in the Chebyshev basis:

$$\tau_0 T_0(u) + \tau_1 T_1(u) + \tau_2 T_2(u) + \tau_3 T_3(u) + \tau_4 T_4(u) + \tau_5 T_5(u) + \tau_6 T_6(u) = 0\,,$$

where $\tau_0 = (1-\alpha^2/2)a_0 - \alpha^2/4a_1 + 2\alpha$, and so on for each coefficient. If the polynomial is zero, then each coefficient must be zero. However, notice that even though we have only $N=4$, there are both $T_5(u)$ and $T_6(u)$ terms because u^2T_k adds 2 to the degree, as a result of applying the recurrence relation. But then, setting all seven of these coefficients to zero would give seven equations, not five, and we have only five unknowns a_k. Fortunately for Robin, setting the first five to zero and hoping for the best turns out to work; this is why the τ-method is a good method. Lanczos' great observation was that the resulting degree-N polynomial for y was the *exact* solution of the differential equation

$$(1-\alpha^2u^2)\frac{dy}{du} = 2\alpha - \tau_5 T_5(u) - \tau_6 T_6(u)\,.$$

Moreover, Lanczos observed that we can expect these τ_ks to be *small* (in virtue of some theorems about fast convergence of Chebyshev series); in any case, we can compute them explicitly, and measure them. This is, in embryo, the idea of residual assessment for the solution of differential equations that we will use in great detail later in this book; it is also an excellent method of constructing an accurate approximation to a transcendental function on a fixed interval.

In the case examined, the linear system for the unknowns a_0, a_1, a_2, a_3, and a_4 turns out to be $\mathbf{A}\boldsymbol{\alpha} = \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} -1/2\alpha^2+1 & 0 & -1/4\alpha^2 & 0 & 0 \\ 0 & -3/4\alpha^2+1 & 0 & -1/4\alpha^2 & 0 \\ -1/2\alpha^2 & 0 & -1/2\alpha^2+1 & 0 & -1/4\alpha^2 \\ 0 & -1/4\alpha^2 & 0 & -1/2\alpha^2+1 & 0 \\ 0 & 0 & -1/4\alpha^2 & 0 & -1/2\alpha^2+1 \end{bmatrix}$$

and $\mathbf{b} = [2\alpha, 0, 0, 0, 0]^T$. We will look at general methods for solving such equations in the next few chapters, but here we see that the equations are simple enough to think about doing by hand. Well, with enough sunshine and sand, anyway. Once we are done, with $\alpha = {}^2\!/\!_3$ and $N = 4$, the solution is $[{}^{288}\!/\!_{161}, 0, {}^{12}\!/\!_{23}, 0, {}^{12}\!/\!_{161}]$; that is,

$$\frac{dy}{du} = \frac{288}{161}T_0(u) + \frac{12}{23}T_2(u) + \frac{12}{161}T_4(u)\,,$$

and $\tau_5 = 0$ while $\tau_6 = {}^4\!/\!_{483}$. That is, this polynomial is the exact solution of a differential equation that is only about 1% different to the one that we wanted to solve—not bad for only five terms, of which only three are nonzero!

Finally, Robin decides to try again and again to find a good enough N. One problem with the Lanczos method is that if you decide your error is not small enough, you have to go back to the original system of equations, make it bigger, and solve again; you can't just add another term. Here, we just present the solution with $N = 34$:

$$\begin{aligned}
L_T = {} & \frac{851461103262246}{557288527109761}T_1(u) + \frac{41408833593612}{557288527109761}T_3(u) \\
& + \frac{18124402203606}{2786442635548805}T_5(u) + \frac{2644314644406}{3901019689768327}T_7(u) \\
& + \frac{42866700804}{557288527109761}T_9(u) + \frac{56287506246}{6130173798207371}T_{11}(u) \\
& + \frac{8212236486}{7244750852426893}T_{13}(u) + \frac{399383052}{2786442635548805}T_{15}(u) \\
& + \frac{174807606}{9473904960865937}T_{17}(u) + \frac{25504086}{10588482015085459}T_{19}(u) \\
& + \frac{1240332}{3901019689768327}T_{21}(u) + \frac{542886}{12817636123524503}T_{23}(u) \\
& + \frac{79206}{13932213177744025}T_{25}(u) + \frac{428}{557288527109761}T_{27}(u) \\
& + \frac{1686}{16161367286183069}T_{29}(u) + \frac{246}{17275944340402591}T_{31}(u) \\
& + \frac{12}{6130173798207371}T_{33}(u) + \frac{6}{19505098448841635}T_{35}(u)\,.
\end{aligned} \tag{3.23}$$

How accurate is this answer, in the forward sense? Robin has also computed the leftover $\tau_{36} = -\frac{4}{1671865581329283} \approx -2.4 \cdot 10^{-15}$. Thus, the polynomial is the exact solution of a differential equation that differs from the desired one by no more than this. As a result, by simple integration, Robin knows that

$$y(u) = \ln\left(\frac{1+\alpha u}{1-\alpha u}\right) - \tau_{36}\int_0^u \frac{T_{36}(u)}{1-\alpha^2 u^2}\,du\,.$$

This warrants the conclusion that the *relative* difference between y and the desired logarithm is bounded by $|\tau_{36}|$. In fact, as we can tell using high-precision computation in MAPLE, the error is somewhat better (see Fig. 3.5).

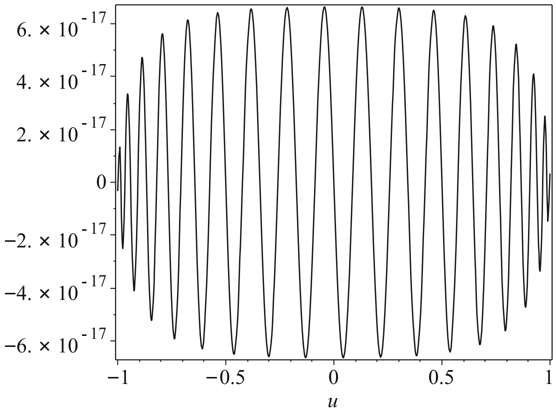


Fig. 3.5 Error in the 17 nonzero term degree-35 Chebyshev series approximation to logarithm on $-1 \le u \le 1$. Note that the error is quite evenly distributed across the interval, but not perfectly "equal-ripple"

There are two issues that remain. First, is this a numerically stable expression? The answer is yes, by the theorem of Smoktunowicz referred to in Chap. 2. Using Clenshaw's algorithm, this polynomial can be evaluated as the *exact* value of a polynomial with only relatively tinily perturbed coefficients (it does so componentwise, so this preserves the zero coefficients).

Second, is the evaluation of this particular polynomial well-conditioned? In other words, how big is $B(u) = \sum_{k=1}^{17} |b_{2k-1}T_{2k-1}(u)|$? When we graph this, we see that it is no larger than about 1.34. That is, an error of about ε in a coefficient translates into an error of about 1.34ε in the value; this is almost perfect conditioning (see Problem 3.19 for a comparison to the monomial basis, which for once does even better). It might be difficult for Robin to ascertain either of these two facts (the desert island library doesn't seem to have all the journals that are needed, and the graphing capabilities on the sand are a little coarse), though they are not hard for us to see.

We can also collapse this theoretical analysis and experimentally examine and assess the *complete* backward error of this formula, in a way that Robin could not. Using the high-precision facilities of MAPLE, and its exponential function, we can compute the *relative backward error*

$$\delta_x = \frac{e^{\mathrm{Lg}(x)}}{x} - 1\,, \tag{3.24}$$

where $\mathrm{Lg}(x)$ is our computed logarithm approximation, done with 15 digits in MAPLE, using the Clenshaw algorithm, and where the exponential function is computed to higher precision (here 30 digits). The result, on $1/5 \le x \le 1$, is shown in Fig. 3.6. Curiously, the backward error is larger on $1 \le x \le 5$, by about a factor of two. In both cases, however, the algorithm appears to have done its job, even in the face of rounding error. Of course, this plot has only *sampled* the relative backward

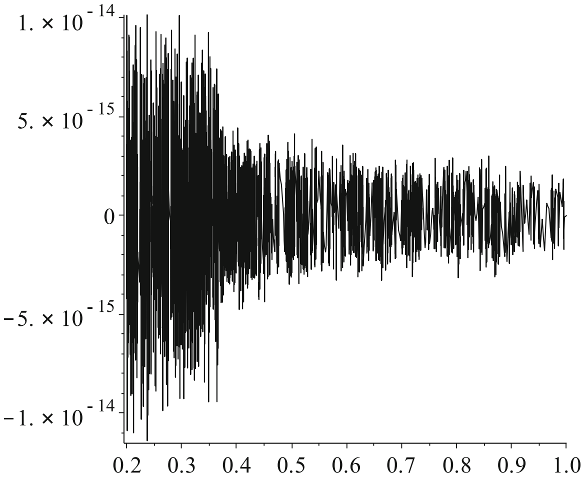


Fig. 3.6 Backward error in the 17-nonzero-term degree-35 Chebyshev series approximation, as computed with 15 digits in MAPLE using the Clenshaw algorithm. Here $1/5 \leq x \leq 1$. Rounding errors are better on this interval than on $1 \leq x \leq 5$, by about a factor of 2. The computed logarithm is seen by this graph to be the *exact* logarithm of $x(1+\delta_x)$, where $|\delta_x| \leq 10^{-14}$, or less than $64\varepsilon_M$

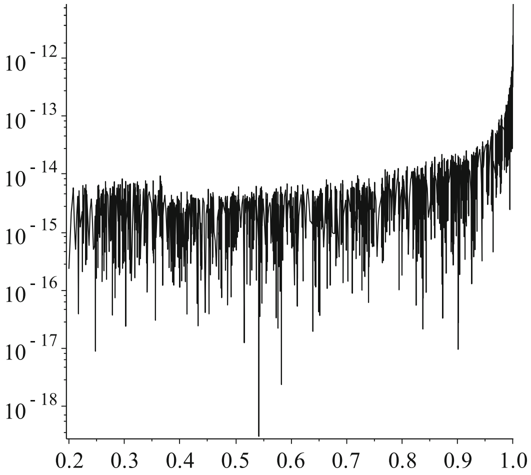


Fig. 3.7 Relative forward error $\mathrm{Lg}(x)/\ln(x) - 1$ in the 17-nonzero-term degree-35 Chebyshev series approximation, as computed with 15 digits in MAPLE using the Clenshaw algorithm. Here $1/5 \leq x \leq 1$. Although the relative backward error is uniformly small on this interval, as seen in Fig. 3.6, the forward error is *not* small near $x = 1$, because the function is ill-conditioned there

error, and it is conceivable that between samples the error skyrockets; but in the face of the theory, this seems quite unlikely. We thus conclude that Robin Crusoe has constructed a method for computing $\log(x)$ that is acceptable by the backward error standard: It computes the exact value of $\log(x(1+\delta_x))$, where δ_x is very small.

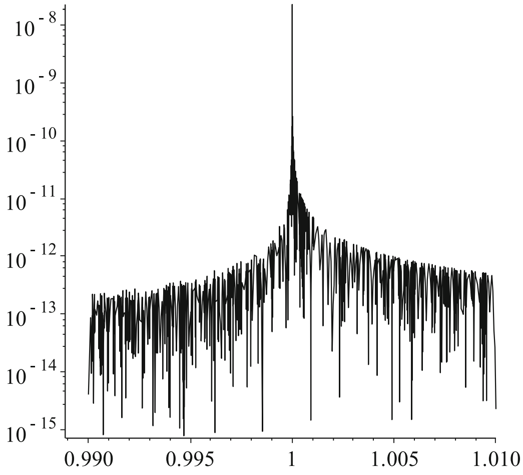


Fig. 3.8 Zoomed: Relative forward error $\mathrm{Lg}(x)/\ln(x) - 1$ in the 17-nonzero-term degree-35 Chebyshev series approximation, as computed in 15 digits in MAPLE using the Clenshaw algorithm. Here $0.99 \le x \le 1.01$. As predicted by the condition number formula $\kappa(\log(x)) = 1/\log(x)$, the forward error seems to be going to infinity at $x = 1$

What about the "gold standard" mentioned on page 105? Does this routine provide a computed value of $\log(x)$ that has small relative *forward* error? No! See Figs. 3.7 and 3.8. Because this function is infinitely ill-conditioned near $x = 1$, tiny rounding errors (or, equivalently, tiny relative backward errors) are amplified by a factor $1/\log(x)$. This is *unavoidable*. Uncertainties in x near 1 will be amplified, full stop. In some sense this is a victory for backward error analysis and a reason to use it; we have judged a method acceptable on the basis of it, and with good reason. This victory, however, does not excuse bad behavior. If one wanted to compute not $\log(x)$ for x near 1 but rather the related function $\log(1+x)$ for x near 0 (the difference is that, if we know, say, 5 figures of each x, in the first case we have 0.9995 and in the second $-0.49832 \cdot 10^{-4}$, which quite different), then it would seem foolish to take the small x, add 1 to it, round the result (thereby throwing away information), and then use an ill-conditioned logarithm to evaluate it, even if the approximation being used there had good backward error.[12]

Similar tricks can be used for all other elementary functions. The next most natural function to want to compute is the exponential function, $u = \exp(z)$. As, first, it is a real function, we can similarly think about reducing the range to (say) $-1 \le x \le 1$ by using $y^2 = \exp(2x)$ repeatedly, and then find a polynomial approximation; see Exercise 3.17. For complex z, we find an equivalence with computing trigonometric functions. Again, argument reduction and polynomial approximation seem to provide a method, although there are difficulties in range reduction for the trigonometric

[12] There is a very clever trick for recovering some of this information, which can be used to do nearly this; see Higham (2002). This trick relies on ensuring that the *correlated rounding errors cancel out*.

functions owing to the transcendence of π and the ill-conditioning of the functions for large argument. Further desert island computations (for arctangent) will be undertaken in Problems 3.20 and 3.21. However, at this point you may resume your previous acceptance of modern computers and their easy and accurate (although not necessarily "correctly rounded") computation of all the elementary functions.

3.3.2 *Roots of Transcendental Functions*

Rootfinding problems for transcendental functions are essentially similar, but some subtleties might enter in the computation. We illustrate this by considering the Lambert W function. The function $W(x)$ is defined by

$$W(x)e^{W(x)} - x = 0\,.$$

See Fig. 3.9. As we see from its definition, the study of this function importantly relates to rootfinding. We have accepted that there are routines for computation of $\exp(\cdot)$, and so computing W given x is a perfectly good rootfinding problem. The rootfinding problem also makes sense over the complex numbers, although in that case there are an infinite number of possible values for $W(z)$ so that $W(z)\exp(W(z)) = z$.

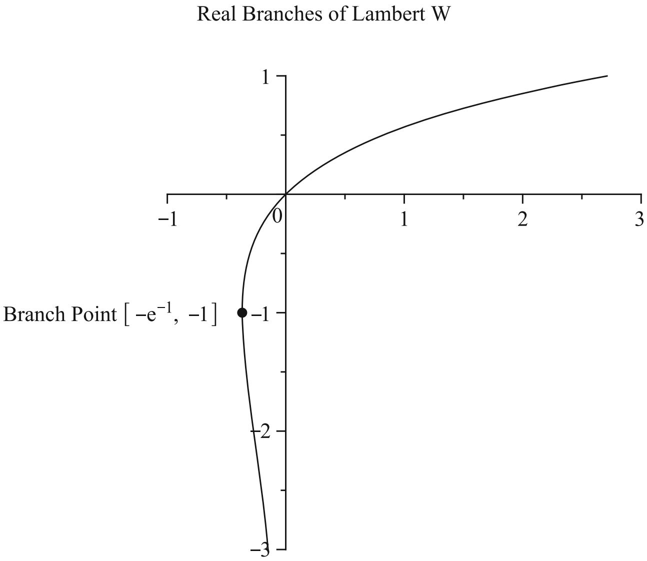


Fig. 3.9 The two real branches of the Lambert W function. This graph can be (nearly, except for the labels and the dot at the branch point) reproduced in MATLAB by using the commands `w = linspace( -3,1,101 );` followed by `plot( w.*exp(w), w, 'k-')`

Newton's method applies immediately to the complex rootfinding problem, since the formula

$$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)} \tag{3.25}$$

can be viewed as coming from the linear Taylor approximation to an analytic function $f(z)$ near $z = z_n$, and is well-studied as to convergence in the complex plane. For Lambert W, we change the name of the variable to w, and thus the iterates are w_n; now z is considered a constant for the duration of the iteration. The iteration is thus

$$w_{n+1} = w_n - \frac{w_n e^{w_n} - z}{(w_n + 1)e^{w_n}} \,. \tag{3.26}$$

Choosing a good initial guess is an issue (see Corless et al. (1996) for a more complete discussion). But if z is near 0, then it makes sense to use an initial guess based on the first few terms of the Taylor series for W, namely,

$$W(z) = z - z^2 + \frac{3}{2}z^3 - \frac{8}{3}z^4 + \frac{125}{24}z^5 + O\left(z^6\right) \,. \tag{3.27}$$

However, if z is large, it makes sense to use an initial guess based on the first few terms of its asymptotic series:

$$\begin{aligned} W(z) \sim \ln(z) - \ln(\ln(z)) + \frac{\ln(\ln(z))}{\ln(z)} \\ + \frac{-\ln(\ln(z)) + \frac{1}{2}(\ln(\ln(z)))^2}{(\ln(z))^2} + O\left((\ln(z))^{-3}\right) \,. \end{aligned} \tag{3.28}$$

For instance, the computation of $W(5+2i)$ might start with that asymptotic series evaluated at $z = 5+2i$ (see the sequence in Table 3.1); starting from this initial guess, Newton's method converges in four iterations (nearly in three).

Table 3.1 Newton iterates and residuals $r_n = z - w_n \exp(w_n)$ for the principal branch of $W(5+2i)$. Note that at every stage, $w_n = W(z + r_n)$ exactly

n	w_n	r_n
0	$1.33231291412843 + 0.244389724585471i$	$0.32494 - 0.12039\,i$
1	$1.36204391549356 + 0.219163426087818i$	$-0.44147 \times 10^{-2} + 0.86916 \times 10^{-2}\,i$
2	$1.36187799739461 + 0.220202860786331i$	$0.60085 \times 10^{-5} + 0.41220 \times 10^{-5}\,i$
3	$1.36187875186406 + 0.220203084384894i$	$-2.6264 \times 10^{-12} - 3.0874 \times 10^{-12}\,i$
4	$1.36187875186369 + 0.220203084384664i$	$-2.3035 \times 10^{-14} - 5.8301 \times 10^{-15}\,i$

What is the condition number of $W(z)$? The function $W(z)$ is defined to be the root w^* (as a function of w, for constant z) of the equation

$$f(w) = we^w - z = 0 \,,$$

and its derivative is

$$f'(w) = (w+1)e^{w}\,.$$

As a result, if we suppose that $\|f\| = O(1)$ on the interval of interest, the condition number is

$$C = \frac{1}{w^*(w^*+1)\exp w^*} = \frac{1}{z(1+w^*)}\,.$$

This expression is problematic when $z = 0$, but only because we are considering relative error. It also has a problem at $w = -1$, that is, at $z = -\exp(-1)$, which is a genuine difficulty—this is a branch point, and convergence is slowed quite a bit: Notice that $f'(x_n)$ appears in the denominator in the error analysis, and notice that for branch points, $f'(x^*) = 0$, and therefore, $f'(x_n)$ will be small. This is how the ill-conditioning affects the convergence: The residuals get quite small near a multiple root of $f(w)$, but the distance to the "true" root remains large.

To conclude, we emphasize again that, in this case, Newton's method has an easy interpretation in terms of backward error. The residual is given by

$$r_n = z - w_n e^{w_n}\,, \tag{3.29}$$

and so every iterate w_n is the *exact* root of a slightly different equation, namely, $f(w) + r_n$. Thus, the interpretation of the computed value comes from the fact that

$$w_n = W(z - r_n)\,. \tag{3.30}$$

That is, each iterate is the *exact* value of the Lambert W function at an increasingly tiny perturbation of the desired argument z. More, this residual r_n is computable (toward the end, we would need more precision to get it accurately, but at least we will know it is small). Finally, since the Lambert W function is well-conditioned (away from its branch point at $z = -1/e$), this small backward error translates into a small forward error.

3.4 Best Rational Approximation

In our investigation of function evaluation and rootfinding, we have made use of polynomials to approximate transcendental functions; specifically, we have used Taylor polynomials and truncated Chebyshev series polynomials. As we have seen, the error in the truncated Chebyshev series was smaller, for the same degree of approximation, and it seemed more equally distributed across the interval of approximation. This fact can be shown rigorously, and it works for rational approximation as well.

Theorem 3.2. *Let $R = P/Q$ be a reduced rational function of degree (n,m); that is, the polynomials $P(x)$ (of degree n) and $Q(x)$ (of degree m) have no common factor. A necessary and sufficient condition that R be the* best approximation *to $f(x)$ in the infinity norm sense on an interval $a \le x \le b$, that is, that*

$$\|R-f\|_\infty = \max_{a\le x\le b} |R(x)-f(x)|$$

is not greater than $\|S-f\|_\infty$ *for any other rational function* $S(x)$ *of degree* (n,m)*, is that the error function* $R(x)-f(x)$ *exhibits at least* $2+\max\{m+\deg P, n+\deg Q\}$ points of alternation *(i.e., values of x for which* $\|R-f\|_\infty$ *is attained, and with alternating sign).*

For a proof, see practically any book on approximation theory. However, we recommend the detailed, extensive, historical and constructive discussion of Trefethen (2013).

This theorem is remarkable in many respects. The most immediately striking thing about it is that it says that the way to get the *best possible answer* is to make sure that the *worst case* occurs as many times as possible—that is, is distributed among as many places as possible. This turns out to be an important idea for approximation: make the error equal-ripple, achieving its maximum as many times as possible in the interval, and you will simultaneously make the total error as small as possible. In some sense it is a technical instantiation of the proverb "many hands make light work"! Speaking of proverbs, the book Trefethen (2013) *also* shows that "best" approximation might not be what you want, nicely illustrating the proverb that "the perfect is the enemy of the good."

As an example, we use an advanced tool in the `numapprox` package in MAPLE to actually find such a best rational approximation to a function; this procedure should seem natural after following the desert island hypothetical on the computation of the logarithm. We consider a function such that

$$\log\left(\frac{1+{}^{2x}\!/_3}{1-{}^{2x}\!/_3}\right) = xF(x^2)\,. \tag{3.31}$$

In MAPLE, we execute the following code:

```
with(numapprox);
minimax(log((1+2*sqrt(x)*(1/3))/(1-2*sqrt(x)*(1/3)))/x^(1/2),
        x = 0 .. 1, [2, 2], 1, 'maxerror');
```

The answer returned, almost instantly, is

$$F(x) = \frac{1.76785424617689+(-0.685178491836968+0.0336567597364259x)x}{1.32589040187406+(-0.710301545774409+0.0780276560350512x)x}.$$

This rational function of degree 2 in the numerator and in the denominator is the *best* degree-$(2,2)$ approximant on this interval. In Fig. 3.10, we see the characteristic "equal-ripple" in the error curve; this characterizes best approximations, as Theorem 3.2 asserts.

Notice that evaluation of the rational approximation $F(x)$ costs only 4 floating-point operations. By rewriting it as a *continued fraction*, we can even reduce it to 2 flops (assuming that a division is really just as cheap as a multiplication). By using the `convert( <>, confrac, x )` function in MAPLE, we find that

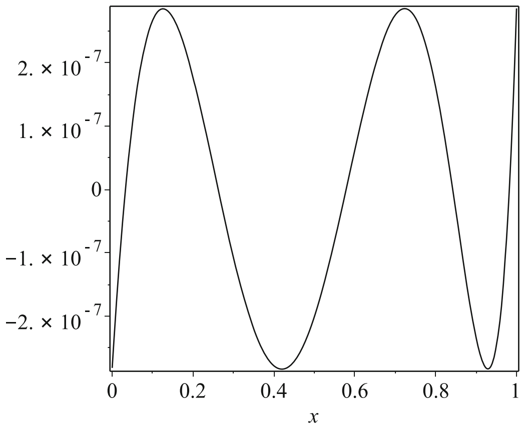


Fig. 3.10 Forward error of the best $(2,2)$ approximant to $x^{-1/2}\ln((1+2\sqrt{x}/3)/(1-2\sqrt{x}/3))$. Note the equal-ripple character: the maximum error is achieved six times in the interval, as required by Chebyshev's theorem

$$F(x) = A_1 - \cfrac{A_2}{x - A_3 - \cfrac{A_4}{x - A_5}}, \tag{3.32}$$

where we find that $A_1 = 0.431343980412622$, $A_2 = 4.85461456921422$, $A_3 = 5.94597503089102, A_4 = 1.78022517753074$, and $A_5 = 3.15722737025930$ are the coefficients returned by the conversion. We now have to worry about two things. First, is the evaluation of this continued fraction numerical stable; that is, is its evaluation exact for slightly perturbed values of each A_k? Second, is this particular expression well-conditioned; that is, how sensitive it is to changes?

We don't have immediately available a standard theorem of backward stability for such expressions (we will see why we don't in a moment), and neither do we have a standard formula for "the" condition number. However, a moment's reflection suggests that the *vector* of condition numbers with entries $(A_k{}^{\partial F}/_{\partial A_k})/F$ will tell us what we want. For example,

$$\frac{A_3{}^{\partial F}/_{\partial A_3}}{F} = -A_3 A_2 \left(x - A_3 - \frac{A_4}{x - A_5}\right)^{-2} F^{-1}\,. \tag{3.33}$$

The other entries are similarly easy to compute in MAPLE. In fact, when we plot these as functions of x by using the values of A_k given above, we find that nowhere on $0 \le x \le 1$ is any of these vector elements larger than 1. We thereby conclude that this expression is *well-conditioned.*

What about numerical stability? It turns out that continued fractions can, in general, be unstable. The combination of division and addition is such that rounding errors cannot (in general) be made equivalent to *relatively* small perturbations to the data (in this case, the A_k and the x); the error bounds include the sums of the absolute values divided by the absolute values of the sums, which can be small. Another way to view the difficulty is that errors made lower down in the fraction (for example, in the computation of $r = x - A_5$) can be revealed by a possible cancellation at a higher level (here, perhaps, $x - A_3 - {}^{A_4}\!/_{r}$). But *in this particular case,* it doesn't happen. Since $0 \leq x \leq 1$, $x - A_5$ must be between about -3.15 and -2.15 and so no cancellation occurs. Hence, $-{}^{A_4}\!/_{(x-A_5)}$ is between 0.56 and 0.82. Continuing, $x - A_3 - {}^{A_4}\!/_{(x-A_5)}$ is between -5.38 and -4.12, and no cancellation occurs here either. Therefore, A_2 over this is between 0.90 and 1.17. Finally, there is no cancellation in subtracting this from A_1, either. Nonetheless, rounding errors *do* occur, and because of the subtractions, they cannot, in general, be accounted for as relatively small perturbations of the A_i or of x. But here, because all of the quantities are of one sign and are of the right magnitude, rounding errors do not accumulate significantly or get revealed by a final subtraction. Indeed, one *can* write a backward error bound for this expression, involving the norm of the A_k vector, and this is sufficient to guarantee a good forward error bound. See Exercise 3.22.

Remark 3.6. By using a degree-$(5,6)$ approximant rather than a degree-$(2,2)$ approximant, we get an equal-ripple error curve that is everywhere less than 8×10^{-17}. This can be converted into a continued fraction that costs only 6 divisions to evaluate and, once again, it is normwise numerically stable. This is considerably cheaper to evaluate than the Chebyshev series that Robin Crusoe found in the story of Sect. 3.3.1. Indeed, we believe that this best rational approximant is more like the methods that are actually used in practice. ◁

Remark 3.7. The equations that determine the equal-ripple curve are multivariate (the maximum error is unknown, and the locations at which the maximum error is achieved are unknown) and nonlinear. The algorithm that the function `minimax` uses to solve these equations is called the Remez algorithm. It is essentially a specialized nonlinear equation solver, a particular multivariate function iteration that starts with an initial guess for all the variables needed, usually derived from a Chebyshev series approximation. It doesn't always converge, but it's pretty good, as is the implementation in MAPLE.

While we were writing this book, Bill Gosper and Warren D. Smith were pushing quite hard on efficient and optimal evaluation of the Gamma function on the interval $[-{}^{1}\!/_{2}, {}^{1}\!/_{2}]$ by applying the Remez algorithm in an interesting way; they were able to achieve 35-decimal-digit accuracy with only 14 parameters, something like a $(7,7)$ approximation. ◁

This kind of rational approximation has been devised and implemented for all elementary functions. As we mentioned, making sure that these elementary functions allow the same guarantee that we have for floating-point arithmetic, namely, that they give the *correctly rounded result*, is a very hard problem, because of the table maker's dilemma. MAPLE, for example, guarantees only that its elementary functions are correct to 0.6 units in the last place (and in order to do so, it chooses quite slow methods to evaluate them).

Finally, we note that the "best approximation" theory is, in practice, not used as often as one might think; while it is valuable for functions that will be used millions (or billions) of times, many functions in applications are not used so often. Simpler methods, such as using interpolation at Chebyshev nodes, often get us nearly as accurate approximations on an interval with comparatively much less effort.

Moreover, there is a theory of best approximation over complex domains similar to that over real intervals; however, again similar to the real case, there is a simpler and almost as efficient alternative. Taylor series and Padé approximants have the useful property of being *near-best* on disks (Geddes and Mason 1975), and we therefore find ourselves using these simple tools in preference to more complicated approximations, most of the time.[13] As an example, consider the function $1/\Gamma(z)$, which is entire. If we wish to approximate it near zero, we can hardly do better than its Taylor series. If we take a series correct to $O(z^{17})$, then the error $|p_{16}(z) - 1/\Gamma(z)|$ has near-circular contours, and it is bounded between $4 \cdot 10^{-9}$ and $7 \cdot 10^{-9}$ on the unit circle. By the well-known result that an analytic function must achieve its maximum magnitude on the boundary of any compact domain, we see that the error is uniformly less than $7 \cdot 10^{-9}$ inside the disk. An alternative might be to interpolate $1/\Gamma(z)$ at 17 equally spaced points on the unit circle (see Chaps. 8 and 9). This gives an error less than 10^{-8} on the contour plotted in Fig. 3.11. While this is nearly as

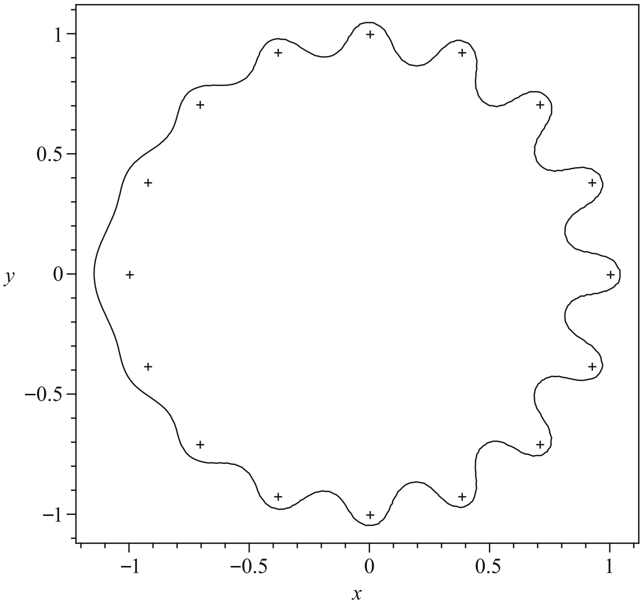


Fig. 3.11 The contour $L_{17}(z) - 1/\Gamma(z) = 10^{-8}\exp(i\theta)$. Inside this contour the Lagrange interpolating polynomial on the 17 nodes (plotted as crosses in the figure) is accurate to an absolute accuracy of better than 10^{-8}

[13] Despite some of its shortcomings, the monomial basis is very good—near-optimal, in fact—if the coefficients don't have too wide a dynamic range—on the unit disk $|z| \leq 1$.

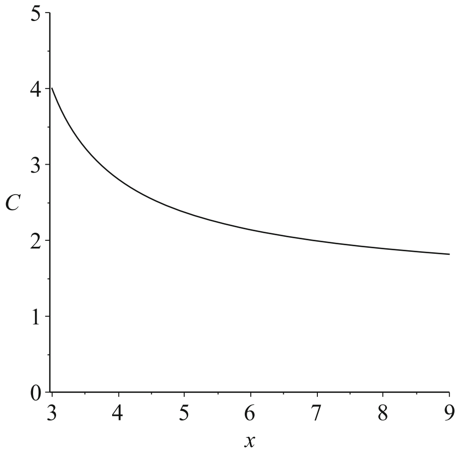


Fig. 3.12 Conditioning of $\ln\Gamma$ on an interval

good as Taylor series (of equivalent degree), it is not quite as good. The optimal approximation (which we have not computed) will be better than Taylor approximation, but not much. For another interesting example, that of $\ln\Gamma(x)$, we obtain Fig. 3.12. However, if we had been working over an interval containing $x = 2$, then the ln function would give trouble near $x = 2$ since $\Gamma(2) = 1$. In general, this kind of thing will be an issue.

The key point is that, with sufficient effort, we can find *good rational approximations to our basic building blocks*, the elementary functions, and this has been done (almost completely). Some nagging issues remain, for *compositions*, because of potential ill-conditioning (e.g., $\ln(1+x)$ *vs* $\ln(x)$).

3.5 Other Rootfinding Methods

In this section, we consider some other useful iterative methods to find the roots of functions. As we will see, there are situations in which their use is advantageous, in terms of computational cost.

3.5.1 Halley's Method

A variation that shares most of the flaws of Newton's method, but converges faster when it does work, is Halley's method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n) - \frac{1}{2}\frac{f(x_n)f''(x_n)}{f'(x_n)}} . \tag{3.34}$$

There is a term in the denominator added to Newton's method, but it provides faster convergence. See, for example, Alefeld (1981), who gives a nice explanation, sufficient conditions for convergence in the real case, and references that show that $(x_{n+1} - x^*) \sim k(x_n - x^*)^3$, namely, ultimately "cubic" convergence to the root x^*.

Because for the Lambert W function the dominant cost in the iteration is the computation of $\exp w$, derivatives are almost free, and this makes Halley's method attractive. Halley's method is used by MAPLE to compute Lambert W.

Example 3.9. Consider the seemingly simple function giving the Pythagorean distance, namely, $d = \sqrt{a^2 + b^2}$, for real a and b. This function is used extremely frequently, for example to find the absolute value of a complex number $z = x + iy$, whence $|z| = \sqrt{x^2 + y^2}$. This function has a bad habit of either overflowing (if one of a or b is larger than the square root of `realmax`, i.e., $1.34 \cdot 10^{154}$) or underflowing (if one of a or b is smaller than the square root of `realmin`, i.e., $1.49 \cdot 10^{-154}$). One might think that this could never happen in a realistic example, but this ignores intermediate computations; it is quite possible, and even likely, that during the course of a computation the size of the intermediate results might be larger or smaller than these limits; as N.J. Higham points out, half of all floating-point numbers lie outside these bounds! In particular, since the Pythagorean function first takes the square and then takes the square root, it is obviously possible that overflow or underflow would prevent us from getting an answer, quite unnecessarily.

Several remedies have been put forward for this, but the one described in Problem 27.6 of Higham (2002), originally due to Moler and Morrison (1983), is particularly interesting and is sketched below. Notionally, one thinks first of computing $a^2 + b^2$; call this p^2, and then p is the quantity that we wish to evaluate (accurately, and avoiding overflow and underflow). If we compute $a^2 + b^2$ to begin with, then we have already lost; so we must continue the analysis a bit. In any case, the quantity we want is a root of the equation $x^2 - p^2 = 0$. We can think of applying Halley's method to this equation:

$$x_{n+1} = x_n - \frac{x_n^2 - p^2}{2x_n - \dfrac{x_n^2 - p^2}{2x_n}} . \tag{3.35}$$

We take $x_0 = a$, and (this is admirably clever) take $y_n = \sqrt{p^2 - x_n^2}$, so that at every step we have $p^2 = x_n^2 + y_n^2$ and initially $y_0 = b$. To make it work, we have to assume $0 < y_0 \le x_0$; if this is not so, interchange the roles of x and y. Then the iteration above can be rewritten as the *pair* of iterations

$$\begin{aligned} x_{n+1} &= x_n \left(1 + \frac{2y_n^2}{4x_n^2 + y_n^2}\right) \\ y_{n+1} &= \frac{y_n^3}{4x_n^2 + y_n^2} . \end{aligned}$$

In this case, overflow and underflow can be avoided by *scaling* these equations: Dividing each of x_n and y_n by the same constant means that each of x_{n+1} and y_{n+1} will also be divided by that constant. In particular, we may take $r_n = ({}^{y_n}/_{x_n})^2$, that is, divide by x_n, and the iterations become

$$\begin{aligned} s &= \frac{r_n}{4+r_n} \\ y_{n+1} &= y_n s \\ x_{n+1} &= x_n(1+2s)\,. \end{aligned}$$

These equations have only a single squaring per step. Since, by assumption, $y_0 < x_0$, it can be proved not to cause a problem thereafter (it cannot overflow and if it underflows, we stop and x_n is the answer). Therefore, it is "immune to underflow and overflow (unless the [final] result overflows)" (Higham 2002). Moreover, it can be shown that $y_n \to 0$ cubically and that $x_n \to p$ cubically, which is the desired constant. Convergence is so rapid, in fact, that if the machine epsilon is larger than 2^{-20}, then no more than three passes are necessary. ◁

Example 3.10. The transcendental equation

$$y + \ln y = z \tag{3.36}$$

has as the solution

$$y = \begin{cases} \omega(z) & \text{if } z \neq t \pm i\pi \quad \text{and } t \leq -1 \\ \omega(z), \omega(z-2\pi i) & \text{if } z = t + i\pi \quad \text{and } t \leq -1 \\ \text{no solution} & \text{if } z = t - i\pi \quad \text{and } t \leq -1 \end{cases}, \tag{3.37}$$

where $\omega(z) := W_{K(z)}(\exp(z))$ is the Wright ω function (see Corless and Jeffrey (2002)) and $K(z) = (z - \ln\exp(z))/(2\pi)$ is the *unwinding number*. This function is a relative of the Lambert W function, but each is useful in different applications. The Wright ω function is better-behaved as $z \to \infty$, and indeed $\omega(z) \sim z$ outside its two rays of discontinuity. However, its interest here is as an example of rootfinding, because Halley's method and higher-order methods can be used on Eq. (3.36) as a way of computing $\omega(z)$, and the function also makes a good example of complex conditioning. In Lawrence et al. (2012), we find that the complex condition number of $\omega(z)$ is just

$$C = \frac{z}{1+\omega(z)} \tag{3.38}$$

by using implicit differentiation. In Exercise 3.10, you are asked to verify this. The function is ill-conditioned in the complex sense only when $\omega(z) \to -1$, which happens at the two singularities $z = -1 \pm i\pi$. The conditioning of the real and imaginary parts separately is also discussed in Lawrence et al. (2012). In Problem 3.13, you are asked to rederive those results and describe the locations where either the real part or the imaginary part of the function is ill-conditioned. These are different and more extensive than just $z=-1\pm i\pi$.[14] ◁

3.5.2 The Secant Method

We now turn to the secant method, which is defined by the iteration

$$x_{n+1}=x_n-\frac{f(x_n)(x_n-x_{n-1})}{f(x_n)-f(x_{n-1})}\,. \tag{3.39}$$

This method uses *two* previous initial guesses and essentially replaces the derivative $f'(x_n)$ in Newton's method by the difference quotient ${}^{(f(x_n)-f(x_{n-1}))}\!/\!{}_{(x_n-x_{n-1})}$. It is thus cheaper to take a secant step than to take a Newton step (usually). However, we will have to take more of them, because it converges more slowly.

Theorem 3.3. *The secant method, when it converges, ultimately has the golden mean as convergence rate.*

Proof. Let $e_n=x_n-x^*$ be the error in the kth iterate. Then

$$e_{n+1}=e_n-\frac{f(x^*+e_n)(e_n-e_{n-1})}{f(x^*+e_n)-f(x^*+e_{n-1})}\,.$$

If we take the Taylor expansion of $f(x^*+e_n)$ and $f(x^*+e_{n-1})$, we find that

$$\begin{aligned} f(x^*+e_n)&=\cancel{f(x^*)}+f'(x^*)e_n+\frac{1}{2}f''(x^*)e_n^2+\cdots\\ f(x^*+e_{n-1})&=\cancel{f(x^*)}+f'(x^*)e_{n-1}+\frac{1}{2}f''(x^*)e_{n-1}^2+\cdots\,, \end{aligned}$$

where the $f(x^*)$ terms are just zero. As a result, we find that

$$\begin{aligned} e_{n+1}&=e_n-\frac{e_n(f'(x^*)+{}^1\!/\!_2f''(x^*)e_n)(e_n-e_{n-1})}{f'(x^*)(e_n-e_{n-1})+{}^1\!/\!_2f''(x^*)(e_n^2-e_{n-1}^2)}\\ &=e_n-e_n\frac{f'(x^*)+{}^1\!/\!_2f''(x^*)e_n}{f'(x^*)+{}^1\!/\!_2f''(x^*)(e_n+e_{n-1})}\\ &=e_n-e_n\frac{f'(x^*)+{}^1\!/\!_2f''(x)(e_n+e_{n-1})-{}^1\!/\!_2f''(x^*)e_{n-1}}{f'(x^*)+{}^1\!/\!_2f''(x^*)(e_n-e_{n-1})}\,, \end{aligned}$$

[14] Lawrence et al. (2012) concentrate heavily on finding a good initial guess function to start the iteration, and *regularize* the problem near the lines of discontinuity. Indeed, the bulk of the paper is on those two aspects. However, some time is spent on the iteration methods that might be used, as well. In addition to considering Newton iteration and Halley iteration, the paper also considers a family of higher-order methods and settles on one of them as being (marginally) the most efficient for that function.

and finally that

$$e_{n+1} = e_n - e_n \left(1 + \frac{{}^1\!/\!{}_2 f''(x^*)}{f'(x^*)} e_{n-1} + \text{h.o.t.} \right) = -\frac{{}^1\!/\!{}_2 f''(x^*)}{f'(x^*)} e_n e_{n-1} + \text{h.o.t.}\,,$$

where “h.o.t.” stands for “higher-order terms.” Thus,

$$e_{n+1} \doteq -\frac{f''(x^*)}{2f'(x^*)} e_n e_{n-1}\,,$$

which is similar to e_n^2. Also,

$$e_n = -\frac{f''(x^*)}{2f'(x^*)} e_{n-1} e_{n-2}\,.$$

Therefore, the ratio $\rho_{n+1} := {}^{e_{n+1}}\!/\!{}_{e_n}$ satisfies the relation $\rho_{n+1} = \rho_n \rho_{n-1}$. Taking logarithms, we obtain $\log \rho_{n+1} = \log \rho_n + \log \rho_{n+1}$. Finally, solving this linear recurrence relation, we see that $\log \rho_n \sim c_1 \phi_1^n + c_2 \phi_2^n$. Now, using $\phi_1 = {}^{1+\sqrt{5}}\!/\!{}_2 = 1.618\ldots$ and $\phi_2 = {}^{1-\sqrt{5}}\!/\!{}_2 = -0.618\ldots$, we see that, asymptotically,

$$\rho_{n+1} \sim e^{c_i \phi^{n+1}} = \left(e^{c_1 \phi^n} \right)^{\phi} = \rho_n^{\phi}\,.$$

As a result, we have

$$\frac{e_{n+1}}{e_n^{\phi}} = \frac{e_n}{e_{n-1}^{\phi}}\,,$$

which is asymptotically constant, that is, $e_{n+1} \sim K e_n^{\phi}$, where ϕ is the golden ratio. ♮

Neumaier (2001) gives an accounting that estimates that if the cost of computing a derivative exceeds 40% of the cost of computing the function itself, then the secant method will usually be cheaper to use.

Example 3.11. Consider the nonlinear equation $f(w) = w + \ln w - 2 = 0$. By inspection, $f(1) < 0$ and $f(2) > 0$, so we know there is a root between 1 and 2. We set $w_0 = 1$ and $w_1 = 2$, and we let

$$w_{n+1} = w_n - f(w_n) \frac{w_n - w_{n-1}}{f(w_n) - f(w_{n-1})}\,. \tag{3.40}$$

In MATLAB, this can be done with a simple one-line function and a loop (but this is not optimized for efficiency—here we call f twice per iteration, which is a waste):

```
f = @(w) w + log(w) - 2;
w = ones(7,1);
w(2) = 2;
for i=3:7,
  w(i) = w(i-1) - f(w(i-1))*(w(i-1)-w(i-2))/(f(w(i-1))-f(w(i-2))
      );
end;
```

```
r = f(w);
plot( abs(r(2:7)./r(1:6).^(1.618)), 'kx','markersize',12 )
xlabel('iteration','fontsize',16);
ylabel('asymptotic␣constant','fontsize',16);
set(gca,'fontsize',16);
```

The final value of the residual is about 10^{-14}, and the values of r_n/r_{n-1}^{ϕ} are plotted in Fig. 3.13. We see that the ratios are bounded. The convergence of the answer is too rapid to really see the rate, but it is plausible that convergence is at a golden-mean rate. ◁

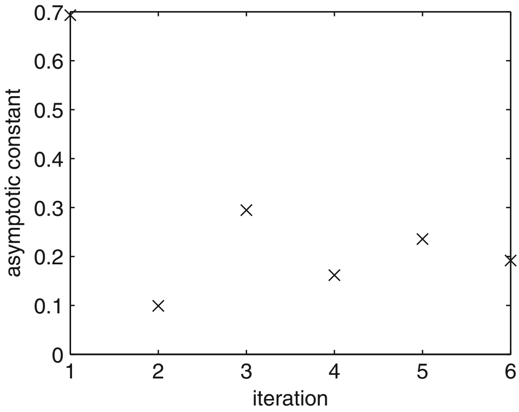


Fig. 3.13 Convergence ratios r_n/r_{n-1}^{ϕ} of the residuals for the secant method applied with $w_0 = 1$, $w_1 = 2$, to $f(w) = w + \log w - 2 = 0$

3.5.3 Inverse Quadratic Interpolation

Inverse quadratic interpolation is an important method that forms a part of the Algorithm `fzero` used by MATLAB.[15] The idea is represented in Fig. 3.14. Assume no two y-values are identical. The function in Fig. 3.14 is quadratic in y and fits the data $(x_0, \eta_0), (x_1, \eta_1), (x_2, \eta_2)$. Here, $\eta_k = f(x_k)$. In Lagrange form, we would thus write

$$x(y) = x_2 + \frac{(y-\eta_1)(y-\eta_2)(x_0-x_2)}{(\eta_0-\eta_1)(\eta_0-\eta_2)} + \frac{(y-\eta_0)(y-\eta_2)(x_1-x_2)}{(\eta_1-\eta_0)(\eta_1-\eta_2)}.$$

Now, consider the point $x_3 = x(0)$, namely, the point where the quadratic cuts the x-axis at $y = 0$; this approximates the root. A simple calculation reveals that

$$x_3 = x_2 + \frac{\eta_1\eta_2(x_0-x_2)}{(\eta_0-\eta_1)(\eta_0-\eta_2)} + \frac{\eta_0\eta_2(x_1-x_2)}{(\eta_1-\eta_0)(\eta_1-\eta_2)}.$$

[15] It is based on work by Richard Brent (e.g., 1973) and earlier work by Dekker (e.g., 1969).

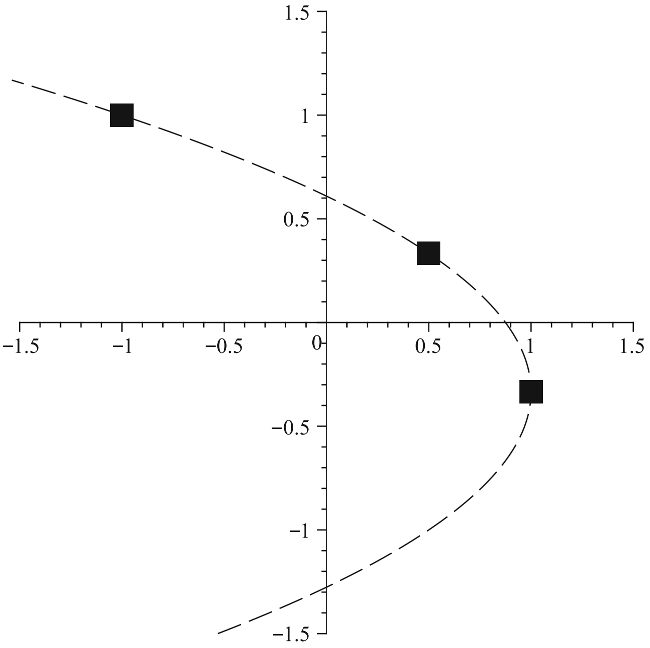


Fig. 3.14 Inverse quadratic interpolation

This can be rephrased in a way that gives a general iteration:

$$x_{n+1} = x_n + \frac{f(x_{n-1})f(x_n)(x_{n-2}-x_n)}{(f(x_{n-2})-f(x_{n-1}))(f(x_{n-2})-f(x_n))} \\ + \frac{f(x_{n-2})f(x_n)(x_{n-1}-x_n)}{(f(x_{n-1})-f(x_{n-2}))(f(x_{n-1})-f(x_n))} .$$

How accurate is this expression? If we let $x_k = x^* + e_k$, where $f(x^*) = 0$, then the first terms of the series analysis gives

$$e_3 = -\frac{1}{6}\frac{e_2e_1e_0\left(D^{(3)}(f)(x^*)D(f)(x^*) - 3D^{(2)}(f)(x^*)^2\right)}{D(f)(x^*)^2} ,$$

so that $e_{n+1} \propto e_ne_{n-1}e_{n-2}$. Taking ratios as before, we obtain the relation $\rho_{n+1} = \rho_n\rho_{n-1}\rho_{n-2}$, and thus the logarithms satisfy the linear recurrence relation $\log\rho_{n+1} = \log\rho_n + \log\rho_{n-1} + \log\rho_{n-2}$. From this, we find that $\theta^3 = \theta^2 + \theta + 1$ becomes important. This equation has the root

$$\theta_1 = \frac{1}{3}(19+3\sqrt{33})^{1/3} + \frac{4}{3(19+3\sqrt{33})^{1/3}} + \frac{1}{3} \doteq 1.83928$$

and two complex roots of modulus about 0.73. Therefore, for large n,

$$\rho_{n+1} \sim \left(e^{c_1\theta_1^{n+1}}\right) = \left(e^{c_1\theta_1^{n}}\right)^{\theta} = \rho_n^{\theta_1}\,, \tag{3.41}$$

from which it follows that, ultimately,

$$\frac{e_{n+1}}{e_n} \doteq \frac{e_n^{\theta_1}}{e_{n-1}^{\theta_1}}\,. \tag{3.42}$$

So again, $e_{n+1}/e_n^{\theta_1}$ is ultimately constant.

As with Newton's method, Halley's method, and the secant method, this method has superlinear convergence; that is,

$$e_{n+1} \doteq k e_n^{1.839\ldots} \qquad \text{for large } n\,. \tag{3.43}$$

It is asymptotically faster than the secant method, but hardly more expensive. Asymptotically, it takes more iterations than Newton or Halley because $1.83928 < 2$, but the steps are cheaper. It is only in exceptional circumstances, when derivatives are almost free, that Newton or Halley will be more efficient.

3.5.4 Taking a Step Back

Now that we have seen a number of methods that can be used to tackle rootfinding problems, let us take a step back and take a look at the tools we have. To begin with, there is no general answer to the question, "Which algorithm should be used?" With regards to efficiency, we need to consider both the rate of convergence and the cost of individual steps. Moreover, all these methods have some common limitations and difficulties. To begin with, these are univariate methods; what about bivariate (or multivariate) problems such as

$$\begin{aligned} f(x,y) &= 0 \\ g(x,y) &= 0 \end{aligned} \quad ?$$

Also, these methods find *one root at a time*; but there might, of course, be more than one. Perhaps even more importantly, these methods may converge very slowly if the initial guess is bad and *may fail to converge* at all. In theory, there are theorems of the form "if x_0 is close enough to x^* and $f(x)$ is smooth enough, then convergence will be obtained" that guarantee convergence. In practice, it is often simplest just to try it and see if it works.

This is why it is often said that correctly using iterative rootfinding methods *is all about the initial guess*. In our experience, the most common question asked by people encountering iterative methods for the first time is, "How do you know where to start?" And, crucially, this is the most important question. A good initial guess means the difference between rapid success and expensive time-consuming failure. To belabor the obvious, nonlinear problems are hard. Globally convergent algorithms are known only for a few classes of problems.

What is done in practice comes from a variety of tricks. The first one is common-sensical enough: *Use your knowledge of the problem.* If you have solved similar problems before, then use their solutions as initial guesses for this one. This leads to the next interesting trick: *If you haven't solved a similar problem before, then invent one and do so now*! This idea of looking at similar problems, when developed rigorously, leads to the very powerful idea of *homotopy*; but we will look in this book only at its simplest expression, that of *simple continuation*. To do so, we embed our problem in a family of problems, with a tunable parameter, as in the following example.[16]

Example 3.12. Suppose we wish to solve the equation

$$y + \ln y - 2 = 0\,,$$

and we have no clue about what to use for an initial guess. Introducing a new parameter, λ, gives us the equation

$$y + \lambda \ln y - 2 = 0\,.$$

We immediately notice that we have our original problem when $\lambda = 1$, and that the case $\lambda = 0$ is easy to solve (we find $y = 2$). One could then try $y_0 = 2$ in the original problem and use a Newton iteration (and, of course, it works on this simple example). Suppose that it didn't work—one could then instead set $\lambda = 0.1$, and hope for success on that (more closely related) problem instead. Indeed, for *very* small λ, success is almost guaranteed. Here this gets us $y(0.1) = 1.93$. We then let $\lambda = 0.2$ and $y_0 = 1.93$ and iterate using Newton's method once more, finding this time that $y(0.2) = 1.87$. Continuing in this way, we find that the solution to our original problem is $y = 1.557$ after 8 more steps. ◁

There is a lot of scope for creativity in this process of introducing a new parameter to simplify things. A generic template is as follows: If you are given a hard problem $H(y) = 0$ to solve, then create an easy problem $E(y) = 0$ and consider the parameterized family $\lambda H(y) + (1-\lambda)E(y) = 0$. Then for $\lambda = 0$, this is an easy problem, and one tracks the zeros for increasing values of λ until one gets to $\lambda = 1$, at which point we have solved the hard problem. However, we should warn you that it doesn't always work as nicely as in our example. As Hamming (1973 p. 77) remarks,

> It is a complex, difficult task to design a foolproof method of tracking zeros since sooner or later almost every possible trouble will occur.

[16] This is taken up further in the section on multivariate rootfinding in this chapter, and in Chap. 12, and again in Chap. 14. For an extensive and thorough treatment of this idea for systems of polynomial equations, see Morgan (1987) and Sommese and Wampler (2005).

3.6 The Multivariate Case

Multivariate generalizations of Newton's method abound. In all cases, the basic idea is to replace the nonlinear problem with a sequence of linear ones. Thus, we have a process of *linearization* and *reduction* of the nonlinear problem to a *sequence* of linear systems of equations. We will leave the discussion of how to solve such linear systems aside, since it is the object of Part II. But for an explicit, motivating example, consider the bivariate case in which we wish to find values of x and y such that both $f(x,y)=0$ and $g(x,y)=0$. Suppose also that we have an approximation (x_n,y_n) already computed or guessed. Then a linear approximation of f and g near that point gives

$$\begin{aligned} f(x,y) &\doteq f(x_n,y_n)+f_x\Delta x+f_y\Delta y \\ g(x,y) &\doteq g(x_n,y_n)+g_x\Delta x+g_y\Delta y\,. \end{aligned}$$

Equivalently, but using vector-matrix notation, we have

$$\begin{bmatrix} f_x(x_n,y_n) & f_y(x_n,y_n) \\ g_x(x_n,y_n) & g_y(x_n,y_n) \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -f(x_n,y_n) \\ -g(x_n,y_n) \end{bmatrix}. \tag{3.44}$$

The matrix is the *Jacobian*, evaluated at the current guess. The right-hand side is the negative of the residual vector, $-[f(x_n,y_n),g(x_n,y_n)]^T$.

Remark 3.8. Notice that, as in the scalar case, the residual vector allows us to reverse-engineer a problem exactly solved by the current guess, namely, $f(x,y)-f(x_n,y_n)=0$, and $g(x,y)-g(x_n,y_n)=0$. Of course, this trivially follows from the definitions; but this obvious fact means that if the residuals are *small*, compared to physical or modeling error, we are done! For all we know, we have a solution that is entirely satisfactory. Of course, we need to know if the multivariate root system is *well-conditioned*, but we need to know this anyway. ◁

To continue with the method, we solve this 2 × 2 linear system for $(\Delta x,\Delta y)$ (again, we will see how to do so in Part II), and then let

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} x_n+\Delta x \\ y_n+\Delta y \end{bmatrix}. \tag{3.45}$$

This iterative process can be repeated as necessary. The generalization to n variables is immediate:

$$\mathbf{x}_{k+1}=\mathbf{x}_k-\mathbf{J}^{-1}(\mathbf{x}_k)\mathbf{F}(\mathbf{x}). \tag{3.46}$$

Remark 3.9. Note that the inverse matrix is (generally) never formed explicitly—instead, the techniques of Part II are used to solve the linear system.

Efficient variations such as BFGS (for optimization problems) or damped Newton iteration (in the solution of differential equations) are used extremely frequently. This is a serious workhorse of numerical computation. ◁

Let us now consider examples of the multivariate version of Newton's method.

Example 3.13. To begin with, consider these equations:

$$f_1 - (f_1 - f_0)\mathrm{sn}^2\left(K(k) - \frac{\pi\sqrt{f_1 - f_2}}{2\sqrt{6}}, k\right) = 0 \tag{3.47}$$

$$-\frac{2}{3}Re = f_2\frac{\pi}{2} + \sqrt{6}\sqrt{f_1 - f_2}\left[E(k) - E\left(\sqrt{\frac{f_1}{f_1 - f_0}}, k\right)\right]. \tag{3.48}$$

They arise in the analytical solution of a problem of fluid flow in a wedge-shaped channel.[17] The unknowns are f_0 and z, and the equations arise in matching the boundary conditions. Once they are found, the velocity profile across the channel can be plotted or otherwise analyzed. The equations have a parameter, the Reynolds number, Re, which is a nondimensional velocity. The equations above are fairly easily solved when Re is small, but become awkward when Re is large, chiefly because we have no useful initial guess for f_0 or z in that case. As discussed in an earlier section, the process known as *simple continuation* is quite helpful: We use the solution for a slightly smaller Re as the initial guess (and, of course, this idea can be used recursively).

There are two difficulties that may strike you on examining Eqs. (3.47) and (3.48). First, they require computation of the Jacobian elliptic functions—which may be unfamiliar—and, second, to use Newton's method, we have to take derivatives—and these too may be unfamiliar. In any case, the equations are quite complicated. It turns out that MAPLE (but not MATLAB[18]) possesses all the requisite resources; in fact, MAPLE even has a built-in multivariate solver called `fsolve`. The algorithm it uses is more complicated than the Newton iteration but is related and has similar characteristics. The execution of

```
fsolve( eval( {e1,e2a}, R=3 ), {f0,z} );
```

returns (once we edit the output to make it more readable)

```
{ f0 = -2.02494131289478, z = -2.02599605566010 }
```

To get this result, MAPLE used an initial guess of $f_0 = 0 = z_0$. However, this answer tells us little about any other roots there might be—and there are many, and indeed many of them correspond to physically realizable flows. Just like Newton's method, `fsolve` has trouble with multiple roots. ◁

[17] See, for example, the discussion in Corless and Assefa (2007).

[18] If we were to try to solve these equations in MATLAB, we would have to first implement the Jacobian elliptic functions and their derivatives (see the Google project at http://code.google.com/p/elliptic/, and note that a partial implementation exists in vanilla MATLAB, `ellipj`).

Rather than pursuing the inspection of Jacobi elliptic functions and related rootfinding problems in more detail, we turn to a simpler example.[19]

Example 3.14. Consider the equations

$$f(x,y) = x^2 + y^2 - 1 = 0$$
$$g(x,y) = 25xy - 8 = 0.$$

In Chap. 6, we will show that a similar equation system, with $25xy - 12 = 0$ instead, has a solution $(x,y) = (3/5, 4/5)$. This will do for our initial guess. The Jacobian is simple enough to do by hand:

$$\boldsymbol{J} = \begin{bmatrix} 2x & 2y \\ 25y & 25x \end{bmatrix} . \tag{3.49}$$

Observe that $\boldsymbol{J}$ is singular when $50x^2 - 50y^2 = 0$, that is, $x = \pm y$. Now, let $(x_0, y_0) = (3/5, 4/5)$. Then the first step of Newton's method is

$$\boldsymbol{J}(x_0,y_0)\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} 6/5 & 8/5 \\ 20 & 15 \end{bmatrix}\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = -\begin{bmatrix} f(3/5, 4/5) \\ g(3/5, 4/5) \end{bmatrix} = -\begin{bmatrix} 0 \\ 4 \end{bmatrix} , \tag{3.50}$$

which has the solution $[\Delta x, \Delta y]^T = [-16/35, 12/35]^T$. This in turn gives

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} 1/7 \\ 15/7 \end{bmatrix} , \tag{3.51}$$

and, using MATLAB, we find the next iterations (see Table 3.2). ◁

Table 3.2 Iterates for the multivariate Newton method for the equation of Example 3.14

Iterates	x_k	y_k
(x_2, y_2)	0.3003	0.9803
(x_3, y_3)	0.3380	0.9427
(x_4, y_4)	0.34030	0.94032
(x_5, y_5)	0.3403312423	0.9403212423
(x_6, y_6)	0.340312423743285	0.940312423743285

This simple example already shows features common to Newton's method applied to larger systems. At each stage we solve a linear system of equations to find the update. Therefore, at each stage we must evaluate the Jacobian matrix at the current guess (which is here trivial but can itself be expensive for larger systems). Also, we must have some reliable way of solving the system (usually simple LU factoring will do, but iterative methods are also common: after all, we don't need a perfect $(\Delta x, \Delta y)$, just a good one). Finally, convergence can initially be slow. Notice that

[19] We return to it when we study boundary value problems in Chap. 14.

the error did not decrease in both components on the first iteration; quadratic convergence only settles in eventually. Here again, the most important thing is getting a good initial guess; all the other tricks are secondary.

3.7 Chebfun for Evaluation and Rootfinding

Before ending this chapter, we briefly discuss the Chebfun package, based on commented examples. Let us see what Chebfun can do for the evaluation of the function $\ln(\Gamma(x))$ on the interval $3 \leq x \leq 4$. The idea is to get Chebfun to construct a best approximant to this function; this turns out to be easy to do since MATLAB has a built-in Γ function that we can use (while pretending that we didn't). Execute

```
lng = chebfun('log(gamma(x))', [3,4] );
length(lng)
```

returns 14. Then we obtain what we want by executing the following code:

```
t = linspace( 3, 4, 3011 );
err = log(gamma(t)) - lng(t);
figure(2), plot( t, err, 'k')
%help chebfun/remez
[p,err] = remez( lng, 8 );
```

The value of err is $1.2714 \cdot 10^{-11}$. We thus continue with this code:

```
[p,err] = remez( lng, 10 );
figure(3), plot( t, p(t)-log(gamma(t)), 'k' )
```

See Fig. 3.15, and, for the sake of comparison, see also Fig. 3.15. As we see, it turns out that Chebfun is *very* good at representing functions—in some sense, that

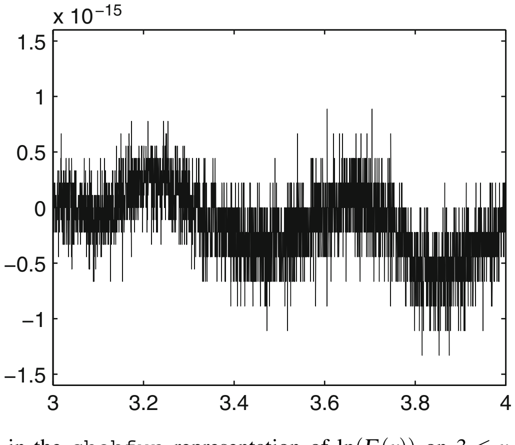


Fig. 3.15 The error in the chebfun representation of $\ln(\Gamma(x))$ on $3 \leq x \leq 4$. The equivalent Chebyshev series has 14 coefficients (and therefore is degree 13)

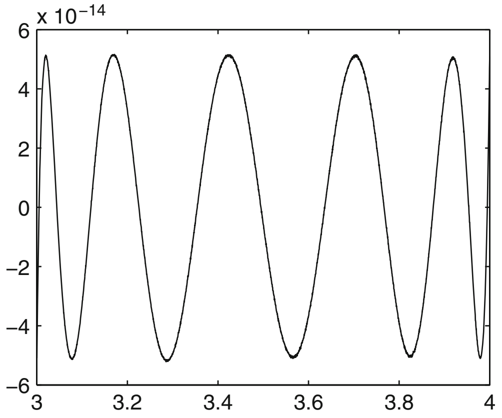


Fig. 3.16 The error in the best possible degree-10 polynomial representation of $\ln(\Gamma(x))$ on $3 \leq x \leq 4$ as computed by Chebfun/remez, rapidly

is what it was designed for—and is, almost incidentally, very good at computing the best (polynomial) approximations, because it has an excellent initial guess to the optimum, namely, the Chebyshev series representation, for the iterative Remez algorithm (Pachón and Trefethen 2009). Interestingly, the rational approximation we found using MAPLE earlier in the chapter, which was of degree $(4,4)$, has fewer free parameters and has slightly better accuracy: It seems that a rational approximation has some advantages over a polynomial approximation. However, the advantage seems marginal.[20]

What about rootfinding? We first look at an easy example, and then a very hard one. Consider again the equation $y+\ln(y)-2=0$. A direct use of the command `roots(y+log(y)-2)` in Chebfun yields a complaint about the logarithmic singularity, which probably can be fixed by the use of various options and flags. An easy alternative is to replace this equation with the equivalent equation $y\exp(y)-\exp(2)=0$ and then executing `y = chebfun('x',[-1,2])`, followed by `roots( y.*exp(y)-exp(2))` yields $\alpha = 1.55715$. The residual is $\alpha+\ln(\alpha)-2 = 4\cdot 10^{-15}$. Thus, it seems that Chebfun can find simple roots of nonlinear equations.

Now, we consider a very hard example. Consider the function $G:[0,1)\to[0,1)$ given by

$$G(x) = \begin{cases} \frac{1}{x} \bmod 1 & x \neq 0 \\ 0 & \text{otherwise} \end{cases} . \tag{3.52}$$

[20] We will see more examples of approximation of functions using Chebfun in Chap. 8, and many more can be found at `http://www2.maths.ox.ac.uk/chebfun/examples/approx/`. A place that Chebfun also shines, however, is in rootfinding. You can find several very impressive examples at `http://www2.maths.ox.ac.uk/chebfun/examples/roots/`.

This function, called the Gauss map, arises in the construction of simple continued fractions.[21] It has an infinite number of jump discontinuities, at $x = {}^1\!/\!_n$, for positive integers n. Because of its connection to the theory of continued fractions, quite a lot is known about it. It seems harsh to try to approximate it by a single smooth polynomial! Boldly, we try it. The command

```
G = chebfun( @(x) mod(1./x,1), [0,1] )
```

succeeds (apparently), with a single smooth polynomial, albeit of length $65,537$. Obviously, it can't be quite right, as polynomials are *not* discontinuous, and certainly can't have an *infinite* number of discontinuities! But the plot looks pretty good (but not great, so we don't show it here). However, when we ask Chebfun to find all the solutions to $G(x) - 0.5 = 0$ by the command

```
roots( G - 0.5 )
```

it seems that Chebfun has met its match at last. On our computer, it went away, thinking, and didn't come back before we lost our patience and hit Ctrl-C to interrupt it.

Well, it really was an unfair test. We didn't even tell Chebfun the function was discontinuous! We can do so, by setting the flag `splitting` to `on`:

```
splitting on;
```

We can also mellow our harshness a bit more by asking for G only on the restricted subinterval $[{}^1\!/\!_n, 1)$, where, say, $n = 1000$. That way, G will only have about a thousand jump discontinuities, not an infinite number (we are giving Chebfun quite a bit of a break with this concession, don't you agree?). But we don't have to be *too* mellow: We can let Chebfun worry about exactly where the singularities are. We mean, we know where they are (at the points ${}^1\!/\!_k$ for positive integers k), and we *could* tell Chebfun, but let's not. Rather, we execute the following code:

```
n = 10^3;
G = chebfun( @(x) mod(1./x,1), [1/n,1] )
plot(G,'k')
```

See Fig. 3.17. This time, Chebfun thinks for a bit before answering but comes back with a 1046-piecewise `chebfun` as its answer. Examining the pieces, we see that it puts breaks at what look to be all the right places—its singularity detection seems pretty good!

Now what about the roots? If we *now* ask for `roots(G - 0.5)`, we get an answer back, within the limits of our impatience. But it turns out that there are still problems: It returns some spurious roots—at the discontinuities. It returns rather a lot of them: In total, we computed 1997 roots, when we were expecting only 999 (one per subinterval); see Fig. 3.18. Chebfun *also* (correctly) located the 998 discontinuities, with the `roots` command. If we use the MATLAB command `find` to select the ξ_j that actually satisfy $|G(\xi_j) - {}^1\!/\!_2| < 0.05$, we find that there are only 999 of them, as expected. Now, in this case, we can show that the reference roots

[21] See, for example, Olds (1963). The discussion here relates to Corless (1992).

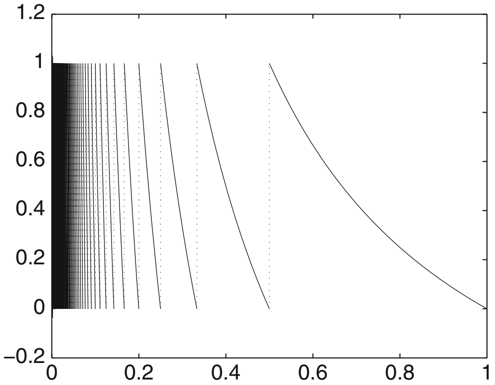


Fig. 3.17 The `chebfun` for the Gauss map on the subinterval $[1/1000, 1]$

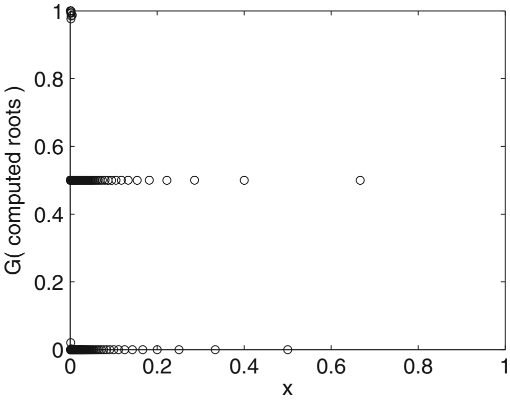


Fig. 3.18 $G(\xi_j)$, where the ξ_j are the results of `roots( G - 0.5 )`. Clearly, not all of the returned ξ_j are actually roots

are $x_j = 1/(j + 1/2) = 2/(2j+1)$ for $j = 1, 2, \ldots, 999$. We can therefore compute the forward errors $\xi_j - x_j$. When we plot them (see Fig. 3.19), we see that the largest relative error is about $5\varepsilon_M$. That is, Chebfun has located all roots of this discontinuous function to essentially full accuracy. We find this impressive.

The `roots` command is based on eigenvalue techniques. It changes bases, from the Lagrange basis on the Chebyshev extreme points to the Chebyshev basis itself (via the FFT, so this isn't expensive). Then it constructs the Chebyshev companion matrices and finds the roots by computing their eigenvalues. Then it throws away roots that are not in the interval being considered. This is a robust and accurate method.

In summary, Chebfun has performed truly remarkably on this very hard example. It successfully located nearly a thousand discontinuities. When asked to find the values where $G(x) = 1/2$, it found them all, to basically full accuracy. And it did so very quickly.

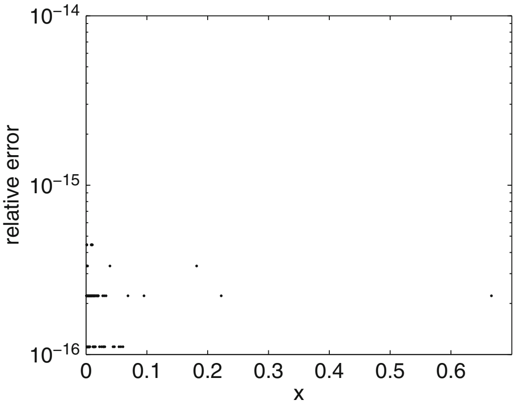


Fig. 3.19 The relative forward errors $|\xi_j/x_j - 1|$

3.8 Notes and References

The polynomial discussed in Sect. 3.2.3 seems to need little introduction; Wilkinson's Chauvenet prize-winning paper Wilkinson (1984) is probably the best possible treatment, but there are hundreds, if not thousands, of other discussions of this apparently innocuous polynomial. It is somewhat surprising to find that in this large volume of work on the polynomial, there are very few explicit computations of its condition number. We rectify that omission in this section.

Evaluation of the complex elementary functions is surprisingly involved once all details are taken care of. See, for example, Hull et al. (1994).

A candidate for a fast method to compute the logarithm would be the arithmetic–geometric mean (AGM) iteration. See Borwein and Borwein (1984) for more details and a description of earlier work.

For a historical survey of families of high-order iterative rootfinding methods, see Petković et al. (2010), who finish by showing that a wide variety of rediscovered methods are actually equivalent to the second method of Schröder, published first in 1870. An attempt at a comprehensive listing of references for polynomial rootfinding can be found in McNamee (1993), which was followed up by McNamee (1997) and McNamee (2007).

Numerical stability of the evaluation of continued fractions is studied in Jones and Thron (1974).

We have talked about near-best approximation of interpolation at equally spaced points on the boundary; that optimality was an Erdös conjecture, proved by De Boor and Pinkus (1978) (for odd n) and Brutman and Pinkus (1980) (for even n).

Problems

Theory and Practice

3.1. Use Newton's method to find the zero of $f(x) = x - \cos(x)$ starting from an initial guess obtained by graphing $y = x$ and $y = \cos(x)$ and visually picking out when the curves intersect.

3.2. Use Newton's method to try to find the (obvious) zero $x = 0$ of $f(x) = x - \sin(x)$ starting with the initial guess $x_0 = 0.1$. This will fail; explain why, in detail. Include in your explanation a plot of $f(x)$ on the interval $-3 \times 10^{-8} \leq x \leq 3 \times 10^{-8}$. Notice that this plot, which may look surprising, shows the correctly rounded result of computing $f(x)$ for x near 0.

3.3. A team of pranksters sneak onto a flat railroad track one cold night and weld an extra 1 foot of track into a mile-long section of track. The next day, as it warms up, the $5280 + 1$-foot-long track expands and bows up into a perfect arc of a circle. How high is the track at the top of the arc?

3.4. A man has a circular field, a pole, a rope, and a goat. He puts the pole firmly into the ground at the edge of the field and ties the rope to the pole and the goat in such a way that the goat is able to eat the grass on exactly half the field. Given the radius of the field, find the length of the rope.

3.5. If you can, find a copy of the Mathematica poster "Solving the Quintic," which uses elliptic functions to solve $n = 5$. Abel and Galois proved that there was no general solution using radicals for the $n \geq 5$ case. Discuss.

3.6. Consider the Airy function $\mathrm{Ai}(z)$ defined in Eq. (1.22) again. Both MAPLE and MATLAB know about $\mathrm{Ai}(z)$ and its derivative, which MATLAB calls `airy(1,x)`. Plot the condition number of $\mathrm{Ai}(z)$ on $0 \leq z \leq 100$. Is $\mathrm{Ai}(z)$ ill-conditioned on that interval? Modestly ill-conditioned? Well-conditioned? Is the Taylor polynomial approximation discussed in Chap. 2 much worse-conditioned than the function itself?

3.7. Compute the overall condition numbers and the separate condition numbers of the real and imaginary parts of each of the following functions:

1. $w = \exp(z)$
2. $w = \sin(z)$
3. $w = \arcsin(z)$
4. $w = \tan(z)$. Note that

$$\tan(x+iy) = \frac{\sin(x)\cos(x)}{(\cos(x))^2 + (\sinh(y))^2} + \frac{i\sinh(y)\cosh(y)}{(\cos(x))^2 + (\sinh(y))^2}\,.$$

Where is each function ill-conditioned overall? Where is the real part of each function ill-conditioned? Where is the imaginary part of each function ill-conditioned? (A computer algebra system might be helpful for this problem.)

3.8. Catastrophic cancellation. Let

$$f(x) = \frac{1-\cos x}{x^2}$$

Execute

```
g = @(x) ( (1-cos(x))./(x.^2) )
x = -4*10^(-8):1e-12:4*10^(-8);
plot( x, g(x), 'k.' ), set(gca,'fontsize',16)
```

in MATLAB. You should get Fig. 3.20.

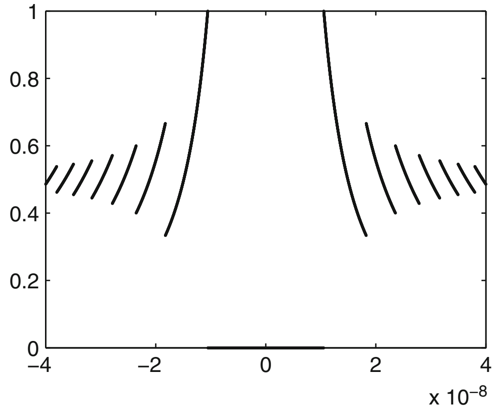


Fig. 3.20 The function $(1-\cos x)/x^2$ should be approximately constant on this interval

However, on that interval, $f(x)$ is approximately constant, nearly equal to 0.5! Indeed, show that

$$|f(x) - \frac{1}{2}| \le \frac{1}{24}(4 \cdot 10^{-8})^4 \le \frac{64}{24} \cdot 10^{-32} \, ,$$

and explain the plot.

3.9. Use MATLAB to compute $\exp(x)\ln(1+\exp(-x))$ on, first, $0 \le x \le 20$, and then on $20 \le x \le 40$, and plot the results, with commands such as these:

```
x = linspace( 0, 20, 1001 );
y = exp(x).*log(1+exp(-x));
plot( x, y, 'k.' )
x = linspace( 20, 40, 1001 );
y = exp(x).*log(1+exp(-x));
plot( x, y, 'k.' )
```

Discuss any surprising results that you see. Try to find a range of t that has similar behavior for $y = t\ln(1+{}^1\!/\!_t)$.

3.10. Show that the condition number of the Wright ω function is given by Eq. (3.38).

3.11. If $a_0 + a_1 r + a_2 r^2 = 0$, find ${}^{\partial r}/_{\partial a_0}$, ${}^{\partial r}/_{\partial a_1}$, and $\frac{\partial r}{\partial a_2}$ without using the quadratic formula. Since

$$\Delta r \doteq \left[{}^{\partial r}/_{\partial a_0}, {}^{\partial r}/_{\partial a_1}, {}^{\partial r}/_{\partial a_2}\right] \begin{bmatrix} \Delta a_0 \\ \Delta a_1 \\ \Delta a_2 \end{bmatrix}, \tag{3.53}$$

identify all cases where small changes Δa_k lead to (relatively) large changes in the roots. Does this contradict Ostrowski's theorem?

3.12. Suppose that you wish to evaluate the function $w = u + iv = {}^{(a+ib)}/_{z}$ at the point $z = c + id$. Find the relative condition number of this function. Then find the relative condition numbers of the real part and the imaginary part separately. Are there any loci where either u or v is ill-conditioned but w is not? In retrospect, is this surprising? See also the discussion in Example 4.15.

3.13. Let $z = x + iy$ and $\omega(z) = u + iv$ (where the latter is the Wright ω function).

1. Show that, away from the lines of discontinuity $x \le -1$, $y = \pm\pi$,

$$x = u + \frac{1}{2}\ln(u^2 + v^2) \tag{3.54}$$

$$y = v + \arctan(v, u) \tag{3.55}$$

2. Show that

$$\begin{bmatrix} {}^{\Delta u}/_{u} \\ {}^{\Delta v}/_{v} \end{bmatrix} = \mathbf{C} \begin{bmatrix} {}^{\Delta x}/_{x} \\ {}^{\Delta y}/_{y} \end{bmatrix} \tag{3.56}$$

where $\left((1+u)^2 + v^2\right)\mathbf{C} =$

$$\begin{bmatrix} \left(u+u^2+v^2\right)\left(\ln\left(u^2+v^2\right)/2+u\right)/u & -v\left(\arctan\left(v,u\right)+v\right)/u \\ \ln\left(u^2+v^2\right)/2+u & \left(u+u^2+v^2\right)\left(\arctan\left(v,u\right)+v\right)/v \end{bmatrix}.$$

The function $\arctan(y, x)$ is the two-argument arctan function that accounts correctly for quadrant information (`atan2` in MATLAB).

3. Describe all the locales in $\mathbb{C}$ where either the real part or the imaginary part of the Wright ω function is ill-conditioned.

3.14. Show that the mixed absolute-relative condition number of rootfinding for an analytic function $w = f(z)$ at a root z_1 is given by ${}^{1}/_{(z_1 f'(z_1))}$ if z_1 is not zero. Find a formula for the separate condition numbers of the real part of the root and of the imaginary part of the root, considered as bivariate real functions $x = x(u, v)$ and $y = y(u, v)$.

3.15. If $w(z) = u(z)v(z)$, so that the series for $w(z)$ is given by Cauchy convolution of the series for $u(z)$ and for $v(z)$, that is,

$$w_k = \sum_{j=0}^{k} u_j v_{k-j}\,, \tag{3.57}$$

find an expression that tells you how the relative change in w_k is related to relative changes in each coefficient u_i and v_ℓ of the multiplicand series. That is, find an expression for the (scalar) change in each w_k induced by changes in the (vectors) of series coefficients for $u(z)$ and $v(z)$. Give an example of a well-conditioned product of series and an example of an ill-conditioned product of series.

3.16. Briggs' method says that

$$\ln(x) \doteq 2^k L_k(x) := 2^k \left(x^{2^{-k}} - 1 \right) . \tag{3.58}$$

Why does this work, and why is

$$\ln(x) \doteq 2^k \left(L_k(x) - \frac{1}{2} L_k^2(x) + \frac{1}{3} L_k^3(x) - \cdots + (-1)^{N-1} \frac{1}{N} L_k^N(x) \right) \tag{3.59}$$

a better approximation when k is large?

3.17. Lanczos' own example demonstrating the τ-method was the computation of the exponential function, $y = \exp(x)$. The differential equation it satisfies is $y' = y$, with initial condition $y(0) = 1$. Consider the interval $-1 \le x \le 1$. Set up and solve, by hand, the equations for $N = 4$ (that is, use $N = 4$ in your Chebyshev series for y'). How accurate is your approximation (use the τ left over, to answer this)? Is evaluation of the Chebyshev form numerically stable? Is the (degree-5) Chebyshev polynomial you get for $y(x)$ well-conditioned to evaluate? Because you have all the coefficients as exact rationals, you may convert to the monomial basis without error. (It turns out to be surprisingly similar, but not identical, to the Taylor polynomial of degree 5.) Is the monomial basis expression better-conditioned than the Chebyshev basis expression, worse-conditioned, or about the same? You may use a computer to draw the graphs of the condition number. Everything else can be done by hand.

Investigations and Projects

3.18. The algorithm Robin chose for the logarithm consisted of taking some number of square roots until the answer was inside the interval $[1/5, 5]$, and then of using the truncated Chebyshev series approximant (found by Lanczos' τ-method) with $N = 34$ terms (half of which were zero). Robin did no serious investigation of whether it would have been better to take more square roots and fewer series terms, or vice versa. From one point of view, because the square roots are computed by some finite number of Newton iterations, the whole scheme can be thought of as a family of rational approximations to the logarithm function. Make a reasonable model of the cost of executing this kind of algorithm, with k square roots and N (even) Chebyshev terms, and give a rationale for choosing k and N. You may choose to optimize the cost of evaluation, but be sure to guarantee that your method of choice is numerically stable.

3.19. In the hypothetical story in the text, Robin Crusoe uses a polynomial approximation for $\log((1+{}^{2u}\!/_3)/(1-{}^{2u}\!/_3))$ that is expressed in the Chebyshev basis, because that's the way it comes out naturally from the Lanczos τ-method. Robin notices, however, that the coefficients in Eq. (3.23) are all exact rationals, and since the Chebyshev basis coefficients are also exact rationals, therefore the polynomial basis may be converted *without rounding errors* to the monomial basis. When Robin does this (there goes another month of sunny afternoons on the island), the polynomial turns out to be

$$\begin{aligned}
L_m = {} & \frac{743051369479680}{557288527109761}u + \frac{2342163497184}{11857202704463}u^3 \\
& + \frac{29355115812768}{557288527109761}u^5 + \frac{65233594664064}{3901019689768327}u^7 \\
& + \frac{3221402159616}{557288527109761}u^9 + \frac{12886831097856}{6130173798207371}u^{11} \\
& + \frac{5716404264960}{7244750852426893}u^{13} + \frac{870244319232}{2786442635548805}u^{15} \\
& + \frac{838993575936}{9473904960865937}u^{17} + \frac{1464995414016}{10588482015085459}u^{19} \\
& - \frac{714130587648}{3901019689768327}u^{21} + \frac{4465524473856}{12817636123524503}u^{23} \\
& - \frac{5984835600384}{13932213177744025}u^{25} + \frac{228438573056}{557288527109761}u^{27} \\
& - \frac{4498441371648}{16161367286183069}u^{29} + \frac{2261300281344}{17275944340402591}u^{31} \\
& - \frac{231928233984}{6130173798207371}u^{33} + \frac{103079215104}{19505098448841635}u^{35}\,.
\end{aligned} \tag{3.60}$$

It can be shown that (surprisingly) in this case the condition number for the monomial basis expression is actually better than the condition number for the Chebyshev basis expression (only by about 34%, which means that rounding errors will be amplified by a factor 1.34 instead of nearly 1, which isn't likely significant, but still). What feature of the coefficients in the monomial expression explains the nearness to 1 of the condition number?

3.20. Use Lanczos' method and the differential equation

$$(1+x^2)\frac{dy}{dx} - 1 = 0 \tag{3.61}$$

subject to $y(0) = 0$ to give a polynomial, expressed in the Chebyshev basis, that approximates $y(x) = \arctan(x)$ on $-1 \le x \le 1$. Note that if $x > 1$, then $\arctan(x) = {}^{\pi}\!/_2 - \arctan({}^{1}\!/_{x})$, reducing the domain; one could reduce it further using the fact that arctan(x) is odd. Choose N so that your answer is accurate to double-precision. You may use MAPLE to help with the computations. Is your polynomial well-conditioned? Convert it to the monomial basis (if you have computed the coefficients as exact rationals; if not, comment on the expected inaccuracy in the resulting monomial basis coefficients if you did). Is the monomial expression well-conditioned?

3.21. Robin also considered an iterative scheme for the computation of real-valued arctan. This problem asks you to recapitulate Robin's steps.

1. Prove that
$$\arctan(v) = 2\arctan\left(\frac{v}{1+\sqrt{1+v^2}}\right). \tag{3.62}$$
2. Show also that the mathematically equivalent form
$$\arctan(v) = 2\arctan\left(\frac{-1+\sqrt{1+v^2}}{v}\right) \tag{3.63}$$
suffers from catastrophic cancellation for small v.
3. Show that the iteration
$$v_{n+1} = \frac{v_n}{1+\sqrt{1+v_n^2}} \tag{3.64}$$
with $v_0 = v > 0$ converges initially quickly but ultimately only linearly to 0, and that $\arctan(v) = 2^n \arctan(v_n)$.
4. This analysis suggests the following iterative algorithm for the arctangent: Use the iteration a few times, say k times, until v_k is small enough that it is easy to compute $\arctan(v_k)$ accurately by using only a few (say N) terms of the Chebyshev series (or even the Taylor series) for arctan. What choices of N and k are "best"? Are there numerical difficulties with this iteration?

3.22. Use the MAPLE command `numapprox[minimax]` (or another package implementing the Remez algorithm) to find a best rational double-precision approximant to $F(u)$ on $0 \le u \le 1$, where
$$\arctan(x) = xF(x^2). \tag{3.65}$$

Convert your result to continued fraction form. Compare the cost and stability of evaluating this expression to that of the Chebyshev series in Problem 3.20. If this function is to be evaluated millions of times, is its construction worthwhile?

3.23. Kepler's equation $\theta - e\sin\theta = M$ is an interesting example of an equation we desire to solve for θ given the eccentricity e and the mass M. Take $e = 0.083$ and $M = 1$ and use a program similar to that on page 116 (or better—that program can certainly be improved) to draw a picture of the fractal boundary of the basins of attraction for Newton's method for this equation. We found Fig. 3.21. Have fun with this one!

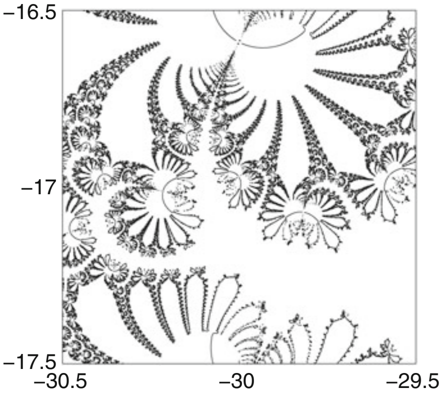


Fig. 3.21 Using contour plot to draw a fractal boundary for the Kepler equation. The *half-circles* are artifacts

3.24. Compute the evaluation condition numbers $B(z)$ for the Wilkinson polynomial of degree 20 expanded in the monomial basis, and plot $B(z)/|p(z)|$ on $0 \le z \le 21$. Do the same for a similar polynomial of degree 30, defined by

$$p(z) = \prod_{k=1}^{30}(z-k)\,. \tag{3.66}$$

To accurately evaluate the ratio, you will need higher precision than is available in MATLAB. Note that the condition number of both of these polynomials in a Lagrange basis on (say) $\tau_0 = 0$ and $\tau_k = k$ is just 1. Plot also the rootfinding condition numbers $B(r)/|rp'(r)|$ at the roots $r = 1,2,\ldots,30$.

3.25. Split $\exp(x)$ into its even and odd parts, $\exp(z) = \cosh(z) + \sinh(z)$, where, as usual, $\cosh(z) = \frac{1}{2}(\exp(z)+\exp(-z))$ and $\sinh(z) = \frac{1}{2}(\exp(z)-\exp(-z))$. If one has an accurate method to evaluate both $\cosh(z)$ and $\sinh(z)$, suppose, for example, a "correctly rounded" method whereby the computed value of $\cosh(x)$ for a machine number x is guaranteed to be the correctly rounded value, that is, $\cosh(x)(1+\delta)$ for some $|\delta| < \mu_M$ and similarly for $\sinh(x)$, can one then accurately evaluate $\exp(x)$ by the split? What happens as $x \to -\infty$? Are any of these functions especially ill-conditioned?

3.26. The function $f(u) = \sqrt{-2\log\cos u^2}/u^2$ is difficult to evaluate numerically near $u = 0$. Compute its condition number and show it is actually well-conditioned. Show experimentally that the formulation $y = \cos(u^2)$, and then if $y = 1$ to all bits return 1 and otherwise return $\sqrt{-2\log y}/\cos^{-1}(y)$, is more expensive but more accurate (using the built-in `acos` to compute the arccosine). This formulation is due to Kahan (1980). Compare it with use of the series

$$1 + \frac{1}{12}u^4 + \frac{3}{160}u^8 + \frac{209}{40320}u^{12} + O\left(u^{16}\right) \tag{3.67}$$

for small u. You may take more terms if you like, but the function has a singularity at $u = \pm\sqrt{\pi/2}$ anyway. Show that the function is ill-conditioned at its singularities.

3.27. Compute the condition number of the Bessel function $J_0(z)$. Write a program to compute also the condition numbers of the real and imaginary parts separately. The derivative of $J_0(z)$ is $-J_1(z)$. These are `besselj(0,z)` and `-besselj(1,z)`, respectively. Are there any regions where $J_0(z)$ or its parts are ill-conditioned?

3.28. Show that both $\sin\theta$ and $\cos\theta$ are ill-conditioned for large values of θ. One proposal to improve the preservation of identities such as $\sin^2\theta + \cos^2\theta = 1$, due to W. Kahan, is to consider instead the functions $\sin(\pi t)$ and $\cos(\pi t)$. Notice that the zeros and maxima of the trigonometric functions occur at integer or half-integer values of t, and so in a machine environment, this may offer opportunities to recognize these points. Discuss this proposal.

Part II
Numerical Linear Algebra

This part of the book gives an overview of some of the material contained in works that aim to give a more complete, self-contained presentation of numerical linear algebra, such as Wilkinson (1963), Golub and van Loan (1996), Demmel (1997), Trefethen and Bau (1997), Meyer (2001), Higham (2002), and Hogben (2006). That last book, for example, has over 70 chapters, many of them relevant for numerical linear algebra, and is highly recommended for the reader as supplementary material for this course. It's twice as long as this book, though! We do not try to duplicate that material here, although there is some overlap with the numerical chapters there, of course. Instead, the goal of this part, being part of an introductory course, is only to open doors for you; you'll have to walk through on your own.

The main emphasis of the treatment here is to place the analysis of numerical methods for linear algebra completely in the framework of backward error analysis. As some readers will already know, backward error analysis was first used in numerical linear algebra; Wilkinson credits Givens as the first person to do so, but it is generally agreed that Wilkinson was the first to use it with the generality it deserves. John von Neumann seems to deserve some credit for the complementary notion of *condition number*, which again first appears in numerical analysis in the field of linear algebra.

This part of the book assumes that the basics of theoretical linear algebra are known to the reader. The Handbook referred to above is an excellent source for filling in gaps. Some important notions are reviewed in Appendix C, for convenience.

This part is perhaps the most important in the book because, sooner or later, every computation comes down to linear algebra. However, the number and length of the chapters is not proportional to their importance; instead, we limit ourselves to a "motivated list of facts" approach, together with some discussion of the meaning of the algorithms; for lack of space, there are very few proofs. We also use MATLAB as an exemplar of a collection of high-quality numerical linear algebra software tools; of course there are others. In particular, MAPLE uses a high-quality LAPACK backend to provide efficient and robust numerical linear algebra, and can be used with confidence.[22]

If we look at linear algebra from the point of view of its applications, four equations prove to have a central importance[23]:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{II.1}$$

$$\mathbf{A}^H\mathbf{A}\mathbf{x} = \mathbf{A}^H\mathbf{b} \tag{II.2}$$

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \tag{II.3}$$

$$\frac{d\mathbf{u}}{dt} = \mathbf{A}\mathbf{u} \tag{II.4}$$

The first two equations will be treated together in Chap. 4; Eq. (II.2)—the so-called *normal equations*—will be examined in the context of the solution of overspecified

[22] See Anderson et al. (1999) and Hogben (2006 chapter 72) for useful references.

[23] This point is emphasized by Strang (2002).

systems $\mathbf{Ax} \approx \mathbf{b}$.[24] Equation (II.3) determines the nature of eigenvalue problems, which will be examined in Chap. 5. Equation (II.4) arises from the study of linear systems of differential equations, which will be treated in Part IV.

Suppose we are given a matrix $\mathbf{A} \in \mathbb{C}^{m\times n}$ and a vector $\mathbf{b} \in \mathbb{C}^m$. A central problem in linear algebra consists in finding vector(s) $\mathbf{x} \in \mathbb{C}^n$ such that $\mathbf{Ax} = \mathbf{b}$. There are two main classes of methods for solving such linear systems of equations:

1. Direct methods for dense matrices, such as Gaussian Elimination (which is equivalent to an LU factoring, called a Turing factoring if $\mathbf{A}$ is rectangular) or Gram–Schmidt orthogonalization (which is equivalent to a QR factoring);
2. Iterative methods, which are especially suited to sparse matrices (see Chaps. 6 and 7).

To solve problems involving dense matrices, we typically factor the matrices numerically so that the factors have structural properties that make solution easy. The factorings should be *numerically stable*, that is, be the exact factors of a matrix near to the original one. Moreover, the factors should be reasonably cheap to obtain, and the use of the factors to solve the problem should be similarly cheap (or cheaper). Some of the most important factorings in linear algebra are the LU factoring and the Jordan Canonical Form $\mathbf{A} = \mathbf{XJX}^{-1}$. Nonetheless, as it will become clear, the most important factorings for *numerical* linear algebra include the QR factoring, the Schur factoring $\mathbf{A} = \mathbf{QTQ}^H$ and the singular value decomposition. The latter factoring will allow us to introduce two of the most important notions, namely, conditioning of a matrix problem and numerical rank.

Before we begin, we note that most books present the real-number case, $\mathbf{A} \in \mathbb{R}^{m\times n}$, and expect the reader to be able to switch to $\mathbb{C}$ when necessary. For various reasons, we will do the opposite; we assume that $\mathbf{A} \in \mathbb{C}^{m\times n}$, and switch to $\mathbb{R}^{m\times n}$ when we want simpler examples. Accordingly, instead of talking about the transpose $\mathbf{A}^T$ to a matrix $\mathbf{A}$, we will talk about the complex conjugate transpose (alternatively, Hermitian transpose) matrix $\mathbf{A}^H$. If $\mathbf{A} = [a_{ij}] \in \mathbb{C}^{m\times n}$, $\mathbf{A}^H = [\bar{a}_{ji}] \in \mathbb{C}^{n\times m}$. This is denoted `A'` in MATLAB. The real transpose, unconjugated, is accessed *via* `A.'`. In addition, we note that the word ‘orthogonalization’ is often used in the literature in a way that includes the complex case. We will stick to the common terminology, instead of using the more peculiar term ‘unitarization.’

[24] The normal equations will mostly be left as exercises, since we wish the emphasize the importance of using the QR factoring. As explained by Stewart (1998 77), “[i]t is hard to argue against the normal equations in double precision. [. . .] On the other hand, if one wants a general-purpose algorithm to run at all levels of precision, orthogonal triangularization is the winner, because of its stability.” However, we note in passing that the normal equations provide a theoretical approach and, in many cases, especially when $\mathbf{A}$ is sparse and well-conditioned, a practical approach as well.

Chapter 4
Solving Ax=b

Abstract This chapter first shows how to solve $\mathbf{Ax} = \mathbf{b}$ in the simple cases in which $\mathbf{A}$ is unitary or triangular, and then explains how the QR factoring can be used to reduce other problems to these simple cases. We show that these methods are *backward stable*; that is, they exactly solve a slightly perturbed problem. In order to understand how these small perturbations affect the solution, we then introduce the crucial notion of *condition number* in relation to the most important factoring, namely, the *singular value decomposition* (SVD). We also examine the LU factoring (equivalent to Gaussian elimination) and a number of applications of the main factorings. We end the chapter with a short discussion of *nonlinear* systems. ◁

The main goal of this chapter is to give the reader confidence when choosing a method for solving linear systems numerically: that is, confidence in assessing the method's reliability and in assessing the problem's sensitivity to data error. Implicit in our objective of giving the reader confidence is that he or she should not have been too overconfident to begin with. Often, even for simple problems, the numerical results can be very surprising.

Example 4.1. Consider the linear system

$$\mathbf{Bx} = \begin{bmatrix} 888445 & 887112 \\ 887112 & 885781 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \mathbf{b}\,. \tag{4.1}$$

It is easy to verify that

$$\mathbf{x} = \begin{bmatrix} 885781 \\ -887112 \end{bmatrix}.$$

Moreover, since $\det(\mathbf{B}) = 1 \neq 0$, this solution is unique. Now, what happens if you solve the problem in MATLAB? In MATLAB, this can be achieved by typing the command `B\b`, which returns

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_4,

$$\hat{\mathbf{x}} = \begin{bmatrix} 885644.0223037927 \\ -886974.8164771678 \end{bmatrix}.$$

The result is different, even in the thousands place! ◁

In what follows, we will examine one way to understand such surprising events, and see what the options are if you face such a situation.

In order to achieve this goal, we will apply the general concepts and strategies articulated in Chap. 1 (the next few paragraphs may be omitted if you have not yet read Chap. 1—indeed you may as well skip ahead to Sect. 4.1). Our objective is to solve systems of linear equations of the form $\mathbf{Ax} = \mathbf{b}$ for the vector of unknowns $\mathbf{x}$; the reference problem handled in this chapter can thus be represented by the map

$$\varphi : \langle \mathbf{A}, \mathbf{b} \rangle \to \{\mathbf{x} \mid \phi(\mathbf{A}, \mathbf{b}, \mathbf{x}) = \mathbf{0}\}, \tag{4.2}$$

where $\phi(\mathbf{A}, \mathbf{b}, \mathbf{x}) = \mathbf{b} - \mathbf{Ax} = \mathbf{0}$ is what we called the defining function. Often, we will restrict our attention to problems where $\mathbf{A}$ has a particular structure. A numerical method computes an approximate solution $\hat{\mathbf{x}}$ such that $\mathbf{A}\hat{\mathbf{x}} \approx \mathbf{b}$. Alternatively, from the point of view of backward error analysis, we can say that the numerical method computes the value of an engineered map $\hat{\varphi}$ for the input data $\mathbf{A}$ and $\mathbf{b}$, as in Fig. 4.1a. For this problem, the forward error is $\Delta\mathbf{x}$. For a backward error analysis, we are interested in finding a perturbation of the input data that amounts to the error occasioned by using the numerical method computing $\hat{\varphi}$. Since the input data is a pair of objects, there are three possible kinds of backward error: Only $\mathbf{A}$ is perturbed; only $\mathbf{b}$ is perturbed; and both $\mathbf{A}$ and $\mathbf{b}$ are perturbed. Following the literature, we will focus on the former case in the beginning of the chapter. Along this line, a backward error analysis seeks the smallest matrix $\mathbf{E}$ (for "error matrix") such that $\varphi(\mathbf{A}+\mathbf{E}) = \hat{\varphi}(\mathbf{A})$, that is, such that the diagram in Fig. 4.1b commutes. By default, "smallest" will be based on the matrix 2-norm, but we will sometimes use other norms (see Appendix C.2). Later in the chapter, we will see that there are great advantages in doing backward error analysis with perturbation of $\mathbf{b}$ only. The main advantage is to make it easy to combine a priori and a posteriori error analyses.

a

$$\begin{tikzcd} \langle \mathbf{A}, \mathbf{b} \rangle \arrow[r, "\varphi"] \arrow[dr, "\hat{\varphi}"'] & \mathbf{x} \arrow[d, dash, "\Delta\mathbf{x}"] \\ & \hat{\mathbf{x}} \end{tikzcd}$$

b

$$\begin{tikzcd} \langle \mathbf{A}, \mathbf{b} \rangle \arrow[r, "\varphi"] \arrow[d, "\mathbf{E}"'] \arrow[dr, "\hat{\varphi}"'] & \mathbf{x} \arrow[d, "\Delta\mathbf{x}"] \\ \langle \mathbf{A}+\mathbf{E}, \mathbf{b} \rangle \arrow[r, "\varphi"'] & \hat{\mathbf{x}} \end{tikzcd}$$

Fig. 4.1 Commutative diagrams for the solution of $\mathbf{Ax} = \mathbf{b}$. (**a**) Engineered problem and forward error. (**b**) Backward error diagram

As explained in Chap. 1, the main tool by which we will assess the reliability of our method a posteriori is the *residual*. More formally, for the reference problem under consideration, the residual is the vector

$$\mathbf{r} = \phi(\mathbf{A}, \mathbf{b}, \hat{\mathbf{x}}) = \mathbf{b} - \mathbf{A}\hat{\mathbf{x}}\,, \tag{4.3}$$

that is, what is left over when we substitute our computed solution back into the original equation. Because we can compute $\hat{\mathbf{x}}$ by means of the numerical method we are analyzing, and because we can compute matrix–vector products and vector differences, the residual is easily computable. A posteriori error analyses will thus be very handy.

The main tool by which we will assess the sensitivity of our problem is called the *condition number*. We note immediately that, despite the fact that we write "*the* condition number," there are many condition numbers, corresponding to differing vector and matrix norms that are called for in differing physical situations. Condition numbers were first studied in numerical linear algebra, and the theory is still best-developed here, but we have condition numbers in every chapter of this book, for all kinds of computational problem. For the linear algebraic case, we will make heavy use of the so-called singular value decomposition (or SVD).[1] This factoring is extremely important, and we will discuss it in detail in Sect. 4.6.

This being said, one might wonder: Why not solve linear systems exactly, instead of solving them numerically and then analyze the error? We begin this chapter by addressing this question. Then, we will turn to a discussion of how to solve the simplest kinds of linear systems and, from there, how to solve less simple linear systems. At the end, we will briefly discuss nonlinear systems.

4.1 Why Not Solve Linear Systems Exactly?

We saw in Example 4.1 that floating-point solutions of even very small systems can surprise the person doing the computation. One response in keeping with our support for the point of view "I don't care how quickly you give me the wrong answer" is to abandon floating-point and do only exact arithmetic.

Let's consider only a small subset of the possibilities here, and look at the problem of doing exact rational arithmetic on matrices that contain rational entries and, more importantly, *have no data errors*. For the moment, we ignore square roots, other algebraic numbers, and also transcendental numbers like π. Let's further simplify things and suppose that we start with simple rational numbers whose numerators and denominators are no more than (say) d decimal digits long. We could simplify even further and suppose that we are dealing only with integers, and that helps more than a little in computation but not completely since we fairly quickly

[1] The word "decomposition" is used in the theory of functions: A function $f(z)$ can be decomposed into two (usually simpler) functions if it can be written as $f(z) = (g \circ h)(z) = g(h(z))$. For linear transformations, a decomposition like this is equivalent to factoring the matrix of the linear transformation. We will usually use the word "factoring" in this book (and eschew the longer word "factorization"); and we can get away with this for the QR and the LU factorings, but for the SVD the word "decomposition" is indeed entrenched.

are forced back to rational numbers (solution of linear systems gives us rational numbers immediately, in general, and the denominator must be a divisor of the determinant).

When we try to find an exact solution, we seem to run immediately into a problem from the other point of view: "I don't care how right your answer is, if it takes a hundred years to get it." The complexity—that is, the minimal possible cost—of solving linear systems with rational coefficients must take into account the length of the exact answer. The solution of a linear system of equations must necessarily allow for the exact determinant showing up in the answer (you can easily convince yourself of that by solving a few random linear systems in MAPLE, for example). And the length of the determinant grows with the dimension of the system. Experiments show (see Problem 4.13) that the length of the decimal representation of the determinant seems to be of size nd, where n is the dimension of the matrix (this growth can be estimated using what is known as the Hadamard bound), while we started with d-digit numbers.[2] In contrast, a floating-point answer takes only constant storage, regardless of the dimension. The *cost* of doing arithmetic on exact rational numbers depends on how long they are as well, whereas again floating-point operations cost the same no matter what the floating-point number is. The best-known algorithms for multiplication of d-digit rational numbers have a cost proportional to $d \log d$; hence, nd-digit number operations will cost $O(nd\log(nd))$.

The true complexity of solving a linear system of equations of dimension n is not known; the algorithms we study in this book take $O(n^3)$ operations, but there are faster algorithms that take $O(n^\omega)$ operations, where ω is known to be at least 2, but the least-known ω is, at this time of writing, a little bigger than 2.37. However, with exact arithmetic, the *operations* have a cost that depends on the length of the operands (and it depends on n as well), here $O(nd\log(nd))$ by the observation above. Thus, the total cost of the exact solution using the algorithms we talk about here is not $O(n^\omega)$ but apparently rather $O(n^\omega nd\log(nd))$. For $n = 1000$, this is already more than a thousand times as expensive as floating-point solution, *and that's not taking into account the speed of special-purpose hardware for floats*.

Except in special circumstances, we just cannot afford to wait for the exact solution. Moreover, we would be wasting our time anyway, since the floating-point solution is (as described in this chapter) the *exact* solution of a nearby problem—a much cheaper one to solve, but just as good from the modeling context. Even in the pure mathematical context, once you add back in numbers like π or square roots, you must approximate the initial data, and those errors have consequences.

All this being said, a lot of progress has been made recently on exact algorithms. Dixon (1982) gives a p-adic $O(n^3\log^2 n)$ algorithm[3] that is actually practical, and Storjohann (2005) shows how the exponent 3 can be reduced to ω as above; if this is done, then these algorithms become more competitive with floating-point at least for some problems (one might well be willing to live with a factor of a hun-

[2] In fact, this is a tight bound, and it is quite close to the average case.

[3] The idea of the algorithm in Dixon (1982) is quite strongly related to the algorithm in Sect. 7.1, and there is a catch to the cost in that it depends on something like the "growth factor" that we will study in Sect. 4.7. So, even if we do want to use exact algorithms, numerical ideas will help.

dred increase in cost, for instance). As a result, they become quite attractive in some circumstances, such as when there are no data errors and no approximations made initially. However, once approximations are introduced into the *data* (e.g., even by replacing π by a finite approximation, even if there are no actual measurements in the data), then the advantage of exact computation is already lost. Hence, we need to study the effects of error propagation, anyway.

4.2 Solving Unitary or Orthogonal Systems

Let us begin our study of numerical solutions of systems of linear equations with a particular case of reference, Problem (4.2), namely, the case in which $\mathbf{A}$ is unitary. If $\mathbf{A}$ is unitary, that is, $\mathbf{A}^H\mathbf{A} = \mathbf{I}$, then solving $\mathbf{Ax} = \mathbf{b}$ is very simple, since

$$\mathbf{Ax} = \mathbf{b} \quad \Rightarrow \quad \mathbf{A}^H\mathbf{Ax} = \mathbf{A}^H\mathbf{b} \quad \Rightarrow \quad \mathbf{Ix} = \mathbf{A}^H\mathbf{b} \quad \Rightarrow \quad \mathbf{x} = \mathbf{A}^H\mathbf{b}\,. \tag{4.4}$$

If the matrix $\mathbf{A}$ is known explicitly, then computation of the conjugate transpose seems simple enough.[4] It merely consists in rearranging the entries of the original matrix followed by complex conjugation. Thus, the cost of the solution seems to be one matrix multiplication, plus whatever it costs for the rearranging and conjugation, which we normally think of as negligible. Hence, a quick estimate of the cost in terms of the number of floating-point operations (flops) is $O(n^2)$.

Now, we turn to the more important question: Is this method numerically stable in floating-point arithmetic? That is, does floating-point multiplication by $\mathbf{A}^H$ give the exact solution of a nearby problem? The answer is yes. It is quite straightforward to show that this is true in good floating-point arithmetic systems.[5] Matrix multiplication is usually implemented as a collection of inner products (of the rows of $\mathbf{A}^H$ with the column $\mathbf{b}$), so that the method described is Algorithm 4.1. Thus, we only need to examine the stability of inner products.

Algorithm 4.1 Solving a system with a unitary matrix

```
Require: A unitary $n \times n$ matrix $\mathbf{A}$ and an $n \times 1$ vector $\mathbf{b}$
  for $i$ from 1 by 1 to $n$ do
    $x_i := \mathbf{A}^H(i,:) \cdot \mathbf{b}$
  end for
  return The vector $\mathbf{x}$ solving $\mathbf{Ax} = \mathbf{b}$
```

To begin with, observe that inner products are not necessarily *relatively* accurate, in the forward sense: If the value of the inner product in exact arithmetic is 0,

[4] However, not all linear systems have matrices that are so known. For example, some matrices can be accessed only by calling a subroutine to perform matrix–vector multiplication; we reserve discussion of this case until later, in Chap. 6. Similarly, the case of a very large matrix $\mathbf{A}$ is not so straightforward either owing to memory issues.

[5] Readers who have not yet looked at Chap. 1 may postpone reading the rest of this subsection.

then the result computed in floating-point arithmetic might be infinitely different. However, more importantly, inner products turn out to be backward stable, in that the computed inner product $\mathbf{a}\cdot\mathbf{b}$ is the exact value of $(\mathbf{a}+\Delta\mathbf{a})\cdot\mathbf{b}$ (with a perturbation wholly in one of the vectors—we get to choose which one) and that, moreover, the perturbations $\Delta\mathbf{a}$ are very small. As we have shown in Chap. 1, Theorem 1.2, it turns out that $|\Delta a_i| \leq \gamma_n |a_i|$ if real arithmetic is used, which is a relatively small perturbation.[6]

This leads us to the following theorem:

Theorem 4.1. *Assume* $\mathbf{A}$ *is a machine-representable unitary matrix. The method of Algorithm 4.1 is backward stable in floating-point arithmetic; that is, it exactly solves* $(\mathbf{A}+\Delta\mathbf{A})\mathbf{x}=\mathbf{b}$*, where each entry of* $\Delta\mathbf{A}$ *is relatively small.*

Proof. Left as Exercise 4.1.

This works because each row of $\mathbf{A}^H$ can be considered as an independent vector, and all the rounding errors in the inner product with $\mathbf{b}$ to obtain the corresponding result can be considered as perturbations in that row independently, without touching the vector being multiplied. Therefore, the algorithm for matrix multiplication is backward stable. However, this supposes that each row of $\mathbf{A}^H$ is independent. That is, this part of the argument fails if the entries are correlated, because the rounding errors need not be correlated. We return to this in a later chapter, but for now we suppose entries in $\mathbf{A}$ are uncorrelated. This being understood, we conclude that solving unitary linear systems is expected to be easy, even numerically.[7]

Example 4.2. In this example we extend the algorithm a bit: The matrix $\mathbf{A}$ we choose will have $\mathbf{A}^H\mathbf{A}$ equal to a diagonal matrix but not the identity. Specifically,

$$\mathbf{A} = \begin{bmatrix} -3+i & 1-i \\ 2+i & 2 \end{bmatrix}.$$

Suppose the right-hand vector $\mathbf{b} = [1/3, -1/3]^T$. Then, in MATLAB, we compute $\mathbf{x} = \mathrm{diag}(1/15, 1/6)\mathbf{A}^H\mathbf{b}$, because $\mathbf{A}^H\mathbf{A} = \mathrm{diag}(15,6)$, not the identity. This gives[8]

$$\begin{bmatrix} -0.1111111111111111 \\ -0.05555555555555555 + 0.05555555555555555\,i \end{bmatrix}.$$

[6] The γ_n notation is introduced in Chap. 1. By definition, $\gamma_n = n\mu_M/(1-n\mu_M)$, where μ_M is the unit roundoff, that is, half the machine epsilon ε_M. If complex arithmetic is used, the bound increases to γ_{n+2}, which is only marginally larger.

[7] A caveat that should be kept in mind is that, in Theorem 4.1, $\mathbf{A}+\Delta\mathbf{A}$ is not in general unitary or nearly unitary.

[8] The MATLAB solution is represented internally in binary. The solution as printed is in decimal arithmetic. The decimal solution, cut and pasted into a MAPLE worksheet, was what was analyzed. Conversion to decimal introduces the possibility of further rounding errors, which we ignore here.

A short computation in MAPLE (not shown here—you will learn the method later) shows this to be the exact solution of $\hat{\mathbf{A}}\mathbf{x} = \mathbf{b}$, where $\hat{\mathbf{A}}$ has entries $\hat{a}_{i,j} = a_{i,j}(1+\delta_{i,j})$, with the matrix of $\delta_{i,j}$ approximately equal to

$$\begin{bmatrix} 0.74+0.25\,i & -0.25\,i \\ 0.77-0.39\,i & 0.39+0.39\,i \end{bmatrix} \mu_M\,.$$

We remind you that $\mu_M = 2^{-53} \approx 10^{-16}$. Each of these perturbations is less than μ_M in size. The bound from Theorem 4.1 was larger, being γ_{k+2} for some k. Thus, this (essentially) unitary matrix equation has a numerical solution that is the exact solution of a nearby linear system of equations. ◁

Remark 4.1. If $\mathbf{A}$ is unitary but not machine-representable, then the true inverse of the rounded matrix is not obtained by transposing. We have $\hat{\mathbf{A}}^H\hat{\mathbf{A}} = \mathbf{I}+\mathbf{E}$, and each entry of $\mathbf{E}$ is $O(n\mu_M)$. We will ignore this complication, which can be interpreted as increasing the size of the relative forward error by $\|\mathbf{E}\|$, which is $O(\mu_M)$. See Exercise 4.22. ◁

4.3 Solving Triangular Systems

If $\mathbf{A}$ is *lower-triangular*, that is, if it is equal to some matrix $\mathbf{L}$ of the form

$$\mathbf{L} = \begin{bmatrix} \ell_{11} & & & & \\ \ell_{21} & \ell_{22} & & & \\ \ell_{31} & \ell_{32} & \ell_{33} & & \\ \vdots & \vdots & \vdots & \ddots & \\ \ell_{n1} & \ell_{n2} & \ell_{n3} & \cdots & \ell_{nn} \end{bmatrix}, \tag{4.5}$$

and if $\ell_{kk} \neq 0$ for all $1 \le k \le n$, then we may easily solve $\mathbf{Lx} = \mathbf{b}$ by *forward elimination*, as described by Algorithm 4.2. This includes, by the way, the even easier case when $\mathbf{A}$ is diagonal, in which case all $\ell_{kj} = 0$ for $k > j$. The process is simple. First, find x_1 from $\ell_{11}x_1 = b_1$, namely, $x_1 = {}^{b_1}/_{\ell_{11}}$. Then find x_2 from a rearrangement of the second equation, $\ell_{22}x_2 = b_2 - \ell_{21}x_1$, or $x_2 = {}^{(b_2-\ell_{21}x_1)}/_{\ell_{22}}$. Obviously, the process continues with x_3, and so on. At each stage only one new unknown needs to be solved for, and we need $\ell_{kk} \neq 0$ at every step. As a result, we find that the components of $\mathbf{x}$ are

$$x_i = \frac{b_i - \sum_{k=1}^{i-1} \ell_{ik}x_k}{\ell_{ii}}\,, \tag{4.6}$$

for $i = 1, 2, \ldots, n$, in turn. Note that if $\ell_{kk} = 1$ for all ks, then we say that the matrix is *unit* lower-triangular.

Algorithm 4.2 Forward elimination to solve a nonsingular lower-triangular system

```
Require: A lower-triangular $n \times n$ matrix $\mathbf{L}$ with $\ell_{kk} \neq 0$ $(1 \leq k \leq n)$ and an $n \times 1$ vector $\mathbf{b}$
  for $i$ from 1 by 1 to $n$ do
    $x_i := b_i$
    for $j$ from 1 to $i-1$ do
      $x_i := x_i - \ell_{ij} x_j$
    end for
    $x_i := {}^{x_i}/_{\ell_{ii}}$
  end for
  return The vector $\mathbf{x}$ solving $\mathbf{Lx} = \mathbf{b}$
```

Similarly, if $\mathbf{A}$ is upper-triangular, that is, if it is equal to some matrix $\mathbf{U}$ of the form

$$\mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ & u_{22} & u_{23} & \cdots & u_{2n} \\ & & u_{33} & & u_{3n} \\ & & & \ddots & \vdots \\ & & & & u_{nn} \end{bmatrix}, \tag{4.7}$$

and if $u_{kk} \neq 0$ for all $1 \leq k \leq n$, then we may easily solve $\mathbf{Ux} = \mathbf{b}$ by *back substitution*, as described by Algorithm 4.3. Starting from x_n and proceeding backward but otherwise in a similar fashion, we find that the components of $\mathbf{x}$ are

$$x_i = \frac{b_i - \sum_{k=i+1}^{n} u_{ik} x_k}{u_{ii}}, \tag{4.8}$$

for $i = n, n-1, \ldots, 1$, in turn.

The cost of back substitution and forward elimination, in terms of the number of floating-point operations, is

$$n + 1 + 2 + 3 + \cdots + (n-1) = \frac{1}{2}n(n-1) + n = \frac{1}{2}n^2 + O(n) = O(n^2) \text{ flops}. \tag{4.9}$$

Algorithm 4.3 Back substitution for a nonsingular upper-triangular system

```
Require: An upper-triangular $n \times n$ matrix $\mathbf{U}$ with $u_{kk} \neq 0$ $(1 \leq k \leq n)$ and an $n \times 1$ vector $\mathbf{b}$
  for $i$ from $n$ by $-1$ to 1 do
    $x_i := b_i$
    for $j$ from $i+1$ to $n$ do
      $x_i := x_i - u_{ij} x_j$
    end for
    $x_i := {}^{x_i}/_{u_{ii}}$
  end for
  return The vector $\mathbf{x}$ solving $\mathbf{Ux} = \mathbf{b}$
```

Remember that a "flop" is one multiplication or division together with one addition or substraction. Note that a *multiplication* by a triangular matrix costs

$$n+(n-1)+\cdots+1=\frac{1}{2}n(n+1) \text{ flops}\,, \tag{4.10}$$

which is about the same. Thus, we observe that for triangular matrices, solving and multiplying costs about the same.

As we see, the problem of solving $\mathbf{Ax}=\mathbf{b}$ when $\mathbf{A}$ is triangular is straightforward. But are these algorithms numerically stable? The answer is an emphatic *yes*.

Theorem 4.2. *Given a real system* $\mathbf{Tx}=\mathbf{b}$ *with a nonsingular triangular matrix* $\mathbf{T}$*, the methods of Algorithms 4.2 and 4.3 are backward stable in floating-point arithmetic; that is, they exactly solve* $(\mathbf{T}+\Delta\mathbf{T})\mathbf{x}=\mathbf{b}$*, where each entry of* $\Delta\mathbf{T}$ *is relatively small. Specifically,* $|\Delta\mathbf{T}_{ij}|\le\gamma_n|T_{ij}|$*. Complex-valued systems have a slightly larger bound,* $\gamma_{n+9}|T_{ij}|$*, because complex division, done in a way to avoid overflow or underflow, incurs a rounding penalty of* $\sqrt{2}\gamma_7$ *at most; but this happens only once per equation, and otherwise, each component of the solution is an inner product.*

The proof of this theorem is similar to the inner-product proof sketched earlier, in Theorem 1.2.[9] Consequently, the algorithms compute the *exact* solution of a triangular system that differs only by about rounding level from the original—and zero elements are not disturbed. That is, these algorithms are both componentwise backward stable, which is just as good as the stability result for solving a unitary system. In fact, it's about as good as it gets in numerical analysis.

Example 4.3. Consider the upper-triangular matrices $\mathbf{U}$ with diagonal entries 1 and all entries in the strict upper triangle -2; that is,

$$u_{i,j}=\begin{cases}1 & i=j\\ -2 & j>i\end{cases}\,.$$

Moreover, let the right-hand-side vector $\mathbf{b}$ be defined by

$$b_i=\begin{cases}1 & i \text{ is odd}\\ -{}^1\!/\!_3 & i \text{ is even}\end{cases}\,.$$

Let us first consider the 4×4 case. By construction, the exact solution of the linear system $\mathbf{Ux}=\mathbf{b}$ is $\mathbf{x}=[1,-1,1,-1]^T/3$.

Does it agree with the results obtained with MATLAB? We can simply execute this:

```
U = diag(ones(1,n)) - triu(2*ones(n),1);
b = [1,-1/3,1,-1/3]';
x = U\b;
```

Then, by executing `3*x-[1,-1,1,-1]'`, we find a numerical estimate of the forward error, which in this case is

[9] For full details of the real case, see Higham (1989b, 2002).

$$3\Delta x = 3\hat{\mathbf{x}} - 3\mathbf{x} = 10^{-15}\left[0.4441\ 0.2220\ 0\ 0\right]^T ,$$

which shows that the forward error for this small matrix is about what we expect.

However, this example was carefully chosen to show some bad behavior for larger n.[10] If we increase the size of the matrix just to $n = 32$, then the forward error is not the same size as the machine epsilon. Generate the system as follows:

```
b = ones(n,1);
for j=2:2:n,
     b(j) = -1/3;
end;
U = diag(ones(1,n)) - triu(2*ones(n),1);
x = U\b;
```

Then, if we inspect Δx_1 by running `x(1)-1/3` (where the exact x_1 is $^1\!/\!_3$), we find that about 0.004, or 13 orders of magnitude larger than the machine epsilon! Note that this is in spite of the excellent backward error: The computed solution is *guaranteed* to be the exact answer of a linear system whose matrix differs only by about 10^{-15} from the specified one. ◁

Something very strange is going on in this example! Once we get to the SVD in Sect. 4.6, we will begin an explanation; for now, we will put this (very important!) issue aside and continue with our discussion of simple algorithms.

4.4 Factoring as a Step Toward Solution

Although the problem of solving $\mathbf{Ax} = \mathbf{b}$ is simple for unitary and for triangular systems, we nonetheless have to face the fact that most systems are not unitary or triangular. Our strategy will then be to reduce these more difficult problems to sequences of simpler problem that we know how to solve. To this effect, our main tool will be *matrix factorings*: If we can *factor* the matrices, then we can generate a product of matrices with special structure that is equal to the original matrix. Generically, factoring a matrix $\mathbf{A}$ into simpler factors $\mathbf{F}_1$ and $\mathbf{F}_2$ such that $\mathbf{A} = \mathbf{F}_1\mathbf{F}_2$ allows us to convert the problem

$$\mathbf{Ax} = \mathbf{b} \tag{4.11}$$

into a sequence of simpler problems:

1. First solve $\mathbf{F}_1\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$;
2. Then solve $\mathbf{F}_2\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$.

This works because matrix multiplication is associative. Denoting $\mathbf{F}_2\mathbf{x}$ by $\mathbf{y}$, we find that

$$\mathbf{Ax} = (\mathbf{F}_1\mathbf{F}_2)\mathbf{x} = \mathbf{F}_1(\mathbf{F}_2\mathbf{x}) = \mathbf{F}_1\mathbf{y} = \mathbf{b}. \tag{4.12}$$

[10] The example is borrowed from Higham (1989b).

Given that many factorings provide easily solved systems, this simple idea is very powerful. For instance, if $\mathbf{F}_1$ is lower-triangular and $\mathbf{F}_2$ is upper-triangular, then solving $\mathbf{Ax} = \mathbf{b}$ only requires the use of Algorithms 4.2 and 4.3.

With this strategy in mind, we will investigate and use a number of factorings:

The QR factoring In this case, $\mathbf{A} = \mathbf{QR}$, where $\mathbf{Q}$ is unitary (in the real case, orthogonal) and $\mathbf{R}$ is upper-triangular (i.e., *R*ight-triangular). See Sect. 4.5.

The SVD "SVD" stands for singular value decomposition: $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H$, where $\mathbf{U}$ and $\mathbf{V}$ are unitary (in the real case, orthogonal) and $\boldsymbol{\Sigma}$ is diagonal. See Sect. 4.6.

The LU factoring The basic form is $\mathbf{A} = \mathbf{LU}$, but it also includes the variants $\mathbf{PA} = \mathbf{LU}$ and $\mathbf{PA} = \mathbf{LD}^{-1}\mathbf{UR}$, where $\mathbf{P}$ is a permutation matrix, $\mathbf{L}$ is lower-triangular, $\mathbf{D}$ is a nonsingular diagonal matrix, $\mathbf{U}$ is upper-triangular, and $\mathbf{R}$ is the unique reduced row echelon form of $\mathbf{A}$. These factorings correspond to variants of the well-known Gaussian elimination. See Sect. 4.7.

Notice the reoccurrence of triangular, diagonal, and unitary matrices. These factorings are useful precisely because matrices with such structures are extremely convenient. Many other useful factorings exist, and we will introduce some of them in due course. The idea of these factorings is always that they allow us to solve problems (not necessarily $\mathbf{Ax} = \mathbf{b}$) by decomposing them into simpler problems.

4.5 The QR Factoring

Let us begin our journey with the QR factoring. Given an equation $\mathbf{Ax} = \mathbf{b}$ in which $\mathbf{A}$ is not triangular, one cannot directly apply back substitution or forward elimination. That is where the QR factoring comes in; as we will explain, the QR factoring will in many cases be the method of choice to solve $\mathbf{Ax} = \mathbf{b}$. If we can factor $\mathbf{A}$ as $\mathbf{A} = \mathbf{QR}$, where $\mathbf{Q}$ is unitary (orthogonal, if $\mathbf{A}$ is real) and $\mathbf{R}$ is upper-triangular, then the problem simplifies significantly, since it becomes $\mathbf{QRx} = \mathbf{b}$, which is decomposed as follows:

1. Solve $\mathbf{Qy} = \mathbf{b}$ for $\mathbf{y}$;
2. Solve $\mathbf{Rx} = \mathbf{y}$ for $\mathbf{x}$.

Both steps are simple, in that we can use the algorithms just discussed. Solving $\mathbf{Qy} = \mathbf{b}$ for $\mathbf{y}$ only involves the computation of $\mathbf{Q}^H\mathbf{b}$ since $\mathbf{Q}$ is unitary, and as we have seen, this poses no problem once $\mathbf{Q}$ is known. Moreover, solving $\mathbf{Rx} = \mathbf{y}$ is simply achieved by back substitution, since $\mathbf{R}$ is upper-triangular.

Thus, the original problem can be solved by solving two simpler subproblems in sequence; each of these subproblems comes with a backward error guarantee. Note that we do *not* (at this point in this textbook) know whether the overall problem comes with a backward error guarantee. We reserve discussion of this point until later, but note for reference that backward error in *general* does *not* compose well in this fashion: It is certainly possible to fail to have good backward error overall even though each subproblem does; consider the outer product as in Problem 1.19, for example; no backward error is possible there even though each subcomputation has excellent backward error.

Example 4.4. Suppose that we know that the three-by-three matrix $\mathbf{A}$ factors as below:

$$\begin{bmatrix} 2 & 1 & 1 \\ -1 & 1 & 1 \\ -1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 2/\sqrt{6} & 1/\sqrt{21} & 2/\sqrt{14} \\ -1/\sqrt{6} & 4/\sqrt{21} & 1/\sqrt{14} \\ -1/\sqrt{6} & -2/\sqrt{21} & 3/\sqrt{14} \end{bmatrix} \begin{bmatrix} \sqrt{6} & \sqrt{6}/3 & 0 \\ 0 & \sqrt{21}/3 & \sqrt{21}/7 \\ 0 & 0 & 3\sqrt{14}/7 \end{bmatrix}, \tag{4.13}$$

where $\mathbf{Q}$, the first factor on the right-hand side, is orthogonal. This factoring can be verified simply by multiplying it out. Now suppose also that we want to solve $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{b} = [1, -1, 1]^T$. Then proceeding as described above, we form

$$\mathbf{y} = \mathbf{Q}^T\mathbf{b} = \begin{bmatrix} 2/\sqrt{6} \\ -5/\sqrt{21} \\ 4/\sqrt{14} \end{bmatrix} \tag{4.14}$$

and then solve the triangular system $\mathbf{Rx} = \mathbf{y}$ to get $\mathbf{x} = [2/3, -1, 2/3]^T$. When we compute the residual $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$, we get the zero vector, showing that our arithmetic was done correctly. The numerical procedure with floating-point arithmetic proceeds in the same way. ◁

Now, how do we find the factors $\mathbf{Q}$ and $\mathbf{R}$, given a matrix $\mathbf{A}$? There are three widely used numerically stable algorithms:

1. Gram–Schmidt orthogonalization (modified for stability);
2. Householder reflections;
3. Givens rotations.

The third method is appropriate for sparse or structured matrices and is easily parallelized. Reluctantly, we leave the topic aside and focus on the first two. We first begin with classical Gram–Schmidt, followed by modified Gram–Schmidt, and ending with the method based on Householder reflections. We will investigate their respective numerical properties at the end of the chapter.

4.5.1 Classical (Unstable) Gram–Schmidt

Let us begin with a warning: Do not use this method for numerical computations! We will see why. But for pedagogical reasons, the algorithm is important, so a discussion is included here.

Write the $m \times n$ matrix $\mathbf{A}$ as a vector of columns,

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 \ \mathbf{a}_2 \ \cdots \ \mathbf{a}_n \end{bmatrix},$$

where each column $\mathbf{a}_k$ is $m \times 1$. Also, write the unknown matrix $\mathbf{Q}$ as a similar vector of columns:

$$\mathbf{Q} = \begin{bmatrix}\mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_n\end{bmatrix}$$

Then $\mathbf{A} = \mathbf{QR}$ is

$$\begin{bmatrix}\mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_n\end{bmatrix} = \begin{bmatrix}\mathbf{q}_1 & \mathbf{q}_2 & \dots & \mathbf{q}_n\end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & \cdots & r_{2n} \\ & & \ddots & \vdots \\ & & & r_{nn} \end{bmatrix}$$

and we see on multiplying out that

$$\begin{aligned} \mathbf{a}_1 &= r_{11}\mathbf{q}_1 \\ \mathbf{a}_2 &= r_{12}\mathbf{q}_1 + r_{22}\mathbf{q}_2 \\ &\vdots \\ \mathbf{a}_n &= r_{1n}\mathbf{q}_1 + r_{2n}\mathbf{q}_2 + \cdots + r_{nn}\mathbf{q}_n \,. \end{aligned} \tag{4.15}$$

Since $\mathbf{Q}$ is unitary, its columns have unit 2-norm, that is, $\|\mathbf{q}_k\|_2 = 1$, and they are orthogonal, namely,

$$\mathbf{q}_k^H \mathbf{q}_j = \delta_k^j ,$$

where here δ_i^j is the Kronecker delta (not to be confused with a relative error term). Thus, we see immediately that $r_{11} = \|\mathbf{a}_1\|$ and $\mathbf{q}_1 = r_{11}^{-1}\mathbf{a}_1$.

The next equation is processed by multiplying Eq. (4.15) by $\mathbf{q}_1^H$ on the left:

$$\mathbf{q}_1^H \mathbf{a}_2 = r_{12}\mathbf{q}_1^H \mathbf{q}_1 + r_{22}\mathbf{q}_1^H \mathbf{q}_2 = 1 \cdot r_{12} + 0 \cdot r_{22} = r_{12} \,.$$

Then, from Eq. (4.15), we obtain

$$\mathbf{a}_2 - r_{12}\mathbf{q}_1 = r_{22}\mathbf{q}_2 \,.$$

Taking the 2-norm of each side and simplifying, we obtain

$$r_{22} = \|\mathbf{a}_2 - r_{12}\mathbf{q}_1\|_2$$

and, just by dividing, we obtain

$$\mathbf{q}_2 = \frac{\mathbf{a}_2 - r_{12}\mathbf{q}_1}{r_{22}} \,.$$

Therefore, we have found the second vector component of $\mathbf{Q}$.

We see the recursive pattern emerge. This procedure allows us to identify r_{13}, r_{23}, r_{33} and then $\mathbf{q}_3$ from the next equation:

$$\mathbf{a}_3 = r_{13}\mathbf{q}_1 + r_{23}\mathbf{q}_2 + r_{33}\mathbf{q}_3 . \tag{4.16}$$

By manipulating the terms as above, we obtain

$$\mathbf{q}_1^H \mathbf{a}_3 = r_{13}$$
$$\mathbf{q}_2^H \mathbf{a}_3 = r_{23}$$

and, from Eq. (4.16),

$$r_{33} = \|\mathbf{a}_3 - r_{13}\mathbf{q}_1 - r_{23}\mathbf{q}_2\|_2 ,$$

together with

$$\mathbf{q}_3 = \frac{\mathbf{a}_3 - r_{13}\mathbf{q}_1 - r_{23}\mathbf{q}_2}{r_{33}} .$$

These are the first steps of "classical" Gram–Schmidt orthogonalization, which is fully described in Algorithm 4.4. As we see, it fails if any $r_{kk} = 0$, which happens if and only if some columns of $\mathbf{A}$ are linearly dependent; our assumption of full column rank n for $\mathbf{A}$ prevents this.

Algorithm 4.4 Classical (unstable) Gram–Schmidt orthogonalization

```
Require: A basis $[\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n]$ for a subspace $S$.
  for $j$ from 1 to $n$ do
    $\mathbf{q}_j := \mathbf{a}_j$
    for $i$ from 1 to $j-1$ do
      $r_{ij} := \mathbf{q}_i^H \mathbf{a}_j$
      $\mathbf{q}_j := \mathbf{q}_j - r_{ij}\mathbf{q}_i$
    end for
    $r_{jj} := \|\mathbf{q}_j\|_2$
    $\mathbf{q}_j := r_{jj}^{-1}\mathbf{q}_j$, so we require $r_{jj} \neq 0$
  end for
  return A unitary basis $\mathbf{Q} = [\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n]$ for the same subspace, and a matrix $\mathbf{R}$ such that
  $\mathbf{A} = \mathbf{QR}$.
```

Mathematically, there is nothing wrong with this procedure, but numerically there is: In floating-point arithmetic, rounding errors can build up in such a way that the columns of $\mathbf{Q}$ are not truly orthogonal. So we will investigate an alternative, apparently trivially different algorithm, after examining an example.

Example 4.5. Consider the following matrix, chosen by taking integers at random in the interval $[-99, 99]$.

$$\mathbf{A} = \begin{bmatrix} -81 & -98 & -76 & -4 & 29 \\ -38 & -77 & -72 & 27 & 44 \\ -18 & 57 & -2 & 8 & 92 \\ 87 & 27 & -32 & 69 & -31 \\ 33 & -93 & -74 & 99 & 67 \end{bmatrix} . \tag{4.17}$$

When we use classical Gram–Schmidt on this matrix (using a routine we wrote that implements Algorithm 4.4), we get the matrix

$$\mathbf{Q} = \begin{bmatrix} -0.6215 & -0.3574 & -0.2833 & -0.3015 & -0.5611 \\ -0.2916 & -0.3711 & -0.3417 & -0.1309 & 0.8021 \\ -0.1381 & 0.4370 & -0.7039 & 0.5393 & -0.0598 \\ 0.6675 & -0.1291 & -0.5544 & -0.4624 & -0.1287 \\ 0.2532 & -0.7258 & -0.0110 & 0.6224 & -0.1469 \end{bmatrix}.$$

We can assess the orthogonality of $\mathbf{Q}$ by also computing $\mathbf{Q}^T\mathbf{Q} - \mathbf{I}$ in MATLAB as follows:

```
q'*q-eye(5)
```

In this case, we obtained the result

$$10^{-14} \begin{bmatrix} 0 & -0.0139 & 0.0067 & -0.0028 & 0.0326 \\ -0.0139 & 0 & 0.0023 & -0.0500 & 0.4940 \\ 0.0067 & 0.0023 & 0.0222 & -0.0112 & 0.3028 \\ -0.0028 & -0.0500 & -0.0112 & -0.0222 & 0.0167 \\ 0.0326 & 0.4940 & 0.3028 & 0.0167 & 0.0222 \end{bmatrix}$$

and as we see, the matrix $\mathbf{Q}$ is fairly close to being orthogonal—not completely satisfactory, but not too bad. When we try with a 10 by 10 matrix (not shown, for brevity), the loss of orthogonality is about 10^{-15}, again not too bad. ◁

When we look more systematically, still taking matrices at random so as not to give the impression that we are looking for hard cases, the picture that emerges is less comforting. In the following script, we first set up a Fibonacci sequence, because that provides a set of dimensions that looks good on a log scale and is fine enough to show a trend. We then take 30 random matrices of each dimension (as generated by `rand`), compute their QR factoring by classical Gram–Schmidt, and measure the departure from orthogonality of each result. This test is performed as follows:

```
n = [2,3,ones(1,10)];
for i=3:12,
     n(i) = n(i-1)+n(i-2);
end;
nrms = zeros(30,12);
for i=1:12,
     for j=1:30,
       a = rand(n(i));
       q = gs(a);
       nrms(j,i) = norm(q'*q-eye(n(i)),inf);
     end;
end;
loglog(n,nrms','k.',n,n.^3/n(12)^3*mean(nrms(:,12)),'k')
```

Figure 4.2 shows essentially $O(n^3)$ growth of the average departure from orthogonality; by $n = 377$, the resulting $\mathbf{Q}$ is orthogonal only in single precision, not double precision.

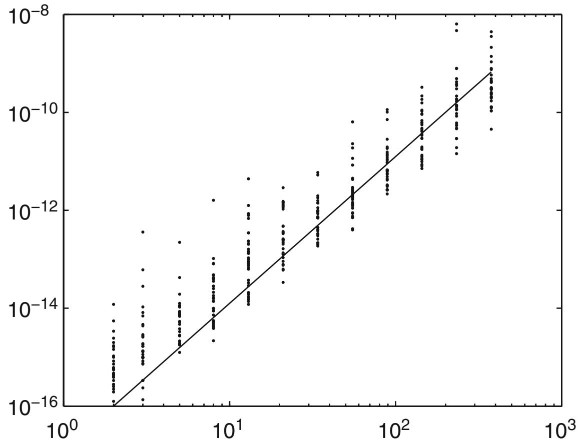


Fig. 4.2 For each dimension $n = 2, 3, 5, \ldots, F_{13}$, where F_k is the kth Fibonacci number, 30 random matrices computed by `rand` are given to the classical Gram–Schmidt procedure, and the departure from orthogonality $\|\mathbf{Q}^H\mathbf{Q}-\mathbf{I}\|_\infty$ of the result is recorded, and displayed on a log–log scale, with an $O(n^3)$ line for reference. A substantial loss of orthogonality is seen even for such relatively benign matrices

4.5.2 Modified Gram–Schmidt Orthogonalization

As before, let us write the equation $\mathbf{A} = \mathbf{QR}$ in the expanded form

$$\begin{bmatrix}\mathbf{a}_1 & \mathbf{a}_2 & \ldots & \mathbf{a}_n\end{bmatrix} = \begin{bmatrix}\mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_n\end{bmatrix}\left[\begin{array}{c|ccc} r_{11} & r_{12} & \cdots & r_{1n} \\ \hline & r_{22} & \cdots & r_{2n} \\ & & \ddots & \vdots \\ & & & r_{nn}\end{array}\right],$$

but now $\mathbf{R}$ is partitioned in the specific way indicated. We find r_{11} and $\mathbf{q}_1$ as before, but then immediately compute

$$\begin{aligned} r_{12} &= \mathbf{q}_1^H\mathbf{a}_2 \\ r_{13} &= \mathbf{q}_1^H\mathbf{a}_3 \\ &\vdots \\ r_{1n} &= \mathbf{q}_1^H\mathbf{a}_n. \end{aligned}$$

We then use this information to generate new vectors $\hat{\mathbf{a}}_2, \hat{\mathbf{a}}_3, \ldots, \hat{\mathbf{a}}_n$ as follows:

$$\hat{\mathbf{a}}_k = \mathbf{a}_k - r_{1k}\mathbf{q}_1, \quad 2 \leq k \leq n.$$

Then the partition gives

$$\begin{bmatrix} \hat{\mathbf{a}}_2 \; \hat{\mathbf{a}}_3 \ldots \hat{\mathbf{a}}_n \end{bmatrix} = \begin{bmatrix} \mathbf{q}_2 \; \mathbf{q}_3 \; \ldots \; \mathbf{q}_n \end{bmatrix} \begin{bmatrix} r_{22} & r_{23} & \cdots & r_{2n} \\ & r_{33} & \cdots & r_{3n} \\ & & \ddots & \vdots \\ & & & r_{nn}, \end{bmatrix} \tag{4.18}$$

which is a problem of the same type but one dimension smaller. Repeat until the last 1×1 system gives r_{nn}.

Obviously, this solves the same problem. Indeed, it is not immediately clear that the algorithm differs at all from the previous one. It turns out that the resulting algorithm (see Algorithm 4.5) is nearly identical, but not quite, to classical Gram–Schmidt (see Algorithm 4.4). Modified Gram–Schmidt turns out to be better at producing a numerically unitary $\mathbf{Q}$. However, we will see that it is not perfect either. We leave it to the exercises to duplicate the "average" random matrix example we used previously and to show that the growth is $O(n^2)$ and not $O(n^3)$. By $n = 377$, modified Gram–Schmidt seems to produce answers that are usually around n times better than those of Gram–Schmidt. This is not the worst-case scenario, however.

Algorithm 4.5 Modified Gram–Schmidt orthogonalization

```
Require: A full-rank matrix $\mathbf{A} \in \mathbf{C}^{n \times n}$
  for $i$ from 1 to $n$ do
    $\mathbf{q}_i := \mathbf{a}_i$ (it leaves $\mathbf{a}_i$ unchanged)
  end for
  for $i$ from 1 to $n$ do
    $r_{ii} = \|\mathbf{q}_i\|_2$
    $\mathbf{q}_i := r_{ii}^{-1}\mathbf{q}_i$, so we require $r_{ii} \neq 0$
    for $j$ from $i+1$ to $n$ do
      $r_{ij} := \mathbf{q}_i^H \mathbf{q}_j$
      $\mathbf{q}_j := \mathbf{q}_j - r_{ij}\mathbf{q}_i$
    end for
  end for
  return A unitary matrix $\mathbf{Q}$ and an upper-triangular matrix $\mathbf{R}$ such that $\mathbf{A} = \mathbf{QR}$.
```

Example 4.6. We now consider a more difficult example, the 5×5 Hilbert matrix. In general, the entry h_{ij} of the $n \times n$ Hilbert matrix is

$$h_{ij} = \frac{1}{i+j-1}. \tag{4.19}$$

The matrix is famously awkward numerically; we will see more of the Hilbert matrices. There are certainly others as awkward! For now, if we compute the QR factoring of this matrix in MATLAB, using the commands

```
a = hilb(5);
[q1,r1] = gs(a); %using algorithm classical Gram--Schmidt
[q2,r2] = mgs(a); %using algorithm modified Gram-Schmidt
[q3,r3] = qr(a); %built-in qr routine
```

we obtain the results displayed in Table 4.1. This represents a dramatic difference.

Table 4.1 Comparing the loss of orthogonality for CGS, MGS, and built-in QR

Method	$\|\mathbf{Q}^H\mathbf{Q}-\mathbf{I}\|_2$
Gram–Schmidt	$3.7\cdot 10^{-8}$
Modified Gram–Schmidt	$4.5\cdot 10^{-12}$
Built-in `qr`	$6.4\cdot 10^{-16}$

Even on this 5×5 matrix, Gram–Schmidt in its classical form behaves very badly, losing 8 figures of accuracy in $\mathbf{Q}$; mind you, $\|\mathbf{QR}-\mathbf{A}\|$ is zero! ◁

The factoring is very good—it's just that $\mathbf{Q}$ is not orthogonal, and so our solving process, which involved multiplying by $\mathbf{Q}^H$, is therefore compromised. MGS suffers much less from this loss of orthogonality, losing only 4 figures of accuracy. But the real winner in the orthogonality sweepstakes is the built-in `qr`. See Fig. 4.3. How does it achieve such good orthogonality? This is the object of the next subsection.

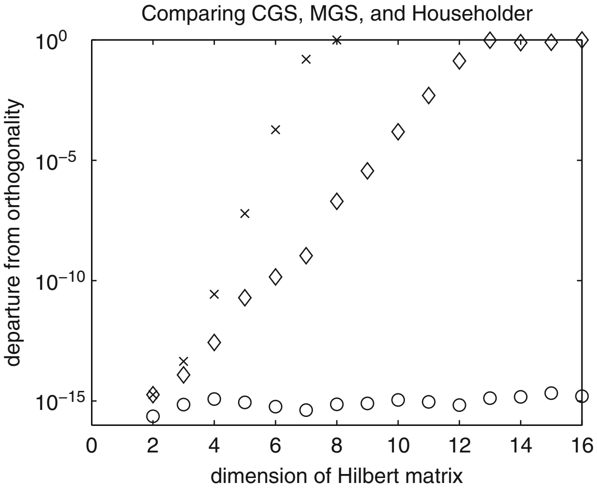


Fig. 4.3 Comparison of CGS (*cross*), MGS (*open diamond*), and Householder (*open circle*). We plot $\|\mathbf{Q}^H\mathbf{Q}-\mathbf{I}\|$ for each method, where $\mathbf{H}_n = \mathbf{QR}$ and $\mathbf{H}_n$ is the $n\times n$ Hilbert matrix. Observe that the slope of CGS is twice the slope of MGS. However, Smoktunowicz et al. (2006) shows that unless computation is done carefully, there are no bounds for CGS of this kind in general. There are simple examples where the loss of orthogonality is complete for CGS

4.5.3 Householder Reflections

This section describes the method of choice for computing a QR factoring. We begin with some preliminary formulæ. Given a nonzero vector $\mathbf{v}$, form the matrix

$$\mathbf{H} = \mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^H}{\mathbf{v}^H\mathbf{v}}, \tag{4.20}$$

which is known as a Householder reflector. Note that $\mathbf{v}\mathbf{v}^H$ is a rank-1 matrix and that $\mathbf{v}^H\mathbf{v}$ is a scalar. Moreover, let $\mathbf{a} \in \mathbb{C}^n$ be a nonzero column vector, and define

$$\mathbf{v} = \text{signum}(a_1)\|\mathbf{a}\|_2\mathbf{e}_1 + \mathbf{a}, \tag{4.21}$$

where

$$\text{signum}(z) = \begin{cases} z/|z| & \text{if } z \neq 0 \\ 1 & \text{if } z = 0 \end{cases}$$

and $\mathbf{e}_1 = [1, 0, 0, \ldots, 0]^T$.

We may proceed recursively to build the factor $\mathbf{Q}$ as a product of unitary matrices (see Problem 4.16) built from such elementary "Householder reflectors." The basic idea is simple. Take $\mathbf{a}$ to be $\mathbf{a}_1$, the first column of $\mathbf{A}$. Form $\mathbf{v}$ as in Eq. (4.21) and then the resulting $\mathbf{H}$; call it $\mathbf{H}_1$. Then (with the help of Problem 4.17) we obtain

$$\mathbf{H}_1\mathbf{A} = \begin{bmatrix} \alpha & \hat{a}_{12} & \cdots & \hat{a}_{1n} \\ 0 & \hat{a}_{22} & \cdots & \hat{a}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \hat{a}_{n2} & \cdots & \hat{a}_{nn} \end{bmatrix}$$

(note that

$$\mathbf{H}_1\mathbf{z} = \left(\mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^H}{\mathbf{v}^H\mathbf{v}}\right)\mathbf{z} = \mathbf{z} - 2\frac{\mathbf{v}(\mathbf{v}^H\mathbf{z})}{\mathbf{v}^H\mathbf{v}}$$

is easy to compute for each column $\mathbf{z} = \mathbf{a}_1, \mathbf{z} = \mathbf{a}_2, \ldots, \mathbf{z} = \mathbf{a}_n$). Now partition the result as

$$\left[\begin{array}{c|ccc} \alpha & \hat{a}_{12} & \cdots & \hat{a}_{1n} \\ \hline 0 & \hat{a}_{22} & \cdots & \hat{a}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \hat{a}_{n2} & \cdots & \hat{a}_{nn} \end{array}\right],$$

and suppose that we have computed the $\hat{\mathbf{Q}}\hat{\mathbf{R}}$ factoring of this smaller matrix (by Householder reflections also). Then let

$$\mathbf{H}_1\mathbf{A} = \left[\begin{array}{c|ccc} \alpha & \hat{a}_{12} & \cdots & \hat{a}_{1n} \\ \hline \mathbf{0} & & \hat{\mathbf{Q}}\hat{\mathbf{R}} & \end{array}\right].$$

So, by rearranging the terms, we obtain

$$\mathbf{A} = \mathbf{H}_1^H \begin{bmatrix} 1 & \\ & \hat{\mathbf{Q}} \end{bmatrix} \left[\begin{array}{c|ccc} \alpha & \hat{a}_{12} & \cdots & \hat{a}_{1n} \\ \hline \mathbf{0} & & \hat{\mathbf{R}} & \end{array}\right] \tag{4.22}$$

and, as result, our factor $\mathbf{Q}$ is given by

$$\mathbf{Q} = \mathbf{H}_1^H \begin{bmatrix} 1 & \\ & \hat{\mathbf{Q}} \end{bmatrix}, \tag{4.23}$$

which is unitary. Moreover, the right-hand matrix in Eq. (4.22) is upper-triangular (though not necessarily with nonnegative diagonal entries). We are done, since this is what we were after. The method is fully described in Algorithm 4.6.

Algorithm 4.6 Householder QR factoring

```
Require: $\mathbf{A} \in \mathbb{C}^{m\times n}$, $m \geq n$, full column rank.
  $\mathbf{Q} := \mathbf{I}_m$
  $\mathbf{R} := \mathbf{A}$
  for $k$ from 1 to $n$ do
    $\mathbf{a} := \mathbf{R}(k:m,k)$ (the current col)
    $\mathbf{v}_k := \text{signum}(a_1)\|\mathbf{a}\|_2\mathbf{e}_1 + \mathbf{a}$ (which is in $\mathbb{C}^{m-k+1}$)
    $\mathbf{v}_k := \mathbf{v}_k/\|\mathbf{v}_k\|$
    $\mathbf{R}(k:m,k:n) := (\mathbf{I}_{m-k+1} - 2\mathbf{v}_k\mathbf{v}_k^H)\mathbf{R}(k:m,k:n)$
    $\mathbf{Q}(1:k-1,k:m) := \mathbf{Q}(1:k-1,k:m)(\mathbf{I}_{m-k+1} - 2\mathbf{v}_k\mathbf{v}_k^H)$
    $\mathbf{Q}(k:m,k:m) := \mathbf{Q}(k:m,k:m)(\mathbf{I}_{m-k+1} - 2\mathbf{v}_k\mathbf{v}_k^H)$
  end for
  return A unitary matrix $\mathbf{Q}$ and an upper-triangular matrix $\mathbf{R}$ such that $\mathbf{A} = \mathbf{QR}$.
```

Time spent doing a QR factoring can't be spent doing something else. The time we take to do the factoring, therefore, can be regarded as a cost. Using this measure, the cost of the classical Gram–Schmidt method is exactly the same as that of the modified Gram–Schmidt method: $2mn^2$ flops. The Householder reflection method is slightly cheaper at $2n^2(m - {}^{n}\!/_{3})$ flops, provided that one accumulates the $\mathbf{H}_k$s but does not form $\mathbf{Q}$. It is more expensive if you multiply the $\mathbf{H}_k$s together to form $\mathbf{Q}$, but not grossly more. In practice, the Householder method's greater stability makes it very attractive.

4.5.4 Numerical Stability of the QR Algorithms

In this section, we examine the numerical stability of the three algorithms we examined to compute the QR factoring. Both CGS and MGS are stable in a normwise backward sense. Specifically, if $\hat{\mathbf{Q}}, \hat{\mathbf{R}}$ are the computed factors, then there exists an $\mathbf{E}$ such that

$$\mathbf{A} + \mathbf{E} = \hat{\mathbf{Q}}\hat{\mathbf{R}} \tag{4.24}$$

and $\|\mathbf{E}\|_F \leq c_n \mu_M \|\mathbf{A}\|_F$, where c_n is a slowly growing function of n and $\|\cdot\|_F$ is the Frobenius norm. That is, both CGS and MGS give the exact factors of a slightly perturbed matrix, differing normwise by only a small amount from the original.

However, both CGS and MGS lose orthogonality in the computed factor $\hat{\mathbf{Q}}$. For MGS,

$$\|\hat{\mathbf{Q}}^H \hat{\mathbf{Q}} - \mathbf{I}\| \leq c_n \kappa(\mathbf{A}) \varepsilon_M\,,$$

where c_n is a slowly growing function of n. The function $\kappa(\mathbf{A})$ is described in detail in Sect. 4.6. For now, note that it can be large, as it is for the Hilbert matrices, where it grows exponentially with the dimension. For CGS, we don't even have this much: No such bound exists, unless special care is taken (Smoktunowicz et al. 2006).

The situation is better with Householder's method. If $\hat{\mathbf{Q}}$ and $\hat{\mathbf{R}}$ are the computed factors, then there exists a unitary matrix (an exactly unitary matrix) $\mathbf{Q}$ near $\hat{\mathbf{Q}}$ with

$$\mathbf{A} + \mathbf{E} = \mathbf{Q}\hat{\mathbf{R}} \tag{4.25}$$

and each column $\mathbf{e}_j$ of $\mathbf{E}$ satisfies $\|\mathbf{e}_j\|_2 \leq \gamma_{mn} \|\mathbf{a}_j\|_2$, where $\mathbf{a}_j$ is the corresponding column of the $m \times n$ matrix $\mathbf{A}$. Moreover, $\hat{\mathbf{Q}}$ is close to $\mathbf{Q}$: We have

$$\|\hat{\mathbf{Q}} - \mathbf{Q}\|_F \leq \sqrt{n}\gamma_{cmn}\,,$$

where c is a small integer constant. This is Theorem 19.4 in Higham (2002). Notice that the potentially large function $\kappa(\mathbf{A})$ does not appear in this bound: The Householder factoring returns matrices that are guaranteed to be close to orthogonal.

Now, let us turn to the residual $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$, which turns out to be as important. Using the above statements, we can say that, when solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ using the Householder QR factoring, the residual is guaranteed to be small in the following sense:

$$\|\mathbf{r}\|_2 \leq \gamma_{cn^2} \|\mathbf{A}\|_F \|\mathbf{x}\|_2\,, \tag{4.26}$$

where c is a small constant.

We could have introduced the residual already for triangular systems, but a stronger and more satisfactory backward error guarantee—namely, exact solution with a relatively tinily perturbed $\mathbf{T}$—was available. However, the vector $\mathbf{r}$ will be used henceforth as appropriate.

Remark 4.2. The residual, as defined above, is important in assessing the credibility of a numerical solution of square systems. If the computed residual $\mathbf{r}$ has a "small enough" norm, compared to errors in the right-side vector $\mathbf{b}$, then the method (*whatever* it was) has produced an acceptable answer. As we see above, there is a nice guarantee for the QR method that the residual will always be small in a normwise sense. ◁

The residual gives an extraordinarily simple method for a posteriori backward error assessment: Simply compute the residual and examine it. This works for solving square linear systems ($m = n$) and for overspecified systems ($m > n$). It is so simple, it seems like cheating. To realize how simple its use is, we make the following key (but trivial) observation: If $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$, then $\mathbf{x}$ is the *exact* solution of

$$\mathbf{A}\mathbf{x} = \mathbf{b} - \mathbf{r}\,; \tag{4.27}$$

that is, we have computed the *exact* solution of a slightly different system of linear equations, one with a perturbed right-hand side. If the maximum entry in $\mathbf{r}$ is (say) 10^{-13}, and the errors in your data vector $\mathbf{b}$ are about 10^{-7}, then you are done: For all you know, you have found the exact solution you are looking for.

A second major point (which we pursue in Sect. 4.6) is that one needs to know how sensitive a solution is to such changes in the data. Importantly, given our backward error perspective, we have put numerical errors on exactly the same footing as errors in the data. But one *has* to understand the effects of perturbation on the data anyway; so this is already (albeit trivially) a successful analysis.

There is a more subtle point that we have to examine: How do we *compute* $\mathbf{r}$? A traditional approach is to use higher precision (such as double precision if you were working in single precision, or quadruple if you were working in double). The difficulty is that $\mathbf{A}\mathbf{x}$ is very nearly equal to $\mathbf{b}$, and when you subtract, you reveal rounding errors and don't leave (much) forward accuracy in $\mathbf{r}$. Sometimes, however, it is possible to use backward error *again*: Remember our discussion of matrix multiplication and backward error. Notice that the computation of $\mathbf{A}\mathbf{x}$ (let this be $\mathbf{y}$) gives you the *exact* result of $(\mathbf{A} + \Delta\mathbf{A})\,\mathbf{x}$, where each entry of $\Delta\mathbf{A}$ is only a tiny relative perturbation of the corresponding entry of $\mathbf{A}$: That is, we can regard the result of the matrix–vector product as being exact if we allow some tiny uncorrelated perturbations in $\mathbf{A}$. Now when we subtract the result from $\mathbf{b}$, the subtraction is *benign*: Each entry is just $(b_i - y_i)(1 + \delta_i)$, which gives the correctly rounded result (in IEEE floating-point arithmetic, which has guard digits). That is, each entry of the residual has been computed to full accuracy if the matrix $\mathbf{A}$ is considered to be perturbed to some very nearby matrix. Thus, we have an argument that the computed residual and the computed solution together satisfy the following theorem:

Theorem 4.3. *The computed residual* $\hat{\mathbf{r}}$ *is the exactly rounded representation of* $\mathbf{r}$ *satisfying*

$$(\mathbf{A} + \Delta\mathbf{A})\,\mathbf{x} = \mathbf{b} - \mathbf{r}\,, \tag{4.28}$$

where each entry in $\Delta\mathbf{A}$ *is* $O(\mu_M)$ *times the corresponding entry in* $\mathbf{A}$.

In practice, the perturbations $\Delta\mathbf{A}$ are usually much, much smaller than the entries that appear in $\mathbf{r}$, and are safely ignored. But when the arguments get delicate, one can fall back to this position and use Eq. (4.28).

Example 4.7. To make this method of analysis more concrete, let us continue Example 4.1. It turns out that the symmetric matrix $\mathbf{B}$ factors into the product of two slightly nicer matrices: $\mathbf{B} = \mathbf{A}^H\mathbf{A}$, where $\mathbf{A}$ is as below, as was noticed in Corless (1993). Here, we want to solve

$$\mathbf{Ax} = \begin{bmatrix} 666 & 665 \\ 667 & 666 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \mathbf{b} \tag{4.29}$$

using the QR factoring, and assess the result on the basis on the computed residual. The solution is found in MATLAB by executing

```
[Q,R] = qr(A)
y = Q'*[1;0]
x = R\y
```

It returns the vector

$$\hat{\mathbf{x}} = 10^2 \begin{bmatrix} 6.660000000445352 \\ -6.670000000446023 \end{bmatrix},$$

which is very close to the exact solution $[666, -667]^T$. If we compute the residual with the command `r = b - A*x`, we obtain

$$\hat{\mathbf{r}} = 10^{-9} \begin{bmatrix} 0.0582 \\ 0.1164 \end{bmatrix}.$$

Thus, the computed solution (which is near to, but not exactly, the exact solution) is the exact solution to $\mathbf{Ax} = \mathbf{b} - \mathbf{r}$, where $\mathbf{r}$ has an entry about $1.1 \cdot 10^{-10}$. Moreover, by Theorem 4.3, all figures in each entry of $\mathbf{r}$ are correct if we allow tiny (of the order of μ_M) changes in the entries of $\mathbf{A}$.

However, the truly surprising results in Example 4.1 did not arise for $\mathbf{Ax} = \mathbf{b}$, but rather for $\mathbf{Bx} = \mathbf{A}^H\mathbf{Ax} = \mathbf{b}$. Using the same solution procedure based on the QR factoring (using MATLAB's `qr` command), we obtain

$$\hat{\mathbf{x}} = 10^5 \begin{bmatrix} 8.8575 \\ -8.8709 \end{bmatrix} \qquad \text{and} \qquad \hat{\mathbf{r}} = 10^{-3} \begin{bmatrix} 0.1221 \\ 0.1221 \end{bmatrix}.$$

That seems large, until we scale by the norm of $\mathbf{B}$ and the size of the solution $\mathbf{x}$: We can expect rounding errors of size $\|\mathbf{B}\|\,\|\mathbf{x}\|$ times $O(\mu_M)$, and indeed $\|\mathbf{r}\|/\|\mathbf{B}\|\,\|\mathbf{x}\| \doteq 7 \times 10^{-17}$.

Even *more* puzzling though is the computed solution of $\mathbf{Bx} = [0, 1]^T$ and its residual. Proceeding as above, we find that $\mathbf{r}$ is identically zero, not $O(10^{-3})$! This is puzzling, since our solution (not shown here) differs from the exact solution (in the thousands' place, already), but we have a zero residual. In fact, the residual is smaller than when we solved the system with $\mathbf{A}$, which is surprising because we would expect a larger error when the entries of the matrix are larger. However, Theorem 4.3 gives us grounds for an explanation. The fact that the computed residual is $\mathbf{0}$ tells us that, for some matrix $\delta\mathbf{B}$ with $|\delta b_{ij}| = O(\mu_M |b_{ij}|)$, we have exactly solved $(\mathbf{B} + \delta\mathbf{B})\mathbf{x} = \mathbf{b} - \mathbf{r}$, where $\hat{\mathbf{r}} - \mathbf{r}$ is truly zero (because it is the exactly rounded result). ◁

Remark 4.3. It is now clear that a zero computed residual is *not* a guarantee that we have found the reference solution. It is *only* a guarantee that we have found the exact solution to a nearby problem. ◁

This will prove to be true if the computed residual is merely "small," not exactly zero, as well. The forward error (which we have not shown here) is not explained by this approach. This will become possible in Sect. 4.6 when we explore the condition number of matrices.

4.5.5 Solving Overspecified Systems with the QR Factoring

Suppose we are given $\mathbf{A} \in \mathbb{C}^{m\times n}$ and $\mathbf{b} \in \mathbb{C}^{m}$ and we want to find $\mathbf{x} \in \mathbb{C}^{n}$ such that

$$\mathbf{Ax} = \mathbf{b}\,. \tag{4.30}$$

If $m = n$, then we have a system for which we have the same number of equations and unknowns. When $m > n$ and when the equations are linearly independent, then we call the system *overspecified*[11]: There are more equations than unknowns. In overspecified cases, there is often no $\mathbf{x}$ that satisfies Eq. (4.30) exactly. This situation occurs very often in applications; perhaps the most common is found in linear least-square problems. In such cases, our solution will be a vector $\mathbf{x}$ in the equation $\mathbf{Ax} \approx \mathbf{b}$ for which the 2-norm of the residual is to be minimized. Here, we are looking for an $\mathbf{x}$ such that

$$\mathbf{x} = \arg\min_{\mathbf{x}} \|\mathbf{r}\|_2 = \arg\min_{\mathbf{x}} \|\mathbf{b} - \mathbf{Ax}\|_2 = \arg\min_{\mathbf{x}} \|\mathbf{b} - \mathbf{Ax}\|_2^2\,, \tag{4.31}$$

where arg min is the value of $\mathbf{x}$ at which the minimum occurs. We then say that $\mathbf{x}$ *is the solution of* $\mathbf{Ax} = \mathbf{b}$ *in the least-square sense.* Note that when presented an overspecified system and asked to find `x=A\b`, MATLAB will automatically return the solution in the least-square sense.

We will now examine how to find $\mathbf{x}$ for cases in which $\mathbf{A}$ is upper-triangular and for cases where it is not. If $\mathbf{A}$ is upper-triangular and has full column rank, such as the following example in which $\mathbf{Ax} = \mathbf{b}$ is

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 8 \\ 9 \\ b_4 \\ b_5 \\ b_6 \end{bmatrix},$$

[11] We are indebted to David Jeffrey for this term. The more commonly used term, "overdetermined," isn't accurate if conflicting information is given by the equations, and once you realize this, it's bothersome: If $m > n$, there is generally too much conflicting information to determine a solution at all.

for any values of b_4, b_5 and b_6, it is easy to determine $\mathbf{x}$. The exact solution of the upper part of the system gives $x_3 = 3/2$, then $x_2 = 1/8$ and $x_1 = 57/12$. Then, we will have the residual

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ b_4 \\ b_5 \\ b_6 \end{bmatrix}.$$

If b_4, b_5, and b_6 are 0, then we have found (by back substitution) the vector $\mathbf{x}$ that satisfies $\mathbf{A}\mathbf{x} = \mathbf{b}$. If b_4, b_5 and b_6 are not 0, we easily find by inspection that

$$\begin{aligned} ||\mathbf{r}||_2^2 &= |r_1|^2 + |r_2|^2 + |r_3|^2 + |r_4|^2 + |r_5|^2 + |r_6|^2, \\ r_i &= b_i - \mathbf{A}(i,:)x_i. \end{aligned}$$

$\mathbf{r}$ is as small as it gets, for any choice of x_1, x_2 and x_3, since $r_i = 0$ for $1 \le i \le 3$ and $r_i = b_i$ for $4 \le i \le 6$. So

$$||\mathbf{r}||_2^2 \ge |b_4|^2 + |b_5|^2 + |b_6|^2$$

(absolute values are needed because b_4, b_5 and b_6 might be complex). So, for an upper-triangular $\mathbf{A}$, the process of back substitution for the upper part of the system gives us the least-squares solution.

However, not all matrices $\mathbf{A}$ are upper-triangular. That is where the QR factoring comes into play. The usefulness of the QR factoring for least-square problems stems from the following straightforward theorem:

Theorem 4.4. *Product by a unitary matrix preserves the 2-norm; that is,*

$$||\mathbf{x}||_2 = ||\mathbf{Q}\mathbf{x}||_2 .$$

Proof. Observe that

$$||\mathbf{x}||_2^2 = \mathbf{x}^H \mathbf{x} = \mathbf{x}^H \mathbf{Q}^H \mathbf{Q} x = ||\mathbf{Q}\mathbf{x}||_2^2$$

follows from definitions. ♮

It follows from Theorem 4.4 that

$$\min_{\mathbf{x}} ||\mathbf{b} - \mathbf{A}\mathbf{x}||_2^2 = \min_{\mathbf{x}} ||\mathbf{Q}^H \mathbf{b} - \mathbf{Q}^H \mathbf{A}\mathbf{x}||_2^2 = \min_{\mathbf{x}} ||\mathbf{Q}^H \mathbf{b} - \mathbf{R}\mathbf{x}||_2^2 ; \tag{4.32}$$

that is, the residuals $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$ and $\mathbf{Q}^H \mathbf{r} = \mathbf{Q}^H \mathbf{b} - \mathbf{Q}^H \mathbf{Q}\mathbf{R}\mathbf{x} = \mathbf{Q}^H \mathbf{b} - \mathbf{R}\mathbf{x}$ have the same 2-norm. So, the vector $\mathbf{x}$ that minimizes $||\mathbf{r}||_2$ will also minimize $||\mathbf{Q}^H \mathbf{r}||_2$. Consequently, finding $\min_{\mathbf{x}} ||\mathbf{Q}^H \mathbf{r}||_2$ is equivalent to finding the least-squares solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$.

Since $\mathbf{R}$ is obtained from the QR factoring, we can let

$$\mathbf{R} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} \\ 0 & 0 & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{0} \end{bmatrix}.$$

Then we write

$$\mathbf{Rx} = \begin{bmatrix} \mathbf{R}_1\mathbf{x} \\ \mathbf{0} \end{bmatrix}, \tag{4.33}$$

and the entries of $\mathbf{Q}^H\mathbf{b} - \mathbf{Rx}$ in rows $n+1, n+2, \ldots, m$ cannot be changed by any choice of $\mathbf{x}$. But, in a way similar to what we have done for upper-triangular matrices $\mathbf{A}$, we may choose $\mathbf{x}$ to make

$$\mathbf{R}_1\mathbf{x} = \mathbf{Q}(1:n,:)^H\mathbf{b}.$$

This straightforwardly makes the 2-norm of the residual

$$\mathbf{b} - \mathbf{Ax}$$

minimum.

Example 4.8. Suppose that we have the following data gathered from timing the execution of an algorithm operating on matrices:

Size n	5	55	105	155	205	255	305	355	405	455	505
Time t	0.0004	0.0012	0.0024	0.0046	0.0080	0.0120	0.0168	0.0232	0.0320	0.0682	0.0604

See Fig. 4.4. We want to find a third-degree polynomial expressing the execution time in terms of the size n of square matrices.[12] Because of other processes on any given machine, and perhaps because of minor differences in execution paths for a given matrix, the fit cannot be exact. This gives 11 equations of the form $\text{cost}(n_k) = c_3n_k^3 + c_2n_k^2 + c_1n_k + c_0$ in the four unknowns c_js, clearly overspecifying the answer. The corresponding matrix is then

[12] For theoretical reasons, namely, the flop count, we expect that the computing time will increase as the cube of the dimension for the QR factoring. Therefore, we may wish to summarize our cpu time data as a third-degree polynomial.

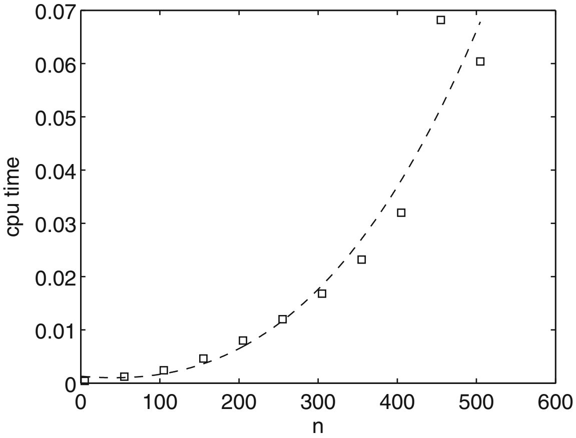


Fig. 4.4 Time taken to compute the QR factoring for some random $n \times n$ matrices, fitted by a least-squares polynomial. The computer used was a 32-bit tablet PC, circa 2009

$$\mathbf{A} = \begin{bmatrix} 125 & 25 & 5 & 1 \\ 166375 & 3025 & 55 & 1 \\ 1157625 & 11025 & 105 & 1 \\ 3723875 & 24025 & 155 & 1 \\ 8615125 & 42025 & 205 & 1 \\ 16581375 & 65025 & 255 & 1 \\ 28372625 & 93025 & 305 & 1 \\ 44738875 & 126025 & 355 & 1 \\ 66430125 & 164025 & 405 & 1 \\ 94196375 & 207025 & 455 & 1 \\ 128787625 & 255025 & 505 & 1 \end{bmatrix}$$

and we wish to solve $\mathbf{Ac} = \mathbf{b}$, where the vector $\mathbf{b}$ is the timing data and the 4-vector $\mathbf{c}$ contains the coefficients of our polynomial. Simply using MATLAB's backslash command uses a QR factoring internally, but we may do that ourselves with the command `[Q,R] = qr(A, 0)`.[13] We then form the 4-vector $\mathbf{y} = \mathbf{Q}'\mathbf{b}$ and solve the 4×4 triangular system $\mathbf{Rc} = \mathbf{y}$ by back substitution. When we do this, we find that

$$\mathrm{cost}(n) = 3.0427 \cdot 10^{-10} n^3 + 1.3273 \cdot 10^{-7} n^2 - 1.2808 \cdot 10^{-5} n + 1.2887 \cdot 10^{-3}$$

fits the cost, in seconds, to the dimension. The leading coefficient is quite small, but as we see in Fig. 4.4, the fit is good.[14] ◁

[13] We use the argument 0 in order to compute a "thin" matrix $\mathbf{Q}$, 11 by 4; if we leave that argument off, then MATLAB will give an orthogonal completion of those four vectors and return an 11-by-11 matrix (which we don't need).

[14] The two leading terms are equal already for $n = 300$, so the leading coefficient is significant for this fit.

4.6 SVD and Condition Number

It is now time to discuss the relationship between the condition number and the singular value decomposition. We begin with a statement of the factoring theorem that we will use repeatedly.

Theorem 4.5. *Every $m \times n$ matrix $\mathbf{A} \in \mathbb{C}^{m\times n}$ may be factored as*

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H , \tag{4.34}$$

where $\mathbf{U} \in \mathbb{C}^{m\times m}$ and $\mathbf{V} \in \mathbb{C}^{n\times n}$ are unitary matrices and $\boldsymbol{\Sigma}$ is an $m \times n$ nonnegative diagonal matrix such that $\boldsymbol{\Sigma} = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_p)$ with $p = \min(m,n)$. The σs are known as singular values, and the factoring is known as the SVD.

For a proof, see Stewart (1998 p. 156). Note that the diagonal entries σ_k are arranged such that

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > \sigma_{r+1} = 0 = \cdots = \sigma_p \,.$$

The index r of the last nonzero diagonal entry σ_r is the *rank*. Moreover, as shown in Fig. 4.5, the cost of computing the SVD is relatively high, but we will see here that it is worth the price.

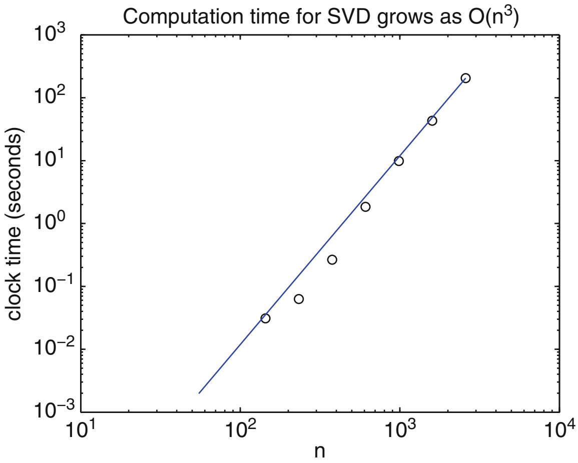


Fig. 4.5 Time taken to compute the SVD for some random $n \times n$ matrices. The computer used was a 64-bit desktop, vintage 2009, running MATLAB 2009a

There are many important ways to unpack the meaning of this theorem. If we know the factors $\mathbf{U}$, $\boldsymbol{\Sigma}$, and $\mathbf{V}$ in the SVD, we effectively know everything we want to know about the matrix $\mathbf{A}$. To begin with a simple example, it follows that we know the inverse, if $m = n$ and $\sigma_n \neq 0$:

$$\mathbf{A}^{-1} = \mathbf{V}\boldsymbol{\Sigma}^{-1}\mathbf{U}^{H}.$$

Of course, the inverse of the diagonal matrix $\boldsymbol{\Sigma}$ is trivial: Just reciprocate the diagonal entries. Thus, the SVD gives us a way to solve $\mathbf{Ax} = \mathbf{b}$. In practice, we would not usually form this inverse, of course, but we will see how to use this factoring to solve various matrix problems.

One of the more important ways to understand singular values is based on the following theorem.

Theorem 4.6. *Suppose* $\mathbf{A}$ *factors as above. If, for* $k \le r$, $\mathbf{U}_k = [\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_k]$, $\mathbf{V}_k = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k]$, $\boldsymbol{\Sigma}_k = \mathrm{diag}(\sigma_1,\ \sigma_2,\ \ldots,\ \sigma_k,\ 0,\ \ldots,\ 0)$ *with* $\boldsymbol{\Sigma}_k$ *being* $m \times n$ *with enough zeros to fill out the diagonal, and* $\sigma_1 \ge \sigma_2 \ge \ldots \ge \sigma_k > 0$, *then if we define* $\mathbf{A_k} = \mathbf{U}_k\boldsymbol{\Sigma}_k\mathbf{V}_k^H$, *this* $m \times n$ *matrix satisfies the following property:*

$$\|\mathbf{A}_k - \mathbf{A}\|_2 = \min_{\mathrm{rank}(\mathbf{B}) \le k} \|\mathbf{B} - \mathbf{A}\|_2 = \sigma_{k+1} \tag{4.35}$$

and $\mathrm{rank}(\mathbf{A}_k) = k$. *That is,* $\mathbf{A}_k$ *is as near (in the 2-norm) to* $\mathbf{A}$ *as any rank-k matrix can get (see Schmidt 1907).*

We do not supply a proof here. One can be found in Golub and van Loan (1996).

Remark 4.4. This theorem has several interesting consequences. First, the distance to the nearest singular matrix is exactly $\sigma_r > 0$. The *relative* distance is, therefore, σ_r/σ_1, because the 2-norm of $\mathbf{A}$ is exactly σ_1 (see Exercise 4.10). This supplies a natural metric for the notion of singularity.

In the first exposure to matrices, one learns that they are either singular or nonsingular. This theorem provides a way of putting shades of gray into that black-and-white distinction: Some matrices, while nonsingular technically, are so nearly singular that they might as well be actually singular. If the entries of $\mathbf{A}$ are not known to great precision, and some matrices near to $\mathbf{A}$ really are singular, then one ought to consider the case of singularity. ◁

In light of this remark, we should ask: Given that the entries of $\mathbf{A}$ are sometimes not known with great precision, how are its singular values affected by small variations in its entries? A great deal can be said about this question, but the simplest result is due to Weyl.

Theorem 4.7. *If* $\mathbf{A} + \mathbf{E}$ *has singular values* $\hat{\sigma}_k$ *and* $\mathbf{A}$ *has singular values* σ_k, *then* $|\hat{\sigma}_k - \sigma_k| \le \|\mathbf{E}\|$ *for* $1 \le k \le n$.

Note that these are not relative perturbations—small singular values can indeed be swamped by changes in the data. However, this theorem says that at least the larger singular values will be accurate if the data are known with any accuracy.[15]

It is useful to explore the meaning of the SVD further in geometrical terms (see Fig. 4.6). Let $\mathbb{T} = \{\mathbf{u} : \|\mathbf{u}\|_2 = 1\}$ be the unit circle and let

$$\mathbf{A}_{\mathbb{T}} = \{\mathbf{z} \mid \mathbf{z} = \mathbf{Au} \text{ where } \mathbf{u} \in \mathbb{T}\} \tag{4.36}$$

[15] See Stewart and Sun (1990).

be a (possibly degenerate) ellipsoid (as you will show in Problem 4.18), where the singular values σ_k $(1 \le k \le r)$ are the lengths of the semiaxes. The aspect ratio of the ellipsoid $\mathbf{A}\mathbb{T}$ is given by the quotient $^{\sigma_n}/_{\sigma_1}$ of the largest and smallest singular values; in other words, this ratio tells us how skinny the ellipsoid is. On the one hand, the best ratio is $^{\sigma_n}/_{\sigma_1} = 1$; this happens for "nice" matrices $\mathbf{A}$ whose singular values are all equal. On the other hand, if $\mathbf{A}$ is singular, the ellipsoid will degenerate to a lower-dimension surface.

Example 4.9. Let

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix}. \tag{4.37}$$

Then $\mathbf{A}\mathbb{T}$ is drawn together with the circle $\mathbb{T}$ in Fig. 4.6. The factors in $\mathbf{A} = \mathbf{U\Sigma V}^H$ are as follows:

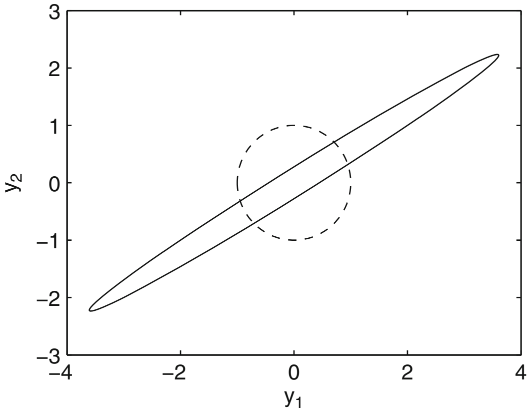


Fig. 4.6 Transformation of the unit circle by the matrix $\mathbf{A} = [2,3;1,2]$. The largest singular value (≈ 4.2361) is the length of the longest semiaxis, and the smallest singular value (≈ 0.2361) is the length of the shortest semiaxis. See also the MATLAB command `eigshow`, for an animated explanation that may be more helpful than this static explanation

$$\mathbf{U} \approx \begin{bmatrix} -0.8507 & -0.5257 \\ -0.5257 & 0.8507 \end{bmatrix}, \quad \mathbf{\Sigma} \approx \begin{bmatrix} 4.2361 & 0 \\ 0 & 0.2361 \end{bmatrix}, \quad \mathbf{V} \approx \begin{bmatrix} -0.5257 & -0.8507 \\ -0.8507 & 0.5257 \end{bmatrix}.$$

The vector $\mathbf{u}_1$ (the first column of $\mathbf{U}$) points in the direction of the longest semiaxis; $\mathbf{u}_2$ in the direction of the shortest semiaxis. ◁

4.6.1 The Effect of Data Error in b

If we know that $\mathbf{A} = \mathbf{U\Sigma V}^H$ is in $\mathbb{C}^{n\times n}$, $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_n]$, and $\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n]$, then we may examine a "worst-case scenario" for the effect of errors in the data. Suppose that we are trying to find $\mathbf{x}$ such that

$$\mathbf{A}\mathbf{x} = \mathbf{u}_1 \,.$$

Then it is easy to see that $\mathbf{x} = ({}^1\!/\!_{\sigma_1})\mathbf{v}_1$ does the trick:

$$\mathbf{A}\frac{\mathbf{v}_1}{\sigma_1} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H\frac{\mathbf{v}_1}{\sigma_1} = \mathbf{U}\boldsymbol{\Sigma}\begin{bmatrix} {}^1\!/\!_{\sigma_1} \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \mathbf{U}\begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \mathbf{u}_1 \,.$$

Since $\mathbf{A}$ is nonsingular by hypothesis, this must be the unique answer.

Now suppose that we are, in fact, horribly unlucky, and the right-side vector $\mathbf{b}$ is not actually $\mathbf{u}_1$ as we thought, but rather is polluted with errors in the direction of $\mathbf{u}_n$. So, let $\hat{\mathbf{b}} = \mathbf{u}_1 + \varepsilon\mathbf{u}_n$. If we solve the system again, the solution we get is not $\mathbf{x} = {}^1\!/\!_{\sigma_1}\mathbf{v}_1$ as it was before, but rather $\mathbf{x} + \Delta\mathbf{x} = {}^1\!/\!_{\sigma_1}\mathbf{v}_1 + {}^\varepsilon\!/\!_{\sigma_n}\mathbf{v}_n$. This can easily be verified:

$$\begin{aligned} \mathbf{A}\left(\frac{1}{\sigma_1}\mathbf{v}_1 + \frac{\varepsilon}{\sigma_n}\mathbf{v}_n\right) &= \frac{1}{\sigma_1}\mathbf{A}\mathbf{v}_1 + \frac{\varepsilon}{\sigma_n}\mathbf{A}\mathbf{v}_n \\ &= \mathbf{u}_1 + \frac{\varepsilon}{\sigma_n}\mathbf{U}\boldsymbol{\Sigma}\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} = \mathbf{u}_1 + \frac{\varepsilon}{\sigma_n}\mathbf{U}\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \sigma_n \end{bmatrix} = \mathbf{u}_1 + \varepsilon\mathbf{u}_n \,. \end{aligned}$$

How big a difference is caused by this variation of the right-side $\mathbf{b}$? The relative difference between these two solutions is

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} = \frac{\|{}^\varepsilon\!/\!_{\sigma_n}\mathbf{v}_n\|}{\|{}^1\!/\!_{\sigma_1}\mathbf{v_1}\|} = \frac{\sigma_1}{\sigma_n}\varepsilon \,,$$

whereas the difference between the two sets of data is only

$$\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} = \frac{\|\varepsilon\mathbf{u}_n\|}{\|\mathbf{u}_1\|} = \varepsilon \,.$$

In other words, the data error ε has been amplified to $({}^{\sigma_1}\!/\!_{\sigma_n})\varepsilon$ in the solution; note that ${}^{\sigma_1}\!/\!_{\sigma_n} \geq 1$ can be very large indeed if the matrix is nearly singular. A little thought shows that this is the worst possible amplification (see Problem 4.24).

Consequently, we find that

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \kappa_2(\mathbf{A})\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} \tag{4.38}$$

where, if $\mathbf{A}$ is nonsingular, we define

$$\kappa_2(\mathbf{A}) = \|\mathbf{A}\|_2\|\mathbf{A}^{-1}\|_2 = \frac{\sigma_1}{\sigma_n} \tag{4.39}$$

since $\sigma_1 = \|\mathbf{A}\|_2$ and $\sigma_n^{-1} = \|\mathbf{A}^{-1}\|_2$. We call $\kappa_2(\mathbf{A})$ the 2-norm *condition number* of $\mathbf{A}$.

Figure 4.7 displays the condition number of random matrices as a function of their sizes. This number, or related numbers using other norms, was the beginning of the study of conditioning or sensitivity in the solution of linear systems of equations. Using this number, we can bound (or estimate) the forward error in a solution given the backward error of a method.

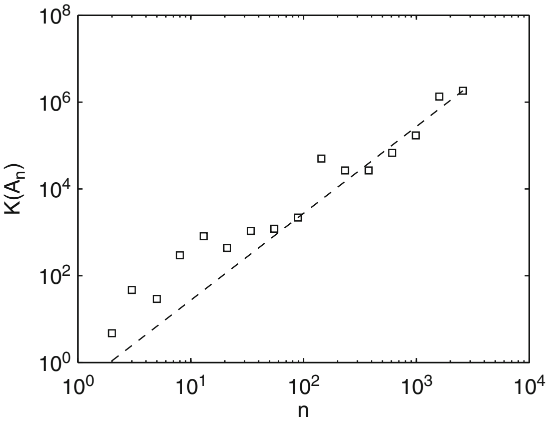


Fig. 4.7 Condition numbers of some random matrices, $A = \texttt{rand}(n)$. The *dashed line* shows a constant times n^2

Example 4.10. Consider again the upper-triangular matrices from Example 4.3. We generate them in MATLAB as follows:

```
b = ones(n,1);
for j=2:2:n,
      b(j) = -1/3;
end;
U = diag(ones(1,n)) - triu(2*ones(n),1);
x = U\b;
```

We saw that for $n = 32$ the accuracy of x_1 was about 0.0038. This was in spite of the excellent backward error: The computed x is the *exact* solution of an upper triangular system with entries different from those of $\mathbf{U}$ by less than simple rounding.

The difficulty, of course, is the conditioning. We compute the SVD in MATLAB for $n = 32$ and find that the first 31 singular values are large enough—we find approximately $\sigma_1 = 39.5134$ and $\sigma_{31} = 2.0006$, but $\sigma_{32} = 2.1 \cdot 10^{-15}$, which is less than $\mu_M \sigma_1$. Thus, this matrix is within roundoff error of a singular matrix. Therefore, nearby matrices (even matrices essentially within rounding distance from $\mathbf{U}$) will have greatly differing solutions to $(\mathbf{U} + \Delta\mathbf{U})\mathbf{x} = \mathbf{b}$.

As it turns out, the actual error (about 0.0038) is quite a bit less than the approximate bound given by the condition number (which is greater than $1/\varepsilon_M$) times the bound for the rounding error (approximately ε_M). That is, the real scenario is better than the worst-case scenario. ◁

Remark 4.5. We may compute the inverse of the matrix $\mathbf{U}$ explicitly, and its first row contains the entries 1, 2, 6, 18, 54, 162, ..., which is the sequence $2 \cdot 3^{k-1}$. Thus, we see that the norm of $\mathbf{U}^{-1}$ must grow exponentially with the dimension, being bigger than any individual element of the matrix such as the $U_{1,n} = 2 \cdot 3^{n-1}$ entry; further, since the 2-norm of $\mathbf{U}^{-1}$ is the reciprocal of the smallest singular value of $\mathbf{U}$, we see that the smallest singular value of $\mathbf{U}$ is exponentially small with the dimension n. This is a somewhat surprising situation: A simple upper-triangular matrix whose determinant is 1 and whose entries are not larger than -2 can be made arbitrarily close to being singular simply by increasing the dimension. ◁

4.6.2 Conditioning, Equilibration, and the Determinant

Is 10^{-17} zero? How about 16^{-300}? Of course, neither of these numbers is zero. However, there are situations in which they may as well be, such as in adding $1 + 10^{-17}$ or evaluating $y = x^3$ for $x = 16^{-100}$ in IEEE double precision. In the first case, 10^{-17} is less than the machine epsilon and the result rounds to 1, and in the second case 16^{-300} underflows to zero because it is smaller than 10^{-361}, which is less than `realmin`.

But there are situations where neither should be zero. Consider the 100×100 identity matrix $\mathbf{I}$ and let $\mathbf{A} = 16^{-3}\mathbf{I}$. Then the determinant of $\mathbf{A}$ would underflow to zero, implying that the matrix is singular. Obviously, it isn't, and indeed all the singular values $\sigma_k = 16^{-3}$ and so its condition number is 1. It's even true that solving $\mathbf{Ax} = \mathbf{b}$ can be done without rounding error on a binary system (although the result might overflow if the entries in $\mathbf{b}$ are big enough). The two tests for singularity, namely, the one learned in the first course about the determinant being zero or not, versus the "nearness to singularity" notion that we get from the SVD, have quite different behavior, and the second is a good deal more reliable.

The theory of conditioning is elaborate (and indeed we have only just started). For many reasons, it would be very useful to have a simpler theory, based on something we know and understand, such as determinants. Unfortunately, there is only a weak, one-way connection between condition numbers and determinants, as we will see. Until recently, even this connection was not expected, as is shown by the following quote from a well-known (and loved) numerical linear algebra book.

> It is natural to consider how well determinant size measures ill-conditioning. If $\det(A) = 0$ is equivalent to singularity, is $\det(A) \approx 0$ equivalent to near singularity? Unfortunately, there is little correlation between $\det(A)$ and the condition of $Ax = b$.

It is evident on a quick inspection that a small determinant does not imply an ill-conditioned matrix; and the converse is not true either. The examples for both ways from the book quoted above are

$$\mathbf{B}_n = \begin{bmatrix} 1 & -1 & \cdots & -1 \\ 0 & 1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}, \tag{4.40}$$

which has determinant 1 while $2^{n-1} < \kappa(\mathbf{B}_n) < n^2 2^{n-1}$, and the perfectly conditioned matrix $\mathrm{diag}(1/10, 1/10, \ldots, 1/10)$ whose determinant is 10^{-n}.

One problem is *scaling*: If we multiply a matrix $\mathbf{A}$ by 10, then the determinant changes by a factor 10^n, while the singular values each change by a factor only of 10, so that the ratio σ_1/σ_n remains unchanged. Note that scaling by *nonconstant* diagonal matrices will indeed affect the singular values, by the way.

If we follow this idea and scale the matrix to make things fair—in particular, if we *equilibrate* the matrix by multiplying each row by a factor so that the Euclidean norm of each new row is just 1—then it turns out that there is a connection, and it is a very simple one. If $\mathbf{M}$ is the equilibrated version of $\mathbf{A}$, then we have

$$\kappa(\mathbf{M}) < \frac{2}{\det(\mathbf{M})} \tag{4.41}$$

and the constant 2 is the best possible. The proofs of this, given in Guggenheimer et al. (1995), are very instructive and show when this inequality is likely to be tight (namely, when all singular values but the smallest are roughly equal) and when it will be loose (namely, when there are many singular values of different orders of magnitude).

A further difficulty not noted there but worth worrying a little about is the possibility that row-equilibration might not always improve the condition number; that is, it might not be necessary that $\kappa(\mathbf{M}) \leq \kappa(\mathbf{A})$. But it is shown by Higham (2002 p. 136) that row-equilibration is the optimal row scaling, and in fact we always have $\kappa(\mathbf{M}) \leq \kappa(\mathbf{A})$.

This brings up the possibility of *column* scaling, and there is some interesting work on this hard problem. As noted by Skeel (1980), to do proper column scaling for solving a linear system, you have to already know something about the solution, so this is difficult. However, as noted by van der Sluis (1969), one can find (relatively simply) a diagonal matrix $\mathbf{D}_1$ such that the condition number of $\mathbf{AD}_1$ is not too much larger than the optimal column scaling—at worst, a factor $n^{1/2}$ larger. The method is to choose $\mathbf{D}_1$ so that

$$\mathbf{D}_1\mathbf{A}^H\mathbf{AD}_1$$

has unit diagonal entries. Once that is done, a further row-equilibration can be done as well. This helps, but automatic scaling isn't a panacea, and in any case, as the example below shows, this still doesn't rescue the determinant as a measure of nearness to singularity.

Example 4.11. The row-equilibrated version of the matrix $\mathbf{B}_n$ from Eq. (4.40) has $\det(\mathbf{M}_n) = 1/\sqrt{n!}$, and hence we know $\kappa(\mathbf{M}_n) < 2\sqrt{n!}$. However, this upper bound is

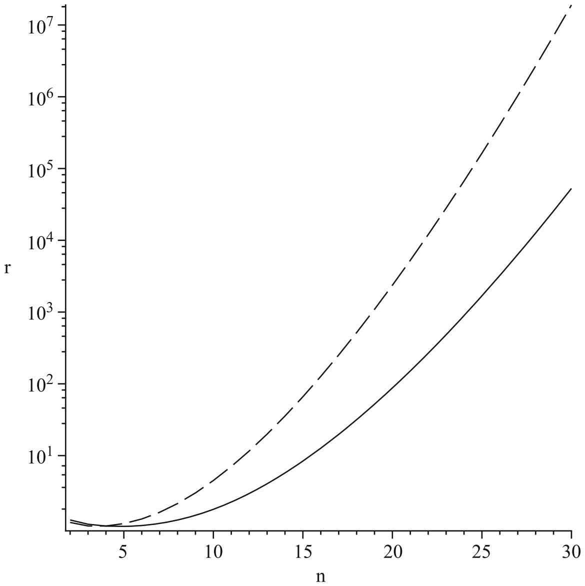


Fig. 4.8 Ratio $r = (2/\det \mathbf{M})/\kappa_2(\mathbf{M})$ for the matrices $\mathbf{B}_n$ after row-equilibration. *Dashed line* is without column-equilibration, and we see the bound becoming arbitrarily loose, quickly. With the simple column scaling first (*solid line*), the situation is better, but still bad. The closeness of the determinant to zero cannot be used to measure nearness to singularity

significantly larger than $\kappa(\mathbf{B}_n)$ for large n. That is, this bound can be arbitrarily bad. Experiments verify that for $n \leq 80$, we have $\kappa(\mathbf{M}_n) < \kappa(\mathbf{B}_n) < n^2 2^{n-1} \ll \sqrt{n}!$. In this case, the upper bound given by the theorem of Guggenheimer et al. (1995) is very loose.

For column scaling, computation shows that $\mathbf{D}_1 = \mathrm{diag}(1, 1/\sqrt{2}, 1/\sqrt{3}, \ldots, 1/\sqrt{n})$ makes the diagonal elements of $\mathbf{B}_n^T \mathbf{B}$ equal to 1. This is within a factor $\sqrt{n}$ of optimal column scaling to improve the condition number. When we apply this column scaling, and then do row-equilibration, we find that, for the example case $n = 30$, $2/\det \mathbf{M}$ is about 52,000 times the 2-norm condition number of $\mathbf{M}$. Without prior column scaling, the ratio is even higher, namely, about $1.9 \cdot 10^7$. Other dimensions n are computed and displayed in Fig. 4.8, and we see faster than exponential growth in the overpredictions by both bounds. This demonstrates that the smallness of the determinant cannot, even with row and column scaling, always be a good predictor of nearness to singularity.

Incidentally, the column scaling for this example does improve the condition number, but not by much; at $n = 30$, it is only about 6 times better. Most of the reduction in the ratio is because the determinant is scaled up. In other examples, column scaling can make the difference between success and failure. The problem of finding the *optimal* column scaling is harder than using this simple scaling and is studied by Watson (1991), for example.

◁

4.6.3 A Naive Way to Compute the SVD

We will not, in this book, study practical or efficient methods to compute the SVD. However, it is pedagogically valuable to know that there are methods to do so. Properly, we should leave this story aside until we know how to compute eigenvalues. Yet, at this moment, most readers will believe that there are decent methods to compute eigenvalues, and so a story that says that we can compute the SVD if we can compute eigenvalues should be satisfying. This being said, consider $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H \in \mathbb{C}^{m\times n}$. If we further consider the product

$$\mathbf{A}^H\mathbf{A} = \mathbf{V}\boldsymbol{\Sigma}^H\mathbf{U}^H\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H = \mathbf{V}\boldsymbol{\Sigma}^H\boldsymbol{\Sigma}\mathbf{V}^H\,,$$

we notice that $\boldsymbol{\Sigma}^H\boldsymbol{\Sigma}$ is diagonal and in $\mathbb{C}^{n\times n}$, with entries $\sigma_1^2, \sigma_2^2, \ldots, \sigma_r^2, 0, 0, \ldots, 0$ (if $r < n$) on the diagonal. That is, the eigenvalues of $\mathbf{A}^H\mathbf{A}$ are the squares of the singular values of $\mathbf{A}$. Incidentally, this shows that, in general, the singular values of $\mathbf{A}$ are not the eigenvalues of $\mathbf{A}$, although if $\mathbf{A}$ is symmetric positive definite, they are. Similarly, we obtain

$$\mathbf{A}\mathbf{A}^H = \mathbf{U}\boldsymbol{\Sigma}\boldsymbol{\Sigma}^H\mathbf{U}^H,$$

and $\boldsymbol{\Sigma}\boldsymbol{\Sigma}^H \in \mathbb{C}^{m\times m}$ and, again, has diagonal entries $\sigma_1^2, \sigma_2^2, \ldots, \sigma_r^2$ and possibly some zeros.

In exact arithmetic or for very small well-conditioned matrices, this is a feasible method, provided that one grants the capability to compute eigenvalues. But numerically, we are in trouble if $\sigma_r \ll \sigma_1$, because then σ_r^2 is even more disadvantaged by σ_1^2: Rounding errors or data errors will just destroy any accuracy. A better method, albeit requiring a bit more work, might be to look at the eigenvalues of the matrix

$$\begin{bmatrix} 0 & \mathbf{A} \\ \mathbf{A}^H & 0 \end{bmatrix} \in \mathbb{C}^{(m+n)\times(m+n)}\,,$$

sometimes called the Jordan–Wielandt matrix, which will include $\pm\sigma_1$, $\pm\sigma_2$, ..., $\pm\sigma_r$. This seems to require $O((m+n)^3)$ flops. If $m \approx n$, this is 8 times as much work as an eigenvalue problem of size $n \times n$.

4.6.4 Using Preexisting Software to Compute the SVD

The main algorithms used in practice, the Golub–Kahan–Reinsch and the Chan algorithms, are described, for example, in Golub and van Loan (1996) and Datta (2010). Here, we note only that they cost $O(n^3)$ flops (with a bigger constant than QR) and that both algorithms are backward stable (see Fig. 4.9).[16] In MATLAB, the

[16] In practice, we rely on MATLAB or on LAPACK (Anderson et al. 1999); MAPLE uses the NAG library code, which itself uses LAPACK.

computation of the SVD is simplicity itself. For example, let us consider a random matrix generated by executing `a = rand(5)`; here, in short format, we got

$$\begin{bmatrix} 0.8147 & 0.0975 & 0.1576 & 0.1419 & 0.6557 \\ 0.9058 & 0.2785 & 0.9706 & 0.4218 & 0.0357 \\ 0.1270 & 0.5469 & 0.9572 & 0.9157 & 0.8491 \\ 0.9134 & 0.9575 & 0.4854 & 0.7922 & 0.9340 \\ 0.6324 & 0.9649 & 0.8003 & 0.9595 & 0.6787 \end{bmatrix}.$$

We compute the SVD factoring of this matrix by simply executing

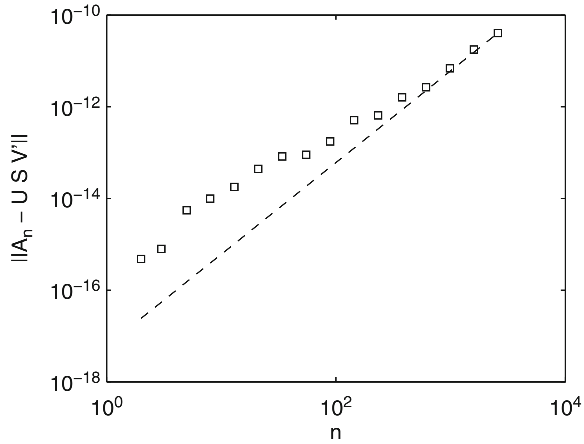


Fig. 4.9 Residual in SVD for various random matrices computed by MATLAB's effective algorithm. The *dashed line* is a constant times n^2

```
[u s v] = svd(a)
```

MATLAB returns the following factors:

$$\mathbf{U} = \begin{bmatrix} -0.2475 & -0.5600 & 0.4131 & 0.5759 & 0.3504 \\ -0.3542 & -0.5207 & -0.7577 & -0.0111 & -0.1707 \\ -0.4641 & 0.6013 & -0.1679 & 0.6063 & -0.1652 \\ -0.5475 & -0.1183 & 0.4755 & -0.3314 & -0.5919 \\ -0.5460 & 0.1992 & -0.0298 & -0.4369 & 0.6859 \end{bmatrix}$$

$$\boldsymbol{\Sigma} = \begin{bmatrix} 3.3129 & 0 & 0 & 0 & 0 \\ 0 & 0.9431 & 0 & 0 & 0 \\ 0 & 0 & 0.8358 & 0 & 0 \\ 0 & 0 & 0 & 0.4837 & 0 \\ 0 & 0 & 0 & 0 & 0.0198 \end{bmatrix}$$

$$\mathbf{V} = \begin{bmatrix} -0.4307 & -0.8839 & 0.0530 & -0.0884 & 0.1503 \\ -0.4309 & 0.2207 & 0.1961 & -0.7322 & -0.4370 \\ -0.4617 & 0.0890 & -0.7467 & 0.3098 & -0.3539 \\ -0.4730 & 0.3701 & -0.0798 & -0.1023 & 0.7890 \\ -0.4380 & 0.1585 & 0.6283 & 0.5913 & -0.1968 \end{bmatrix}$$

Moreover, we can compute the residual by executing this command:

```
resid = a - u * s * v'
```

In this case, MATLAB returns the matrix

$$10^{-15} \begin{bmatrix} -0.1110 & 0.1943 & -0.0278 & -0.0278 & -0.6661 \\ -0.1110 & -0.0555 & -0.6661 & -0.3331 & 0.2637 \\ 0.1665 & -0.3331 & 0.1110 & -0.2220 & -0.3331 \\ 0 & -0.3331 & 0.1110 & 0 & -0.4441 \\ 0.1110 & -0.6661 & 0 & -0.1110 & -0.2220 \end{bmatrix}$$

The computed residual above is not, unfortunately, an exact residual (as in the case of simple matrix–vector multiplication); after all, there are two matrix–matrix multiplications. But it is a good indication that the factoring is accurate. Given this easy, reliable way to compute the SVD, we examine some of its applications in what follows.

4.6.5 Solving $\mathbf{Ax} = \mathbf{b}$ *with the SVD*

Let $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H$, so that $\mathbf{U}(\boldsymbol{\Sigma}(\mathbf{V}^H\mathbf{x})) = \mathbf{b}$. Then we obtain

$$\boldsymbol{\Sigma}(\mathbf{V}^H\mathbf{x}) = \mathbf{U}^H\mathbf{b}\,,$$

which is computed in m^2 flops. If $m = n = p = r$, we also have

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{bmatrix} \quad \text{and} \quad \boldsymbol{\Sigma}^{-1} = \begin{bmatrix} \sigma_1^{-1} & & & \\ & \sigma_2^{-1} & & \\ & & \ddots & \\ & & & \sigma_n^{-1} \end{bmatrix}.$$

It is thus easy to compute $\mathbf{V}^H\mathbf{x} = \boldsymbol{\Sigma}^{-1}(\mathbf{U}^H\mathbf{b})$, an operation that requires $O(n)$ flops. As a result, we find that

$$\mathbf{x} = \mathbf{V}(\boldsymbol{\Sigma}^{-1}(\mathbf{U}^H\mathbf{b}))\,.$$

Alternatively, if we let $\mathbf{V}^H\mathbf{x} = \mathbf{y}$, $\boldsymbol{\Sigma}\mathbf{y} = \mathbf{z}$, and $\mathbf{U}\mathbf{z} = \mathbf{b}$, then we decompose the problem as follows:

1. Solve $\mathbf{Uz} = \mathbf{b}$ for $\mathbf{z}$.
2. Solve $\mathbf{\Sigma y} = \mathbf{z}$ for $\mathbf{y}$.
3. Solve $\mathbf{V}^H\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$.

This is equivalent to solving $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$, but written as a sequence of three simple "solve" steps.

4.6.6 The SVD and Overspecified Systems

Let $\mathbf{A}$ be a tall, skinny matrix with full column rank (i.e., $m > n$ but $\sigma_n \gg 0$). Then

$$\mathbf{\Sigma} = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \\ & & & \\ & & & \end{bmatrix}.$$

In this case, $\mathbf{Ax} = \mathbf{U\Sigma V}^H\mathbf{x} = \mathbf{b}$ has too many equations, and it is to be solved in the least-squares sense. But as we have seen, using the SVD, we can minimize the 2-norm of the residual $\mathbf{r}$:

$$||\mathbf{r}||_2^2 = ||\mathbf{b} - \mathbf{Ax}||_2^2 = ||\mathbf{U}^H\mathbf{b} - \mathbf{U}^H\mathbf{Ax}||_2^2 = ||\mathbf{U}^H\mathbf{b} - \mathbf{\Sigma V}^H\mathbf{x}||_2^2.$$

Now, if we let $\mathbf{U}^H\mathbf{b} = \mathbf{w}$ and $\mathbf{V}^H\mathbf{x} = \mathbf{z}$, our objective is to minimize

$$\left\| \mathbf{w} - \begin{bmatrix} \sigma_1 & & & & & \\ & \ddots & & & & \\ & & \sigma_r & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \\ & & & & & \\ & & & & & \end{bmatrix} \mathbf{z} \right\|_2^2.$$

We can choose $\mathbf{x}$, which uniquely specifies $\mathbf{z}$. However, we have no choice about $\mathbf{w}$. In order to minimize the residual, we can choose $\mathbf{x}$ so that the entries of $\mathbf{z}$ satisfy

$$z_k = \frac{w_k}{\sigma_k},$$

but we are helpless thereafter. Choosing z_{r+1} to z_n makes no difference, so that the residual is given by

$$\|\mathbf{r}\|_2^2 = \overbrace{\sum_{i=1}^{r} |w_i - \sigma_i z_i|^2}^{\text{can change}} + \overbrace{\sum_{i=r+1}^{n} |w_i|^2 + \sum_{i=n+1}^{m} |w_i|^2}^{\text{cannot change}} . \tag{4.42}$$

This is *minimized* with the choice $z_k = {}^{w_k}/_{\sigma_k}$, for $1 \le k \le r$. We may as well choose $z_k = 0$ for $r+1 \le k \le n$, if any. Then $\mathbf{x} = \mathbf{Vz}$ gives a least-squares solution. Written explicitly, it is

$$\mathbf{x} = \sum_{i=1}^{r} \frac{w_i}{\sigma_i} \mathbf{v_i} ,$$

where $w_i = \mathbf{u_i}^H \mathbf{b}$, that is, the ith column of $\mathbf{U}$, conjugated and transposed, multiplied by the right-hand side. Therefore,

$$\mathbf{x} = \sum_{i=1}^{r} \frac{\mathbf{u}_i^H \mathbf{b} \mathbf{v}_i}{\sigma_i} \tag{4.43}$$

is "the" least-squares solution. In fact, any choice for $z_{r+1}, \ldots, z_n$ gives the same value for $\mathbf{r}$.

If $r < n$, $\mathbf{A}$ is said to be *column rank-deficient*, and least-squares solutions are then nonunique. In that case, the SVD can be used to chose the least-square solution with minimum norm, which matters in some applications. This idea can be used to solve linearly constrained least-squares problems, as we will see in Examples 6.9 and 6.10.

4.6.7 Other Applications of the SVD

In what follows, we briefly examine three other interesting applications of the SVD:

1. numerical rank and null space;
2. Moore–Penrose inverse;
3. data compression and approximation by a low-rank matrix.

In addition to being useful, the last one is also very entertaining. There are many other very nice applications of the SVD given by Muller et al. (2004). We will see another one in Chap. 6 that has to do with the GCD of approximate polynomials.

Numerical Rank

Suppose that $\sigma_{r+1} \neq 0$, but that it is pretty tiny, say $10^{-12}\sigma_1$. If ${}^{\sigma_{r+1}}/_{\sigma_1}$ is small enough, $\mathbf{A}$ can be considered *numerically rank-deficient*.

Example 4.12. We again return to Example 4.1. As we have noticed, $\mathbf{B} = \mathbf{A}^T\mathbf{A}$, where

$$\mathbf{A} = \begin{bmatrix} 666 & 665 \\ 667 & 666 \end{bmatrix} .$$

We saw some peculiar behavior when we tried to solve $\mathbf{Bx} = [0,1]^H$. Now, we can explain the forward error, and look at $\mathbf{A}$ as well. If we execute `A` to compute the singular values, MATLAB returns

$$10^3 \begin{bmatrix} 1.3320 \\ 0.0000 \end{bmatrix} ,$$

and we can find $\kappa(\mathbf{A}) = {}^{\sigma_1}/_{\sigma_n}$ by executing `kappa = ans(1)/ans(2)`, finding the value $1.7742 \cdot 10^6$. Moreover, as we have seen, the reference value of $\mathbf{x}$ is $[-665, 666]^T$. Let this be `refx`. We can then compute the relative forward error, in 2-norm, by executing

```
relerr = norm( x - refx, 2)/norm( refx )
```

and find that MATLAB returns $6.6870 \cdot 10^{-11}$. For the sake of comparison, computing a bound for the forward error using Eq. (4.38) by executing

```
errorbound = kappa*norm(residual,2)/norm([0,1],2)
```

gives $1.0327 \cdot 10^{-4}$. We see in this case that the condition number estimate for the forward error is a great overestimate; that is, the error is better than it might have been.

Now let us look at $\mathbf{B}$. In Example 4.7, we have seen how to find a computed solution as well as its residual for $\mathbf{Bx} = [0,1]^H$ by the QR method. We have found the residual to be exactly zero and have explained this surprising fact with Theorem 4.1: The residual is correct only if we allow tiny perturbations to $\mathbf{B}$; thus, we have found the exact solution to $(\mathbf{B} + \Delta\mathbf{B})\mathbf{x} = [0,1]^H$. This is remarkable, and worth further exploration, a thing that can be done using the SVD. In Exercise 4.23, you are asked to see if you can *compute* (in high precision, say in MAPLE) a tiny perturbation of this matrix that has this solution. This is followed up by using the SVD in Example 6.9. Here we also see, as you are asked to show in Exercise 4.25, that the relative forward error is bounded by the condition number multiplied by the relative backward error, this time ${}^{\|\Delta\mathbf{A}\|}/_{\|\mathbf{A}\|}$, which we know to be about the size of the machine epsilon. Thus, for this matrix, the size of the forward error should be about 10^{12} times 10^{-16}, that is, 10^{-4}. In fact, if we execute

```
refx = [-B(2,1), B(1,1)]';
relerr = norm( x - refx, 2)/norm( refx )
```

MATLAB returns $2.9833 \cdot 10^{-5}$. This time, the condition number gave quite a good estimate of the forward error; that is, the error was very nearly the worst possible.

Notice that $\mathbf{B}$ is “almost” a rank-1 matrix. In fact, its singular values are

$$\begin{aligned} \sigma_1 &= 887113 + 1332\sqrt{443557} \approx 1.774 \cdot 10^6 \\ \sigma_2 &= \frac{1}{\sigma_1} \approx 5.6362 \cdot 10^{-7} . \end{aligned}$$

The matrix $\mathbf{A}$ has singular values that are just the square roots of these, so they are about 10^3 and 10^{-3}. The condition number of $\mathbf{A}$ is about 10^6, while the condition number of $\mathbf{B}$ is about 10^{12}. Using Theorem 4.6, we can find the nearest rank-1 matrix:

$$\mathbf{B}_1 = \mathbf{U}\begin{bmatrix}\sigma_1 & \\ & 0\end{bmatrix}\mathbf{V}^H = \begin{bmatrix}\frac{888445}{2}+\frac{591705037\sqrt{443557}}{887114} & 443556+\frac{295408629\sqrt{443557}}{443557} \\ 443556+\frac{295408629\sqrt{443557}}{443557} & \frac{885781}{2}+\frac{589930811\sqrt{443557}}{887114}\end{bmatrix}$$

$$\approx \mathbf{B} + \begin{bmatrix}-0.28128988\cdot 10^{-6} & 0.28181270\cdot 10^{-6} \\ 0.28181270\cdot 10^{-6} & 0.28223616\cdot 10^{-6}\end{bmatrix}$$

Remember that $\|\mathbf{B}\|_2 = \sigma_1 \approx 1.774\cdot 10^6$. This matrix, which is so close to $\mathbf{B}$, is, in fact, singular; in fact, *it is the nearest singular matrix*. It ought not to be surprising, then, that solution of this linear system in a floating-point environment will be quite sensitive to rounding errors; after all, the solution would be quite sensitive to data errors too. ◁

If the SVD of $\mathbf{A}$ has tiny singular values, say σ_{q+1}, σ_{q+2}, ..., σ_r, and none of them is zero, then technically the *null space* of $\mathbf{A}$ is empty; but it might be better to think of the space spanned by $\mathbf{v}_{q+1}, \mathbf{v}_{q+2}, \ldots, \mathbf{v}_r$ (which is an orthogonal basis, a fact that is quite convenient) as your “approximate” null space. After all, the norm of $\mathbf{Av}$ will be less than about σ_{q+1} for any vector in that space; in fact, a small $O(\sigma_{q+1})$ perturbation of $\mathbf{A}$ will have $\mathbf{v}$ as a null vector. We remark that each individual vector in that space is quite sensitive to perturbations, but that the approximate null space as a whole is relatively stable.

Moore–Penrose Inverse

Let the matrix $\mathbf{\Sigma}^+$ be such that $\mathbf{\Sigma}^+ = \mathrm{diag}(\sigma_1^{-1}, \sigma_2^{-2}, \ldots, \sigma_r^{-r}, 0, \ldots, 0)$ and

$$\mathbf{A}^+ = \mathbf{V}\mathbf{\Sigma}^+\mathbf{U}^H .$$

Then, the Moore–Penrose conditions, which are

$$\begin{aligned}\mathbf{AA}^+\mathbf{A} &= \mathbf{A} \\ \mathbf{A}^+\mathbf{AA}^+ &= \mathbf{A}^+ \\ (\mathbf{AA}^+)^H &= \mathbf{AA}^+ \\ (\mathbf{A}^+\mathbf{A})^H &= \mathbf{A}^+\mathbf{A}\,,\end{aligned}$$

all hold (see, e.g. Golub and van Loan 1996). For singular $\mathbf{A}$, and some applications, $\mathbf{A}^+$ is as good as an inverse.

Data Compression or Approximation by a Lower-Rank Matrix

Let the matrix **A** be factored using the SVD, so that

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H = \left(\sum_{i=1}^{r} \sigma_i \mathbf{u}_i\right)\mathbf{V}^H = \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^H .$$

Now, keep only the few largest singular values σ_i; that is, take

$$\mathbf{A} \approx \sum_{i=1}^{\text{few}} \sigma_i \mathbf{u}_i \mathbf{v}_i^* =: \mathbf{A}_{\text{few}} \,, \tag{4.44}$$

which uses only a few $\mathbf{u}_i$ and $\mathbf{v}_i$ to represent **A**. The next singular value tells us how closely this low-rank approximation is to **A**; namely,

$$\frac{\|\mathbf{A} - \mathbf{A}_{\text{few}}\|}{\|\mathbf{A}\|} = \frac{\sigma_{\text{few}+1}}{\sigma_1} \,.$$

Example 4.13. Consider the closeup photograph of a tawny frogmouth (*Podargus strigoides*), displayed in Fig. 4.10. Taking the data for this picture, we get three matrices each of size 985 × 1314 (some digital zoom and select was used). We take the SVD of those matrices and find by inspection of the singular values that most of them are quite small compared to the first 25. We compute the nearest rank-25 matrices to these by zeroing out all singular values but these, and reconstruct the matrices using Eq. (4.44). This gives the picture displayed in Fig. 4.11. This compression has been done using the following code, which is worth examining in order to understand the power and simplicity of the SVD.

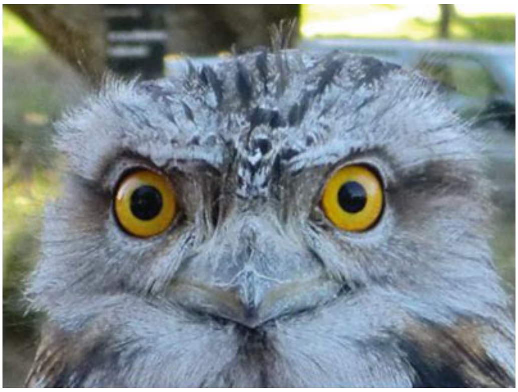


Fig. 4.10 A close-up of a tawny frogmouth sitting on RMC's rental car at Australian National University in 2011

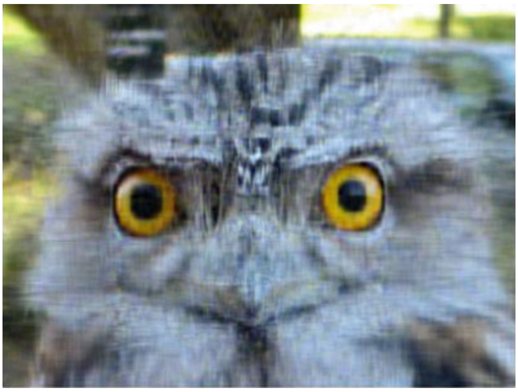


Fig. 4.11 The same picture, but with only 25 nonzero singular values instead of 985. This represents a significant compression: The picture can be stored or transmitted using only 25 **u**-vectors, **v**-vectors, and singular values, reducing the storage per matrix from 985 × 1141 bytes (or about 1000 Mbytes) to about 50 Kbytes. Of course, the image quality suffers, but the bird is recognizable

```
function pcargbtawny2

%Get data
Y = imread('tawny.bmp','bmp');
[m,n,o]=size(Y)
image(Y)
%colormap(gray(256))
axis image, axis off

class(Y)

%Convert matrix to double so that we can use the command svd
%we'll have to re-convert to uint8 later to use image properly.
X=double(Y);

% %Now, let's look at a logarithmic plot of the singular values
Xr=X(:,:,1);
Xg=X(:,:,2);
Xb=X(:,:,3);
[Ur,Sr,Vr] = svd(Xr);
sigmar=diag(Sr);
figure
semilogy(sigmar,'.')
[Ug,Sg,Vg] = svd(Xg);
sigmag=diag(Sg);
figure
semilogy(sigmag,'.')
[Ub,Sb,Vb] = svd(Xb);
sigmab=diag(Sb);
figure
semilogy(sigmab,'.')

```

```
%Finally, let's compress the image, keeping only k singular
    values.
k=25;
AppXr=zeros(m,n);
AppXg=zeros(m,n);
AppXb=zeros(m,n);
 for i=1:k
     AppXr=AppXr+sigmar(i)*Ur(:,i)*Vr(:,i)';
     AppXg=AppXg+sigmag(i)*Ug(:,i)*Vg(:,i)';
     AppXb=AppXb+sigmab(i)*Ub(:,i)*Vb(:,i)';
 end
AppX(:,:,1)=AppXr;
AppX(:,:,2)=AppXg;
AppX(:,:,3)=AppXb;


AppX=uint8(AppX);
figure
image(AppX)
axis image, axis off

end
```

◁

4.7 Solving Ax=b with the LU Factoring

We at last rejoin the mainstream of numerical analysis texts, with a consideration of Gaussian elimination. We begin with a small example. Suppose we are given a system of equations such as

$$\begin{aligned} 2x_1 + 2x_2 + 4x_3 &= -10 \\ 3x_1 + 5x_2 + 7x_3 &= -11 \\ x_1 + 7x_2 + 7x_3 &= -6\,. \end{aligned}$$

We know that we can solve this system using Gaussian elimination, adding multiples of one row to another in order to remove (eliminate) variables from equations, one by one. To begin with, we could add $-3/2$ times the first row to the second, and $-1/2$ times the first row to the third. Continuing in this way, working with the augmented matrix, we obtain

$$\left[\mathbf{A} \mid \mathbf{b}\right] = \left[\begin{array}{ccc|c} 2 & 2 & 4 & -10 \\ 3 & 5 & 7 & -11 \\ 1 & 7 & 7 & -6 \end{array}\right] \xrightarrow{\text{elimination}} \left[\begin{array}{ccc|c} 2 & 2 & 4 & -10 \\ 0 & 2 & 1 & 4 \\ 0 & 0 & 2 & -12 \end{array}\right] = \left[\mathbf{U} \mid \mathbf{b}_2\right]. \tag{4.45}$$

From the upper-trapezoidal augmented matrix, we then find that $x = [2,5,-6]^T$ by back substitution.

In the process of reduction, we have subtracted multiples of rows from other rows. Specifically, to reduce the first column, we have added $-{}^{3}\!/_{2}$ times row 1 to row 2, and $-{}^{1}\!/_{2}$ times row 1 to row 3. To reduce the second column, we have added -3 times the new second row to the new third row. In this form, however, the operations executed are not recorded. But we *can* record those operations, by encoding the row operations as matrix multiplications. The so-called elementary matrix $\mathbf{L}_1$ and its inverse $\mathbf{L}_1^{-1}$ (also an elementary matrix), given by

$$\mathbf{L}_1 = \begin{bmatrix} 1 & 0 & 0 \\ {}^{3}\!/_{2} & 1 & 0 \\ {}^{1}\!/_{2} & 0 & 1 \end{bmatrix} \qquad \text{and} \qquad \mathbf{L}_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -{}^{3}\!/_{2} & 1 & 0 \\ -{}^{1}\!/_{2} & 0 & 1 \end{bmatrix},$$

play a starring role. When multiplied on the left to $\mathbf{A}$, $\mathbf{L}_1^{-1}$ can be interpreted as adding $-{}^{3}\!/_{2}$ times row 1 to row 2, and adding $-{}^{1}\!/_{2}$ times row 1 to row 3. Thus, we can write

$$\mathbf{A} = \mathbf{L}_1\mathbf{L}_1^{-1}\mathbf{A} = \mathbf{L}_1 \begin{bmatrix} 2 & 2 & 4 \\ 0 & 2 & 1 \\ 0 & 6 & -1 \end{bmatrix},$$

and we have now reduced the 3×3 problem to a 2×2 problem. The next step proceeds in the same way; indeed, this construction provides a viable recursive formulation for the LU factoring if we embed the smaller factoring in the lower-right-hand corner, as follows:

$$\mathbf{A} = \mathbf{L}_1 \begin{bmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{L}_2 \end{bmatrix} \begin{bmatrix} 2 & 2 & 4 \\ 0 & 2 & 1 \\ 0 & 0 & -4 \end{bmatrix} \qquad \text{with} \qquad \mathbf{L}_2 = \begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix}.$$

Grouping the lower-triangular elementary matrices together, we arrive at $\mathbf{A} = \mathbf{LU}$, all by the simple act of recording the multiples of the "pivot" row used for elimination.

Just to be absolutely clear, the key step requires understanding what the inverse $\mathbf{L}^{-1}$ of an elementary matrix $\mathbf{L}$ is. Another example of this would be

$$\mathbf{L} = \begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&4&1&0&0&0&0 \\ 0&0&5&0&1&0&0&0 \\ 0&0&6&0&0&1&0&0 \\ 0&0&7&0&0&0&1&0 \\ 0&0&8&0&0&0&0&1 \end{bmatrix} \qquad \text{and} \qquad \mathbf{L}^{-1} = \begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&-4&1&0&0&0&0 \\ 0&0&-5&0&1&0&0&0 \\ 0&0&-6&0&0&1&0&0 \\ 0&0&-7&0&0&0&1&0 \\ 0&0&-8&0&0&0&0&1 \end{bmatrix}.$$

The product of a sequence of such elementary triangular pieces (in the right order) will build up a lower-triangular matrix; inserting pairs $\mathbf{LL}^{-1}$ into $\mathbf{A}_1\mathbf{A}_2{=}\mathbf{A}_1\mathbf{LL}^{-1}\mathbf{A}_2$

allows the gradual splitting of two initial factors **IA** into a product of a lower-triangular factor and an upper-triangular factor. That is so, provided that no zero pivots are encountered.

Why factor, when one can use elimination? The first answer to this is that one might wish to solve $\mathbf{Ax} = \mathbf{b}$ for several right-hand-side vectors **b**. In that case, rather than redo the elimination each time, one can store the factors and for each **b** solve the two triangular systems to get the correct **x** each time. Another answer is that the factoring often reveals useful information about the matrix itself; sometimes the factoring is the solution to the question one ought to have been asking.

4.7.1 Instability of Elimination Without Pivoting

That Gaussian elimination is the same as LU factoring is a basic fact of linear algebra. However, the story does not end here in *numerical* linear algebra. Consider the following system:

$$\begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 7 \end{bmatrix}.$$

By inspection, we see that if $|\varepsilon| \ll 1$, the solution should be close to $\mathbf{x} = [4,\ 3]^T$. Using Gaussian elimination, but this time with floating-point arithmetic, we obtain the following:

$$\left[\begin{array}{cc|c} \varepsilon & 1 & 3 \\ 1 & 1 & 7 \end{array}\right] \xrightarrow{R_2 \ominus \frac{1}{\varepsilon} \otimes R_1} \left[\begin{array}{cc|c} \varepsilon & 1 & 3 \\ 0 & 1 - {}^{1}\!/\!_{\varepsilon} & 7 - {}^{3}\!/\!_{\varepsilon} \end{array}\right],$$

We follow this operation with a back substitution, also in floating-point arithmetic:

$$\begin{aligned} x_2 &= \left(7 - \frac{3}{\varepsilon}\right) \oslash \left(1 - \frac{1}{\varepsilon}\right) \\ x_1 &= (3 \ominus x_2) \oslash \varepsilon \end{aligned}$$

In MATLAB, if we let $\varepsilon = 10^{-16}$, we obtain the solution

$$\mathbf{x} = \begin{bmatrix} 8.881784197001252 \\ 2.999999999999999 \end{bmatrix},$$

which is a far cry from the reference answer near $[4,3]$. And yet, this matrix is quite well-conditioned. It is easy to see that, for floating-point arithmetic of any given precision, we will be able to find some values of ε for which Gaussian elimination will return dramatically wrong answers. The source of the error lies in the bad numerical management of matrix entries of very different orders of magnitude. Since the pivot (i.e., the a_{kk} entry occurring as divisor in the multiplier) is very small compared with the other subdiagonal entries, the multiplier is very large, much larger than the largest subdiagonal entries. Consequently, significant floating-point error will accumulate through the algorithm.

The backward error way of viewing the difficulty is to realize that there is, in general, no bound on $\Delta\mathbf{A}$ such that the computed $\mathbf{L}$ and $\mathbf{U}$ are the exact factors of $\mathbf{A}+\Delta\mathbf{A}$. For this example,

$$\begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1/\varepsilon & 1 \end{bmatrix} \begin{bmatrix} \varepsilon & 1 \\ 0 & 1-1/\varepsilon \end{bmatrix}$$

if exact arithmetic is used. The problematic case in floating-point is the computation of $1-1/\varepsilon$, which will round to $-1/\varepsilon$ (assuming ε is such that this is machine-representable) if $\varepsilon < \mu_M$, the unit roundoff. Even with just this single rounding error (no accumulation of error is necessary to demonstrate the problem) we have that the computed factors are the exact factors not of the original matrix, but of

$$\begin{bmatrix} \varepsilon & 1 \\ 1 & 0 \end{bmatrix}$$

which differs by 1 from the original matrix, which is unboundedly larger than the size of ε. That is, the computed LU factors are *not* the exact factors of any nearby matrix.

On the other hand, a conventional way of viewing the difficulty is to use the *growth factor* bound ρ, which is the ratio of the maximum element growth that occurs in elimination. Let $a_{i,j}^{(k)}$ be the entries in the matrix after k steps of elimination. Then ρ is defined as follows:

$$\rho := \frac{\max_{i,j,k} \left| a_{i,j}^{(k)} \right|}{\max_{i,j} |a_{i,j}|}\,.$$

In the example above, the multiplier for the row-reduction is ε^{-1}. Thus, the *growth factor* is such that $\rho = O(1/\varepsilon)$. The significance of the growth factor is seen from the following consideration. Assuming that no actual zero pivots are encountered (in which case the algorithm terminates incomplete), Gaussian elimination produces a solution $\hat{\mathbf{x}}$ that is the exact solution of $(\mathbf{A}+\Delta\mathbf{A})\,\hat{\mathbf{x}} = \mathbf{b}$, with

$$\|\Delta\mathbf{A}\|_\infty \le cn^3\rho u\|\mathbf{A}\|_\infty\,,$$

where c is a modest constant. However, for Gaussian elimination without pivoting, ρ can be unboundedly large as we have seen. As a result. the forward error can subsequently be large as well.

One solution to this problem consists in *partial pivoting*, which simply exchanges two rows in the current submatrix to ensure that the pivot used has the largest magnitude in the remainder of the column. This is not the only possible strategy, although it is very commonly used. *Complete* pivoting consists in looking at the not-yet-reduced submatrix for the largest entry in absolute value; we then interchange rows *and* columns so that the pivot is this entry. *Partial* pivoting is less costly, since it only searches one column used to find a multiplier. We refer to Gaussian elimination with partial pivoting as GEPP.

To be concrete, if, at the kth step of Gaussian elimination, we have a matrix

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1k} & \dots & a_{1n} \\ & a_{22} & \dots & a_{2k} & \dots & a_{2n} \\ & & \ddots & \vdots & & \vdots \\ & & & a_{kk} & \dots & a_{kn} \\ & & & \vdots & & \vdots \\ & & & a_{mk} & \dots & a_{mn} \end{bmatrix},$$

then we want the pivot p_k to be

$$p_k = \max\left\{ |a_{kk}|, |a_{k+1,k}|, \dots, |a_{m-1,k}|, |a_{mk}| \right\} .$$

We thus exchange the rows (or, equivalently, pointers to the rows) so that p_k becomes the new a_{kk} entry.

As is the case for simple Gaussian elimination, GEPP can be understood as a factoring. First, observe that just as adding a multiple of a row to another can be encoded as a multiplication on the left by an elementary matrix, also note that a row exchange can be encoded as a left multiplication by an elementary permutation matrix, namely, an identity matrix in which rows have been interchanged. For instance, if

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix},$$

then **PA** would exchange rows 1 and 4 in **A**. Second, one should understand how the elementary permutations that we need for pivoting percolate through elementary matrix encodings of row operations. Once that is understood, then it is obvious that GEPP can be understood as a matrix factoring. The key observations, which are easier to understand by running the MATLAB demo `lugui` than reading this, are that elementary row-permutation matrices are their own inverses (exchanging rows back again) and that when multiplied on the *left*, they interchange *columns*; in order to make the result lower-triangular again, one has to interchange rows again. Briefly, $\mathbf{A} = \mathbf{LU} = \mathbf{L}\,(\mathbf{PP})\,\mathbf{U}$, and then $\mathbf{A} = (\mathbf{LP})\,\hat{\mathbf{U}}$, so $\mathbf{PA} = (\mathbf{PLP})\,\hat{\mathbf{U}}$. If the matrix **L** has an identity matrix block in the lower-right corner, then exchanging any two columns of that block, followed by exchanging any two rows of that block, as the product **PLP** does, leaves the result lower-triangular.

Thus, GEPP proceeds in a way similar to GE, except that a sequence of permutation matrices is included, so that the pivots have the largest magnitude entries. If all possible pivots have zero magnitude (unlikely as this is in floating-point arithmetic), then the matrix is singular. The permutation matrix is determined by the indices k and the number of the row j ($k \leq j \leq m$) with the largest entry. We thus obtain a product of matrices of the form

$$\mathbf{U} = \mathbf{R}_{n-1}\mathbf{P}_{n-1}\ldots\mathbf{R}_k\mathbf{P}_k\ldots\mathbf{R}_2\mathbf{P}_2\mathbf{R}_1\mathbf{P}_1\mathbf{A}\,.$$

Note that all the subdiagonal entries of $\mathbf{R}_k\mathbf{P}_k\ldots\mathbf{R}_2\mathbf{P}_2\mathbf{R}_1\mathbf{P}_1\mathbf{A}$ are zero, so $\mathbf{U}$ is upper-triangular.

Percolating the permutation matrices through the elementary lower-triangular matrices as discussed shows that Gaussian elimination with partial pivoting is equivalent to a

$$\mathbf{PA} = \mathbf{LU}$$

factoring, where $\mathbf{P}$ is a permutation matrix (being a product of the elementary permutations that encode the row exchanges that took place), $\mathbf{L}$ is unit lower-triangular, and $\mathbf{U}$ is upper-triangular. If $\mathbf{A}$ is rectangular, we can use the Turing factoring $\mathbf{PA} = \mathbf{LD}^{-1}\mathbf{UR}$, where $\mathbf{R}$ is the row-echelon form.[17] Indeed, there are a great many variations of the LU factoring.

On the basis of the $\mathbf{PA} = \mathbf{LU}$ factoring, we can also solve the equation $\mathbf{Ax} = \mathbf{b}$ in a straightforward manner. We need only notice the following implications (in the square nonsingular case):

$$\mathbf{Ax} = \mathbf{b} \quad \Rightarrow \quad \mathbf{PAx} = \mathbf{Pb} \quad \Rightarrow \quad \mathbf{LUx} = \mathbf{Pb}.$$

So, we can split the system so that $\mathbf{L}(\mathbf{Ux}) = \mathbf{Pb}$ and solve

$$\mathbf{Ly} = \mathbf{Pb}\,,$$

where $\mathbf{y} = \mathbf{Ux}$. Then we solve

$$\mathbf{Ux} = \mathbf{y}\,.$$

These are cheap to solve: $O(n^2)$ given the factoring, which costs $O(n^3)$ (see Fig. 4.12). In MATLAB, we use `lu` or \ (backslash).

The alert reader will have noticed that we have not provided a formal algorithm for $\mathbf{PA} = \mathbf{LU}$ factoring. Algorithms can be found, for example, in Golub and van Loan (1996) and in many other places; but implementing such algorithms is another story. There are a great many details that deserve attention (for example, efforts to avoid overflow and underflow if it isn't necessary). Frankly, we're all better served to use the very fine implementations provided by LAPACK (and thereby MATLAB and MAPLE, for instance). One hopes that the essence of the algorithms will have been conveyed by the foregoing discussion.

[17] See Corless and Jeffrey (1997).

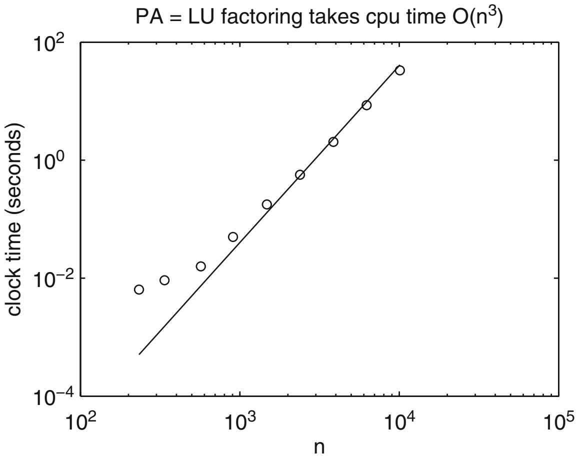


Fig. 4.12 Computing time for $\mathbf{PA} = \mathbf{LU}$. Notice that we may solve larger systems with $\mathbf{PA} = \mathbf{LU}$ in the time it takes SVD to solve smaller systems; but there is more information in an SVD solution. The reference line is a constant times n^3

4.7.2 Numerical Stability of Gaussian Elimination

If Gaussian elimination without pivoting does not encounter a zero pivot, Wilkinson (1963) showed that there exists a matrix $\mathbf{E}$ such that

$$(\mathbf{A} + \mathbf{E})\hat{\mathbf{x}} = \mathbf{b}\,,$$

with $\|\mathbf{E}\|_\infty \le cn^3\rho\|\mathbf{A}\|_\infty$, where ρ measures the growth of the maximum element during the process. It can happen that ρ is arbitrarily large, for general matrices. For some restricted classes of matrices, such as diagonally dominant or tridiagonal matrices, ρ is bounded.

As we have mentioned, pivoting improves the situation. The $\mathbf{PA} = \mathbf{LU}$ factoring (which can be computed with $n^3/3$ flops) is such that

$$\rho \le 2^{n-1}$$

for nonsingular matrices and attains this bound only in rare examples (see Trefethen and Schreiber 1990). Both complete pivoting and rook pivoting have better bounds, but are not used as often because they are more expensive and partial pivoting usually works well in practice.

In practice, ρ is almost always small for GEPP, but this fact calls out for a serious explanation, such as is begun in the reference just cited. We do not discuss this further in this book, but instead rely on a posteriori computation of the residual, which provides a guarantee for any particular computation.[18] See Fig. 4.13 this backward

[18] Moreover, the Oettli–Prager result that we will see in Sect. 6.6 gives us a computable minimal backward error. Also, as we will see in Chap. 6, a single pass of iterative improvement will fix many cases where the growth factor is unacceptably large.

error bound difficulty is why we have postponed the LU factoring until the end of the chapter.

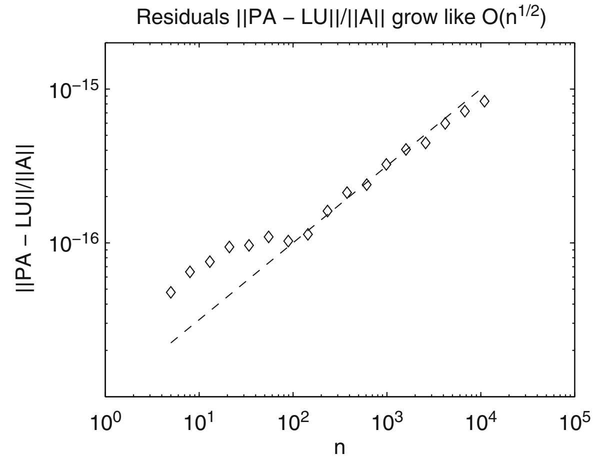


Fig. 4.13 Scaled residual in $\mathbf{PA} = \mathbf{LU}$, that is, $\|\mathbf{PA} - \mathbf{LU}\|/\|\mathbf{A}\|$ vs n, for $\mathbf{A} = \mathtt{rand}(n)$, with n being some Fibonacci number up to $n = 10,946$

Example 4.14. Again, let $\mathbf{A} = \mathbf{B}^T\mathbf{B}$, where

$$\mathbf{B} = \begin{bmatrix} 666 & 665 \\ 667 & 666 \end{bmatrix} .$$

We will again solve the equation $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{b} = [0,1]^T$. Here, instead of using the QR factoring, we use MATLAB's built-in backslash solver, which in this case uses the LU factoring. We will see a behavior similar to what was observed when we used the QR factoring or the SVD to solve the problem. So let us execute the command

```
x = A\b,
```

which returns

$$\mathbf{x} = 10^5 \begin{bmatrix} -8.871583031996495 \\ 8.884913727761688 \end{bmatrix} .$$

We can then compute the residual a posteriori by simply executing `r = b - A*x`; once again, we find that the computed residual is $\mathbf{r} = [0,0]^T$.

The entries of $\mathbf{x}$ ought to be (and would be if we were using exact arithmetic) just integers; they are not. Indeed, they are different from what we expect, already in the fifth significant digit. When we compute the residual (using IEEE double-precision) the answer is exactly zero, as it was for the SVD solution—that is, MATLAB is contending that we have found the exact solution. And so we have. We have found the exact solution to $(\mathbf{A}+\Delta\mathbf{A})\mathbf{x} = [0,1]^H$, and moreover we know that the entries of $\Delta\mathbf{A}$ are at most ε_M times the corresponding entries of $\mathbf{A}$. We saw this before, but it seems so extraordinary that we want to verify it by computing in higher precision (as you are asked to do yourself in Problem 4.23). We import the MATLAB results into MAPLE, and compute using (say) 100 digits there. We look for minimal perturbations δ_{ij} so that the perturbed matrix equation has this exact solution, and we find that

$$\Delta\mathbf{A} = \begin{bmatrix} -1.8214\times 10^{-11} & 1.8242\times 10^{-11} \\ 1.1186\times 10^{-11} & -1.1203\times 10^{-11} \end{bmatrix}$$

works very well.[19] Notice that each entry of $\mathbf{A}$ is about 10^6; indeed, when we divide each of the entries in $\Delta\mathbf{A}$ above by the corresponding entry of $\mathbf{A}$, we get

$$\begin{bmatrix} -2.0501\times 10^{-17} & 2.0563\times 10^{-17} \\ 1.2609\times 10^{-17} & -1.2648\times 10^{-17} \end{bmatrix}.$$

That is, MATLAB did not lie to us: That solution, strange as it seems, is the *exact* solution of a linear system that is within machine epsilon (in a relative sense) of the specified (integer) system.

We can again use the SVD to help us to explain the forward error. As we noted, the ratio of the largest to the smallest singular values was about 10^{12}. Thus, the forward error can be expected to be about $10^{12}\cdot 10^{-16}$, that is, 10^{-4}. When we compute the relative forward error by executing

```
relerr = norm( x - refx, 2)/norm( refx )
```

(where `x=A\[0,1]'`), we find the value $5.2195\cdot 10^{-5}$, which is the right order of magnitude (and about twice what the forward error in the SVD solution was). ◁

Remark 4.6. Above we have shown how to compute the residual a posteriori. This being said, we emphasize that until you compute the condition number, you haven't finished solving your problem. The condition number explains forward error, given numerical rounding and approximation errors in the solution, yes; but that's not its most important purpose. It also explains the sensitivity of the solution to *data error*, which you have to think about anyway. ◁

4.8 Why Not Use $\mathbf{A}^{-1}$?

In a first course in linear algebra, one is taught about the inverse: If the determinant is not zero, the inverse exists, and the solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$ is just $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. In this section, we ask whether the matrix $\mathbf{A}^{-1}$ should be used in numerical linear algebra.

[19] It's not symmetric, though; but with about the same effort we can, in fact, find a symmetric perturbation that is very nearly as small.

The question arises because we typically don't have to use it; as Forsythe and Moler (1967) observe, "[A]lmost anything you can do with $\mathbf{A}^{-1}$ can be done without it."

The first reason for which one does not usually compute the inverse in order to solve the system is simply cost. One way to compute the inverse, often the cheapest way, is to use Gaussian elimination on an augmented matrix, and one can easily check that the cost is about the same as that of one LU factoring plus $2n$ solutions of triangular systems, plus a final matrix multiplication. This should be compared with the cost of one LU factoring plus 2 solutions of triangular systems. Since a matrix multiplication costs about twice what the solution of a triangular system is, one sees that, in terms of cost, the $\mathbf{PA} = \mathbf{LU}$ factoring wins.

The second reason is more subtle. Backward error results exist for the solution of $\mathbf{Ax} = \mathbf{b}$; each computed solution is the exact solution of a nearby linear system. But the computed matrix inverse is *not* usually the exact inverse of a very nearby matrix! Instead, each column of the computed inverse is the exact column of the inverse of a nearby matrix—which is not quite the same thing. This distinction complicates some numerical analyses.

Nonetheless, for some applications, one indeed wants to know the inverse—the entries of the inverse may be the numbers one wants to know. In that case, it turns out that a mixed backward and forward error analysis is best. The computed entries of the inverse are then supposed to be (and estimated to be) nearly the exact entries of the inverse of nearly the right matrix. We do not pursue this further here.

Example 4.15. Consider the usual formula for complex division:

$$u + iv = \frac{a+ib}{c+id} = \frac{(a+ib)(c-id)}{c^2+d^2} = \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2}. \tag{4.46}$$

As discussed by Higham (2002), although this formula looks innocuous enough, it is susceptible to overflow or underflow at intermediate stages in the computation—both bc and c^2 may overflow, for example, while their ratio would fit quite nicely into the range of standard floating-point numbers. This difficulty has produced several responses.[20] Here we look at a very simple approach[21]: Replace a formula with an algorithm directly motivated by factoring a matrix problem. The underlying idea is that complex numbers can be concretely realized as matrices; indeed, the complex number $c + di$ can be thought of as the real matrix

$$\begin{bmatrix} c & -d \\ d & c \end{bmatrix}$$

and the process of complex division above can be interpreted as the solution of the linear system

[20] They include a much-used algorithm due to Smith (1962), a modification for increased robustness and accuracy by Stewart (1985), and most recently an efficient and very robust algorithm discussed in Priest (2004).

[21] We admit that this approach is not practical since it is at least 50% too expensive, because it doesn't use the symmetry. This is really just for practice with factoring.

$$\begin{bmatrix} c & -d \\ d & c \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} .$$

Of course, the inverse of this 2×2 matrix is easily available and leads us directly to the formula used above, which suffers from overflow and/or underflow.[22] What if, instead of using the inverse, we factor the matrices? For example, we might choose to use the $\mathbf{PA} = \mathbf{LU}$ factoring, pivoting if $|d| > |c|$ (or, equivalently, dividing $-i(a+ib)$ by $-i(c+id)$).[23] Because of the pivoting, we can assume therefore that $|d| \le |c|$. Then no overflow can occur if we divide c by d, although underflow certainly may. If we do so, we find that the factoring is

$$\begin{bmatrix} c & -d \\ d & c \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ {}^{d}\!/\!{}_{c} & 1 \end{bmatrix} \begin{bmatrix} c & -d \\ 0 & c + ({}^{d}\!/\!{}_{c})d \end{bmatrix} ,$$

where the parenthesized formula $c + ({}^{d}\!/\!{}_{c})d$ is used because again it almost always won't overflow (whereas ${}^{d^2}\!/\!{}_{c}$ is much more likely to). Then, solving the factored equations and unrolling the loops gives us the following algorithm for complex division:

$$\begin{aligned} t_0 &= {}^{d}\!/\!{}_{c} \\ t_1 &= b - t_0 a \\ t_2 &= c + t_0 d \\ v &= \frac{t_1}{t_2} \\ u &= {}^{a}\!/\!{}_{c} + t_0 v \, . \end{aligned} \tag{4.47}$$

This is no more susceptible to overflow or underflow than the ordinary solution of 2×2 systems,[24] and it has the usual normwise backward error together with the usual normwise condition number forward error. This is quite good—much better than the formula arising from the inverse matrix—but not perfect, and in particular not as good as the method described by Stewart (1985). Note that while the backward error is good, it does not respect the structure of the matrix (i.e., it does not automatically show that we have done an exact division by a different complex number $c(1+\delta_c) + id(1+\delta_d)$, although we will show how this can work, below), and it does not guarantee relative accuracy in each of the real and imaginary parts of the answer, u and v. But it works and avoids overflow and underflow to a large extent.[25]

[22] This is a bit of a straw man; of course, we could fix up the inverse formula, so this example isn't really a strong argument against using the inverse explicitly but still it gives some flavor.

[23] An alternative would be to use QR, which is greatly assisted by the fact that the columns of the matrix are already orthogonal; indeed, this amounts to converting the denominator to polar coordinates. We ignore that option for now.

[24] And it is no less susceptible, either. We did say this wasn't a practical algorithm. But it's better than the naive formula.

[25] Its real flaw is that it takes 6 real flops, whereas the formula of Smith takes only 4, making this example only of didactic interest.

Continuing the analysis, we want to find the residual, which is given by

$$\mathbf{r} = \begin{bmatrix} a \\ b \end{bmatrix} - \begin{bmatrix} c & -d \\ d & c \end{bmatrix} \begin{bmatrix} \hat{u} \\ \hat{v} \end{bmatrix} .$$

We can then try to look at the minimum backward error (in some sense; below we use least-squares); that is, find perturbations $c+\Delta c$, $d+\Delta d$, $a+\Delta a$, and $b+\Delta b$ such that

$$(\hat{u}+i\hat{v})\,(c+\Delta c+i(d+\Delta d)) = a+\Delta a+i(b+\Delta b)\,,$$

in a way that makes some norm of the vector $[\Delta a, \Delta b, \Delta c, \Delta d]$ as small as possible. When we use least-squares, a short computation gives

$$\Delta a = -\frac{r_1}{1+\rho^2},\ \ \Delta b = -\frac{r_2}{1+\rho^2},\ \ \Delta c = \frac{\hat{u}r_1+r_2\hat{v}}{1+\rho^2} \ \text{ and } \ \Delta d = \frac{\hat{u}r_2-r_1\hat{v}}{1+\rho^2}\,, \tag{4.48}$$

where ρ is the magnitude of the computed answer, $\rho = \sqrt{\hat{u}^2+\hat{v}^2}$. Since by the standard backward error results we have that the residual components r_1 and r_2 are of the order of $\varepsilon_M \|[a,b,c,d]\|_\infty$, this shows that the backward error of this approach is also small. For the condition number, see Problem 3.12. ◁

4.9 Relative Costs of the Different Factorings

We have already discussed the cost of various factorings and algorithms in the previous sections of this chapter by means of the flop count. As previously mentioned, the flop count does not fully reflect the actual computation time that an actual modern computer will use, although for MATLAB it does give a rough idea. In this section, we explain how to evaluate the relative costs of different factorings in a simple and practical way. The Schur factoring is discussed in Chap. 5. To begin, consider this code:

```
function [tlu,tqr,tschur,tsvd] = cost(n)
% [tlu,tqr,tschur,tsvd] = cost(n)
% avg time for five tries.
% Each routine called once before timing starts, to eliminate
    loading effects
a   = rand(n);
t0  = clock;
[u,s,v]  = svd(a);
[q,r]    = qr(a);
[u,t]    = schur(a);
[L,u,p]  = lu(a);
t = etime(clock,t0);
f=1; % retain dependence on n, expected to be O(n^3)
tlu=0;
tqr=0;
```

```
tschur=0;
tsvd=0;
for i = 1:5,
    a  = rand(n);
    t0 = clock;
    [L,u,p]  =  lu(a);
    tlu = tlu + etime(clock,t0);
    t0 = clock;
    [q,r] = qr(a);
    tqr = tqr + etime(clock,t0);
    t0 = clock;
    [u,t] = schur(a);
    tschur = tschur + etime(clock,t0);
    t0 = clock;
    [u,s,v] = svd(a);
    tsvd = tsvd + etime(clock,t0);
end;
tlu = tlu/5/f;
tqr = tqr/5/f;
tschur = tschur/5/f;
tsvd = tsvd/5/f;
```

It measures the computation time in MATLAB for four different factorings: LU, QR, Schur, and the SVD. We can then execute this code:

```
nstart = 11;
nmax = 50;
data = zeros(nmax-nstart+1,4);
 for n=nstart:nstart+nmax,
  [tlu,tqr,tschur,tsvd] = cost(5*n);
  data(n-nstart+1,1) = tlu;
  data(n-nstart+1,2) = tqr;
  data(n-nstart+1,3) = tschur;
  data(n-nstart+1,4) = tsvd;
end;
en = linspace( 5*nstart, 5*(nmax+nstart)+nmax, 6*nmax-5*nstart );
n = (nstart:nstart+nmax);
A = [ (5*n).^3', (5*n).^2', 5*n', ones(size(n))' ];
lufit = A\data(:,1);
qrfit = A\data(:,2);
schurfit = A\data(:,3);
svdfit = A\data(:,4);
lun = polyval( lufit, en );
qrn = polyval( qrfit, en );
schurn = polyval( schurfit, en );
svdn = polyval( svdfit, en );
figure(1), plot( en, lun, 'k-', 5*n, data(n-nstart+1,1), 'k+' )
figure(2), plot( en, qrn, 'k-', 5*n, data(n-nstart+1,2), 'k+' )
figure(3), plot( en, schurn, 'k-', 5*n, data(n-nstart+1,3), 'k+'
    )
figure(4), plot( en, svdn, 'k-', 5*n, data(n-nstart+1,4), 'k+' )
figure(5), semilogy( en, lun, 'k-', 5*n, data(n-nstart+1,1), 'k+'
    , ...
                   en, qrn, 'k--', 5*n, data(n-nstart+1,2), 'ko',
                        ...
```

```
                      en, schurn, 'k-.', 5*n, data(n-nstart+1,3), '
                          kx', ...
                      en, svdn, 'k:', 5*n, data(n-nstart+1,4), 'k.'
                          )...
                      , axis( [5*nstart,nstart+6*nmax,10^(-3)
                          ,2*10^(0)]);
set(gca,'fontsize',16);
xlabel('n','fontsize',16);
ylabel('cpu␣time␣(seconds)','fontsize',16);
```

On a 2009 tablet PC, we obtained the following results:

```
>> data(end,:)
ans =

    0.0166    0.0622    0.8326    0.2678
```

These numbers are the average number of seconds required to compute the LU factoring, the QR factoring, the Schur factoring, and the SVD of five random matrices of dimension 555×555, respectively. We observe that

$$\begin{aligned} \mathrm{SVD} &\approx 16 \cdot \mathrm{LU} \\ \mathrm{Schur} &\approx 50 \cdot \mathrm{LU} \\ \mathrm{QR} &\approx 3.7 \cdot \mathrm{LU}\,. \end{aligned}$$

This explains why the LU factoring is so often the method of choice: It is much the fastest. We also observe an $O(n^3)$ behavior, as theory says that we should. See Fig. 4.14. With this type of behavior in mind, we can estimate answers to questions

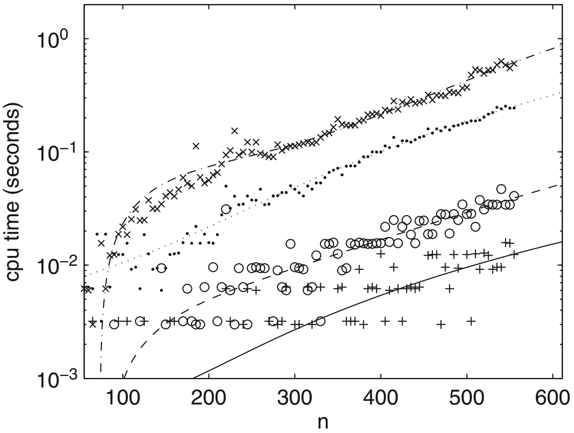


Fig. 4.14 Average timings for various factorings: LU (*solid line*, with + for data); QR (*dashed line*, with circles); SVD (*dotted line*, with . for data); and the Schur factoring (*dash-dot line*, with x's). The lines are least-squares fit to the data of polynomials cubic in the dimension n

of this kind: If LU decomposition takes 0.0166s for $n = 555$, how long would it take for $n = 600,000$? Evaluating our polynomial fit, we predict by executing

```
polyval( lufit, 600000 )/3600/24/7/4
```

that it will take 1.2813 (in units of four-week months). Or rather, it would be the case if the computer could even store the $600,000 \times 600,000$ matrix!

4.10 Solving Nonlinear Systems

At the end of Chap. 3, we showed how to transform the question of solving a nonlinear system of equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ into a *sequence* of linear systems of equations. We began with an initial guess $\mathbf{x}_0$, and linearized the problem about that point:

$$\mathbf{0} = \mathbf{F}(\mathbf{x}_0) + \mathbf{J}_\mathbf{F}(\mathbf{x}_0)\,(\mathbf{x} - \mathbf{x}_0) + \cdots \tag{4.49}$$

leading to the *linear* equations

$$\mathbf{J}_\mathbf{F}(\mathbf{x}_0)\Delta_x = -\mathbf{F}(\mathbf{x}_0)\,. \tag{4.50}$$

To solve this, the matrix $\mathbf{J}$ is then factored, often with a $\mathbf{PA} = \mathbf{LU}$ factoring, and the unknown vector Δ_x is solved for. We then put $\mathbf{x}_1 = \mathbf{x}_0 + \Delta_x$ and repeat the process. On every step of this process as laid out here we must evaluate the function $\mathbf{F}$, evaluate the Jacobian matrix $\mathbf{J}_\mathbf{F}(\mathbf{x}_k)$, and factor the matrix once evaluated (it is this cost that is often the largest on any step).

Notice also that at each stage the function $\mathbf{F}$ must be evaluated at the current estimate of the root. The size—the norm of the vector—of this function value tells us something about the backward error: $\mathbf{x}_k$ is the *exact* solution of the (possibly nearby) function $\mathbf{F}(\mathbf{x}) - \mathbf{F}(\mathbf{x}_k)$. It is up to the modeler to decide if this modified problem is near enough to the one whose solution was desired.

As usual, it is also possible to estimate the *forward* error in some cases, by looking at the norm of the Jacobian matrix $\mathbf{J}_\mathbf{F}(\mathbf{x}_k)$. Once the iteration has started to converge, the *next* iterate will be more correct than the current one: Comparing two successive iterates is equivalent to using the condition number estimate from the Jacobian to decide the accuracy of the earlier iterate.

Newton's method can be sped up, or made more reliable (sometimes both) by various modifications. Reusing a Jacobian matrix (and its factoring) for a few iterations can be helpful sometimes; this lowers the asymptotic convergence rate but also lowers the cost per step. One can choose a *damping* parameter $\mu < 1$ and update $\mathbf{x}_{k+1} = \mathbf{x}_k + \mu\Delta_x$; that is, we don't take the full Newton step but proceed cautiously. Again, this lowers the asymptotic convergence rate but can damp unwanted oscillations about the reference solution. There are many variants. For a thorough exploration of the current state-of-the-art, see Deuflhard (2011).

Example 4.16. Consider the Lorenz system, which is a system of three first-order differential equations:

$$\begin{aligned} \dot{x} &= yz - \beta x & x(0) &= a \\ \dot{y} &= \sigma(z-y) & y(0) &= b\,. \\ \dot{z} &= y(\rho - x) - z & z(0) &= c \end{aligned} \tag{4.51}$$

We use Saltzman's values of the parameters: $\sigma = 10, \rho = 28$, and $\beta = {}^{8}\!/_{3}$. We will take $a = 27$, $b = -8$, and $c = 8$ initially. The following nonlinear system of equations arises on using the implicit midpoint rule (to be studied in Chap. 13) for solving the Lorenz system:

$$\begin{aligned} x &= a + \Delta_t \left(\frac{(b+y)(c+z)}{4} - \frac{\beta(a+x)}{2} \right) \\ y &= b + \frac{\Delta_t \sigma\,(c+z-b-y)}{2} \\ z &= c + \Delta_t \left(\left(\frac{b}{2} + {}^{y}\!/_{2} \right) \left(\rho - \frac{a}{2} - \frac{x}{2} \right) - \frac{c}{2} - \frac{z}{2} \right) , \end{aligned} \tag{4.52}$$

where the unknowns to be solved for are (x, y, z), the constants a, b, and c are known (they are the midpoint rule approximations of the components at the current value of t), and the time-step Δ_t is also known during the solution process and presumed to be "small," although in practice we want to take it as large as possible. Moreover, we find that the Jacobian matrix is

$$\mathbf{J} = \begin{bmatrix} 1 + {}^{\Delta_t \beta}\!/_{2} & -{}^{\Delta_t (c+z)}\!/_{4} & -{}^{\Delta_t (b+y)}\!/_{4} \\ 0 & 1 + {}^{\Delta_t \sigma}\!/_{2} & -{}^{\Delta_t \sigma}\!/_{2} \\ {}^{\Delta_t (b+y)}\!/_{4} & -{}^{\Delta_t}\!/_{2} \left(\rho - {}^{a}\!/_{2} - {}^{x}\!/_{2} \right) & 1 + {}^{\Delta_t}\!/_{2} \end{bmatrix} . \tag{4.53}$$

As you can see, it depends not only on the parameters and on a, b, c, and Δ_t, but also on the (so far unknown) x, y, and z.

To get started, we need an initial guess. One could use an Euler method predictor, or a higher-order explicit predictor, but for the purposes of this example we will just use $x_0 = a$, $y_0 = b$, $z_0 = c$ (with the Saltzman values), and $\Delta_t = 0.05$. Then the value of $\mathbf{F}$ is

$$\begin{bmatrix} 6.8 \\ -8.0 \\ 0.80 \end{bmatrix} , \tag{4.54}$$

which isn't very small. Moreover, the value of the Jacobian at the initial guess is

$$\mathbf{J}_0 = \begin{bmatrix} 1.06666666666667 & -0.2 & 0.2 \\ 0 & 1.25 & -0.25 \\ -0.2 & -0.025 & 1.025 \end{bmatrix} , \tag{4.55}$$

and this matrix factors as follows:

$$\mathbf{J}_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0.0 & 1 & 0 \\ -0.1875 & -0.05 & 1 \end{bmatrix} \begin{bmatrix} 1.06666666666667 & -0.2 & 0.2 \\ 0.0 & 1.25 & -0.25 \\ 0.0 & 0.0 & 1.05 \end{bmatrix} . \tag{4.56}$$

Finally, the first change is

$$\Delta_0 = \begin{bmatrix} -4.93571428571436055 \\ 6.08095238095237888 \\ -1.59523809523810932 \end{bmatrix}$$

As a result, the next iterate $[x^{(1)}, y^{(1)}, z^{(1)}]$ is

$$\begin{bmatrix} 22.0642857142856386 \\ -1.91904761904762111 \\ 6.40476190476189089 \end{bmatrix} . \tag{4.57}$$

This is already a better solution (mostly because $h = 0.05$ is pretty small); its residual is

$$\mathbf{F}(\mathbf{x}_1) = \begin{bmatrix} 0.121257086167793737 \\ 8.88178419700125232 \times 10^{-16} \\ -0.375173044217692376 \end{bmatrix} . \tag{4.58}$$

But three more iterations get us to

$$\mathbf{x}_4 = \begin{bmatrix} 21.9216316009086648 \\ -1.84806892654072974 \\ 6.75965536729635108 \end{bmatrix} , \tag{4.59}$$

and the norm of the residual $\mathbf{F}(\mathbf{x}_4)$ is then $5 \cdot 10^{-16}$; then no further iterations are necessary.

We remark that in this example the Jacobian matrix was very well-conditioned so the linear system solving was very accurate. The *nonlinear* system is *also* very well-conditioned, and these two facts are related. We also remark that the nonlinear system of equations has more than just this one solution. In fact, there are three pairs (x, y) that solve these polynomial equations (we found these by an eigenvalue technique that will be sketched in Chap. 5):

$$\begin{bmatrix} 21.9216 & -1.8481 & 6.7597 \\ -380.1483 + 47.8430i & 6.5240 + 82.9117i & 48.6202 + 414.5585i \\ -380.1483 - 47.8430i & 6.5240 - 82.9117i & 48.6202 - 414.5585i \end{bmatrix} .$$

You see the solution found by Newton's method appears in the first line and also that there are in addition two complex roots. The solution found by Newton's method is indeed the desired solution for this application—the solution near the values of $x(t), y(t)$, and $z(t)$ at the previous value of t is what is wanted—but you can also see that in other cases there may be trouble if some of the unwanted solutions are too close to the initial guess; this happens especially if the Jacobian is singular or nearly singular, for example. Finally, note that this solution process must be carried out at every time-step of the method, so it seems important to worry about efficiency as well as stability. ◁

4.11 Notes and References

The notion of condition number of a linear system goes back at least to Alan Turing. However, the quantities Turing called "condition numbers" were each different to the 2-norm condition number favored here, while Von Neumann and Goldstine (1947) had already used exactly this, though they called it a "figure of merit." See Grcar (2011), and also Wilkinson (1971).

It isn't traditional to begin with the QR factoring; here, we follow Trefethen and Bau (1997), who make a persuasive case that the QR factoring is numerically simpler than Gaussian elimination (the LU factoring), being provably stable and much better understood. We left aside the method of Givens rotation; for details, consult Golub and van Loan (1996) or Higham (2002). A complex-valued Givens rotation is explored briefly in Problem 4.15. A beautiful paper on the development of Gram–Schmidt orthogonalization has been written by Leon et al. (2013).

Theorem 4.6 related to the SVD has been generalized to other norms by Kahan, who attributes the result to Gastinel. See Higham (2002).

In addition to the elimination methods discussed here, there is also "rook" pivoting, less expensive than complete pivoting, but more stable than partial pivoting. See Higham (2002). There is also Neville elimination, less expensive and less stable than partial pivoting, but which is useful for totally positive matrices.

Finally, our running example that we first introduce in Example 4.1 is taken from Nievergelt (1991).

Problems

Theory and Practice

4.1. Prove Theorem 4.1. (Hint: Fill in the discussion that follows it in the text.)

4.2. Implement Algorithm 4.2 in MATLAB.

4.3. Implement Algorithm 4.3 in MATLAB.

4.4. Implement Algorithms 4.4 and 4.5 and test your programs.

4.5. Suppose the upper-triangular matrix $\mathbf{U}$ is in addition *bidiagonal*:

$$\mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & & & \\ & u_{22} & u_{23} & & \\ & & u_{33} & \ddots & \\ & & & \ddots & u_{n-1,n} \\ & & & & u_{nn} \end{bmatrix}.$$

Write an algorithm to solve $\mathbf{Ux} = \mathbf{b}$ under the condition that each $u_{kk} \neq 0$, and then show that your algorithm costs about $2n$ flops.

4.6. Factoring structured matrices is often cheaper than factoring unstructured matrices. For example, tridiagonal or pentadiagonal matrices can be factored into banded lower and upper factors if no pivoting is used. Discuss.

4.7. Show that the cost, in flops, of solving the sequence of problems

$$\begin{aligned}\mathbf{Ly} &= \mathbf{Pb}\\ \mathbf{Ux} &= \mathbf{y}\,,\end{aligned}$$

where $\mathbf{P}$ is a permutation matrix that exchanges rows, $\mathbf{L}$ is unit lower-triangular, and $\mathbf{U}$ is upper-triangular, is almost the same as multiplying to get $\mathbf{x} = \mathbf{Zb}$, where $\mathbf{Z} = \mathbf{A}^{-1}$ is known and presumed dense, or full. The conclusion we want you to draw is that knowing $\mathbf{P}, \mathbf{L}, \mathbf{U}$ for which $\mathbf{PA} = \mathbf{LU}$ is, in some sense, just as useful as knowing $\mathbf{Z} = \mathbf{A}^{-1}$. (But not for all applications—sometimes you want $\mathbf{Z}$ itself.)

4.8. Show that the cost, in flops, of solving the sequence of problems

$$\begin{aligned}\mathbf{Qy} &= \mathbf{b}\\ \mathbf{Rx} &= \mathbf{y}\,,\end{aligned}$$

where $\mathbf{Q}$ is unitary and $\mathbf{R}$ is upper-triangular, is 50% more than that of problem 4.7. Note, however, that this is applied to rectangular systems. Compare also to the cost of solving $\mathbf{A}^H\mathbf{Ax} = \mathbf{A}^H\mathbf{b}$ if the Cholesky factors $\mathbf{LL}^H$ are known for $\mathbf{LL}^H = \mathbf{A}^H\mathbf{A}$.

4.9. Find by hand the singular values of

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}\,.$$

4.10. Show that the largest singular value of $\mathbf{A}$ is the 2-norm of $\mathbf{A}$. Recall that

$$\|\mathbf{A}\|_2 := \max_{\|\mathbf{x}\|\neq 0} \frac{\|\mathbf{Ax}\|_2}{\|\mathbf{x}\|_2}\,.$$

4.11. Consider the Pascal matrices (`pascal(n)` in MATLAB). How does their condition number grow with n? You may do this experimentally.

4.12. Choose $n = 100$. Choose a random complex vector of length n, and call it $\mathbf{x}$. Choose a random complex matrix of size $n \times n$, and call it $\mathbf{A}$. Solve the equation $\mathbf{b} = \mathbf{Ax}$ in MATLAB using the QR factoring, the SVD, and the $\mathbf{PA} = \mathbf{LU}$ factoring (for the latter, you may use the backslash operator). In each case, compute the residual of the computed solution and compute the condition number of the matrix $\mathbf{A}$. Since you know what $\mathbf{x}$ you were supposed to get, compare the forward error with the estimate obtained by the product of the condition number with the norm of the residual. Discuss your results.

4.13. This problem requires MAPLE or another computer algebra system. Take dimensions equal to the first 9 distinct Fibonacci numbers $1,2,3,\ldots,34$, and create random rational matrices of each dimension n. Compute the lengths of the exact rational determinants of these matrices and show that the lengths grow as a power of n.

4.14. By using a variation of Gaussian elimination where you add a multiple of the previous row, show that the determinant of the Vandermonde matrix

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ \tau_0 & \tau_1 & \tau_2 & \tau_3 & \tau_4 \\ {\tau_0}^2 & {\tau_1}^2 & {\tau_2}^2 & {\tau_3}^2 & {\tau_4}^2 \\ {\tau_0}^3 & {\tau_1}^3 & {\tau_2}^3 & {\tau_3}^3 & {\tau_4}^3 \\ {\tau_0}^4 & {\tau_1}^4 & {\tau_2}^4 & {\tau_3}^4 & {\tau_4}^4 \end{bmatrix}$$

is $\prod_{i>j}(\tau_i - \tau_j)$.

4.15. A complex-valued Givens rotation matrix $\mathbf{G}(i,j,\theta)$ is equal to the identity matrix except that four matrix elements are different:

$$\mathbf{G}(i,j,\theta) = \mathbf{I} + (c-1)\mathbf{e}_i\mathbf{e}_i^T - \bar{s}\mathbf{e}_i\mathbf{e}_j^T + s\mathbf{e}_j\mathbf{e}_i^T + (\bar{c}-1)\mathbf{e}_j\mathbf{e}_j^T\ ,$$

where $|c| = \cos\theta$ and $|s| = \sin\theta$. Show that $\mathbf{G}$ is unitary independent of the complex sign of either c or s. Show that c and s can be chosen to zero out the (i,j) entry of $\mathbf{GA}$. Show that a sequence of such transformations can factor $\mathbf{A}$ into a product of a unitary matrix $\mathbf{Q}$ and an upper-trapezoidal matrix $\mathbf{R}$.

4.16. Show that $\mathbf{HH}^H = \mathbf{I}$ so $\mathbf{H}$ is unitary if $\mathbf{H}$ is defined as in Eq. (4.20).

4.17. Consider Eq. (4.21). Show that

$$\mathbf{H} = \mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^H}{\mathbf{v}^H\mathbf{v}} \tag{4.60}$$

satisfies $\mathbf{Ha} = \alpha\mathbf{e}_1$ with $\alpha = -\text{signum}(a_1)\|\mathbf{a}\|_2$.

4.18. Show that $\mathbf{A}_{\mathbb{T}}$ as defined in Eq. (4.36) is an ellipsoid.

4.19. Show that the product of row-exchange matrices $(\mathbf{P}_{35}\mathbf{P}_{45})^2 \neq \mathbf{I}$.

4.20. The so-called normal equations for solving overspecified systems $\mathbf{Ax} = \mathbf{b}$ are $\mathbf{A}^H\mathbf{Ax} = \mathbf{A}^H\mathbf{b}$. Show that if we solve these equations, we minimize the 2-norm of the residual $\|\mathbf{b} - \mathbf{Ax}\|$. Use this method to solve the least-squares problem of Example 4.8.

4.21. Modify the `cost` program so that it uses complex random matrices, and run it. Comment on what happens. (Timing results on some machines can be quite different. The results on one machine surprised us. Perhaps yours will also be surprising.)

4.22. Fill in the details of the argument in Remark 4.1.

Investigations and Projects

4.23. In Example 4.12, we solved $\mathbf{A}\mathbf{x} = [0,1]'$ by QR factoring, and got (printing all digits this time)

```
x =
   -8.870855351158672e+005
    8.884184953489713e+005
```

and the computed residual was $\mathbf{r} = [0,1]' - \mathbf{A}\mathbf{x} = [0,0]'$. Using high precision, perhaps in MAPLE or in another CAS, say working to 32 digits, explicitly find a matrix $\Delta\mathbf{A}$ such that (for this $\mathbf{x}$ exactly!) $(\mathbf{A}+\Delta\mathbf{A})\,\mathbf{x} = [0,1]'$ and each entry of $\Delta\mathbf{A}$ is less than $\mu_M A_{i,j}$. One method is to try to minimize the sum of the squares of the entries of $\Delta\mathbf{A}$, using Lagrange multipliers to ensure the two constraints are satisfied; but the problem only asks for a "small enough" perturbation and other approaches will also work. Do it again but this time looking for a *symmetric* perturbation matrix $\Delta\mathbf{A}$.

4.24. Let $\|\cdot\|$ be any vector norm, for example, any p-norm

$$\|\mathbf{x}\|_p := \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p},$$

and let

$$\|\mathbf{A}\|_p := \max_{\mathbf{x}\neq 0} \frac{\|\mathbf{A}\mathbf{x}\|_p}{\|\mathbf{x}\|_p},$$

be the *subordinate* (or "induced") norm of the matrix $\mathbf{A}$.

1. Show that this matrix norm is submultiplicative, that is, that $\|\mathbf{AB}\| \le \|\mathbf{A}\|\,\|\mathbf{B}\|$.
2. Show that if $\mathbf{A}\mathbf{x} = \mathbf{b}$, where $\mathbf{A} \in \mathbb{C}^{n\times n}$ is nonsingular, and $\mathbf{x},\mathbf{b} \in \mathbb{C}^n$, and perturbing $\mathbf{b}$ gives
$$\mathbf{A}(\mathbf{x}+\Delta\mathbf{x}) = \mathbf{b}+\Delta\mathbf{b},$$
then
$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \le \kappa(\mathbf{A})\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}, \tag{4.61}$$
where $\kappa(\mathbf{A}) := \|\mathbf{A}\|\,\|\mathbf{A}^{-1}\|$. $\kappa(\mathbf{A})$ is called the *condition number* of $\mathbf{A}$.
3. Show that $\kappa(\mathbf{A}) \ge 1$, and that $\kappa_2(\mathbf{U}) = 1$ for unitary matrices $\mathbf{U}$ when the 2-norm is used: $\|\mathbf{A}\| = \|\mathbf{A}\|_2$.
4. Show that $\kappa(\mathbf{AB}) \le \kappa(\mathbf{A})\kappa(\mathbf{B})$, unlike the case of determinants where there is an equality sign. Give an example where inequality is strict.
5. Draw the unit "circles" $\|\mathbf{x}\|_\infty = 1$ and $\|\mathbf{x}\|_1 = 1$. Compare to the usual circle, $\|\mathbf{x}\|_2 = 1$.
6. Show that

$$\begin{aligned} \|\mathbf{A}\|_1 &= \max_j \sum_{i=1}^{n} |a_{ij}| \ \text{(i.e., maximum column sum)} \\ \|\mathbf{A}\|_\infty &= \max_i \sum_{j=1}^{n} |a_{ij}| \ \text{(i.e., maximum row sum)}. \end{aligned} \tag{4.62}$$

7. If $\mathbf{A} = [2,3;-1,2]$, compute each of $\|\mathbf{A}\|_1$, $\|\mathbf{A}\|_2$, $\|\mathbf{A}\|_F$, and $\|\mathbf{A}\|_\infty$. Show your work, but you may check your answers with MATLAB.
8. Explain in words the geometric meaning of Eq. (4.61), using the 2-norm and the singular value decomposition. You may suppose $\mathbf{b}$ and $\Delta\mathbf{b}$ are chosen diabolically, so as to induce the worst possible error $\Delta\mathbf{x}$, in relation to the size of $\|\mathbf{x}\|$.

4.25. Show that, in a linearized sense, the solution to

$$(\mathbf{A}+\Delta\mathbf{A})\,(\mathbf{x}+\Delta\mathbf{x}) = \mathbf{b} \tag{4.63}$$

differs from the solution to $\mathbf{Ax} = \mathbf{b}$ in norm by at most

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \kappa(\mathbf{A})\frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|} + O(\Delta^2)\,. \tag{4.64}$$

4.26. The determinant of a square matrix is justifiably famous for being useful in the theory of matrices. For example, it is used in the well-known theoretical algorithm known as Cramer's rule for the solution of nonsingular linear systems: If $\det\mathbf{A} \neq 0$, then the ith component of the solution of $\mathbf{Ax} = \mathbf{b}$ is

$$x_i = \frac{\det\left(\mathbf{A} \underset{i}{\leftarrow} \mathbf{b}\right)}{\det\mathbf{A}}\,,$$

where $\det\left(\mathbf{A} \underset{i}{\leftarrow} \mathbf{b}\right)$ means that we replace the ith column of $\mathbf{A}$ with the vector $\mathbf{b}$. Thus, the solution of the linear system has been reduced to the computation of determinants. But how does one compute determinants? Laplace expansion, as taught in a first linear algebra class, is combinatorially expensive—that is, the growth in cost is faster than exponential in the dimension n—and becomes hideously impractical even for modest n. That is, unless $\mathbf{A}$ has some very special structure that can be exploited.

A more or less practical method for computing the determinant is to factor the matrix, and use $\det(\mathbf{F}_1\mathbf{F}_2) = \det(\mathbf{F}_1)\det(\mathbf{F}_2)$. If the QR factoring is used, for example, then $\det\mathbf{A}$ is just $\det\mathbf{R}$, because $\det\mathbf{Q} = 1$. Moreover, since the determinant of an upper triangular matrix is the product of the diagonal entries, $\det\mathbf{R}$ is easily evaluated. LU factoring with partial pivoting is, of course, even cheaper. This being said, consider these problems:

1. Show that the cost of evaluating the determinant of a dense $n \times n$ matrix by using Laplace expansion is n times the cost of evaluating the determinant of an $(n-1) \times (n-1)$ matrix, plus some arithmetic to combine the results. Give a three-by-three example and count the multiplications.
2. Estimate the cost of solving $\mathbf{Ax} = \mathbf{b}$ using Cramer's rule, with QR factoring to evaluate the determinants. Compare with the cost of solving $\mathbf{Ax} = \mathbf{b}$ by direct use of the factoring.

3. As if the overly high cost were not already enough to discourage its use in general, Cramer's rule by Laplace expansion is also known not to be backward stable. For $n = 2$, it has reasonable forward accuracy (see Problem 1.9 in Higham (2002)), but not if $n \geq 3$. Find an example that shows instability. That is, find a 3×3 matrix $\mathbf{A}$ and a right-side $\mathbf{b}$ for which the solution computed in MATLAB by Cramer's rule using Laplace expansion for the determinants has a much worse residual than is expected from (say) solving the problem by QR factoring instead. Note that even for a 3×3 matrix there are several choices for Laplace expansion, so you can see that already analysis is difficult; and indeed no good error bounds are known. Recently, Habgood and Arel (2011) have found that the use of the so-called condensation (which seems to be just the Schur complement) can improve the cost of Cramer's rule, and seems to confer stability; but this is not really a "pure" Cramer's rule anymore.

4.27. The matrices

$$\mathbf{A}_0 = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \quad \mathbf{A}_1 = \begin{bmatrix} 1 & 0 & 1 \\ -1 & 1 & 0 \\ -1 & -1 & 1 \end{bmatrix} \quad \mathbf{A}_2 = \begin{bmatrix} 1 & & & 1 \\ -1 & 1 & & \\ -1 & -1 & 1 & \\ -1 & -1 & -1 & 1 \end{bmatrix}$$

and so on, which are lower-triangular except for a 1 in the northeast corner, have an interesting didactic role to play with LU factoring.

1. Show that $\mathbf{A}_k = \mathbf{L}_k\mathbf{U}_k$, where
$$\mathbf{L}_k\mathbf{U}_k = \begin{bmatrix} 1 & & & \\ -1 & 1 & & \\ -1 & -1 & 1 & \\ \vdots & \vdots & & \ddots \\ -1 & -1 & -1 & \cdots 1 \end{bmatrix} \begin{bmatrix} 1 & & & & 1 \\ & 1 & & & 1 \\ & & 1 & & 2 \\ & & & \ddots & \vdots \\ & & & & 2^k+1 \end{bmatrix}$$
and the final column of $\mathbf{U}_k$ is $[1, 1, 2, 4, 8, \ldots, 2^{k-1}, 2^k + 1]^T$.
2. Show that *both* $\mathbf{L}_k$ and $\mathbf{U}_k$ are ill-conditioned, that is, that $\kappa(\mathbf{L}_k) = O(2^k)$ and $\kappa(\mathbf{U}_k) = O(2^k)$. You may use any norm you like in $\kappa(\mathbf{A}) = \|\mathbf{A}\|_p\|\mathbf{A}^{-1}\|_p$ (clearly, both $\mathbf{L}_k$ and $\mathbf{U}_k$ are invertible).
3. Show that, in contrast, $\mathbf{A}_k$ is well-conditioned (you may show this experimentally if you like).
4. Since pivoting is not needed in this factoring, this example shows that the *algorithm* for solving $\mathbf{A}_k\mathbf{x} = \mathbf{b}$ given by Gaussian elimination with partial pivoting (solve $\mathbf{L}_k\mathbf{y} = \mathbf{b}$, then solve $\mathbf{U}_k\mathbf{x} = \mathbf{y}$) is *numerically unstable*. Discuss. Wilkinson claimed that matrices like these were almost never seen in practice, and indeed GEPP is in heavy use today:

   > There are rather special matrices for which the upper bound is attained, so that the final pivot is $2^{n-1}a$, but in practice such growth is very rare. (Wilkinson 1963 p. 97)

   Trefethen and Schreiber (1990) go some way toward explaining this.

4.28. The Gram–Schmidt orthogonalization procedure can be carried out on polynomials, not just vectors. Indeed, this is one way of generating orthogonal polynomials. With the Legendre–Sobolev inner product, defined as

$$\langle f,g\rangle := \int_{-1}^{1} f(t)\,g(t)\,dt + \mu\int_{-1}^{1} f'(t)\,g'(t)\,dt$$

for $\mu > 0$, and the initial polynomials $L_0 = 1/2$ and $L_1 = 3t/\sqrt{6+18\mu}$, find the first few orthogonal polynomials by using the modified Gram–Schmidt process. Be aware that using symbolic μ generates enormously long expressions, even for computer algebra systems. Is it easier using a vector of values for μ, or perhaps quasimatrices in Chebfun? See also Trefethen (2010). If using Chebfun, it may be helpful to orthogonalize starting from `chebpoly( i-1, [-1,1] )`, that is, the Chebyshev polynomials themselves. The interval used in this problem is different from the one used in Golubitsky and Watt (2010), to make the answers look simpler.

4.29. The Clement matrix family (Clement 1959) is one of the example families in MATLAB's gallery:

```
gallery('clement',6)
```

yields

$$\begin{bmatrix} 0 & 5 & & & & \\ 1 & 0 & 4 & & & \\ & 2 & 0 & 3 & & \\ & & 3 & 0 & 2 & \\ & & & 4 & 0 & 1 \\ & & & & 5 & 0 \end{bmatrix}.$$

These matrices have a number of interesting properties. With these integer entries, it is also known as the Kac matrix, after Mark Kac.

1. Show that the 6×6 Clement matrix

$$\begin{bmatrix} 0 & a_5 & & & & \\ a_1 & 0 & a_4 & & & \\ & a_2 & 0 & a_3 & & \\ & & a_3 & 0 & a_2 & \\ & & & a_4 & 0 & a_1 \\ & & & & a_5 & 0 \end{bmatrix} \tag{4.65}$$

   is nonsingular if the odd entries a_{2k+1} are nonzero.
2. Find the inverse (MAPLE or another CAS is helpful).
3. Show that if $a_{2k} = 1$ and $a_{2k-1} = 1/2$, then the $n\times n$ Clement matrix is ill-conditioned; that is, $\kappa(\mathbf{C}_n)$ grows exponentially with n.
4. Estimate the growth of $\kappa(\mathbf{C}_n)$ for the $n\times n$ Kac matrix, which has $n-1$, $n-2$, … on the superdiagonal and the same numbers in reverse order on the subdiagonal.

4.30. Let $\mathbf{A},\mathbf{B} \in \mathbb{C}^{m\times n}$. The *Hadamard product* or *elementwise product* $\mathbf{A}\circ\mathbf{B}$ is defined by

$$(\mathbf{A}\circ\mathbf{B})_{ij} = (\mathbf{A})_{ij}\cdot(\mathbf{B})_{ij} = a_{ij}b_{ij}. \tag{4.66}$$

The Hadamard product is what one obtains in MATLAB by typing `A.*B`.

1. Let $\mathbf{A},\mathbf{B},\mathbf{C} \in \mathbb{C}^{m\times n}$ and let $\alpha \in \mathbb{C}$. Show that

   a. It is commutative; that is, $\mathbf{A}\circ\mathbf{B} = \mathbf{B}\circ\mathbf{A}$.
   b. It is associative; that is, $\mathbf{A}\circ(\mathbf{B}\circ\mathbf{C}) = (\mathbf{A}\circ\mathbf{B})\circ\mathbf{C}$.
   c. It is distributive; that is, $\mathbf{A}\circ(\mathbf{B}+\mathbf{C}) = \mathbf{A}\circ\mathbf{B}+\mathbf{A}\circ\mathbf{C}$.
   d. $\mathbf{A}\circ(\lambda\mathbf{B}) = \alpha(\mathbf{A}\circ\mathbf{B})$.
   e. $\mathbb{C}^{m\times n}$ is closed under $\circ$.

2. What are the identity under $\circ$ and the inverse under $\circ$ (when it exists)?
3. Show that if $\mathbf{A},\mathbf{B}$ are diagonal, then $\mathbf{A}\circ\mathbf{B} = \mathbf{AB}$.
4. Show that if $\mathbf{A} \in \mathbb{C}^{n\times n}$ has an eigenvalue factoring $\mathbf{A} = \mathbf{X}\boldsymbol{\Lambda}\mathbf{X}^{-1}$ [so that $(\boldsymbol{\Lambda})_{ii} = \lambda_i$ for $1 \le i \le n$], and if we let $(\mathbf{A})_{ij} = a_{ij}$, then

$$\begin{bmatrix} a_{11} \\ a_{22} \\ \vdots \\ a_{nn} \end{bmatrix} = \left[\mathbf{X}\circ(\mathbf{X}^{-1})^T\right] \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{bmatrix} \tag{4.67}$$

   So, if you have the eigenvectors of $\mathbf{A}$, finding the eigenvalues only involves solving a system of linear equation of the form $\mathbf{Ax} = \mathbf{b}$.
5. Show that if $\mathbf{A} \in \mathbb{C}^{n\times n}$ has an SVD factoring $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H$ (so that $(\boldsymbol{\Sigma})_{ii} = \sigma_i$ for $1 \le i \le n$), and if we let $(\mathbf{A})_{ij} = a_{ij}$, then

$$\begin{bmatrix} a_{11} \\ a_{22} \\ \vdots \\ a_{nn} \end{bmatrix} = \left[\mathbf{U}\circ\overline{\mathbf{V}}\right] \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \vdots \\ \sigma_n \end{bmatrix} \tag{4.68}$$

So, if you have the eigenvectors of $\mathbf{AA}^H$, finding the singular values of $\mathbf{A}$ only involves solving a system of linear equation of the form $\mathbf{Ax} = \mathbf{b}$.

4.31. One extremely common use of Newton's method is to find zeros of the gradient ∇F, in attempting to optimize the scalar objective $F(x)$. This is often called Gauss–Newton iteration, and the Jacobian needed is actually the matrix of *second* derivatives of F, also called the Hessian matrix. This particular application is so important that it has attracted a significant amount of attention, and there is a great deal to be said about how to do this efficiently, reliably, and robustly. We will content ourselves here with a single problem, namely, that of fitting the nonlinear function

$$y = A\exp(-\lambda t) + B\exp(-\mu t)$$

to data; that is, the coefficients A and B that appear linearly are not known, but neither are the decay rates λ and μ.[26] This exponential fitting problem is well understood to be ill-conditioned, as you will see.

Suppose that we are given the data

$$y = [2.3751, 1.4591, 0.90357, 0.57181,\\ 0.35802, 0.23126, 0.14492, 0.099135]$$

attained at $t = 30$, 60, 90, $\ldots$, $240\,\mathrm{s}$. Try to find A, B, λ, and μ that minimize the objective function

$$O = \sum_{j=0}^{8} \left(y_j - Ae^{-\lambda t_j} - Be^{-\mu t_j} \right)^2 .$$

There are many ways to do this. Try an initial guess of $\lambda = 0.01$ and $\mu = 0.02$, together with a linear least-squares fit for A and B with those values of the decay rates. (Once the nonlinear parameters are guessed, then the linear parameters must then be the solution of the linear subproblem. Your first task is to find that A and B.)

After you have found your full nonlinear solution, try changing the initial guess to $\lambda = 0.015$ and $\mu = 0.025$, solve it again, and see if the answer changes much. If your second solution is different, which one is better? Now change the data a little bit and solve the problem yet again. Is this instance of this exponential fitting problem ill-conditioned?

4.32. We look again at the backward stability of the QR factoring.[27] Consider doing a QR factoring of a matrix for which we know the answer already, namely, one we build ourselves:

```
% Taken directly from Ray Spiteri's notes,
% http://www.cs.usask.ca/~spiteri/M313/notes/Lecture16.pdf
% which themselves were taken from Trefethen and Bau:
R=triu(randn(50)+1i*randn(50)); % Set R to a 50x50 upper-triangular matrix
% with normal random entries
[Q,X]=qr(randn(50)+1i*randn(50)); % Set Q to a 50x50 random orthogonal matrix
% by orthogonalizing a random matrix
A=Q*R; % Set A=QR, up to rounding errors
[Q2,R2]=qr(A); % Compute QR factorization by Householder
%------------------------------------
% Modified from here (RMC)
% Make sure the signs are the same even
% in the complex case (which this isn't now, but might be)
D1 = diag(diag(sign(R)));
D2 = diag(diag(sign(R2)));
% in each case Dbar * D = eye
```

[26] This particular optimization problem has a method all its own, by the way, called Prony's method (see the references in Giesbrecht et al. (2009)); but here we simply use Newton's method on the gradient.

[27] This problem owes its genesis to Trefethen and Bau (1997 Lecture 16).

```
% A = Q2*R2
% Good factoring initially
norm( A - Q2*R2, inf )
% A = Q2*Dbar*D*R2
Dbar = conj( D1*D2 );
D    = D1*D2;
Q3 = Q2*Dbar;
R3 = D*R2;
norm( A - Q3*R3, inf )
```

We test that we have adjusted MATLAB's output so that the signs match:

```
norm( A - Q2*R2, inf )
norm( A - Q3*R3, inf )
```

We find $2.6906 \cdot 10^{-14}$ and $2.6906 \cdot 10^{-14}$, respectively, which is fine. But now we look at the *forward errors*:

```
norm( Q - Q3, inf )
norm( R - R3, inf )
```

This time, we find $2.1400 \cdot 10^{-5}$ and $2.5755 \cdot 10^{-5}$, respectively. These are, as stated in the source, "*huge*." Yet we are able to compute the solution accurately to $\mathbf{Ax} = \mathbf{b}$ using these computed factorings, because of the backward error results that are available.

Write a one-paragraph summary of why the backward error theorems discussed in the text justify the accuracy of the solution to linear systems using Householder QR. You may also wish to comment on the perturbations in $\mathbf{A} + \Delta\mathbf{A} = (\mathbf{Q} + \Delta\mathbf{Q})(\mathbf{R} + \Delta\mathbf{R})$, which you can compute explicitly for this case.

Chapter 5
Solving Ax = λx

Abstract This chapter aims to introduce the reader to the numerical treatment of *eigenvalue problems*, that is, to the solution of the equation $\mathbf{Ax} = \lambda\mathbf{x}$. This chapter is shorter than the previous one, as it relies on many notions already introduced in the context of numerical linear algebra: factoring, backward error, condition number, and residual. We examine additional *factorings* relevant to eigenvalue problems, namely, the *Schur factoring* and the *Jordan canonical form*. We also outline algorithms to compute eigenvalues and eigenvectors, namely, the *power method* and the *QR algorithm*. Then, a *condition number of simple eigenvalues* is derived and, finally, the concept of *pseudospectrum* is introduced to further characterize the conditioning of eigenvalue problems. ◁

5.1 Generalized Eigenvalues

An eigenvalue λ of a square matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ is a complex number λ such that there exists a nonzero vector $\mathbf{x} \in \mathbb{C}^n$ for which

$$\mathbf{Ax} = \lambda\mathbf{x}\,. \tag{5.1}$$

The set $\Lambda(\mathbf{A})$ of eigenvalues is called the *spectrum* of $\mathbf{A}$. Also, the vector $\mathbf{x}$ satisfying this equation is called the *right eigenvector* of $\mathbf{A}$ corresponding to λ. It is easy to geometrically interpret the meaning of eigenvalues and eigenvectors of a matrix $\mathbf{A}$ from Eq. (5.1): A vector $\mathbf{x}$ is an eigenvector of $\mathbf{A}$ precisely when the vector $\mathbf{Ax}$ is in the direction of $\mathbf{x}$, where λ represents by how much the vector $\mathbf{Ax}$ is stretched in comparison to $\mathbf{x}$. It is important to remember that eigenvectors are not unique;

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_5,

indeed, if $\mathbf{x}$ is an eigenvector, so is $k\mathbf{x}$ for some constant k. However, if we normalize them using some norm, for instance, if we impose the condition $\|\mathbf{x}\|_2 = 1$ and also insist that the first nonzero component x_ℓ be strictly positive by dividing by $\exp(i\theta)$ =signum(x_ℓ), then we may speak of *the* eigenvector corresponding to an eigenvalue.

Similarly, it will be useful in what follows to introduce the concept of *left eigenvector*. A left eigenvector $\mathbf{y}^H$ (which is a row vector) corresponding to an eigenvalue λ is a vector such that

$$\mathbf{y}^H \mathbf{A} = \lambda \mathbf{y}^H . \tag{5.2}$$

The ordered set $(\mathbf{y}^H, \lambda, \mathbf{x})$ is called an *eigentriplet*. If we leave off $\mathbf{y}^H$, we have the *eigenpair* $(\lambda, \mathbf{x})$.

A *generalized* eigenvalue of a matrix pair $(\mathbf{A}, \mathbf{B})$ is a *pair* of numbers (α, β) that are not both zero and such that there exists a nonzero vector $\mathbf{x}$ satisfying

$$\alpha \mathbf{A}\mathbf{x} = \beta \mathbf{B}\mathbf{x} . \tag{5.3}$$

If $\mathbf{B} = \mathbf{I}$, then this reduces to the ordinary eigenvalue problem and $\lambda = {}^{\beta}\!/\!{}_{\alpha}$. It can happen in the generalized case when $\mathbf{B}$ is singular that one or more of the pairs (α, β) has $\alpha = 0$, in which case we say that the matrix pair has an infinite generalized eigenvalue (we will get lazy and drop the word "generalized" henceforth except when we wish to emphasize the distinction). It can also happen that $\det(z\mathbf{B} - \mathbf{A})$ is identically zero, in which case we say that the problem itself is singular. As is easily seen from the definition, if (α, β) is a generalized eigenvalue, then so is $(\mu\alpha, \mu\beta)$ for any nonzero μ. Again, the eigenvalues are usually normalized in some convenient fashion; for ease of interpretation in terms of standard eigenvalues, one often takes $\mu = {}^{1}\!/\!{}_{\alpha}$ if $\alpha \neq 0$, but it is often better to impose the condition $|\alpha|^2 + |\beta|^2 = 1$, $\alpha > 0$, or if $\alpha = 0$ then $\beta > 0$. Finally, the expression $\mathbf{A} - z\mathbf{B}$, for indeterminate z, is called a *matrix pencil*, for a rather obscure reason.[1] Because of the one-to-one correspondence between pencils and pairs, precise use of the terminology is not necessary.

Example 5.1. Eigenvalues are easily computed in MATLAB and MAPLE. Consider the matrix

$$\mathbf{A} = \begin{bmatrix} 1.0 + 1.0\,i & 0.50000 - 0.14286\,i & 0.33333 + 0.020408\,i \\ 0.50000 + 1.0\,i & 0.33333 + 0.090909\,i & 0.25000 + 0.0082645\,i \\ 0.33333 + 1.0\,i & 0.25000 + 0.23077\,i & 0.20000 + 0.053254\,i \end{bmatrix} .$$

Then, if we execute the command `eig(A)` in MATLAB or `Eigenvalues(A)` in MAPLE, we obtain the eigenvalues

$$\begin{bmatrix} 1.6019 + 1.1621i & -0.1189 - 0.0518i & 0.0504 + 0.0339i \end{bmatrix} .$$

[1] There *is* a reason, but we feel it takes longer to explain than it is worth.

Moreover, in MAPLE, the eigenvectors can be obtained by executing the command `Eigenvectors(A)`. ◁

Note that MATLAB's command `eig` can be used to compute generalized eigenvalues in a similar way. In fact, to find generalized eigenvalues as in Eq. (5.3), one simply executes the command with two arguments, that is, `eig(A,B)`. Let's look at an example with an infinite eigenvalue.

Example 5.2. Consider the matrices associated with MATLAB's commands `A = gallery('grcar',5)` and `B = gallery('clement',5)`, respectively,

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ -1 & 1 & 1 & 1 & 1 \\ 0 & -1 & 1 & 1 & 1 \\ 0 & 0 & -1 & 1 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} \text{ and } \mathbf{B} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 4 & 0 & 2 & 0 & 0 \\ 0 & 3 & 0 & 3 & 0 \\ 0 & 0 & 2 & 0 & 4 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} .$$

The matrix pair $(\mathbf{A},\mathbf{B})$ has a singular $\mathbf{B}$, and therefore at least one infinite eigenvalue. This can be seen by executing `svd(B)`, which returns

$$\begin{bmatrix} 4.8990 & 4.3589 & 4.0000 & 1.0000 & 0 \end{bmatrix} ,$$

showing that the last singular value of $\mathbf{B}$ is zero. Thus, as we expect, executing

```
eig(A,B)
```

shows that the last generalized eigenvalue is infinite:

$$\begin{bmatrix} 5.0874 & 1.1322 & -0.3946 & -0.8250 & \texttt{inf} \end{bmatrix}$$

A separate computation in MAPLE (which makes numerical sense since the entries of these example matrices are all integers) shows that the characteristic polynomial $\det(z\mathbf{B}-\mathbf{A})$ is the degree-4 polynomial $-8z^4+40z^3+12z^2-40z-15$, which has roots approximately $[-0.82502, -0.39457, 1.1322, 5.0874]$ computed via MAPLE's `fsolve` routine. Since the degree of the determinant is 4 although the matrices are 5×5, we know that the pencil has an infinite eigenvalue. The relative forward error in the finite eigenvalues computed by MATLAB is about 5×10^{-15}. ◁

In the next example, we show how to obtain not only the eigenvalues, but some useful factors, and how to verify how accurate the factoring is by computing its residual.

Example 5.3. The `gallery(3)` matrix from MATLAB is a well-known eigenvalue example matrix:

$$\mathbf{A} = \begin{bmatrix} -149 & -50 & -154 \\ 537 & 180 & 546 \\ -27 & -9 & -25 \end{bmatrix}$$

In MATLAB, we compute both eigenvalues and eigenvectors as follows:

```
A = gallery(3);
[V E] = eig(A)
```

These commands return

$$\mathbf{V} = \begin{bmatrix} 0.3162 & -0.4041 & -0.1391 \\ -0.9487 & 0.9091 & 0.9740 \\ -0.0000 & 0.1010 & -0.1789 \end{bmatrix} \quad \text{and} \quad \mathbf{E} = \begin{bmatrix} 1.0000 & 0 & 0 \\ 0 & 2.0000 & 0 \\ 0 & 0 & 3.0000 \end{bmatrix}.$$

The column $\mathbf{V}(:,k)$ of the matrix $\mathbf{V}$ returned by MATLAB is the eigenvector corresponding to the eigenvalue λ_k, which is given as the entry $(\mathbf{E})_{kk}$ of the matrix $\mathbf{E}$ returned by MATLAB.

The exact eigenvalues of this matrix are 1, 2, and 3, but they're a bit delicate to obtain (as we'll soon say, they're a bit ill-conditioned). For now, let $\mathbf{A} - \mathbf{VEV}^{-1}$ be the residual. Computing it in MATLAB, we find that it has norm about 10^{-10}, which seems surprisingly large given that the computation happened in double precision. ◁

Let us consider one more example exploring the relation between eigenvalues and generalized eigenvalues in a problem that may arise in applications.

Example 5.4. Linear vibration analysis in the undamped case studies differential equations of the form $\mathbf{M}\ddot{\mathbf{x}} + \mathbf{Kx} = 0$. Both $\mathbf{M}$ and $\mathbf{K}$ are symmetric and positive definite matrices (so long as no mass is zero and no spring constant is zero). Substitution of $\mathbf{x}(t) = \sin(\omega t)\mathbf{v}$ into the equation leads to the matrix equation

$$\left(\mathbf{K} - \omega^2\mathbf{M}\right)\mathbf{v} = 0\,.$$

Consider first

$$\mathbf{M} = \begin{bmatrix} 4 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 4 \end{bmatrix} \quad \text{and} \quad \mathbf{K} = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}.$$

We can convert this generalized eigenproblem into a standard eigenproblem as follows. To begin with, take the Cholesky factoring of $\mathbf{M} = \mathbf{LL}^T$, which is possible and reasonably stable since $\mathbf{M}$ is symmetric positive definite. Then, form the matrix

$$\mathbf{A} = \mathbf{L}^{-1}\mathbf{K}\left(\mathbf{L}^T\right)^{-1}.$$

It is easily verified that

$$\det(\mathbf{K} - \omega^2\mathbf{M}) = \det\mathbf{L}\det\left(\mathbf{A} - \omega^2\mathbf{I}\right)\det\mathbf{L}^T\,.$$

Thus, the generalized eigenvalues ω^2 of the original problem are standard eigenvalues of $\mathbf{A}$. Alternatively, we could have formed $\mathbf{B} = \mathbf{KM}^{-1}$, but this destroys the symmetry. We will see that symmetric matrices have perfectly conditioned eigenvalues, and so we want to preserve this property if we can.

To compute the value of ω in MATLAB, we first enter the matrices, and then we compute the Cholesky factoring (plus its residual to make sure that things are as expected):

```
M = [4 1 0; 1 4 1; 0 1 4];
K = [2 1 0; 1 2 1; 0 1 2;];
U = chol(M)
U'*U - M
```

This gives us, respectively,

$$\begin{bmatrix} 2.0000 & 0.5000 & 0 \\ 0 & 1.9365 & 0.5164 \\ 0 & 0 & 1.9322 \end{bmatrix} \qquad \text{and} \qquad 10^{-15}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -0.1110 \\ 0 & -0.1110 & 0 \end{bmatrix},$$

and so we see that the factoring has a small residual. We then form the matrix **A** and check its symmetry. Executing

```
A = inv(U')*K*inv(U)
A-A'
```

gives us, respectively,

$$\begin{bmatrix} 0.5000 & 0.1291 & -0.0345 \\ 0.1291 & 0.4333 & 0.1514 \\ -0.0345 & 0.1514 & 0.4238 \end{bmatrix} \qquad \text{and} \qquad 10^{-16}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -0.2776 \\ 0 & 0.2776 & 0 \end{bmatrix}.$$

We can then simply execute `eig(A)`, which returns the eigenvalues

$$\begin{bmatrix} 0.2265 \\ 0.5000 \\ 0.6306 \end{bmatrix}.$$

Their square roots are $[0.4760, 0.7071, 0.7941]^T$; those values indeed match those resulting from the direct computation of the generalized eigenvalues, using the command `eig(K,M)` and only displaying MATLAB's short format.

In this example, the cost of solving the original generalized eigenproblem is still very small. For larger systems, the cost of a generalized eigenproblem can be a modest factor (say, 5) times that of solving a standard eigenproblem; so, if one can make the transformation, one ought to do so.

Now, if the mass matrix **M** is singular, this transformation is not possible. For example, if the final entry is not 4 but rather $4/15$, then **M** is singular. Then, we can directly compute the generalized eigenvalues as follows:

```
M = [4 1 0; 1 4 1; 0 1 0.4e1 / 0.15e2;];
K = [2 1 0; 1 2 1; 0 1 2;];
eig( K, M )
ans.^(1/2)
```

The resulting values are 0.5338, 0.7691, and $\sqrt{2.8503 \cdot 10^{16}}$. This compares well with the true answers of $\sqrt{3471 \pm 89\sqrt{186}}/89$ and ∞. Of course, if we had instead performed the Cholesky factoring of **K** and put $\lambda = {}^1\!/\!_{\omega^2}$, we could have again used a standard eigenproblem. This simple example is intended to show you what can be done. ◁

As we see, the use of MATLAB or MAPLE to find eigenvalues and eigenvectors is straightforward. Formerly, eigenvalues and eigenvectors were considered difficult to compute; nowadays, unless the matrices are very large or have some other special property, the solution is pretty much routine, with the right software, although somewhat more expensive than, say, LU factoring. Nonetheless, there are important numerical subtleties to consider; in what follows, we will look at the numerical aspects of the study of eigenvalues, eigenvectors, and eigenspaces. The main purpose of this chapter is to help you to realize exactly the way in which these computations are routine: The methods pretty quickly give you the exact eigenvalues of slightly different problems, and you have to worry (a bit) about the conditioning of the eigenvalues (and worry a bit more about the conditioning of the eigenvectors, which tend to be quite a bit more sensitive).

5.2 Schur Factoring Versus Jordan Canonical Form

In this section, we examine and compare two factorings that can be used to solve problems involving eigenvalues: the Jordan canonical form and the Schur factoring. We will show that the first one might be problematic numerically and that the second is to be preferred. Let us begin the discussion with this theorem:

Theorem 5.1. *Every square matrix* **A** *can be brought by similarity transformation to Jordan canonical form (abbreviated as "JCF")* **J***; that is, we have the relation*

$$\mathbf{A} = \mathbf{X}\mathbf{J}\mathbf{X}^{-1}, \tag{5.4}$$

where the eigenvalues of **A** *are arranged in convenient blocks, called Jordan blocks, of the form*

$$\begin{bmatrix} \lambda & 1 & & & \\ & \lambda & 1 & & \\ & & \ddots & \ddots & \\ & & & \lambda & 1 \\ & & & & \lambda \end{bmatrix}$$

along the diagonal of **J**.

For a proof, see, for instance, Meyer (2001). There is a related form, the Weyr form, that has been receiving some attention at the time of this writing.

Remark 5.1. The Jordan form is, as a function of the entries of **A**, *discontinuous*, because the eigenvector's existence is discontinuous. The discontinuity of the JCF and of its generalization, the Krönecker canonical form, for matrix pencils, does not entirely preclude its numerical utility in some circumstances, as shown in Demmel and Kågström (1993a,b), but it does make life difficult.

Why is the discontinuity of the Jordan form as a function of the entries of $\mathbf{A}$ a problem? It is because a nonzero backward error, however small, possibly means an $O(1)$ forward error; that is, the condition number of the eigenvectors and hence the JCF may be infinite. ◁

To illustrate the last remark concretely, we examine a simple case.

Example 5.5. Consider the matrix

$$\mathbf{A}_\varepsilon = \begin{bmatrix} 1 & 1 \\ \varepsilon^2 & 1 \end{bmatrix} ,$$

whose eigenvalues are $1 \pm \varepsilon$. For $\varepsilon \neq 0$, the JCF is

$$\mathbf{J} = \begin{bmatrix} 1-\varepsilon & 0 \\ 0 & 1+\varepsilon \end{bmatrix} ,$$

for which $\lim_{\varepsilon \to 0} \mathbf{J} = \mathbf{I}$. However, the JCF of $\mathbf{A}_0$ is not such that $\mathbf{A_0} = \mathbf{I}$; rather, we have

$$\mathbf{J}_0 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \neq \lim_{\varepsilon \to 0} \begin{bmatrix} 1-\varepsilon & 0 \\ 0 & 1+\varepsilon \end{bmatrix} .$$

Note that there are two eigenvectors for all $\varepsilon > 0$, but there is only one for $\varepsilon = 0$. ◁

For most purposes, a factoring that is computationally superior to the Jordan form—that is, one that does not suffer the numerical difficulties that arise from discontinuity—is the Schur form, which uses unitary similarity.

Theorem 5.2. *Every square matrix* $\mathbf{A} \in \mathbb{C}^{n \times n}$ *can be brought by unitary similarity transformation to triangular form. The eigenvalues of* $\mathbf{A}$ *may appear in any order on the diagonal of the result. That is, there exist unitary* $\mathbf{U}$ *and upper-triangular* $\mathbf{T}$ *such that*

$$\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{U}^H . \tag{5.5}$$

The columns of $\mathbf{U}$ *are called Schur vectors, and Eq.* (5.5) *is called the Schur factoring.*

For a proof, see, for instance, Stewart (1998), who gives a proof by induction.

Remark 5.2. There is a "real-only" form for real matrices, which does not give a strict upper triangle if any roots are complex but rather two-by-two blocks that specify a complex conjugate pair of eigenvalues. This bookkeeping device improves speed for some applications but just gets in the way for our purpose; henceforth, we ignore the option (which is the default in MATLAB, and when we issue the command `schur` below, we must use the "complex" option to override the real form). ◁

For Hermitian matrices, the Schur factoring has an important property. If $\mathbf{A}^H = \mathbf{A}$, that is, $\mathbf{A}$ is Hermitian, then because

$$(\mathbf{UTU}^H)^H = \mathbf{UT}^H\mathbf{U}^H = \mathbf{A}^H = \mathbf{A} = \mathbf{UTU}^H\,,$$

we must have $\mathbf{T} = \mathbf{T}^H$, so that $\mathbf{T}$ is also Hermitian. But then $\mathbf{T}$ is diagonal (and not just upper-triangular) and, moreover, the diagonal entries satisfy $\bar{t}_{ii} = t_{ii}$ and hence are real.

What of the effect of perturbations of the entries of $\mathbf{A}$ on this factoring? Suppose that $\mathbf{A} = \mathbf{UTU}^H$, and consider

$$\mathbf{U}^H(\mathbf{A} + \Delta\mathbf{A})\mathbf{U} = \mathbf{T} + \mathbf{U}^H\Delta\mathbf{A}\mathbf{U}\,. \tag{5.6}$$

If the subdiagonal elements are small, and they will be if $\Delta\mathbf{A}$ is small, say $O(\varepsilon)$, because multiplication by unitary matrices does not amplify the size of entries much, then the eigenvalues of the result are going to be within $O(\varepsilon^{1/n})$ of those of $\mathbf{T}$—this is a standard eigenvalue perturbation result, and handles the case when all eigenvalues are equal (and therefore most sensitive to perturbation). In essence, multiplication by unitary matrices does not disturb the well-known result that eigenvalues are continuous functions of the matrix parameters, although they may be sensitive. But at least they are not discontinuous.

To be fair, the eigenvalues in the Jordan form are not discontinuous either—it is the eigenvectors that are the problem. For the Schur form, eigenvector computation is not involved, and the Schur vectors are better behaved because the matrices are unitary.

Example 5.6. Execute `A=rand(3)` in MATLAB to generate a random 3×3 matrix; in our case, it has generated the matrix

$$\mathbf{A} = \begin{bmatrix} 0.8147 & 0.9134 & 0.2785 \\ 0.9058 & 0.6324 & 0.5469 \\ 0.1270 & 0.0975 & 0.9575 \end{bmatrix}.$$

We then use the command

```
[U,T]=schur(A,'complex')
```

As stated previously, using the `complex` option so as to guarantee that we get a genuine upper-triangular matrix $\mathbf{T}$ results in the matrices

$$\mathbf{U} = \begin{bmatrix} 0.6752 & -0.7248 & -0.1368 \\ -0.7375 & -0.6604 & -0.1413 \\ -0.0120 & -0.1963 & 0.9805 \end{bmatrix} \quad \text{and} \quad \mathbf{T} = \begin{bmatrix} -0.1879 & 0.0326 & -0.2271 \\ 0 & 1.7527 & -0.4150 \\ 0 & 0 & 0.8399 \end{bmatrix}.$$

As it happened, the matrix had real eigenvalues and so the option wasn't needed—but running the command again with another random matrix as input might have produced a different result. The column vector $\mathbf{U}(:,1)$ is an eigenvector, but $\mathbf{U}(:,2)$ and $\mathbf{U}(:,3)$ are not; they are Schur vectors. The diagonal elements $d_T = [-0.1879, 1.7527, 0.8399]$ are eigenvalues.

Now, we compute a residual—in this case, the residual is $\mathbf{A} - \mathbf{UTU}^H$, which is a matrix—to assess the quality of this computation. By executing

```
A-U*T*U'
```

we find the matrix

$$10^{-14}\begin{bmatrix}0.0111 & 0.0999 & 0\\ 0.0444 & 0.1110 & 0.0111\\ 0.0250 & 0.0389 & -0.0333\end{bmatrix}.$$

We see that the residual is small. Strictly speaking, that residual may not be very accurate; inner products in *one* matrix–vector product can be considered (backward) exact, but it's trickier with matrix–matrix–matrix products. By executing

```
U*U'-eye(3)
```

to determine whether a loss of orthogonality happened, we find that

$$10^{-15}\begin{bmatrix}-0.1110 & 0.0035 & -0.1110\\ 0.0035 & -0.5551 & -0.0833\\ -0.1110 & -0.0833 & 0.2220\end{bmatrix}.$$

The 10^{-15} shows that the factor $\mathbf{U}$ is orthogonal up to a quantity of the order of roundoff error. ◁

The implementation of the Schur factoring in MATLAB does not always reflect the continuity of the Schur vectors. Choices are made about the possible signs of the Schur vectors, and these can differ when the input differs by trivial amounts. Let us consider an example.

Example 5.7. Consider the matrices

$$\mathbf{A}=\begin{bmatrix}1 & 1 & 0\\ 0 & 1 & 1\\ 0 & 0 & 1\end{bmatrix} \quad \text{and} \quad \mathbf{A}+\Delta\mathbf{A}=\begin{bmatrix}1 & 1 & 0\\ 0 & 1 & 1\\ \mu_M & 0 & 1\end{bmatrix}.$$

They are identical, to the exception of the unit roundoff μ_M entry in the lower-left corner. We can compute their respective Schur factorings; that is, we can compute the matrices $\mathbf{U}$ and $\mathbf{T}$ factoring $\mathbf{A}$ and the matrices $\mathbf{U}+\Delta\mathbf{U}$ and $\mathbf{T}+\Delta\mathbf{T}$ factoring $\mathbf{A}+\Delta\mathbf{A}$. By executing

```
[U,T] = schur(A,'complex')
[UDU,TDT] = schur(ADA,'complex')
```

we find the following matrices:

$$\mathbf{U}=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix} \qquad \mathbf{T}=\begin{bmatrix}1 & 1 & 0\\ 0 & 1 & 1\\ 0 & 0 & 1\end{bmatrix}$$

$$\mathbf{U}+\Delta\mathbf{U}=\begin{bmatrix}1 & 0 & 0\\ 0 & -1 & 0\\ 0 & 0 & 1\end{bmatrix} \qquad \mathbf{T}+\Delta\mathbf{T}=\begin{bmatrix}1 & -1 & 0\\ 0 & 1 & -1\\ 0 & 0 & 1\end{bmatrix}.$$

Notice that while the eigenvalues on the diagonal of $\mathbf{T}$ are identical to those of $\mathbf{T} + \Delta\mathbf{T}$, the Schur vectors have differing signs. This will occasionally require bookkeeping. $\triangleleft$

In the next section, after some preamble, we introduce the QR algorithm to compute the Schur factoring. This algorithm computes $\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{U}^H$ and tries to guarantee that the computed matrices $\hat{\mathbf{T}}$ and $\hat{\mathbf{U}}$ are such that $(\mathbf{A}+\mathbf{E}) = \mathbf{U}\hat{\mathbf{T}}\mathbf{U}^H$ for some $\mathbf{U}$ near $\hat{\mathbf{U}}$ and $\|\mathbf{E}\|_F \leq c_n \mu_M \|\mathbf{A}\|_F$. The algorithm also ensures that the computed $\mathbf{U}$ is nearly unitary; that is, $\|\hat{\mathbf{U}}^H \hat{\mathbf{U}} - \mathbf{I}\| = O(\mu_M)$.

5.3 Algorithms for Eigenvalue Problems

In this section, we examine the algorithms used to compute eigenvalues and eigenvectors. The objective is not to provide a detailed analysis of multiple algorithms, but rather to simply outline the main ideas that come into the best algorithms that are used, for instance, to compute the Schur factoring.

5.3.1 Simple Iterative Methods

We begin with a very simple method that can be used to find approximate eigenvalues. As we will see, more refined algorithms build on its core idea. Suppose that a square matrix $\mathbf{A}$ has distinct eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ and, moreover, that λ_1 is strictly the largest one. How can we find λ_1? As we know, the defining equation of eigenvalue problems is $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$. Thus, if the jth entry of the eigenvector $\mathbf{x}$ is x_j and is nonzero, then

$$\lambda = \frac{(\mathbf{A}\mathbf{x})_j}{x_j}.$$

This suggests that if we find an eigenvector first, then we will have our eigenvalue. How might we do that? We first describe an amazingly simple algorithm known as the *power method* or *power iteration*.

To begin, take some random unit vector as an initial guess for $\mathbf{x}$ and call it $\mathbf{x}_0$. Then, let $\mathbf{x}_1 = \mathbf{A}\mathbf{x}_0$, $\mathbf{x}_2 = \mathbf{A}\mathbf{x}_1$, and so on. The hope is that this sequence of iterates converges to the eigenvector corresponding to λ_1; and so it does, in certain circumstances. We also take care to normalize each iterate $\mathbf{x}_k$ in order to avoid overflow and underflow (also, by normalizing, the iterates will form a sequence on the unit sphere). That is, we let $\mathbf{y}_k = \mathbf{A}\mathbf{x}_{k-1}$ and

$$\mathbf{x}_k = \frac{\mathbf{y}_k}{\|\mathbf{y}_k\|},$$

so that each $\mathbf{x}_k$ is a unit vector.

Example 5.8. Consider the matrix

$$\mathbf{A} = \begin{bmatrix} -2 & 1 & \\ 1 & -2 & 1 \\ & 1 & -2 \end{bmatrix}.$$

Take $\mathbf{x}_0 = [1,0,0]^T$ (which isn't very random) as the initial guess. Then our first iteration results in

$$\mathbf{y}_1 = \begin{bmatrix} -2 & 1 & \\ 1 & -2 & 1 \\ & 1 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -2 \\ 1 \\ 0 \end{bmatrix} \quad \text{and so} \quad \mathbf{x}_1 = \begin{bmatrix} {}^{-2}\!/\!_{\sqrt{5}} \\ {}^{1}\!/\!_{\sqrt{5}} \\ 0 \end{bmatrix} \doteq \begin{bmatrix} -0.8944 \\ 0.4472 \\ 0 \end{bmatrix}.$$

Continuing our computation with the MATLAB commands

```
%% Power iteration
A = [-2, 1, 0; 1,-2,1;0,1,-2]
x=[1;0;0]
for i=1:19,
    y = A*x
    x = y/norm(y,2)
end
```

we eventually find

$$\mathbf{x}_9 = \begin{bmatrix} -0.5081 \\ 0.7071 \\ -0.4918 \end{bmatrix} \quad \text{and} \quad \mathbf{x}_{10} = \begin{bmatrix} 0.5647 \\ -0.7071 \\ 0.4952 \end{bmatrix},$$

which looks promising. If we look further, we find

$$\mathbf{x}_{18} = \begin{bmatrix} -0.5001 \\ 0.7071 \\ -0.4999 \end{bmatrix} \quad \text{and} \quad \mathbf{x}_{19} = \begin{bmatrix} 0.5000 \\ -0.7071 \\ 0.5000 \end{bmatrix},$$

which appears to have converged. (The sign changes at each iteration, but we could normalize the $\mathbf{y}_k$s differently to account for the direction of the eigenvector.) Indeed, we can verify that this result is correct:

$$\begin{bmatrix} -2 & 1 & \\ 1 & -2 & 1 \\ & 1 & -2 \end{bmatrix} \begin{bmatrix} {}^{-1}\!/\!_{2} \\ {}^{1}\!/\!_{\sqrt{2}} \\ {}^{-1}\!/\!_{2} \end{bmatrix} = \begin{bmatrix} 1 + {}^{1}\!/\!_{\sqrt{2}} \\ -\sqrt{2} - 1 \\ {}^{1}\!/\!_{\sqrt{2}} + 1 \end{bmatrix} = \lambda \begin{bmatrix} -{}^{1}\!/\!_{2} \\ {}^{1}\!/\!_{\sqrt{2}} \\ -{}^{1}\!/\!_{2} \end{bmatrix}.$$

From this, we find that

$$\lambda_1 = \frac{1 + {}^{1}\!/\!_{\sqrt{2}}}{{}^{-1}\!/\!_{2}} = -2 - \sqrt{2} \doteq -3.414$$

is the largest, dominant eigenvalue. ◁

Why did this simple iterative method work? First of all, it is because, as we supposed, the matrix $\mathbf{A}$ had distinct eigenvalues and that one of them, λ_1, was strictly largest. When a matrix such as $\mathbf{A}$ has distinct eigenvalues, then its eigenvectors form a basis for $\mathbb{C}^n$. That is, every vector $\mathbf{x}_0$ can be expressed as a linear combination of the eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$:

$$\mathbf{x}_0 = \sum_{k=1}^{n} \alpha_k \mathbf{v}_k .$$

We know neither the coefficients α_k nor the eigenvectors $\mathbf{v}_k$ at this point, but we know that they exist and are unique. Then we can rewrite our first iterate $\mathbf{x}_1$ as follows:

$$\mathbf{x}_1 = \mathbf{A}\mathbf{x}_0 = \mathbf{A} \sum_{k=1}^{n} \alpha_k \mathbf{v}_k = \sum_{k=1}^{n} \alpha_k \mathbf{A}\mathbf{v}_k = \sum_{k=1}^{n} \alpha_k \lambda_k \mathbf{v}_k \, .$$

Similarly, the second iterate can be written as $\mathbf{x}_2 = \sum_{k=1}^{n} \alpha_k \lambda_k^2 \mathbf{v}_k$. It is then a simple matter to show by induction that all the iterates x_m can be written as

$$\mathbf{x}_m = \sum_{k=1}^{n} \alpha_k \lambda_k^m \mathbf{v}_k \, .$$

From this expression, it is easy to see why the iterative method works: Since λ_1 is larger than all other eigenvalues, λ_1^m grows faster than all others; that is, we have

$$\mathbf{x}_m = \lambda_1^m \alpha_1 \mathbf{v}_1 + \sum_{k=2}^{n} \alpha_k \lambda_k^m \mathbf{v}_k = \lambda_1^m \alpha_1 \mathbf{v}_1 \left(1 + O\left(\left(\frac{\lambda_2}{\lambda_1} \right)^m \right) \right) ,$$

where λ_2 is the next-largest eigenvalue. Thus, the iteration picks out the eigenvector $\mathbf{v}_1$ corresponding to the largest eigenvalue λ_1.

Under the same assumptions, a similar iterative method can be used to find other eigenvalues λ_k with $k \neq 1$. This method, known as the *inverse-power iteration*, is investigated in Problem 5.16. The underlying idea for this method is that if $\lambda_1, \lambda_2, \ldots, \lambda_n$ are the eigenvalues of $\mathbf{A}$, then $(\lambda_1 - \mu)^{-1}, (\lambda_2 - \mu)^{-1}, \ldots, (\lambda_n - \mu)^{-1}$ are the eigenvalues of $(\mathbf{A} - \mu\mathbf{I})^{-1}$. Moreover, by an argument similar to that for the simple power iteration above, the dominant eigenvalue of $(\mathbf{A} - \mu\mathbf{I})^{-1}$ will be the eigenvalue of $\mathbf{A}$ closest to μ. So, with an estimate of an eigenvalue, we can choose a μ to find a refined computed value.

However, as usual, we don't want to actually compute inverse matrices. Instead, we start from a random vector $\mathbf{x}_0$, normalize it to get $\mathbf{z}_0 = \mathbf{x}_0 / \|\mathbf{x}_0\|$, and then solve

$$(\mathbf{A} - \mu\mathbf{I})\mathbf{x} = \mathbf{z}_0$$

for $\mathbf{x}$. Then we let $\mathbf{z}_1 = \mathbf{x}$, and repeat the process.

Example 5.9. We continue Example 5.8, but this time we will use inverse-power iteration to find the smallest magnitude eigenvalue λ_3. For this matrix, we can use the shift value $\mu = 0$ and simply execute this code:

```
%% Inverse power iteration
A = [-2, 1, 0; 1,-2,1;0,1,-2]
x=[1;0;0]
mu=0
for i=1:10,
    y = (A-mu*eye(3))\x
    x = y/norm(y,2)
end
```

After 10 iterations only, we obtain the eigenvector

$$\mathbf{x} = \begin{bmatrix} 0.5000 \\ 0.7071 \\ 0.5000 \end{bmatrix},$$

which seems to be correct to four digits (i.e., MATLAB's short format). From it, we find that the smallest eigenvalue is $\lambda_3 = (\mathbf{Ax})_1/x_1 = -0.5858$. By comparing the result with MATLAB's built-in `eig` function, we find that the forward error is of the expected order; namely, it is $-1.3204 \cdot 10^{-5}$. ◁

For the rest of this section, we will not look at all the theory of convergence and stability for such algorithms. Instead, we turn immediately to the go-to algorithm for eigenvalue problems.

5.3.2 The QR Algorithm for $\mathbf{Ax} = \lambda\mathbf{x}$

In this section, we convey the main idea of the QR algorithm for computing eigenvalues. This algorithm is almost magical in the simplicity of its basic formulation and is one of the most highly valued numerical algorithms of the 20th century. Its key step is connected with the inverse-power iteration. Jumping ahead a bit, we are trying to find a unitary similarity $\mathbf{Q}$ transform that changes

$$\mathbf{A} = \begin{bmatrix} \mathbf{B} & \mathbf{h} \\ \mathbf{g}^H & \mu \end{bmatrix} \quad \text{to} \quad \begin{bmatrix} \hat{\mathbf{B}} & \hat{\mathbf{h}} \\ \hat{\mathbf{g}}^H & \hat{\mu} \end{bmatrix} \tag{5.7}$$

in a way that makes $\|\hat{\mathbf{g}}^H\|$ *smaller* than $\|\mathbf{g}^H\|$. By the Gershgorin circle theorem, it will follow that $\hat{\mu}$ is a better approximate eigenvalue than the original μ was. Moreover, as Problem 5.9 asks you to show, if we can make $\hat{\mathbf{g}}^H = 0$, then we will have found an eigenvalue, namely, $\hat{\mu}$ (to show that the ideal choice for $\mathbf{Q}$ in $\mathbf{Q}^H\mathbf{AQ}$ is $\mathbf{Q} = [\mathbf{Q}_1, \mathbf{q}]$, where $\mathbf{q}$ is the eigenvector associated with λ).

This being said, we know neither $\mathbf{q}$ nor λ: That's what we're trying to find out. We instead try to "bootstrap": If $\mathbf{g}^H$ is already small in norm, then the standard basis vector $\mathbf{e}_n^H$ is an approximate eigenvector, though not good enough of one for the transform to make $\mathbf{g}^H$ smaller (try it!). But we can improve $\mathbf{e}_n^H$ by one step of the inverse-power iteration:

1. Solve $\mathbf{q}^H(\mathbf{A}-s\mathbf{I}) = \mathbf{e}_n^H$ for $\mathbf{q}$
2. Let $\mathbf{q} = \mathbf{q}/\|\mathbf{q}\|$.

Here the shift s might be the Rayleigh quotient $\mathbf{e}_n^H\mathbf{A}\mathbf{e}_n$, or we might choose another; a better choice will mean faster convergence.

Now comes the real magic of the QR algorithm. Using the QR factoring, we factor the matrix $\mathbf{A}-s\mathbf{I}$:

$$\mathbf{A}-s\mathbf{I} = \mathbf{QR}\,.$$

The final column of this $\mathbf{Q}$ is exactly the $\mathbf{q}$ that arises on the step of the inverse-power iteration. To see this, note $\mathbf{Q}^H = \mathbf{R}(\mathbf{A}-s\mathbf{I})^{-1}$ and, as a result,

$$\mathbf{e}_n^H\mathbf{Q}^H = \mathbf{q}^H = r_{nn}\mathbf{e}_n^H(\mathbf{A}-s\mathbf{I})^{-1}\,. \tag{5.8}$$

Moreover, $\|\mathbf{q}^H\| = 1$. But another way to express the inverse iteration $\mathbf{q}$ is

$$\mathbf{q}^H = \frac{\mathbf{e}_n^H(\mathbf{A}-s\mathbf{I})^{-1}}{\|\mathbf{e}_n^H(\mathbf{A}-s\mathbf{I})^{-1}\|}, \tag{5.9}$$

and these are now seen to be the same. Finally, we wish to form the similarity transformation $\mathbf{Q}^H\mathbf{AQ}$. Since we have

$$\mathbf{RQ} = \mathbf{Q}^H(\mathbf{A}-s\mathbf{I})\mathbf{Q} = \mathbf{Q}^H\mathbf{AQ} - s\mathbf{I}\,,$$

we find that $\mathbf{RQ}+s\mathbf{I} = \mathbf{Q}^H\mathbf{AQ}$ is what we want. We thus reach the heart of the QR algorithm:

1. $\mathbf{A}-s\mathbf{I} = \mathbf{QR}$ (shift and factor)
2. $\mathbf{A}_1 = \mathbf{RQ}+s\mathbf{I}$ (reverse the factors and put back the shift).

Let's look at an example to grasp how it works more concretely.

Example 5.10. In this example, we carry out three explicit steps of this process for this small matrix randomly generated:

$$\mathbf{A} = \begin{bmatrix} 0.9649 & 0.9572 & 0.1419 \\ 0.1576 & 0.4854 & 0.4218 \\ 0.9706 & 0.8003 & 0.9157 \end{bmatrix}.$$

We take a small complex shift to begin with (this also works for real eigenvalues, but a real shift won't help find a complex eigenvalue). The first step of the iteration is obtained by executing

```
s = 0.5i;
[Q,R] = qr( A - s*eye(3) );
A1 = R*Q + s*eye(3)
```

which produces the matrix

$$\mathbf{A}_1 = \begin{bmatrix} 1.5888-0.1579i & 0.8763-0.4759i & -0.5453+0.2804i \\ 0.3077+0.3348i & 0.4831-0.1208i & 0.1224+0.1884i \\ 0.3461-0.0000i & -0.0019-0.1338i & 0.2941+0.2787i \end{bmatrix}.$$

This does not look like it has made progress, but we now take the corner element $0.2941+0.2787i$ as our approximate eigenvalue and shift by that amount. We then take a second step:

```
s = A1(3,3);
[Q,R] = qr( A1 - s*eye(3) );
A2 = R*Q + s*eye(3)
```

From the newly produced matrix

$$\mathbf{A}_2 = \begin{bmatrix} 1.7227-0.0076i & -0.3520+0.5627i & -0.6601+0.5240i \\ -0.0466-0.1935i & 0.3711-0.2913i & 0.3187-0.2840i \\ -0.0076+0.0000i & -0.0107-0.0032i & 0.2722+0.2989i \end{bmatrix},$$

we already see progress, since the elements in the last row are concentrated on the diagonal. Once again, we take the corner element $0.2722+0.2989i$ as our approximate eigenvalue and shift and iterate one more time:

```
s = A2(3,3);
[Q,R] = qr( A2 - s*eye(3) );
A3 = R*Q + s*eye(3)
```

This time, we find the matrix

$$\mathbf{A}_3 = \begin{bmatrix} 1.8036+0.0241i & 0.3029+0.3767i & -0.7026-0.4352i \\ -0.0216-0.0827i & 0.2865-0.3301i & -0.0238+0.4331i \\ -0.0000+0.0000i & -0.0002-0.0000i & 0.2758+0.3061i \end{bmatrix},$$

and convergence is now obvious. We have one eigenvalue to approximately four places. In comparison, the result of `eig` is $[1.8146, 0.2757+0.3061i, 0.2757-0.3061i]$. ◁

There are some important details left out: converting **A** into Hessenberg form for efficiency, deciding when to stop iterating, managing deflation in case the problem becomes reducible, determining how best to pick the shifts, and how to do it efficiently in real arithmetic. There are important theoretical considerations, too: No global convergence theorem is known, although in practice it works very well, in $O(n^3)$ flops. The stability can always be checked afterward by forming a posteriori the residual matrix $\mathbf{R} = \mathbf{QTQ}^H - \mathbf{A}$ and checking that it satisfies $\|\mathbf{R}\| = O(\mu_M)$—as it should, so that the computed eigenvalues are the eigenvalues of a nearby matrix $\mathbf{A}+\hat{\mathbf{R}}$ (thus giving computed elements in what we will call the pseudospectrum of **A** in Sect. 5.5). Note that one cannot have an eigenvalue without (at least one) eigenvector; therefore, the computed eigenvectors will also be exact eigenvectors of nearby matrices. But in the end, the QR algorithm for finding the Schur factoring is one of the most efficient and stable algorithms going—one of the jewels in the crown of numerical analysis.

The algorithm used to solve *generalized* eigenvalue problems is similar; it is called the QZ algorithm. A generalization of the Schur factoring is available: Using two unitary matrices $\mathbf{U}$ and $\mathbf{Z}$, we can bring the matrix pair simultaneously to upper-triangular form. Then, from the diagonal entries of each triangular matrix, the generalized eigenvalues can be read off. The algorithm tries to guarantee that

$$\mathbf{Q}_0^H(\mathbf{A}+\mathbf{E})\mathbf{Z}_0 = \hat{\mathbf{R}}$$
$$\mathbf{Q}_0^H(\mathbf{B}+\mathbf{F})\mathbf{Z}_0 = \hat{\mathbf{S}}\,,$$

where $\mathbf{Q}_0$, $\mathbf{Z}_0$ are unitary and $\|\mathbf{E}\| = O(\mu_M\|\mathbf{A}\|)$ and $\|\mathbf{F}\| = O(\mu_M\|\mathbf{B}\|)$. The cost of the QZ algorithm is approximately $15n^3$ flops to compute $\hat{\mathbf{R}}$ and $\hat{\mathbf{S}}$, plus an additional $8n^3$ to compute $\mathbf{Q}$ and $10n^3$ to compute $\mathbf{Z}$.

5.4 Condition Number of a Simple Eigenvalue

In the preceding sections, we have once again interpreted computation errors as backward errors, similar to data errors. What, then, are the consequences of these errors? We find an answer to this question by determining the conditioning of eigenproblems. Let us consider the case of a simple (nonrepeated) eigenvalue.

Suppose that $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$, where $\mathbf{x}$ is a right eigenvector. Also suppose that $\mathbf{y}^H\mathbf{A} = \lambda\mathbf{y}^H$, where $\mathbf{y}$ is a left eigenvector. Consider the perturbed system

$$(\mathbf{A}+\mathbf{E})(\mathbf{x}+\Delta\mathbf{x}) = (\lambda+\Delta\lambda)(\mathbf{x}+\Delta\mathbf{x})\,,$$

and simplify this by ignoring quadratically small terms:

$$\mathbf{A}\Delta\mathbf{x}+\mathbf{E}\mathbf{x}+\cancel{\mathbf{E}\Delta\mathbf{x}} = \lambda\Delta\mathbf{x}+\Delta\lambda\mathbf{x}+\cancel{\Delta\lambda\Delta\mathbf{x}}.$$

We multiply by $\mathbf{y}^H$, so that

$$\cancel{\mathbf{y}^H\mathbf{A}\Delta\mathbf{x}}+\mathbf{y}^H\mathbf{E}\mathbf{x} = \cancel{\lambda\mathbf{y}^H\Delta\mathbf{x}}+\Delta\lambda\mathbf{y}^H\mathbf{x}.$$

We require that $\mathbf{y}^H\mathbf{x} \neq 0$; in the exercises, you will prove that $\mathbf{y}^H\mathbf{x} \neq 0$ for a simple eigenvalue. As a result, we find that

$$\Delta\lambda = \frac{\mathbf{y}^H\mathbf{E}\mathbf{x}}{\mathbf{y}^H\mathbf{x}}\,. \tag{5.10}$$

This says that if $\mathbf{y}^H\mathbf{x}$ is small relative to $\mathbf{y}^H\mathbf{E}\mathbf{x}$, then $\Delta\lambda$ will be large. This can happen. Data errors may make λ inaccurate. In Problem 5.4, you will show that $(\mathbf{y}^H\mathbf{x})^{-1}$ can indeed be used as an absolute condition number for a simple eigenvalue.

Multiple eigenvalues are ill-conditioned: A tiny change in the data of size ε can cause a change in the eigenvalues of $O(\varepsilon^{1/m})$, where m is the multiplicity of the eigenvalue. The problem of ill-conditioned *eigenvectors* is worse. Eigenvectors can fail to exist when eigenvalues are multiple, so the problem of computing them is ill-posed (and thus ill-conditioned). Even when eigenvalues are simple, eigenvalues can be close together; and this closeness strongly affects the conditioning of the eigenvectors. See Hogben (2006 chapter 43).

The conditioning of eigenvalues is crudely measured by the (ordinary) condition number of the matrix of eigenvectors.

Example 5.11. Consider the matrix

$$\mathbf{A} = \begin{bmatrix} 4-\varepsilon^2 & 2-2\varepsilon \\ -2-2\varepsilon & \varepsilon^2 \end{bmatrix},$$

for a small ε. This matrix has eigenvalues $2 \pm \varepsilon^2$, which are simple, but close. The matrix of eigenvectors is

$$\begin{bmatrix} -(1+\varepsilon)^{-1} & -1+\varepsilon \\ 1 & 1 \end{bmatrix},$$

and its inverse has norm $O(\varepsilon^{-2})$. Thus, the eigenvalues are ill-conditioned. ◁

5.5 Pseudospectra and Eigenvalue Conditioning

5.5.1 Spectra and Pseudospectra

We have already seen that the set of eigenvalues $\Lambda(\mathbf{A})$ of a matrix $\mathbf{A}$ is called its spectrum. Pseudospectra are eigenvalues of perturbed matrices. Given an $\varepsilon > 0$, a pseudospectrum $\Lambda_\varepsilon(\mathbf{A})$ is defined by

$$\Lambda_\varepsilon(\mathbf{A}) = \left\{ z \;\middle|\; \exists \Delta\mathbf{A} \text{ with } \|\Delta\mathbf{A}\| \le \varepsilon \text{ and } \det(z\mathbf{I} - (\mathbf{A} + \Delta\mathbf{A})) = 0 \right\}. \tag{5.11}$$

That is, z is an eigenvalue of a matrix not too different from $\mathbf{A}$. For just one eigenvalue at a time, Stewart (1998) has a simple pseudospectral result. If $\mu \in \mathbb{C}$ is an approximate eigenvalue and

$$\mathbf{r} = \mathbf{A}\mathbf{x} - \mu\mathbf{x},$$

then let

$$\mathbf{E} = -\frac{\mathbf{r}\mathbf{x}^H}{\|\mathbf{x}\|_2^2},$$

and notice that $\mathbf{Ex} = -\mathbf{r}$. It follows that

$$(\mathbf{A}+\mathbf{E})\mathbf{x} = \mu\mathbf{x}\,.$$

As a result, μ is an exact eigenvalue of $\mathbf{A}+\mathbf{E}$, and

$$\|\mathbf{E}\|_2 = \frac{\|\mathbf{r}\|_2}{\|\mathbf{x}\|_2}. \tag{5.12}$$

So, if the residual $\mathbf{r}$ has a small norm, then we have the exact eigenvalue of a computably nearby matrix; again, μ is in the pseudospectrum of $\mathbf{A}$.

How different from $\Lambda(\mathbf{A})$ can $\Lambda_\varepsilon(\mathbf{A})$ be? That is, how sensitive to changes in $\mathbf{A}$ are the eigenvalues of $\mathbf{A}$? In one sense, not very sensitive: Eigenvalues are continuous functions of the entries of $\mathbf{A}$. In another, they can be arbitrarily sensitive. Consider the matrix

$$\begin{bmatrix} 1 & 1 & \\ & 1 & 1 \\ & & 1 \end{bmatrix},$$

which, being a defective matrix, is particularly sensitive. Perturb the $(3,1)$ entry to ε^3, and the eigenvalues change to $1+\omega^k\varepsilon$, with $\omega = \frac{1}{2}(i\sqrt{3}-1)$. That is, changing $\mathbf{A}$ by $O(\varepsilon^3)$ changes the eigenvalues by $O(\varepsilon)$. Increasing the dimension makes it worse.

More generally, consider the following pragmatic observation or rule of thumb. The MATLAB function `schur` will always produce a matrix $\hat{\mathbf{U}}$ unitary up to the order of unit roundoff, that is, such that

$$\hat{\mathbf{U}}^H\hat{\mathbf{U}} = \mathbf{I} + O(\mu_M)\,,$$

and an upper-triangular matrix $\mathbf{T}$ such that

$$\mathbf{A} = \hat{\mathbf{U}}\mathbf{T}\hat{\mathbf{U}}^H + O(\mu_M)\|\mathbf{A}\|\,.$$

That is, the residual $\mathbf{R} = \mathbf{A} - \hat{\mathbf{U}}\mathbf{T}\hat{\mathbf{U}}^H$ will have entries not much bigger than $\mu_M\|\mathbf{A}\|$—at worst, a small constant depending only on n times worse. Therefore, by computing $\mathbf{T}$, we are computing elements of the pseudospectra of $\mathbf{A}$, not exactly eigenvalues, where the ε is just a smidgen bigger than $\|\mathbf{R}\|$.

Remark 5.3. The QR algorithm that underlies MATLAB's `schur` function has a very clever design, many years of refinement, and has been tested on millions of matrices. But the global convergence of the iteration is not proved. One of the more interesting lines of attack might be via isospectral flows and the Toda lattice, where the QR iteration is interpreted as steps along the flow of a dynamical system (see Watkins 1982, 1984; Chu 1984). ◁

5.5.2 Powers and Exponentials with Spectra and Pseudospectra

The connection between powers and eigenvalues is well known, and between matrix exponentials and eigenvalues also: If $\mathbf{A}$ is diagonalizable, say $\mathbf{A} = \mathbf{X}\boldsymbol{\Lambda}\mathbf{X}^{-1}$, then both matrix powers $\mathbf{A}^k = \mathbf{X}\boldsymbol{\Lambda}^k\mathbf{X}^{-1}$ and the matrix exponential $\exp(t\mathbf{A}) = \mathbf{X}\exp(t\boldsymbol{\Lambda})\mathbf{X}^{-1}$ become simpler, essentially reducing to a collection of scalar cases. If $\mathbf{A}$ is not diagonalizable, then in theory one uses the Jordan canonical form; in practice, the Schur form can be used instead. In either situation, more bookkeeping is needed, but the connection between powers and spectra is still unquestioned. We get well-known theorems that say, for example, that the limiting behavior of $\mathbf{A}^k$ is determined entirely by its eigenvalues. To be precise, if $\mathbf{x}_{n+1} = \mathbf{A}\mathbf{x}_n$ or $\dot{\mathbf{y}} = \mathbf{B}\mathbf{y}$, then the following results hold. In the first case, $\mathbf{x}_n = \mathbf{A}^n\mathbf{x}_0$ goes to zero eventually if all eigenvalues of $\mathbf{A}$ satisfy $|\lambda| \le \rho < 1$. In the second case, $\mathbf{y}(t) = e^{\mathbf{B}t}\mathbf{y}_0$ goes to zero eventually if all eigenvalues satisfy $\mathrm{Re}\lambda < 0$. But this is not the whole story!

Let $\mathbf{A}$ be an $m \times m$ matrix such that

$$\mathbf{A} = \begin{bmatrix} 1/2 & 1 & & & \\ & 1/2 & 1 & & \\ & & 1/2 & \ddots & \\ & & & \ddots & 1 \\ & & & & 1/2 \end{bmatrix}.$$

This is in Jordan form, but that doesn't matter. All eigenvalues are 1/2, so there ought to be no problem. But even for $m = 5$, we don't get what we expect! The matrix–vector products $\mathbf{A}x_0$, $\mathbf{A}^2x_0$, $\mathbf{A}^3x_0, \ldots$ initially *grow*. And when $m = 50$, they grow very rapidly. Let us look at the details, using MATLAB. To generate the matrix, we can use this command:

```
nna=@(n) eye(n)/2+triu(ones(n),1)-triu(ones(n),2);
a=nna(5);
```

This being done, we can produce the $\mathbf{A}^k\mathbf{x}_0$ as follows:

```
x=rand(5,1);
nrms=zeros(10,1);
nrms(1)=1;
for i=2:10, x=a*x; nrms(i)=norm(x);end;
```

The resulting values are 1.0000, 2.1400, 3.0207, 4.1047, 5.1064, 5.6831, 5.6926, 5.2177, 4.4492, and 3.5781. As we see, the size initially increases, only eventually decaying. It is true that $\mathbf{A}^n \to 0$ as $n \to \infty$, just like $(1/2)^n$. But the eigenvalues told us nothing about transient growth.

Now, consider now the same example but with $n = 50$:

```
a=nna(50);
x=rand(50,1);
nrms=zeros(200,1);
nrms(1)=1;
for i=2:200, x=a*x; nrms(i)=norm(x); end;
semilogy(1:200,nrms)
```

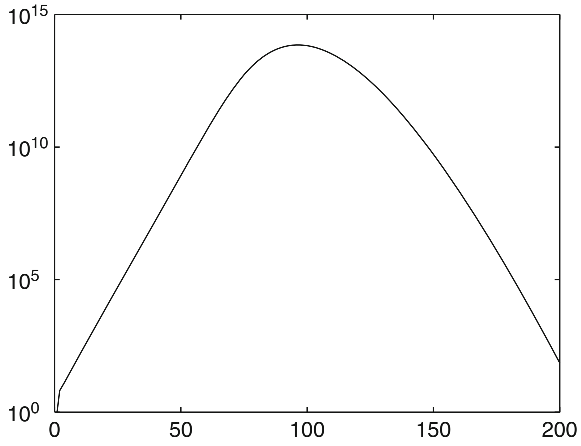


Fig. 5.1 Initial growth of matrix powers of a 50 × 50 matrix, not forbidden by all eigenvalues being 1/2 but not predicted by that, either

We see in Fig. 5.1 that the powers are going to zero via 10^{14}, which *might* present a problem. Compare the related system $\tilde{\mathbf{x}}_{n+1} = \mathbf{A}\mathbf{x}_n +$ small noise. In this case, the noise may continually activate the growth; the decay may never happen.

To understand such difficulties, one trick is instead to compute *pseudospectra* according to (5.11). The reason that pseudospectra bear on this dynamical behavior comes from the following observation. Define the ε-*pseudospectral abscissa* of $\mathbf{A}$ to be

$$\alpha_\varepsilon(\mathbf{A}) := \sup_{z \in \Lambda_\varepsilon(\mathbf{A})} \mathrm{Re}(z)\,,$$

that is, the largest that any real part of any pseudo-eigenvalue of $\mathbf{A}$ can be. Then a lower bound for the norm of $\exp(t\mathbf{A})$ is[2]

$$\sup_{t \geq 0} \|e^{t\mathbf{A}}\|_2 \geq \frac{\alpha_\varepsilon(\mathbf{A})}{\varepsilon}\,.$$

Similarly, define the ε-*pseudospectral radius* $\rho_\varepsilon(\mathbf{A})$ by

$$\rho_\varepsilon(\mathbf{A}) := \sup_{z \in \Lambda_\varepsilon(\mathbf{A})} |z|\,. \tag{5.13}$$

Then powers of $\mathbf{A}$ have norms that must grow at least as so much that

$$\sup_{k \geq 0} \|\mathbf{A}^k\| \geq \frac{\rho_\varepsilon(\mathbf{A}) - 1}{\varepsilon}\,.$$

[2] See Trefethen and Embree (2005).

For the 50×50 matrix above, the package *eigtool* computes[3] the pseudospectral radius for $\varepsilon = 10^{-12}$ to be 1.08018, which gives a lower bound on the norms of powers of $\mathbf{A}$ as $8 \cdot 10^{10}$ (see Exercise 5.12). We do indeed have growth at least this large in Fig. 5.1. That is, pseudospectra provide a partial explanation of behavior that we see in this graph, behavior that eigenvalues alone cannot explain.

Of course, matrix powers are easier to compute than the pseudospectrum is; therefore, using pseudospectra to *predict* such behavior is something that will happen only if we have prior knowledge. An alternative characterization that is easier to use computationally is the following:

$$\Lambda_\varepsilon = \left\{ z : \|(\mathbf{A} - z\mathbf{I})^{-1}\| \geq \varepsilon^{-1} \right\}; \tag{5.14}$$

that is, the norm of the *resolvent* $(\mathbf{A} - z\mathbf{I})^{-1}$ is large. Rather than prove this alternative characterization for the ordinary (simple) eigenproblem, we generalize to *matrix* polynomials,

$$\mathbf{P}(z) = \sum_{k=0}^{n} \mathbf{C}_k \phi_k(z)\,,$$

where the $\mathbf{C}_k \in \mathbb{C}^{n \times n}$ are matrices and the $\phi_k(z)$ are basis polynomials. The simple eigenvalue problem has $\mathbf{C}_0 = \mathbf{A}$ and $\mathbf{C}_1 = -\mathbf{I}$, $\phi_0(z) = 1$ and $\phi_1(z) = z$. A nonlinear eigenvalue λ has $\det \mathbf{P}(\lambda) = 0$. The spectrum

$$\Lambda(\mathbf{P}) := \left\{ z \;\middle|\; \det \mathbf{P}(z) = 0 \right\}$$

is the set of nonlinear eigenvalues. This is a simultaneous generalization of polynomial zeros $p(z) = 0$ and generalized eigenvalues $\det(z\mathbf{B} - \mathbf{A}) = 0$. Matrix polynomials find many applications, and we will see some later.

The following proposition allows us to separate the influence of the basis from the influence of the matrix polynomial.

Theorem 5.3 (Amiraslani 2006; Green and Wagenknecht 2006). *Given weights $w_k \geq 0$, not all zero, and a basis $\phi_k(z)$, define the weighted* ε-pseudospectrum *of $\mathbf{P}(z)$ as*

$$\Lambda_\varepsilon(\mathbf{P}) = \{\lambda \in \mathbb{C} : \det(\mathbf{P} + \Delta\mathbf{P})(\lambda) = 0, \|\Delta\mathbf{C}_k\| \leq \varepsilon w_k, k = 0, \ldots, n\}$$

and suppose that

$$\Delta\mathbf{P}(z) = \sum_{k=0}^{n} \Delta\mathbf{C}_k \phi_k(z)\,.$$

Moreover, let

$$B(\lambda) = \sum_{k=0}^{n} w_k |\phi_k(\lambda)|\,.$$

[3] The routine issues a warning, stating that it is not confident that the result is accurate. For $\varepsilon = 10^{-9}$, the result is 1.168, giving a more certain lower bound of $1.6 \cdot 10^8$.

Then the pseudospectrum of $\mathbf{P}(z)$ *may be alternatively characterized as*

$$\Lambda_\varepsilon(\mathbf{P}) = \{\lambda \in \mathbb{C} : \|\mathbf{P}^{-1}(\lambda)\| \geq (\varepsilon B(\lambda))^{-1}\}\,. \tag{5.15}$$

Proof. Let

$$\mathscr{S} = \{\lambda \in \mathbb{C} : \|\mathbf{P}^{-1}(\lambda)\| \geq (\varepsilon B(\lambda))^{-1}\}.$$

We show that this set is equal to $\Lambda_\varepsilon(\mathbf{P})$.

First, take $\lambda \in \Lambda_\varepsilon(\mathbf{P})$, and we show that $\lambda \in \mathscr{S}$. There are two cases. First, if λ is an eigenvalue of $\mathbf{P}(z)$, then by convention, $\|\mathbf{P}^{-1}(\lambda)\| = \infty$, and so $\lambda \in \mathscr{S}$. Second, if λ is not an eigenvalue of $\mathbf{P}(z)$, then $\mathbf{P}(\lambda)$ is nonsingular. Since $\mathbf{P}(\lambda) + \Delta\mathbf{P}(\lambda) = \mathbf{P}(\lambda)\left(\mathbf{I} + \mathbf{P}^{-1}(\lambda)\Delta\mathbf{P}(\lambda)\right)$ is singular, $\|\mathbf{P}^{-1}(\lambda)\Delta\mathbf{P}(\lambda)\| \geq 1$ must hold and so we obtain

$$\begin{aligned} 1 &\leq \|\mathbf{P}^{-1}(\lambda)\| \left(\sum_{k=0}^{n} \|\Delta\mathbf{C}_k\| |\phi_k(\lambda)|\right) \leq \|\mathbf{P}^{-1}(\lambda)\| \left(\sum_{k=0}^{n} \varepsilon w_k |\phi_k(\lambda)|\right) \\ &\leq \|\mathbf{P}^{-1}(\lambda)\| \varepsilon B(\lambda)\,. \end{aligned}$$

Hence, $\lambda \in \mathscr{S}$.

Now, let $\lambda \in \mathscr{S}$ and assume $\mathbf{P}(\lambda)$ is nonsingular. The insight comes from this:

1. Choose a unit vector $\mathbf{y}$ such that $\|\mathbf{P}^{-1}(\lambda)\mathbf{y}\| = \|\mathbf{P}^{-1}(\lambda)\|$.
2. Consider the vector $\mathbf{u} = \frac{\mathbf{P}^{-1}(\lambda)\mathbf{y}}{\|\mathbf{P}^{-1}(\lambda)\|}$, which is also a unit vector.

Then, there exists a matrix $\mathbf{H}$ such that $\|\mathbf{H}\| = 1$ and $\mathbf{H}\mathbf{u} = \mathbf{y}$ (see Higham (2002)). Now define $\mathbf{E}$ to be $\mathbf{E} = -\mathbf{H}/\|\mathbf{P}^{-1}(\lambda)\|$. Then

$$(\mathbf{P}(\lambda) + \mathbf{E})\mathbf{u} = \frac{\mathbf{y}}{\|\mathbf{P}^{-1}(\lambda)\|} - \frac{\mathbf{y}}{\|\mathbf{P}^{-1}(\lambda)\|} = 0$$

and

$$\|\mathbf{E}\| = \frac{1}{\|\mathbf{P}^{-1}(\lambda)\|} \leq \varepsilon B(\lambda)\,.$$

Define

$$\Delta\mathbf{C}_k = \operatorname{signum}(\overline{\phi_k(\lambda)}) w_k B^{-1}(\lambda)\mathbf{E}\,,$$

where $\operatorname{signum}(z) = z/|z|$ if $z \neq 0$, and 0 otherwise. Then it follows that

$$\begin{aligned} \Delta\mathbf{P}(\lambda) &= \sum_{k=0}^{n} \Delta\mathbf{C}_k \phi_k(\lambda) = \sum_{k=0}^{n} \operatorname{signum}(\overline{\phi_k(\lambda)}) \phi_k(\lambda) w_k B^{-1}(\lambda)\mathbf{E} \\ &= \sum_{k=0}^{n} |\phi_k(\lambda)| w_k B^{-1}(\lambda)\mathbf{E} = B(\lambda)B^{-1}(\lambda)\mathbf{E} = \mathbf{E}\,. \end{aligned}$$

This proves the theorem. ♮

Remark 5.4. This proposition allows us to separate some of the properties of the polynomial $\mathbf{P}(z)$ from the properties of the basis. In particular, notice that the left-hand side of the inequality in this characterization of the pseudospectrum is *basis-independent*, being merely a property of the size of $\mathbf{P}^{-1}(\lambda)$, whereas the right-hand side of the inequality depends only on the tolerance ε and the value of the scalar function $B(z)$ at $z = \lambda$, which depends only on the basis and on the weights w_k. More, it was noticed in Green and Wagenknecht (2006) that the basis functions need not even be polynomials—this gives a pseudospectral theorem for general *nonlinear* eigenvalues as well. ◁

Remark 5.5. When all weights $w_k = 1$, the function $B(z)$, in the case of Lagrange interpolation, is precisely what is known as the *Lebesgue function* of the set of interpolation nodes (Rivlin 1990). There is an extensive theory of such functions, and their connection to the problem of conditioning. In the standard notation, for $w_k \equiv 1$, $B(z) = \lambda_n(\mathbf{x};z)$. ◁

To end this section, we provide an example of how to generate a contour plot of pseudospectra.

Example 5.12. The following degree-2 matrix polynomial is expressed in the Lagrange basis at the three points $\tau_k = 0, -1, -2$. The pseudospectra can be analyzed with the help of the following MATLAB code:

```
%
% Example 2 by 2 matrix polynomial pseudospectrum
% in the Lagrange basis, using Mandelbrot polynomials
%
% RMC 13.12.2010 after an idea by Piers Lawrence
% and after discussion with Nic Fillion
%
%
% This first section just sets up the matrix polynomial of degree
% 2^level - 1, formed by interpolating at n+2 = 2^(level-1)+1
% points giving 2 by 2 matrices at each interpolation point.
%
level = 6;
p = 1;
rts = [-1];
for i=2:level,
    % Evaluate the matrix polynomials at x=0, x=-2, and x=all
    % roots previous.
    n = 2^(i-1) - 1; % n roots available
    tau = [0, rts', -2 ]; % n+2 evaluation points
    gamma = genbarywts( tau, 1 );
    gamma = gamma/norm(gamma,inf);
    A = zeros( 2*(n+2)+2, 2*(n+2)+2 ); % extra block at end
    B0 = eye( 2*(n+2)+2, 2*(n+2)+2 );
    B0(end-1:end, end-1:end ) = zeros(2,2);
    A(1:2, 1:2) = [ 0, 0; 0, 0 ];
    A(1:2, end-1:end) = [ 0, -1; 1, 1];
    A(end-1:end,1:2) = [gamma(1), 0; 0, gamma(1)];
    for j=1:n,
        A( 2 + 2*j-1:2+2*j, 2+2*j-1:2+2*j ) = [ tau(j+1), 0 ; 0, tau(j+1) ];
        A( 2+2*j-1:2+2*j, end-1:end ) = [ 0, -1; 1, 0 ];
        A( end-1:end, 2+2*j-1:2+2*j ) = [ gamma(j+1), 0; 0, gamma(j+1) ];
    end;
    A(end-3:end-2,end-3:end-2) = [-2, 0; 0, -2];
    A(end-3:end-2,end-1:end) = [2, -1; 1, -1];
    A(end-1:end,end-3:end-2) = [gamma(end),0; 0, gamma(end)];
    rts = eig( A, B0 );
    k = find( abs(rts) < 3 );
    rts = rts(k);
end;
%
% At this point, we have our matrix, interpolated at the nonlinear
% eigenvalues of the previous level matrix polynomial.

% Compute the weights for B as the 2-norms of the polynomial coefficients
alpha = zeros( 1, n+2 );
for i=1:n+2,
    alpha(i) = norm( A( 2*i-1:2*i, end-1:end ), 2 );
end;

% Now evaluate the matrix polynomial at a bunch of points and compute
% the 2-norm of its inverse, by simple svd computation.
nsamp = 400;
x = linspace(-2.5,0.7,nsamp);
y = linspace(-2,2,nsamp);
% z(j,k) = x(j) + i*y(k)
V = zeros(nsamp,nsamp);
for j=1:nsamp,
    for k=1:nsamp,
        z = x(j)+sqrt(-1)*y(k);
        w = prod( z - tau );
        P = gamma(1)*[ 0, -1; 1, 1]/(z-0) + gamma(end)*[2, -1; 1, -1]/(z+2);
        B = abs(alpha(1)*gamma(1)/z) + abs(alpha(end)*gamma(end)/(z+2));
        for ell=1:n,
            P = P + gamma(ell+1)*[ 0, -1; 1, 0 ]/(z-tau(ell+1));
            B = B + abs( alpha(ell+1)*gamma(ell+1)/(z-tau(ell+1)) );
        end;
        S = svd( w*P );
        V(k,j) = S(2)/(abs(w)*B);
    end;
end;
% Choose among the following options by commenting out as appropriate.
% The first two lines, with level=2, were used to generate the
    figure
% in the text.
%[C,h] = contour( x, y, log(V)/log(10), [-2,-1.5,-1,-0.75, -0.5,
    -0.25], 'k-' );
[C,h] = contour( x, y, log(V)/log(10), [-22,-18,-14,-10,-6,-2], '
    k-' );
clabel( C, h );
%contour( x, y, log(V)/log(10),[0,-2,-4], 'k-' );
%[C,h] = contour( x, y, log(V)/log(10), 'k-' );
%clabel( C, h );
axis('square')
```

The resulting contour plot of the pseudospectra is displayed in Fig. 5.2. ◁

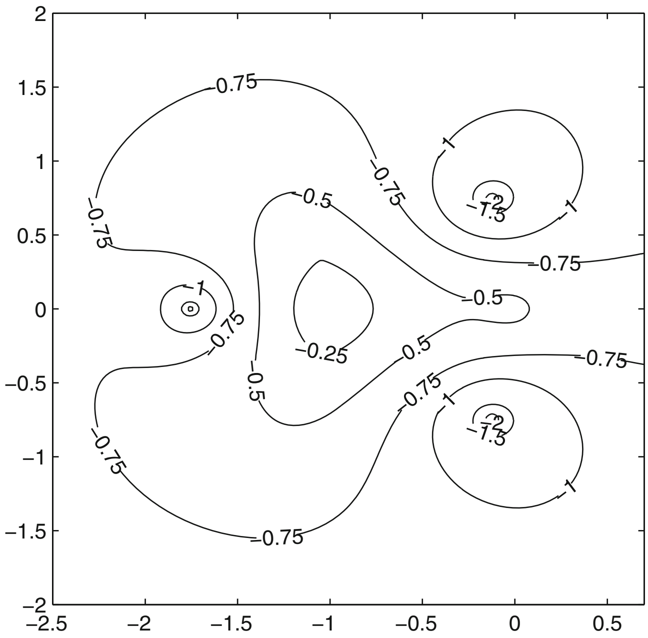


Fig. 5.2 Pseudospectrum of a 2×2 matrix polynomial of degree 3

5.6 Notes and References

Once again, a useful reference for the themes discussed in this chapter is the *Handbook* by Hogben (2006). For an introduction to the Weyr form, which we have merely mentioned, see O'Meara et al. (2011).

Our discussion of the QR algorithm follows that of Stewart (1998 chap. 15), where the algorithm is described in detail. Golub and Uhlig (2009) contains historical details surrounding the development of the QR algorithm.

The proof of Theorem 5.3 is taken from Amiraslani (2006) and was modeled there exactly on the one for the monomial basis in Tisseur and Higham (2001).

Problems

Theory and Practice

5.1. Show that a left eigenvector $\mathbf{y}^H$ of $\mathbf{A}$ with eigenvalue λ corresponds to a right eigenvector $\mathbf{y}$ of $\mathbf{A}^H$ with eigenvector $\bar{\lambda}$.

5.2. Suppose that $\mathbf{Ax} = \lambda\mathbf{x}$. Show that

1. $\mathbf{A} - s\mathbf{I}$ has eigenvalues $\lambda - s$.
2. $\mathbf{A}^k$ has eigenvalues λ^k.
3. Consider the polynomial $f(z) = \sum_{k=0}^{n} \alpha_k z^k$ and the matrix polynomial $f(\mathbf{A}) = \sum_{k=0}^{n} \alpha_k \mathbf{A}^k$. Show that the eigenvalues of $f(\mathbf{A})$ are $f(\lambda)$.
4. If $\mathbf{A}$ is nonsingular, then the eigenvalues of $\mathbf{A}^{-1}$ are λ^{-1}.
5. Let $h(z) = f(z)/g(z)$ be a rational function (f and g are polynomials). Suppose $g(\mathbf{A})$ is nonsingular. Define $h(\mathbf{A}) = g(\mathbf{A})^{-1}f(\mathbf{A})$. Show that $h(\mathbf{A})$ has eigenvalues $h(\lambda)$.
6. Show that $g(\mathbf{A})$ is singular if $g(\lambda) = 0$ for any eigenvalue λ of $\mathbf{A}$.

5.3. If λ is a simple eigenvalue of $\mathbf{A}$, that is, of multiplicity 1, show that its left and right eigenvectors $\mathbf{y}^H$ and $\mathbf{x}$ satisfy

$$\mathbf{y}^H\mathbf{x} \neq 0.$$

(Hint: Unitary similarity transformations of $\mathbf{A}$, say to $\mathbf{T} = \mathbf{U}^H\mathbf{AU}$, do not change $\mathbf{y}^H\mathbf{x}$ (show this). Then you can work with $\mathbf{T}$, and you may assume λ is in the $(1,1)$ position, and because it is simple, that's the only diagonal entry where it appears.)

5.4. Show that if λ is a simple eigenvalue of $\mathbf{A}$ with left and right eigenvectors $\mathbf{y}^H$ and $\mathbf{x}$, and $\mathbf{E}$ is a matrix with small enough norm, then there is an eigenvalue $\hat{\lambda}$ of $\mathbf{A}+\mathbf{E}$ such that

$$\hat{\lambda} = \lambda + \frac{\mathbf{y}^H\mathbf{E}\mathbf{x}}{\mathbf{y}^H\mathbf{x}} + O(\|\mathbf{E}\|_2^2)\,.$$

Moreover, show that, as a result,

$$|\hat{\lambda} - \lambda| \leq \sec\theta\|\mathbf{E}\|_2 + O(\|\mathbf{E}\|_2^2)\,,$$

where θ is the angle between the vectors $\mathbf{x}$ and $\mathbf{y}$. Therefore, $\sec\theta = (\mathbf{y}^H\mathbf{x})^{-1}$ serves as an absolute condition number for a simple eigenvalue λ if $\|\mathbf{x}\| = \|\mathbf{y}\| = 1$.

5.5. Show that the eigenvalues of the matrix

$$\begin{bmatrix} 2 & 1 \\ \varepsilon^2 & 2 \end{bmatrix}$$

are increasingly ill-conditioned as $\varepsilon \to 0$.

5.6. MAPLE or another CAS may be helpful for this problem.

1. Show that you may choose $a_1, a_2, \ldots, a_{10}$ leaving $a_{11}, a_{12}, \ldots, a_{19}$ free in such a way that the eigenvalues of the 20×20 Clement matrix of Problem (4.29)(a) are $\pm 1/2, \pm 3/2, \pm 5/2, \ldots, \pm 9/2$, so that $21/2\mathbf{I} + \mathbf{C}$ has eigenvalues $1, 2, 3, \ldots, 20$ (the same as the roots of the first Wilkinson polynomial).
2. Choose a_{11} through a_{19} to minimize $\kappa_1(\mathbf{V}) = \|\mathbf{V}\|_1 \|V^{-1}\|_1$ (the 1-norm condition number) as best you can.
3. Letting $\alpha_i = \|\mathbf{V}^{-1}_{(i)}\|_2$, the 2-norm of the ith row of $\mathbf{V}^{-1}$, and $\beta_i = \|\mathbf{V}_i\|_2$, the 2-norm of the ith column of $\mathbf{V}$, show that $\alpha_i\beta_i$ is the condition number of the ith eigenvalue. What are these numbers, and how do they compare to $\kappa_1(\mathbf{V})$?

5.7. Use the normalized power method to find the largest eigenvalue of the Clement matrix of order 10, to three digits of accuracy. Use the fact that if $\mathbf{r} = \mathbf{A}\mathbf{x} - \mu\mathbf{x}$, then μ is an eigenvalue of $\mathbf{A} + \mathbf{E}$ with $\mathbf{E} = -\mathbf{r}\mathbf{x}^H / \|\mathbf{x}\|_2^2$ to argue for the accuracy of your result. What is the condition number of the largest eigenvalue?

5.8. The most common use of the power method is to find an eigenvalue of $(\mathbf{A} - \mu\mathbf{I})^{-1}$ close to μ. That is, for random $\mathbf{x}_0$, define $\mathbf{z}_0 = \mathbf{x}_0 / \|\mathbf{x}_0\|$ and $\mathbf{z}_m$ for $m = 1, 2, \ldots$ by solve $(\mathbf{A} - \mu\mathbf{I})\mathbf{x} = \mathbf{z}_i$ and put $\mathbf{z}_{i+1} = \mathbf{x} / \|\mathbf{x}\|$. Use this iteration to find an eigenvector for the smallest eigenvalue of the symmetric Clement matrix of order 10.

5.9. If $\mathbf{q}^H$ is a left eigenvector of $\mathbf{A}$ corresponding to λ, and $\mathbf{Q} = [\mathbf{Q}_1, \mathbf{q}]$ is unitary, show that

$$\mathbf{Q}^H\mathbf{A}\mathbf{Q} = \begin{bmatrix} \mathbf{Q}_1^H\mathbf{A}\mathbf{Q}_1 & \mathbf{Q}_1^H\mathbf{A}\mathbf{q} \\ 0 & \lambda \end{bmatrix}$$

and that the other eigenvalues of $\mathbf{A}$ are the eigenvalues of the smaller matrix $\mathbf{Q}_1^H\mathbf{A}\mathbf{Q}_1$.

5.10. In this problem, we examine how to construct a unitary matrix $\mathbf{Q}$ with a given unit vector $\mathbf{q} \in \mathbb{C}^n$ as the last column, that is, such that $\mathbf{Q} = [\mathbf{Q}_1, \mathbf{q}]$.

1. Put $\mathbf{u} = \mathbf{q} + \mu\mathbf{e}_n$, where $\mu = \operatorname{signum}(q_n) = q_n/|q_n|$ unless $q_n = 0$, when you take $\mu = 1$. Show that $\mathbf{u}^H\mathbf{u} = 2(1 + |q_n|)$.
2. Show that $\mathbf{R} = \mathbf{I} - 2\frac{\mathbf{u}\mathbf{u}^H}{\mathbf{u}^H\mathbf{u}}$ is unitary.

3. Show that $\mathbf{R}\mathbf{e}_n = -\mu^{-1}\mathbf{q}$. Put $\mathbf{Q} = \mathbf{R} \cdot \mathrm{diag}(1, 1, \ldots, -\mu)$ and show that $\mathbf{Q}$ is unitary and $\mathbf{Q}\mathbf{e}_n = \mathbf{q}$.
4. Show that $\mathbf{Ru} = -\mathbf{u}$ (so $\mathbf{R}$ is a "reflector"), $\mathbf{R}_q = -\mu\mathbf{e}_n$.

5.11. Let

$$\mathbf{T} = \begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 0 & 0 & 1/2 & 0 \end{bmatrix} \tag{5.16}$$

and $\mathbf{P}(z) = (z\mathbf{T})^7 - \mathbf{I}$. Draw the pseudospectra of the matrix polynomial $\mathbf{P}(z)$, choosing pleasing or informative contour levels (or both).

5.12. Suppose $\mathbf{J}_n(r\exp(i\theta))$ is an $n \times n$ Jordan block matrix with eigenvalue $z = r\exp(i\theta)$, where $0 \leq r < 1$, so powers of $\mathbf{J}$ must ultimately decay. By taking $\varepsilon = er^n$, where e is the base of the natural logarithms, and using the analytic formula for $\mathbf{R} = (z\mathbf{I} - \mathbf{J})^{-1}$ [all you need is the $(1, n)$ entry], give a lower bound on the pseudospectral radius and thus a lower bound on the maximum value of $\|\mathbf{J}^k\|$. Take $r = 1/2$ and $n = 50$ and compare your results with Fig. 5.1.

5.13. Show that by using Householder reflectors, one may find a matrix $\mathbf{H} = Q^{\mathbf{HAQ}}$ similar to $\mathbf{A}$ that is upper Hessenberg in structure, meaning that the entries below the first subdiagonal are zero.

5.14. Show that the QR factoring of an upper Hessenberg matrix costs $O(n^2)$ flops. So, QR iteration is much cheaper if it is preceded by a similarity transformation to upper Hessenberg form.

Investigations and Projects

5.15. The matrix `gallery(3)` in MATLAB

$$\mathbf{A} = \begin{bmatrix} -149 & -50 & -154 \\ 537 & 180 & 546 \\ -27 & -9 & -25 \end{bmatrix}, \tag{5.17}$$

is known to have somewhat sensitive eigenvalues. The exact eigenvalues of this matrix are $\lambda = 1, 2$, and 3. If we perturb this to $\mathbf{A} + t\mathbf{E}$, where

$$\mathbf{E} = \begin{bmatrix} -390 & 1170 & 0 \\ -129 & 387 & 0 \\ -399 & 1197 & 0 \end{bmatrix}, \tag{5.18}$$

which happens to be an outer product $\mathbf{y}\mathbf{x}^T$, where $\mathbf{x}$ and $\mathbf{y}$ are eigenvectors of the smallest eigenvalue $\lambda = 1$ of $\mathbf{A}$, then we find for

$$t = -2.612641635322647 \times 10^{-7}$$

(approximately) that $\mathbf{A} + t\mathbf{E}$ has a double eigenvalue. This means that if we change the matrix by approximately 10^{-7}, one of the eigenvalues must change at least by $1/2$. Compute the condition number of each eigenvalue of $\mathbf{A}$ in MATLAB or MAPLE and plot the eigenvalues of $\mathbf{A} + t\mathbf{E}$ on $-5 \times 10^{-7} \le t \le 0$ for, say, 300 values of t. Compute the pseudospectral contours of $\mathbf{A}$ using `eigtool`. Discuss.

5.16. Suppose $\mathbf{AX} = \mathbf{X}\boldsymbol{\Lambda}$ for $\boldsymbol{\Lambda} = \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$ and nonsingular matrix $\mathbf{X}$ of eigenvectors. Suppose further that the eigenvalues λ_k are distinct.

1. If we know μ, and it happens to be true that $\mu = \lambda_1 + \varepsilon$, for some $|\varepsilon| \ll 1$, show that one step of *inverse-power iteration*

$$(\mathbf{A} - \mu\mathbf{I})\,\mathbf{y} = \mathbf{z}$$

improves on $\mathbf{z}$ as an estimate of the first eigenvector $\mathbf{x_1}$ by a factor $1/\varepsilon$. Hint: Write

$$\mathbf{z} = a_1\mathbf{x}_1 + \sum_{k=2}^{n} a_k\mathbf{x}_k$$

and (as a theoretical tool) use the above factoring in the conceptual solution process. (Of course, when we actually *solve* this system, we'd use something like an LU factoring. But for the purpose of analysis, you can use the (unknown) eigenvector–eigenvalue decomposition.)

2. Fill in the details as to why, if the residual

$$\mathbf{r} = \mathbf{z} - (\mathbf{A} - \mu\mathbf{I})\,y$$

is small compared to $\|\mathbf{A}\|$, then rounding errors in the solution of the system for the inverse-power iteration don't matter much even if ε is very tiny, so that $\mathbf{A} - \mu\mathbf{I}$ is very nearly singular and thus extremely ill-conditioned. Can we expect, though, that the residual $\mathbf{r}$ will be small?

5.17. A companion matrix for $z^2 - 2bz + 1 = 0$ is

$$\mathbf{C} = \begin{bmatrix} 0 & -1 \\ 1 & 2b \end{bmatrix}. \tag{5.19}$$

1. Compute the condition number for each eigenvalue of $\mathbf{C}$. Identify any values of b for which the eigenproblem is ill-conditioned.
2. Forgetting the companion matrix for the moment, compute the condition number of each root of $z^2 - 2bz + 1 = 0$ as a function of b. This is a *structured* condition number of the eigenvalue problem. Compare with your previous answer.

3. In Question 1.18, you used the quadratic formula to compute each root and computed the product, which should have been 1, but instead gave curious “tiger stripes” for $b \approx 10^7$ and ultimately became zero if b was large enough. Is this behavior a result of ill-conditioning, or is it instead a result of numerical instability? Justify your choice.

Chapter 6
Structured Linear Systems

Abstract We define *structured* linear systems to include *sparse* systems or systems with *correlated entries* or *both*. We define the *structured backward error* and a *structured condition number*. We give examples of various classes of structured linear systems and *examples* of algorithms that *take advantage* of the special structure. ◁

6.1 Taking Advantage of Structure

Taking advantage of structure is an very important aspect of numerical linear algebra. We articulate the discussion that follows around the notions of sparsity and correlation of matrix entries.

Definition 6.1. A *sparse* matrix has a large proportion of entries that are equal to zero.

The zero entries do not need to be stored, and arithmetic with zero is, of course, easy and accurate.[1] See Fig. 6.1 for a simple visualization of a particular sparse matrix.[2]

Definition 6.2. A matrix has *correlated entries* when all entries of the matrix are determined wholly by a small number of parameters.

The following simple example should make the idea of correlation clear.

[1] See Davis and Hu (2011) for a beautiful collection of useful sparse matrices. See http://www.cise.ufl.edu/research/sparse/matrices/synopsis/ for an introduction to visualizing large sparse matrices by minimizing the "energy" in a graph related to the matrix.

[2] Also, see Fig. 16.15 for `spy` pictures of sparse matrices arising in finite-difference solutions to partial differential equations.

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_6,

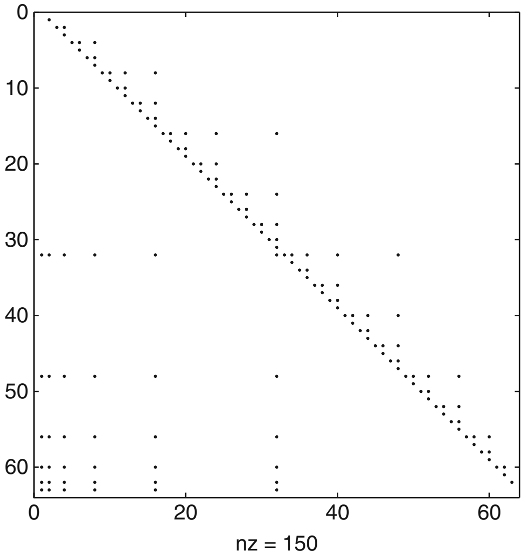


Fig. 6.1 A simple visualization of the inverse of the 64 × 64 Mandelbrot matrix, using the `spy` command. This sparse matrix has only 150 nonzero entries out of a possible $64 \times 64 = 4096$ entries, or about 3.6%

Example 6.1. The 2 × 2 matrix

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

is determined entirely by one parameter, θ. Thus, all entries are correlated, in this case nonlinearly. ◁

Definition 6.3. We call a matrix *structured* if it is sparse, has correlated entries, or both.

Example 6.2. We have already seen examples of structured matrices, including the generalized Vandermonde matrices whose n^2 entries are completely determined by just the n nodes τ_0, τ_1, …, τ_{n-1} and the choice of basis functions. Another especially nice example family are the *circulant* matrices, which we discuss in detail in Chap. 9. Consider a 4 × 4 circulant matrix:

$$\mathbf{C} = \begin{bmatrix} a & b & c & d \\ d & a & b & c \\ c & d & a & b \\ b & c & d & a \end{bmatrix} .$$

The pattern is as follows: The first row is repeated but shifted cyclically, then again for the next row, and so on. Thus, each row is correlated with the first one. We will shortly see other structures that are important in practice. ◁

Until now, we took no advantage of any structure that happened to be present in our examples, except for unitary or triangular matrices: otherwise, $\mathbf{A} \in \mathbb{C}^{m\times n}$ was treated as a collection of $m \cdot n$ independent complex nonzero numbers. But if there is one lesson to draw from the applications of linear algebra, it is this: *Many, if not most, system matrices found in applications are structured.* To solve such systems accurately, efficiently, or at all (when they are really large), *the structure must be used.* Sparsity is especially helpful. Correlations in the entries also have a large impact but often introduce nonlinearities and so are usually much harder to take advantage of. This chapter gives a too-brief introduction to these ideas. We look at some of the most elementary techniques for solving structured linear systems and for structured eigenproblems.

First, though, we remind the reader of the unstructured case. The techniques of LU factoring, pivoting, QR factoring, and the SVD were presented in the two previous chapters for dense matrices. Such matrices are called *dense* in the literature because most entries a_{ij} in the matrix $\mathbf{A}$ are presumed to be nonzero and independent. In that case, the cost of solving a typical $n \times n$ dense system by each of those methods, using a serial (nonparallel, nonvector) computer, is reasonably well known and typically is considered to vary as the cube of n:

- LU factoring costs $\frac{2}{3}n^3 + o(n^3)$ flops.
- QR factoring costs $4n^3 + o(n^3)$ flops.
- SVD costs (about) $11n^3 + o(n^3)$ flops.

We see that the SVD is in theory the most expensive. These estimates are reasonable models for many computers, as we see in simulation.

Nonetheless, as a matter of fact, GOOGLE plays with its PageRank matrix, where n is measured in *billions*. They get answers in (pretty much) real time. How? Fast computers, sure. But really: *fast algorithms*. The key to fast algorithms is *parsimony*. As the philosopher Seneca is said to have said:

> Economy is in itself a significant source of revenue.

A parsimonious approach can often be adopted in order to take advantage of the structure of matrices.

Example 6.3. Consider the following structured matrix:

$$\mathbf{T} = \begin{bmatrix} 4 & 1 & & & & \\ 1 & 4 & 1 & & & \\ & 1 & 4 & 1 & & \\ & & 1 & 4 & 1 & \\ & & & 1 & 4 & 1 \\ & & & & 1 & 4 \end{bmatrix}.$$

We focus first on its sparsity and ignore the correlations between the nonzero entries. Such a matrix is an example of a "banded matrix," in this case with three bands, or in other words "tridiagonal." It is not necessary to store the zeros. We just store the diagonal entries. If we note the symmetry (which is a correlation), then we only need to store the diagonal and one copy of the other bands, that is,

$$\begin{bmatrix}4\;4\;4\;4\;4\;4\end{bmatrix} \qquad \text{and} \qquad \begin{bmatrix}1\;1\;1\;1\;1\end{bmatrix}.$$

We will take further advantage later of the fact that all entries of these vectors are identical. Note that the LU factoring of this tridiagonal matrix, in which $\mathbf{T} = \mathbf{LDL}^H$ with

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ {}^{1}\!/\!{}_{4} & 1 & 0 & 0 & 0 \\ 0 & {}^{4}\!/\!{}_{15} & 1 & 0 & 0 \\ 0 & 0 & {}^{15}\!/\!{}_{56} & 1 & 0 \\ 0 & 0 & 0 & {}^{56}\!/\!{}_{209} & 1 \end{bmatrix} \tag{6.1}$$

and $\mathbf{D} = \mathrm{diag}({}^{15}\!/\!{}_{4}, {}^{56}\!/\!{}_{15}, {}^{209}\!/\!{}_{56}, {}^{780}\!/\!{}_{209})$, may be done with $O(n)$ work, not $O(n^3)$. The savings are substantial for large n. Again, the zeros in $\mathbf{L}$ need not be stored, and neither need the entries along the diagonal, which are all 1. ◁

Remark 6.1. In the previous example, notice also that the inverse matrix $\mathbf{T}^{-1}$ is *full*:

$$\mathbf{T}^{-1} = \begin{bmatrix} {}^{209}\!/\!{}_{780} & -{}^{14}\!/\!{}_{195} & {}^{1}\!/\!{}_{52} & -{}^{1}\!/\!{}_{195} & {}^{1}\!/\!{}_{780} \\ -{}^{14}\!/\!{}_{195} & {}^{56}\!/\!{}_{195} & -{}^{1}\!/\!{}_{13} & {}^{4}\!/\!{}_{195} & -{}^{1}\!/\!{}_{195} \\ {}^{1}\!/\!{}_{52} & -{}^{1}\!/\!{}_{13} & {}^{15}\!/\!{}_{52} & -{}^{1}\!/\!{}_{13} & {}^{1}\!/\!{}_{52} \\ -{}^{1}\!/\!{}_{195} & {}^{4}\!/\!{}_{195} & -{}^{1}\!/\!{}_{13} & {}^{56}\!/\!{}_{195} & -{}^{14}\!/\!{}_{195} \\ {}^{1}\!/\!{}_{780} & -{}^{1}\!/\!{}_{195} & {}^{1}\!/\!{}_{52} & -{}^{14}\!/\!{}_{195} & {}^{209}\!/\!{}_{780} \end{bmatrix}.$$

Thus, the inverse does not have the sparsity $\mathbf{T}$ has. Therefore, knowledge of the factors is *better*, that is, more economical, and has the same utility as knowledge of the inverse. ◁

Tridiagonal LU factoring (without pivoting, true, so we have to worry about numerical stability) is good in this case, but we may do even better by writing an algorithm specifically conceived for this type of problem. One trick is to not even *store* the 4s and 1s, and to use a *program* to multiply $\mathbf{Tx}$:

```
function y=Seneca(x)
[n,ignore]=size(x(:));
y = zeros(n,1);
y(1)=4*x(1)+x(2);
   for i=2:n-1,
      y(i)=-x(i-1)+4*x(i)+x(i+1);
   end;
y(n)=-x(n-1)+4*x(n);
end;
```

Notice this program occupies *constant* storage, independent of n. Even better than this code is to eliminate the loop, as follows:

```
function y=Seneca(x)
[n,ignore]=size(x(:));
y = zeros(n,1);
y(1)=4*x(1)+x(2);
y(2:n-1)=-x(1:n-2)+4*x(2:n-1)+x(3:n);
y(n)=-x(n-1)+4*x(n);
end;
```

This makes things a constant factor better. As Cleve Moler emphasized, "[W]hen you have mastered the colon, you have mastered MATLAB."

But if all we can do is call a subroutine to multiply $\mathbf{x}$ by a matrix $\mathbf{A}$ (such as $\mathbf{T}$ above), how can we solve $\mathbf{Ax} = \mathbf{b}$? Basically, we use *powers of the matrix*, as we will see in Chap. 7; if we can compute $\mathbf{y} = \mathbf{Ax}$, then it is easy to compute $\mathbf{w} = \mathbf{Ay} = \mathbf{A}^2\mathbf{x}$, and so on. These vectors can be used to find a solution. But for now, before turning to those iterative methods, we will look at examples of classes of structured matrices. Even simple structures can be very important.

6.2 Real Symmetric Positive-Definite Matrices

We begin with a venerable and benign class: real symmetric positive-definite (SPD) matrices. It is easy to verify that the matrix $\mathbf{T}$ from Example 6.3 was of this type.

Definition 6.4. A matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ is *real symmetric positive-definite* if

1. $\mathbf{A} \in \mathbb{R}^{n\times n}$, that is, $\mathbf{A}$ is real,
2. $\mathbf{A}^H = \mathbf{A}^T = \mathbf{A}$, that is, $\mathbf{A}$ is symmetric,
3. $\mathbf{A}$ is positive-definite, a condition that can be formulated in three equivalent ways:

   a. $\mathbf{x}^T\mathbf{Ax} > 0$ for all nonzero $\mathbf{x} \in \mathbb{R}^n$;
   b. all eigenvalues λ of $\mathbf{A}$ are real and positive;
   c. all pivots in the LU factoring (without pivoting) are positive.

Such matrices are very common in applications.

In what follows, we will expand on this appropriate description of PSD matrices by Higham (2002,196):

> Symmetric positive definiteness is one of the highest accolades to which a matrix can aspire. Symmetry confers major advantages in the eigenproblem and [...] positive definiteness permits economy and numerical stability in the solution of linear systems.

To begin with, if $\mathbf{A}$ is SPD, $\mathbf{A} = \mathbf{LU}$, the factoring arising from Gaussian elimination *without pivoting*, can be expressed as $\mathbf{A} = \hat{\mathbf{L}}\hat{\mathbf{L}}^T$ (the Cholesky factoring); in this book, we will also loosely call the factoring $\mathbf{A} = \mathbf{LDL}^T$ [the symmetric (no square root) factoring] Cholesky factoring. This saves a (modest) factor of two in computation cost and in storage cost; but a factor of two is not nothing, especially for large systems.

More importantly, if we had to pivot, we would potentially have had to *destroy the structure*. Avoidance of pivoting means more efficiency, in this case. Of course, eschewing pivots means accepting possibly worse accuracy because, as we saw in Chap. 4, LU factoring without pivoting was numerically unstable in general, giving rise to growth factors arbitrarily larger than the worst-case 2^{n-1} of partial pivoting. Indeed, the condition number of $\mathbf{A}$, $\kappa_2(\mathbf{A})$, for solving $\mathbf{Ax} = \mathbf{b}$, can be large, and is in some applications. That is, even if we did use pivoting, our answers might be inaccurate; numerical instability could only make things worse. For SPD matrices, though, we hope that any instability is not too serious, because we are in a hurry for a solution. And we are in luck: Theorem 10.4 of Higham (2002) assures us that for SPD matrices, Cholesky factoring is perfectly stable in a normwise backward error sense. Indeed, the growth factor is just 1 and is not much worse even in a componentwise sense.

In addition, the simple eigenvalues of $\mathbf{A}$ are perfectly conditioned, because left eigenvectors are also right eigenvectors; that is,

$$\mathbf{Ax} = \lambda \mathbf{x} \quad \Leftrightarrow \quad \mathbf{x}^T\mathbf{A} = \lambda \mathbf{x}^T .$$

So the condition number $(\mathbf{y}^T\mathbf{x})^{-1}$ is just 1 (normalizing so $\|\mathbf{x}\| = 1$). In fact, the Schur factoring

$$\mathbf{A} = \mathbf{UTU}^T \tag{6.2}$$

has a diagonal matrix $\mathbf{T}$ (not just upper triangular) and so $\mathbf{A}$ is unitarily diagonalizable, even if $\mathbf{A}$ has multiple eigenvalues. Up to this point, this is true of all real-symmetric matrices; we haven't used positive-definiteness. If all $\lambda > 0$, then Eq. (6.2) gives the SVD of $\mathbf{A}$, not just the Schur factoring! For symmetric positive-definite matrices $\mathbf{A}$, eigenvalues are singular values.

Algorithms for solving the symmetric eigenproblem are significantly faster and more reliable than those for the unsymmetric eigenproblem, as a result of the symmetry (see Problem 6.16). This extends to Hermitian matrices $\mathbf{A} \in \mathbb{C}^{m\times n}$ with $\mathbf{A}^H = \mathbf{A}$, but *not* to "complex symmetric" matrices with $\mathbf{A}^T = \mathbf{A}$ (about which more appears later). For example,

$$\mathbf{A} = \begin{bmatrix} 2 & i \\ i & 0 \end{bmatrix}$$

is complex symmetric but not Hermitian, since $\mathbf{A}^T = \mathbf{A}$ but

$$\mathbf{A}^H = \begin{bmatrix} 2 & -i \\ -i & 0 \end{bmatrix} \neq \mathbf{A} .$$

Also, by inspection (actually by construction),

$$\mathbf{A}\begin{bmatrix} 1 \\ i \end{bmatrix} = \begin{bmatrix} 2 & i \\ i & 0 \end{bmatrix}\begin{bmatrix} 1 \\ i \end{bmatrix} = \begin{bmatrix} 2-1 \\ i \end{bmatrix} = \begin{bmatrix} 1 \\ i \end{bmatrix} .$$

So 1 is an eigenvalue with eigenvector $[1,i]^T$. By symmetry, $[1,i]$ is the left eigenvector, but $\mathbf{y}^T\mathbf{x}$ is

$$\begin{bmatrix} 1 & i \end{bmatrix} \begin{bmatrix} 1 \\ i \end{bmatrix} = 1 - 1 = 0,$$

and so 1 is infinitely ill-conditioned as an eigenvalue (in fact, 1 is a double eigenvalue). Contrast this case with a different $\mathbf{A}$, such as

$$\mathbf{A} = \begin{bmatrix} 1 & i \\ -i & 1 \end{bmatrix},$$

which is Hermitian, since $\mathbf{A}^H = \mathbf{A}$, and has eigenvectors $[1,i]^T$ and $[1,-i]^T$. The corresponding left eigenvectors are $[1,\mp i]$, with

$$\begin{bmatrix} 1 & \mp i \end{bmatrix} \begin{bmatrix} 1 \\ \pm i \end{bmatrix} = 2, \tag{6.3}$$

not 0, and indeed, if we normalize the eigenvectors, we have perfect condition number 1.

6.3 Banded Matrices

Perhaps the next most commonly useful structured matrices are *banded*, either tridiagonal, such as we have already seen, or as below:

$$\begin{bmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & -1 & 2 & -1 \\ & & -1 & 2 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 2 & -1 & & \\ -1 & 3 & -1 & \\ & -1/2 & 4 & -3/2 \\ & & -17 & 1 \end{bmatrix}.$$

The first is also symmetric positive-definite, a double benefit, while the second is not even symmetric. If the matrix has five bands, it is sometimes called pentadiagonal. For instance, consider the matrix

$$\begin{bmatrix} 2 & -1/2 & -1/4 & & \\ -1/2 & 2 & -1/2 & -1/4 & \\ -1/4 & -1/2 & 2 & -1/2 & -1/4 \\ & -1/4 & -1/2 & 2 & -1/2 \\ & & -1/4 & -1/2 & 2 \end{bmatrix},$$

which has two subdiagonals and two superdiagonals. Again, this example is symmetric positive-definite; it thus permits LU factoring without pivoting, in which either the Cholesky factoring or the symmetric factoring can be banded; in fact, we have $\mathbf{A} = \mathbf{LDL}^T$ with

$$\mathbf{L} = \begin{bmatrix} 1 & & & & \\ -1/4 & 1 & & & \\ -1/8 & -3/10 & 1 & & \\ & -2/25 & -23/72 & 1 & \\ & & -5/36 & -324/1027 & 1 \end{bmatrix} \text{ and } \mathbf{D} = \begin{bmatrix} 1 & & & & \\ & 15/8 & & & \\ & & 9/5 & & \\ & & & 1027/576 & \\ & & & & 5474/3081 \end{bmatrix}.$$

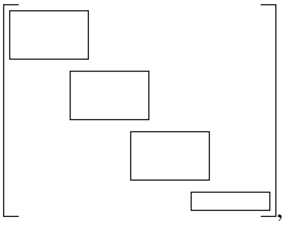
Once again, we emphasize that the zero entries need not be stored, for a banded matrix. That is, a matrix with k bands occupies about $k \cdot n$ units of memory, not n^2; for $n > 100$ and $k \leq 5$, this become significant, indeed crucial. A less obvious point (which we have made already but which we repeat here) is that $\mathbf{A}^{-1}$ is *dense* and takes $O(n^2)$ storage and costs $O(n^2)$ flops to use. We are thus in better position with a banded symmetric positive-definite matrix *not* to have the inverse explicitly! To compute $\mathbf{A}^{-1}\mathbf{b}$, just solve the sequence of problems

$$\begin{aligned} \mathbf{L}\mathbf{z} &= \mathbf{b} \\ \mathbf{D}\mathbf{w} &= \mathbf{z} \\ \mathbf{L}^T\mathbf{x} &= \mathbf{w} \end{aligned}$$

instead, for $O(n)$ cost in total. Savings are so great that we will even tolerate some instability and fix that by iterative refinement (see the next chapter, and Problem 6.6).

6.4 Block Structure

The third most commonly useful structure has entries occurring in distinct blocks, such as



which is "almost block diagonal" in that the blocks align on the diagonal, with overlap. Among other things, this type of matrix occurs in the numerical solution of boundary value problems for ODE (see Ascher et al. (1988) and Chap. 14).

A slightly different block idea occurs when matrices are naturally represented by the Krönecker product (tensor product), defined by

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2n}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & a_{m2}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix}.$$

Finally, the following partitioned matrix,

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & & & \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{A}_{23} & & \\ & \mathbf{A}_{32} & \mathbf{A}_{33} & \ddots & \\ & & \ddots & \ddots & \mathbf{A}_{n-1,n} \\ & & & \mathbf{A}_{n,n-1} & \mathbf{A}_{nn} \end{bmatrix},$$

which has been carved up into blocks, is block tridiagonal. Often the blocks are the same size, but this is not necessary. If we take, say, the matrix

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix},$$

then the blocks $\mathbf{A}_{11}$ and $\mathbf{A}_{21}$ must have the same column dimension, and the blocks $\mathbf{A}_{11}$ and $\mathbf{A}_{12}$ must have the same row dimension; in other words, the blocks must be such that the partition is *conformal.*

Matrix multiplication of conformally blocked matrices follows the same sort of rules as scalar matrix multiplication:

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{bmatrix},$$

where each product inside is itself a smaller matrix–matrix product. $\mathbf{B}_{11}$ has to have the same number of rows as $\mathbf{A}_{11}$ has columns, and so on, and these products are not, of course, commutative. Given this, we have block LU factoring and the Schur complement (see Appendix C.4):

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11} & \\ \mathbf{A}_{21} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{A}_{11}^{-1}\mathbf{A}_{12} \\ & \mathbf{S} \end{bmatrix},$$

where $\mathbf{S} = \mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$ is the Schur complement of $\mathbf{A}_{22}$ in $\mathbf{A}$; clearly, $\mathbf{A}_{11}$ has to be nonsingular for this to make sense (this is the block analog of a nonzero pivot).

Block algorithms have interesting data locality advantages and on some computers can perform significantly better when organized one way rather than another. However, we note that there can be severe compromises to numerical stability as a price.[3]

6.5 Other Structured and Sparse Matrices

In the context of polynomial interpolation and approximation, we often encounter the Vandermonde matrices, defined by $(\mathbf{V})_{ij} = \phi_j(\tau_i)$ for some polynomial basis $\phi_j(x)$. Obviously, the entries of this matrix are correlated. These can be inverted or

[3] Here, we only warn the reader. See Higham (2002, chap. 13) for a detailed introduction to these issues.

solved in $O(n^2)$ time, often stably in spite of notorious ill-conditioning. We have also encountered circulant matrices in Example 6.2. They have remarkable properties and can be easily solved using the Fourier transform.

Toeplitz and Hankel matrices (defined in the problems) similarly depend only on $O(n)$ parameters, and the Sylvester matrices consist of two stacked Toeplitz blocks, for instance,

$$\begin{bmatrix} f_3 & f_2 & f_1 & f_0 & \\ & f_3 & f_2 & f_1 & f_0 \\ g_2 & g_1 & g_0 & & \\ & g_2 & g_1 & g_0 & \\ & & g_2 & g_1 & g_0 \end{bmatrix}.$$

Such matrices arise in computing the GCD of $f(z) = f_3z^3 + f_2z^2 + f_1z + f_0$ and $g(z) = g_2z^2 + g_1z + g_0$. This is taken up further in Example 6.10. Many structured matrices including Sylvester matrices have "low displacement rank"(see Kailath and Sayed 1995). Low-displacement-rank matrices are determined by an $O(n)$ number of entries, and algorithms exist to take advantage of these correlations. For example, "diagonal-plus-rank-one" matrices are useful in several applications (see, e.g., Gemignani 2005); and there are yet more.

In contrast, a *general sparse matrix* has no apparent pattern. It is characterized by not having many nonzero entries; perhaps only a few percent of the entries of $\mathbf{A}$ are nonzero. MATLAB caters to this kind of matrix, where instead of storing entries contiguously, entries are recorded together with (i, j) indices, which necessitates careful bookkeeping by the computer. True advantage can be taken of sparsity and structure by using graph-theoretic ideas.[4]

Example 6.4. In MATLAB, we can generate random sparse matrices with the command `sprand`. For example, we can solve a random sparse $3{,}000 \times 3{,}000$ system by executing

```
A=sprand(3000,3000,0.1);
b=sprand(3000,1,0.2);
x=A\b;
res=b-A*x;
norm(res,inf)
```

which returns the residual

$$\|\mathbf{r}\|_\infty = 1.253702003323198 \cdot 10^{-13}.$$

The sparse random matrix here has about 10% nonzeros, so instead of $(3 \cdot 10^3)^2 = 9 \cdot 10^6$ entries, it has about $9 \cdot 10^5$ entries; MATLAB then quickly solves $\mathbf{Ax} = \mathbf{b}$ (for a random 3000 vector with about 600 nonzeros) with an accuracy in the residual of about 10^{-13}—all in time barely noticeable. ◁

[4] See Davis (2006), whose techniques are used under the hood in MATLAB and in many other problem-solving environments.

Consider another example in which sparsity can be used to our advantage by reordering matrices.

Example 6.5. Consider the example `bfwa398` from the Florida Sparse Matrix Collection.[5] Its spygraph is shown in Fig. 6.2. The matrix is not symmetric, though it looks as though it might be. Because it's not very large, we can actually do a $\mathbf{PA} = \mathbf{LU}$ factoring directly, although the resulting factors are *not very sparse*. The original matrix is about 2.3% nonzero and the rest zero. However, as we see in Fig. 6.3, each of the factors $\mathbf{L}$ and $\mathbf{U}$ is about 30% full. In this example, we could still solve the problem because it is of small enough dimension. However, for larger problems, such "fill-in" can make the factoring completely impractical.

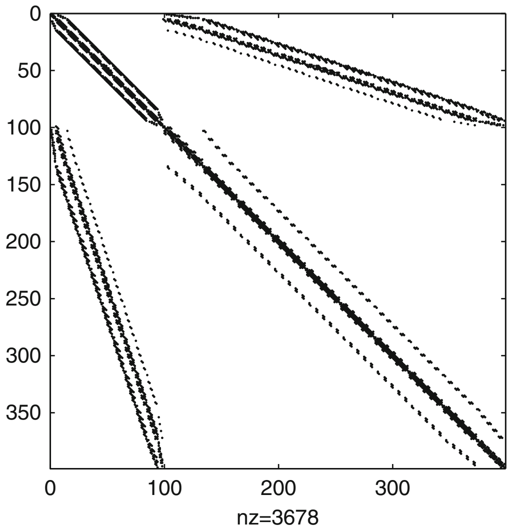


Fig. 6.2 The structure of the sparse matrix bfwa393 from the Florida Sparse Matrix Collection

For various classes of matrices, it can help significantly to reorder the variables and thus the columns. MATLAB has several routines to do so, such as `colamd`, which tries to reorder the variables to get an "approximate minimal degree" ordering. When we do this for this example, by executing

```
p = colamd( Problem.A );
[L,U,P] = lu( Problem.A(:,p) );
figure(3), spy( L, 'k' );
```

we get the factor shown in Fig. 6.4. ◁

Finally, we mention the *black-box* matrix. This is a matrix given only in procedure form, which perhaps all you are permitted to do with is call with a vector $\mathbf{v}$,

[5] Available at `http://www.cise.ufl.edu/research/sparse/matrices/Bai/bfwa398.html`.

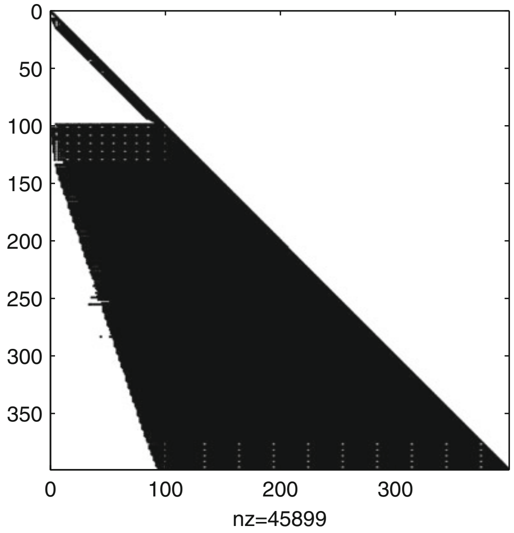


Fig. 6.3 The LU factoring of bfwa398 exhibits considerable fill-in. The factors are not sparse

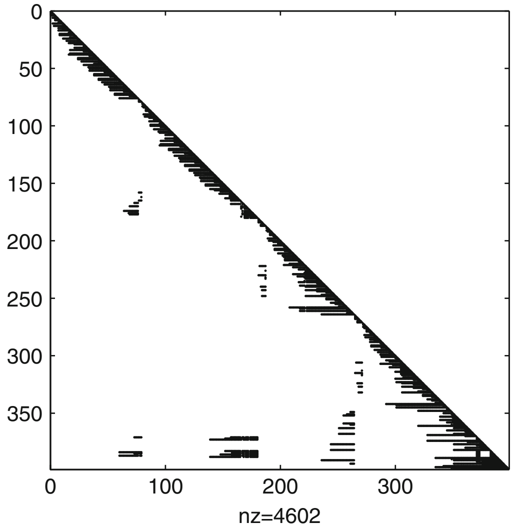


Fig. 6.4 Using `colamd` significantly reduces the fill-in for this example, bfwa398

and get back another vector $\mathbf{w}$ together with a certificate that $\mathbf{w}$ really is $\mathbf{Av}$ (an example of this is the Seneca function on page 272). You could, of course, probe the procedure n times with $\mathbf{v} = \mathbf{e}_1, \mathbf{v} = \mathbf{e}_2, \ldots, \mathbf{v} = \mathbf{e}_n$ in turn and recover the matrix: But if $n = 10^9$, perhaps you don't want to do that (even one call may take a while!).

What can be done with just a black box? We can take $\mathbf{v}_0$ at random (or, for example, if we're trying to solve $\mathbf{Ax} = \mathbf{b}$, we might try $\mathbf{v}_0 = \mathbf{b}$) and compute the sequence $\mathbf{v}_k = \mathbf{BB}(\mathbf{v}_{k-1})$, that is, $\mathbf{v}_k = \mathbf{A}\mathbf{v}_{k-1} = \mathbf{A}^k\mathbf{v}_0$. The sequence $\mathbf{v}_0, \mathbf{A}\mathbf{v}_0, \mathbf{A}^2\mathbf{v}_0, \ldots, \mathbf{A}^k\mathbf{v}_0$ is called a Krylov sequence, and if we can discover (somehow) a relation between these vectors, say

$$\alpha_0\mathbf{v}_0 + \alpha_1\mathbf{v}_1 + \cdots + \alpha_k\mathbf{v}_k = 0\,,$$

then (if $\alpha_0 \neq 0$), we have

$$\mathbf{A}\left(-\frac{\alpha_1}{\alpha_0}\mathbf{v}_0 - \frac{\alpha_2}{\alpha_0}\mathbf{v}_1 - \cdots - \frac{\alpha_k}{\alpha_0}\mathbf{v}_{k-1}\right) = \mathbf{v}_0 = \mathbf{b}\,,$$

and so we will have solved $\mathbf{Ax} = \mathbf{b}$. This trick works in exact arithmetic[6] because if $p(x)$ is the minimal polynomial of $\mathbf{A}$, then

$$p(\mathbf{A}) = \sum_{j=0}^{k} p_k\mathbf{A}^k = 0$$

(note that $p(x)$ divides the characteristic polynomial), and thus

$$p_0\mathbf{I} = -\sum_{j=1}^{k} p_j\mathbf{A}^j = \mathbf{A}(-\sum_{j=1}^{k} p_j\mathbf{A}^{j-1})\,.$$

As a result, we have effectively identified the inverse. To discover the α_ks, you only have to solve a $(k-1)\times(k-1)$ linear system.

Example 6.6. In Chap. 11, we will take up finite differences, which replace derivatives with sums of function values, somewhat like

$$f'(x) = \frac{f(x+h)-f(x-h)}{2h} + O(h^2)\,.$$

The use of finite differences to approximate derivatives generates sparse matrices from linear differential equations. MATLAB has several two-dimensional grids built into its example function `numgrid` and has an approximation to the Laplacian operator $\nabla^2 u$ called `delsq` that relates the value of $u(x,y)$ to the four surrounding values $u(x-h,y)$, $u(x+h,y)$, $u(x,y+h)$, and $u(x,y-h)$. At each node, therefore, there is an equation relating these five values. There may be many nodes and equations, but each one relates only a few variables. Therefore, the matrix will be sparse. Asking for a square 5 × 5 grid by executing `numgrid('S',5)` returns

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 4 & 7 & 0 \\ 0 & 2 & 5 & 8 & 0 \\ 0 & 3 & 6 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

[6] Using exact arithmetic, this is really practical only if $k \ll n$ (Chen et al. 2002).

The boundary conditions $u(x,y) = 0$ are applied on the edges, leaving only the values at the nine interior nodes unknown. The finite-difference Laplacian matrix is constructed and shown by

```
A = delsq( numgrid( 'S',5 ) );
spy(A,'k')
```

This graph is shown in Fig. 6.5. Larger square grids are sparser; if one makes the grids large enough, the spy picture looks as though it has a thick diagonal, but it's really banded. The matrix is symmetric.

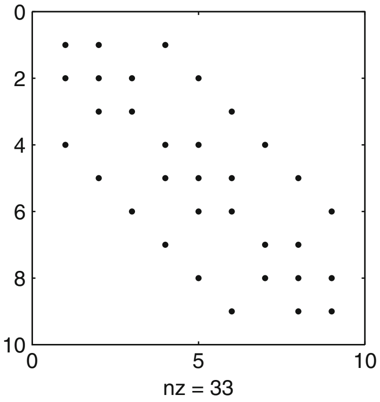


Fig. 6.5 The Laplacian approximation on a 5 × 5 square grid, numbered as in the text

Other built-in grids are available. See the documentation for `numgrid` for a listing. The `C` option cuts out a semicircular section of the square grid (as closely as it can, given the discretization). See Fig. 6.6. When we construct the Laplacian approximation on this 20 × 20 grid, we get a "widening" band matrix. See Fig. 6.7.

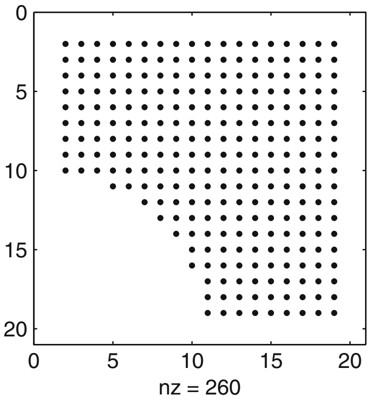


Fig. 6.6 The result of `spy( numgrid('C',20), 'k.' )`

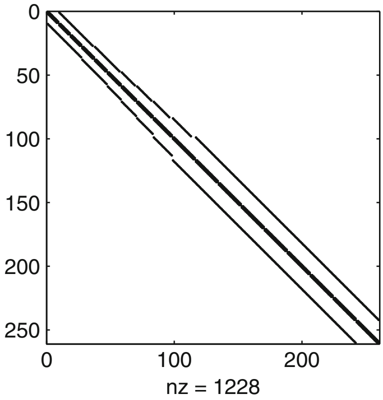


Fig. 6.7 The shape of the discrete Laplacian on the 20×20 grid in Fig. 6.6. Notice that the bands appear to spread apart as we move farther down the matrix

This entails "fill-in" if we perform Cholesky factoring: The original $\mathbf{A}$ has 1228 nonzero entries out of a possible 260×260, or about 1.8%. This count is from the call `nnz(A)`. If we form

```
L = lu( A );
```

we find that $\mathbf{L}$ has 3971 nonzero entries, meaning that about 5% of the possible entries are nonzero. This growth of the number of nonzero entries (a factor of about three for $n = 20$, and this factor grows linearly with n) is not terribly significant for this example, but can be a problem for other examples. Sometimes the grid numbering needs to be reordered (and can be reordered) to minimize the bandwidth or minimize the fill-in in the factoring. For an example of that, see the sparsity demo (type `doc sparsity` at the command line, or simply just `sparsity`) and also the documentation for `symamd`. ◁

6.6 Structured Backward Error and Conditioning

For dense matrices, we have concentrated on the relatively accessible normwise stability results, for instance, that the matrix $\hat{\mathbf{R}}$ computed by Householder QR factoring is the exact $\mathbf{R}$ of $\mathbf{A}+\mathbf{E}$ with $\|\mathbf{E}\| \le c_n\mu_M\|\mathbf{A}\|$. However, for matrices with particular structure—because they either are sparse or have correlated entries—some additional guarantees may be required. In this section, we examine such cases.

6.6.1 Structured Backward Errors and Componentwise Bounds

When they are available, componentwise bounds—for example, for triangular systems, $(\mathbf{U}+\mathbf{E})\hat{\mathbf{x}} = \mathbf{b}$ with $|e_{ij}| \le c_n\mu_M|u_{ij}|$—are much superior: Tiny but important entries u_{ij} are not potentially polluted by errors from larger coefficients, due to the fact that $\|\mathbf{U}\|$ could be very much larger than $|u_{ij}|$. This is a real difficulty with many problems, and in particular sometimes with the SVD: The smaller singular values may not be accurate, especially if $\sigma_k \leq \mu_M \sigma_1$. As a consequence, a componentwise backward error bound is a tighter constraint on an algorithm; it is easier to satisfy normwise bounds, since a given solution may require a larger ε to have a componentwise backward error of ε. In some structured cases, componentwise bounds can be obtained.

The componentwise backward error Higham (2002,122)

$$\omega_{\mathbf{E},\mathbf{f}}(\mathbf{y}) := \min\left\{\varepsilon \;\middle|\; (\mathbf{A}+\Delta\mathbf{A})\mathbf{y} = \mathbf{b}+\Delta\mathbf{b}, |\Delta\mathbf{A}| \leq \varepsilon\mathbf{E}, |\Delta\mathbf{b}| \leq \varepsilon f\right\} \tag{6.4}$$

uses absolute values on matrices and vectors to mean that the inequalities hold componentwise: For all i, j,

$$|\Delta a_{ij}| \leq \varepsilon e_{ij} \qquad \text{and} \qquad |\Delta b_i| \leq \varepsilon f_i \,.$$

e_{ij} and f_i are assumed to be nonnegative, and some, though not all, may be zero. In such a case, no change is permitted in that particular component, which may represent an *intrinsic* coefficient, in Stetter's terminology; If the coefficient has arisen from measurement, then it may more reasonably be permitted to change; Stetter calls such a coefficient *empiric*.

We have at hand all the machinery we need to prove the following version of a classic result in structured backward error, originally due to Oettli et al. (1965).

Theorem 6.1. *If the matrices* $\mathbf{A}$ *and* $\mathbf{E}$ *are elements of* $\mathbb{C}^{m\times n}$*, the vectors* $\mathbf{x}$*,* $\mathbf{b}$*, and* $\mathbf{f}$ *are in* $\mathbb{C}^n$*, and the elements of* $\mathbf{E}$ *and* $\mathbf{f}$ *are nonnegative, then the* minimum structured backward error ε *is*

$$\begin{aligned}\varepsilon &= \min\{t : (\mathbf{A}+\Delta\mathbf{A})\mathbf{x} = \mathbf{b}+\Delta\mathbf{b} \;\&\; |\Delta\mathbf{A}| \leq t\mathbf{E} \;\&\; |\Delta\mathbf{b}| \leq t\mathbf{f}\}\\ &= \max_{1\leq i\leq m} \frac{|r_i|}{\sum_{j=1}^n e_{i,j}|x_j| + f_i}\,,\end{aligned} \tag{6.5}$$

where $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$ *is the residual. If any* $r_i \neq 0$ *while the denominator is zero, the structured backward error is infinite.*

Proof. As pointed out in Oettli et al. (1965), we may consider this problem one row at a time. Taking the ith row of the rearrangement $\Delta\mathbf{A}\mathbf{x} - \Delta\mathbf{b} = \mathbf{r}$, we have

$$\sum_{j=1}^{n} \Delta a_{i,j} x_j - \Delta b_i = r_i \,. \tag{6.6}$$

We define the components of the $(n+1)$-vectors $\mathbf{c}$ and $\mathbf{d}$ as follows:

$$c_j = \begin{cases} {}^{\Delta a_{i,j}}\!/_{e_{i,j}}\,, & \text{if } e_{i,j} \neq 0\\ 0\,, & \text{otherwise}\end{cases} \qquad\qquad 1 \leq j \leq n$$

$$c_{n+1} = \begin{cases} {}^{\Delta b_i}\!/_{f_i}\,, & \text{if } f_i \neq 0\\ 0\,, & \text{otherwise}\,.\end{cases}$$

Then put $d_j = e_{i,j}x_j$ for $1 \le j \le n$ and $d_{n+1} = -f_i$. Equation (6.6) is then just $\mathbf{c}\cdot\mathbf{d} = r_i$, and we may use the Hölder inequality (see Appendix C.2) to find the minimum infinity norm of the vector $\mathbf{c}$. Hölder's inequality gives

$$|r_i| = |\mathbf{c}\cdot\mathbf{d}| \le \|\mathbf{c}\|_\infty \|\mathbf{d}\|_1$$

with equality iff each $|c_j| = \lambda$, a constant, and all the complex angles of $c_j d_j = \theta$, again a constant. That is,

$$\|\mathbf{c}\|_\infty \ge \frac{|r_i|}{\|\mathbf{d}\|_1},$$

and note that the right-hand side is independent of our choice of $\Delta a_{i,j}$ or Δb_i. Putting $c_j = |c_j|\exp(i\phi_j)$ (here i is the square root of -1, not to be confused with the row index) and $d_j = |d_j|\exp(i\psi_j)$, we must then have $\phi_j + \psi_j = \theta$. Notice that the ψ_j are given—they are the arguments of the x_j for $1 \le j \le n$, and $\psi_{n+1} = \pi$. Therefore, $c_j = |c_j|\exp(i(\theta - \psi_j))$. Using $|c_j| = \lambda$ and substituting into $\mathbf{c}\cdot\mathbf{d} = r_i$, we may solve for λ:

$$\lambda = \frac{e^{-i\theta} r_i}{\sum_{j=1}^{n} e_{i,j}x_j + f_i}.$$

If $r_i = 0$, we simply take the perturbations to be zero, regardless. If $r_i \ne 0$, then this expression for λ gives

$$\Delta a_{i,j} = \begin{cases} \dfrac{e^{-i\arg x_j} e_{i,j} r_i}{\sum_{j=1}^{n} e_{i,j}x_j + f_i} & \text{if } \sum_{j=1}^{n} e_{i,j}x_j + f_i \ne 0 \\ \infty & \text{otherwise} \end{cases}$$

$$\Delta b_i = \begin{cases} -\dfrac{f_i r_i}{\sum_{j=1}^{n} e_{i,j}x_j + f_i} & \text{if } \sum_{j=1}^{n} e_{i,j}x_j + f_i \ne 0 \\ \infty & \text{otherwise} \end{cases}.$$

By the Hölder inequality, each entry in $\mathbf{c}$ has the same magnitude, which means that with this choice we have the guaranteed minimum ε for the ith row. Taking the maximum over all rows gives us the theorem. ♮

Example 6.7. Consider the matrix

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & \varepsilon^{-1}+\varepsilon+2 \\ 0 & 1 & 0 & -2\varepsilon^{-1}-2\varepsilon-2 \\ 0 & 0 & 1 & \varepsilon^{-1}+\varepsilon+2 \end{bmatrix}.$$

This kind of matrix will be taken up in detail in Sect. 6.6.3. The characteristic polynomial of this matrix is $(\lambda-1)^2(\lambda-\varepsilon)(\lambda-1/\varepsilon)$, which is a "reciprocal polynomial" in that all its roots come in reciprocal pairs. The polynomial is monic, which means that its trailing coefficient (in the monomial basis, which we're using here) is also 1. Because this is a "companion matrix" (we explain more about this in the next section), the only matrix entries that can reasonably be perturbed are the ones coming from the coefficients of the polynomial, namely, the last column. Because this is a reciprocal polynomial, the $(1,4)$ entry should not be changed either. This means that the matrix $\mathbf{E}$ can be taken quite reasonably to be zero unless $j = 4$ and $i = 2$, 3, or 4. Taking $e_{i,j} = |a_{i,j}|$ in those cases also seems reasonable; we will allow only relatively small perturbations in those coefficients.

Let $\varepsilon = 3.27 \times 10^{-8}$. We choose as our right-hand side an approximate eigenvector for the double eigenvalue 1, namely,

$$\mathbf{b} = \begin{bmatrix} (-0.0158857 - 4.6244530i) \cdot 10^{-8} \\ 0.0048580 + 1.4142050i \\ -0.0044964 - 1.4142060i \\ (0.0147031 + 4.6244550i) \cdot 10^{-8} \end{bmatrix} .$$

When we solve this linear system using LU factoring, we get an answer near

$$\mathbf{x} = \begin{bmatrix} (-0.0142444 - 5.4172453i) \cdot 10^{-7} \\ 0.0052197 + 1.4142051i \\ -0.0048580 - 1.4142058i \\ (0.0158857 + 4.6244528i) \cdot 10^{-8} \end{bmatrix} .$$

The residual, computed in 30 digits precision but printed only to 2 significant figures here, is

$$\mathbf{r} = \begin{bmatrix} 0.0 + 0.0i \\ -2.0 \cdot 10^{-18} - 6.5 \cdot 10^{-16} i \\ -1.0 \cdot 10^{-17} - 3.2 \cdot 10^{-15} i \\ -1.6 \cdot 10^{-18} - 6.2 \cdot 10^{-16} i \end{bmatrix} .$$

Since the residual in the first component is exactly zero, no change need be made in the first row of $\mathbf{A}$ or $\mathbf{b}$ to accommodate it—which is lucky, as there are no coefficients that can be perturbed in the first row of $\mathbf{A}$. For the other rows, we have $\sum e_{i,j}|x_j| = |a_{i,4}||x_4|$, and so

$$\varepsilon_i = \frac{|r_i|}{|a_{i,4}||x_4| + |b_i|} ,$$

which gives 2.3×10^{-16}, 7.5×10^{-16}, and 4.4×10^{-16} for $i = 2$, 3, and 4. Thus, the maximum structured backward error is 7.5×10^{-16}.

If instead we distributed this error over *all* entries of $\mathbf{A}$, the overall size might be less (the optimum cannot be larger, of course), but it probably isn't much less; after all, it can't be much smaller than the unit roundoff, after scaling. ◁

Although the structured backward error must be at least as large as the unstructured backward error, a structured *condition number* may be much, much smaller than the unstructured condition number. Perturbations that don't change tiny but important components much may produce better-quality solutions. The following definition makes this idea more precise. Let

$$\mathrm{cond}_{\mathbf{E},\mathbf{f}}(\mathbf{A},\mathbf{x}) := \lim_{\varepsilon\to 0}\sup\left\{\frac{\|\Delta\mathbf{x}\|_\infty}{\varepsilon\|x\|_\infty}\,\middle|\,(\mathbf{A}+\Delta\mathbf{A})(\mathbf{x}+\Delta\mathbf{x}) = \mathbf{b}+\Delta\mathbf{b},\right.$$
$$\left.|\Delta\mathbf{A}| \le \varepsilon\mathbf{E}, |\Delta\mathbf{b}| \le \varepsilon f\right\}. \quad (6.7)$$

In the important case when $\mathbf{E} = |\mathbf{A}|$ and $\mathbf{f} = |\mathbf{b}|$, this can be shown to be within a factor of 2 of

$$\mathrm{cond}(\mathbf{A},\mathbf{x}) = \frac{\|\,|\mathbf{A}^{-1}|\,|\mathbf{A}|\,|\mathbf{x}|\,\|_\infty}{\|\mathbf{x}\|_\infty} \quad (6.8)$$

(a ratio of two vector norms, with a product by two positive matrices $|\mathbf{A}|$ and $|\mathbf{A}^{-1}|$ on top). See Problem 6.10. This is (up to an irrelevant constant factor) bounded by what is called the Skeel condition number:

$$\mathrm{cond}(\mathbf{A}) = \mathrm{cond}(\mathbf{A},\mathbf{e}) = \|\,|\mathbf{A}^{-1}|\,|\mathbf{A}|\,\|_\infty \le \kappa_\infty(\mathbf{A}). \quad (6.9)$$

The Skeel condition number is invariant under row scaling $\mathbf{DAx} = \mathbf{Db}$. Moreover, the gain can be great:

$$\min\{\,\kappa_\infty(\mathbf{DA})\,|\,\mathbf{D}\text{ is diagonal}\} = \mathrm{cond}(\mathbf{A}),$$

and if $\mathbf{D}_R$ equilibrates the rows of $\mathbf{A}$, then

$$\frac{\kappa_\infty(\mathbf{A})}{\kappa_\infty(\mathbf{D}_R)} \le \mathrm{cond}(\mathbf{A}) \le \kappa_\infty(\mathbf{A}).$$

We will see further discussion of this in the next chapter. The computation of $\mathrm{cond}(\mathbf{A})$ involves the computation of $|\mathbf{A}^{-1}|$, which is usually too expensive to do, especially if the matrix is sparse.

Example 6.8. In this example, we continue to examine the companion matrix of Example 6.7. The Skeel condition number of $\mathbf{A}$ is

$$\mathrm{cond}(\mathbf{A}) = \||\mathbf{A}||\mathbf{A}|^{-1}\|_\infty = 1.22\cdot 10^8,$$

which seems large, but the unstructured condition number is $3.74\cdot 10^{15}$, which is considerably larger. Multiplying the structured condition number by the structured backward error gives an error bound of $8.76\cdot 10^{-8}$, whereas using the unstructured condition number gives a bound of $O(1)$. The true error, found by computing the solution at 30 digits, is approximately $1.4\cdot 10^{-9}$, and the structured bound is within a factor of two of being tight. Thus, we can say with confidence that the solution of this linear system produced an exact solution of a nearby problem *with the same structure*. ◁

As noted already by Oettli et al. (1965), this backward error result can be used to guarantee that a *single* computed eigenvalue μ with corresponding eigenvector $\mathbf{x}$ is an exact eigenpair of a nearby structured matrix $\mathbf{A}+\Delta\mathbf{A}$ with $|\Delta\mathbf{A}| \le \varepsilon\mathbf{E}$. It cannot be used to show that *all* computed eigenpairs are the exact eigenpairs of the *same* nearby structured matrix; but often we are interested in only the largest eigenvalue, for example, and so this is a useful application of the theorem. To see this extension, suppose that $\mathbf{A}$, $\mathbf{x}$, and μ are given. Define $\mathbf{r} = \mu\mathbf{x} - \mathbf{A}\mathbf{x}$. Take $\mathbf{f} = \mathbf{0}$ and $\mathbf{E}$ as desired to preserve structure, and then consider the problem

$$\min\{t \mid \Delta\mathbf{A}\mathbf{x} = \mathbf{r} \;\&\; |\Delta\mathbf{A}| \le t\mathbf{E}\}\,.$$

Using the same reasoning as before, we find that the minimum is attained if $\Delta\mathbf{A}$ is chosen by using the witness vectors for Hölder's inequality as before, and the minimum value is

$$\varepsilon = \max_{1\le i\le n} \frac{|r_i|}{\sum_{j=1}^{n} e_{i,j}|x_j|}$$

if this is finite.

6.6.2 Structured Backward Error for Matrices with Correlated Entries

We have already seen in the previous chapter some computation of a "minimal" backward error for a matrix with correlated entries, namely, a symmetric matrix, where we used Lagrange multipliers to enforce the constraints and used least squares. The Oettli–Prager infinity norm-minimization approach can also be extended and results in a linear programming problem. We do something different here, and use the SVD. Specifically, we use the property of the SVD where solution of a column-rank-deficient matrix produces the minimal 2-norm solution. We demonstrate by example.

Example 6.9. We begin with the symmetric 2×2 linear system that we have seen before:

$$\mathbf{B} = \begin{bmatrix} 885781 & 887112 \\ 887112 & 888445 \end{bmatrix}.$$

In MATLAB using backslash to find the $\mathbf{x}$ that solves $\mathbf{B}\mathbf{x} = [0,1]^T$, we find $\mathbf{x} = [-8.871583031996495, 8.884913727761688]^T \cdot 10^5$, which is not the integer answer we were expecting. Nonetheless, the residual is zero, and so we know that there exists a relatively small perturbation $\Delta\mathbf{B}$ for which $(\mathbf{B}+\Delta\mathbf{B})\mathbf{x} = [0,1]^T$ exactly. Is there a symmetric such perturbation? We have already answered this in the affirmative using Lagrange multipliers in Problem 4.23; let us now try to use the SVD. The equations are $\delta b_{1,1}b_{1,1}x_1 + \delta b_{1,2}b_{1,2}x_2 = 0$ and $\delta b_{2,1}b_{2,1}x_1 + \delta b_{2,2}b_{2,2}x_2 = 0$, with the further equation $\delta b_{1,2} - \delta b_{2,1} = 0$, which we simply apply. This gives us two equations in three unknowns, which will then be column rank-deficient. Notice that the x_k will appear in the matrix entries! The 2×3 matrix is

$$10^{11}\begin{bmatrix} -7.882 & 7.882 & 0.0 \\ 0.0 & -7.870 & 7.870 \end{bmatrix},$$

and its singular values are both large. The SVD has a 2×2 matrix $\mathbf{U}$, a rectangular 2×3 matrix $\boldsymbol{\Sigma}$, and a 3×3 matrix $\mathbf{V}$. Using the SVD to solve this gives us the minimum 2-norm solution; since we are looking for *any* small perturbation that makes our residual actually zero, this works. When we do this, we find that our perturbed matrix is, to 20 digits,

$$\begin{bmatrix} 888444.99999999998316 & 887112.00000000001961 \\ 887112.00000000001961 & 885780.99999999999721 \end{bmatrix}$$

and that (computing to the appropriate internal precision) the residual of the MATLAB-computed $\mathbf{x}$ with *this* matrix is indeed zero. Moreover, this matrix, when entered into MATLAB, rounds to $\mathbf{B}$.

Now, the residual as computed in higher precision in MAPLE is $[0.323664 \cdot 10^{-4} - 0.198776 \cdot 10^{-4}]^T$. Using that, we solve the system using the SVD. This says a symmetric perturbation of relative size less than μ_M in each of the entries of $\mathbf{B}$ guarantees that the computed solution is exact for that perturbed system. Thus, we have used the SVD to compute a structured, *correlated*, backward error that verifies Theorem 4.3. ◁

Example 6.10. The matrix $\mathbf{A}$ below is a *Sylvester* matrix, which occurs in trying to find the greatest common divisor of two polynomials. If the polynomials $f(x) = f_0 + f_1x + f_2x^2 + f_3x^3$ and $g(x) = g_0 + g_1x + g_2x^2 + g_3x^3$ have a common zero, say x^*, then all of $f(x^*)$, $x^*f(x^*)$, $(x^*)^2f(x^*)$, and so on are zero, and so are $g(x^*)$, $x^*g(x^*)$, $(x^*)^2g(x^*)$, and so on. By taking enough powers to make a square matrix, we find that the following linear system of equations must hold: $\mathbf{SX} = 0$, where

$$\mathbf{S} = \begin{bmatrix} f_3 & f_2 & f_1 & f_0 & 0 \\ 0 & f_3 & f_2 & f_1 & f_0 \\ g_2 & g_1 & g_0 & 0 & 0 \\ 0 & g_2 & g_1 & g_0 & 0 \\ 0 & 0 & g_2 & g_1 & g_0 \end{bmatrix} \tag{6.10}$$

and $\mathbf{X} = [1, x^*, (x^*)^2, \ldots, (x^*)^4]^T$. A little thought shows that if f and g have any other common roots, there must be another null vector of the same form; and if the common root is a double root, then $[0, 1, 2x^*, \ldots, 4(x^*)^3]^T$ is in the nullspace. The rank deficit of $\mathbf{S}$ is the degree of the GCD, in fact.

What could it mean, though, to have only "approximate" common roots? This question has been the subject of quite a bit of research at least since 1988, mostly because of applications in computer-aided geometric design. We won't go into much detail in this example, but we will consider the following question. Suppose we have (by some means or other) identified some potential approximate common roots. Can we find 'minimal' perturbations in the coefficients, call them Δf_k and Δg_j, so that all the approximate common roots are *exact* common roots? That is, can we find the "nearest" $f+\Delta f$ and $g+\Delta g$ that have the specified GCD? The answer depends on what you mean by "nearest," but one answer could be to look for perturbations such that Δf_k was small compared to f_k, and similarly for g.

Accordingly, suppose our approximate common zeros were x_1 and x_2. Compute the residuals $f(x_1)$, $x_1 f(x_1)$, ..., $x_1^4 f(x_1)$, $g(x_1)$, $x_1 g(x_1)$, ..., and similarly for x_2. This gives 10 residuals apparently, but they are obviously dependent, and the SVD will pick that up automatically. Writing our equations for the perturbations, we have $\mathbf{Bd}=\mathbf{r}$, where the vector $\mathbf{d}$ contains our desired perturbations Δf_k, divided by our normalization constants f_k, and similarly for g. That is, we put $\Delta f_k = f_k \delta f_k$ and $\Delta g_k = g_k \delta g_k$, and use the relative variables instead. We will want to keep both f and g monic, so we insist that $\Delta g_2 = \Delta f_3 = 0$. Rearranging the equations so that all the δ's appear on the left-hand side [for example, with $x_1^2(g(x_1)+\Delta g(x_1))=0$], we get

$$g_2 {x_1}^4 + (g_1 + \delta g_1 g_1){x_1}^3 + (g_0 + \delta g_0 g_0){x_1}^2 = 0\,,$$

which we rearrange to get

$$g_0 x_1^2 \delta g_0 + g_1 x_1^3 \delta g_1 = -g_2 {x_1}^4 - g_1 {x_1}^3 - g_0 {x_1}^2\,.$$

Doing this for all 10 equations, we get the linear system $\mathbf{B}\delta = \mathbf{R}$, where

$$\mathbf{B} = \begin{bmatrix} f_0 x_1 & f_1 {x_1}^2 & f_2 x_1^3 & 0 & 0 \\ f_0 & f_1 x_1 & f_2 x_1^2 & 0 & 0 \\ 0 & 0 & 0 & g_0 x_1^2 & g_1 x_1^3 \\ 0 & 0 & 0 & g_0 x_1 & g_1 x_1^2 \\ 0 & 0 & 0 & g_0 & g_1 x_1 \\ f_0 x_2 & f_1 x_2^2 & f_2 x_2^3 & 0 & 0 \\ f_0 & f_1 x_2 & f_2 x_2^2 & 0 & 0 \\ 0 & 0 & 0 & g_0 x_2^2 & g_1 x_2^3 \\ 0 & 0 & 0 & g_0 x_2 & g_1 x_2^2 \\ 0 & 0 & 0 & g_0 & g_1 x_2 \end{bmatrix}.$$

The residuals $\mathbf{R}$ are just the negatives of the values of $x_1^j f(x_1)$ and similarly for x_2 and g:

$$\begin{bmatrix} -f_3x_1^4 - f_2x_1^3 - f_1x_1^2 - f_0x_1 \\ -f_3x_1^3 - f_2x_1^2 - f_1x_1 - f_0 \\ -g_2x_1^4 - g_1x_1^3 - g_0x_1^2 \\ -g_2{x_1}^3 - g_1{x_1}^2 - g_0x_1 \\ -g_2{x_1}^2 - g_1x_1 - g_0 \\ -f_3{x_2}^4 - f_2{x_2}^3 - f_1{x_2}^2 - f_0x_2 \\ -f_3{x_2}^3 - f_2{x_2}^2 - f_1x_2 - f_0 \\ -g_2{x_2}^4 - g_1{x_2}^3 - g_0{x_2}^2 \\ -g_2{x_2}^3 - g_1{x_2}^2 - g_0x_2 \\ -g_2{x_2}^2 - g_1x_2 - g_0 \end{bmatrix}.$$

These look complicated, but given numerical values for the coefficients of f and g, and putative common roots x_1 and x_2, they are just a numerical matrix (10×5) and a numerical vector, of length 10, which is really just values of the polynomial, times some constants. With numerical values input, the matrix is not rank 5, but rank 4 (one could predict that from the structure); thus, the least-squares problem is column rank-deficient. If the residuals are small (that is, the values of f and g are small at the putative common roots), then the solution (by the SVD) will also be small.

To make this analysis more concrete, we take $f = x^3 - 6.0x^2 + 11.0196x - 6.0588$ and $g = x^2 - 3.04x + 2.0503$, and we suppose that our approximate common roots are 1 and 2 (just to make it a bit simpler). We allow changes in all coefficients except the leading coefficients, and we look for small relative changes—that is, we take $\Delta f_k = f_k \delta f_k$ and similarly for g. Our matrix becomes

$$\begin{bmatrix} -6.0588 & 11.0196 & -6.0 & 0 & 0 \\ -6.0588 & 11.0196 & -6.0 & 0 & 0 \\ 0 & 0 & 0 & 2.0503 & -3.04 \\ 0 & 0 & 0 & 2.0503 & -3.04 \\ 0 & 0 & 0 & 2.0503 & -3.04 \\ -12.1176 & 44.0784 & -48.0 & 0 & 0 \\ -6.0588 & 22.0392 & -24.0 & 0 & 0 \\ 0 & 0 & 0 & 8.2012 & -24.32 \\ 0 & 0 & 0 & 4.1006 & -12.16 \\ 0 & 0 & 0 & 2.0503 & -6.08 \end{bmatrix}. \qquad (6.11)$$

The right-hand side is (printing the column vector on two lines horizontally to save space)

$$\begin{bmatrix} 0.0392 & 0.0392 & -0.0103 & -0.0103 & -0.0103 \\ 0.0392 & 0.0196 & 0.1188 & 0.0594 & 0.0297 \end{bmatrix},$$

and the entries are all modestly small. Thus, we expect that the structured backward error will also be small. Now, take the SVD, $\mathbf{B} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H$. The dimensions are 10×10 for $\mathbf{U}$, there are four nonzero entries in $\boldsymbol{\Sigma}$, and $\mathbf{V}$ is 10×5 (called the "thin" SVD). On using this factoring to solve the system, we find that our relative perturbations are

$$\begin{bmatrix} \delta f_0 \\ \delta f_1 \\ \delta f_2 \\ \delta g_0 \\ \delta g_1 \end{bmatrix} \doteq \begin{bmatrix} -0.00393 \\ 0.00298 \\ 0.00291 \\ -0.0245 \\ -0.0132 \end{bmatrix} .$$

These are all tolerably small. When we add these perturbations to f and g, we get polynomials whose roots are (for f), 1.00000000000000, 2.00000000000000, and 3.01748941581046, and for g just the first two. Thus, we have used the SVD to find a structured, correlated backward error. It's true that the correlations were linear (the change in one entry in the matrix was exactly the same as the change in some other entries of the matrix, in fact); otherwise, we would not have been able to use the SVD. However, you see that it can be done.

It is, of course, a nonlinear problem to find the putative common roots (here 1 and 2), but again it can be attacked at least initially with the SVD, or even just with the QR factoring. A very practical method uses Gauss–Newton iteration (see Zeng 2004), although the use of displacement rank in Zhi (2003) and, most recently, Bini and Boito (2010), seems the most promising theoretically. See Problem 4.31 for an introduction to the Gauss–Newton method for solving nonlinear optimization problems.

One final remark on the use of the SVD to find the putative common roots: If we take the SVD of the Sylvester matrix (6.10), with the numerical values of the coefficients used in this example, the singular values are [19.03, 8.250, 1.880, 0.004775, 0.0006711]. The fact that two of the singular values are small suggests that the degree of the GCD is two; the nearest rank-deficit-two matrix is a distance ${}^{0.004775}/_{19.03} \doteq 2.5 \cdot 10^{-4}$ away. But that finds the nearest matrix of rank-deficit-two, not the nearest *Sylvester* matrix of rank deficit two. However, this is often enough all by itself, and the approximate null vectors from $\mathbf{V}$ can give us decent putative common roots; alternatively, they can be used as starting points for Gauss–Newton iteration, which in effect computes a kind of structured SVD. Note also the discussion in Dahlquist and Björck (2008). ◁

6.6.3 *Structured Backward Error for Eigenvalue Problems*

One of the things we learn in a first-year algebra course is that, to find eigenvalues, we find the roots of the characteristic polynomial. Yet, "[a]lthough the characteristic equation is important in theory, it plays no role in practical eigenvalue computations."[7] In fact, it is quite practical (though somewhat more expensive than necessary) to use eigenvalues to find polynomial roots. In the last section, we talked about structured backward error for matrices with correlated entries. Prior to that, we looked at structured backward error for a single eigenvalue, using the Oettli–Prager theorem. Something can also be done about structured backward error for all eigenvalues simultaneously, at least for some classes of eigenproblems. We sketch one such example in this section and give another in Chap. 8.

6.6.3.1 Univariate Polynomials

Let us begin with univariate polynomial roots where the polynomial is expressed, as usual, in the monomial basis about $x = 0$:

$$p(x) = p_0 + p_1 x + p_2 x + \cdots + p_n x^n\,. \tag{6.12}$$

If $p_n \neq 0$, there are n roots in $\mathbb{C}$. We can find these by eigenvalues as follows. First, observe this fact:

Theorem 6.2. *Consider the matrix*

$$\mathbf{R} = \begin{bmatrix} 0 & 1 & & & & \\ 0 & 0 & 1 & & & \\ 0 & 0 & 0 & 1 & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ -p_0 & -p_1 & -p_2 & -p_3 & \cdots & -p_{n-1} \end{bmatrix} - x \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & p_n \end{bmatrix}.$$

This matrix has zero determinant when $p(x) = 0$. This is a companion matrix pencil.

Proof. One can show this either by direct expansion or by the Schur complement as below (see Appendix C). We have

$$\mathbf{R} = \left[\begin{array}{cccccc|c} -x & 1 & & & & & \\ & -x & 1 & & & & \\ & & -x & 1 & & & \\ & & & -x & \ddots & & \\ & & & & \ddots & 1 & \\ & & & & & -x & 1 \\ \hline -p_0 & -p_1 & -p_3 & -p_4 & \cdots & -p_{n-2} & -xp_n - p_{n-1} \end{array}\right]$$

and, by inspection,

[7] This is a quote from David S. Watkins, in Hogben (2006 chapter 43).

$$\mathbf{A}^{-1} = \begin{bmatrix} -1/x & -1/x^2 & \cdots & -1/x^{n-1} \\ & -1/x & \cdots & -1/x^{n-2} \\ & & \ddots & \vdots \\ & & & -1/x \end{bmatrix}.$$

As a result,

$$\det(\mathbf{C}_0 - x\mathbf{C}_1) = x^{n-1}\det(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}),$$

where $\mathbf{B} = [0, 0, \ldots, 0, 1]^T$, so $\mathbf{A}^{-1}\mathbf{B}$ is the final column of $\mathbf{A}^{-1}$, namely,

$$\left[-\frac{1}{x^{n-1}}, -\frac{1}{x^{n-2}}, \ldots, -\frac{1}{x}\right].$$

Thus, $\mathbf{C}\mathbf{A}^{-1}\mathbf{B}$ is

$$\begin{bmatrix} -p_0 & -p_1 & \cdots & -p_{n-2} \end{bmatrix} \begin{bmatrix} -1/x^{n-1} \\ -1/x^{n-2} \\ \vdots \\ -1/x \end{bmatrix} = \frac{p_0}{x^{n-1}} + \frac{p_1}{x^{n-2}} + \cdots + \frac{p_{n-1}}{x},$$

and, finally, $\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}$ is

$$-p_{n-1} - xp_n - \frac{p_0}{x^{n-1}} - \frac{p_1}{x^{n-2}} - \cdots - \frac{p_{n-2}}{x}$$

and $\det(\mathbf{C}_0 - x\mathbf{C}_1)$ is x^{n-1} times this or

$$-\left(p_0 + p_1 x + p_2 x^2 + \cdots + p_{n-2}x^{n-2} + p_{n-1}x^{n-1} + p_n x^n\right) = -p(x).$$

Thus, the determinant is zero when $p(x) = 0$, and we may find polynomial roots by eigenvalues. ♮

This is how MATLAB's `roots` function works. A matrix polynomial version is coded in `polyeig`. A series of interesting papers has been written on the structured numerical stability of this approach.[8] The reason, as deduced in Arnol'd (1971) and refined by others, is that perturbations tangent to the similarity orbit do not matter, to first order, and the other (normal) direction can be accounted for by small changes in the polynomial coefficients. We will discuss a simpler version of this further in subsequent chapters. In essence, the method is numerically stable: It produces pseudozeros, which are exact roots of polynomials with only slightly perturbed coefficients. That is, the unstructured backward error results from ordinary eigenvalue computations can be improved to show that the eigenvalues returned are not just exact eigenvalues of $\mathbf{C} + \Delta\mathbf{C}$ for some small but dense perturbation $\Delta\mathbf{C}$, but also exact

[8] To name two, Toh and Trefethen (1994) and Edelman and Murakami (1995).

roots of a polynomial $p+\Delta p$, where the changes to the coefficients are small. As usual, one has to worry about the conditioning of the polynomial roots with respect to such perturbations, but that is what we have just done in the previous chapter, with the theory of pseudospectra for matrix polynomials; for scalar polynomials, the theory in Chap. 3 suffices and one needs to compute the function $B(z)$.

6.6.3.2 Multivariate Polynomials

Similarly, multivariate systems of polynomials occur very frequently in applications. Two common problems consist of *finding all complex roots* and *finding all real roots*. The first problem is already difficult, and there may be many roots in $\mathbb{C}$. The second problem is harder to do efficiently: there may be many *fewer* real roots, and we don't want to waste time with the complex roots only to throw them away. For systems of polynomial equations, we may compute companion-like matrices for each variable, using what are known as *Gröbner bases* or *resultants* or *Bézoutians*. The companion-like matrices, which commute with each other, have a common set of eigenvectors. For each such eigenvector, the eigenvalues give the components of the roots.

Example 6.11. Consider the following system of equations:

$$
\begin{aligned}
x^2+y^2&=1\\
25xy-12&=0
\end{aligned}
$$

We can compute with MAPLE that

$$
\mathbf{X}=\begin{bmatrix} 0 & 0 & 1 & 0\\ 12/25 & 0 & 0 & 0\\ 1 & 0 & 0 & -1\\ 0 & 12/25 & 0 & 0\end{bmatrix}\quad\text{and}\quad \mathbf{Y}=\begin{bmatrix} 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 12/25 & 0 & 0 & 0\\ 0 & 1 & -12/25 & 0\end{bmatrix}.
$$

It can be shown that $\mathbf{XY}=\mathbf{YX}$, from which it follows that $\mathbf{X}$ and $\mathbf{Y}$ are simultaneously upper-triangularizable by orthogonal similarity.[9] Indeed, $\mathbf{XV}=\mathbf{V}\boldsymbol{\Lambda}_x$ and $\mathbf{YV}=\mathbf{V}\boldsymbol{\Lambda}_y$, where

$$
\mathbf{V}=\begin{bmatrix} 25/9 & 25/16 & 25/16 & 25/9\\ 5/3 & -5/4 & \frac{5}{4} & -5/3\\ 20/9 & -15/16 & 15/16 & -20/9\\ 1 & 1 & 1 & 1\end{bmatrix}
$$

and

$$
\boldsymbol{\Lambda}_x=\operatorname{diag}\left(\frac{4}{5},-\frac{3}{5},\frac{3}{5},-\frac{4}{5}\right)
$$

[9] See Corless et al. (1997) and, for example, Graillat and Trébuchet (2009).

$$\boldsymbol{\Lambda}_y = \operatorname{diag}\left(\frac{3}{5}, -\frac{4}{5}, \frac{4}{5}, -\frac{3}{5}\right).$$

Therefore, the four roots (x,y) are

$$\left(\frac{4}{5}, \frac{3}{5}\right) \quad \left(-\frac{3}{5}, -\frac{4}{5}\right) \quad \left(\frac{3}{5}, \frac{4}{5}\right) \quad \left(-\frac{4}{5}, -\frac{3}{5}\right),$$

and these are *all* the roots. See Fig. 6.8, which is not drawn to scale, in order to show the separate roots clearly. ◁

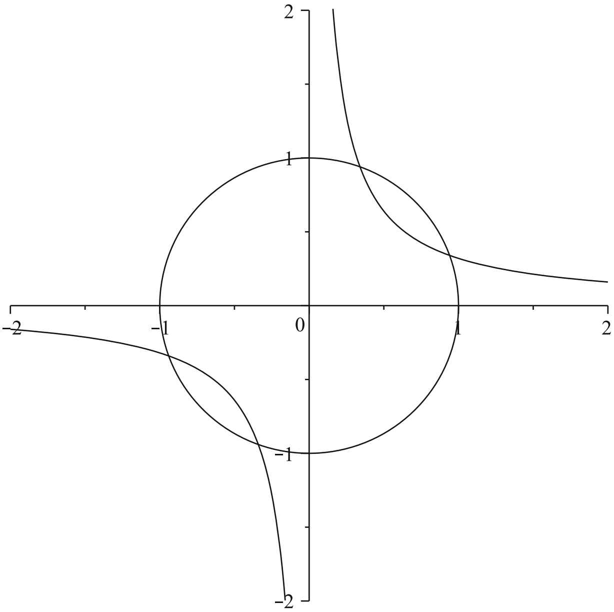


Fig. 6.8 The implicit curves $25xy = 12$ and $x^2 + y^2 = 1$

A similar method that uses a simpler construction, called the *Bézoutian*, runs as follows. Suppose $f(x,y)$ and $g(x,y)$ are given bivariate polynomials. One forms the *Cayley quotient* after introducing a new variable η:

$$C(x,\eta,y) = \frac{f(x,y)g(\eta,y) - f(\eta,y)g(x,y)}{x-\eta}.$$

Then, one notices that this quotient is, in fact, a polynomial since, considered as a univariate polynomial in x, the numerator is exactly divisible by $x-\eta$. Then, one notices that the resulting *trivariate* polynomial can be written as a quadratic form:

$$C(x,\eta,y) = [\phi_0(\eta), \phi_1(\eta), \ldots, \phi_m(\eta)]\, \mathbf{B}(y) \begin{bmatrix} \phi_0(x) \\ \phi_1(x) \\ \vdots \\ \phi_m(x) \end{bmatrix} .$$

Here the $\phi_j(x)$ are some polynomial basis, usually $\phi_j(x) = x^j$ in the literature. Finally, one notices that, at a *common zero* (x^*, y^*) of $f(x,y)$ and $g(x,y)$, it must be that $C(x^*, \eta, y^*) = 0$, and that means that the vector $[\phi_0(x^*), \phi_1(x^*), \ldots, \phi_m(x^*)]$ is an eigenvector of $\mathbf{B}(y^*)$. Solving this polynomial eigenproblem (for example, by linearization) identifies all possible such y^*. This is the method that is implemented in the MAPLE package `RootFinding[BivariatePolynomial]`.[10] This can be generalized using the Macaulay resultant to higher-dimensional systems. One difficulty with it is that it often introduces spurious roots, which must be examined and discarded.

Example 6.12. Take $f = x^2 + y^2 - 1$ and $g = 25xy - 12$ as before. When we form the Cayley quotient as above, we find that

$$\frac{f(x,y)g(\eta,y) - f(\eta,y)g(x,y)}{x - \eta} = [1, \eta]\, \mathbf{B}(y) \begin{bmatrix} 1 \\ x \end{bmatrix} ,$$

where

$$\mathbf{B}(y) = \begin{bmatrix} -25y & 12 \\ 12 & 25\left(y^2 - 1\right) y \end{bmatrix} .$$

This matrix polynomial is of degree 3 in y, and we may find a linearization—that is, a companion matrix for the matrix polynomial—simply by using a block version of the companion matrix pencil used earlier. All values of y that make $\mathbf{B}(y)$ singular will be eigenvalues of the matrix pencil $(\mathbf{A}_0, \mathbf{B}_0)$, where

$$\mathbf{A}_0 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & -12 \\ 0 & 0 & 0 & 0 & -12 & 0 \\ 1 & 0 & 0 & 0 & 25 & 0 \\ 0 & 1 & 0 & 0 & 0 & 25 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{B}_0 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 25 \end{bmatrix} .$$

The finite eigenvalues of this pencil are exactly the y-values that make $\mathbf{B}(y)$ singular, namely, $3/5$, $4/5$, $-3/5$, and $-4/5$. Once y has been found, the corresponding x can be found, indeed from the null vectors of $\mathbf{B}(y)$. ◁

[10] See Manocha (1994) and Shakoori (2008).

Finally, we remark that, to our knowledge, no structured backward error results have been proven for multivariate rootfinding by means of eigenvalue problems as sketched in this section. This material was included here partly for continuity reasons, and partly to point out a place where the reader may contribute.

6.7 Cauchy Matrices

We close this chapter with a discussion of a special-purpose algorithm for LU factoring of Cauchy matrices, that is, matrices $\mathbf{C}$ with entries

$$c_{ij} = \frac{1}{x_i + y_j}$$

for some fixed vectors $\mathbf{x}$ and $\mathbf{y}$ with nonnegative entries. These are matrices with *correlated entries* and are not sparse, but $O(n)$ parameters determine the $O(n^2)$ entries of the matrix. This class includes the infamous Hilbert matrix, with $x_i = y_i = i - {}^1\!/_2$. The Hilbert matrix is notoriously ill-conditioned in an unstructured sense, but the following algorithm allows an accurate computation of the LU factors of a Cauchy matrix.

```
function [ L, D, U ] = CauchyDecomp( x, y )
%CAUCHYDECOMP Accurate LDU factoring [1/(x(i)+y(j)]
%   From Jim Demmel's ISSAC talk 2001
%   See also N. J. Higham, Accuracy and Stability
%   in Numerical Analysis, 2nd ed, SIAM, 2002, p. 514
%   which points to Cho, Math Comp 2(104) 819--825 1968.
%   Cho and Higham give explicit O(n) formulas
%   for entries of L, D, and U.
n = length(x);

U = 1.0./(diag(x)*ones(n)+ones(n)*diag(y));
L = eye(n);
D = eye(n);

for k=1:n,
    D(k,k) = U(k,k);
    L(k:n,k) = U(k:n,k)/D(k,k);
    U(k,k:n) = U(k,k:n)/D(k,k);
    U(k+1:n,k) = zeros(n-k,1);
    for i=k+1:n,
        for j=k+1:n,
            U(i,j) = U(i,j)*(x(i)-x(k))*(y(j)-y(k))/(x(k)+y(j))/(
                x(i)+y(k));
        end
    end
end
% Shockingly accurate.
```

Why does this work? Consider partitioning the matrix $\mathbf{C}$ as

$$\begin{bmatrix} c_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{bmatrix}$$

with c_{11} being 1×1. Then the Schur complement gives (see Sect. C.4)

$$\begin{bmatrix} c_{11} & \\ \mathbf{C}_{21} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & c_{11}^{-1}\mathbf{C}_{12} \\ & \mathbf{C}_{22} - \mathbf{C}_{21}c_{11}^{-1}\mathbf{C}_{12} \end{bmatrix}$$

(this is really just LU factoring). Now, we do some analytic work to do the cancellations analytically, which dramatically improves the stability:

$$\frac{1}{x_i + y_j} - \frac{1}{x_i + y_1}(x_1 + y_1)\frac{1}{x_1 + y_j} = \frac{1}{x_i + y_j}\,\frac{(x_i - x_1)(y_j - y_1)}{(x_1 + y_j)(x_i + y_1)}.$$

Amazingly, this pattern persists, because every minor of $\mathbf{C}$ is a Cauchy matrix, and thus each entry in $\mathbf{D}$ is a ratio of determinants of such minors. This allows us to replace the central statement in the LU decomposition loop by

```
U(i,j) = U(i,j)*(x(i)-x(k))*(y(j)-y(k))/(x(k)+y(j))/(x(i)+y(k));
```

For the 150×150 Hilbert matrix, this gives

```
6.046669307767589e-180
```

in Matlab, in an eyeblink. Maple, using exact arithmetic, gets (in less than 12 s)

```
6.04666930776759510e-180
```

Well, really it gets a fraction, which we then approximate. So, we see that Matlab was wrong by 6 units in the 16th place, a relative error of nearly 10^{-15}. This is outstanding, given that $K_\infty(\mathbf{H}_{150})$ is about $4 \cdot 10^{227}$! For $n = 250$, Matlab takes 0.33 s to get the factoring, while Maple takes 79 s (only about 8 times what it took for $n = 150$; this is actually a remarkable performance on a tablet PC), which is about 180 times slower. Again, the relative error in d_{nn} was less than 10^{-15}; now, however, $d_{nn} \approx 2 \cdot 10^{-300}$, getting close to underflow! The (unstructured) condition number? $4.35 \cdot 10^{380}$, which does overflow in Matlab. So, with this special technique, we get an accurate (perfectly accurate!) answer to a question that involves a matrix so ill-conditioned that its unstructured condition number overflows!

However, we do remark that for solving $\mathbf{H}_n\mathbf{x} = \mathbf{b}$, we're still in vast trouble: errors in $\mathbf{b}$ will be magnified by this huge condition number, giving us no correct digits in $\mathbf{x}$ at all. But, the accurate $\mathbf{LDL}^T$ factoring is used for finding eigenvalues and singular values via special techniques, and this is useful.[11]

In conclusion, we have seen that it is important to consider the structure of matrices and to take advantage of it. Moreover, we have reached a very important lesson: "It is important to use the right condition number for the occasion" (Higham 2002 124). But as this last example shows, it is also important to understand the actual limitations imposed by the condition numbers in reference to the particular problems under consideration.

[11] For more on this, see Demmel and Koev (2005).

6.8 Notes and References

The literature on algorithms and applications of structured matrices is vast. A good entry point is Golub and van Loan (1996); another is the *Handbook of Linear Algebra* (Hogben 2006), especially Chap. 48. Asymptotically fast algorithms are discussed in Bini and Pan (1994). Analytical properties are discussed in Horn and Johnson (1990). The papers in the conference proceedings *Structured Matrices in Mathematics, Computer Science and Engineering* (Olshevsky 2001a,b) and *Fast Algorithms for Structured Matrices: Theory and Applications* (Olshevsky 2003a) are particularly valuable. The book Böttcher and Grudsky (1987) contains many results on spectra and pseudospectra of Toeplitz matrices, and they point out that the structured condition numbers of Toeplitz matrices grow roughly as quickly as the unstructured condition numbers do. Structured condition numbers are studied in Higham and Higham (1992), and structured backward error for Toeplitz systems is studied in Varah (1994). Together these results give the curious result that the forward error bound predicted by the product of the structured condition number and the structured backward error is larger than the product of the unstructured condition number and the unstructured backward error. This shows that structured error analysis is not always helpful.

The literature on general sparse matrices is equally vast, and the work of Davis and Hu (2011) has already been mentioned.

Problems

Warm up and Review

6.1. Verify that the structured condition number in Eq. (6.7) is invariant under row scaling.

6.2. Compute or estimate the growth of cond$(\mathbf{A})$ for the symmetric tridiagonal matrix of Problem 6.17 as $n \to \infty$ and compare it with $\kappa_2(\mathbf{A})$.

6.3. Download the Problem `ncvxqp9` from the Florida Sparse Matrix Collection. This symmetric matrix is quite large, with dimension $n = 16,554$ and has over 50,000 nonzero entries. LU factoring without reordering does not work on our tablet PC, but reordering helps. Reorder the matrix using `colamd` and spy on the result. Factor the reordered matrix and spy on the result. Estimate the structured condition number cond$(\mathbf{A})$.

6.4. Consider the "Filbert" matrix, with $A_{j,k} = {}^{1}\!/\!{}_{F_{j+k-1}}$, where $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ thereafter; that is, each entry of the Filbert matrix is the reciprocal of a Fibonacci number. Estimate the condition number of the $n \times n$ Filbert matrix (this problem is from Bill Gosper).

6.5. A variation, which we shall call the Cauchy–Filbert matrix, is defined as follows: $C_{j,k} = {}^1/_{(F_j+F_k-1)}$. Show that this can be written as a Cauchy matrix. Estimate its condition number, but show that the code of this chapter can factor it accurately.

6.6. Write a MATLAB program to solve tridiagonal systems by LU factoring without pivoting. Compare the speed and accuracy of your program with the built-in backslash operator of MATLAB on several large tridiagonal systems of your own choosing. Include both symmetric and nonsymmetric problems, diagonally dominant and nondominant problems in your test suite.

6.7. The `west0479` matrix can be loaded into MATLAB by the command `load west0479`. The matrix is sparse and unsymmetric. Graph the sparsity pattern using the `spy` command. Show that using $\mathbf{PA} = \mathbf{LU}$ factoring on this matrix is more costly than solving the system using the `colamd` reordering. Suppose the right-side vector $\mathbf{b}$ is all ones for this example. You may estimate the size of the forward error by computing the residual of your solution (using the original matrix, after reordering your solution).

6.8. Download in MATLAB format from the Florida Sparse Matrix Collection the matrix `memplus`, at `http://www.cise.ufl.edu/research/sparse/matrices/Hamm/memplus.html`. Once loaded, the data are contained in a structure named `Problem` and the matrix is available in the field `Problem.A`. Compare LU factoring for the original matrix versus the reordered matrix from `colamd`. Compare also the solution of the linear system $\mathbf{Ax} = \mathbf{b}$, where the right-hand side is available in the field `Problem.b` by use of the MATLAB backslash. Estimate the structured condition number.

6.9. A *stochastic matrix* has each column containing only nonnegative entries that sum to 1, representing probabilities. Alternatively, each row could contain only nonnegative entries that sum to 1; if both are true, the matrix is termed *doubly stochastic*. Therefore, in the first case, the row vector `ones(1,n)` is a left eigenvector with eigenvalue 1. Show that unstructured pseudospectra of stochastic matrices always have eigenvalues larger than 1 if $\varepsilon > 0$, and therefore one should consider only structured pseudospectra. For the 2×2 case,

$$\begin{bmatrix} \alpha & 1-\beta \\ 1-\alpha & \beta \end{bmatrix},$$

find the eigenvalues explicitly and show that structured pseudospectra correspond precisely to changes $\alpha(1+\delta_\alpha)$ and $\beta(1+\delta_\beta)$.

6.10. Prove that Eq. (6.7) implies that $\mathrm{cond}_{|\mathbf{A}|,|\mathbf{b}|}(\mathbf{A},\mathbf{x}) \leq 2\mathrm{cond}(\mathbf{A},\mathbf{x})$. Since the factor 2 is relatively unimportant, the Skeel condition number is therefore useful for understanding the effect of perturbations on structured systems.

6.11. Write down a formula for the *structured* condition number of the solution of a Cauchy linear system $\mathbf{Cz} = \mathbf{b}$, where the Cauchy matrix depends on the parameters x_i and y_j by $c_{i,j} = {}^1/_{(x_i+y_j)}$. That is, find a procedure to compute ${}^{\partial z}/_{\partial x_i}$ and similarly for y_j. Is the structured condition number that you get from this analysis much less than the unstructured condition number $\|\mathbf{C}^{-1}\|\|\mathbf{C}\|$? You may look at small-dimensional examples instead of a general analysis.

6.12. We once again return to Example 4.1, and note here that is an instance of the structured family of matrices

$$\mathbf{A}z = \begin{bmatrix} z & z+1 \\ z-1 & z \end{bmatrix},$$

which has $\det\mathbf{A} = 1$ for all z. Forming $\mathbf{B} = \mathbf{A}^H\mathbf{A}$, we have another matrix with correlated entries. The solution of $\mathbf{Bx} = [0,1]^H$ is a function of the parameter z. Is the numerical solution of this system by MATLAB, call it $\hat{\mathbf{x}}$, the exact solution of this linear system with a perturbed matrix $\mathbf{B}(z+\Delta z)$ for some Δz? If not, then the structured backward error (in this sense) is *infinite*.

6.13. Prove the Sherman–Morrison formula:

$$\left(\mathbf{A}+\mathbf{uv}^H\right)^{-1} = \mathbf{A}^{-1} + \frac{\mathbf{A}^{-1}\mathbf{uv}^H\mathbf{A}^{-1}}{1+\mathbf{v}^H\mathbf{A}^{-1}\mathbf{u}} \tag{6.13}$$

if $\mathbf{A}$ is nonsingular and $1+\mathbf{v}^H\mathbf{A}^{-1}\mathbf{u} \neq 0$. This is useful in computation if $\mathbf{Ax} = \mathbf{b}$ can be solved quickly, because then the related system $\left(\mathbf{A}+\mathbf{uv}^H\right)\mathbf{x} = \mathbf{b}$, which has a matrix related by a rank-1 update, can also be solved quickly.

6.14. This question was inspired by a discussion from Neumaier (2001, pp. 97–101).

1. Prove that if $\|\mathbf{I}-\mathbf{A}\| \le \beta < 1$, then $\mathbf{A}$ is nonsingular and

$$\|\mathbf{A}^{-1}\| \le \frac{1}{1-\beta}.$$

2. If $\mathbf{A}$ is nonsingular and $\mathbf{B}$ is a *singular* matrix such that $|\mathbf{B}-\mathbf{A}| \le \delta|\mathbf{A}|$, where $|\cdot|$ of a matrix, means that the inequality holds componentwise, that is, $|B_{ij}-A_{ij}| \le \delta|A_{ij}|$, and in particular means that if $A_{ij} = 0$ then so is B_{ij}, then prove that

$$\delta \ge \frac{1}{\kappa(\mathbf{D_1AD_2})},$$

where $\mathbf{D_1}$ and $\mathbf{D_2}$ are any nonsingular diagonal matrices, and $\kappa(\mathbf{A}) = \|\mathbf{A}\|\|\mathbf{A}^{-1}\|$ is the condition number of $\mathbf{A}$ in this norm. [Hint: If $\mathbf{B}$ is singular, then so is $\mathbf{A}^{-1}\mathbf{B}$; now consider $I-\mathbf{A}^{-1}\mathbf{B}$ and use the result of part (a).]
3. Finally, if $\mathbf{A}$ is well-conditioned, can it be close to a singular matrix?

Investigations and Projects

6.15. This investigation concerns companion matrices.

1. Recall that orthogonal polynomials satisfy a three-term recurrence relation of the form $x\phi_n(x) = \alpha_{n-1}\phi_{n+1}(x) + \beta_{n-1}\phi_n(x) + \gamma_{n-1}\phi_{n-1}(x)$ [with special cases happening for $x\phi_0(x)$ and possibly $x\phi_1(x)$]. By thinking about multiplying the vector $[\phi_0(x), \phi_1(x), \ldots, \phi_n(x)]^T$, write down a companion matrix for $p(x) = \sum_{k=0}^{n} c_k\phi_k(x)$. These are implemented in MAPLE (see `CompanionMatrix`, but don't look unless you're stuck). The original invention is due to Good (1961) and independently to Specht (1960).
2. Use the result of Problem 6.15 and the recurrence relation $xT_n(x) = (T_{n+1}(x) + T_{n-1}(x))/2$ for $n \geq 1$ to write down the companion matrix for polynomials expressed in the Chebyshev basis. Use your companion matrix to find the zeros of $T_5(x) + T_4(x) + T_3(x) + T_2(x) + T_1(x) + T_0(x)$. Compare with Boyd (2002). Are the computed eigenvalues the exact eigenvalues of a nearby polynomial? Give the changes in the Chebyshev coefficients explicitly.
3. Invent a companion matrix pencil for polynomials expressed in Bernstein–Bézier form. Again this is implemented in MAPLE, so if you get stuck, you can find an answer (but there is more than one way to do this, so perhaps you will find a better answer).

6.16. Consider the Rayleigh quotient iteration:

```
Require: $\mathbf{x}_0$ (or else random $\mathbf{x}_0$) and Hermitian matrix $\mathbf{A}$.
  for $i = 1, 2, \ldots$ until converged do
    $\mu_i := (\mathbf{x}_{i-1}^H \mathbf{A}\mathbf{x}_{i-1})/(\mathbf{x}_{i-1}^H \mathbf{x}_{i-1})$
    Solve $(\mathbf{A} - \mu_i \mathbf{I})\mathbf{x} = \mathbf{x}_{i-1}$.
  end for
  $\mathbf{x}_i := \mathbf{x}/\|\mathbf{x}\|$.
```

It is (nearly) everyone's favorite method for the symmetric eigenvalue problem, although the general-purpose codes use QR iteration. Given an approximate eigenvector $\mathbf{x}$ for a Hermitian matrix $\mathbf{A}$, you proved elsewhere that $\mu = \mathbf{x}^H\mathbf{A}\mathbf{x}/\mathbf{x}^H\mathbf{x}$ is the best 2-norm estimate for the eigenvalue corresponding to $\mathbf{x}$.

1. Prove that μ_i is an eigenvalue of $\mathbf{A} + \mathbf{E}$ with $\mathbf{E}$ small enough if at any stage $\|\mathbf{x}_{i-1}\|/\|\mathbf{x}\|$ is small enough.
2. Convergence of this iteration is supposed to be usually cubic, that is, $\mathbf{e}_{m+1} \propto \mathbf{e}_m^3$, for large enough m (while rounding errors don't dominate). Under what circumstances would you expect this to break down? Try to find an example.
3. The two-sided RQI needs initial approximations to $\mathbf{y}^H$ as well as $\mathbf{x}$, and it costs at least twice as much, but it can work for nonsymmetric matrices $\mathbf{A}$:

```
for $1, 2, \ldots$ until converged do
  $\mu_i := (\mathbf{y}_{i-1}^H \mathbf{A}\mathbf{x}_{i-1})/(\mathbf{y}_{i-1}^H \mathbf{x}_{i-1})$
  Solve $(\mathbf{A} - \mu_i \mathbf{I})\mathbf{x} = \mathbf{x}_{i-1}$
  $\mathbf{x}_i := \mathbf{x}/\|\mathbf{x}\|$
```

```
    Solve $\mathbf{y}^H(\mathbf{A} - \mu_i\mathbf{I}) = \mathbf{y}^H_{i-1}$
    $\mathbf{y}^H_i := \mathbf{y}^H/\|\mathbf{y}^H\|$
end for
```

Try this algorithm out on a nonsymmetric matrix, say the Clement matrix of order 10, using random $\mathbf{x}_0, \mathbf{y}^H_0$. Try a different matrix with complex eigenvalues, of your choice; choose complex $\mathbf{x}_0$ and $\mathbf{y}^H_0$ in that case.

6.17. Let $\mathbf{A}$ be the $n \times n$ symmetric tridiagonal matrix with constant diagonal 2 and unit sub- and superdiagonals. Consider its factoring without pivoting into $\mathbf{A} = \mathbf{LDL}^T$ with $\mathbf{D} = \mathrm{diag}(d_1, d_2, \ldots, d_n)$ and

$$\mathbf{L} = \begin{bmatrix} 1 & & & & \\ \ell_2 & 1 & & & \\ & \ell_3 & 1 & & \\ & & \ddots & \ddots & \\ & & & \ell_n & 1 \end{bmatrix}. \tag{6.14}$$

1. Show that $d_1 = 2$, $d_k = 2 - {}^1\!/_{d_{k-1}}$ for $2 \le k \le n$, and $\ell_k = {}^1\!/_{d_{k-1}}$.
2. Write a short, memory-efficient MATLAB script to solve $\mathbf{Ax} = \mathbf{b}$, given an n-vector $\mathbf{b}$ that you may overwrite with the solution. Use as little extra memory as is consistent with good practice. (Hint: Use an analytical formula for d_k.)
3. Use the formula

$$\lambda_k = b + 2(ac)^{1/2}\cos\left(\frac{k\pi}{n+1}\right), \quad 1 \le k \le n,$$

for the eigenvalues of a constant-tridiagonal matrix

$$\begin{bmatrix} b & c & & & \\ a & b & c & & \\ & a & b & c & \\ & & \ddots & \ddots & \ddots \end{bmatrix}$$

and the fact that this $\mathbf{A}$ is symmetric ($a = c$) to find a formula for the condition number $\kappa_2(\mathbf{A}_n) = {}^{\sigma_1}\!/_{\sigma_n}$. How does $\kappa_2(\mathbf{A}_n)$ grow as $n \to \infty$?
4. Find a formula for $\mathbf{L}^{-1}$ and thereby estimate $\kappa_1(\mathbf{L}) = \|\mathbf{L}\|_1\|\mathbf{L}^{-1}\|_1$.
5. Is this algorithm for solving $\mathbf{Ax} = \mathbf{b}$ numerically stable? (A full proof is not necessary: Use the preceding answers, together with numerical experiments, to justify your case.)

6.18. A Toeplitz matrix is constant along diagonals:

$$\begin{bmatrix} t_0 & t_1 & t_2 & t_3 & t_4 \\ t_{-1} & t_0 & t_1 & t_2 & t_3 \\ t_{-2} & t_{-1} & t_0 & t_1 & t_2 \\ t_{-3} & t_{-2} & t_{-1} & t_0 & t_1 \\ t_{-4} & t_{-3} & t_{-2} & t_{-1} & t_0 \end{bmatrix}. \tag{6.15}$$

A Hankel matrix (which is strongly related) is instead constant along antidiagonals:

$$\begin{bmatrix} h_4 & h_3 & h_2 & h_1 & h_0 \\ h_3 & h_2 & h_1 & h_0 & h_{-1} \\ h_2 & h_1 & h_0 & h_{-1} & h_{-2} \\ h_1 & h_0 & h_{-1} & h_{-2} & h_{-3} \\ h_0 & h_{-1} & h_{-2} & h_{-3} & h_{-4} \end{bmatrix} \tag{6.16}$$

These matrices arise in many application areas, including signal processing and sparse reconstruction.

1. Prove that if **T** is a Toeplitz matrix and **P** is the "anti-identity"

$$\begin{bmatrix} & & & & 1 \\ & & & 1 & \\ & & \cdot^{\cdot^{\cdot}} & & \\ & 1 & & & \\ 1 & & & & \end{bmatrix}$$

(also called an exchange matrix) having ones along its principal antidiagonal but otherwise zero, then $\mathbf{H} = \mathbf{PT}$ is a Hankel matrix. Similarly show that if **H** is Hankel, then $\mathbf{T} = \mathbf{PH}$ is Toeplitz.

2. Faster-than-$O(n^3)$ algorithms for solving Toeplitz (or Hankel) systems are widely known. Golub and van Loan (1996) discuss $O(n^2)$ algorithms and their normwise, unstructured stability for symmetric positive-definite Toeplitz matrices. Luk and Qiao (2003) discuss an $O(n^2 \log n)$ algorithm for computing the SVD of Hankel matrices. Heinig (2001) analyzes the stability of some (potentially FFT-based) algorithms. Bini and Pan (1994) discuss cost and complexity. Choose one or more of these. Read and discuss.

Chapter 7
Iterative Methods

Abstract This chapter looks at iterative methods to solve linear systems and at some alternative methods to solve eigenvalue problems. That is, we now look at *iteration instead of using a finite number of noniterative steps*. Iterative methods for solving eigenvalue problems are, of course, completely natural. We looked at power iteration and at the QR iteration in Chap. 5; here we look at some methods that *take advantage of sparsity or structure*. We also use *one pass of iterative refinement* to *improve structured backward error*. ◁

7.1 Iterative Refinement and Structured Backward Error

Let us begin with the simplest possible iterative method for solving a linear system. We first consider a 3×3 example that hardly needs iteration, but we will shortly extend to larger matrix sizes. So suppose we wish to solve

$$\begin{bmatrix} 4 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} .$$

The exact solution, which is easy to find by any method, is $\mathbf{x} = [5, -6, 5]/14$. Let us imagine that we don't know that, but that due to a prior computation, we do know that the matrix

$$\mathbf{B} = \begin{bmatrix} 2+\sqrt{3} & 1 & \\ 1 & 4 & 1 \\ & 1 & 4 \end{bmatrix}$$

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_7,

has the Cholesky factoring $\mathbf{LDL}^T$ with

$$\mathbf{L} = \begin{bmatrix} 1 & & \\ \alpha & 1 & \\ & \alpha & 1 \end{bmatrix},$$

$\alpha = {}^1\!/\!{\scriptstyle(2+\sqrt{3})}$, and $\mathbf{D} = \text{diag}(2+\sqrt{3}, 2+\sqrt{3}, 2+\sqrt{3})$. As a result, $\mathbf{B}^{-1} = \mathbf{L}^{-T}\mathbf{D}^{-1}\mathbf{L}^{-1}$ is easy to compute, or, more properly,

$$\mathbf{Bx} = \mathbf{b} \quad \Leftrightarrow \quad \mathbf{LDL}^T\mathbf{x} = \mathbf{b}$$

is easy to solve. Here, if we let $\mathbf{P} = \mathbf{B}^{-1}$ (at least in thinking about it, not in actually doing it), we have

$$\mathbf{PAx} = \mathbf{Pb} = \begin{bmatrix} 0.3847 \\ -0.4359 \\ 0.35898 \end{bmatrix}.$$

Notice that $\mathbf{PA}$ is nearly the identity, that is, $\mathbf{PA} = \mathbf{I} - \mathbf{S}$, where $\mathbf{S}$ is a matrix with small entries:

$$\mathbf{S} = \begin{bmatrix} -0.073216421430700 & 0 & 0 \\ 0.0206191045714862 & 0 & 0 \\ -0.00515477614287156 & 0 & 0 \end{bmatrix}.$$

Our equation has thus become

$$(\mathbf{I} - \mathbf{S})\mathbf{x} = \mathbf{Pb} = \begin{bmatrix} 0.3847 \\ -0.4359 \\ 0.35898 \end{bmatrix},$$

and we are left with the problem, seemingly as difficult, of solving a linear system with matrix $\mathbf{I} - \mathbf{S}$. However, we have made some progress, since we can use the smallness of $\mathbf{S}$ to solve the system by means of an iterative scheme. First, observe that $(\mathbf{I}-\mathbf{S})\mathbf{x} = \mathbf{Pb} = \mathbf{x}_0$ implies $\mathbf{x} = \mathbf{x}_0 + \mathbf{Sx}$. Hence, we can then write the following natural iteration:

$$\mathbf{x}_{k+1} = \mathbf{x}_0 + \mathbf{Sx}_k\,.$$

This is the *Richardson iteration*, which is about as simple an iterative method as it gets. Then we obtain

$$\mathbf{x}_1 = \mathbf{x}_0 + \mathbf{Sx}_0\,.$$

Similarly, we find that

$$\begin{aligned} \mathbf{x}_2 &= \mathbf{x}_0 + \mathbf{Sx}_0 + \mathbf{S}^2\mathbf{x}_0 \\ \mathbf{x}_3 &= \mathbf{x}_0 + \mathbf{Sx}_0 + \mathbf{S}^2\mathbf{x}_0 + \mathbf{S}^3\mathbf{x}_0. \end{aligned}$$

In general, the kth iteration results in

$$\mathbf{x}_k = \sum_{j=0}^{k} \mathbf{S}^j \mathbf{x}_0 \,.$$

This series converges if $\|\mathbf{S}^k\|$ goes to zero, which it does, exactly as for the geometric series, if there is a $\rho < 1$ for which $\|\mathbf{S}^k\| \le \rho^k$. In this case, $\|\mathbf{S}\| \le 0.01$. An obvious induction gives $\|\mathbf{S}^k\| \le \|\mathbf{S}\|^k \le (0.01)^k$ and so this iteration converges; indeed, already $\mathbf{x}_4$ is correct to four digits. Note that $\max|\lambda| \le \|\mathbf{S}\|$ in general, and it is very possible that $\max|\lambda| < 1$. In this case the powers eventually decay even though $\|\mathbf{S}\| > 1$. We will see examples shortly.

Before we look at larger matrices, let's look at this iteration in a different way. Using a matrix $\mathbf{P}$, which is close to the inverse of $\mathbf{A}$, we make the initial guess $\mathbf{x}_0 = \mathbf{Pb}$ (since $\mathbf{Ax} = \mathbf{b}$ then implies $\mathbf{x} \approx \mathbf{Pb}$). The residual resulting from this choice is

$$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0 = \mathbf{b} - \mathbf{APb} \,.$$

Since $\mathbf{0} = \mathbf{b} - \mathbf{Ax}$, we find that

$$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0 - (\mathbf{b} - \mathbf{Ax}) = \mathbf{Ax} - \mathbf{A}\mathbf{x}_0 = \mathbf{A}(\mathbf{x} - \mathbf{x}_0) = \mathbf{A}\Delta\mathbf{x} \,.$$

Thus, we see that $\Delta\mathbf{x} = \mathbf{x} - \mathbf{x}_0$ solves

$$\mathbf{A}\Delta\mathbf{x} = \mathbf{r}_0 \,.$$

Now, with this equation, we can use $\mathbf{P}$ as above and let $\mathbf{x}_1 - \mathbf{x}_0 = \mathbf{P}\mathbf{r}_0$. Then

$$\mathbf{x}_1 = \mathbf{x}_0 + \mathbf{P}\mathbf{r}_0 \,.$$

The process can clearly be repeated:

$$\begin{aligned}\mathbf{x}_2 &= \mathbf{x}_1 + \mathbf{P}\mathbf{r}_1 \\ \mathbf{x}_3 &= \mathbf{x}_2 + \mathbf{P}\mathbf{r}_2 \,,\end{aligned}$$

where $\mathbf{r}_2 = \mathbf{b} - \mathbf{A}\mathbf{x}_2$ and $\mathbf{r}_1 = \mathbf{b} - \mathbf{A}\mathbf{x}_1$ are the corresponding residuals. This process is called *iterative refinement*. Note that

$$\begin{aligned}\mathbf{x}_1 &= \mathbf{x}_0 + \mathbf{P}(\mathbf{b} - \mathbf{A}\mathbf{x}_0) = \mathbf{x}_0 + \mathbf{Pb} - \mathbf{PA}\mathbf{x}_0 = \mathbf{x}_0 + \mathbf{x}_0 - \mathbf{PA}\mathbf{x}_0 \\ &= \mathbf{x}_0 + (\mathbf{I} - \mathbf{PA})\mathbf{x}_0 = \mathbf{x}_0 + \mathbf{S}\mathbf{x}_0 \,,\end{aligned}$$

since $\mathbf{PA} = \mathbf{I} - \mathbf{S}$ in our earlier notation. Similarly, one obtains

$$\begin{aligned}\mathbf{x}_2 &= \mathbf{x}_1 + \mathbf{P}(\mathbf{b} - \mathbf{A}\mathbf{x}_1) = \mathbf{x}_0 + \mathbf{S}\mathbf{x}_0 + \mathbf{Pb} - \mathbf{PA}(\mathbf{x}_0 + \mathbf{S}\mathbf{x}_0) \\ &= \mathbf{x}_0 + \mathbf{S}\mathbf{x}_0 + \mathbf{x}_0 - (\mathbf{I} - \mathbf{S})(\mathbf{x}_0 + \mathbf{S}\mathbf{x}_0) \\ &= \mathbf{x}_0 + \mathbf{S}\mathbf{x}_0 + \mathbf{S}^2\mathbf{x}_0 \,,\end{aligned}$$

which is mathematically equivalent to what we had before and converges under the same conditions.

The matrix $\mathbf{P}$, our approximate inverse, is called a *preconditioner* (and its inverse is usually denoted $\mathbf{M}$). Probably the most important part of any iterative method is choosing the right preconditioner. For solving $\mathbf{Ax} = \mathbf{b}$, we need for $\mathbf{P}$ to allow fast evaluation of products $\mathbf{Pv}$ and simultaneously be close to $\mathbf{A}^{-1}$. Unfortunately, these goals are often in opposition. It is useful in practice to use even quite crude approximations to $\mathbf{A}^{-1}$ as preconditioners, though.

Let us illustrate the usefulness of this method. Suppose we want to solve $\mathbf{Ax} = \mathbf{b}$ and, moreover, suppose $\mathbf{A} = \mathbf{F}_n(\mathbf{I}+\mathbf{S})$, where

$$\mathbf{F}_n = \begin{bmatrix} 2+\sqrt{3} & 1 & & & \\ 1 & 4 & 1 & & \\ & 1 & 4 & 1 & \\ & & \ddots & \ddots & \ddots \end{bmatrix}$$

is $n \times n$ and $\mathbf{S}$, off its diagonal, is small [we will allow $s_{11} = (4-(2+\sqrt{3}))/(2+\sqrt{3})$ to be sort of big]. Then, let $\mathbf{P} = \mathbf{F}_n^{-1}$, although because $\mathbf{F}_n^{-1}$ is full, we never compute it. Instead, we note that by symmetric factoring, we have $\mathbf{F}_n = \mathbf{L}_n\mathbf{D}\mathbf{L}_n^T$, where

$$\mathbf{L}_n = \begin{bmatrix} 1 & & & \\ \alpha & 1 & & \\ & \alpha & 1 & \\ & & \ddots & \ddots \end{bmatrix}$$

and $\mathbf{D} = \mathrm{diag}(2+\sqrt{3}, 2+\sqrt{3}, \ldots, 2+\sqrt{3})$. Note that we won't compute $\mathbf{S}$, either. Instead, we solve the sequence of equations

$$\begin{aligned} \mathbf{L}_n\mathbf{z}_0 &= \mathbf{b} \\ \mathbf{D}_n\mathbf{y}_0 &= \mathbf{z}_0 \\ \mathbf{L}_n^T\mathbf{x}_0 &= \mathbf{y}_0 \end{aligned}$$

in $O(n)$ flops to get $\mathbf{x}_0$, by means of which we will use iterative refinement to get an accurate value of $\mathbf{x}$ as shown below:

```
for $k = 1, 2, \ldots$ do
    Compute $\mathbf{r}_{k-1} = \mathbf{b} - \mathbf{A}\mathbf{x}_{k-1}$
    % Now, we compute $\mathbf{x}_k - \mathbf{x}_{k-1} = \mathbf{P}\mathbf{r}_{k-1}$
    Solve $\mathbf{L}\mathbf{z}_k = \mathbf{r}_{k-1}$
    Solve $\mathbf{D}\mathbf{y}_k = \mathbf{z}_k$
    Solve $\mathbf{L}^T \Delta\mathbf{x}_k = \mathbf{y}_k$
    Let $\mathbf{x}_k = \mathbf{x}_{k-1} + \Delta\mathbf{x}_k$
end for
```

This is an iterative refinement formulation of the iteration. Because $\|\mathbf{S}\| \doteq 0.01$, 10 or so iterations of this process gets $\mathbf{x}$ accurate to most significant digits; and each iteration costs $O(n)$ flops. Thus, in $O(n)$ flops, we have solved our system. This is significantly better than the $O(n^3)$ cost for full matrices!

Note that $\mathbf{A}$ need not really be tridiagonal: It can have a few more entries here and there off the main diagonals, contributing to $\mathbf{S}$, if they're not too large. Even if there are lots of them, the cost of computing the residual is at most $O(n^2)$ per iteration, and if $\mathbf{S}$ is small, we will need only $O(1)$ iterations.

It's hard to overemphasize the importance of this seemingly trivial change from direct, algorithmic finite-number-of-steps solution to a convergent iteration, but most large systems are, in practice, solved with such methods. As Greenbaum notes,

> With a sufficiently good preconditioner, each of these iterative methods can be expected to find a good approximate solution quickly. In fact, with a sufficiently good preconditioner $\mathbf{M}$, an even simpler iteration method such as $\mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{M}^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}_{k-1})$ may converge in just a few iterations, and this avoids the cost of inner products and other things in the more sophisticated Krylov space methods (in Hogben 2006, p. 41–10)

(which highlights the importance of choosing $\mathbf{P}$ well). The iterative methods included in MATLAB are (for $\mathbf{Ax} = \mathbf{b}$)

- `bicg`—biconjugate gradient
- `bicgstab`—biconjugate gradient stabilized
- `cgs`—conjugate gradient squared
- `gmres`—generalized minimum residual
- `lsqr`—least squares
- `minres`—minimum residual
- `pcg`—preconditioned conjugate gradient
- `qmr`—quasiminimal residual
- `symmlq`—symmetric LQ

but there is no explicit program for iterative refinement, because it is so simple. See, for example, Olshevsky (2003b) for pointers to the literature, or perhaps Hogben (2006).

It was Skeel who first noticed that a single pass of iterative refinement could be used to improve the structured backward error. He noticed that computing the residual in the same precision (not twice the precision, which might not be easily available) gives the exactly rounded residual for $(\mathbf{A} + \Delta\mathbf{A})\mathbf{x} = \mathbf{b} + \mathbf{r}$ for some $|\Delta\mathbf{A}| \leq O(\mu_M)|\mathbf{A}|$. That is, the computed residual is the exact residual for only $O(\mu_M)$ relative backward errors in $\mathbf{A}$, preserving structure. Notice that the computed solution $\mathbf{x}$ usually comes only with a *normwise* backward error guarantee: It is the correct solution to $(\mathbf{A} + \Delta\mathbf{A})\mathbf{x} = \mathbf{b} + \Delta\mathbf{b}$ with $\|\Delta\mathbf{A}\| = O(\mu_M\|\mathbf{A}\|)$ and $\|\Delta\mathbf{b}\| = O(\mu_M\|\mathbf{b}\|)$, which does not preserve structure. A single pass of iterative refinement can, if the condition number of $\mathbf{A}$ is not too large, improve this situation considerably. Let $\mathbf{x}_1 = \mathbf{x} + \Delta\mathbf{x}$, where

$$\mathbf{A}(\Delta\mathbf{x}) = \mathbf{r}\,.$$

Then solving this system gives us, more nearly, a solution of the same sort of problem.

The following argument, though not "tight," gives some idea of why this is so. Suppose we have approximately solved $\mathbf{Ax} = \mathbf{b}$ and found a computed solution, which we will call $\mathbf{x}_0$. Then, on computing the residual $\mathbf{r}_0 = \mathbf{b} - \mathbf{Ax}_0$ in single precision, we know that we have found the exact solution of

$$(\mathbf{A} + \Delta\mathbf{A}_0)\,\mathbf{x}_0 = \mathbf{b} - \mathbf{r}_0\,,$$

where $|\Delta\mathbf{A}_0| \leq c\mu_M|\mathbf{A}|$ and c is a small constant that depends linearly on the dimension n. Notice that the $\Delta\mathbf{A}_0$ is componentwise small. The working-precision residual $\mathbf{r}_0$ is included (it might not be very small), and what this statement says is merely that we have an accurate residual for a closely perturbed system. How small is $\mathbf{r}_0$? It is easy to see that, normwise,

$$\begin{aligned}\|\mathbf{r}_0\| &\doteq \rho\|\mathbf{A}\|\,\|\mathbf{A}^{-1}\|\,\|\mathbf{b}\|\mu_M\\ &\doteq \rho\,\kappa(\mathbf{A})\|\mathbf{b}\|\mu_M\,,\end{aligned} \tag{7.1}$$

at most (being sloppy with constants, though). ρ is called a growth factor. Now we suppose that in solving $\mathbf{A}\Delta\mathbf{x} = \mathbf{r}_0$ in the same approximate fashion (call the solution $\Delta\mathbf{x}_0$), we get the same approximate growth, so that the residual in *this* equation can be written

$$(\mathbf{A} + \Delta\mathbf{A}_1)\,\Delta\mathbf{x} = \mathbf{r}_0 - \mathbf{s}_0\,,$$

where again the perturbation $\Delta\mathbf{A}_1$ is small componentwise compared to $\mathbf{A}$, and $\mathbf{s}_0$ is the residual that we could compute using working precision in the update equation:

$$\mathbf{s}_0 = \mathbf{r}_0 - \mathbf{A}\Delta\mathbf{x}_0\,.$$

Our "similar growth" assumption says that $\|\mathbf{s}_0\| \doteq \rho\,\kappa(\mathbf{A})\|\mathbf{r}_0\|$. This will be, roughly speaking, $\rho^2\kappa(\mathbf{A})^2\|\mathbf{b}\|\mu_M^2$ and might, if we are lucky, be quite a bit smaller. Adding together the two equations, we find that

$$\begin{aligned}(\mathbf{A} + \Delta\mathbf{A}_0)\,(\mathbf{x}_0 + \Delta\mathbf{x}_0) &= \mathbf{b} + (\Delta\mathbf{A}_0 - \Delta\mathbf{A}_1)\,\Delta\mathbf{x}_0 - \mathbf{s}_0\\ &= \mathbf{b} + O(\mu_M^2)\,,\end{aligned}$$

where we have suppressed the $\rho^2\kappa^2(\mathbf{A})$ and the dependence on $\kappa(\mathbf{A})$ from the other small term in the order symbol. This loose argument leads us to expect that a single pass *ought* to give us nearly the exact solution to a perturbed problem where the perturbation is componentwise small.

Of course, it takes more effort to establish in detail that it actually does so under many circumstances, and to describe exactly what those circumstances are. We can easily see in the above argument though that if the condition number of $\mathbf{A}$ or the growth factor ρ or both are "too large," there will be trouble. Full details of a much tighter argument are in Skeel (1980).

Example 7.1. This idea helps in coping with examples where the residual is unacceptably large. This can happen even with well-scaled matrices (in theory, though as we have discussed it is almost unheard of in practice). Consider the family of matrices shaped like the following (we show the $n = 6$ case):

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 & 0 \\ -1 & -1 & -1 & 1 & 0 & 0 \\ -1 & -1 & -1 & -1 & 1 & 0 \\ -1 & -1 & -1 & -1 & -1 & 1 \end{bmatrix} . \tag{7.2}$$

This well-known example has a growth factor for Gaussian elimination with partial pivoting (although pivoting doesn't actually happen because it is arranged that the pivots are already in the right place) that is as bad as possible: The largest element in $\mathbf{U}$ where $\mathbf{A} = \mathbf{LU}$ is $2^n + 1$. The condition number of the matrix is quite reasonable, however; it is only 33 or so when $n = 32$. But the solution with GEPP is not acceptable, without iterative refinement, as we will see. As proved in Skeel (1980), a single pass of iterative refinement is enough to stabilize the algorithm in the strong sense discussed above.

Suppose we take $\mathbf{b}$ to be the vector $\mathbf{v}_n$ corresponding to the smallest singular value of $\mathbf{A}$. The choice of $\mathbf{b}$ doesn't really matter very much, though this choice is especially cruel. When we compute (for $n = 32$) the solution of $\mathbf{Ax} = \mathbf{b}$, we should get $\mathbf{u}_n$, the final vector of the $\mathbf{U}$ matrix from the SVD. Call our computed solution $\mathbf{x}_0$. We compute the residual $\mathbf{r}_0 = \mathbf{b} - \mathbf{Ax}_0$, using the same 15-digit precision used to compute $\mathbf{x}_0$. The norm of $\mathbf{r}_0$ is about 10^{-9}, and thus the nearest matrix $\mathbf{A} + \Delta\mathbf{A}$ for which $\mathbf{x}_0$ really solves the problem is about the same distance away, componentwise. If we now solve $\mathbf{A}\Delta\mathbf{x} = \mathbf{r}$ and put $\mathbf{x}_1 = \mathbf{x}_0 + \Delta\mathbf{x}$, then when we compute the residual again, we find that $\|\mathbf{r}_1\|_\infty$ is about 10^{-17}. This produces an entirely satisfactory backward error.

For $n = 64$, the situation is much worse, at the beginning. The zeroth solution has a residual with infinity norm nearly 1; that is, almost no figures in the solution are correct. A single pass of iterative refinement gives $\mathbf{x}_1$ with $\|\mathbf{r}_1\|_\infty \doteq 1.22 \cdot 10^{-13}$, 13 orders of magnitude better. The 2-norm condition number of the matrix is only about 56.8, mind, and the ∞-norm condition number is 128. The Skeel condition number (see Eq. (6.9)) $\mathrm{cond}(\mathbf{A}) = \||\mathbf{A}^{-1}||\mathbf{A}|\|_\infty$ is not very different, being very close to 66. However, the structured condition number *for this* $\mathbf{x}$ is quite a bit smaller:

$$\mathrm{cond}(\mathbf{A}, \mathbf{x}) = \frac{\||\mathbf{A}^{-1}|\,|\mathbf{A}|\,|\mathbf{x}|\,\|_\infty}{\|\mathbf{x}\|_\infty} \doteq 5.548\,.$$

Thus, for $n = 64$, we can expect nearly 13 figures of accuracy in $\mathbf{x}_1$, because the residual is so small. ◁

Remark 7.1. We should point out that $|\mathbf{A}|$ does not commute with $|\mathbf{A}^{-1}|$ in general, and in particular does not commute for this example. The Skeel condition number uses the inverse first. ◁

7.2 What Could Go Wrong with an Iterative Method?

Let us now return to the iterative idea itself, and no longer think about the effects of just one pass, but rather now think about what happens if many iterations are needed. Indeed, thousands of iterations are common in some applications. The basic theoretical question is now: when does $\mathbf{S}^k \to 0$, and how fast does it do so? A theorem of eigenvalues, $\mathbf{S}^k \to 0$ if all eigenvalues have $|\lambda| \le \rho < 1$, seems to characterize things completely. However, as we saw in Sect. 5.5.2, pseudospectra turn out to play a role for nonnormal $\mathbf{S}$. There are other methods to look at this problem, and there is an extensive discussion in Higham (2002, chapter 18). We content ourselves here with an example.

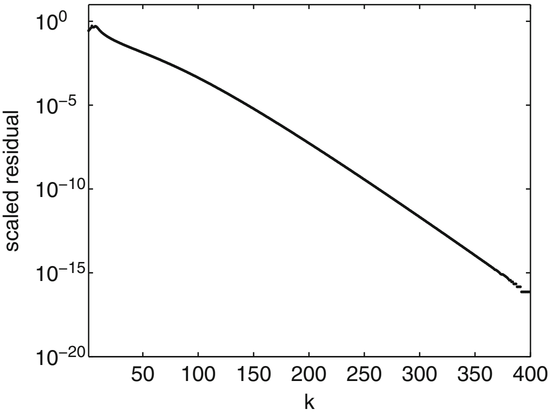


Fig. 7.1 Scaled residuals for the Richardson iteration solution of a nonnormal matrix with $n = 5$. We see fairly monotonic convergence

Example 7.2. Suppose that $\mathbf{A} = \mathbf{I} - \mathbf{S}$, where $\mathbf{S}$ is bidiagonal, with all diagonal entries equal to $8/9$ and all entries of the first superdiagonal equal to -1. This is similar to the example matrix that was used in Sect. 5.5.2. Now, we wish to solve $\mathbf{Ax} = \mathbf{b}$, where, say, $\mathbf{b}$ has all entries equal to 1. Because all eigenvalues of $\mathbf{S}$ are less than 1 in magnitude, we know that the series $\mathbf{I} + \mathbf{S} + \mathbf{S}^2 + \cdots$ converges. Moreover, we know that ultimately the error goes to zero like "some constant" times $(8/9)^k$, and that $k = 400$ gives $(8/9)^{400} \doteq 1 \times 10^{-21}$. Therefore, the Richardson iteration

$$\mathbf{x}_{k+1} = \mathbf{b} + \mathbf{S}\mathbf{x}_k$$

should converge to the reference solution. Incidentally, the reference solution has $x_n = 9$, $x_j = O((9/8)^{n-j})$ for $j = n-1, \ldots, 1$ by back substitution. This exponential growth in the solution suggests that we should evaluate the quality of our solution by examining the *scaled* residual,

$$\delta = \frac{\|\mathbf{b} - \mathbf{Ax}\|}{\|\mathbf{A}\|\|\mathbf{x}\|}\,.$$

We will use the kth iterate to scale the residual of the kth solution in the figures below.

Because the pseudospectrum of this matrix (when the dimension is large) pokes out into the region $|\lambda| > 1$—that is, the pseudospectral radius ρ_ε of Eq. (5.13) is larger than 1—we expect that this iteration will encounter trouble for large dimensions. In other words, the "constant" that we hid under the blanket called "some constant" in the previous discussion actually grows exponentially with the dimension n. While it is constant for any given iteration, the size of the constant gets ridiculously large. In Problem 7.5, you are asked to give an explicit lower bound, confirming this. Thus, as might be expected, the iteration works quite well for a 5×5 matrix, as shown in Fig. 7.1. Also, as predicted, our expectation of trouble is confirmed by an 89×89 matrix, as shown in Fig. 7.2. ◁

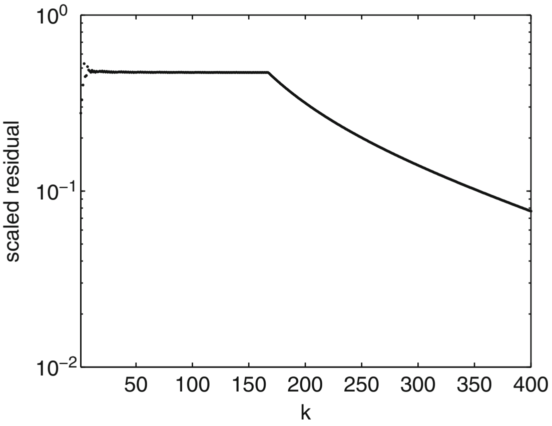


Fig. 7.2 Scaled residuals for the Richardson iteration solution of a nonnormal matrix of dimension 89×89. Convergence is very slow, which would be unexpected if we were not aware of the pseudospectra of the matrix $\mathbf{S}$

7.3 Some Classical Variations

In this section, we look at a few variations of the iterative method we have discussed thus far, namely, Jacobi iteration, Gauss–Seidel iteration, and successive overrelaxation (SOR).

Let us begin with *Jacobi iteration*. Take $\mathbf{P} = \mathbf{D}^{-1}$, the inverse of the diagonal part of the matrix (so, write the matrix as $\mathbf{D}+\mathbf{E}$). Then, mathematically, $\mathbf{PA} = \mathbf{D}^{-1}\mathbf{A}$ and $\mathbf{S} = \mathbf{I} - \mathbf{D}^{-1}\mathbf{A}$ is pretty simple, but unless the off-diagonal elements of $\mathbf{A}$ are small compared to $\mathbf{D}$, this won't converge: $\mathbf{I} - \mathbf{D}^{-1}\mathbf{A}$ has *only* off-diagonal elements, $-{}^{a_{ij}}/_{a_{ii}}$, and we want (ideally) $\|\mathbf{S}\| < 1$. As an iteration to solve $\mathbf{Ax} = \mathbf{b}$, we proceed as follows. $\mathbf{Ax} = \mathbf{b}$ is equivalent to $(\mathbf{D}+\mathbf{E})\mathbf{x} = \mathbf{b}$. Therefore,

$$\begin{aligned}
\mathbf{D}\mathbf{x} &= \mathbf{b} - \mathbf{E}\mathbf{x} \\
\mathbf{x}_{n+1} &= \mathbf{D}^{-1}(\mathbf{b} - \mathbf{E}\mathbf{x}_n) \\
&= \mathbf{x}_n - \mathbf{x}_n + \mathbf{D}^{-1}(\mathbf{b} - \mathbf{E}\mathbf{x}_n) \\
&= \mathbf{x}_n + \mathbf{D}^{-1}(\mathbf{b} - \mathbf{D}\mathbf{x}_n - \mathbf{E}\mathbf{x}_n) \\
&= \mathbf{x}_n + \mathbf{D}^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}_n),
\end{aligned}$$

which is the Jabobi iteration.

The Gauss–Seidel method is also worth considering. As Strang (1986, p. 406) said, "[T]his is called the Gauss–Seidel method, even though Gauss didn't know about it and Seidel didn't recommend it. Nevertheless it is a good method." Take $\mathbf{P} = \mathbf{L}^{-1}$, where $\mathbf{L}$ is the lower-triangular part of $\mathbf{A}$, including the diagonal:

$$\mathbf{L} = \begin{bmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ \vdots & & \ddots & \\ a_{n1} & a_{n2} & \cdots & a_{nn}. \end{bmatrix}$$

The iteration demands, for $\mathbf{A} = \mathbf{L} + \mathbf{U}$, that we solve

$$\mathbf{L}\mathbf{x}_{k+1} = \mathbf{b} - \mathbf{U}\mathbf{x}_k$$

for $\mathbf{x}_{k+1}$ or, alternatively, that use the map

$$\mathbf{x}_{k+1} = \mathbf{L}^{-1}\mathbf{b} - \mathbf{L}^{-1}\mathbf{U}\mathbf{x}_k$$

(at least in theory—in practice, we can write this as a simple iteration, reusing the same vector $\mathbf{x}$ as we go so; it uses less storage than Jacobi iteration). Because $\mathbf{L}$ is a better approximation to $\mathbf{A}$, this often converges twice as fast as Jacobi. This is usually win–win, although Jacobi iteration can in some cases win by use of parallelism.

But there is a dramatically better method using only trivially more effort, *successive overrelaxation* (SOR). Split $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$, with $\mathbf{L}$ now being strictly lower-triangular. We get, with an "overrelaxation parameter" $\omega \in (0,2)$,

$$(\mathbf{D} + \omega\mathbf{L})\mathbf{x} = \omega\mathbf{b} - (\omega\mathbf{U} - (\omega - 1)\mathbf{D})\mathbf{x}$$

from the following:

$$\begin{aligned}
\mathbf{A}\mathbf{x} &= \mathbf{b} \\
\omega\mathbf{A}\mathbf{x} &= \omega\mathbf{b} \\
\mathbf{D}\mathbf{x} + \omega\mathbf{A}\mathbf{x} &= \omega\mathbf{b} + \mathbf{D}\mathbf{x} \\
\mathbf{D}\mathbf{x} + \omega(\mathbf{L} + \mathbf{D} + \mathbf{U})\mathbf{x} &= \omega\mathbf{b} + \mathbf{D}\mathbf{x} \\
(\mathbf{D} + \omega\mathbf{L})\mathbf{x} &= \omega\mathbf{b} + \mathbf{D}\mathbf{x} - \omega\mathbf{D}\mathbf{x} - \omega\mathbf{U}\mathbf{x} \\
&= \omega\mathbf{b} - (\omega\mathbf{U} - (\omega - 1)\mathbf{D})\mathbf{x}.
\end{aligned}$$

Here, $\mathbf{P} = \omega(\mathbf{L} + \omega\mathbf{D})^{-1}$ and we have a free parameter ω, the relaxation parameter, to choose. We may choose it differently for every iteration, to try to minimize the maximum eigenvalue of what we have been calling $\mathbf{S}$. As information is extracted from the solution estimating the largest Jacobi iteration matrix eigenvalue, we may improve our choice. Here $\mathbf{S} = (\mathbf{L} + \omega\mathbf{D})^{-1}((\omega - 1)\mathbf{D} - \omega\mathbf{U})$, and for some finite-difference applications the optimal ω is known. For the right choice of ω, this can seriously outperform Gauss–Seidel.

Example 7.3. We use `A = delsq( numgrid( 'B', n ) )` as an example for SOR, even though direct methods are actually better for this nearly banded matrix. We look first at small-dimension matrices, specifically for $n = 5$, 8, 13, 21, and 34. The dimension of $\mathbf{A}$ is $O(n^2) \times O(n^2)$. By fitting the data from these smaller matrices, the largest eigenvalue of the Jacobi iteration matrix $\mathbf{D}^{-1}(\mathbf{A} - \mathbf{D})$ seems to be $\mu = 1 - {}^{16.65}\!/\!{}_{n^2}$, which means that the optimal $\omega = 2/(1 + \sqrt{1 - \mu^2})$ is about $2/(1 + {}^{16.65}\!/\!{}_{n})$, and the eigenvalues of the SOR error matrix are then less than $(1 - {}^{16.65}\!/\!{}_{n})/(1 + {}^{16.65}\!/\!{}_{n})$, approximately.

When we use 150 iterations of SOR to solve the system for $n = 80$ (so the matrix is 4808×4808), we find that the residual behaves on the kth iteration as approximately $10^3 \times (\omega - 1)^k$, and after 150 iterations, the residual is 4.5×10^{-7}. In contrast, the same number of Jacobi iterations cannot be expected even to give one figure of accuracy, and Gauss–Seidel is not much better. The difference between $(1 - O({}^{1}\!/\!{}_{n}))^k$ and $(1 - O({}^{1}\!/\!{}_{n^2}))^k$ is huge. The constant 10^3 above changes, of course, with the dimension n. It seems experimentally to vary as $(n^2)^2$ or the square of the dimension of $\mathbf{A}$, which, though growing with n, is at least not growing *exponentially* with n. ◁

Remark 7.2. These classical methods are still useful in some circumstances, but there have been serious advances in iterative methods since these were invented. Multigrid methods and conjugate gradient methods seem to be the methods of choice. See Hogben (2006, chapter 41), by Anne Greenbaum, for an entry point to the literature. ◁

7.4 Large Eigenvalue Problems

All methods for finding eigenvalues are iterative[1]; so, unlike the case where we were solving $\mathbf{A}\mathbf{x} = \mathbf{b}$, where there was a distinction between finite, terminating "direct" methods (such as QR factoring or LU factoring) and nonterminating "iterative" methods such as SOR, when we tackle $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$, the distinction in algorithm classes is a bit fuzzy and depends chiefly on how large a "large matrix" is today. On a tablet PC in 2010, not a high-end machine by any means, it took MATLAB five seconds to compute all 1,000 eigenvalues and eigenvectors of a random $1{,}000 \times 1{,}000$ matrix, as follows:

[1] Yes, even for $n = 2$, because while square roots are "legal," they are not finite—extracting them is iterative, too.

```
%% Eigenvalues of a 1000 by 1000 Random Matrix
A = rand( 1000 );
e = eig( A );
plot( real(e), imag(e), 'k.' )
axis('square'), axis([-10,10,-10,10]),set(gca,'Fontsize',16)
xlabel('Real␣Part'),ylabel('Imaginary␣Part')
```

So today a $1,000 \times 1,000$ matrix is not large, even though it and its matrix of eigenvectors have a million entries each. See Fig. 7.3.

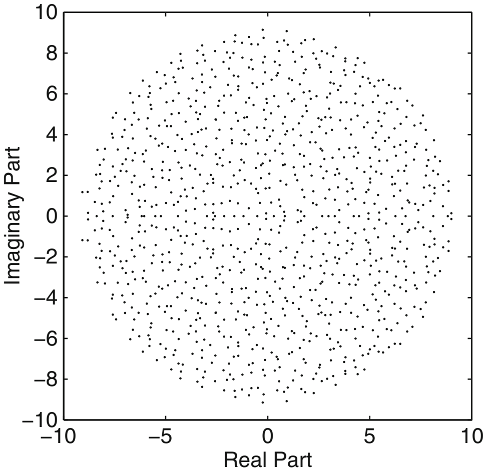


Fig. 7.3 Nine hundred ninety-nine eigenvalues of a random $1,000 \times 1,000$ real matrix. The odd eigenvalue is about 500.3294 (because all entries of this matrix are positive, the Perron–Frobenius theorem applies, and thus there is a unique eigenvalue with largest magnitude, which is real). Note the conjugate symmetry, and the confinement to a disk with radius about 10

For many applications, however, we might not need all $1,000$ eigenvalues and eigenvectors, but perhaps just the six largest, or six smallest. Consider the following situation. Suppose we execute

```
a=rand(1000);
eigs(a)
```

in MATLAB and receive the following warning:

```
Warning: Only 5 of the 6 requested eigenvalues converged.
In eigs>processEUPDinfo at 1474
In eigs at 367
```

This command had some sort of iteration failure—it only found five of the six largest eigenvalues. We will see in a moment a possible way to work around this failure. But before, notice that if we execute

```
eigs(a,6,0)
```

we successfully and quickly find the six smallest eigenvalues. Note that eigs is not eig. The “s” is for “sparse,” although it works (as in this case) on a dense matrix. The following simple kludge avoids the convergence failure in this example:

```
eigs( a - 10.032*speye(1000) )
ans + 10.032
```

That is, we simply shifted the matrix a random amount, and this was enough to kick the iteration over its difficulties. Then we correctly find the eigenvalues:

$$10^2 \begin{bmatrix} 5.0033 \\ -0.0908-0.0118i \\ -0.0908+0.0118i \\ -0.0882+0.0119i \\ -0.0882-0.0119i \\ -0.0880+0.0016i \end{bmatrix} .$$

This is, of course, not entirely satisfactory, but we shall pursue this in a bit of detail shortly.

For large sparse matrices, special methods of iterating are needed: The construction of an upper Hessenberg intermediate matrix is already too expensive, so the QR iteration (as is) is also too expensive. The techniques of choice are Arnoldi iteration (as implemented in ARPACK and in MATLAB’s eigs routine) and other special-purpose routines, such as Rayleigh quotient iteration for the symmetric eigenproblem. Before moving on to this method, we consider the so-called Krylov subspaces

$$\begin{bmatrix} \mathbf{v} & \mathbf{A}\mathbf{v} & \mathbf{A}^2\mathbf{v} & \mathbf{A}^3\mathbf{v} & \dots & \mathbf{A}^k\mathbf{v} \end{bmatrix} ,$$

which can be generated using only k matrix–vector multiplications. The power method considered only the latest $\mathbf{A}^k\mathbf{v}$ (and perhaps the previous). In exact arithmetic, as noted before, the characteristic polynomial can be constructed from the finite sequence $[\mathbf{v}, \mathbf{A}\mathbf{v}, \dots, \mathbf{A}^n\mathbf{v}]$ because these vectors must be linearly dependent; but in the presence of rounding errors, we are much better off using other techniques; if we’re at all lucky, we will get good eigenvalue information with k iterations for $k \ll n$.

Rayleigh quotient iteration—or RQI—is easily described (see Problem 6.16). Given an initial guess for an eigenvector $\mathbf{x}_0$, form

$$\mu = \frac{\mathbf{x}_0^H \mathbf{A}\mathbf{x}_0}{\mathbf{x}_0^H \mathbf{x}_0} ,$$

the Rayleigh quotient. We make the crucial simplification of assuming $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\mathbf{A}^H = \mathbf{A}$; that is, $\mathbf{A}$ is symmetric. More, let $\mathbf{A}$ be positive-definite, and sparse (or at least fast to make matrix–vector products $\mathbf{y} = \mathbf{A}\mathbf{v}$ with). Finally, we suppose eigenvalues are simple. Once we have μ, which is the best least-squares approximation to an eigenvalue corresponding to $\mathbf{x}_0$, we now use it to improve $\mathbf{x}_0$. Solve

$$(\mathbf{A} - \mu\mathbf{I})\mathbf{z} = \mathbf{x}_0, \tag{7.3}$$

and put $\mathbf{x}_1 = \mathbf{z}/\|\mathbf{z}\|$. You may use any convenient method to solve Eq. (7.3); since $\mathbf{A}$ is sparse (or $\mathbf{Av}$ is easy), you may choose a sparse iterative method. You may choose not to solve it very accurately; after all, $\mathbf{x}_1$ will just be another approximate eigenvector, and we're going to do the iteration again. When do we stop? If

$$\|\mathbf{A}\mathbf{x}_i - \mu_i \mathbf{x}_i\| < \varepsilon\,,$$

then we know that μ_i is an exact eigenvalue for $\mathbf{A} + \Delta\mathbf{A}$ with $\|\Delta\mathbf{A}\| \leq \varepsilon\|\mathbf{A}\|$. Hence, this is a reliable test for convergence, from a backward error point of view. Since symmetric matrices have perfectly conditioned eigenvalues (normwise), this may be satisfactory from the forward point of view, too. Thus, we get Algorithm 7.1.

Algorithm 7.1 Rayleigh quotient iteration

```
Require: A vector $\mathbf{x}_0$, a method to compute $\mathbf{y} = \mathbf{Av}$, a method to solve $(\mathbf{A} - \mu\mathbf{I})\mathbf{z} = \mathbf{b}$
  for $i = 1, 2, \ldots$ until converged do
    $\mu_{i-1} = \mathbf{x}_{i-1}^T(\mathbf{A}\mathbf{x}_{i-1})/(\mathbf{x}_{i-1}^T\mathbf{x}_{i-1})$
    Solve $(\mathbf{A} - \mu_{i-1}\mathbf{I})\mathbf{z} = \mathbf{x}_{i-1}$
    $\mathbf{x}_i = \mathbf{z}/\|\mathbf{z}\|$
  end for
```

We may want to find generalizations of this method; for example, we wish to find more than one eigenvector at a time. Suppose $\mathbf{x}_0 \in \mathbb{R}^{n\times k}$ $(k \ll n)$. Then if $\mathbf{x}_0^T\mathbf{x}_0 = \mathbf{I}$,

$$\mathbf{H} = \mathbf{x}_0^T\mathbf{A}\mathbf{x}_0 \in \mathbb{R}^{k\times k}$$

shares some interesting features with the 1×1 case. The eigenvalues of $\mathbf{H}$, called Ritz values, are approximations to eigenvalues of $\mathbf{A}$, in some sense. Alternatively, one can think of the following iteration:

```
for $i = 1, 2, \ldots$ until converged do
  $\mathbf{H} = \mathbf{x}_{i-1}^T\mathbf{A}\mathbf{x}_{i-1}$
  $\mu = \mathrm{diag}(\mathbf{H})$
  for $j = 1, 2, \ldots, k$ do
    Solve $(\mathbf{A} - \mu_{jj}\mathbf{I})\,\mathbf{z}_j = (\mathbf{x}_{i-1})_j$
    $(\mathbf{x}_i)_j = \mathbf{z}_j$
  end for
  $(\mathbf{X}_j, \mathbf{R}) = \mathtt{qr}(\mathbf{X}_j)$
end for
```

This essentially does k independent Rayleigh iterations at once; the `qr` step just makes sure the eigenvalues are kept separate.

We might also wish to solve unsymmetric problems. The difficulties here are worse, as we must solve for left eigenvectors, too; this is called broken iteration, or Ostrowski iteration for some variations. In the symmetric case, convergence is often cubic; for the nonsymmetric case, this is true only sometimes. More seriously, if all we can do with $\mathbf{A}$ is make $\mathbf{Av}$, how do we make $\mathbf{y}^H\mathbf{A}$? This can be done without constructing $\mathbf{A}$ explicitly [which costs $O(n^2)$], but it can be awkward.[2] Still, we have a method:

```
Require: For $\mathbf{x}_0, \mathbf{y}_0 \in \mathbb{C}^n$, a way to compute $\mathbf{Av}$ and a way to solve both
  $(\mathbf{A} - \mu\mathbf{I})\mathbf{z} = \mathbf{x}$ and $\left(\mathbf{A}^H - \overline{\mu}\mathbf{I}\right)\mathbf{w}^H = \mathbf{y}^H$
  for $i = 1, 2, \ldots$ until converged do
    $\mu_{i-1} = (\mathbf{y}_{i-1}^H \mathbf{A}\mathbf{x}_{i-1})/(\mathbf{y}_{i-1}^H \mathbf{x}_{i-1})$    (N.B. fails if $\mathbf{y}_{i-1}^H \mathbf{x}_{i-1}$ is too small)
    Solve $(\mathbf{A} - \mu_{i-1}\mathbf{I})\mathbf{z} = \mathbf{x}_{i-1}$
    $\mathbf{x}_i = \mathbf{z}/\|\mathbf{z}\|$
    Solve $(\mathbf{A}^H - \overline{\mu}\mathbf{I})\mathbf{w} = \mathbf{y}_{i-1}$
    $\mathbf{y}_i = \mathbf{w}/\|\mathbf{w}\|$
  end for
```

Convergence in residual happens if

$$\|\mathbf{A}\mathbf{x}_i - \mu_i \mathbf{x}_i\| \leq \varepsilon$$

as before, but note that now the eigenvalue may be very ill-conditioned, in which case $\mu_i \in \Lambda_\varepsilon(\mathbf{A})$ does not mean $|\lambda - \mu_i| = O(\varepsilon)$ for a modest multiple of ε.[3]

Again, when to *stop* the iteration? Since the residuals are being computed at each stage, one can in principle stop if the residuals get small enough that the backward error interpretation of $\mathbf{r}$, namely, that we have solved $\mathbf{Ax} = \mathbf{b} - \mathbf{r}$, suggests that the residual is negligible. However, rounding errors (especially if the matrix $\mathbf{S}$ is not normal) can prevent the residuals from getting as small as we like.[4]

Example 7.4. The popular Jenkins–Traub method (Jenkins and Traub 1970) for finding roots of polynomials expressed in the monomial basis has at its core an iteration related to the Rayleigh quotient iteration on the companion matrix for the polynomial. In this example, we use RQI on the companion matrix of a polynomial to find some of its roots, as follows. Recall that a companion matrix for a monic polynomial $p(z) = a_0 + a_1 z + \cdots + z^n$ can be written as a sparse matrix, all zero except for the first subdiagonal, which is just 1s, and the final column, which is the negative of the polynomial coefficients. It is a short exercise to see that if z is a root of $p(z)$, then the vector $[1, z, z^2, \ldots, z^{n-1}]$ is a left eigenvector of $\mathbf{C}$, and a corresponding right eigenvector is $[\alpha_1(z), \alpha_2(z), \ldots, \alpha_n(z)]$, where $\alpha_n(z) = 1$, $\alpha_{n-1}(z) = a_{n-1} + z$, $\alpha_{n-2}(z) = a_{n-2} + z(a_{n-1} + z)$, and so on up until $\alpha_1(z) = a_1 + z(a_2 + z(a_3 + \cdots)$, which must also equal ${}^{-a_0}/_{z}$ if $z \neq 0$ (and, of course, $a_0 = 0$ if $z = 0$). These are the successive evaluations of the polynomial that one gets by executing Horner's method. That is, for *this* kind of matrix, a guess at an eigenvalue λ will automatically give us a pair of approximate left and right eigenvectors. It is simple to form the Rayleigh quotient $(\mathbf{x}^H\mathbf{C}\mathbf{x})/(\mathbf{x}^H\mathbf{x})$ or the Ostrowski quotient $(\mathbf{y}^H\mathbf{C}\mathbf{x})/(\mathbf{y}^H\mathbf{x})$ from these to give us a hopefully improved estimate of the eigenvalue (which then can be

[2] See Bostan et al. (2003). For a history of the transposition principle, see `http://cr.yp.to/transposition.html`.

[3] Please consult Demmel (1997) or Hogben (2006) for more information on general techniques such as the implicitly restarted Arnoldi iteration.

[4] For more on this, see the discussion in Higham (2002).

fed back into the eigenvector formulae to use on the next iteration). This works, and it's faster than solving (which also works, and works more generally).

Consider Newton's example, $p(z) = z^3 - 2z - 5$. A companion matrix for this is

$$\mathbf{C} = \begin{bmatrix} 0 & 0 & 5 \\ 1 & 0 & 2 \\ 0 & 1 & 0 \end{bmatrix} .$$

If we start with an initial approximation $z_0 = -1 + i$ and use the formulae above for Ostrowski iteration, we get convergence in five iterations. If instead we *solve* for our approximate eigenvectors at each step via $\mathbf{C} - z^{(i)})\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)}$, and similarly for the left eigenvector, neither of which is hard because this matrix is sparse, then this is more like a normal Rayleigh quotient case where we don't know what the eigenvectors look like. In both cases the convergence appears to be quadratic, but the Rayleigh quotient only converges if solving for the new eigenvector happens each time. That is, with the formulae for the left and right eigenvectors instead of solving, only Ostrowski (also called "broken") iteration converges, but Rayleigh quotient iteration converges if the new eigenvectors are solved for.

Once a root has been found, it is necessary to *deflate* the matrix (or the polynomial); we do not discuss this in any detail here, although note that this is entirely possible within the framework of matrices—using either the left or right eigenvectors, one can in theory find a matrix one dimension smaller that has all the remaining roots as eigenvalues. Let

$$\mathbf{X} = \begin{bmatrix} \alpha_1 & 0 & 0 \\ \alpha_2 & 1 & 0 \\ \alpha_3 & 0 & 1 \end{bmatrix} ,$$

where the first column is the right eigenvector corresponding to the root z that we have found. Note that $\alpha_1 = {}^{-a_0}\!/_{z}$, which we assume is nonzero, so that $\mathbf{X}$ is invertible. Then $\mathbf{X}^{-1}\mathbf{C}\mathbf{X}$ has $[z, 0, 0]^T$ as its first column, and the remaining two eigenvalues of $\mathbf{C}$ are the two eigenvalues of the 2×2 block in the second two rows and columns. Similarly, one could deflate instead with the left eigenvector (which works even if $a_0 = 0$, though trivially since the matrix is already deflated in that case).

This is mathematically equivalent to synthetic division if the right eigenvector is used, and the deflated matrix is also a companion matrix; if the left eigenvector is used, then a different matrix is obtained. However, there is a tendency for rounding errors to accumulate in this process when one works with polynomials of high degree.

One can use a code such as this to implement this idea:

```
%% Rayleigh Quotient Iteration for a Companion Matrix
%
% Newton's example polynomial was $p(z) = z^3 - 2z - 5 = 0$.
%
C = [0 0 1.67608204095197550; 1 0 2; 0 1 -0.66478359180960489;];
x0 = -6 + 5i;
```

```
x = @(z) [-C(2,end)+z*(C(3,end)+z); C(3,end)+z; 1];
niters = 19;
xi  = zeros( niters, 1 );
xia = zeros( niters, 1 );
% Now solve at each step for new eigenvector.
xi(1)  = x0;
xia(1)= x0;
x1 = x(x0); % Initial eigenvector
xa = x1;
x1 = x1/norm(x1,2);
for i=2:niters,
    x1 = (C-xi(i-1)*eye(3))\x1;
    x1 = x1 / norm(x1,2) ;
    xi(i) = (x1' * C * x1 ); %(x1'*x1) =1
    xia(i) = (xa' * C * xa )/(xa'*xa);
    xa = x(xi(i)); % analytic eigenvector formula
end
ers = xi(:) - xi(end);
close( figure(1) )
figure(1), semilogy( abs(ers), 'ko' ), set(gca,'fontsize',16),
    hold on
ersa = xia(:)-xi(end);
semilogy( abs( ersa ), 'kS' )
```

It is straightforward to adapt this code for other similar problems. ◁

Problems

7.1. Add an iterative refinement step to your solution of Problem 6.6. Note that evaluation of the residual is comparable in cost to the solution of the system, so this is a significantly costly step in this case. Does this help?

7.2. Consider the following system:

$$\begin{aligned} 2x_1 - x_2 &= 1 \\ -x_{j-1} + 2x_j - x_{j+1} &= j, \qquad j = 2, \ldots, n-1 \\ -x_{n-1} + 2x_2 &= n \end{aligned}$$

with $n = 100$. Parts 1–2 are from Moler (2004, prob. 2.19).

1. Use `diag` or `spdiags` to form the coefficient matrix and then use `lu`, \, and `tridisolve` to solve the system.
2. Use `condest` to estimate the condition of the coefficient matrix.
3. Solve the same problem as above, but changing 2 to be $\theta > 2$, say $\theta = 2.1$, and using the approach of `Seneca.m` to encode the matrix–vector product, use Jacobi iteration instead (note that $\mathbf{P}^{-1} = \theta \mathbf{I}$ and so $\mathbf{P}\mathbf{x} = \frac{1}{\theta}\mathbf{x}$ is particularly easy). How large can the size of the problem be, before it takes MATLAB at least 60 s to solve the problem this way? How large can the problem be using a direct method?

(And, even more, the comparison is unfair; MATLAB's method is built-in, and Jacobi iteration must be "interpreted." Still, ...)

7.3. Implement in MATLAB the SOR method as described in the text. Be careful not to invert any matrices. Use your implementation with $\omega = 2 - O(1/n)$ to solve the linear system described in Problem 7.2 with $\theta = 2.1$.

7.4. Take $\mathbf{A} = \texttt{hilb}(8)$, the 8×8 Hilbert matrix. Use MGS to factor $\mathbf{A}$ approximately:

$$\mathbf{A} = \mathbf{QR}$$

with $\mathbf{Q}^T\mathbf{Q} \doteq \mathbf{I}$. In fact, $\mathbf{Q}^T\mathbf{Q} = \mathbf{I} + \mathbf{E}$, where $\|\mathbf{E}\| \le \kappa(\mathbf{A}) \cdot c \cdot \mu_M$, where c is a modest constant and μ_M is the unit roundoff. Solve $\mathbf{Ax} = \mathbf{b}$ by using this $\mathbf{Q}$ and $\mathbf{R}$ in a factoring, as follows:

$$\begin{aligned} \mathbf{Qy} &= \mathbf{b} \\ \mathbf{Rx} &= \mathbf{y}\,, \end{aligned}$$

and use the solution process

$$\begin{aligned} \hat{\mathbf{y}} &= \mathbf{Q}^T\mathbf{b} \\ \hat{\mathbf{x}} &= \mathbf{R}\backslash\hat{\mathbf{y}}\,. \end{aligned}$$

Use one or two iterations of refinement to improve your solution. Discuss.

7.5. Consider the matrix from Example 7.2. Use the formula for the pseudospectral radius, namely, Eq. (5.13), and the estimate $\|\,((\mathbf{A}) - z\mathbf{I})^{-1}\,\|_2 \ge |z - 8/9|^n$ [this is easy to see, because the $(n,1)$ entry of the resolvent is just that, and the 2-norm must be at least as large as any element of the matrix] to derive a reasonably tight lower bound on the maximum $\|\mathbf{S}^k\|_2$ when $n = 89$. Verify your bound by computation of $\mathbf{S}^k$ for $1 \le k \le 1600$. Hint: Take $\varepsilon = e/9^n$ and use $e^{1/n} > 1 + 1/n$. Ultimately, of course, $\|\mathbf{S}^k\|_2$ must go to zero as $k \to \infty$, but this analysis shows that it gets quite large along the way. This is why Richardson iteration is so slow for the system $(\mathbf{I} - \mathbf{S})\,\mathbf{x} = \mathbf{b}$.

7.6. The diagonal dominance of the matrix

$$\mathbf{A} = \begin{bmatrix} -10 & 1 & & & & \\ 1 & -10 & 1 & & & \\ & 1 & -10 & 1 & & \\ & & 1 & -10 & 1 & \\ & & & 1 & -10 & 1 \\ & & & & 1 & -10 \end{bmatrix}$$

tempts us to try Jacobi iteration $x_{n+1} = x_n + \mathbf{D}^{-1}\,(\mathbf{b} - \mathbf{A}x_n)$.

1. For $\mathbf{b} = [1, 1, 1, 1, 1, 1]^T$ and an initial guess of $x_0 = -[1, 1, 1, 1, 1, 1]^T/10$, carry out two iterations by hand. (The arithmetic for this problem is not out of reach:

The numbers were chosen to be nice enough to do on a midterm exam.) Can you estimate how accurate your final answer is?

2. Using symmetry and the eigenvalue formula for tridiagonal Toeplitz matrices $\lambda_k = -10 + 2\cos(\pi k/(n+1))$ (here $n = 6$), estimate the 2-norm condition number. The Skeel condition number $\mathrm{cond}(\mathbf{A}) = \| \, |\mathbf{A}^{-1}| \, |\mathbf{A}| \, \|$ can be shown to have exactly the same value. Using the phrases “structured condition number” and “structured backward error” in a sentence, explain what this means.

Part III
Interpolation, Differentiation, and Quadrature

Numerical analysis itself is concerned with the problems of continuous mathematics, as we have discussed. We have examined in previous chapters the use of numerical methods for evaluating continuous functions of various kinds at discrete points, and solving sets of discrete equations. Experimental sampling also produces discrete points. If we wish to then apply numerical methods further on the results of either of these processes, it seems natural to consider the problem of "filling in the gaps," moving from the discrete samples to a continuous function—back to the original function if possible, but in any case back to a convenient representation of something close to that function, if not.

This is the problem of *interpolation*. Let $f(t)$ be a (possibly unknown) function defined on a certain interval or complex set I. Suppose that are given the values ρ of f at $n+1$ points (hereafter called *nodes*) τ_0, τ_1, τ_2, ..., τ_n. In other words, we are given a set of equations

$$f(\tau_i) = \rho_i, \qquad i = 0, 1, 2, \ldots, n, \tag{III.1}$$

Given this data, we may then want to find the values of $f(t)$ for $t \notin I$, or perhaps to find values of the derivatives $f'(t)$. We may simply want to find a formula for such t, for further mathematical investigation, or we may want to find an efficient method to evaluate these quantities given a numerical value for t.

Our first problem, then, is to find *a* function $f(t)$, for $t \in I$ at points $t \neq \tau_i$ $(i = 0, 1, \ldots, n)$, given the numerical data available (see Fig. III.1). Of course this problem does not have a unique solution. We will make restrictions on the space of functions that we look for solutions, in order to make a choice of solution. To help remember that this problem is called interpolation, notice that the same problem for $t \notin I$ is called, in contrast, *extrapolation*.

As one would expect (and as previously stated), because only a finite amount of information is specified, the information given (the vectors $\boldsymbol{\tau}$ and $\boldsymbol{\rho}$) will not in general be sufficient to uniquely identify the function $f(t)$. Typically, we select

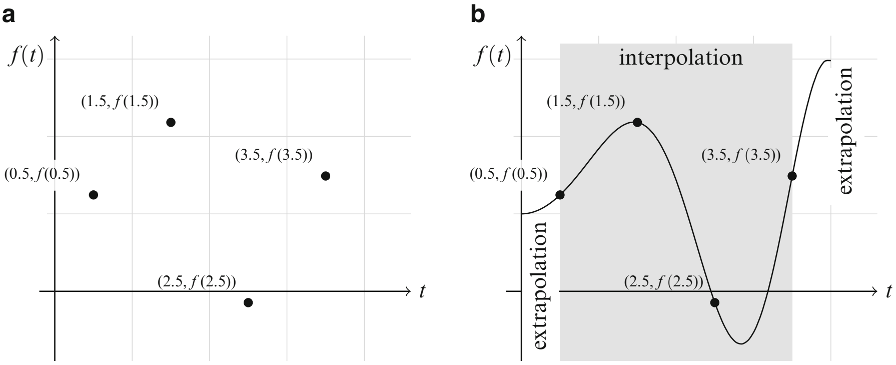


Fig. III.1 The problem of interpolation. (**a**) Data of the problem. In some cases, we also have some values of $f^{(k)}(\tau)$. (**b**) Interpolation and extrapolation: two function evaluation problems

a finite-dimensional subspace of functions in which to look for the $p(t)$ that will approximate the function. At the end of this introduction we will consider what happens if we do not restrict to a finite-dimensional subspace. Here, the resulting *absolute interpolation error* will be

$$f(x) - p(x)\,, \qquad x \in I\,.$$

We will also be concerned with the size of the difference between our reconstructed ("reverse engineered") $p(t)$ and the underlying reference function $f(t)$ from which the samples were taken, namely

$$\|f(x) - p(x)\|\,, \qquad x \in \Omega \setminus I\,.$$

We may not always be able to say much about this kind of interpolation error, however.

How do we proceed to select $p(t)$? Four things have to be determined (see Hamming 1973, 230):

1. What data is available?
2. Given the type of data available, to what class of functions do we want to restrict $p(t)$?
3. Given the type of data and the class of function, which function $p(t)$ should be selected, i.e., what selection criterion should be used?
4. Where should the criterion be applied?

In applications, there are common combinations of answers to these questions; this part of the book is written with these in mind.

With respect to the data, a first question one should ask is whether one can choose the nodes. If choosing the nodes is possible—a case that obtains when we gather experimental data or when we sample a function f at strategic points—then a good node selection method will have to be determined, since different node choices will usually change the interpolation error. In many cases, however, nodes are just given to us, without possibility of choosing them strategically. Sometimes, we also have information concerning the values of the derivatives $f^{(k)}(\tau_i)$. This information should allow us to obtain more accurate approximations of f. Next, one should ask how reliable the data are. If the data are very accurate, then it makes sense to restrict our selection criteria by requiring that

$$f(\tau_i) = p(\tau_i), \qquad i = 0, 1, 2, \ldots, n, \tag{III.2}$$

i.e., by requiring that p be exact at the nodes. If we require this condition, and that the degree be less than or equal to n, then we notice that $\boldsymbol{\tau}$ and $\boldsymbol{\rho}$ together determine a unique *polynomial*, called the *interpolational polynomial* (see Sect. 8.1). Further, if values of $f^{(k)}(\tau_i)$ are available, we can also require that the values of the kth derivatives also match, i.e., we can require that

$$f^{(k)}(\tau_i) = p^{(k)}(\tau_i), \qquad i = 0, 1, 2, \ldots, n \tag{III.3}$$

in addition to the satisfaction of (III.2). Then $p(t)$ is called the *osculating*[5] polynomial (see Sect. 8.2). As we will see in Sect. 8.5, we can then find a general formula for interpolation error.

As one sees, we spend a significant amount of space on polynomial interpolation. Polynomial interpolation must be considered fundamental for many reasons. To begin with, polynomials are extremely useful, since they are easy to compute, as are their derivatives and integrals; in practice, this is a tremendous advantage. Moreover, their error theory serves as a paradigm for the more difficult cases. Finally, if they are used judiciously, they often provide satisfying approximations.

One important problem that we do not discuss in as much detail as perhaps we should is the question of *degree*. Deciding the degree of a polynomial is not always straightforward, if all one has to go on is imperfect data, data containing unknown errors. We do return to this question, but there's a lot to be said, here.

We note that it could also be possible to generate polynomials in a way that Eqs. (III.1) and (III.3) are only approximately satisfied, using least-squares or minimax, but we will not examine this problem much here (see Hamming 1973). Nonetheless, simple polynomial interpolation can often be improved upon by means of so-called *piecewise* interpolation, in which $p(t)$ is taken to be a piecewise polynomial function. Working with piecewise polynomials allows a greater flexibility for adaptation, which we will take great advantage of.

What about interpolation with other functions? A natural extension is to *rational functions*, and this provides some new challenges but some useful flexibility. Rational interpolation will be discussed briefly. But there are other, more exotic forms of interpolation. The most famous mathematical example is the interpolation of the factorial function with the Gamma function, a problem posed by Goldbach and Daniel Bernoulli, and solved first by Euler. For this famous problem, one does not need a finite-dimensional space in which to work, but rather only a short list of desirable geometric features.[6]

Another interesting class of interpolation problems arises from trying to interpolate *discrete dynamical systems*—that is, to find a continuous dynamical system that interpolates the discrete one. What does it mean to take half a Newton iteration step, when the integer steps are given by $x_{n+1} = x_n - {}^{f(x_n)}/_{f'(x_n)}$? This is also called "taking the logarithm of the homomorphism." We have already mentioned one application of this idea in Chap. 5, namely the Toda lattice differential equations used to analyze the QR iteration: the QR iteration is treated as a discrete dynamical system, which is then interpolated by a Toda flow, that can then be analyzed. We will see this again when we look at the numerical solution of differential equations in Part IV.

[5] From the latin *osculare*, to kiss.

[6] See Andrews et al. (1999 chapter 1) for a discussion.

Chapter 8
Polynomial and Rational Interpolation

Abstract This chapter gives a detailed discussion of *barycentric Lagrange and Hermite interpolation* and extends this to *rational interpolation* with a specified denominator. We discuss the *conditioning* of these interpolants. A *numerically stable* method to *find roots of polynomials expressed in barycentric form* via a generalized eigenvalue problem is given. We conclude with a section on *piecewise polynomial interpolants*. ◁

This chapter gives a detailed discussion of one of this book's key tools, namely, polynomial interpolation. Polynomial interpolation is widely used and has a long history that is slightly complicated by numerical pitfalls, which we shall point out how to avoid.

> Generations of textbooks have warned readers that polynomial interpolation is dangerous. In fact, if the interpolation points are clustered and a stable algorithm is used, it is bulletproof. (Trefethen 2013 summary of chapter 14)

The approach we use is somewhat different to that of most numerical analysis books, in that we concentrate on the use of Lagrange bases and the Hermite interpolational bases. We use contour integration as our fundamental tool in deriving the formulæ that we need. The process boils down to the computation of partial fractions: If you can do a partial fraction expansion, and the "most singular" coefficients are not zero, then you can interpolate.

Interpolation is needed in several senses; for instance, we need interpolants as a means of thinking about constructive approximation (in which case we need formulæ). We also need ways to stably evaluate those formulæ. It will turn out to be important in several applications that interpolants not only accurately approximate the original function, but also that the *derivatives* of the interpolant should also accurately approximate the derivatives of the original function. We reserve discussion of how to *compute* derivatives of polynomials expressed in a Lagrange or Hermite interpolational basis until Chap. 11, though.

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_8,

8.1 Lagrange Interpolation

We here repeat (for convenience) and expand on the material in Sect. 2.2.6 from Chap. 2. We suppose first that we are given a set of distinct nodes τ_i and some values of a function $\rho_i = f(\tau_i)$, and we want to find a polynomial $p(t)$ with degree at most n such that $p(\tau_i) = f(\tau_i)$ for $i = 0, 1, 2, \ldots, n$. Since the nodes are distinct, there is a unique such polynomial. Here we will find the Lagrange representation of that unique interpolating polynomial,

$$p(z) = \sum_{k=0}^{n} \rho_k L_k(z)\,, \tag{8.1}$$

where the Lagrange basis polynomials $L_k(z)$ are[1]

$$L_k(z) = \prod_{\substack{i=0\\ i\neq k}}^{n} \frac{z-\tau_i}{\tau_k-\tau_i}\,. \tag{8.2}$$

As is easy to verify, the Lagrange polynomials L_k corresponding to the node τ_k satisfy

$$L_k(\tau_j) = \begin{cases} 1 & j = k \\ 0 & j \neq k \end{cases}$$

In this form, the Lagrange interpolating polynomial is simple to compute.

Example 8.1. Let $\boldsymbol{\tau} = [3, -2, 1, 4]$ and $\boldsymbol{\rho} = [2, 5, -3, 1]$. First, we compute the $L_k(z)$. We find $L_0(z)$ as follows:

$$L_0(z) = \frac{(z-(-2))(z-1)(z-4)}{(3-(-2))(3-1)(3-4)} = -\frac{1}{10}(z+2)(z-1)(z-4)$$

Similarly,

$$\begin{aligned}
L_1(z) &= \frac{(z-3)(z-1)(z-4)}{(-2-3)(-2-1)(-2-4)} = -\frac{1}{90}(z-3)(z-1)(z-4)\\
L_2(z) &= \frac{(z-3)(z-(-2))(z-4)}{(1-3)(1-(-2))(1-4)} = \frac{1}{18}(z-3)(z+2)(z-4)\\
L_3(z) &= \frac{(z-3)(z-(-2))(z-1)}{(4-3)(4-(-2))(4-1)} = \frac{1}{18}(z-3)(z+2)(z-1)\,.
\end{aligned}$$

Therefore,

[1] Note the different use of "Lagrange interpolating polynomial" and "Lagrange basis polynomials."

$$p(z) = -\frac{1}{5}(z+2)(z-1)(z-4) - \frac{1}{18}(z-2)(z-1)(z-4) \\ -\frac{1}{6}(z-3)(z+2)(z-4) + \frac{1}{18}(z-3)(z+2)(z-1). \quad (8.3)$$

It is easy to verify that $p(z) = f(z)$ at τ_i, $i = 0,1,2,3$. The resulting polynomial is plotted in Fig. 8.1. Note that this can be done in a straightforward way in MAPLE:

```
with(CurveFitting);
PolynomialInterpolation([tau[0], tau[1], tau[2],tau[3]], [rho[0],
     rho[1], rho[2],rho[3]], z, form = Lagrange)
```

results in

$$\frac{\rho_0\,(z-\tau_1)\,(z-\tau_2)\,(z-\tau_3)}{(\tau_0-\tau_1)\,(\tau_0-\tau_2)\,(\tau_0-\tau_3)} + \frac{\rho_1\,(z-\tau_0)\,(z-\tau_2)\,(z-\tau_3)}{(\tau_1-\tau_0)\,(\tau_1-\tau_2)\,(\tau_1-\tau_3)} \\ + \frac{\rho_2\,(z-\tau_0)\,(z-\tau_1)\,(z-\tau_3)}{(\tau_2-\tau_0)\,(\tau_2-\tau_1)\,(\tau_2-\tau_3)} + \frac{\rho_3\,(z-\tau_0)\,(z-\tau_1)\,(z-\tau_2)}{(\tau_3-\tau_0)\,(\tau_3-\tau_1)\,(\tau_3-\tau_2)}$$

for a given set of nodes. ◁

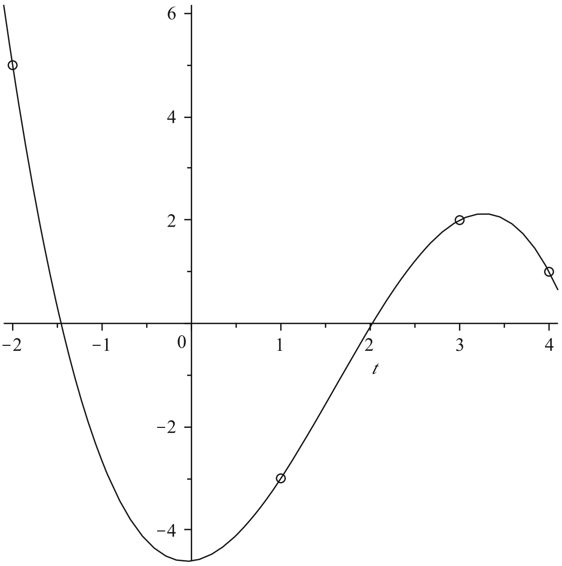


Fig. 8.1 Interpolant of Eq. (8.3) for $\boldsymbol{\tau} = [3,-2,1,4]$ and $\boldsymbol{\rho} = [2,5,-3,1]$

One sees that this form of the Lagrange interpolating polynomial cannot handle repeated nodes, that is, two nodes $\tau_i = \tau_j$ with $0 \le i \ne j \le n$, since it would entail a division by zero. Also, from the numerical point of view, this formula seems likely to be numerically unstable when the nodes are close, that is, when for some $0 \le i < j \le n$, $|\tau_i - \tau_j| \ll 1$. This fact used to bring about comments of the following sort concerning the virtues of Lagrange interpolation:

> Lagrangian interpolation is praised for analytic utility and beauty but deplored for numerical practice. (cited in Berrut and Trefethen 2004)

Nonetheless, that paper goes on to show that the Lagrange interpolating polynomial can be written in another form that *is* numerically stable. Quoting further from that paper,

> the Lagrange approach is in most cases the method of choice for dealing with polynomial interpolants. The key is that the Lagrange polynomial must be manipulated through the formulæ of *barycentric interpolation*. (Berrut and Trefethen 2004)

Thus, in this chapter, we will focus on barycentric Lagrange interpolation and its extensions to different types of data.

Here is how we obtain the barycentric forms of the Lagrange polynomials. First, we define $w(z)$ to be

$$w(z) = (z-\tau_0)(z-\tau_1)\cdots(z-\tau_n) = \prod_{i=0}^{n}(z-\tau_i).$$

Now, let us reconsider the Lagrange polynomials:

$$L_k(z) = \prod_{\substack{i=0\\ i\neq k}}^{n} \frac{z-\tau_i}{\tau_k-\tau_i} = \frac{\displaystyle\prod_{\substack{i=0\\ i\neq k}}^{n}(z-\tau_i)}{\displaystyle\prod_{\substack{i=0\\ i\neq k}}^{n}(\tau_k-\tau_i)}.$$

The numerator can then be rewritten as

$$\prod_{\substack{i=0\\ i\neq k}}^{n}(z-\tau_i) = \frac{w(z)}{z-\tau_k},$$

where $w(z) = \prod_i (z-\tau_i)$. Moreover, we define the $n+1$ *barycentric weights* β_k, $k = 0, 1, 2, \ldots, n$, as follows:

$$\beta_k = \frac{1}{\displaystyle\prod_{\substack{i=0\\ i\neq k}}^{n}(\tau_k-\tau_i)}, \qquad k = 0, 1, 2, \ldots, n. \tag{8.4}$$

The Lagrange basis polynomials can then be rewritten as

$$L_k(z) = w(z)\frac{\beta_k}{z-\tau_k}.$$

Definition 8.1. From Eq. (8.1), we then obtain (for $z \neq \tau_k$ for any node τ_k)

$$p(z) = w(z) \sum_{k=0}^{n} \frac{\beta_k \rho_k}{z - \tau_k} \,. \tag{8.5}$$

This is the *first barycentric form of the Lagrange interpolation polynomial.* ◁

We will discuss the advantages of this form shortly. However, before that, we want to obtain yet another and in some ways even better form of the Lagrange interpolation polynomial. Suppose we interpolate the constant polynomial $p(z) = 1$. Then, for any selection of nodes τ_k, we will have $p(\tau_k) = \rho_k = 1$. Applying this in Eq. (8.5), we obtain

$$1 = w(z) \sum_{k=0}^{n} \frac{\beta_k}{z - \tau_k}. \tag{8.6}$$

Now, if we divide Eq. (8.5) by this equation, we obtain (again for $z \neq \tau_k$ for any node τ_k) this second form:

Definition 8.2. The expression

$$p(z) = \frac{\displaystyle\sum_{k=0}^{n} \frac{\beta_k \rho_k}{z - \tau_k}}{\displaystyle\sum_{k=0}^{n} \frac{\beta_k}{z - \tau_k}} \tag{8.7}$$

is the *second barycentric form of the Lagrange interpolating polynomial.*[2] ◁

The weights corresponding to a set of nodes are easily computed with formula (8.4), and only divisions and summations will then remain to be done. With regard to cost, these formulæ allow us to evaluate the Lagrange polynomial at a single point t using $O(n)$ flops only. Computing all the β_k requires $O(n^2)$, so this is best done ahead of time, once and for all.

The stability of this formula can be grasped intuitively:

> [...] what if the value of x [we use t] in [(8.7)] is very close to one of the interpolation points x_k or, in an extreme case, exactly equal? Consider first the case $x \approx x_k$ but $x \neq x_k$. The quotient $w_k/(x - x_k)$ will be very large, and it would seem that there might be a risk of inaccuracy in this number associated with the subtraction of two nearby quantities in the denominator. However, as pointed out [by] Henrici [1982], this is not, in fact, a problem. Loosely speaking, there is indeed inaccuracy of this kind, but the same inaccurate numbers appear in both the numerator and the denominator of [(8.7)], and the inaccuracies cancel out; the formula remains stable overall. Rigorous arguments that make this intuitive idea precise are provided by Higham [2004]. (Berrut and Trefethen 2004 508-9)

This last point is *crucial*. The barycentric forms are numerically stable to use. If the forms were *not* stable, it wouldn't matter that they are reasonably cheap to evaluate. Once we get to the section on *rational* interpolation, we expand briefly on the accuracy of the second form.

[2] A useful fact, from the numerical point of view, is that we may scale all β_k by any common factor. This can prevent unnecessary overflow or underflow.

Example 8.2. Consider the nodes $\boldsymbol{\tau} = [3,-2,1,4]$ and the values $\boldsymbol{\rho} = [2,5,-3,1]$ again. The barycentric weights are

$$\begin{aligned}
\beta_0 &= ((3+2)(3-1)(3-4))^{-1} = -\frac{1}{10} \\
\beta_1 &= ((-2-3)(-2-1)(-2-4))^{-1} = -\frac{1}{90} \\
\beta_2 &= ((1-3)(1+2)(1-4))^{-1} = \frac{1}{18} \\
\beta_3 &= ((4-3)(4+2)(4-1))^{-1} = \frac{1}{18} .
\end{aligned}$$

Thus, we find that (unless t is exactly equal to one of our nodes 3, -2, 1, or 4, in which case we simply return the known values)

$$p(z) = \frac{\dfrac{-\frac{1}{10}\cdot 2}{z-3} + \dfrac{-\frac{1}{90}\cdot 5}{z+2} + \dfrac{\frac{1}{18}\cdot(-3)}{z-1} + \dfrac{\frac{1}{18}\cdot 1}{z-4}}{\dfrac{-\frac{1}{10}}{z-3} + \dfrac{-\frac{1}{90}}{z+2} + \dfrac{\frac{1}{18}}{z-1} + \dfrac{\frac{1}{18}}{z-4}} .$$

This can be evaluated using 9 flops for every value of t thereafter, which compares reasonably with the cost of evaluation by other methods. We emphasize that it is the numerical stability of the method that recommends it, however: For a good set of nodes, this form is accurate even if z differs by *one bit* from a node, where we would expect the most numerical difficulty through cancellation. In Algorithm 8.1, you will see a useful test for exact floating-point equality, which is quite unusual in a floating-point algorithm: If any input t is *exactly* any τ_k, then the result of the computation (which will be an `Inf` over an `Inf` resulting in a `NaN`) is replaced with the correct data value ρ_k. ◁

Remark 8.1. Not all sets of nodes are created equal, though. The *condition number* of the polynomial expressed in the Lagrange basis on one set of nodes can be quite a bit larger than the condition number of the same polynomial expressed in another Lagrange basis on another set of nodes: Equally spaced nodes are notoriously bad, in that the barycentric weights vary widely, making the expression ill-conditioned and very sensitive to errors and uncertainties in the data (and also to rounding errors). As we will see, Chebyshev nodes and (even better) uniformly spaced nodes on the complex unit circle, on the other hand, have near-constant barycentric weights (after scaling, for use in the second barycentric form) and are well-conditioned. With an ill-conditioned Lagrange basis, it might not be enough that you use the good barycentric forms—your answer still may be inaccurate. With a well-conditioned Lagrange basis, then using a bad numerical method instead of the barycentric forms can destroy the accuracy that is available. It is important to compute the barycentric weights accurately as well—we will return to this point. ◁

Example 8.3. Take the polynomial $p(z) = T_{16}(2z-1)$ on $0 \le z \le 1$ and construct barycentric interpolants for it on two different sets of nodes: first, the scaled

Algorithm 8.1 Computation of the second barycentric form of the Lagrange interpolating polynomial

```
Require: A vector $\mathbf{z}$ of evaluation points, a vector of distinct nodes $\boldsymbol{\tau} \in \mathbb{C}^{n+1}$, and a vector of
  function values $\boldsymbol{\rho} \in \mathbb{C}^{n+1}$.
  for $k = 0$ to $n$ (if not already precomputed) do
    $\beta_k := \prod_{\substack{i=0\\ i\neq k}}^{n} (\tau_k - \tau_i)^{-1}$
  end for
  $\mathbf{Q}$ := vector the same length as $\mathbf{z}$ containing zeros
  $\mathbf{P}$ := vector the same length as $\mathbf{z}$ containing zeros
  set exact := logical vector the same length as $\mathbf{t}$ containing zeros (zero meaning false)
  for $k = 0$ to $n$ do
    $\Delta\mathbf{z} := \mathbf{z} - \tau_k$ (componentwise subtraction)
    exact$(i) = k$ for each $i$ such that $\Delta\mathbf{z}(i) = 0$ with exact floating-point equality
    $\mathbf{Q} := \mathbf{Q} + \beta_k(\mathbf{z} - \tau_k)^{-1}$
    $\mathbf{P} := \mathbf{P} + \beta_k \rho_k (\mathbf{z} - \tau_k)^{-1}$
  end for
  $\mathbf{p} = \mathbf{P}/\mathbf{Q}$ Hadamard (componentwise) division
  $j$ = indexes where exact is true (use find(exact) in MATLAB)
  $\mathbf{p}(j) = \boldsymbol{\rho}$(exact$(j)$) replace with known data for any $z$ such that $z$ is exactly equal in floating-
  point to one of the $\tau_k$
  return $p(t) = \left(\sum_{k=0}^{n} \frac{\beta_k \rho_k}{z - \tau_k}\right) \Big/ \left(\sum_{k=0}^{n} \frac{\beta_k}{z - \tau_k}\right)$, which is the second barycentric form of the
  Lagrange interpolating polynomial $p(t)$.
```

Chebyshev–Lobatto nodes $\tau_k = {}^{(1+\eta_k)}/_2$, where $\eta_k = \cos({}^{\pi k}/_{16})$ for $0 \leq k \leq 16$. Second, use equispaced nodes $x_k = {}^{k}/_{N}$. Each barycentric interpolant is mathematically equivalent: They represent the same polynomial. Now compute the condition numbers for each of the two expressions: $B(z) = \sum |\rho_k L_k(z)|$ (see Chap. 2). These two condition numbers are plotted in Fig. 8.2. We see that the condition number for the equally spaced nodes is vastly worse, near the edges, being many orders of magnitude larger. Near the middle, it's actually slightly better than the scaled Chebyshev node interpolant, but not enough to outweigh the difficulty near the ends. This behavior gets exponentially worse as the degree of the polynomial (and the number of necessary nodes) gets larger. ◁

8.2 Interpolating from Values of a Function and Its Derivatives (Hermite Interpolation)

We now allow a different problem specification. We still assume that we are given a set of nodes τ_i, $i = 0, 1, 2, \ldots, n$, where the nodes are distinct, that is, $\tau_i = \tau_j \Leftrightarrow i = j$. However, in addition to the function values $f(\tau_i) = \rho_i$, we now have values of some of the derivatives of f at the nodes. We can then introduce the so-called Hermite data as follows:

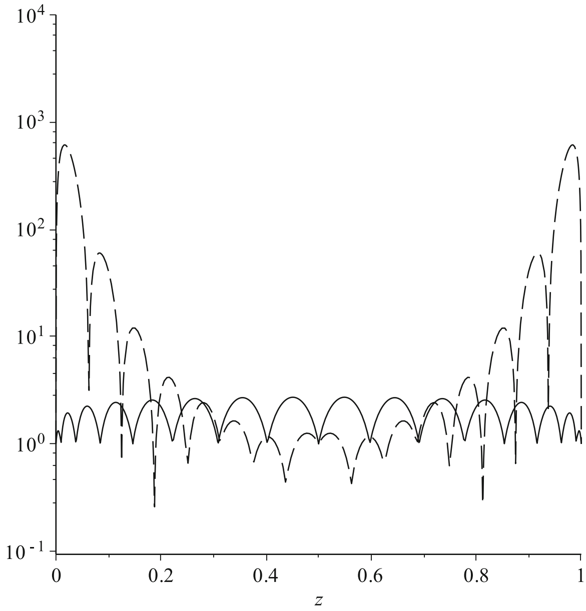


Fig. 8.2 The *solid line* is the condition number $B(z)$ of the Lagrange interpolant to $p(z) = T_{16}(2z-1)$ on the scaled Chebyshev nodes, and the *dashed line* is the condition number $B_E(z)$ of the Lagrange interpolant to the same polynomial on equispaced nodes $k/16$ for $0 \le k \le 16$. Even for so modest a polynomial, the condition number for equally spaced nodes is many orders of magnitude worse at the edges, although slightly better in the middle, where the Chebyshev nodes are more widely spaced than $1/16$

$$\frac{f^{(k)}(\tau_i)}{k!} = \rho_{ik}, \qquad 0 \le i \le n, \quad 0 \le k \le s_i - 1, \tag{8.8}$$

where the integers $s_i \ge 1$ are the "confluencies," that is, the number of values of derivatives available at a node (including the 0th derivative). Note that the ρ_i we had for Lagrange interpolation are just the $\rho_{i,0}$ we have here; the $1/k!$ factor is added to ensure that ρ_{ik} are the coefficients of the terms $(x-a)^k$ in the local Taylor series of f. See Table 8.1 for a schematic example.

As we will see, we can then obtain a formula analogous to Eq. (8.7):

Definition 8.3. The second barycentric form of the Hermite interpolant is

$$p(z) = \frac{\displaystyle\sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\sum_{k=0}^{j} \beta_{i,j}\rho_{ik}(z-\tau_i)^{k-j-1}}{\displaystyle\sum_{i=0}^{n}\sum_{j=0}^{s_i-1} \beta_{i,j}(z-\tau_i)^{-j-1}}. \tag{8.9}$$

Table 8.1 A tabulated schematic of Hermite data

ρ_{ik} τ_i	$\rho_{i,0}$	$\rho_{i,1}$	$\rho_{i,2}$	$\rho_{i,3}$	$\rho_{i,4}$	$\rho_{i,5}$	s_i
τ_0	$f(\tau_0)$	$f'(\tau_0)$	$\frac{f''(\tau_0)}{2}$				3
τ_1	$f(\tau_1)$	$f'(\tau_1)$	$\frac{f''(\tau_1)}{2}$	$\frac{f^{(3)}(\tau_1)}{3!}$	$\frac{f^{(4)}(\tau_1)}{4!}$	$\frac{f^{(5)}(\tau_1)}{5!}$	6
τ_2	$f(\tau_2)$	$f'(\tau_2)$	$\frac{f''(\tau_2)}{2}$	$\frac{f^{(3)}(\tau_2)}{3!}$	$\frac{f^{(4)}(\tau_2)}{4!}$		5
τ_3	$f(\tau_3)$						1
τ_4	$f(\tau_4)$	$f'(\tau_4)$	$\frac{f''(\tau_4)}{2!}$	$\frac{f^{(3)}(\tau_4)}{3!}$			4

The *generalized barycentric weights* $\beta_{i,j}$ that appear in that formula are just the coefficients in the partial fraction expansion that will be shown in Eq. (8.11), and which can be stably computed by Algorithm 2.4. ◁

We put off discussion of the stability, conditioning and cost of this formula, but note that if the confluencies s_k are not "too large," then it has similar properties to the Lagrange interpolant.

Example 8.4. Consider again the Mandelbrot polynomials that you met in Problem 2.35, defined by $p_0(x) = 1$, and $p_{k+1}(x) = xp_k^2(x) + 1$ for $k \geq 0$. Expanding the first three fully in the monomial basis, we have $p_0(x) = 1$, $p_1(x) = x + 1$, and $p_2(x) = x^3 + 2x^2 + x + 1$. Quite clearly, the degrees of p_k increase with k, and it doesn't take very long to realize that the expansion process is unwieldy, with large coefficients being generated. But at certain points, there is an elegant Hermite interpolational representation. Let ω_k be the three zeros of $p_2(x)$, for $k = 1, 2, 3$, say, in this order (to 15 digits, by MAPLE):

$$\omega = [-1.75487766624669, -0.122561166876654 - 0.744861766619744i,\\ -0.122561166876654 + 0.744861766619744i].$$

At these points, $p_3(\omega_k) = \omega_k p_2^2(\omega_k) + 1 = 1$. Also, $p_3'(x) = p_2^2(x) + 2xp_2(x)$, so $p_3'(\omega_k) = 0$. This gives six pieces of information, which is not enough to describe $p_3(x)$ because it is degree 7. Finally, $p_3(0) = 1$, and $p_3'(0) = p_2^2(0) = 1$.

x	$p(x)$	$p'(x)$
0	1	1
ω_1	1	0
ω_2	1	0
ω_3	1	0

This gives us eight pieces of information, enough to describe this degree-7 polynomial perfectly. In MAPLE, this comes out in the first barycentric form as something quite ugly to look at. Instead of printing it explicitly, we give a matrix of the values of β, namely (printing only 5 decimals of the computed values),

$$\begin{bmatrix} -2.0{+}0.0\,i & 0.099256{+}0.0\,i & 0.95037{-}0.54389\,i & 0.95037{+}0.54389\,i \\ 1.0{+}0.0\,i & 0.031332{+}0.0\,i & 0.093030{+}0.22733\,i & 0.093030-0.22733\,i \end{bmatrix},$$

where the first row is $\beta_{i,0}$ and the second row is $\beta_{i,1}$ for $1 \le i \le 4$. Putting $w(z) = z^2(z-\omega_1)^2(z-\omega_2)^2(z-\omega_3)^2$, we then have the first barycentric representation:

$$p(z) = w(z)\sum_{i=1}^{4}\sum_{j=0}^{1}\sum_{k=0}^{j}\beta_{i,j}\rho_{i,k}(z-\tau_i)^{k-j-1}\,.$$

This can be written out explicitly, but it occupies several lines of text and we'd rather not see it.

This indeed *looks* quite a bit uglier than the monomial basis form, $p_3(x) = x^7+4x^6+6x^5+6x^4+5x^3+2x^2+x+1$, but most of that ugliness is because each of the barycentric weights (and the ω_k's) are ugly. We have also suppressed some rounding errors: $2.00\ldots02$, for instance, and $5.5\ldots\cdot 10^{-16}$. We left in the $0.0i$, which is a *complex signed zero* in MAPLE. However, written as a *table* of values, as above, the polynomial is quite simple—in that we can safely leave the evaluation and differentiation of $p_3(x)$ to a computer and need not look on its ugliness. The results of doing so on the interval $-2 \le z \le 0$ are shown in Fig. 8.3. This form will show its advantages better in the next section. ◁

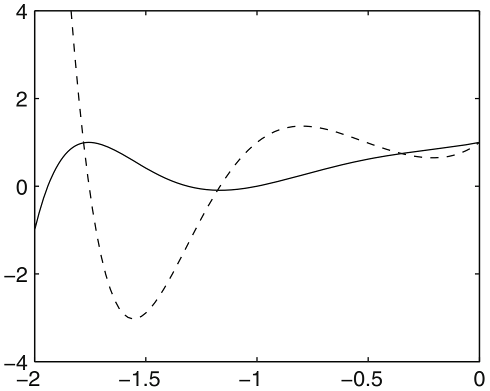


Fig. 8.3 Hermite interpolation of $p_3(z)$ (*solid line*) and its derivative (*dashed line*). Though the interpolation nodes and values were complex, the results are real (although MATLAB complains about roundoff-level imaginary parts when it plots)

8.2.1 Rootfinding for Polynomials Expressed in Hermite Interpolational Bases

Recall the discussion in Sect. 6.6.3 on companion matrices as methods for finding roots of polynomials. Companion pencils are known for other bases and, in particular, are known for the Lagrange and Hermite bases. This can be used without the creation of a specialized rootfinding routine to find easily the zeros of $p(z)$ given in a Lagrange or Hermite interpolational basis *without* changing basis.

Example 8.5. Consider the Lagrange interpolational case first, because it is simpler. Suppose that we know $p(\tau_k) = \rho_k$ for $k = 0$, 1, 2, and 3, so that the polynomial $p(z)$ is of degree at most 3. For simplicity, let $\boldsymbol{\tau} = [-1, -1/2, 1/2, 1]$ (these are Chebyshev nodes of the second kind, places where $T_3(z)$ achieves extrema). Construct the following 5×5 matrix pencil $(\mathbf{A}, \mathbf{B})$ from the barycentric weights $\boldsymbol{\beta} = [-2/3, 4/3, -4/3, 2/3]$ and from the values of the polynomial $\rho_k = p(\tau_k)$:

$$\mathbf{A} = \begin{bmatrix} 0 & -2/3 & 4/3 & -4/3 & 2/3 \\ -\rho_3 & 1 & 0 & 0 & 0 \\ -\rho_2 & 0 & 1/2 & 0 & 0 \\ -\rho_1 & 0 & 0 & -1/2 & 0 \\ -\rho_0 & 0 & 0 & 0 & -1 \end{bmatrix},$$

and $\mathbf{B} \approx \mathbf{I}$, the 5×5 identity matrix, *except that we set* $B_{1,1} = 0$. Then it can easily be shown by the Schur complement that $\det(z\mathbf{B} - \mathbf{A}) = p(z)$ is a degree-3 polynomial and has $p(\tau_k) = \rho_k$ for each τ_k. This companion matrix pencil allows us to find the roots of $p(z)$ by finding the generalized eigenvalues of a companion matrix pencil.

This process is numerically stable: That is, if we use the QZ algorithm on a system using IEEE arithmetic, then the computed eigenvalues will include two copies of `Float(infinity)` and n other values $\lambda_1, \lambda_2, \ldots, \lambda_n$. These eigenvalues are the exact eigenvalues of the pencil $(\mathbf{A} + \Delta\mathbf{A}, \mathbf{B} + \Delta\mathbf{B})$ with $\|(\Delta\mathbf{A}, \Delta\mathbf{B})\|_F \leq c(n)\|\mathbf{A}, \mathbf{B}\|_F \varepsilon_M$, but, moreover—and this is important—both $\Delta A_{11} = 0$ and $\Delta B_{11} = 0$, and so the two spurious infinite eigenvalues are computed exactly. Then, it can be shown that with proper scaling, $(p + \Delta p)(z) := \det(z(\mathbf{B} + \Delta\mathbf{B}) - (\mathbf{A} + \Delta\mathbf{A}))$ has $\deg(p + \Delta p) \leq n$ and

$$(p + \Delta p)(\tau_i) = \rho_i + k_i \varepsilon + O(\varepsilon^2),$$

for some constants k_i bounded by a constant times $\|\boldsymbol{\rho}\| \cdot \max|\beta_i| / \min|\beta_i| / \min|\tau_i - \tau_j|$. See Problem 8.34.[3] This result, in turn, allows the theory of polynomial conditioning in the Lagrange basis to be used directly. The conclusion is that this is a stable numerical method, and (depending on the nodes) the polynomial is often much better conditioned than the mathematically equivalent polynomial expressed in the more usual monomial basis.

Here, suppose that $\rho_k = (-1)^k$. The eigenvalues computed by MAPLE are two copies of `-Float` (∞) +0.0*i*, and 15-digit-accurate approximations to $-\sqrt{3}/2$, 0, and $\sqrt{3}/2 = 0.866\ldots$. ◁

Example 8.6. Consider the polynomial $p(z) = (z - 1/4)^2(z - 1/2)^2(z - 3/4)^2$, which is degree 6. Interpolate $p(z)$ using Hermite data at $z = 0$, $z = 1/4$, $z = 3/4$, and $z = 1$. That is, we specify both $p(z)$ and $p'(z)$ at those four points, giving eight pieces of

[3] See also the forthcoming paper *Stability of rootfinding for barycentric Lagrange enterpolants* by Lawrence et al. (2013).

data to (over)specify this degree-6 polynomial. We then plot the resulting Hermite interpolant evaluated using the barycentric form at many points close to the midpoint $z = 1/2$ at which there should be a double zero. We have seen these plots already, in Figs. 1.3 and 1.4.

Because of interpolation error, there is not exactly a double zero. Because of rounding errors, even those simple-but-close zeros are somewhat fuzzily specified. The example is a good one in that the evaluation and interpolation are all quite stable, giving the exact interpolant of nearby data and for each evaluation point giving the exact value (up to nearly roundoff) of a nearby polynomial. Still, roundoff errors are visible if you zoom in close enough.

The zeros, computed as the eigenvalues of the following companion pencil, are also accurate. The cluster near $1/2$ has average $1/2$ up to roundoff, as expected. The companion pencil has the same kind of matrix **B**, which is an identity matrix except the $(1,1)$ entry is set to zero, and the matrix

$$\mathbf{A} = \begin{bmatrix} 0 & 28.4444 & -360.2963 & 113.7778 & 151.7037 & 113.7778 & -151.7037 & 28.4444 & 360.2963 \\ -0.1289 & 1.0000 & 1.0000 & 0 & 0 & 0 & 0 & 0 & 0 \\ -0.0088 & 0 & 1.0000 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.7500 & 1.0000 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.7500 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.2500 & 1.0000 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.2500 & 0 & 0 \\ 0.1289 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0000 \\ -0.0088 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The computed eigenvalues (once the first row of **A** has been scaled to have norm 1, which we can do because the second barycentric form has generalized barycentric weights both in numerator and denominator, so scaling the $\beta_{i,j}$ doesn't affect the eigenvalues) are exactly $1/4$ (twice), exactly $3/4$ (twice), and $0.499\ldots 9 \pm 1.31 \times 10^{-8}$. Because multiple roots often split symmetrically when perturbed, their *arithmetic mean* is often less sensitive to rounding errors; indeed, the arithmetic mean of these two roots is 0.5 to one ulp, or within $\varepsilon_M = 2\mu_M$ of the exact result. The imaginary part cancels exactly. ◁

Example 8.7. For another Hermite interpolational example, if we know that the degree-5 polynomial $p(z)$ satisfies the six conditions $p(\tau_k) = \rho_{k,0}$ and $p'(\tau_k) = \rho_{k,1}$ for $k = 0$, 1, and 2, where $\tau_0 = -1$, $\tau_1 = 0$, $\tau_2 = 1$, then the barycentric weights are found by the following partial fraction expansion:

$$\frac{1}{(z+1)^2 z^2 (z-1)^2} = \frac{1/4}{(z+1)^2} + \frac{3/4}{z+1} + \frac{1}{z^2} + \frac{0}{z} + \frac{1/4}{(z-1)^2} - \frac{3/4}{z-1}.$$

Then, if we construct the matrices

$$\mathbf{B} = \begin{bmatrix} 0 & & & & & & \\ & 1 & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ & & & & 1 & & \\ & & & & & 1 & \\ & & & & & & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{A} = \begin{bmatrix} 0 & -1/4 & 3/4 & -1 & 0 & -1/4 & -3/4 \\ -\rho_{3,1} & 1 & 1 & 0 & 0 & 0 & 0 \\ -\rho_{3,0} & 0 & 1 & 0 & 0 & 0 & 0 \\ -\rho_{2,1} & 0 & 0 & 0 & 1 & 0 & 0 \\ -\rho_{2,0} & 0 & 0 & 0 & 0 & 0 & 0 \\ -\rho_{1,1} & 0 & 0 & 0 & 0 & -1 & 1 \\ -\rho_{1,0} & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix},$$

we have the relation

$$\det(z\mathbf{B} - \mathbf{A}) = p(z)\,. \tag{8.10}$$

Note the occurrence of Jordan blocks for each of the nodes τ_k, of size equal to the confluency of the node (here, each node has two pieces of information associated with it, so the blocks are each 2×2). In the Lagrange case, the Jordan block is just 1×1, so scalar. Note also that this formulation has the local Taylor coefficients in reverse order in the first column. Each local Taylor series is in the same rows as the corresponding block node τ_i. The generalized barycentric weights $\beta_{i,j}$ appear in the first row, in the columns corresponding to node τ_i. The general pattern should be clear from this example, with the proviso that confluency 3 at a node should give rise to a 3×3 Jordan block, and that mixed confluencies are possible. The original derivation of this matrix came from letting nodes $\tau_i \to \tau_j$, or "flow together," and performing a similarity transformation that replaced ρ_i with ${}^{(\rho_i - \rho_j)}/_{(\tau_i - \tau_j)}$, but a direct Schur complement proof is simpler. This matrix pencil has been coded in `gcmp.m` in MATLAB and `bhip.mpl` in MAPLE.

The reader is invited to prove this eigenvalue formula, perhaps using the Schur complement as suggested above. We could give the proof here, but it amounts to no more than a representative computation of the row vector of the $\beta_{k,j}$s for a fixed k times an inverse Jordan-like matrix times the column vector of the $\rho_{k,j}$ for the same k; the other blocks are the same and showing that this row-matrix-column product gives the same result as the inner two sums of the barycentric form then gives the complete barycentric form by adding all the blocks for each k. We believe it's better that you do it for yourself (a 3×3 block is quite convincing, and not too tedious). In any case, the generalized eigenvalues of the matrix pencil $(\mathbf{A}, \mathbf{B})$ are indeed the roots of $p(z)$.

Thus, we may find roots of polynomials expressed in the Hermite basis (or the Lagrange basis), by finding the eigenvalues of this matrix pencil, *without changing into the monomial basis*. The reason for that concern is, as usual, that a polynomial may be well-conditioned to evaluate (or rootfind) in the given Hermite interpolational basis, but very *poorly* conditioned in the monomial basis. Indeed, if the roots are outside the unit disk, this is almost guaranteed to be the case. ◁

Example 8.8. Continuing the Mandelbrot polynomial example, suppose again that $p_0(x) = 1$, and $p_{k+1}(x) = xp_k^2(x) + 1$ for $k \geq 0$. Then $\deg p_k(x) = 2^k - 1$, and if $\xi_j^{(k)}$ are the zeros of $p_k(x)$, then the next iterate $p_{k+1}(x)$ satisfies $p_{k+1}(\xi_j^{(k)}) = 1$ and $p'_{k+1}(\xi_j^{(k)}) = 0$; finally, $p_{k+1}(0) = p'_{k+1}(0) = 1$. This gives 2^{k+1} values for us to form the companion matrix pencil as above [which now is $(2^{k+1}+1) \times (2^{k+1}+1)$]. For $k = 10$, the eigenvalues are plotted in Fig. 8.4. This was constructed in MATLAB by starting with small k and working up; the eigenvalues were computed by a dense LAPACK eigenvalue routine called by MATLAB, and then polished by a single Newton iteration before constructing the matrix for the polynomial at stage $k+1$. In MAPLE it is possible to go farther, by using a little bit of extra precision; we got up to $k = 11$ or $k = 12$ before having to look for a better way. ◁

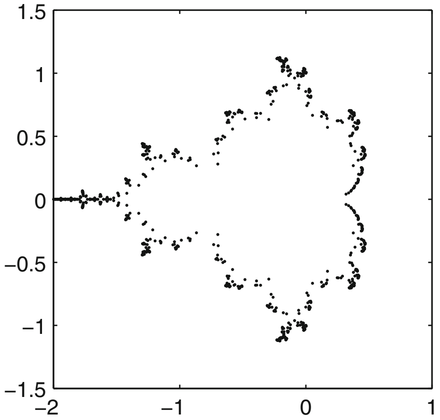


Fig. 8.4 Roots of the Mandelbrot polynomial of degree $2^{10} - 1 = 1023$, computed with eigenvalue techniques. This is the logo for the software `MPsolve` available at `http://www.dm.unipi.it/cluster-pages/mpsolve/`

Remark 8.2. The monomial basis form for $p_3(x)$ was simpler, to human eyes, than the Hermite interpolational basis form. What about $p_{12}(x)$? Computing the monomial expression explicitly in MAPLE is possible, but we get the message "[Length of output exceeds limit of 1,000,000]" when we try to display it. The coefficients are in some cases 722 digits long, and there are 4096 of them (to be sure, the leading coefficient is just 1). Of course, there are 4096 nontrivial (even complex!) coefficients in the Hermite interpolational basis, too, but they are each only 15 decimal digits long (actually, the "coefficients" are 1 or 0; it is the generalized barycentric weights that are 15 digits long—often times 2 as there are real and complex parts for each). Even so, the Hermite interpolational representation is quite a bit more compact, and although it is not exact, it is perfectly adequate to compute the zeros accurately, for example. In contrast, the monomial basis expression is so ill-conditioned that approximation of each coefficient to 15 digits, which brings a comparable compression, destroys almost all accuracy outside $|x| < 1$. ◁

Example 8.9. The Hermite interpolational basis is not quite as well-conditioned as the Lagrange basis. In order to use the Lagrange basis for the Mandelbrot polynomials, the following trick (due to Piers Lawrence) is quite interesting. The 2×2 *matrix polynomial* below has $\det \mathbf{P}_{k+1} = zp_k^2(z) + 1 = p_{k+1}(z)$:

$$\mathbf{P}_{k+1} = \begin{bmatrix} zp_k(z) & -1 \\ 1 & p_k(z) \end{bmatrix} .$$

Therefore, looking for z such that $\mathbf{P}_{k+1}$ is singular will find the zeros of $p_{k+1}(z)$. We may easily interpolate this matrix polynomial on a *Lagrange* basis at the $2^{k-1} - 1$ zeros of $p_k(z)$. Since the degree of this matrix polynomial is that of $zp_k(z)$ or 2^{k-1}, we need two more interpolation points to specify it: We choose $z = 0$, where $p_k(z) = 1$, and $z = -2$, where $p_k(z) = -1$ if $k \geq 1$. The companion matrix pencil now has

2×2 matrix entries where the $\rho_j = \mathbf{P}_k(\tau_j)$ and the $\beta_k\mathbf{I}_2$ go and similarly has diagonal 2×2 blocks $\tau_j\mathbf{I}_2$. Using this, we can get a little farther than we could with the Hermite basis.

Explicitly, the companion matrix pencil for $\mathbf{P}_3$ is given by the following data: The nodes $\boldsymbol{\tau} = [0, \omega_1, \omega_2, \omega_3, -2]$, where the ω_i are the roots of $p_2(z) = z(z+1)^2+1$ we have already seen, and the matrix values of the matrix polynomial are

$$\left[\begin{bmatrix} 0 & -1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 2 & -1 \\ 1 & -1 \end{bmatrix}\right].$$

The companion matrix pencil $(\mathbf{A}, \mathbf{B})$ consists of a pair of 12×12 matrices; $\mathbf{B}$ is the identity matrix except the top *two* diagonal entries are zero, and the matrix $\mathbf{A}$ is an arrowhead matrix with nonzero entries in the first two columns and the first two rows, but is zero in the 2×2 point of the arrow. The generalized eigenvalues of this pencil have *five* spurious infinite eigenvalues (which don't bother the computation) and seven good approximations to the roots of $p_3(z)$. As k increases, there are always five spurious eigenvalues and 2^k-1 good roots (up until about $k = 15$), so the extra cost of spurious eigenvalues is not significant. ◁

8.2.2 Derivation of the Barycentric Forms of the Hermite Interpolation Polynomial

This section is a conceptually simple but algebra-intensive derivation of Eq. (8.9). Except possibly for Remarks 8.3–8.4, this section can be omitted by those willing to take the algebra on faith. We also derive the first barycentric form of the Hermite interpolation polynomial. The derivations use an approach based on contour integrals and partial fractions.

To begin with, let

$$w(z) = \prod_{i=0}^{n}(z-\tau_i)^{s_i},$$

which is a generalization of what we had in the Lagrange case to the confluent case, and compute the partial fraction decomposition of $1/w(z)$:

$$\frac{1}{w(z)} = \sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\frac{\beta_{i,j}}{(z-\tau_i)^{j+1}}. \tag{8.11}$$

Now, consider the contour integral

$$\frac{1}{2\pi i}\oint_C \frac{p(z)}{(t-z)w(z)}dz, \tag{8.12}$$

where t is not equal to any τ_i. This integral is zero if the contour C includes t and all τ_i and if $\deg p \le d = -1 + \sum_{i=0}^{n} s_i$, since the degree of the denominator is $d+2$. We now compute the residues at $z = t$ and at all $z = \tau_i$. At $z = t$, we have

$$\operatorname{res}_t \frac{p(z)}{(t-z)w(z)} = -\frac{p(t)}{w(t)}.$$

Now, let us choose an arbitrary node, say τ_ℓ, and try to find the residues at $z = \tau_\ell$. This is achieved by finding series for each term in the product and multiplying them using Cauchy convolution (see Chap. 2). We then use the known partial fraction decomposition of ${}^1\!/_{w(z)}$ to obtain

$$\begin{aligned} \frac{p(z)}{(t-z)w(z)} &= \frac{p(z)}{t-z} \sum_{i=0}^{n} \sum_{j=0}^{s_i-1} \frac{\beta_{i,j}}{(z-\tau_i)^{j+1}} \\ &= \frac{p(z)}{t-z} \left(\sum_{j=0}^{s_\ell-1} \frac{\beta_{i\ell}}{(z-\tau_\ell)^{j+1}} + O(1) \right), \end{aligned} \tag{8.13}$$

where, for completeness although we don't need it, the $O(1)$ term as $z \to \tau_\ell$ is

$$\sum_{\substack{i=0 \\ i\neq\ell}}^{n} \sum_{j=0}^{s_i-1} \frac{\beta_{i,j}}{(z-\tau_i)^{j+1}} \sim \sum_{\substack{i=0 \\ i\neq\ell}}^{n} \sum_{j=0}^{s_i-1} \frac{\beta_{i,j}}{(\tau_\ell-\tau_i)^{j+1}} + O(z-\tau_\ell)$$

(we have used the fact that the τ_i are distinct here). Now, by Taylor expansion, we have

$$p(z) = \sum_{m\ge 0} \rho_{\ell,m} (z-\tau_\ell)^m, \tag{8.14}$$

since, being a polynomial, $p(z)$ is analytic; indeed, this is a finite sum. Also, we have

$$\frac{1}{t-z} = \frac{1}{t-\tau_\ell-(z-\tau_\ell)} = \frac{1}{t-\tau_\ell} \cdot \frac{1}{1-\dfrac{z-\tau_\ell}{t-\tau_\ell}}, \tag{8.15}$$

and because the second term is the sum of a geometric series,[4] we find that

$$\frac{1}{t-z} = \frac{1}{t-\tau_\ell} \sum_{m\ge 0} \left(\frac{z-\tau_\ell}{t-\tau_\ell} \right)^m = \sum_{m\ge 0} \frac{(z-\tau_\ell)^m}{(t-\tau_\ell)^{m+1}}. \tag{8.16}$$

Then, by Cauchy convolution,

[4] This series needs to be convergent for these manipulations to be valid. It is, for z close enough to τ_ℓ—in fact, for all z closer to τ_ℓ than t is. Since t is different from each node by hypothesis, this is possible.

$$\frac{p(z)}{t-z} = \sum_{m\geq 0} c_m (z-\tau_\ell)^m , \tag{8.17}$$

where

$$c_m = \sum_{j=0}^{m} \rho_{\ell,j} \frac{1}{(t-\tau_\ell)^{m-j+1}} = \sum_{j=0}^{m} \rho_{\ell,m-j} \frac{1}{(t-\tau_\ell)^{j+1}} .$$

Now, let $[z^k](f)$ be the coefficient of z^k in the series of f about $z=0$ (as in Graham et al. (1994)). Then,

$$\begin{aligned}
\operatorname{res}_{\tau_\ell} \frac{p(z)}{(t-z)w(z)} &= \left[(z-\tau_\ell)^{-1}\right] \left(\frac{p(z)}{(t-z)w(z)}\right) \\
&= \left[(z-\tau_\ell)^{-1}\right] \left(\left(\sum_{m\geq 0} c_m (z-\tau_\ell)^m\right) \left(\sum_{j=0}^{s_\ell-1} \frac{\beta_{\ell,j}}{(z-\tau_\ell)^{j+1}}\right)\right) \\
&= \left[(z-\tau_\ell)^{-1}\right] \left(\left(\sum_{m\geq 0} c_m (z-\tau_\ell)^m\right) \left(\frac{\sum_{j=0}^{s_\ell-1} \beta_{\ell,j}(z-\tau_\ell)^{s_\ell-j-1}}{(z-\tau_\ell)^{s_\ell}}\right)\right) \\
&= \left[(z-\tau_\ell)^{s_\ell-1}\right] \left(\left(\sum_{m\geq 0} c_m (z-\tau_\ell)^m\right) \left(\sum_{j=0}^{s_\ell-1} \beta_{\ell,j}(z-\tau_\ell)^{s_\ell-1-j}\right)\right) .
\end{aligned}$$

Now, we let $k = s_\ell - 1 - j$ to change the order of summation in order to apply Cauchy convolution:

$$\begin{aligned}
&= \left[(z-\tau_\ell)^{s_\ell-1}\right] \left(\left(\sum_{m\geq 0} c_m (z-\tau_\ell)^m\right) \left(\sum_{k=0}^{s_\ell-1} \beta_{\ell,s_\ell-1-k}(z-\tau_\ell)^{k}\right)\right) \\
&= \left[(z-\tau_\ell)^{s_\ell-1}\right] \left(\sum_{m\geq 0} \left(\sum_{k=0}^{m} c_{m-k}\beta_{\ell,s_\ell-1-k}\right) (z-\tau_\ell)^m \right) \\
&= \sum_{k=0}^{s_\ell-1} c_{s_\ell-1-k}\beta_{\ell,s_\ell-1-k} .
\end{aligned} \tag{8.18}$$

If we let $q = s_\ell - 1 - k$, we obtain

$$\operatorname{res}_{\tau_\ell} \frac{p(z)}{(t-z)w(z)} = \sum_{q=0}^{s_\ell-1} c_q \beta_{\ell,q} = \sum_{q=0}^{s_\ell-1} \beta_{\ell,q} \sum_{j=0}^{q} \rho_{\ell,j}(t-\tau_\ell)^{j-q-1} ,$$

and, after relabeling the indices $\ell \to i$, $q \to j$, $j \to k$, we obtain

$$\operatorname{res}_{\tau_i} \frac{p(z)}{(t-z)w(z)} = \sum_{j=0}^{s_i-1} \beta_{i,j} \sum_{k=0}^{j} \rho_{i,k}(t-\tau_i)^{k-j-1} . \tag{8.19}$$

We can then conclude that the sum of residues is zero, that is,

$$\begin{aligned} 0 &= -\frac{p(t)}{w(t)} + \sum_{i=0}^{n} \operatorname{res}_{\tau_i} \frac{p(z)}{(t-z)w(z)} \\ &= -\frac{p(t)}{w(t)} + \sum_{i=0}^{n} \sum_{j=0}^{s_i-1} \sum_{k=0}^{j} \beta_{i,j}\rho_{i,k}(t-\tau_i)^{k-j-1}\,, \end{aligned} \tag{8.20}$$

as desired. Rearranging terms, we obtain

Definition 8.4. The expression of $p(t)$ as

$$p(t) = w(t) \sum_{i=0}^{n} \sum_{j=0}^{s_i-1} \sum_{k=0}^{j} \beta_{i,j}\rho_{i,k}(t-\tau_i)^{k-j-1} \tag{8.21}$$

is the *first barycentric form of the Hermite interpolation polynomial.* ◁

Remark 8.3. As a result, if we can compute the partial fraction decomposition of $1/w(z)$, then we have the generalized barycentric weights, and so we can solve the Hermite interpolation problem. ◁

Remark 8.4. It is occasionally useful to rearrange that barycentric form to arrive at an explicit expression for the entries of the Hermite interpolational basis, as follows:

$$H_{i,j}(z) = \sum_{\ell=j}^{s_i-1} \beta_{i,\ell} w(z)(z-\tau_i)^{j-\ell-1} \tag{8.22}$$

$$= \sum_{k=0}^{s_i-1-j} \beta_{i,j+k} w(z)(z-\tau_i)^{-k-1}\,. \tag{8.23}$$

To derive these, simply interchange the order of summation of the inner two sums in (8.21). Note that the apparent division by powers of $(z-\tau_i)$ is exact cancellation, because $w(z)$ contains s_i factors of $(z-\tau_i)$. One can thus write down

$$H_{i,j}(z) = \left(\sum_{k=0}^{s_i-1-j} \beta_{i,j+k}(z-\tau_i)^{s_i-k-1}\right) \prod_{\substack{\ell=0\\ \ell\neq i}}^{n} (z-\tau_\ell)^{s_\ell} \tag{8.24}$$

$$= (z-\tau_i)^j \left(\sum_{m=0}^{s_i-1-j} \beta_{i,s_i-1-m}(z-\tau_i)^m\right) w_i(z) \tag{8.25}$$

as an explicit polynomial form for $H_{i,j}(z)$ if desired. ◁

Example 8.10. In Chap. 2, we computed the partial fraction decomposition of $1/\theta^2(\theta-1)^2$, namely,

$$\frac{1}{\theta^2(\theta-1)^2} = \frac{1}{\theta^2} + \frac{2}{\theta} - \frac{2}{\theta-1} + \frac{1}{(\theta-1)^2}\,. \tag{8.26}$$

This can be used to find the unique cubic polynomial passing through the Hermite data $f(\tau_i) = \rho_{i,0}$, $f'(\tau_i) = \rho_{0,1}$ for $i = 0,\ 1$. Let $\theta = {}^{(t-\tau_0)}/_{(\tau_1-\tau_0)} = {}^{(t-\tau_0)}/_{h}$, and $p(\theta) = f(t)$ so that ${}^{dp}/_{d\theta} = hf'(t)$. In θ form, the interpolating polynomial $p(\theta)$ is

$$\theta^2(\theta-1)^2\left(2\,\frac{\rho_{0,0}}{\theta}+\frac{\rho_{0,0}+h\rho_{0,1}\theta}{\theta^2}-2\,\frac{\rho_{1,0}}{\theta-1}+\frac{\rho_{1,0}+h\rho_{1,1}(\theta-1)}{(\theta-1)^2}\right) \tag{8.27}$$

in first barycentric form; putting $\theta = (t-\tau_0)/h$ and simplifying, we get

$$\begin{aligned} f(t) = (t-\tau_0)^2(t-\tau_1)^2\Bigg(&2\,\frac{\rho_{0,0}}{(\tau_1-\tau_0)^3(t-\tau_0)}+\frac{\rho_{0,0}+\rho_{0,1}(t-\tau_0)}{(\tau_0-\tau_1)^2(t-\tau_0)^2}\\ &-2\,\frac{\rho_{1,0}}{(\tau_1-\tau_0)^3(t-\tau_1)}+\frac{\rho_{1,0}+\rho_{1,1}(t-\tau_1)}{(\tau_0-\tau_1)^2(t-\tau_1)^2}\Bigg)\,. \end{aligned}$$

After collecting the coefficients $\rho_{i,j}$, so as to get explicit expressions for the cubic Hermite interpolational basis, we get

$$\begin{aligned} f(t) = &-\frac{(2t-3\,\tau_0+\tau_1)(t-\tau_1)^2\rho_{0,0}}{(\tau_0-\tau_1)^3}+\frac{(t-\tau_0)(t-\tau_1)^2\rho_{0,1}}{(\tau_0-\tau_1)^2}\\ &+\frac{(2t-3\,\tau_1+\tau_0)(t-\tau_0)^2\rho_{1,0}}{(\tau_0-\tau_1)^3}+\frac{(t-\tau_0)^2(t-\tau_1)\rho_{1,1}}{(\tau_0-\tau_1)^2}\,. \end{aligned}$$

The reader should work through the details of the transformation from Eqs. (8.26) to (8.27). These formulæ will prove useful. In Exercise 8.40, the reader will be asked to do a similar computation for *quintic* Hermite interpolation, which matches all derivatives up to the second derivative at either end. ◁

We obtain the second barycentric form simply by interpolating $p(t) = 1$ and then using the result as denominator, as we have done in the Lagrange case. Since

$$\begin{aligned} 1 &= w(t)\sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\sum_{k=0}^{j}\beta_{i,j}\delta_0^k(t-\tau_i)^{k-j-1}\\ &= w(t)\sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\beta_{i,j}(t-\tau_i)^{-j-1}\,, \end{aligned} \tag{8.28}$$

dividing (8.21) by (8.28) gives Eq. (8.9). Once we have discussed rational interpolation, we will take up the accuracy of this form.

The MATLAB program `hermiteval` implements an extension of Algorithm 8.1 to the Hermite interpolational case. It is available in the code repository for this book. Given the data ρ, a set of values t, the nodes τ and their confluences s, the barycentric weights β, and possibly a differentiation matrix **D** (the theory of which is discussed later in Chap. 11), this program evaluates the Hermite interpolant at the given values t and its derivative if required.

8.2.3 Computing the Generalized Barycentric Weights

The MATLAB program `genbarywts` implements the method of computing partial fractions discussed in Sect. 2.7.

```
function [w,D] = genbarywts( tau, s_in )
%
% [w,D] = genbarywts( tau, s, <optional Taylor=true> )
% Generalized  Barycentric weights
% (c)  Robert M. Corless, 2007, revised 2010
% and (optionally) Differentiation matrix
% in Taylor form, so
% D[rho; rho'; rho''/2] ==> [rho'; rho''; rho'''/2]
%
% on distinct nodes tau with integer confluencies s
% size(w)  = [n,smax]
% [n,1] = size(tau(:))
% smax = max(s)
```

It may be useful to inspect this code, which can be found in its entirety in the code repository for this book. The second half of the code, which computes the *differentiation matrix* associated with the given set of interpolation nodes, is explained later in Chap. 11.

Example 8.11. Let us use Hermite interpolation on the exponential function $w = \exp(z)$. In the first instance, let us use Chebyshev nodes of the 2nd kind $\eta_k = \cos(\pi k/n)$ for $0 \le k \le n$ to approximate $\exp(z)$ on the line $-1 \le z \le 1$. Take $n = 4$ and consider increasing confluencies $s = 1, 2, 3, \ldots$. Since the derivative of $\exp(z)$ is just $\exp(z)$, once we have some accurate values of $\exp(z)$ on these nodes, we ought to be able to build more and more accurate interpolants merely by increasing the degree.

Indeed, this works quite well. For confluency $s = 1$, we already have about 5 digits of accuracy on the interval $-1 \le z \le 1$, while for confluency $s = 2$, we have about 10-digit accuracy. For confluencies 3 and 4, the interpolant is essentially accurate to machine precision. The degrees of the interpolants are 4 for $s = 1$, 9 for $s = 2$, 14 for $s = 3$, and 19 for $s = 4$. The code is shown below:

```
%% Hermite interpolation of the exponential on -1 <= z <= 1
%
% We use $\eta_k = \cos( \pi k /n )$ with $n=4$ and increasing
% confluencies in an experimental fashion.
%
n = 15;
tau = [cos( pi*(n:-1:0)/n ) ];
z = linspace(-1,1,2012);
```

```
yref = exp(z);
vals = exp( tau );
niters = 4;
yint = zeros( niters, length(z) );
ypint= zeros( niters, length(z) );
errs = zeros( size(yint) );
errps= zeros( size(errs) );
for s=1:niters,
    fac = ones(s,1);
    for j=3:s,
        fac(j) = fac(j-1)/(j-1);
    end
    rho = fac*vals; % Local Taylor coeffs have factorials!
    [w,D] = genbarywts( tau, s );
    [y,yp] = hermiteval( rho, z, tau, s, w, D );
    yint(s,:) = y;
    ypint(s,:) = yp;
    errs(s,:) = yint(s,:)./yref - 1;
    errps(s,:) = ypint(s,:)./yref - 1;
end
figure(1), semilogy( z, abs(errs), 'k' ), set(gca,'fontsize',16)
xlabel('z'), ylabel('y/exp(z)␣-␣1')
figure(2), semilogy( z, abs(errps), 'k' ), set(gca,'fontsize',16)
xlabel('z'), ylabel('y''/exp(z)-1')
```

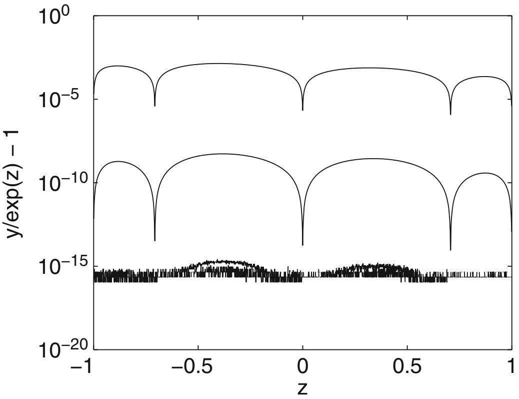


Fig. 8.5 Error in interpolating $y = \exp(x)$ at five nodes with varying confluencies; as the confluency (and degree of the interpolant) increases, so does the accuracy

The results are in Figs. 8.5 and 8.6. In contrast, a degree-15 Taylor approximation at $z = 0$ to the exponential function has relative error about 1.3×10^{-13} at $z = -1$, which is a bit worse but not that much worse.

If we instead use complex nodes, $\tau_k = \exp(2\pi ik/(n+1))$ for $0 \le k \le 4$, then the code is virtually identical. We evaluate the error on the rim of the unit circle and by the maximum principle we will have bounded the error everywhere on the unit disk. ◁

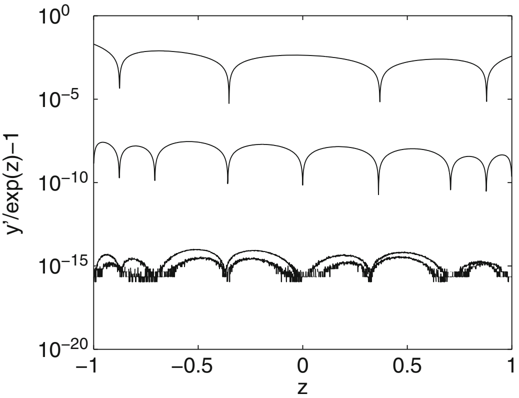


Fig. 8.6 Forward error in the derivative of the interpolant for varying confluencies. Higher confluency and thus higher degree again give better accuracy, but we see in comparison with Fig. 8.5 that the derivative of the interpolant is not as accurate as the interpolant of the function itself

8.3 Conditioning of the Generalized Barycentric Weights

One may wonder how the barycentric weights $\beta_{i,j}$ change as the τ_k are varied. Computing $\partial\beta_{i,j}/\partial\tau_k$ can be done simply by another partial fraction decomposition, as follows. First, let $w(z) = \prod_{i=0}^{n}(z-\tau_i)^{s_i}$ as usual. Then we find that

$$\begin{aligned} \frac{\partial}{\partial\tau_k}\frac{1}{w(z)} &= \frac{s_k}{(z-\tau_k)w(z)} \\ &= -\sum_{j=0}^{s_k-1}\frac{(j+1)\beta_{k,j}}{(z-\tau_k)^{j+2}} + \sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\frac{\partial\beta_{i,j}/\partial\tau_k}{(z-\tau_i)^{j+1}}\,. \end{aligned} \tag{8.29}$$

The derivatives can then be read off from the partial fraction decomposition of the term on the left. This allows us to see how well-conditioned (or ill-conditioned) the barycentric weights themselves are.

Example 8.12. Consider again the cubic Hermite interpolation partial fraction decomposition. But this time, fix one of the nodes at $\tau_0 = 0$ and put the other at $\tau_1 = h$, and consider what happens as h varies:

$$\frac{1}{z^2(z-h)^2} = \frac{1/h^2}{z^2} + \frac{2/h^3}{z} - \frac{2/h^3}{z-h} + \frac{1/h^2}{(z-h)^2}.$$

Now, $\beta_{0,0}$ is the coefficient of $1/z$ in the above, namely, $2/h^3$. Clearly, $\partial\beta_{0,0}/\partial\tau_1 = -6/h^4$. If we had computed the partial fraction expansion of

$$
\begin{aligned}
\frac{\partial}{\partial h}\left(\frac{1}{z^2(z-h)^2}\right) &= \frac{2}{z^2(z-h)^3} \\
&= -\frac{6}{h^4 z} - \frac{4}{h^3\,(z-h)^2} + \frac{2}{h^2\,(z-h)^3} - \frac{2}{h^3 z^2} + \frac{6}{h^4\,(z-h)}\,,
\end{aligned}
$$

then the same information can be read directly off from the coefficient of ${}^1\!/_z$, again ${}^{-6}\!/_{h^4}$. The advantage of this procedure instead of symbolic differentiation is that the same MATLAB program that is used to generate the $\beta_{i,j}$ can be used to generate the derivatives.

Note here that the *condition number* of the barycentric weight $\beta_{0,0}$ is (via $C = {}^{zf'}\!/_f$, which is the formula for the condition number of evaluation of a function $f(z)$ as explained previously) $\tau_1 {}^{\beta'_{0,0}}\!/_{\beta_{0,0}}$; that is, $h({}^{-6}\!/_{h^4})/({}^2\!/_{h^3}) = -3$. Therefore, in spite of the singularity of the coefficients as $h \to 0$, this barycentric weight is actually well-conditioned—similarly for the other three. This fact might be surprising (it was to us, at first), but it is quite reassuring: Cubic Hermite interpolation in barycentric form is well-conditioned for all distinct node pairs. In Exercise 8.6, you are asked to look at some more complicated cases, where ill-conditioning may be present. ◁

8.4 Condition Number of the First Barycentric Form of the Lagrange Interpolating Polynomial

We quoted in Sect. 2.2.6.1 the backward stability result for the numerical evaluation of a polynomial given in the first barycentric form: For each x at which we evaluate the polynomial, the first form gives the *exact* value of a polynomial whose coefficients (that is, values at the nodes) are changed by at most a small multiple of the unit roundoff. What, then, is the sensitivity of the polynomial to such changes?

We begin by repeating here the analysis of the conditioning of the Lagrange polynomial. Suppose we change ρ_k to $\rho_k(1+\delta_k)$, with $|\delta_k| \le \varepsilon$. Then,

$$
(p+\Delta p)(t) - p(t) = w(t)\sum_{k=0}^{n} \frac{\beta_k \rho_k \delta_k}{t-\tau_k} \tag{8.30}
$$

and

$$
|\Delta p(t)| \le |w(t)| \left(\sum_{k=0}^{n} \frac{|\beta_k \rho_k|}{|t-\tau_k|} \right) \varepsilon . \tag{8.31}
$$

But this is just our old friend

$$
B(t) = \sum_{k=0}^{n} |c_k||\phi_k(t)|
$$

from Chap. 2, where $c_k = \rho_k$ and $\phi_k = L_k$.

As shown by Farouki and Goodman (1996), the Bernstein–Bézier polynomial basis on a given interval is *optimally conditioned*, in the following sense. As discussed in Remark 2.8, no other nonnegative basis yields systematically smaller condition numbers for evaluation or rootfinding. This result follows from a beautiful partial ordering induced on nonnegative polynomial bases by *nonnegative change-of-basis* matrices: If a nonnegative basis ϕ can be constructed from a nonnegative ψ by a nonnegative change-of-basis matrix, in which case we say $\phi \succ \psi$, then ψ will have no worse conditioning than ϕ (and possibly quite a bit better). The nonnegativity of the bases is also useful numerically. This is a strong result: Bernstein–Bézier bases really are the best, in this useful sense.

Now, let us explore the Lagrange and Hermite interpolational bases in this context. We will *relax the nonnegativity condition* and show that by doing so we can achieve (on a subset) an even better condition number than can be obtained by the Bernstein–Bézier basis. For the moment, we consider only polynomials defined on the interval $[0,1]$.

Lemma 8.1. *Both the Bernstein–Bézier and the power basis functions can be expressed as a nonnegative combination of any Lagrange basis with nodes taken on* $[0,1]$. *By* nonnegative combination, *we mean that each coefficient in the combination is nonnegative. Of course, there must be some positive coefficients.*

Proof. Elements x^k of the power basis may be written as

$$x^k = \sum_{i=0}^{n} x_i^k L_i(x)$$

and the coefficients x_i^k are obviously nonnegative. Similarly, elements $b_k^n(x)$ of the Bernstein–Bézier basis may be written as

$$b_k^n(x) = \binom{n}{k}(1-x)^{n-k}x^k = \sum_{i=0}^{n}\binom{n}{k}(1-x_i)^{n-k}x_i^k L_i(x)$$

and, again, the coefficients are obviously nonnegative since $0 \le x_i \le 1$. ♮

Remark 8.5. The power basis functions can be expressed as a nonnegative combination of any Hermite interpolational basis with nodes taken on $[0,1]$, since the slopes of all x^k are positive on this interval; however, some slopes of Bernstein–Bézier bases are negative, so Bernstein–Bézier bases cannot be so expressed as a nonnegative combination of Hermite interpolation bases if any $s_k > 1$. ◁

Lemma 8.2. *The Lagrange polynomials* $L_i(x)$ *are nonnegative on the interpolation points.*

Proof. This is obvious: They take on only the values 0 or 1 on the interpolation points. ♮

Remark 8.6. We would like nonnegativity in an open set around the interpolation points, which we do not have. However, sometimes we get something nearly as good, as we will see. ◁

Theorem 8.1. *Fix a set of interpolation points* $[x_0, x_1, \ldots, x_n]$. *If any basis* $\phi(z)$ *can be expressed as a nonnegative combination of the Lagrange basis on this set of points (this ensures* ϕ *itself is nonnegative there), then there exists a set* T*, depending on* f *and containing the interpolation points and hence not empty, in which* $B_\phi(f,T) \succ B_{\text{Lagrange}}(f,T)$*; that is, the Lagrange basis expression is at least as well conditioned as* ϕ *is for all* $z \in T$. *This includes the case where* ϕ *is the Bernstein–Bézier basis. If, further, the inequality is strict on an interpolation point, that is then the set* T *has a nonempty interior.*

Proof. The proof is an elegant consequence of the triangle inequality, as follows. Let $\mathbf{A}$ be the (nonnegative) matrix of change of basis from the Lagrange basis to ϕ, so that $[\phi_0(z), \ldots, \phi_n(z)] = [L_0(z), \ldots, L_n(z)]\mathbf{A}$. Then since $\mathbf{A}$, ϕ, and L_i are nonnegative on the interpolation points, we have for every x_k

$$\sum_{j=0}^{n} |c_j \phi_j(x)| = \sum_{j=0}^{n} |c_j| \phi_j(x) = \sum_{i=0}^{n} \left(\sum_{j=0}^{n} |c_j| a_{ij} \right) L_i(x) \geq \sum_{i=0}^{n} \left| \sum_{j=0}^{n} c_j a_{ij} \right| L_i(x) \,.$$

Therefore, T is not empty, containing at least all x_k.

If, for some interpolation point, say x_k, the inequality is strict, then we observe that points near x_k also belong to T, because all the terms in the inequality are continuous. This establishes that the set T has nonempty interior if the inequality is strict at any interpolation point. ♮

Remark 8.7. The relative size of T compared to the region of interpolation is interesting. In Fig. 8.7, we plot the sign of the difference $B_{\text{Bernstein}}(W_1, t) -$

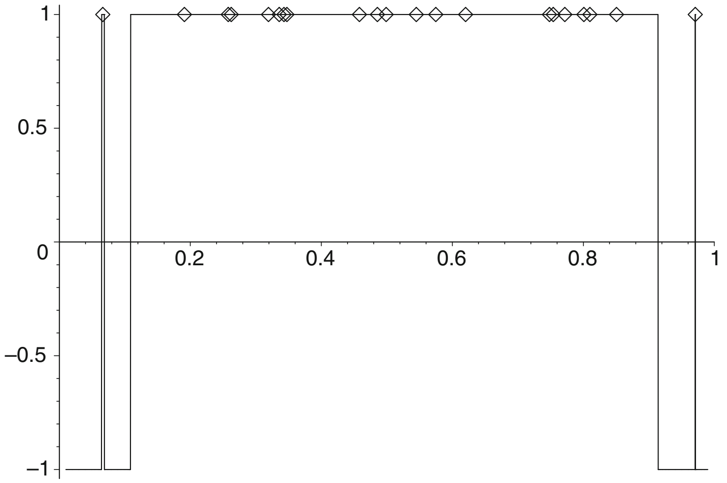


Fig. 8.7 The sign of the difference between the condition number in the Bernstein–Bézier basis and in a Lagrange basis for the first Wilkinson polynomial. The set T, where the Lagrange basis is better than the Bernstein basis, is precisely the set of x-values where the sign is positive

$B_{\text{Lagrange}}(W_1, t)$ for a Lagrange basis on nodes chosen "at random" from a uniform distribution on the interval. The set T is exactly the set where this graph is nonnegative. Note that the set contains a large region around the interior interpolation points, but only a small region around the two points near the edge of the interval. ◁

Remark 8.8. Since the monomial basis has nonnegative values and nonnegative derivatives on $[0,1]$, it can be expressed as a nonnegative combination of any Hermite interpolational basis using points in that interval. The theorem above thus shows that *any* such Hermite interpolational basis will not be worse-conditioned (near the nodes) than the monomial basis is. To our knowledge, this fact has not been noted in the literature. ◁

To compare a Hermite interpolational basis with the Bernstein–Bézier basis is, however, more difficult. There exist examples for which a Hermite interpolational basis is uniformly better; there exist other examples where the Bernstein–Bézier basis is uniformly better. Indeed, there does not exist a nonnegative change-of-basis matrix either way, in general, so the theorem does not apply.

Example 8.13. For instance, take $W(z) = (z - {}^1\!/\!_4)^2 (z - {}^1\!/\!_2)^2 (z - {}^3\!/\!_4)^2$. If we express it in the monomial basis it is

$$z^6 - 3z^5 + \frac{29}{8}z^4 - \frac{9}{4}z^3 + \frac{193}{256}z^2 - \frac{33}{256}z + \frac{9}{1024},$$

while in the Bernstein–Bézier basis it is

$$p(z) = \sum_{k=0}^{n} a_k \binom{n}{k} (1-z)^{n-k} z^k$$

with the vector of coefficients

$$\mathbf{a} = \left[\frac{9}{1024}, -\frac{13}{1024}, \frac{247}{15360}, -\frac{89}{5120}, \frac{247}{15360}, -\frac{13}{1024}, \frac{9}{1024} \right].$$

We have already seen the expression in the Hermite interpolational basis on the nodes $[0, {}^1\!/\!_4, {}^3\!/\!_4, 1]$ with confluency 2 in Example 8.6. We compare the relative condition number for evaluation of each of these bases $B(z)/|p(z)|$ in Fig. 8.8. The Hermite basis is best for this example, followed by the Bernstein–Bézier basis, and the monomial basis is worst (as it often is). ◁

We end this section with one more important fact:

Theorem 8.2. *If we choose n of our $n+1$ interpolation points to be the roots, then $B_{\text{Lagrange}}(f,r) = 0$ at the zeros. That is, if we are lucky enough to interpolate at all the roots, the conditioning of the zeros is perfect.*

Proof. This is a simple computation. The coefficients of the expansion in the Lagrange basis are, except for one coefficient (say y_0), all zero: $y_k = 0$ for $1 \leq k \leq n$. Therefore, the expression for the condition of any x becomes $|y_0 L_0(x - r_1)$

$(x-r_2)\cdots(x-r_n)|$, and this is obviously zero at each root r_k. More, the *relative* condition number $B(z)/|f(z)|$ is just 1, everywhere—expressed in this basis, the polynomial is perfectly conditioned to evaluate, perfectly conditioned, that is, with respect to small relative changes in its coefficients. ♮

Theorem 8.1 is in some sense the entire reason for which Lagrange (and Hermite) interpolation is interesting at all. What it says is that it is *often true* that polynomials expressed in the Lagrange basis are very well-conditioned. It cannot *always* be true, of course, by the theorem of Farouki and Goodman: Bernstein–Bézier bases are genuinely optimal, overall and without regard to the particular polynomial being considered. However, for a *particular* polynomial, and for a *particular* set of nodes and confluencies defining a Lagrange or Hermite basis, the conditioning may be better even than in the Bernstein–Bézier basis! That this happens sometimes is perhaps not too surprising—that it happens so often is remarkable.

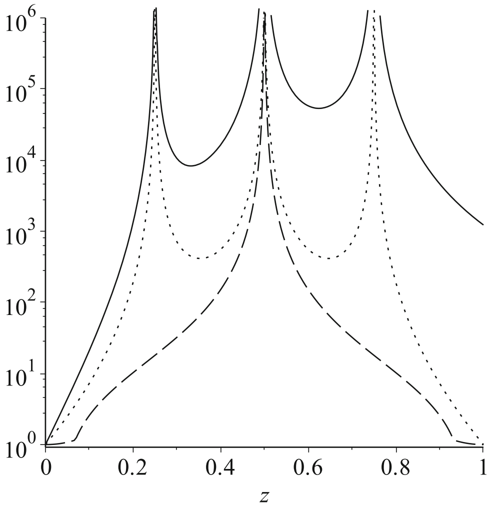


Fig. 8.8 The monomial basis condition number for $W(z) = (z-{}^1\!/\!_4)^2(z-{}^1\!/\!_2)^2(z-{}^3\!/\!_4)^2$ versus the condition number for a Hermite interpolational basis with confluency 2 on each of $[0,1/4,3/4,1]$. The Hermite interpolational basis (*dashed line*) is significantly better even away from the two zeros where it is exact, and better even near the zero where it is inexact. At the right endpoint, away from all zeros, the monomial basis is many orders of magnitude worse. The Bernstein–Bézier basis (*dotted line*) is intermediate in quality: not so good here as the Hermite interpolational basis, but much better than the monomial basis. For other examples, the Bernstein–Bézier basis can be better than the Hermite interpolational basis

8.5 Error in Polynomial Interpolation

Up to this point, we have discussed formulae to fit a polynomial to data (values or consecutive derivatives):

$$p(z) = w(z) \sum_{i=0}^{n} \sum_{j=0}^{s_i-1} \sum_{k=0}^{j} \beta_{i,j} \rho_{ik} (z-\tau_i)^{k-j-1} ,$$

where

$$w(z) = \prod_{i=0}^{n} (z-\tau_i)^{s_i} ,$$

the degree bound $d = -1 + \sum_{i=0}^{n} s_i$ is the maximum possible degree of the polynomial $p(z)$, and the generalized barycentric weights $\beta_{i,j}$ are just the coefficients in the partial fraction expansion of $1/w(z)$, that is,

$$\frac{1}{w(z)} = \sum_{i=0}^{n} \sum_{j=0}^{s_i-1} \frac{\beta_{i,j}}{(z-\tau_i)^{j+1}} ,$$

which can be stably computed in $O(d^2)$ flops. Now, what if our data aren't really from a polynomial? How much error do we make in using the polynomial $p(z)$ instead of the "true" underlying function [call it, say, $f(z)$]? We have supposed our data to be exact at the nodes τ_i. Hence, we also expect

$$f(z) - p(z) = w(z)K(z) , \tag{8.32}$$

for some function $K(z)$ that ought to be "reasonably" behaved [note $w(z)$ and all the correct derivatives are zero at $z = \tau_i$]. What can we say about $K(z)$?

Let us consider the real case, first. Fix z (real, and also assume that all nodes τ_i are real) for the moment and consider

$$E(t) = f(t) - p(t) - w(t)K(z), \tag{8.33}$$

where we will assume that everything there has enough derivatives for the following argument. Note $K(z)$ contains the constant z, not the variable t. We differentiate $d+1$ times with respect to t, to get [because $p(t)$ is polynomial of degree at most d, and the $(d+1)$st derivative of $w(t)$ is $(d+1)!$]

$$E^{(d+1)}(t) = f^{(d+1)}(t) - (d+1)!\,K(z) . \tag{8.34}$$

Now $E(t)$ vanishes at least $d+2$ times—at $t = z$ and at $t = \tau_i$ with confluency s_i—and we can therefore use the extended Rolle theorem to claim that there exists a $\hat{t}$ such that the $(d+1)$st derivative of E vanishes there. This allows us to identify the constant $K(z)$:

$$K(z) = \frac{1}{(d+1)!} f^{(d+1)}(\hat{t}) . \tag{8.35}$$

Thus, the *interpolation error* is

$$f(t) - p(t) = w(t) \frac{1}{(d+1)!} f^{(d+1)}(\hat{t}) , \tag{8.36}$$

for some (unknown) $\hat{t}$ (that depends on t) in an interval containing the nodes τ_i. Like the mean value theorem, this allows us to say something about the interpolation error if we can bound the size of the $(d+1)$st derivative of the "true" function $f(t)$. In general terms, the interpolation error will be small whenever the $(d+1)$st Taylor coefficients of $f(t)$ are not too large, while simultaneously $w(t)$ is small. Notice that this splits consideration into two parts: $w(t)$ depends only on the nodes, and $K(z)$ depends on the function (and weakly on the nodes, so it isn't a perfect split).

Remark 8.9. Observe that since

$$\begin{aligned}|fl(p(z)) - f(z)| &= |fl(p(z)) - p(z) + p(z) - f(z)| \\ &\le |fl(p(z)) - p(z)| + |f(z) - p(z)|\,, \end{aligned} \tag{8.37}$$

the conditioning of the representation of p also plays a role in the first term. ◁

Example 8.14. Consider the case of just two nodes, $\boldsymbol{\tau} = [a, a+h]$, with confluency 2. This example foreshadows Sect. 8.9. This formulation gives us a cubic polynomial with Hermite interpolation of function values $\rho_{k,0}$ and derivative values $\rho_{k,1}$ at each end: $\tau_0 = a$ and $\tau_1 = a+h$ for $k = 0$ and $k = 1$. Here $w(z) = (z-a)^2(z-a-h)^2$. If we put $z = a + \theta h$ so $0 \le \theta \le 1$ in the interior of our interval, we have that $w(z) = \theta^2(\theta-1)^2h^4$. Notice that $0 \le \theta^2(\theta-1)^2 \le 1/16$ on this interval. Suppose that the fourth derivative of $f(z)$ is bounded by $4!\,T_4$, that is, $|f^{(4)}(\hat{t})|/4! \le T_4$. Then the error in interpolating $f(z)$ over $a \le z \le a+h$ is at most $h^4T_4/16$. The important part is that it is $O(h^4)$.

By assuming that $\hat{t}(z)$ is a differentiable function of z and taking derivatives, we see that

$$f'(z) - p'(z) = w'(z)K(\hat{t}(z)) + w(z)K'(\hat{t})\hat{t}'(z)\,,$$

and a short computation shows that $w'(z) = 2\,\theta\,(2\,\theta-1)\,(\theta-1)\,h^3 = O(h^3)$ in this interval, so that the error in the derivative is $O(h^3) + O(h^4) = O(h^3)$. That is, taking derivatives reduces the order of accuracy, but still allows for some accuracy. ◁

Remark 8.10. What about *outside* the interval? This is known as extrapolation, and the error formula predicts that the error may (and probably will) grow very rapidly. The function $w(z)$ is small *only* on the interval defined by the τ_ks. Outside that interval, it grows as a power of z^{d+1}, and indeed, the amount of error in practice can be startling. *Extrapolation by polynomials is of limited use, unless something special is known about the function $f(z)$.* ◁

In the complex case, we do things a little differently. Assume now that $f(z)$ is analytic in a domain containing the interpolation nodes. Notice that $p(z)$, the interpolating polynomial, has degree d, where $1 + d = \sum_{i=0}^{n} s_i$. Therefore,

$$\frac{1}{2\pi i}\oint_C \frac{p(\zeta)}{w(\zeta)(\zeta - t)}\,d\zeta = 0 \tag{8.38}$$

for any contour C large enough to enclose t and all the nodes τ_i. Also, since f agrees with p to order $s_i - 1$ at τ_i, $(f(\zeta)-p(\zeta))/w(\zeta)$ is analytic at $\zeta = \tau_i$ and hence

$$\frac{f(z)-p(z)}{w(z)} = \frac{1}{2\pi i}\oint_C \frac{f(\zeta)-p(\zeta)}{w(\zeta)(\zeta - z)}\,d\zeta \tag{8.39}$$

by the residue theorem. Therefore,

$$f(z)-p(z) = \frac{w(z)}{2\pi i}\oint_C \frac{f(\zeta)}{w(\zeta)(\zeta - z)}\,d\zeta\,, \tag{8.40}$$

which gives a contour integral representation for what we were calling $K(z)$ earlier. We can in fact write this as a *divided difference* (we here introduce a standard notation although we don't use this in this book; the definition of the notation is left to the reader to work out in Problem 8.13):

$$\frac{1}{2\pi i}\oint_C \frac{f(\zeta)}{w(\zeta)(\zeta - z)}\,d\zeta = f\left[\tau_0^{s_0}, \tau_1^{s_1}, \ldots, \tau_n^{s_n}; t\right]\,. \tag{8.41}$$

This can be interpreted as something like a $(d+1)$st Taylor coefficient, when that makes sense. The divided-difference formula is just an identity, and so it is valid for quite general $f(z)$, not necessarily analytic f, even when the contour integral is not valid.

Remark 8.11. The above theorem can be used to show that the derivative of the interpolant also matches the derivative of the function (although not quite so well). Loosely speaking, if the function is approximated to error $O(h^p)$, then the derivative will be approximated to $O(h^{p-1})$. This fact will be used repeatedly in the chapters on numerical solution of differential equations. The reduction comes because $w(z)$ is replaced by $w'(z)$, and if, as is usual, $w(z)$ is a product of factors of the form $z-\tau_i$ where each $z-\tau_i = O(h)$ and there are p factors, then the derivative involves only products of $p-1$ factors of this size. ◁

Remark 8.12. The Chebyshev polynomials $T_k(z)$ have a remarkable minimal norm property: Over all monic polynomials, $T_k(z)/2^{k-1}$ has the smallest possible infinity norm on $-1 \le z \le 1$. This gains an extra factor of smallness in the error expression above if we choose our interpolation nodes equal to the zeros $\xi_k^{(p)} = \cos(\pi(k-1/2)/p)$ for $k = 1, 2, \ldots, p$ of the pth-order Chebyshev polynomial. A similar smallness property holds for the Chebyshev–Lobatto points $\eta_k = \cos(\pi k/p)$ for $k = 0, 1, \ldots, p$, which are locations of extrema of $T_p(z)$. ◁

Example 8.15. We have already seen an example of Hermite interpolation of $f(z) = \exp(z)$ on the interval $-1 \le z \le 1$ and on the unit disk in Example 8.11. Let us here, for example, interpolate the function $f(z) = 1/\Gamma(z)$ on $1 \le z \le 2$. Choosing the shifted Chebyshev–Lobatto points $3/2 + \eta_k/2 = 3/2 + \cos(\pi k/n)/2$ for $0 \le k \le n$, and taking $n = 5$, we have an interpolation error $|p(t) - 1/\Gamma(t)| \le 1 \times 10^{-5}$ approximately and an error in the derivative $|p'(t) + \Psi(t)/\Gamma(t)|$ about $O(10^{-4})$. See Fig. 8.9. This figure (and another which is not shown here) was generated with the `genbarywts` and `hermiteval` programs (available in the code repository) and the commands

```
tau     = 1.5+cos( pi*(0:5)/5 )/2;
rho     = 1.0./gamma(tau);
```

```
t       = linspace( 1, 2, 2001);
yref    = 1.0./gamma(t);
ypref   = -yref .* psi(t);
[w,D]   = genbarywts( tau, 1 );
[y,yp] = hermiteval( rho, t, tau, 1, w, D );
figure(1)
plot( tau, rho, 'ko', t, y, 'k', t, yp, 'k--' ) set(gca,'fontsize
    ',16)
figure(2)
plot( tau, rho-1.0./gamma(tau), 'ko', ...
      t, y-yref', 'k', ...
      t, yp-ypref', 'k--' )
set(gca,'fontsize',16)
xlabel('x')
ylabel('interpolation␣error')
```

◁

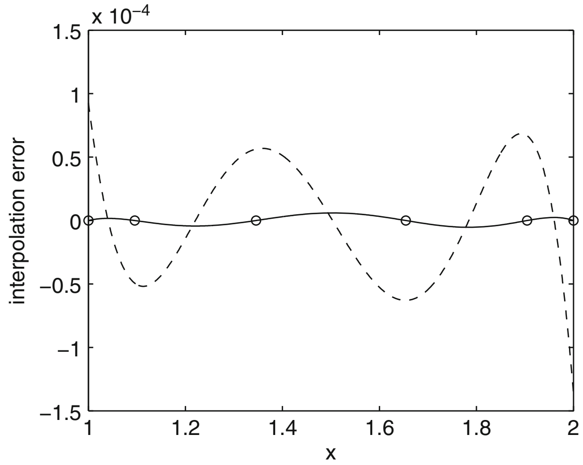


Fig. 8.9 The *solid line* (going through the circles) is the interpolation error $p(t) - {}^{1}/_{\Gamma(t)}$ for a Lagrange interpolant on the nodes ${}^{3}/_{2} + \cos({}^{\pi k}/_{5})$ for $0 \le k \le 5$. The *dashed line* is the error in the derivative of the interpolant $p'(t) + {}^{\psi(t)}/_{\Gamma(t)}$. The $\psi(z)$ function is the derivative of the logarithm of the Γ function, known to MATLAB as `psi(t)`

8.6 Interpolating in Other Polynomial Bases

It is occasionally useful to interpolate directly in other polynomial bases. That is, we may be given Lagrange or Hermite data and asked to explicitly construct the coefficients c_k in the polynomial expansion in another basis, say $\phi_k(z)$. This is a change-of-basis problem and, as will be explored below, may not be a good idea numerically. However, sometimes it is indeed what is wanted. The problem can be phrased using a *generalized Vandermonde matrix*, as follows.

Suppose we start with Hermite data: nodes τ_i with confluencies $s_i \ge 1$. The Hermite interpolational basis is then $H_{i,j}(z)$ for $0 \le i \le n$, $0 \le j \le s_i - 1$, and the degree $d = -1 + \sum_{i=0}^{n} s_i$ as usual. Computing the generalized barycentric weights $\beta_{i,j}$ gives us the explicit expression (8.23) for $H_{i,j}(z)$. Expressing each desired basis polynomial $\phi_k(z)$ in terms of $H_{i,j}(z)$ is simple:

$$\phi_k(z) = \sum_{i=0}^{n} \sum_{j=0}^{s_i-1} \frac{\phi_k^{(j)}(\tau_i)}{j!} H_{i,j}(z)\,. \tag{8.42}$$

This gives us our explicit entries in the change-of-basis matrix (called a generalized Vandermonde matrix, which will be nonsingular if the nodes are distinct) from the Hermite basis to the new basis $\phi_k(z)$. Finding the c_k is thereby reduced to the solution of a linear system of equations.

Example 8.16. As a definite example, suppose that we are given Hermite data on the nodes $[-1, 1/2, 1]$ with confluency 2 at each node. Suppose also that we wish to express our polynomial $p(z)$ in the Chebyshev basis. Since we have six pieces of data, we can interpolate it with a polynomial of degree at most 5. We can write each Chebyshev polynomial $T_k(z)$ in terms of this Hermite data as

$$\begin{aligned} T_k(z) = T_k(-1)H_{1,0}(z) + T_k'(-1)H_{1,1}(z) + T_k(1/2)H_{2,0}(z) \\ + T_k'(1/2)H_{2,1}(z) + T_k(1)H_{3,0}(z) + T_k'(1)H_{3,1}(z)\,, \end{aligned}$$

numbering the nodes from 1 to 3. Writing the row vector of Hermite interpolation basis elements as $\mathbf{H}^H = [H_{1,0}(z), H_{1,1}(z), \ldots, H_{3,1}(z)]$, and similarly the row vector of Chebyshev basis elements as $\mathbf{T}^H = [T_0(z), T_1(z), \ldots, T_5(z)]$, we have the change-of-basis relationship

$$\mathbf{T}^H = \mathbf{H}^H \mathbf{V}\,.$$

The change-of-basis matrix is then (evaluating the first six Chebyshev polynomials and their derivatives at each node)

$$\mathbf{V} = \begin{bmatrix} 1 & -1 & 1 & -1 & 1 & -1 \\ 0 & 1 & -4 & 9 & -16 & 25 \\ 1 & 1/2 & -1/2 & -1 & -1/2 & 1/2 \\ 0 & 1 & 2 & 0 & -4 & -5 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 4 & 9 & 16 & 25 \end{bmatrix}.$$

This was generated in MAPLE with the command

```
V := Matrix(6, 6, shape = Vandermonde[[-1, -1, 1/2, 1/2, 1, 1],
    confluent = true, basis = ChebyshevT]);
```

The relationship $p(z) = \mathbf{H}^H\boldsymbol{\rho}$ expresses the polynomial $p(z)$ in the Hermite interpolational basis with the vector of coefficients $\boldsymbol{\rho}$. We wish to find a vector of coefficients, call it $\mathbf{a}$, for which this same $p(z) = \mathbf{T}^H\mathbf{a}$. Using the relationship $\mathbf{T}^H = \mathbf{H}^H\mathbf{V}$, we see that $\mathbf{V}\mathbf{a} = \boldsymbol{\rho}$ and to change bases, then we must solve a linear system of equations with this (nonsingular) generalized Vandermonde matrix. For the specific case $\rho_{1,0} = \rho_{1,1} = 1$, $\rho_{2,0} = \rho_{3,0} = 1$ and $\rho_{2,1} = \rho_{3,0} = 0$ specifying all the values and slopes, we get

$$p(z) = \frac{25}{24} T_0(z) - \frac{7}{144} T_1(z) - \frac{1}{72} T_2(z) + \frac{1}{24} T_3(z) - \frac{1}{36} T_4(z) + \frac{1}{144} T_5(z)\,,$$

namely, a polynomial expressed in Chebyshev basis. ◁

The general procedure is similar. Many bases are encoded in the MAPLE facility for Vandermonde matrices.

Now, are change of bases numerically advisable? As we have seen, if

$$\phi_i(z) = \sum_{j=0}^{n} \phi_{i,j} \psi_j(z)\,,$$

for $i = 0, 1, \ldots, n$, then the change-of-basis matrix (call it $\mathbf{A}$) has entries $\phi_{i,j}$. If $p(z) = \sum_{i=0}^{n} a_i \phi_i(z)$ and $p(z) = \sum_{j=0}^{n} b_j \psi_j(z)$, then

$$\begin{aligned} p(z) &= \sum_{i=0}^{n} a_i \phi_i(z) = \sum_{i=0}^{n} a_i \sum_{j=0}^{n} \phi_{i,j} \psi_j(z) = \sum_{j=0}^{n} \left(\sum_{i=0}^{n} a_i \phi_{i,j} \right) \psi_j(z) \\ &= \sum_{j=0}^{n} b_j \psi_j(z)\,. \end{aligned} \tag{8.43}$$

As a result, the coefficients b_j are related to a_i by

$$b_j = \sum_{i=0}^{n} a_i \phi_{i,j} \tag{8.44}$$

or, in matrix terms, the row vector $\mathbf{b}^T = \mathbf{a}^T \mathbf{A}$. Now let us examine what happens to errors Δa_i. Quite clearly, these induce a vector of errors Δb_j in the coefficients in the new basis, and

$$\Delta b_j = \sum_{i=0}^{n} \Delta a_i \phi_{i,j}\,, \tag{8.45}$$

or $\Delta \mathbf{b}^T = \Delta \mathbf{a}^T \mathbf{A}$. Taking norms, we have

$$\|\Delta \mathbf{b}\| = \|\Delta \mathbf{a}^T \mathbf{A}\| \le \|\mathbf{A}\| \|\Delta \mathbf{a}\|\,. \tag{8.46}$$

This ought to look familiar: Now, from $\mathbf{a}^T = \mathbf{b}^T \mathbf{A}^{-1}$, we get $\|\mathbf{b}^T\| \ge \|\mathbf{a}^T\| / \|\mathbf{A}^{-1}\|$, and so

$$\frac{\|\Delta \mathbf{b}\|}{\|\mathbf{b}\|} \le \|\mathbf{A}\| \|\mathbf{A}^{-1}\| \frac{\|\Delta \mathbf{a}\|}{\|\mathbf{a}\|}, \tag{8.47}$$

and we see that, in the worst case, the relative errors in the new coefficients might be as bad as the matrix condition number of $\mathbf{A}$ times the relative error in the coefficients in the old basis.

Remark 8.13. Remember now that the condition number for evaluation of the polynomial multiplies a bound on the relative error in the coefficients: $|\Delta f(z)| \le B_{\text{basis}}(z)\|\delta_k\|_\infty$. When we change bases, we change two things: We change the $B_{\text{basis}}(z)$, and this may become smaller if the new basis $\phi_i(z)$ is better than the old one (or get worse if we're doing something silly, such as changing from Lagrange basis into the monomial basis). However, the bound on the $\|\delta_k\|$ will be multiplied by $\kappa(\mathbf{A})$, which is always greater than or equal to 1—that is, on the data error side, changing bases *always* makes things worse (in the worst case). Changing back would make it worse again! More importantly, the condition number of the change-of-basis matrix can often be spectacularly bad. See Exercise 8.14.

In particular, the Vandermonde matrices (for changing basis from Lagrange to monomial basis) are known to have condition numbers that grow *at least* exponentially with the degree. For real nodes, a known bound is $\kappa_2(\mathbf{V}_n) \ge (1+\sqrt{2})^{n-1}\sqrt{2/(n+1)}$. The complex case is often better, and when the nodes are roots of unity, the condition number is just 1. ◁

Remark 8.14. The previous remark, while correct, is perhaps overly pessimistic. Algorithms that make use of the *structure* of generalized Vandermonde matrices can help considerably and in particular can avoid introducing spuriously harmful numerical perturbations Δb_j. The conclusion still holds when there is genuine data error present, but in a certain sense you have to be a bit unlucky to have the "worst case" actually happen. Sometimes you *can* convert to the monomial basis, if the degree isn't too high and the data error isn't too bad and if you use a good algorithm, such as those discussed in Higham (2002). ◁

8.7 Rational Interpolation with Known Denominator

Using the approach based on contour integrals and partial fraction decomposition, we obtain a rational interpolation of our data in a completely analogous way. Instead of the integral

$$0 = \frac{1}{2\pi i}\oint_C \frac{1}{w(z)(t-z)}\cdot\frac{p(z)}{1}dz$$

to find the polynomial interpolant $p(z)$, we consider the integral

$$0 = \frac{1}{2\pi i}\oint_C \frac{q(z)}{w(z)(t-z)}\cdot\frac{p(z)}{q(z)}dz \tag{8.48}$$

to find the rational interpolant $p(z)/q(z)$. By using the partial fraction decomposition [remember, $q(z)$ is assumed to be known, and we assume $\deg q < \deg w$], we have

$$\frac{q(t)}{w(t)} = \sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\frac{\alpha_{ij}}{(t-\tau_i)^{j+1}}\,, \tag{8.49}$$

where degree $q(t) \le d$, and where the numbers $\alpha_{i,j}$ may be used in exactly the same fashion as we used the generalized barycentric weights previously. Thus, we get *rational* Hermite (or Lagrange) interpolation:

$$\frac{p(t)}{q(t)} = \frac{\displaystyle\sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\sum_{k=0}^{j}\alpha_{ij}\rho_{ik}(t-\tau_i)^{k-j-1}}{\displaystyle\sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\alpha_{ij}(t-\tau_i)^{-j-1}}\,, \tag{8.50}$$

which satisfies

$$\frac{1}{k!}\left(\frac{p(t)}{q(t)}\right)^{(k)}\Bigg|_{t=\tau_i} = \rho_{ik}\,, \tag{8.51}$$

provided that $\alpha_{i,s_i-1} \ne 0$.

Remark 8.15. If some $\alpha_{i,s_i-1} = 0$, then the rational interpolant with this denominator has what is called an *unattainable point*. A little thought should convince you that in this case $q(\tau_i) = 0$, and we cannot have reasonable behavior of $p(z)/q(z)$ at $z = \tau_i$ unless also $p(\tau_i) = 0$. ◁

To get the $\alpha_{i,j}$ from Hermite data for $q(z)$, namely,

$$\sigma_{i,j} := \frac{q^{(j)}(\tau_i)}{j!}\,,$$

we may start with the (first or second) Hermite barycentric form and rearrange, as follows:

$$\begin{aligned}\frac{q(z)}{w(z)} &= \sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\sum_{k=0}^{j}\beta_{i,j}\sigma_{i,k}(z-\tau_i)^{k-j-1} = \sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\sum_{\ell=0}^{j}\beta_{i,j}\sigma_{i,j-\ell}(z-\tau_i)^{-\ell-1}\\ &= \sum_{i=0}^{n}\sum_{\ell=0}^{s_i-1}\left(\sum_{j=\ell}^{s_i-1}\beta_{i,j}\sigma_{i,j-\ell}\right)(z-\tau_i)^{-\ell-1}\,,\end{aligned}$$

whence

$$\alpha_{i,j} = \sum_{\ell=j}^{s_i-1}\beta_{i,\ell}\sigma_{i,\ell-j}\,. \tag{8.52}$$

This fact is, from time to time, quite convenient.

Example 8.17. Suppose that we wish to fit a rational function of the form $p(z)/(1+z^2)$ to the data $[1,-1,1,-1]$ at the nodes $[0, 1/4, 3/4, 1]$. The first thing we do is find the $\alpha_{i,j}$; since this is Lagrange interpolation, we may drop the second index $j=0$ and simply write α_i. This requires the following partial fraction expansion:

$$\frac{1+z^2}{z\,(z-{}^{1}\!/\!_{4})\,(z-{}^{3}\!/\!_{4})\,(z-1)} = \frac{\alpha_1}{z} + \frac{\alpha_2}{z-{}^{1}\!/\!_{4}} + \frac{\alpha_3}{z-{}^{3}\!/\!_{4}} + \frac{\alpha_4}{z-1}\,.$$

A short computation gives the vector of α_is as

$$\boldsymbol{\alpha} = \left[-\frac{16}{3}, \frac{34}{3}, -\frac{50}{3}, \frac{32}{3}\right].$$

The numerator of the second barycentric form is just

$$\frac{\rho_1\alpha_1}{z} + \frac{\rho_2\alpha_2}{z-{}^{1}\!/\!_{4}} + \frac{\rho_3\alpha_3}{z-{}^{3}\!/\!_{4}} + \frac{\rho_4\alpha_4}{z-1}\,,$$

and the denominator is the same except without any ρ_i. We compare the results with a simple polynomial interpolant to the same data, in Fig. 8.10. ◁

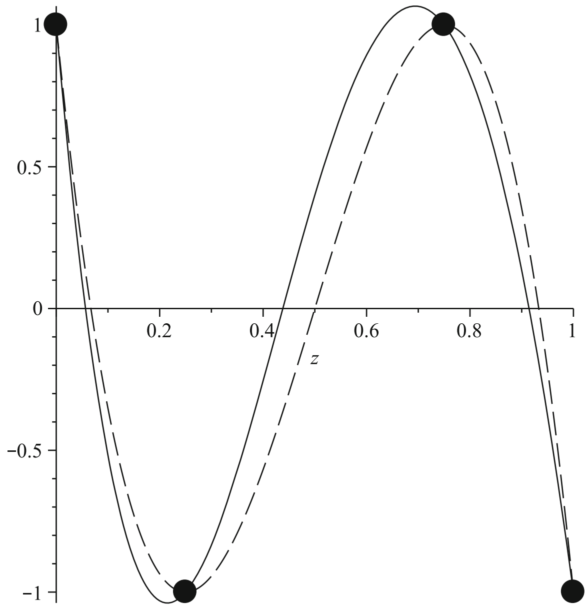


Fig. 8.10 A rational interpolant with known denominator $1+z^2$ (*solid line*) compared with a polynomial interpolant (*dashed line*) on the same data

Example 8.18. Consider interpolating the function

$$f(x) = \frac{{}^{\pi x}\!/\!_{\omega}}{\sinh({}^{\pi x}\!/\!_{\omega})} \tag{8.53}$$

on the interval $-1 \le x \le 1$. [We replace the removable singularity at $x = 0$ with $f(0) = 1$, of course.] This function has poles at $\pm ik\omega$, and if ω is small, say

$\omega = {}^1\!/_{100}$, then these poles are quite close to the interval, and interfere with the interpolation process. It seems reasonable to fit a few, say 10, of those known poles into the interpolant explicitly and take a denominator $q(x) = \prod_{k=1}^{5}(x^2+k^2\omega^2)$. Now, if we use the Chebyshev nodes, say the 21 extrema $\eta_k^{(21)}$ of $T_{20}(x)$, with this denominator, we want the α_j defined by the partial fraction decomposition of

$$\frac{q(x)}{T_{21}(x)} = \sum_{j=1}^{21} \frac{\alpha_j}{x-\xi_j^{(21)}}\,,$$

and then use these α_j in place of the barycentric weights in the second barycentric form of the Lagrange interpolant. It turns out that this is reasonably successful and produces an interpolant that reproduces the data, but it's of quite poor quality in that there are unacceptable oscillations. See Fig. 8.11.

Computation of near-optimal nodes for the interpolation is itself an interesting problem that can be attacked by eigenvalue methods.[5] By an eigenvalue technique, we computed 21 symmetric optimal points (starting from the Chebyshev points and using two iterations); when we use *those* nodes instead of Chebyshev nodes, but the same $q(x)$, computing

$$\frac{q(x)}{\prod_{j=1}^{21}(x-\tau_j)} = \sum_{j=1}^{21} \frac{\alpha_j}{x-\tau_j}\,,$$

where now the α_j are different from the ones computed previously, we see the results in Fig. 8.12. The narrow spike is quite well represented by the rational function now (and, of course, it would be hopeless to try to reproduce that spike with just a polynomial). ◁

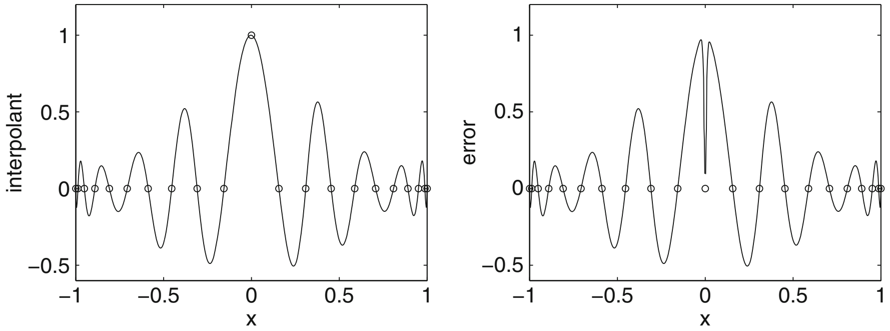


Fig. 8.11 Rational interpolation of Eq. (8.53) at Chebyshev–Lobatto points with denominator $q(x) = \prod_{k=1}^{5}(x^2+k^2\omega^2)$. The error, as seen on the right, is quite large

[5] Or by more efficient ones such as are described in van Deun et al. (2008).

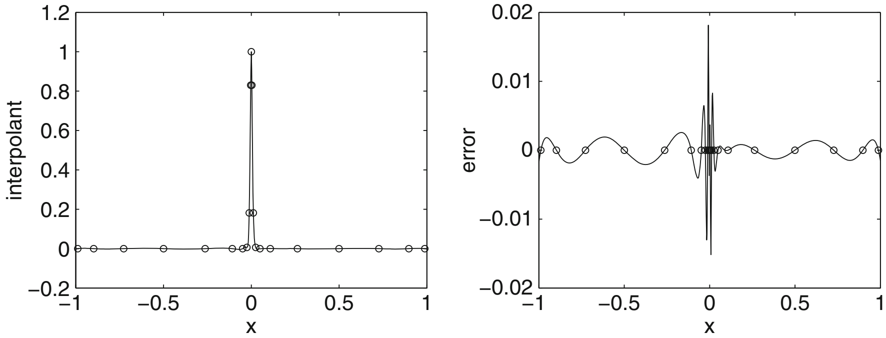


Fig. 8.12 Rational interpolation of Eq. (8.53) at near-optimal nodes, again with denominator $q(x) = \prod_{k=1}^{5}(x^2 + k^2\omega^2)$. This time, the interpolation error, while not equioscillating, is substantially better than in Fig. 8.11

8.8 Numerical Accuracy of the Second Barycentric Form

Having examined how to extend Hermite (and Lagrange) interpolation to the rational case, we can now examine the numerical accuracy of the second barycentric form. It has been observed by several authors that the second barycentric form of polynomial interpolation exhibits great *forward* accuracy. That is, errors in the computation of the generalized barycentric weights do not harm the interpolation property much, if at all. We have now presented enough material to enable the reader to understand this (in some sense, excellent) property. The reason this works is that an erroneous set of $\widehat{\beta}_{i,j}$ can be *interpreted* as the exact barycentric weights $\alpha_{i,j}$ of some rational denominator; and in that case, the second barycentric form is correct for *rational* interpolation by $^{p(z)}/_{q(z)}$. The second barycentric form will certainly fit the data well! So, in some sense, the second barycentric form is also highly backward stable, in that it gives the exact answer to a different question: Find a *rational* function that fits the data.

It will be a good approximation to a *polynomial*, however, only if the reverse-engineered $q(z)$ from the erroneous $\widehat{\beta}_{i,j}$ is, in fact, fairly close to 1 throughout the region of interest. If $q(z)$ has poles in the region, then the overall approximation is likely to be unsatisfactory, even though the rational function $^{p(z)}/_{q(z)}$ fits the data very well. Using the companion matrix for the Hermite interpolational form but with erroneous $\widehat{\beta}_{i,j}$ will give zeros of the numerator $p(z)$, and this may indeed approximate the zeros of the original function rather well; but if $q(z)$ is quite different from 1, then it may not. It is a good idea to check (perhaps by plotting) to see if 1 is well represented by Eq. (8.28)—for some arrangements of nodes, it won't be, and in that case while you will still have forward stability in the sense described above, the locations of zeros or other features of interest may be considerably different from what you expect.

Remark 8.16. Suppose $|q(z)-1| \le \kappa\mu_M$. Then

$$\left|p(z)-\frac{p(z)}{q(z)}\right| \le \left|\frac{p(z)}{q(z)}\right| |q(z)-1| \le \left|\frac{p(z)}{q(z)}\right| \kappa\mu_M\,. \tag{8.54}$$

Thus the nearness of $q(z)$ to 1 gives a measure of how good polynomial approximation by the numerator is. ◁

Example 8.19. Take $n=15$ and each $s_k=2$. Consider equally spaced points $\tau_k = -1+{}^{2k}/_{n}$ on the interval $[-1,1]$. Then computation of the barycentric weights in 15-digit arithmetic in MAPLE introduces some errors, essentially because the barycentric weights vary so widely in size, from about 104 to $1.48\cdot 10^{10}$.

What are the consequences of those errors? If we reverse-engineer the rational function $q(z)$ using (8.28), which is supposed to be 1 on this interval, and plot $|q(t)-1|$, we get Fig. 8.13. ◁

8.9 Piecewise Interpolation

We now consider what to do when we have equally spaced nodes to interpolate from. As we have seen, equally spaced nodes give rise to widely varying generalized barycentric weights for a global polynomial, and very poor conditioning of the resulting global polynomial. Yet equally spaced nodes are very natural, and occur frequently in applications.

One solution to this problem is to *give up* on a global polynomial or rational function and instead look at piecewise polynomials. This is motivated by our error formula for cubic Hermite interpolation on an interval of small width h, where we saw in Example 8.14 that the error was $O(h^4)$.

To be concrete, suppose that we have a fixed mesh (not *necessarily* equally spaced, but that certainly is a case we want to cover). Define a piecewise function

$$p(t) = \begin{cases} p_k(t) & \tau_{k-1} < t < \tau_k \\ \Omega & \text{otherwise} \end{cases} \tag{8.55}$$

for $1 \le k \le n$, where $p_k(t)$ is a polynomial, possibly different on each mesh subinterval $\tau_{k-1} < t < \tau_k$. We will want to specify what happens at the nodes τ_k (also called "knots") as well. Usually we will insist on continuity, in which case the "less than" signs above can be replaced with "less than or equal to" signs. Examples that may be familiar include piecewise linear interpolation (what most people mean when they say just "interpolation") or, a choice that is more smooth, what is known as "cubic splines," where each polynomial is cubic and we ask for not just continuity and continuous first derivatives at the nodes but also continuous second derivatives.

Remark 8.17. We saw earlier that the error $|f(z)-fl(p(z))|$ has two parts: theoretical interpolation error $|f(z)-p(z)|$ and computational error $|p(z)-fl(p(z))| \le B(z)\cdot\kappa\cdot\mu_M$. The first is large if the degree is large and equally-spaced nodes are

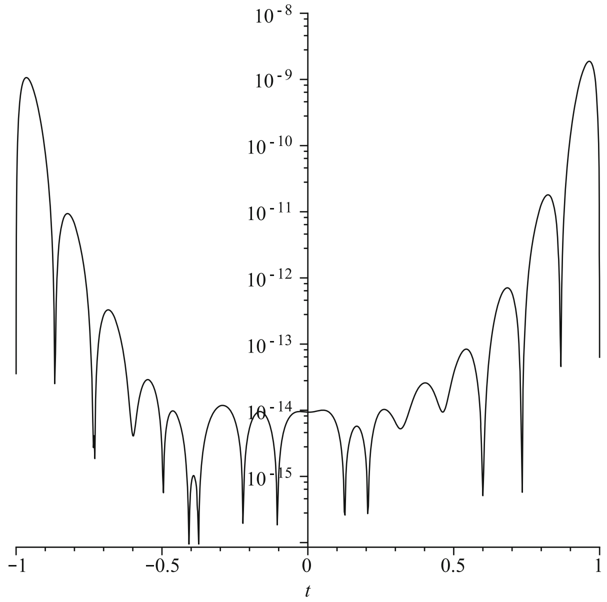


Fig. 8.13 The difference between the reverse-engineered $q(z)$ and 1, on 16 equally spaced data points with all $s_k = 2$. This shows that the second barycentric form does not *quite* give a polynomial interpolant: The denominator differs from 1, even for this small n, by about 10^{-9} out near the ends. The problem gets worse exponentially quickly as n grows. (The difficulty is that equally spaced points are bad—the Hermite form is quite good for the Chebyshev points on this interval, for example)

used, as is the second; both are small if Chebyshev-Lobatto points are used. If d is small, then the first is small if the interval is short and $B(z)$ is small, almost independently of the polynomial basis used. ◁

Remark 8.18. The value of piecewise polynomials isn't really for use with equally spaced points. The real benefit is *adaptivity*. One can adjust the mesh to suit the problem. This is in part because convergence isn't just pointwise—that is, not only do the function values converge, but also the first (and second, if a high enough degree is used) derivatives converge as the mesh spacing goes to zero. This means that as the mesh spacing goes to zero, the correct monotonicity and convexity of the underlying function will show up in the interpolant. Of course, this is also true for global polynomials on a sequence of good meshes (such as the Chebyshev–Lobatto family), with certain restrictions, but the convergence of piecewise polynomials on a well-adapted mesh family provides useful tools. ◁

8.9.1 A Cubic Spline

What follows in this section is a new derivation of the piecewise cubic Hermite spline, using the barycentric forms for a cubic Hermite interpolant on each piece: That is, instead of trying to fit unknown cubic polynomials in a local monomial basis

$a_k + b_k(t-\tau_k) + c_k(t-\tau_k)^2 + d_k(t-\tau_k)^3$ to the data and trying to find reasonable ways to determine the $4n$ unknowns (if there are n subintervals), we instead work directly with

$$p_k(t) = \frac{\sum_{i=k}^{k+1}\sum_{j=0}^{1}\sum_{\ell=0}^{j} \beta_{i,j}\rho_{i\ell}(t-\tau_i)^{\ell-j-1}}{\sum_{i=k}^{k+1}\sum_{j=0}^{1} \beta_{i,j}(t-\tau_i)^{-j-1}}\,, \tag{8.56}$$

which is the second barycentric form of the cubic Hermite interpolant (note that only two nodes are used in this form!). The $\beta_{i,j}$ can be found as usual from the partial fraction decomposition (repeated here for convenience):

$$\begin{aligned}\frac{1}{(t-\tau_{k+1})^2(t-\tau_k)^2} &= \sum_{i=k}^{k+1}\sum_{j=0}^{1}\beta_{i,j}(t-\tau_i)^{-j-1}\\ &= \frac{\frac{-2}{(\tau_{k+1}-\tau_k)^3}}{t-\tau_k} + \frac{\frac{1}{(\tau_{k+1}-\tau_k)^2}}{(t-\tau_k)^2} + \frac{\frac{2}{(\tau_{k+1}-\tau_k)^3}}{t-\tau_{k+1}} + \frac{\frac{1}{(\tau_{k+1}-\tau_k)^2}}{(t-\tau_{k+1})^2}\,.\end{aligned}$$

There are only four $\beta_{i,j}$ for each interval, and we see them written above explicitly in terms of the given nodes τ_k.

Notice that the $\rho_{i,0}$ (not the $\rho_{i,1}$, which represent derivative values) are the known data values. We want to choose the $n+1$ slopes $\rho_{i,1}$ to make the resulting interpolant as smooth as possible. We will see that we can make it $\mathscr{C}^2[\tau_1,\tau_n]$. Notice also that we may choose the $\rho_{i,1}$ in such a way that we automatically have $p(t) \in \mathscr{C}^1[\tau_1,\tau_n]$: Just take the slope at the right end of one interval to be the same slope at the left end of the next. This is very natural because $\rho_{i,1}$ is then interpreted as "the" slope at the node τ_i (indeed, it would be somewhat unnatural to have different slopes on the left and right, though we could do that if we wanted). Having made our interpolant continuously differentiable by this device, we then obtain $p'_{k-1}(\tau_k^-) = \rho_{k,1} = p'_k(\tau_k^+)$. To further ensure $p(t) \in \mathscr{C}^2[\tau_1,\tau_n]$, we want to make the second derivatives equal; that is,

$$p''_{k-1}(\tau_k^-) = p''_k(\tau_k^+) \qquad k = 2,3,\ldots,n-1\,. \tag{8.57}$$

Contrast this with the necessary algebra in the local monomial case. We would have

$$p_k(t) = \rho_{k,0} + \rho_{k,1}(t-\tau_k) + c_k(t-\tau_k)^2 + d_k(t-\tau_k)^3; \tag{8.58}$$

even to make the function just $\mathscr{C}^1$, we would have to impose the condition

$$p'_k(\tau_{k+1}^-) = p'_{k+1}(\tau_{k+1}^+)\,,$$

which isn't automatic; we would have to *enforce*

$$\rho_{k,1} + 2c_k(\tau_{k+1} - \tau_k) + 3d_k(\tau_{k+1} - \tau_k)^2 = \rho_{k+1,1} \,.$$

For $\mathscr{C}^2$, we would have to enforce yet another condition, namely,

$$2c_k + 6d_k(\tau_{k+1} - \tau_k) = 2c_{k+1} \,.$$

Of course, this can be done, and the solution is even elegant. Through an algebraic trick, these equations are reduced to a *tridiagonal* system of equations for the slopes $\rho_{k,1}$, and explicit formulæ for the c_k and d_k are known once the slopes are known. This is shown in `splinetx.m` in Moler (2004).

But here, because we start with the Hermite interpolational basis, we have a simpler (but equivalent) task: Just enforce the second derivative conditions. To do this, we need a formula for the second derivative of the cubic Hermite interpolant at the nodes. A short computation in MAPLE shows that

$$p''_{k-1}(\tau_k^-) = \frac{2}{\tau_k - \tau_{k-1}}(2\rho_{k,1} + \rho_{k-1,1}) - \frac{6}{(\tau_k - \tau_{k-1})^2}(\rho_{k,0} - \rho_{k-1,0}) \tag{8.59}$$

and

$$p''_k(\tau_k^+) = \frac{-2}{\tau_{k+1} - \tau_k}(2\rho_{k,1} + \rho_{k+1,1}) + \frac{6}{(\tau_{k+1} - \tau_k)^2}(\rho_{k+1,0} - \rho_{k,0}) \,. \tag{8.60}$$

Equating these at interior nodes $1 \le k \le n-1$ gives $n-1$ equations constraining the slopes. Explicitly,

$$\begin{aligned}\frac{2}{\tau_k - \tau_{k-1}}\rho_{k-1,1} + 4\left(\frac{1}{\tau_k - \tau_{k-1}} + \frac{1}{\tau_{k+1} - \tau_k}\right)\rho_{k,1} + \frac{2}{\tau_{k+1} - \tau_k}\rho_{k+1,1} \\ = \frac{6}{(\tau_{k+1} - \tau_k)^2}(\rho_{k+1,0} - \rho_{k,0}) + \frac{6}{(\tau_k - \tau_{k-1})^2}(\rho_{k,0} - \rho_{k-1,0})\end{aligned}$$

for $k = 1, 2, 3, \ldots, n-1$. There are $n-1$ equations. The structure of the resulting matrix is tridiagonal.

This leaves the slopes at the end as free parameters. Here are some common choices:

- If we set the second derivatives at the ends $\rho_{0,2} = \rho_{n,2} = 0$, this gives two extra equations to define the slopes. This is called the "natural" spline, which flattens out toward the ends.
- One could ask instead for $\mathscr{C}^3$ continuity at τ_2 and τ_{n-1}. This is known as the "not-a-knot" condition.
- Are the slopes *known* at the edges? Then use them! This is called a "clamped" spline and is in some sense a piecewise Hermite interpolant.

Whatever we choose to do with them, however, we may still fit the given data.

Remark 8.19. The nomenclature for the different classes of piecewise cubic interpolants is confusing. When both function values $\rho_{k,0}$ and derivative values $\rho_{k,1}$ are specified and these determine the cubic on each interval, this is called piecewise cubic Hermite interpolation. When just the function values $\rho_{k,0}$ are given and the derivative values (apart from the end values) are *computed* in order to make the interpolant twice continuously differentiable, this is called a *spline*. Finally, when the values $\rho_{k,0}$ are given and the derivative values $\rho_{k,1}$ are chosen in a way to match the average curvature of neighboring pieces as is done in the MATLAB routine `pchip`, this is also, and confusingly, called piecewise cubic Hermite interpolation (hence the name). The relative rarity of the first use [when true derivatives $\rho_{k,1} = f'(\tau_k)$ are specified] makes this confusion bearable. ◁

Example 8.20. Suppose $n = 5$ and we have a uniform mesh with each subinterval of width h. Then the tridiagonal system is $\mathbf{S}\boldsymbol{\rho}^{(1)} = \mathbf{M}\boldsymbol{\rho}^{(0)}$, where $\boldsymbol{\rho}^{(1)}$ represents the vector of as-yet-unknown derivatives at the nodes τ_0, τ_1, …, τ_n and $\boldsymbol{\rho}^{(0)}$ represents the vector of known function values at the nodes. The matrix $\mathbf{S}$ is

$$\mathbf{S} = \begin{bmatrix} 1 & 4 & 1 & 0 & 0 & 0 \\ 0 & 1 & 4 & 1 & 0 & 0 \\ 0 & 0 & 1 & 4 & 1 & 0 \\ 0 & 0 & 0 & 1 & 4 & 1 \end{bmatrix}$$

and the matrix $\mathbf{M}$ is

$$\mathbf{M} = \frac{3}{h} \begin{bmatrix} -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix} .$$

Notice that these are rectangular matrices—there are two more columns than rows. This shows that there are two free parameters, as stated previously. Notice also that in the nonuniform mesh case, the matrix $\mathbf{M}$ will not have a zero diagonal.

For definiteness, choose here $\tau_k = -1 + {}^{2k}\!/_{n}$, for $0 \le k \le n$. Let the given data be $[0,0,0,0,0,1]$ on these nodes, and add the extra (clamped) conditions that the derivative should be zero at each end: $\rho_{0,1} = 0$ and $\rho_{n,1} = 0$. This takes care of the two free parameters. Then $h = {}^{2}\!/_{n} = {}^{2}\!/_{5}$, and the solution to $\mathbf{S}\boldsymbol{\rho}^{(1)} = \mathbf{M}\boldsymbol{\rho}^{(0)}$ is

$$\boldsymbol{\rho}^{(1)} = \left[-\tfrac{15}{418} \ , \ \tfrac{30}{209} \ , \ -\tfrac{225}{418} \ , \ \tfrac{420}{209} \right]^T .$$

This gives the values of the derivatives at the interior nodes. When we then use this data to construct the n cubic Hermite interpolant pieces, we wind up with

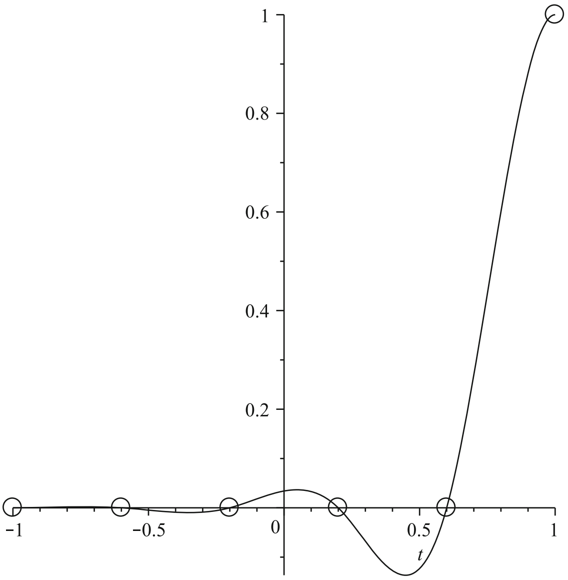


Fig. 8.14 A clamped cubic Hermite spline fit to the data indicated by circles. Derivatives are specified as zero at the ends. Notice the smoothness of the fit, but also the overshoot and oscillations. To try to preserve monotonicity and convexity, `pchip` may be better

$$p(t) = \begin{cases} -\frac{375}{1672}\left(t+1\right)^2 t - \frac{225}{1672}\left(t+1\right)^2 & t < -\frac{3}{5} \\ \frac{25}{4}\left(t+\frac{1}{5}\right)^2\left(-\frac{15}{418}t - \frac{9}{418}\right) + \frac{25}{4}\left(t+{}^{3}\!/_{5}\right)^2\left(\frac{30}{209}t + \frac{6}{209}\right) & t < -\frac{1}{5} \\ \frac{25}{4}\left(t-\frac{1}{5}\right)^2\left(\frac{30}{209}t + \frac{6}{209}\right) + \frac{25}{4}\left(t+\frac{1}{5}\right)^2\left(-\frac{225}{418}t + \frac{45}{418}\right) & t < \frac{1}{5} \\ \frac{25}{4}\left(t-\frac{3}{5}\right)^2\left(-\frac{225}{418}t + \frac{45}{418}\right) + \frac{25}{4}\left(t-\frac{1}{5}\right)^2\left(\frac{420}{209}t - \frac{252}{209}\right) & t < \frac{3}{5} \\ \frac{25}{4}\left(t-1\right)^2\left(\frac{420}{209}t - \frac{252}{209}\right) - \frac{125}{4}\left(t-\frac{3}{5}\right)^2\left(t-1\right) + \frac{25}{4}\left(t-\frac{3}{5}\right)^2 & t < 1 \end{cases},$$

which we plot in Fig. 8.14. ◁

8.9.2 *The Condition Number of a Spline Interpolant*

The algorithms for spline interpolation and for piecewise Hermite interpolation are all quite stable; no problems are reported in the literature except perhaps for quite pathological situations such as when subintervals are ridiculously short or long. That is, we expect that our algorithms will produce the exact interpolants for slightly perturbed data (indeed, data perturbed by only a modest factor of the unit roundoff).

As with global polynomial interpolation, however, it makes sense to ask about the *conditioning* of the spline or piecewise Hermite interpolant itself: If the data change by a small amount, will the (values of) the interpolant change by a lot, or just a little? Is there a "condition number function" analogous to the $B(z)$ function for polynomials global to the whole region of interest that we introduced in Chap. 2?

This is of interest not just for the perturbations that model the rounding errors in our computations, but is especially of interest given errors in the data. Is there a condition number, then?

The answer is that of course there is. The definitive presentation is in de Boor (1978). However, since we have made a novel presentation of splines here based on the local Hermite interpolational basis, it is worthwhile to comment on how the condition number appears here.

Notice that each polynomial piece in the interpolant is of the form $\rho_{i,0}H_{i,0}(z) + \rho_{i,1}H_{i,1}(z) + \cdots$ and the primary data are the $\rho_{i,0}$. The results of changes to the function values, what we call $\rho_{i,0}$ here, are felt directly through the $H_{i,0}(z)$.

They are also felt indirectly through their effect on the computed derivative values $\rho_{i,1}$, and here their influence is more global for splines, though local both for `pchip` and for true cubic Hermite interpolation using known derivative values. Since we solve a tridiagonal system $\mathbf{S}\boldsymbol{\rho}_{i,1} = \mathbf{M}\boldsymbol{\rho}_{i,0}$ to identify the computed derivative values for a spline, the sensitivities (conditioning) of the derivative values to changes in the primary data are felt through the norm of $\mathbf{S}^{-1}$. Specifically, *changes* in the computed derivatives have norm bounded by $\|\mathbf{S}^{-1}\|\|\mathbf{M}\|$ times the norm of the changes $\Delta\boldsymbol{\rho}$ in the primary data, where $\mathbf{M}$ was the tridiagonal matrix appearing on the right-hand side of the expression for the derivatives. Of course, we do not compute $\mathbf{S}^{-1}$, which is full (whereas $\mathbf{S}$ is tridiagonal), but we can think of its influence in this manner.

Since the process is linear, we may as well consider $\boldsymbol{\rho}$ to be identically zero, and $\Delta\boldsymbol{\rho}$ to be zero except in the ith component where it is ε. This leads to $\Delta\boldsymbol{\rho}_{i,1}$ being ε times the ith column of $\mathbf{S}^{-1}\mathbf{M}$. Clearly, the (absolute) condition number of this process then will be bounded by the 1-norm of the matrix in question, which is the maximum 1-norm of any column. It turns out that if the mesh spacing is uniform, as in Example 8.20, then this matrix has a very modest norm: slightly larger than 1 and growing as the dimension increases but really rather good. Rather than try to prove this in general, we do a sample computation and show that it is true for that, as an example.

Example 8.21. We continue the uniform mesh solution of Example 8.20, but let the dimension increase. We do the computation in MAPLE using exact rational arithmetic, to remove the influence of rounding errors—we are studying condition, here.

When we take n fairly large, say $n = 344$, and choose perturbations near the endpoints, we find that the norm of the matrix is indistinguishable from 1. When we take a perturbation near the middle, we find a result somewhat larger than 1. Specifically, if we plot the interpolant with $\rho = 0$ except for $\rho_{173} = 1$, we find that the solution is fairly flat (nearly zero) away from τ_{173}, but near there it looks like the plot in Fig. 8.15. The error curve oscillates, and this can lead to unwanted oscillations in the graph; to address that issue, the derivatives are smoothed out artificially. See Problem 8.40. ◁

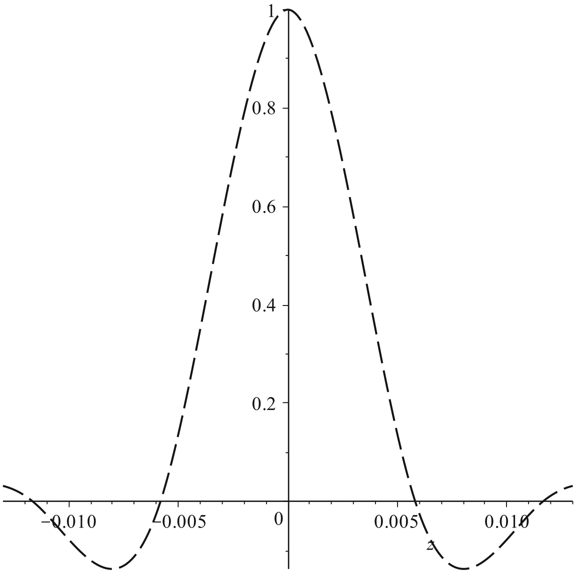


Fig. 8.15 Perturbing the data near the midpoint of a piecewise cubic Hermite interpolant with $n = 344$ leads us to believe the problem is well-conditioned. Of course, there are possibly unwanted oscillations, but in magnitude the error is not amplified

8.10 Chebfun and Interpolation

The Chebfun package is founded on barycentric interpolation at Chebyshev–Lobatto points, scaled to the interval of interest. The ideas explained in this chapter are crucial to how that package works (of course, there are many other ideas needed too). Several examples of Chebfun's use have already been given in this book, in Chaps. 2 and 3.

Here, let us look at the hard-to-interpolate function $({}^{\pi x}/_{\omega})/\sinh({}^{\pi x}/_{\omega})$ for which we earlier used rational interpolation at several of the known poles. The first thing we try is just vanilla Chebfun:

```
f = chebfun( '(pi*x/0.01)/sinh(pi*x/0.01)', [-1,1] );
%f =
%   chebfun column (1 smooth piece)
%       interval          length    endpoint values
%(        -1,         1)      3441   2.3e-134  2.3e-134
%vertical scale =   1
figure(1),plot( f, 'k' )
t = linspace(-1,1,2012);
figure(2),semilogy( t, abs(sinh(pi*t/0.01).*f(t)*0.01./t/pi - 1 )
    , 'k' ),xlabel('x'),ylabel('relative␣error')
figure(3),semilogy( t, abs( f(t) - pi/0.01*t./sinh(pi*t/0.01) ),
    'k' );
xlabel('x','fontsize',16);
```

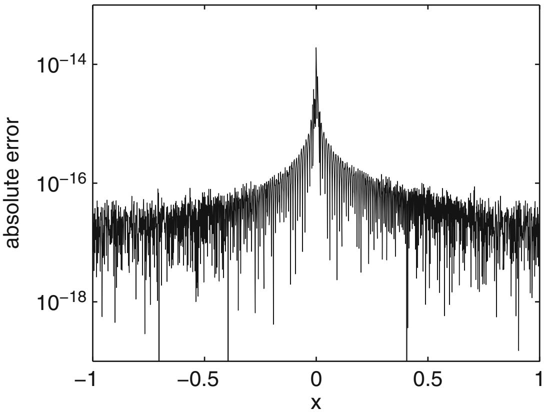


Fig. 8.16 Absolute error in interpolation of $({}^{\pi x}\!/\!_{\omega})/\sinh({}^{\pi x}\!/\!_{\omega})$ at $3,441$ Chebyshev–Lobatto points, constructed by Chebfun

```
ylabel('absolute␣error','fontsize',16);
set(gca,'fontsize',16);
axis([-1,1,10E-20,10E-14]);
```

It succeeds, albeit with a much higher degree (3,440) than our "optimal" rational interpolant that we discussed earlier, which had degree $(20,10)$—though note that it has much better accuracy, too.

The interpolation is remarkably successful, with the absolute error as plotted in Fig. 8.16. The error is worst near the spike, but is still only two orders of magnitude larger than the machine epsilon. Our rational interpolant, on the other hand, had error about 10^{-4} (but it used an approximant of about 1% the complexity). The *relative* error of the `chebfun` is horrid, of course (going up to about 10^{120} by the ends of the interval). The roots, however, as computed above (but not shown here), are spurious. This is only to be expected: the true function is very small near the ends of the intervals (about 10^{-134}) and *any* rootfinder would be hard-pressed to say that the function wasn't really zero anywhere on the real line.

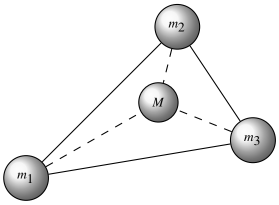
As mentioned in Chap. 3, Chebfun changes bases—from the Lagrange basis on the Chebyshev–Lobatto points to the Chebyshev basis itself, via the FFT—in order to construct a companion matrix to pass to an eigenvalue solver to find the roots. This procedure has been very well tested and is remarkably robust, accurate, and even efficient. But this book has earlier warned about ill-conditioning of the change-of-basis matrix. What is the condition number of this particular change-of-basis matrix? Its entries are $T_i(\eta_j)$ for $0 \le i,j \le n$. When we construct the matrix explicitly for $n = 20$, we find that the largest singular value is 5 and the smallest is $\sqrt{10}$, giving a 2-norm condition number of about 1.58. This explains the success of the approach, then: Changing from the Lagrange basis on these points to the *monomial* basis is (by the results mentioned earlier) exponentially ill-conditioned—but changing to the Chebyshev basis turns out to be very well-conditioned. In fact, this is exactly analogous to the well-conditioning of the change-of-basis from the

Lagrange basis on the roots of unity to the monomial basis: Because that change-of-basis matrix is, in fact, unitary, the change of basis is perfectly conditioned (in a normwise sense). Here we don't quite have a unitary change-of-basis matrix, but almost. We will return to this in Chap. 9, but see Exercise 8.15.

8.11 Notes and References

The influential papers Berrut and Trefethen (2004) and Higham (2004) demonstrate the stability (and efficiency) of the barycentric form for Lagrange interpolation. This was apparently "known" previously—in the sense that some people knew it, and had even proved and published it—but for some reason the knowledge did not spread into the general community, and pessimism about Lagrange interpolation was the general rule. In particular, the facts about the barycentric form were known to Henrici [see, e.g., Henrici (1982) for a nice discussion] and a particularly useful case was derived in Salzer (1972); in Trefethen (2013) we find references to barycentric interpolation papers in the 1940s and going further back to Jacobi's discussion of partial fractions in 1825!

The name "barycentric" is not arbitrary. As explained by Henrici (1979a 229), "Rutishauser would have called the formulæ [...] barycentric because they are formally identical with the formulæ for the center of gravity (barycenter) of a system of masses [...] attached to the points f_k [$= f(\tau_k)$]."



The very clever "exact$(\Delta t = 0) = k$" trick used in Algorithm 8.1 and in the program genbarywts to detect when any member of the input vector of t values is exactly equal to some node τ_k is taken from Berrut and Trefethen (2004); there was a typo in the printed form in that paper, and a printed 1 should have been an index i. We hope there are no typos in the algorithm as printed here, but in any case the MATLAB code in genbarywts.m has it right (as, of course, does the code used by Berrut and Trefethen (2004)). The fact that this code uses a test for exact floating-point equality is—to our minds—remarkable. If the input t differs by one bit (after rounding) from a node τ_k, the second barycentric form is accurate even with all the massive cancellation in numerator and denominator!

The use of companion matrix pencils to solve polynomial equations is very old. The Frobenius companion form used in roots in MATLAB is, of course, the oldest. The comrade/colleague/companion matrices for orthogonal polynomials are due to Good (1961) and Specht (1960) independently, but have been rediscovered many times (the use of differing though cognate words, "colleague" and "comrade," instead of something simple like "generalized companion" probably hindered notice). A recent survey of efficiency in the use of companion matrix methods is Pan and Zheng (2011). The Frobenius companion matrix pencil for matrix polynomials is used to good effect for CAGD problems in Manocha and Demmel (1994). The companion matrix pencil for the Lagrange basis was invented in 2001 and first presented at a Mapstone Lecture workshop at SUNY Geneseo but not published until much later, at the EACA meeting in Santander in 2004; the Hermite interpolational case was worked out later still and given in Shakoori (2008). The case of the Newton basis is discussed in Calvetti et al. (2001). An accessible presentation including the matrix polynomial case is in Amiraslani et al. (2009). For a recent review of these issues, see Boyd (2013).

The numerical stability of the approach was appreciated from the beginning because it worked experimentally; a proof was worked out and sketched in Lawrence and Corless (2011). A more detailed study is in process at this time of writing. Most recently, Piers Lawrence noticed that the companion matrix pencil worked better if the arrow pointed up and to the left instead of down and to the right (the formulations are mathematically equivalent), because the normal QZ iteration for solving the generalized eigenvalue problem detects the two spurious infinite eigenvalues immediately and deflates them without error in this case.

The use of contour integrals to develop interpolation formulas follows the ideas of Butcher (1967), who himself followed in the grand tradition of Cauchy and Hermite.[6] The arguments to show that the contour integrals are zero if the degree of the denominator is 2 more than the degree of the numerator are standard; see, for example, Levinson and Redheffer (1970), but there is a brief discussion in Appendix B.

There are several methods in the literature for computing the generalized barycentric weights used in these interpolation formulæ. For instance, there is the divided-difference method in Schneider and Werner (1991), which has the advantage of being efficient (more efficient, by about a factor of two, than the method we recommend here). However, that method is sensitive to the ordering of the nodes. Since it uses divided differences, it implicitly uses a Newton basis. As you will see in Problems 8.13–4, the condition number of Newton bases depends on the ordering chosen for the nodes. We have found that the simple method recommended by Henrici (1982), namely the use of local Taylor series, can be much more accurate. This method, which we described in some detail in Sect. 2.5 of Chap. 2, has accuracy that is (nearly) independent of the node ordering.

The optimality of the Bernstein–Bézier basis is shown in the elegant paper Farouki and Goodman (1996), and extended to the multivariate case by Lyche and Peña (2004). By weakening the nonnegativity constraint but using the same proof technique, the Lagrange basis was shown to be (sometimes) better in Corless and Watt (2004).

To prove Eq. (8.36), we adapted the proof of Hamming (1973 p. 236). In de Boor (2005), that proof is attributed to Schwarz, in the early 1880s. The proof was ex-

[6] Also, as Gerhard Wanner tells us, Runge used contour integrals too in his celebrated 1901 paper for just this purpose.

tended and simplified by Kansy (1973) (this paper is written in German). Kansy's result is used in Shampine (1985).

Concerning the Hermite integral formula for the complex error in interpolation, it is remarked in de Boor (2005) that Frobenius worked on this before Hermite did. According to L. N. Trefethen (private communication), the formula appears first to have been derived by Cauchy, in 1826, using partial fractions, probably based on the thesis on partial fractions published in 1825 by Jacobi. Hermite then went on to replace the divided-difference formula with an integral average of a derivative of f. Starting with the observation that, even for complex nodes x and y,

$$\frac{f(x)-f(y)}{x-y} = \int_0^1 f'((1-s)x+sy)\,ds\,,$$

and iterating integrals in the following fashion,

$$\int_{[\tau_0,\tau_1,\ldots,\tau_n]} f := \int_{s_1=0}^{1}\int_{s_2=0}^{s_1}\cdots\int_{s_n=0}^{s_{n-1}} f((1-s_1)\tau_0+\cdots+(s_{n-1}-s_n)\tau_{n-1}+\tau_n)\,ds_n\cdots ds_1\,,$$

we find that the error term $K(z)$ becomes an *average value of the* $d+1$st *derivative of* f:

$$\frac{1}{2\pi i}\oint_C \frac{f(\zeta)}{w(\zeta)(\zeta-z)}\,d\zeta = \int_{[\tau_0,\tau_1,\ldots,\tau_n]} f^{(d+1)}\,. \tag{8.61}$$

In the complex case, it is no longer always possible to say that this average value is actually attained at some (possibly unknown) point $z=c$.

We now have almost all the tools in place to show that for *analytic* functions $f(z)$, the error in interpolation at Chebyshev or other good sets of nodes is more than just accurate to a power $O(h^p)$, but is in fact *spectrally accurate* inside the so-called Bernstein ellipses. There is also a fascinating exploration using potential theory (!) that explains good choices of interpolation nodes in the complex plane. Very reluctantly we refrain from giving our own exposition of this beautiful theory; but our reluctance is completely mollified by the existence of the very clear and concise treatment in Chaps. 8 and 12 of Trefethen (2013).

The cost of solving a Vandermonde system by ordinary dense matrix methods is $O(d^3)$, but this takes no advantage of structure. Special algorithms that are both faster ($O(d^2)$) and surprisingly more accurate are discussed in Higham (2002 chapter 22). See Problem 8.35 and, next chapter, Problems 9.9–9.10. The exponential ill-conditioning of Vandermonde matrices on arbitrary real nodes is proved in Beckermann (2000). Bounds for Chebyshev nodes and equally spaced nodes were given in Gautschi (1975) and in Gautschi (1983), and the results are summarized in Higham (2002 table 22.1).

Rational interpolation in barycentric form is discussed in Schneider and Werner (1991), where the good forward accuracy of the second barycentric form for polynomials is noted and explained with backward error. They also note the criterion $\alpha_{k,s_k-1} = 0$ for unattainable points. Rational barycentric interpolation is an active topic of current research: A good entry point to the literature is Berrut et al. (2005). The paper Brezinski and Redivo-Zaglia (2012) discusses a method to choose the barycentric weights in such a way as to achieve the Padé property, namely, to match the Taylor series at the origin up to a specified order; this mixes interpolation with Padé approximation and is a kind of nonlinear generalization of Hermite interpolation. Other excellent and useful works on rational interpolation include Berrut and Mittelmann (2000) and Brezinski (1980). A motivating rationale for the use of rational functions can be found in Trefethen (2013 chapter 23). One reason to use rational interpolation involves what are called "shape-preserving" interpolants. A good introduction can be found in Brankin and Gladwell (1989). The idea is to use a rational interpolant that contains a parameter that can be tuned so as to preserve qualitative features such as convexity of the solution or monotonicity of the solution. This can easily be handled within the framework of barycentric forms, as is demonstrated in an example in Butcher et al. (2011).

The MATLAB routine `spline` is rather similar to the spline discussed here, except it uses local monomial bases. The MATLAB routine `pchip` is somewhat different, giving up continuity of the second derivative in order to obtain a more pleasing visual appearance and better preservation of monotonicity and convexity of the data when the data are sparse. See Moler (2004) for details, and see also Problem 8.40. A fuller discussion can be found in Shampine et al. (1997), and original sources are referenced in de Boor (1978).

It turns out that the equations defining the derivatives $\rho_{k,1}$ to ensure twice-continuous differentiability of a spline are just the same as what are called "compact finite differences," which are discussed later in Sect. 11.6, although there we will also want to find expressions for the derivatives at the edges. Indeed that could be useful here too instead of using natural splines, clamped splines, or the not-a-knot condition, but to our knowledge nobody does splines that way; this might simply be because it hasn't occurred to anyone, or contrariwise there might be a problem with this idea that we don't know of (because we haven't tried it on enough examples). In any case, this connection between cubic splines and compact finite differences does not seem to be widely known. This is possibly because the usual derivation of cubic splines uses the local monomial bases, which obscures the meaning of the tridiagonal system.

A very nice survey of results for barycentric rational interpolation can be found in Pachón et al. (2012), together with a new method for solving the *Cauchy problem*: Find $p(z)$ and $q(z)$ with degrees m and n such that $m+n=N$, where nodes τ_k are given for $0 \le k \le N$ and values f_k are known, and $R(\tau_k) = {}^{p(\tau_k)}/_{q(\tau_k)} = f_k$. Notice this is different to the problem solved in this text, where the denominator $q(z)$ was known; here only the degree constraint is presumed known, that is, the integers m and n. This will be taken up again in Chap. 11, and also in Problem 8.42.

Problems

Theory and Practice

8.1. Consider the nodes $\tau_0 = -1$ and $\tau_1 = 1$.

(i) Find the partial fraction expansion of

$$\frac{1}{(z-\tau_0)(z-\tau_1)}\,.$$

(ii) If $p(\tau_0) = \rho_0$ and $p(\tau_1) = \rho_1$, express the linear polynomial fitting this data in the 2nd barycentric form,

$$p(z) = \frac{\dfrac{\rho_0/2}{z+1} + \dfrac{\rho_1/2}{z-1}}{\dfrac{-1/2}{z+1} + \dfrac{1/2}{z-1}}\,.$$

Obviously, this is simpler to do in other ways; this is just practice.

(iii) By using the partial fraction decomposition of

$$\frac{1}{(z-\tau_0)^2(z-\tau_1)}\,,$$

find a quadratic polynomial with $p(\tau_0) = \rho_{0,0}$, $p'(\tau_0) = \rho_{0,1}$, and $p(\tau_1) = \rho_{1,0}$.

8.2. Use the Cauchy integral theorem and the fact that

$$\oint_C \frac{f(z)}{(z-t)w(z)}dz = 0$$

if $\deg_z f(z) \le d$ and the contour C encloses t and all τ_k in $w(z) = \prod_{k=0}^{n}(z-\tau_k)$ (assuming t and the τ_k are distinct) to show that

$$f(z) = w(t)\sum_{k=0}^{d} \frac{\beta_k f(\tau_k)}{t-\tau_k}\,.$$

Fill in the details of why the integral must be zero for any polynomial $f(z)$ of degree at most d.

8.3. Let $L_k(t)$ be defined by

$$L_k(t) = \frac{\beta_k w(t)}{t-\tau_k}.$$

Show that $\deg L_k(t) = d$ and $L_k(\tau_j) = \delta_j^k$. These are the Lagrange basis polynomials. (Strictly speaking,

$$L_k(t) := \begin{cases} \frac{\beta_k w(t)}{t-\tau_k} & t \neq \tau_k \\ 1 & t = \tau_k \end{cases}$$

removes the singularity to make it polynomial. We presume henceforth that all such removable singularities are automatically removed without comment.) Then, if $f(t)$ has degree at most d,

$$f(t) = \sum_{k=0}^{d} f(\tau_k) L_k(t) \,.$$

(This just rewrites Problem 8.2.)

8.4. For the polynomial and nodes in Example 8.3, swap the nodes in $0 < z < 1/2$ by the transformation $z \mapsto \frac{1}{2} - z$ and swap the nodes in $1/2 < z < 1$ by the transformation $z \mapsto \frac{3}{2} - z$. This should make the nodes dense near $z = 1/2$ and sparse near the edges. Compute the condition number for evaluation of the polynomial $T_{16}(2z-1)$ expressed in the Lagrange basis on these nodes and show that it is orders of magnitude worse even than the condition number for equally spaced nodes.

8.5. Suppose that we have a set of distinct nodes, $\{\tau_k\}$, $0 \le k \le n$, and have previously computed the barycentric weights β_k for Lagrange interpolation. Now suppose that we wish to add a new node τ_{n+1}. How much does it cost to update the existing β_k and to compute a new β_{n+1}? How much does it cost to update the β_k in order to *remove* a node (say τ_n) instead of adding a node τ_{n+1}?

8.6. With the nodes $\boldsymbol{\tau} = [0, h, 1]$, compute the generalized barycentric weights for Hermite interpolation for confluencies 2—that is, compute the partial fraction decomposition of

$$\frac{1}{z^2(z-h)^2(z-1)^2} \,.$$

Compute $\partial\beta_{i,j}/\partial h$. Show that the generalized barycentric weights [specifically, the weight $\beta_{1,0}$ corresponding to $1/(z-h)$] are ill-conditioned as $h \to 1/2$—that is, as the mesh width becomes equally spaced. Consider as well the symmetric problem

$$\frac{1}{(z+h)^2 z^2 (z-h)^2} \,,$$

which is clearly similar, but in this case you are to show that there is no difficulty whatever. Note that in the one case, perturbations δh may be asymmetric, but not in the other. Discuss.

8.7. Note that

$$w(z) = \prod_{k=0}^{n} (z-\tau_k)^{s_k} = w_i(z)(z-\tau_i)^{s_i} \,,$$

where

$$w_i(z) = \prod_{\substack{k=0\\k\neq i}}^{n} (z-\tau_k)^{s_k}\,.$$

Show that

$$H_{ij}(z) = w_i(z)(z-\tau_i)^j \sum_{m=0}^{s_i-1-j} \beta_{i,s_i-1-m}(t-\tau_i)^m$$

is a polynomial satisfying

$$\frac{1}{\ell!}H_{ij}^{(\ell)}(\tau_m) = \delta_i^m \delta_j^\ell$$

for $1 \leq i,m \leq n, 0 \leq j,\ell \leq s_i - 1$ (δ_j^k is the Kroenecker delta, 0 if $j \neq k$ and 1 if $j = k$). This verifies directly that we have an explicit construction of the Hermite interpolation basis on the nodes τ_k with confluencies s_k.

8.8. For $\rho_{1,0} = 1, \rho_{1,1} = -1, \rho_{0,0} = 1, \rho_{0,1} = -1$, on the nodes $\tau = [-1,1]$, consider the condition number of a polynomial taking on values ± 1 in the Hermite interpolational basis with confluency 2 at each node:

$$B(z) = \sum_{i=0}^{1}\sum_{j=0}^{1} |\pm 1| \cdot |H_{ij}(z)|\,.$$

For $z = x + iy \in \mathbb{C}$, near the real segment $-1 \leq x \leq 1$, plot level contours of $B(z)$, using an appropriate scale. You should find that $B(z)$ is smaller when z is nearer the segment and grows rapidly for z away from the segment. This gives a quantitative measure to the intuitive idea that $f(z)$ is well-determined near to where its data are specified.

8.9. Fill in the details of the proof in the text on page 364 for the Lagrange case of rational interpolation. If $\deg q(z) \leq d$ and the partial fraction expansion of q/w is

$$\frac{q(z)}{w(z)} = \sum_{k=0}^{d} \frac{\alpha_k}{z-\tau_k}\,,$$

where no α_k is zero, show that

$$R(z) = \frac{\displaystyle\sum_{k=0}^{d} \frac{\alpha_k f(\tau_k)}{z-\tau_k}}{\displaystyle\sum_{k=0}^{d} \frac{\alpha_k}{z-\tau_k}}$$

is a rational function that interpolates $f(z)$ at $z = \tau_k$, that is,

$$\lim_{z\to\tau_k} R(z) = f(\tau_k), \qquad 0 \le k \le d\,,$$

and moreover that $R(z) = {}^{p(z)}\!/_{q(z)}$ where

$$p(z) = w(z)\sum_{k=0}^{d} \frac{\alpha_k f(\tau_k)}{z-\tau_k}$$

is a polynomial of degree at most d. Remark: This shows that even incorrect barycentric weights (through poorly known nodes or through computational error) will still produce an interpolant, that is, will still fit the data, if the second barycentric form is used.

8.10. We now examine rational Hermite interpolants with preassigned denominators. Fill in the details of the proof in the text on page 364 and show that if

$$q(z) = w(z)\sum_{i=0}^{n}\sum_{j=0}^{s_1-1} \alpha_{ij}(z-\tau_i)^{-j-1}$$

[the partial fraction expansion of ${}^{q(z)}\!/_{w(z)}$] and no α_{i,s_i-1} is zero, then

$$R(z) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\sum_{k=0}^{j} \alpha_{ij}\rho_{ik}(z-\tau_i)^{k-j-1}}{\sum_{i=0}^{n}\sum_{j=0}^{s_i-1} \alpha_{ij}(z-\tau_i)^{-j-1}}$$

is such that

$$\lim_{z\to\tau_i} \frac{R^{(j)}(z)}{j!} = \rho_{ij}$$

for each $0 \le j \le s_i - 1$. If any $\alpha_{i,s_i-1} = 0$, τ_i is said to be an “unattainable point” for ${}^{p(z)}\!/_{q(z)}$. Give an explicit example of a rational interpolation problem with known denominator that has an unattainable point.

8.11. We look at the numerical difficulties that come with near-confluency. Consider Lagrange interpolation with just two nodes, and suppose τ_0 and τ_1 are close, but not equal. Thus, we are interpolating some linear polynomial; for example, take $p(z) = 1 + 2z$.

1. Show that β_1 and β_0 are large, and indeed go to infinity as $\tau_1 \to \tau_0$.
2. Show that evaluation of either barycentric form is subject to catastrophic cancellation when evaluating.
3. The condition number is

$$B(z) = \sum_{k=0}^{1} |\rho_k|\cdot|L_k(z)|$$

and the sum $|\rho_0|\cdot|L_0(z)| + |\rho_1|\cdot|L_1(z)|$ goes to infinity as $\tau_1 \to \tau_0$. What does this mean? (The moral of this is that nodes that are “too close” cause problems.)

8.12. Consider Hermite–Birkhoff interpolation, also referred to as the case of missing data. Suppose we are told that $f(-1) = 1$, $f'(r) = 0$, and $f(1) = 2$ (r is some number in $[-1,1]$). By doing a partial fraction decomposition on

$$\frac{1}{(z+1)(z-r)^2(z+1)},$$

find the value of $f(r)$ for a degree-2 polynomial $f(z)$ that fits the data. If $r = 0$, the problem is not solvable (as you will discover); in this situation, we say that the Birkhoff interpolation problem is not *poised*. (See Butcher et al. (2011) for the background to this problem after you have solved it.)

8.13. The Newton basis

$$a, x-\tau_1, (x-\tau_1)(x-\tau_2), (x-\tau_1)(x-\tau_2)(x-\tau_3), \ldots$$

is well known in classical numerical methods. However, its numerical conditioning depends on the ordering chosen for the nodes, and this can have undesirable consequences. [This fact is noted in Battles and Trefethen (2004), for example, but was certainly known previously.]

1. For the Chebyshev nodes on $[-1,1]$, plot the condition number

$$B(z) = \sum_{k=0}^{n} \left| \beta_k \frac{T_n(x)}{x-\tau_k} \right|$$

(take $n = 10$ and $n = 20$) (this is the Lagrange basis on these nodes). Show these are independent of the ordering of the nodes.

2. Sort the τ_k into increasing order. Let

$$p(x) = \sum_{k=0}^{n} (-1)^k \beta_k \frac{T_k(x)}{x-\tau_k},$$

so $p(\tau_k) = (-1)^k, 0 \le k \le n$. Compute the coefficients c_k in

$$p(x) = \sum_{k=0}^{n-1} c_k N_k(x),$$

where $N_0(x) = 1, N_1(x) = x-\tau_1, N_2(x) = (x-\tau_1)(x-\tau_2)$, etc. [Hint: $p(\tau_1) = c_0$ because $N_k(\tau_1) = 0$ for $k \ge 2$. Then $(p(x)-p(\tau_1))/(x-\tau_1)$ is also a polynomial.] This approach is called divided differences. Now plot

$$B_N(x) = \sum_{k=0}^{n-1} |c_k| \cdot |N_k(x)|$$

on $[-1,1]$. Compare to B_L.

3. Repeat Problem 2, but this time shuffle the nodes τ_k into a random order, first. It turns out that computation of a good ordering that minimizes $B_N(x)$ is fairly simple.
4. The Leja ordering is $\xi_0 = \tau_{\arg\max(|\tau_k|)}$ and, for $i = 1, 2, \ldots, n$, find τ_j maximizing

$$\prod_{k=0}^{i-1} |\tau_j - \xi_k| \,.$$

   Let $\xi_i = \tau_j$. Reorder the nodes into a Leja ordering and repeat Problem 2, using this ordering.

8.14. Compute the condition numbers for the following change-of-basis matrices. Compare with the results of Gautschi (1983).

1. From the Lagrange basis on the equally spaced points $\tau_k = -1 + {}^{2k}\!/_{n}$, for $n = 5$, 8, 13, and 21 (the simple Vandermonde matrix).
2. From the Lagrange basis on the Chebyshev nodes $\tau_k = \cos(\pi(k + {}^{1}\!/_{2})/(n+1))$, for $n = 5$, 8, 13, and 21.
3. From the Lagrange basis on the roots of unity $\tau_k = \exp({}^{2\pi ik}\!/_{n})$.
4. From the Chebyshev orthogonal basis to the monomial basis, for enough n to see the trend.
5. From the Chebyshev orthogonal basis to the Legendre orthogonal basis, for enough n to see the trend.

8.15. Let $\mathbf{T}$ be the change-of-basis matrix from the Lagrange basis on the Chebyshev –Lobatto points $\eta_k = \cos({}^{\pi k}\!/_{n})$, $0 \le k \le n$, to the Chebyshev polynomial basis $T_j(x) = \cos(j\cos^{-1}x)$. Suppose $n \ge 6$. Perhaps by looking at the entries of $\mathbf{C} = \mathbf{T}^H\mathbf{T}$, which are simpler, establish that the largest singular value of $\mathbf{T}$, call it σ_0 because we are indexing from 0 here, satisfies

$$\sigma_0 \sim \sqrt{n} + O(1)$$

as $n \to \infty$, while the smallest singular value (which occurs $n-3$ times) is just $\sigma_n = \sqrt{{}^{n}\!/_{2}}$. (In fact, $\sigma_0^2 = n + {}^{1}\!/_{2} + \sqrt{n + {}^{1}\!/_{4}}$ if n is even, and $\sigma_0^2 = n + {}^{1}\!/_{4} + \sqrt{{}^{n}\!/_{2} + {}^{1}\!/_{16}}$ if n is odd.) Thus establish that the condition number of $\mathbf{T}$ in the 2-norm is asymptotically $\sqrt{2} = 1.414\ldots$ as $n \to \infty$. This shows that changing basis from the Lagrange basis on the Chebyshev–Lobatto points to the Chebyshev polynomial basis is one of the few instances where changing basis is well-conditioned.

8.16. By considering what happens when you multiply the vector $[0, L_0(x), L_1(x), \ldots, L_n(x)]^T$ by each of the matrices $\mathbf{A}$ and $\mathbf{B}$ in the companion pencil for the Lagrange basis, show that roots of the polynomial $p(x) = w(x)\sum_{k=0}^{n} {}^{\beta_k\rho_k}\!/_{(x-\tau_k)}$ are eigenvalues, that is, roots of $\det(x\mathbf{B} - \mathbf{A})$.

8.17. Find a polynomial for which the condition number for evaluation is better in the Bernstein–Bézier basis than in either a Lagrange interpolational basis or a Hermite interpolational basis. Of course, this is easy if one chooses deliberately foolish interpolation points; try instead to find an example that is fair and for which the Bernstein–Bézier basis wins cleanly.

8.18. Suppose a matrix $\mathbf{A}$ has eigenvalues λ_k each with multiplicity $m_k \geq 1$. Suppose we interpolate an analytic function $f(z)$ by choosing nodes $\tau_k = \lambda_k$ with confluencies $s_k = m_k$; that is, we find $p(z)$ of degree at most $d = -1 + \sum_k s_k$ satisfying $p^{(j)}(\lambda_k) = f^{(j)}(\lambda_k)$, for $0 \leq j \leq m_k - 1$ and $1 \leq k \leq \sum_\ell m_\ell$. Then [as you may read in Higham (2008)] it is possible to define the *function $f(\mathbf{A})$ of the matrix* $\mathbf{A}$ by the polynomial $p(\mathbf{A})$. Thus, all matrix functions are polynomials in a sense, although the polynomial will change as the matrix changes—it's not just one polynomial for a given function. A polynomial of a matrix makes sense in the monomial basis, so that if say $p(z) = 1 + 2z + 3z^2$, then $p(\mathbf{A}) = \mathbf{I} + 2\mathbf{A} + 3\mathbf{A}^2$.

1. Show that the first barycentric form of a polynomial can be made to make sense for matrix arguments. Be careful about $\mathbf{A} - \lambda\mathbf{I}$. In practice, for matrix functions one would want the explicit expression in terms of the Hermite interpolational basis, because $(\mathbf{A} - \tau_i\mathbf{I})^{-1}$ does not appear, and, of course, in this case the τ_i are exactly the eigenvalues.
2. Discuss the second barycentric form for matrix arguments.
3. Compute the exponential of the `gallery(3)` matrix in MATLAB using the above ideas. Note that `exp(A)` is not what we want (though `expm(A)` works, using the monomial basis).
4. Compute the logarithm of the `gallery(3)` matrix in MATLAB using the above ideas. Again, `log(A)` isn't what we want, though `logm(A)` is the built-in version.
5. Verify the previous two computations by computing the exponential of the logarithm and the logarithm of the exponential using your method (and possibly comparing with the built-in routines).

8.19. Use the Schur determinantal formula to prove that if the nodes τ_k for $0 \leq k \leq n$ are distinct and a polynomial $p(z)$ of degree at most n takes on the values $\rho_k = p(\tau_k)$, then the roots of the polynomial $p(z)$ are eigenvalues of the *Lagrange companion pencil* $(\mathbf{A}, \mathbf{B})$, where both $\mathbf{A}$ and $\mathbf{B}$ are $(n+1) \times (n+1)$, $\mathbf{B}$ is the same as the identity matrix except that $B_{1,1} = 0$, the first column of $\mathbf{A}$ is $[0, \beta_0, \beta_1, \ldots, \beta_n]$, where the β_k are the barycentric weights, the diagonal of $\mathbf{A}$ [starting in the $(2,2)$ position] are $\tau_0, \tau_1, \ldots, \tau_n$ and the first row of $\mathbf{A}$ is [starting in the $(2,1)$ position] $\rho_0, \rho_1, \ldots, \rho_n$. This is called an arrowhead matrix. Show also that this pencil has at least two infinite eigenvalues. Indeed, if we use the negatives of either the ρs or the βs instead, we get precisely that $\det(z\mathbf{B} - \mathbf{A}) = p(z)$ expressed in the first barycentric form, directly.

8.20. Show that the companion pencil $(\mathbf{A}, \mathbf{B})$ for Hermite interpolation is related to that of the Lagrange form, except that the diagonal now contains Jordan blocks for each confluent node, of dimension equal to the confluency. What happens if the confluency equals n? That is, all the information is given about a single node, say τ_0?

8.21. How do the generalized eigenvalues of the pair $(\mathbf{A},\mathbf{B})$ relate to a rational interpolant of (τ_k,ρ_k) if the β_k in $\mathbf{A}$ are replaced with another vector of nonzero entries α_k?

8.22. Nonlinear eigenproblems ask for those λ making $\mathbf{P}(\lambda)$ singular, where each entry in the matrix $\mathbf{P}$ is a nonlinear function of λ. Here we consider only polynomial functions, in which case we speak of a matrix polynomial. The matrix polynomial can be expressed in any polynomial basis. This is, as mentioned in the text, a simultaneous generalization of polynomials and of eigenvalue problems. One can convert polynomial nonlinear eigenproblems into generalized eigenproblems by a process somewhat misleadingly termed *linearization*: Essentially, it replaces the matrix polynomial (of degree n and where the matrix is $s\times s$ in size) with a matrix pencil of size $ns\times ns$. In the case where the basis is the Lagrange (or Hermite) interpolational basis, it is of size $(n+2)s\times(n+2)s$ and there are $2s$ spurious eigenvalues at infinity (which do not in general bother us). A linearization that works in this case is to replace the ρ_k in the Lagrange (or Hermite) polynomial linearization with $s\times s$ matrix entries, replace the τ_k with $\tau_k\mathbf{I}$, replace any 1s in any Jordan blocks with $\mathbf{I}$, and replace the $\beta_{k,j}$ with $\beta_{k,j}\mathbf{I}$.

1. Take the nodes $\boldsymbol{\tau}=[-1,0,1]$ and suppose that our degree-2 matrix polynomial takes on the 3×3 matrix values `pascal(3)`, `gallery('grcar',3)`, and `gallery('clement',3)` at those nodal values. Find the linearization $(\mathbf{A},\mathbf{B})$ (which is 12×12 and has six spurious infinite eigenvalues) and find its (generalized) eigenvalues.
2. For each of the finite eigenvalues found in the previous step, form (perhaps by explicit polynomial Lagrange interpolation) the matrix $\mathbf{P}(\lambda)$ and verify by computing its SVD that it is numerically singular.

8.23. We remind you that the Chebyshev polynomials are $T_n(x)=\cos(n\arccos(x))$ if $n\geq 0$ and $-1\leq x\leq 1$. The first three are $T_0(x)=1$, $T_1(x)=x$, and $T_2(x)=2x^2-1$.

1. Prove that $T_{n+1}(x)=2xT_n(x)-T_{n-1}(x)$ and find $T_3(x)$.
2. Show that the zeros of $T_n(x)$ are $\xi_k=\cos\left({}^{(2k-1)\pi}/_{2n}\right)$ for $1\leq k\leq n$. (Therefore, all roots are real and lie in $-1\leq\xi\leq 1$.)
3. Let
$$w_n(x)=\prod_{k=1}^{n}(x-\xi_k)=2^{1-n}T_n(x)\,.$$
Show that the barycentric weights corresponding to interpolation at the ξ_k, equivalently, the β_k that appear in the partial fraction decomposition of
$$\frac{1}{w_n(x)}=\sum_{k=1}^{n}\frac{\beta_k}{x-\xi_k}\,,$$
are given by
$$\beta_k=\prod_{j\neq k}(\xi_k-\xi_j)^{-1}=(-1)^{k-1}\frac{2^{n-1}\sqrt{1-\xi_k^2}}{n}\,.$$

4. Interpolate the following data using a polynomial expressed in the Lagrange basis on the given points: at

$$\boldsymbol{\tau} = \left[-\frac{\sqrt{3}}{2}, 0, \frac{\sqrt{3}}{2}\right];$$

we have $\boldsymbol{\rho} = [1, -1, 1]$. You may express your answer as using the polynomial Lagrange basis, or in the first barycentric form, or in the second barycentric form.

5. Sketch the condition number

$$B(x) = \sum_{k=1}^{3} \frac{|\beta_k w_3(x)|}{|x - \xi_k|} .$$

8.24. Again we remind you that the Chebyshev polynomials $T_n(z)$ are defined by $T_n(z) = \cos(n\theta)$, where $\cos\theta = z$. They satisfy the recurrence relation $T_{n+1}(z) = 2zT_n(z) - T_{n-1}(z)$, and it is easy to see that $T_0(z) = 1$ and $T_1(z) = z$. The Chebyshev–Lobatto points $\eta_k^{(n)} = \cos(\pi k/n)$ for $0 \le k \le n$ are the places in $-1 \le z \le 1$ where $T(z)$ is maximal, that is, ±1.

1. Show that the leading coefficient of $T_n(z)$, when expressed in the monomial basis, is 2^{n-1} if $n \ge 1$.
2. Show that the polynomial

$$w(z) = \prod_{k=0}^{n}(z - \eta_k^{(n)}) = \frac{1}{2^n}\left(T_{n+1}(z) - T_{n-1}(z)\right)$$

if $n \ge 1$.

3. Show that a fast way to compute the barycentric weights

$$\beta_k := \prod_{\substack{\ell=0 \\ \ell \ne k}}^{n} (\tau_k - \tau_\ell)^{-1}$$

is to use the formula $\beta_k = 1/w'(\tau_k)$. Identify the barycentric weights for the Chebyshev–Lobatto points using the above results.

4. Explicitly compute the $\eta_k^{(n)}$ for $n = 3$ and by using the formula for a simple partial fraction expansion compute the barycentric weights.
5. If $\rho = [1, -1/2, -1/2, 1]$ on those four points, write the first barycentric form of the Lagrange interpolant going through the points $(\eta_k^{(3)}, \rho_k)$.
6. Use the companion matrix pencil from section 8.2.1 (each of **A** and **B** is 5 × 5) in MATLAB to compute the generalized eigenvalues and thus the roots of $p(z)$. Why are there only two finite eigenvalues?

8.25. For cubic Hermite interpolation, we have been using the barycentric forms. The divided-difference form (which we have not recommended, because of the dependence on ordering) is perfectly fine for this case because there is no possibility of an ordering problem for only two nodes! Solve the cubic Hermite interpolation problem with divided differences.

8.26. Show that cubic Hermite interpolants have second derivative accuracy $O(h^2)$ and third derivative accuracy $O(h)$, but that the fourth derivative will be right only if the original problem has identically zero fourth derivative.

8.27. Show that the following is a companion matrix pencil for the Newton basis (this can be derived similarly to orthogonal bases, because Newton bases have a very simple three-term recurrence). For simplicity, only the degree-3 case $p(z) = c_0 + c_1(z-\tau_0) + c_2(z-\tau_0)(z-\tau_1) + c_3(z-\tau_0)(z-\tau_1)(z-\tau_2)$ case is shown, but the general form should be apparent.

$$\mathbf{A} = \begin{bmatrix} \tau_0 & 1 & \\ & \tau_1 & 1 \\ -c_0 & -c_1 & -c_2 + c_3\tau_2 \end{bmatrix} \qquad \text{and} \qquad \mathbf{B} = \begin{bmatrix} 1 & & \\ & 1 & \\ & & c_3 \end{bmatrix}. \tag{8.62}$$

That is, show that $\det(z\mathbf{B} - \mathbf{A}) = p(z)$.

8.28. Show that if $\tau_j < \tau_k$ for $j < k$, then the Newton basis 1, $z-\tau_0$, $(z-\tau_0)(z-\tau_1)$, …, $(z-\tau_0)\cdots(z-\tau_{n-1})$ can be expressed as a nonnegative combination of the Lagrange basis on those nodes and that therefore the condition number of a polynomial expressed in the Newton basis with this ordering cannot be better on the nodes than the Lagrange basis can; show by example that there are cases where this inequality is strict in a neighborhood of the nodes.

8.29. Show that the barycentric weights for the nodes $\tau_k = \xi_k^{(n)}$, which are the zeros of the Chebyshev polynomial $T_n(z)$, are given by $\beta_k = (-1)^k 2^{n-1}\sqrt{1-\xi_k^{(n)}}/n$ and that the factors that depend only on n, not k, can be dropped for use in the second barycentric form or in the companion matrix pencil. Show that the maximum ratio $|\frac{\beta_j}{\beta_\ell}|$ grows like n^2.

8.30. If the nodes $\tau_k = a + {}^{k(b-a)}/_{n}$ for $0 \le k \le n$ are used, show that the barycentric weights are

$$\beta_k = (-1)^{n-k}\left(\frac{n}{b-a}\right)^n \frac{1}{n!}\binom{n}{k}. \tag{8.63}$$

Show that the maximum ratio $|{}^{\beta_j}/_{\beta_\ell}|$ grows essentially like 2^n as $n \to \infty$.

8.31. Symmetry is sometimes important in a problem. Show how to use the barycentric forms for the Lagrange basis and the Hermite interpolational basis in a way that preserves the even symmetry $f(z) = f(-z)$ and in another way that preserves the odd symmetry $f(z) = -f(-z)$. Give an even barycentric expression that fits the data

z	f(z)
1.725	73.430
1.035	−86.515
0.975	−90.475
0.405	−138.670
0.015	−154.984

and an odd expression for the derivative using your modified barycentric forms.

Investigations and Projects

8.32. The paper Hyman (1983) describes a simple monotonicity-preserving cubic Hermite interpolant; once the slopes $\rho_{i,1}$ are computed as here for the spline interpolant, the slopes are *filtered* and excessive slopes are scaled back so as to preserve monotonicity. Implement the method in MATLAB and test it on monotonic and nonmonotonic examples.

8.33. As we have seen, global polynomial interpolation of functions has error proportional to a high derivative evaluated somewhere (or an average of such values). This suggests that interpolation of functions with singularities will run into problems, and of course it does. Adaptivity and the use of piecewise polynomials come to our rescue here. Consider, for example, the function $f(z) = \sqrt{|z|}$ on $-1 \le z \le 1$. Try interpolating this on 50 Chebyshev–Lobatto points, and show that the error is about 10^{-1} (and worst near the derivative singularity) and that increasing the number of Chebyshev–Lobatto points helps, but not much. Try instead to choose a mesh that more nearly reflects the presence of the singularity—more points near the singularity, that is—and use piecewise-cubic Hermite interpolation (you could use a spline or pchip, but derivative data are available so that you could use genuine piecewise-cubic Hermite interpolation if you desired, and this can be expected to be more accurate). Can you find a family of meshes that regains the theoretical $O(h^4)$ convergence as $h \to 0$ in some sense? (This problem foreshadows the topic of *equidistribution*, which we shall meet in Chap. 10.)

Try also `chebfun` with “splitting on,” which produces a piecewise chebfun approximation. Plotting the relative error at a large number of points shows that it is accurate up to roundoff levels using 14 pieces with degrees at most 126 in the version current at this time of writing.

8.34. The text claims that the computation of zeros of $p(z)$ known by values ρ_k at distinct nodes τ_k for $0 \le k \le n$ is numerically stable, that is, that the computed eigenvalues of the pencil $(\mathbf{A}, \mathbf{B})$ are the exact roots of a polynomial taking on values near to the ρ_k. Prove this. Hint: With the 0 in the upper-left corners, the two spurious infinite eigenvalues are found exactly and the degree of $\det(z\mathbf{B} - \mathbf{A})$ is at most n, so this need not concern you. What you need to show for this problem is that if all of the computed eigenvalues λ_k are the *exact* eigenvalues of some $(\mathbf{A} + \Delta\mathbf{A}, \mathbf{B} + \Delta\mathbf{B})$ with bounds of size $O(\mu_M)$ on the norms of the perturbations, then $\det(\tau_k(\mathbf{B} + \Delta\mathbf{B}) -$

$(\mathbf{A}+\Delta\mathbf{A})) = \rho_k + C_k\mu_M + O(\mu_M^2)$ for $0 \le k \le n$. A useful formula for the derivative of a determinant, named after Jacobi, is given below. The formula is

$$\frac{d}{dt}\det(\mathbf{A}) = \operatorname{trace}\left(\operatorname{adjugate}(\mathbf{A})\frac{d\mathbf{A}}{dt}\right) .$$

In this case, for $z = \tau_k$, the adjugate of $z\mathbf{B}-\mathbf{A}$ is particularly simple. Show also that C_k is bounded by the maximum ratio of barycentric weights and inversely bounded by the minimum separation $|\tau_i - \tau_j|$ for $i \neq j$. An overview of the proof appears in Lawrence and Corless (2011), but try it for yourself first.

8.35. Algorithms 22.2 and 22.3 in Higham (2002) for solving generalized Vandermonde matrices work by first changing the interpolation basis from a Hermite interpolational basis to a Newton basis, and then converting from that basis to the desired basis $\phi_k(z)$. This often works surprisingly well, giving small componentwise forward error. For some examples, however, if the nodes τ_k are taken in a bad order, the algorithm is potentially unstable.

1. Find an example where $\kappa(\mathbf{V})$ is very large but conversion to the Newton basis and then to the monomial basis gives good accuracy.
2. Find an example where $\kappa(\mathbf{V})$ is *small* but where the method does not give good accuracy.
3. Discuss what happens if there is nontrivial *data error* $\Delta\boldsymbol{\rho}$. Does the accurate method of solving the system help much? Does it hurt much?

8.36. In this problem, we investigate the properties of some typical change-of-basis matrices.

1. Show that if $\phi_k(x) = \sum_{j=0}^{n}\phi_{kj}x^j$, then the k'th row of $\mathbf{A}_{\phi,x^k}$ is $[\phi_{k0},\phi_{k1},\ldots,\phi_{kn}]$. Moreover, show that if $\deg\phi_k = k$, then $\mathbf{A}_{\phi,x^k}$ is lower-triangular.
2. Write the $n=5$ change-of-basis matrix for the Bernstein–Bézier polynomial $\phi_k(x) = \binom{5}{k}x^k(1-x)^{5-k}$ explicitly from the monomial basis.
3. For the Bernstein–Bézier and for the Chebyshev change-of-basis matrices (from the monomial basis), compute and plot the singular values σ_1 and σ_n^{-1} for $n = 3,5,6,13,21,34,55,\ldots$ on a log scale. Does the size of $\mathbf{A}_{\psi,x^k}$ grow polynomial with n, exponentially, or faster? What does this imply for the amplification of uncertainty when changing basis? (Moral: Changing bases can be a very bad idea.)
4. Perhaps the most famous change-of-basis matrices are those from Lagrange and Hermite interpolational bases to the monomial basis. Because

$$x^k = \sum_{j=0}^{n}\tau_j^k L_j(x)$$

for $0 \le k \le n$, where

$$L_j(x) = \beta_j\prod_{\substack{\ell=0\\ \ell\neq j}}^{n}(x-\tau_\ell) ,$$

we have

$$\left[1\ x\ x^2\ \dots\ x^n\right] = \left[L_0\ L_1\ \dots\ L_n(x)\right]\mathbf{V},$$

where the matrix

$$\mathbf{V} = \begin{bmatrix} 1 & \tau_0 & \tau_0^2 & \cdots & \tau_0^n \\ 1 & \tau_1 & \tau_1^2 & \cdots & \tau_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \tau_n & \tau_n^2 & \cdots & \tau_n^n \end{bmatrix}$$

is the Vandermonde matrix. For the Hermite interpolation problem, the matrix is the "confluent" Vandermonde matrix

$$\mathbf{V} = \begin{bmatrix} 1 & \tau_1 & \tau_1^2 & \cdots & \tau_1^n \\ 0 & 1 & 2\tau_1 & \cdots & n\tau_1^{n-1} \\ 0 & 0 & 2 & \cdots & n(n-1)\tau_1^{n-2} \\ 1 & \tau_2 & \tau_2^2 & \cdots & \\ \vdots & \vdots & \vdots & & \end{bmatrix},$$

where a node of confluency s_i induces s_i rows containing τ_i, and each row except the first is formed by evaluating the derivative of the row above. The generalized Vandermonde matrix arises from the relation

$$\phi_k(x) = \sum_{j=0}^{n} \phi_k(\tau_j) L_j(x)$$

in the Lagrange case and

$$\phi_k(x) = \sum_{i=1}^{n} \sum_{j=0}^{s_1-1} \frac{\phi_k^{(j)}(\tau_i)}{j!} H_{ij}(x)$$

in the Hermite case.

a. Show that the coefficients of $H_{ij}(x), 1 \le i \le n, 0 \le j \le s_i - 1$, give the entries the inverse of $\mathbf{V}_c$. ($\mathbf{V}_c$ is nonsingular if $\tau_i = \tau_j \Leftrightarrow i = j$.)
b. Show that the condition number of the Vandermonde matrix based on equally spaced nodes grows at least exponentially with n (a computational demonstration is sufficient; see Higham (2002) or Gautschi (1983) for general proofs).
c. Most families of real nodes τ_i are very bad. Some families of complex nodes, however, are not: Let N be fixed and put $\tau_n = \exp\left(2\pi i n/N\right)$ for $0 \le n \le N-1$. The condition number for this change-of-basis matrix (interpolation of Nth roots of unity) is 1. Prove this. [In the infinity norm, $\|\mathbf{A}_{\phi,x^m}\|_\infty = n$.]

8.37. The *confluent Vandermonde matrix* for the monomial basis $\phi_j(z) := z^j$ evaluated on the nodes $[\tau_0, \tau_0, \tau_1, \tau_1, \tau_1, \tau_2, \tau_2]$, where $\tau_i = \tau_j \Leftrightarrow i = j$, is

$$\mathbf{V} = \begin{bmatrix} 1 & \tau_0 & {\tau_0}^2 & {\tau_0}^3 & {\tau_0}^4 & {\tau_0}^5 & {\tau_0}^6 \\ 0 & 1 & 2\tau_0 & 3{\tau_0}^2 & 4{\tau_0}^3 & 5{\tau_0}^4 & 6{\tau_0}^5 \\ 1 & \tau_1 & {\tau_1}^2 & {\tau_1}^3 & {\tau_1}^4 & {\tau_1}^5 & {\tau_1}^6 \\ 0 & 1 & 2\tau_1 & 3{\tau_1}^2 & 4{\tau_1}^3 & 5{\tau_1}^4 & 6{\tau_1}^5 \\ 0 & 0 & 2 & 6\tau_1 & 12{\tau_1}^2 & 20{\tau_1}^3 & 30{\tau_1}^4 \\ 1 & \tau_2 & {\tau_2}^2 & {\tau_2}^3 & {\tau_2}^4 & {\tau_2}^5 & {\tau_2}^6 \\ 0 & 1 & 2\tau_2 & 3{\tau_2}^2 & 4{\tau_2}^3 & 5{\tau_2}^4 & 6{\tau_2}^5 \end{bmatrix} .$$

Each row of $\mathbf{V}$ corresponds to the evaluation of $\phi_j(\tau_i)$ or its derivatives. See `vander` and `gallery` in MATLAB, with attention to the `chebvand` option of the latter. Optionally, look at `index/Vandermonde` in MAPLE, with the `confluent=true` and the `basis=<orthogonal polynomial name>` options.

1. Show that finding the coefficients of the polynomial $p(z)$ of degree at most 6 that fits the given data $p(\tau_0) = y_0$, $p'(\tau_0) = y'_0$, $p(\tau_1) = y_1$, $p'(\tau_1) = y'_1$, $p''(\tau_1) = y''_1$, $p(\tau_2) = y_2$, and $p'(\tau_2) = y'_2$ can be done by solving a linear system $\mathbf{Vx} = \mathbf{y}$. The determinant of this system is known to be nonzero for distinct τ_i, as we have assumed.
2. (Nondimensionalization) Show that for this particular problem we may, without loss of generality, take $\tau_0 = 0$ and $\tau_2 = 1$.
3. Solution of interpolation problems [finding $p(z)$] by solving the confluent Vandermonde matrix directly suffers from two difficulties: Such matrices tend to be ill-conditioned, and the cost of factoring the matrix appears to be $O(n^3)$.

   a. Plot, on a logarithmic scale, the condition number of $\mathbf{V}$ against $r = \tau_1$, where $\tau_0 = 0$ and $\tau_2 = 1$, for $0.01 \le r \le 0.99$. Also plot the Skeel condition number cond($\mathbf{V}$), which is $\| \, |\mathbf{V}^{-1}| \, |\mathbf{V}| \, \|$.

   b. A faster *and more numerically stable* algorithm to solve this (Hermite) interpolation problem is to use the barycentric formulae. First, do a partial fraction expansion of

$$\begin{aligned} \frac{1}{w(z)} &= \frac{1}{z^2(z-r)^3(z-1)^2} \\ &= \frac{\beta_{00}}{z} + \frac{\beta_{01}}{z^2} + \frac{\beta_{10}}{z-r} + \frac{\beta_{11}}{(z-r)^2} + \frac{\beta_{12}}{(z-r)^3} + \frac{\beta_{20}}{z-1} + \frac{\beta_{21}}{(z-1)^2} , \end{aligned}$$

   which can be done stably in $O(n^2)$ work by local series computations, so that

$$\frac{1}{w(z)} = -\frac{1}{r^3 z^2} + \frac{-3-2r}{zr^4} + \frac{2r-5}{(r-1)^4(z-1)} + \frac{1}{(r-1)^2 r^2 (z-r)^3} + \frac{-4r+2}{(z-r)^2 r^3 (r-1)^3} - \frac{1}{(r-1)^3(z-1)^2} + \frac{10r^2-10r+3}{r^4(r-1)^4(z-r)},$$

and then, if the value and derivative data are $\rho_{i,j} = p^{(j)}(\tau_i)/j!$ for $1 \le i \le n = 3$, $0 \le j \le s_i - 1$, and the confluencies are $s_1 = 2$, $s_2 = 3$, and $s_3 = 2$ (local Taylor coefficient form),

$$\frac{p(z)}{w(z)} = \sum_{i=0}^{n} \sum_{j=0}^{s_i-1} \sum_{k=0}^{j} \beta_{i,j} \rho_{i,k} (z-\tau_i)^{k-j-1}.$$

Plot on a logarithmic scale the sum of the absolute values of the derivatives $\partial p/\partial \rho_{i,j}$ as a function of r and z, on $0.1 \le r \le 0.9$, $0 \le z \le 1$. Compare with your previous plot. You should notice that for $r \approx 1/2$, this vector norm is about 10^4 times smaller than the condition number reported previously, for all values of z in $[0,1]$. You may use MAPLE to help with the tedious algebra.

c. Now, consider this *subtle question*. Think carefully about your answer. First, why can that vector norm be interpreted as a "condition number" for the interpolation problem? Second, why is it smaller than the matrix condition number of Part 3a? The fact that it is so very much smaller is quite a surprise. We may draw the following conclusion: *The algorithm "solve the confluent Vandermonde matrix by LU factoring or QR factoring," for solving the Hermite interpolation problem, is numerically unstable.* Does it help to consider the Skeel condition number cond**V**? Why not?

8.38. The text establishes the (real) interpolation error formula

$$f(t) = p(t) + w(t)\frac{f^{(d+1)}(\hat{t})}{(d+1)!}$$

for some unknown $\hat{t}$. If we are free to choose the nodes τ_i, then we might try to choose them so as to make the error term $w(t)$ small [there's not much we can do about $f^{(d+1)}(\hat{t})/(d+1)!$ because this depends mostly on f—although we do get to choose d, as well].

1. Chebyshev polynomials have an interesting property: Over all monic polynomials, $T_{d+1}(t)/2^{d+1}$ has the smallest max-norm on $-1 \le t \le 1$:

$$2^{-d-1} = \max_{-1 \le t \le 1} \frac{|T_{d+1}(t)|}{2^{d+1}} \le \max_{-1 \le x \le 1} |P(t)|,$$

where $P(t)$ is *any* monic polynomial. So $w(t) = T_{d+1}(t)/2^{d+1}$ is a good choice. Show that this makes

$$\tau_k = \cos\left(\frac{\pi}{2(d+1)}(2k-1)\right) \qquad 1 \le k \le d+1\,.$$

This means that interpolation at Chebyshev nodes turns out to be a very good idea.

2. Equally spaced points, however, give surprisingly bad answers sometimes, because

$$w(z) = \prod_{k=1}^{n}\left(z-(-1+\frac{2k}{n+1})\right)$$

has some surprisingly large wiggles near the ends.

   a. Plot, for $n = 21$, $T_n(x)/2^n$ and

   $$\prod_{k=0}^{n}\left(x-(-1+\frac{2k}{n+1})\right)$$

   on $-1 \le x \le 1$. Comment.

   b. Interpolate $R(t) = 1/(1+25t^2)$ on $-1 \le t \le 1$ using, first, equally spaced points ($n = 21$) and, second, Chebyshev points. Plot your errors; also plot

   $$\frac{R(t)-f(t)}{w(t)}$$

   in each case. Is $R^{(n+1)}(\hat{t})/(d+1)!$ similar in each case? There's no reason the $\hat{t}$'s (which are functions of t) should be the same.

This last function $R(t)$ is the famous Runge example. It is often used to show that high-degree interpolation can be a bad idea. Here we see that it is the choice of equally spaced nodes that is really to blame.

3. We saw that Chebyshev nodes are "nice." But they are denser near the edges of the interval, and by Problem 8.11, this ought to cause ill-conditioning. What is true? Moreover, for $n = 21$, plot the condition number of the Lagrange basis at the Chebyshev nodes for the Runge function. Do the same for the equally spaced nodes. Discuss.

8.39. This investigation is about *the problem of monicity*. For the purpose of rootfinding, $p(z)$ and $p(z)/a$ are (in theory) equivalent for nonzero scalars a. By common convention, with the monomial basis one usually chooses $a = a_n$, where $p(z) = \sum_{k=0}^{n} a_k z^k$ has (exact) degree n. The resulting polynomial

$$\tilde{p}(z) = \frac{p(z)}{a_n} = z^n + \frac{a_{n-1}}{a_n}z^{n-1} + \ldots + \frac{a_0}{a_n} = z^n + \sum_{k=0}^{n-1}\tilde{a}_k z^k$$

is called *monic*. This can have some undesirable effects numerically if a_n is not zero but small: If we then look for a z^* such that $\tilde{p}(z^*) = 0$, this means

$$(z^*)^n + \sum_{k=0}^{n-1} \tilde{a}_k (z^*)^k = 0$$

and the $\tilde{a}_k = {}^{a_k}\!/\!_{a_n}$ are large—this means that there *must* be catastrophic cancellation in evaluating $p(z^*)$. But there are sometimes better choices for normalization.

1. Show that

$$\|\mathbf{a}\|_2 := \sum_{k=0}^{n} |a_k|^2$$

(the vector 2-norm of the vector of coefficients) is related to the size of $p(z)$ on the unit disk by

$$\begin{aligned}\|\mathbf{a}\|_2^2 &= \frac{1}{2\pi}\int_0^{2\pi} p(e^{i\theta})\overline{p}(e^{-i\theta})d\theta \\ &= \frac{1}{2\pi i}\oint_C p(z)\overline{p}(z^{-1})\frac{dz}{z},\end{aligned}$$

where C is the unit circle.
2. Let $\mathbf{A}_n$ be a random $n \times n$ matrix, with entries chosen uniformly on the interval $[0,1]$. For large n, say $n \approx 300$, the coefficients a_k of charpoly(A) [poly(A) in MATLAB] are large. Plot them (use an example) on a log scale. Discuss.
3. Another common normalization is to take a random number a and divide by it; this approach is more common in a vector context where one chooses a random unit vector $\mathbf{r}$ and then takes $a = \mathbf{r}\cdot\mathbf{a}$, so that $\mathbf{r}\cdot({}^{\mathbf{a}}\!/\!_{a}) = 1$. Discuss.

8.40. The object of this investigation is PQHIP, that is, piecewise-quintic Hermite interpolation. Compare pchip in MATLAB (or pchiptx in Moler (2004)). If, instead of just data τ_i and $p(\tau_i)$, we also know $p'(\tau_i)$ for $1 \le i \le n$, then we are tempted to try to fit piecewise-*quintic* Hermite interpolants. Choose $\rho_{i2} := {}^{p''(\tau_i)}\!/\!_2$ in such a way as to ensure that the curvatures of adjacent intervals match, in a harmonic mean sense, as follows. Set

$$\eta_{i-1} = \frac{\rho_{i,1} - \rho_{i-1,1}}{\tau_i - \tau_{i-1}} \approx 2\overline{\rho_{i-1,2}}$$

and

$$\eta_i = \frac{\rho_{i+1,1} - \rho_{i,1}}{\tau_{i+1} - \tau_i} \approx 2\overline{\rho_{i,2}}$$

and, exactly as in PCHIP, set $\rho_{i,2} = 0$ if η_i and η_{i-1} are of opposite signs and otherwise use the weighted harmonic mean

$$\frac{w_1 + w_2}{2\rho_{i,2}} = \frac{w_1}{\eta_{i-1}} + \frac{w_2}{\eta_i},$$

with $w_1 = 2(\tau_{i+1} - \tau_i) + (\tau_i - \tau_{i-1})$ and $w_2 = (\tau_{i+1} - \tau_i) + 2(\tau_i - \tau_{i-1})$. The endpoints $\rho_{1,2}$ and $\rho_{n,2}$ should be dealt with as d_1 and d_2 are in `pchiptx` (see `pchipend`, p. 16).

1. Write a MATLAB routine `pqhip.m` that implements these ideas. The header should be

```
function v = pqhip( tau, y, dy, t )
```

   where `t` represents (possibly) a vector of input points at which to evaluate the interpolant, and `PQHIP` should return a vector **v** with `v(k) = p(t(k))`, where $p(t)$ is the piecewise-quintic interpolant to the data; that is, `p(tau(i)) = y(i)` and $p'(\tau_i) =$ `dy(i)`, mixing mathematical and MATLAB notations, and the second derivatives have been chosen as discussed above. Remember that on $\tau_{\ell-1} \le t \le \tau_\ell$,

$$p(t) = (t - \tau_{\ell-1})^3 (t - \tau_\ell)^3 \sum_{i=\ell-1}^{\ell} \sum_{j=0}^{2} \sum_{k=0}^{j} \beta_{i,j} \rho_{i,k} (t - \tau_i)^{k-j-1} ,$$

   and the $\beta_{i,j}$ can be found from the partial fraction expansion of $1/(z(z-1))^3$ by nondimensionalization with $t = \tau_{\ell-1} + hz$, where $h = \tau_\ell - \tau_{\ell-1}$. Alternatively, you may use the confluent Vandermonde matrix, because the instability doesn't matter if you solve the linear system exactly (and this system is small enough that you can do so).

   [The text Moler (2004), available online, comes with MATLAB programs distributed in a directory called NCM. We found it convenient to modify `pchiptx`, which you may find in the NCM directory. Note that there is a potential bug in PCHIP, the cubic version: The program assumes that the nodes are strictly increasing, and since this is occasionally violated (e.g., for the Chebyshev–Lobatto nodes), the nodes must first be in increasing order. This can be fixed by inserting a "sort" command into PQHIP.]
2. Test your code on the Runge example: $f(t) = 1/(1+25t^2)$, sampled at $n = 10, 20, 30, 50$, and 80 equally spaced points on the interval $-1 \le t \le 1$. Plot the relative error $p(t)/f(t) - 1$ for each of these five interpolants.
3. Test your code on an example of your own choice.

8.41. In Wilkinson (1959b), we find a polynomial rootfinding problem that is interesting to attack using Lagrange interpolation. The discussion there begins, "As a second example, we give a polynomial expression which arose in filter design. The zeros were required of the function $f(x)$ defined by"

$$f(z) = \prod_{i=1}^{7} \left(z^2 + A_i z + B_i\right) - k \prod_{i=1}^{6} (z + C_i)^2 , \tag{8.64}$$

with the data values as given below:

$$\mathbf{A} = \begin{bmatrix} 2.008402247 \\ 1.974225110 \\ 1.872661356 \\ 1.714140938 \\ 1.583160527 \\ 1.512571776 \\ 1.485030592 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 1.008426206 \\ 0.9749050168 \\ 0.8791058345 \\ 0.7375810928 \\ 0.6279419845 \\ 0.5722302977 \\ 0.5513324340 \end{bmatrix} \qquad \mathbf{C} = \begin{bmatrix} 0 \\ 0.7015884551 \\ 0.6711668301 \\ 0.5892018711 \\ 1.084755941 \\ 1.032359024 \end{bmatrix}$$

and $k = 1.380 \times 10^{-8}$. Wilkinson claims that this polynomial is very ill-conditioned when expanded into the monomial basis centered at 0: "The explicit polynomial $f(x)$ is so ill-conditioned that the double precision Bairstow programme gave only 2 correct figures in several of the factors and the use of treble precision section was essential." He later observes that if $f(z)$ (we use z here not x as Wilkinson did[7]) is expanded into the shifted monomial basis centred at $z = -0.85$, it's not so badly conditioned.

1. Expand the polynomial in MAPLE or another CAS, thus expressing $f(z)$ in the usual monomial basis. Show that the absolute condition number for evaluation $B(z)$ near $z = -0.85$ is of the order of 10^4 at most, while $f(z)$ has magnitude near 10^{-13} in a disk of radius nearly 0.02 surrounding that point.
2. Compute the relative condition numbers—the *structured* condition numbers, that is—$A_i \cdot (\partial f/\partial A_i)/f$, $B_i \cdot (\partial f/\partial B_i)/f$, $C_i \cdot (\partial f/\partial C_i)/f$ (except for $C_1 = 0$), and $k \cdot (\partial f/\partial k)/f$, and show that these have magnitude about 10^7 near $z = -0.85$.
3. Since $k = 1.380 \times 10^{-8}$ is so small it seems natural to expect that the zeros of $f(z)$ will be near the zeros of the quadratic factors $z^2 + A_i z + B_i$ of the first term, plus an $O(k)$ correction. (This turns out to be true but not as accurate as one might hope, so numerics must in the end be used to get the roots accurately.) This idea suggests though that using the quadratic formula seven times to find those zeros and to use those zeros plus one more point, say the point $z = 0$ at which the second factor is zero, to provide a Lagrange interpolant of $f(z)$, would provide a fairly well-conditioned expression for evaluation and zerofinding. Do so, then construct the Lagrange companion matrix pencil, and find the roots as accurately as you can in MATLAB. For example, the smallest root pair is given by Wilkinson as $-7.42884803195285 \times 10^{-1} \pm 1.32638868266 \times 10^{-4} i$, and using the above idea, we get within 5×10^{-14} of this.
4. Show that using k near $1.38627474705019677767 \times 10^{-8}$ instead (which is a change of less than half a percent in k) makes two of the roots equal. Perhaps by drawing a set of pseudozeros, argue that this polynomial is indeed ill-conditioned.

8.42. Suppose integers m and n are given, together with $N+1$ nodes and values (τ_k, f_k) for $0 \le k \le N$, and that $n + m = N$. The problem is to find ρ_k and σ_k such that $p(\tau_k) = \rho_k$, $q(\tau_k) = \sigma_k$, the degree of p is m, the degree of q is n, and the rational function $p(z)/q(z)$ interpolates the given data. This is similar to the rational

[7] Some people might confuse this problem with the famous Wilkinson polynomial $W(z) = (z-1)(z-2)\cdots(z-20)$. Don't. This problem has *nothing* to do with that polynomial.

interpolation problem that we have already studied, except that now the only thing known about $q(z)$ is its degree (and the degree of the numerator is also constrained). A complete solution to this problem can be found in Pachón et al. (2012), but solve it yourself before you look there, by showing that the unknown ρ_k and σ_k are the entries in a $2(N+1)$-vector satisfying

$$\begin{bmatrix} \mathbf{I} & -\mathrm{diag}(\mathbf{f}) \\ \mathbf{V}_{N-m}\mathrm{diag}(\boldsymbol{\beta}) & 0 \\ 0 & \mathbf{V}_{N-n}\mathrm{diag}(\boldsymbol{\beta}) \end{bmatrix} \begin{bmatrix} \boldsymbol{\rho} \\ \boldsymbol{\sigma} \end{bmatrix} = 0\,, \tag{8.65}$$

where $\mathbf{V}_\ell$ is a suitable truncation of a Vandermonde matrix on the nodes τ_k. The matrix is $2N+1$ by $2N+2$ and so has at least one vector in its null space. Try this out on some examples, and show that you can (at least sometimes) solve the Cauchy problem for rational interpolation (as it is known) in this fashion.

Chapter 9
The Discrete Fourier Transform

Abstract This short chapter introduces the discrete (finite) and *fast Fourier transform*. The numerical stability and *conditioning* of the *Fourier matrix* is mentioned. Applications using *convolution* and *circulant matrices* are considered, as is the *periodogram* or *power spectral density*. ◁

One of the most important tools in applied mathematics is Fourier series. Strang 1986, p. 263 begins his discussion with a question:

> How do we introduce a subject as important as Fourier series? The cowardly way is tremendously tempting—to choose a typical function and expand in a sum of sines and cosines.

Chapter 4 of Strang's book then goes on to provide an excellent (even heroic!) introduction to Fourier series, the continuous Fourier transform, and the discrete Fourier transform. In Chap. 5 of that book, we also find an excellent introduction to the fast Fourier transform, or FFT. Here, we take a somewhat cowardly approach in that we start with polynomials. But again, we're going to do things a little differently, based on the interpolation tools introduced in the previous chapter.

9.1 The Fourier Transform via Interpolation

Consider interpolating the vector $\boldsymbol{\rho}$ of complex data on nodes $\tau_k = e^{2\pi i k/(n+1)}$ for $0 \le k \le n$. These τ_k are the $(n+1)$st roots of unity: $\tau_k^{n+1} = 1$. We'll proceed as in the previous chapter; the barycentric weights turn out to be analytically available with the short formula $\beta_k = 1/w'(\tau_k) = \tau_k/(n+1)$. Writing as usual

$$w(z) = \prod_{k=0}^{n}(z-\tau_k) = z^{n+1} - 1\,, \tag{9.1}$$

we have the partial fraction decomposition

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_9,

$$\frac{1}{w(z)} = \frac{1}{n+1}\sum_{k=0}^{n}\frac{\tau_k}{z-\tau_k}\,. \tag{9.2}$$

The polynomial interpolant is thus seen to be

$$\begin{aligned} f(z) &= (z^{n+1}-1)\sum_{k=0}^{n}\frac{\left(\frac{\tau_k}{n+1}\right)\rho_k}{z-\tau_k} \\ &= \frac{1}{n+1}(z^{n+1}-1)\sum_{k=0}^{n}\frac{\tau_k\rho_k}{z-\tau_k}\,. \end{aligned} \tag{9.3}$$

Now we are going to do something that, for other sets of interpolation nodes, we have explicitly deprecated: We're going to change to the monomial basis, so that we will have

$$f(z) = c_0 + c_1 z + c_2 z^2 + \cdots c_n z^n\,. \tag{9.4}$$

We may say immediately what c_0 is, by evaluating (9.3) at $z=0$:

$$f(0) = c_0 = \frac{1}{n+1}\sum_{k=0}^{n}\rho_k\,. \tag{9.5}$$

By looking at the leading term, we can also see almost immediately that

$$c_n = \frac{1}{n+1}\sum_{k=0}^{n}\tau_k\rho_k\,. \tag{9.6}$$

A little more effort gets us c_{n-1}, which turns out to be

$$c_{n-1} = \frac{1}{n+1}\sum_{k=0}^{n}\tau_k^2\rho_k\,. \tag{9.7}$$

To see this, first note that the coefficient of z^{n-1} is the negative sum of all the τ_j except τ_k, and by subtracting τ_k itself, we get

$$\tau_k + \sum_{j=0}^{n} e^{2\pi ij/(n+1)} = \tau_k\,. \tag{9.8}$$

To see that a bit more clearly and to get the rest of the terms, we distribute the $z^{n+1}-1$ over the sum in Eq. (9.3) to find

$$p(z) = \frac{1}{n+1}\sum_{k=0}^{n}\tau_k\rho_k\prod_{\substack{j=0\\ j\neq k}}^{n}(z-\tau_j) = \frac{1}{n+1}\sum_{k=0}^{n}\tau_k\rho_k p_k(z)\,. \tag{9.9}$$

Having the three computed terms as clues, we suspect that

$$p_k(z) := \prod_{\substack{j=0\\ j\neq k}}^{n} (z-\tau_j) = \tau_k^n + \tau_k^{n-1} z + \cdots + \tau_k z^{n-1} + z^n\,, \tag{9.10}$$

and, in fact, this is easy to prove:

$$p_k(\tau_j) = \tau_k^n \sum_{\ell=0}^{n} \left(\frac{\tau_j}{\tau_k}\right)^{\ell}\,, \tag{9.11}$$

which, being a geometric series with ratio $r = \exp(2\pi i(j-k)/(n+1))$, is just 0 if $j \neq k$ and is $n+1$ if $j=k$. This proves that the n zeros of $p_k(z)$ are just the τ_j with $j \neq k$, and since $p_k(z)$ is monic, it must be the given polynomial. An alternative proof would be to use long division:

$$\begin{aligned} z^{n+1} - 1 &= z^{n+1} - \tau_k^{n+1} \\ &= (z-\tau_k)(z^n + z^{n-1}\tau_k + \cdots + \tau_k^{n-1})\,. \end{aligned} \tag{9.12}$$

This allows us to immediately write down all the coefficients c_j of the monomial basis form for $p(z)$: For $0 \le j \le n$,

$$c_j = \frac{1}{n+1} \sum_{k=0}^{n} \tau_k^{n+1-j} \rho_k = \frac{1}{n} \sum_{k=0}^{n} \tau_k^{-j} \rho_k. \tag{9.13}$$

This gives us an explicit formula for the conversion of Lagrange data on roots of unity to the monomial basis. This can be written as a matrix–vector product relating the vector of coefficients $\mathbf{c}$ with the vector of Lagrange coefficients $\boldsymbol{\rho}$, and we will do so shortly. We will say that $\mathbf{c}$ is the *inverse* FFT of $\boldsymbol{\rho}$.

One common alternative derivation of that previous result uses the Vandermonde matrix as discussed in Chap. 8, and then inverts it:

$$\mathbf{V}\mathbf{c} = \boldsymbol{\rho}\,, \tag{9.14}$$

where $\rho_k = c_0 + c_1\tau_k + \cdots + c_n\tau_k^n$. Here we say that $\boldsymbol{\rho}$ is the FFT of $\mathbf{c}$. Equation (9.13) gives us an explicit equation for the inverse of $\mathbf{V}$. For what follows, we will let

$$\omega := e^{2\pi i/(n+1)}\,. \tag{9.15}$$

Note that $\tau_1 = \omega$, and indeed $\tau_k = \omega^k$. For $n = 3$, the Vandermonde matrix and its inverse are related by

$$\begin{bmatrix} \rho_0 \\ \rho_1 \\ \rho_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega & \overline{\omega} \\ 1 & \overline{\omega} & \omega \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}\,, \tag{9.16}$$

because $\omega^2 = \overline{\omega}$ and $\overline{\omega}^2 = \omega$, since $(e^{4\pi i/3})^2 = e^{8\pi i/3} = e^{6\pi i/3 + 2\pi i/3} = e^{2\pi i/3} = \omega$. Equation (9.13) gives

$$\mathbf{V}^{-1} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega & \overline{\omega} \\ 1 & \overline{\omega} & \omega \end{bmatrix}^{-1} = \frac{1}{3}\begin{bmatrix} 1 & 1 & 1 \\ 1 & \overline{\omega} & \omega \\ 1 & \omega & \overline{\omega} \end{bmatrix} = \frac{1}{3}\overline{\mathbf{V}}\,. \tag{9.17}$$

Because $\omega^{jk} = \omega^{kj}$, the matrix $\mathbf{V}$ is *complex-symmetric*, $\mathbf{V} = \mathbf{V}^T$. Thus, its inverse is not just its conjugate but also its conjugate transpose, $\mathbf{V}^H$, divided by $n+1$. For $n = 4$ we find, with $\omega = e^{2\pi i/5}$ this time,

$$\mathbf{V} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 \\ 1 & \omega^2 & \omega^4 & \omega & \omega^3 \\ 1 & \omega^3 & \omega & \omega^4 & \omega^2 \\ 1 & \omega^4 & \omega^3 & \omega^2 & \omega \end{bmatrix}\,, \tag{9.18}$$

and, as stated previously, that its inverse is its complex conjugate transpose, divided by $n+1$:

$$\mathbf{V}_{n+1}^{-1} = \frac{1}{n+1}\mathbf{V}_{n+1}^H\,. \tag{9.19}$$

As we have seen, this is perfectly general. To find the monomial basis polynomial coefficients c_k of $f(z)$, we need only multiply the data vector $[\rho_0, \rho_1, \ldots, \rho_n]^T$ by $1/(n+1)\mathbf{V}_{n+1}^H$. That is, the monomial basis expression of the interpolating polynomial is as easy to find, almost as the barycentric representation.

We will use the letter $\mathbf{F} := \mathbf{V}$. This is the Fourier matrix, whose inverse we saw first in sum form, in Eq. (9.13). In MATLAB you can compute $\mathbf{F}$ by `fft(eye(n))`, although one will usually never need to see the matrix itself. Except for the factor $n+1$, the inverse $\mathbf{F}^{-1}$ =`ifft(eye(n))` is also a Vandermonde matrix itself, based on powers of $\overline{\omega}$ instead of $\omega = \exp(2\pi i/(n+1))$, because $\omega^n = \overline{\omega}$.

What about its conditioning? For $z \in \mathbb{T}$, that is, $|z| \leq 1$, the monomial basis is near-optimal for approximation; and the complex-symmetric matrix $\mathbf{F}_{n+1}$ (more properly, $\mathbf{F}_{n+1}/\sqrt{n+1}$) is also unitary, so data errors in the ρ_k are normwise about the same size in the vector c_k.

This is nice, but where's the Fourier series? What we have done so far is just polynomial interpolation, and in the conventional monomial basis at that. Write $z = e^{2\pi i t}$. If we think of real t, say $0 \leq t \leq 1$, then the roots of unity τ_k correspond to equally spaced values of t: $t_k = k/(n+1)$, $0 \leq k \leq n$. (One might think after the discussion of the previous chapters that since we are now using the monomial basis *and* equally spaced nodes, we will be doomed numerically, but the trigonometric nature of the interpolant will turn out to be crucial, and, in fact, everything is nearly perfectly conditioned.) In the monomial basis in z,

$$f(z) = \sum_{k=0}^{n} c_k z^k\,, \tag{9.20}$$

but in terms of t, this is

$$f(t) = \sum_{k=0}^{n} c_k e^{2\pi ikt} = \sum_{k=0}^{n} c_k(\cos 2\pi kt + i\sin 2\pi kt), \tag{9.21}$$

and suddenly we are in (discrete) Fourier space. The condition number for evaluation of this expression is studied in Problem 9.2.

Remark 9.1. As an aside, with $z = \exp(2\pi it)$, the first barycentric form is, in contrast,

$$f(t) = \frac{e^{2\pi i(n+1)t} - 1}{n+1} \sum_{k=0}^{n} \frac{e^{2\pi ik/(n+1)}\rho_k}{e^{2\pi it} - e^{2\pi ik/(n+1)}}, \tag{9.22}$$

which certainly looks very strange. It is true that this can be simplified via trigonometric identities to get a trigonometric interpolant. Put $\theta = \pi(t-t_k) = \pi(t - k/(n+1))$, and we obtain

$$\begin{aligned}
\frac{1}{n+1}\frac{z^{n+1}-1}{z-\tau_k} &= \frac{1}{n+1}\frac{e^{2i(n+1)\theta}-1}{e^{2i\theta}-1} = \frac{e^{in\theta}}{n+1}\frac{e^{i(n+1)\theta}-e^{-i(n+1)\theta}}{e^{i\theta}-e^{-i\theta}}\\
&= \frac{e^{in\theta}}{n+1}\frac{\sin(n+1)\theta}{\sin\theta}\\
&= \frac{\cos n\theta}{n+1}\frac{\sin(n+1)\theta}{\sin\theta} + i\frac{\sin n\theta}{n+1}\frac{\sin(n+1)\theta}{\sin\theta} = g_k(t),
\end{aligned}$$

and these $g_k(t)$ [remember $\theta = \pi(t-t_k)$] have the Lagrange property: $g_k(t_j) = \delta_k^j$. Of course, they are just the Lagrange polynomials evaluated with $z = \exp(2\pi it)$. For more on this, see, for instance, Strang (1986 462): They're used in spectral methods. ◁

We do not pursue that seemingly oddball form here, but return to the simpler monomial basis, now a finite Fourier series:

$$f(t) = \sum_{k=0}^{n} c_k e^{2\pi ikt}, \tag{9.23}$$

which interpolates ρ_k at $t = k/(n+1)$.

Remark 9.2. There is a difficulty with this form, however, that demands immediate attention, namely, *aliasing*. Sampling induces its own kind of error, called *aliasing*: Frequencies that differ by the "right" amount cannot be told apart once they have been sampled. The bottom line is the Nyquist rate: We need at least two samples in every period to detect the component. For example,

$$\begin{matrix} 1 & 1 & 1 & 1 \\ \cos 0 & \cos 2\pi & \cos 4\pi & \cos 6\pi \end{matrix}$$

cannot be told apart. In the complex form above,

$$f(t) = \sum_{k=0}^{n} c_k e^{i2\pi kt} = c_0 + \sum_{k=1}^{n/2} c_k e^{i2\pi kt} + \sum_{k=n/2+1}^{n} c_k e^{i2\pi kt} \tag{9.24}$$

supposing n is even, and the second sum may be written

$$\sum_{\ell=1}^{n/2} c_{n+1-\ell} e^{i2\pi(n+1-\ell)t} \tag{9.25}$$

by $\ell + k = n+1$; and at the sample points, $2\pi(n+1)t = 2\pi j$, so this is indistinguishable from

$$f(t) = \sum_{k=-n/2}^{n/2} c_k e^{2\pi ikt} \tag{9.26}$$

with the definition $c_{-k} = c_{n+1-k}$. This second expression contains *lower* frequency terms than the original one, and this means that *this* interpolant is less wiggly. An example may make things clearer. ◁

Example 9.1. Suppose $n = 3$, and thus $\boldsymbol{\tau} = [0, 1/4, 1/2, 3/4]$. Suppose our data are $\boldsymbol{\rho} = [1, 1, -1, -1]$. Then the FFT gives the coefficients $c_0 = 0$, $c_1 = (1+i)/2$, $c_2 = 0$, and $c_3 = (1-i)/2$. The degree-3 polynomial in z gives the trigonometric interpolant

$$f(t) = c_0 + c_1 e^{2\pi it} + c_2 e^{4\pi it} + c_3 e^{6\pi it} . \tag{9.27}$$

Folding the frequencies as above—except that since now n is odd, we have to use what Henrici calls the *prime convention* and take only half the middle-frequency term to negative frequencies—we get

$$F(t) = \frac{1}{2} c_2 e^{-4\pi it} + c_3 e^{-2\pi it} + c_0 + c_1 e^{2\pi it} + \frac{1}{2} c_2 e^{4\pi it} . \tag{9.28}$$

This is a *different interpolant*: The differing exponentials are equal only at the nodes. See Fig. 9.1. See also `fftshift` in MATLAB. ◁

Differentiation and integration in this form are easy:

$$\frac{d}{dt} f(t) = \sum_{k=-n/2}^{n/2} 2\pi ikc_k e^{2\pi ikt} , \tag{9.29}$$

for example, if n is even. See Problem 9.8.

One may wish to artificially attenuate the high-frequency components (with large k) since a noisy signal tends to have higher frequencies than other signals; this is particularly useful when differentiating. Finally, the magnitudes of the coefficients $|c_k|$ are themselves interesting: Plotting them on a log scale is called a *periodogram*.

The coefficients c_k are called the discrete Fourier transform of the data ρ_k. It seems that we have found a pretty fast way to find them, too: The cost is $O(n^2)$,

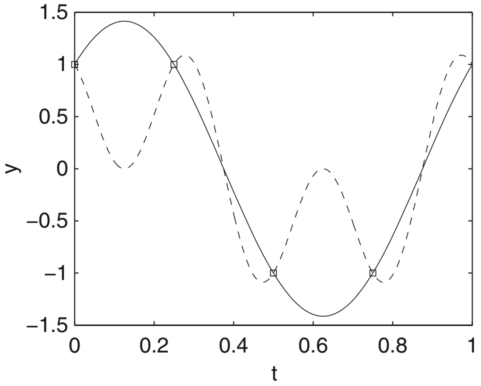


Fig. 9.1 Trigonometric interpolation before and after folding the frequencies at the Nyquist frequency. The lower-frequency interpolant (*solid line*) is less wiggly than the higher-frequency interpolant (*dashed line*)

not $O(n^3)$. Of course, the strange barycentric form costs just $O(n)$ because we have an analytic expression for the barycentric weights, but that form seems too strange to think with. However, fast as it may seem, $O(n^2)$ is not fast enough for many applications. We will shortly sketch a method to reduce the cost dramatically, to $O(n\log n)$, if $n+1$ is "highly composite," that is, has lots of small prime factors. But before we look at how the speed-up works, let's look at some applications.

Example 9.2. The first application is the direct use of periodogram to detect the frequencies present in a time series. Suppose data ρ_k are samples of data taken at times $t_k = k\Delta t$ in, say, seconds. Then, if we could write

$$y(t) = \sum_{k=0}^{n} c_k e^{2\pi i k t/T}, \tag{9.30}$$

where T was some nominal fundamental period, then (after folding) the largest $|c_k|$ would tell us which frequencies ${}^k/_T$ were most active. Let us assume we sample to $t_n = n\Delta t$, giving $n+1$ samples. Assume thereafter that the series repeats, and apply the DFT. For example, if we sample the numerical solution[1] of

$$y'' + 0.04\left(1 - \frac{y^2}{7}\right)y' + \pi^2 y = 0, \tag{9.31}$$

$y(0) = 1, y'(0) = 0$, at 1024 points in $0 \le t \le 102.3$ (part of which we see in Fig. 9.2), spaced at intervals $\delta t = 0.1$ s (i.e., 10 Hz), then taking the DFT of y by

```
Y=fft(y(:,1))/L
```

(here, $L = 1024$ is the length of the time series); $1024 = 2^{10}$ has lots of small factors, which is good as alluded to above we plot the periodogram (also known as power spectral density) in Fig. 9.3 by

[1] We will learn in Chap. 12 how to solve this equation.

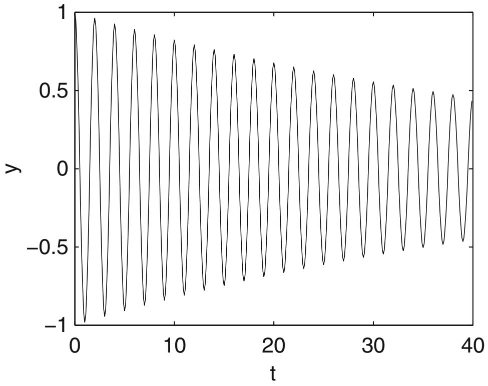


Fig. 9.2 A portion of the solution of the van der Pol equation over a few cycles. We take samples of this solution at equal time intervals

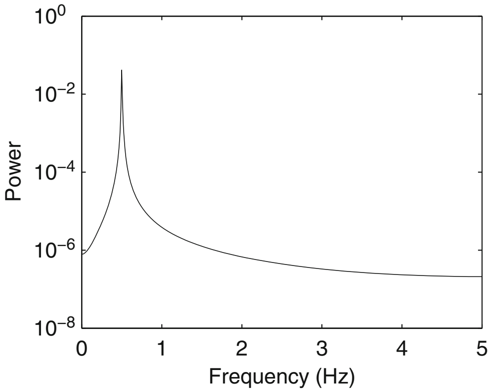


Fig. 9.3 A periodogram for samples of the solution to the van der Pol equation. The discrete values $|c_k|$ are plotted connecting the dots because there are so many that this seems reasonable, and the plot looks better. A clear peak is present at the natural frequency ($1/2$) of the oscillator

```
fre=Fs/2*linspace(0,1,513);
semilogy(fre,abs(Y(1:513)).^2,'k-')
```

with $Fs = 10$ Hz ($= 1/\Delta t$). We only use half the coefficients Y because of folding at the Nyquist frequency and that the data are real. Compare the first example in MATLAB's documentation for FFT. In the periodogram, we see a sharp peak near frequency $f = 1/2$. It's not exactly a spike, because the signal (y, the solution of Eq. (9.31)) is not a pure sinusoid; we see in Fig. 9.2 that it oscillates regularly but also decays. Still, there are 20 peaks in the first 40 s, giving f near $1/2$ Hz, as shown in the periodogram. Now if we add the forcing term $\sin(4\pi t)$ to the equation, so that we have

$$y'' + 0.04\left(1 - \frac{y^2}{7}\right)y' + \pi^2 y = \sin(4\pi t)\,, \tag{9.32}$$

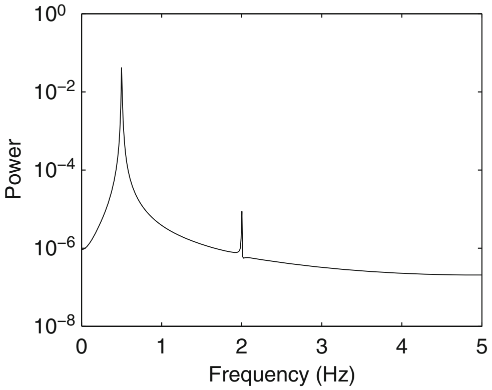


Fig. 9.4 The power spectral density of a forced van der Pol equation. The peak near $f = 2$ is easily seen

and sample the solution of this equation, we get the periodogram shown in Fig. 9.4. Note the clear new peak at $f = 2$; $\sin(2\pi f t)$ is easily detected.

We emphasize that these apparently smooth curves are simply plots of $|c_k|^2$ versus k, for discrete k; but we allow MATLAB to connect the dots because so many samples are taken that this is more like a continuous transform than a series. Thus, the (monomial basis!) coefficients c_k are themselves the desired answer, to one kind of question where we use the DFT.

To work out why we scaled by $^{Fs}/_{2}$, and took only half the resulting coefficients, is bookkeeping, albeit important bookkeeping.[2] Because of aliasing and because the data are real, the periodogram will be symmetric on $0 \le k \le n+1$ and we need only half of it. ◁

The second example of the use of the FFT is *convolution*. We have already seen Cauchy convolution in Taylor series generation, and we have briefly encountered circulant matrices; convolution as seen by the DFT is more general but includes Cauchy convolution as a special case. The convolution product of two sequences f_k, $0 \le k \le n$, and g_k, $0 \le k \le n$, is

$$c_k = \sum_{j+\ell=k \bmod n+1} f_j g_\ell \,, \qquad 0 \le \ell \le n \,,$$

that is, either $j + \ell = k$ or $j + \ell = k + n + 1$. To see that this generalizes Cauchy convolution, embed $a_0, a_1, \ldots, a_m$ and $b_0, b_1, b_2, \ldots, b_\ell$ in larger sequences by

[2] Because of the plethora or plenitude—perhaps even superfluity—of conventions and conventional notations for Fourier series and signal processing, the bookkeeping takes on more importance than usual. For example, what some people call the matrix **F** actually corresponds to what other authors or software packages, for example, MATLAB, mean by the *inverse* discrete Fourier transform. And then there is where to place the factor n, or $n+1$ if you index from zero, or $\sqrt{n+1}$. You get the idea.

appending zeros $a_{m+1} = a_{m+2} = \ldots = a_n = 0$, $b_{\ell+1} = b_{\ell+2} = \ldots = b_n = 0$. Then, for example,

$$c_0 = a_0 b_0 + \overbrace{a_1 b_n + a_2 b_{n-1} + \cdots + a_m b_{n+1-m}}^{\text{all zeros since trailing } b\text{'s } 0},$$

and we choose n large enough so that $b_{n+1-m} = 0$; that is, $n+1-m \geq \ell + 1$ or $n \geq m + \ell$. [This makes sense as we would need degree $z^{m+\ell}$ to represent the product $a(z)b(z) = c(z)$ accurately.] Similarly, $c_1 = a_1 b_0 + a_0 b_1 +$ sums of products $a_j b_\ell$ with $j + \ell = 1$ or $= 1 + n + 1$; if $j \geq 2$, ℓ must be $\geq n - j = m + \ell - j \geq \ell + 1$, and hence b_ℓ is zero.

What does this have to do with circulant matrices? As you will without doubt recall, a circulant matrix is determined by its first column. For $n = 3$,

$$\mathbf{C} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix}.$$

What is $\mathbf{C}[b_0, b_1, b_2, b_3]^T$?

$$\mathbf{C}\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_0 b_0 + a_3 b_1 + a_2 b_2 + a_1 b_3 \\ a_1 b_0 + a_0 b_1 + a_3 b_2 + a_2 b_3 \\ a_2 b_0 + a_1 b_1 + a_0 b_2 + a_3 b_3 \\ a_3 b_0 + a_2 b_1 + a_1 b_2 + a_0 b_3 \end{bmatrix}, \tag{9.33}$$

our convolution product. What does this have to do with the DFT?

Theorem 9.1. *Every circulant matrix*[3] $\mathbf{R}$ *of dimension* $n+1$ *is diagonalized by* $\mathbf{F}_{n+1}$*; moreover,*

$$\mathbf{R}\overline{\mathbf{F}}_{n+1} = \mathbf{F}_{n+1} \begin{bmatrix} a_0 & & & \\ & a_1 & & \\ & & \ddots & \\ & & & a_n \end{bmatrix}, \tag{9.34}$$

where the a_k *are the elements of the discrete Fourier transform of the first column of* $\mathbf{C}$.

Proof. See Davis (1994). ♮

Example 9.3. In the MATLAB session below, we see that the circulant matrix is, in fact, diagonalized both by $\mathbf{F}$ and by $\overline{\mathbf{F}}$, although the ordering of the diagonal is affected:

[3] Note that Matlab's circulant matrix convention is the transpose convention to the above.

```
%% Diagonalization of circulant matrices
R = gallery( 'circul', randn(3,1) )
F = fft( eye(3) )
norm( F*F'/3 - eye(3), inf )
D1 = F*R*F'/3
D2 = F'*R*F/3
norm( D1 - diag(diag(D1)), inf )
norm( D2 - diag(diag(D2)), inf )
norm( D1 - D2, inf )
```

The code creates a random circulant matrix and then diagonalizes it with a Fourier matrix, both ways. ◁

To compute the convolution product $a * b$, form the circulant matrix with first column **a**. Then the desired product is

$$\mathbf{c} = \mathbf{R}\mathbf{b}\,. \tag{9.35}$$

Hence, taking the DFT of both sides, we get

$$\mathbf{F}_{n+1}\mathbf{c} = \mathbf{F}_{n+1}\mathbf{R}\mathbf{b} = \mathbf{F}_{n+1}\mathbf{R}\frac{1}{n+1}\overline{\mathbf{F}}_{n+1}\mathbf{F}_{n+1}\mathbf{b} \tag{9.36}$$

or, if **C** is the DFT of **c** and **B** is the DFT of **b** and **A** is the DFT of **a** and the diagonal of the DFT of **R**,

$$\begin{bmatrix} C_0 \\ C_1 \\ \vdots \\ C_n \end{bmatrix} = \begin{bmatrix} A_0 & & & \\ & A_1 & & \\ & & \ddots & \\ & & & A_n \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ \vdots \\ B_n \end{bmatrix}; \tag{9.37}$$

that is,

$$C_k = A_k B_k\,. \tag{9.38}$$

Then the convolved vector **c** can be obtained from the DFT **C** by

$$\mathbf{c} = \frac{1}{n+1}\overline{\mathbf{F}}_{n+1}\mathbf{C} \tag{9.39}$$

(that is, the inverse discrete Fourier transform). This mathematical application is an abstract version of a very large variety of practical applications. Examples include correlation and autocorrelation, among others. In mechanics, it's the Duhamel integral; in optics, one uses this in diffraction. There are many more.

The MATLAB facilities for the DFT include the routines `fft`, `ifft`, `fft2`, `ifft2` (a two-dimensional version), and `conv` and `filter`. See the documentation.

Discrete? Fast? Which is it? The modern version of the FFT is quite different than the ideas of Gauss, or the hand techniques of Danielson and Lanczos. It's come a long way from the Cooley–Tukey algorithm, too. There are now several ideas in play. The first one, the main one, Gauss' idea, is essentially a clever factoring of $\mathbf{F}_{n+1}$, if $n+1$ can be factored over the integers. That is, if $n+1=(m+1)(\ell+1)$, then we can factor $\mathbf{F}_{n+1}$ into two matrices that are essentially recombinations of $\mathbf{F}_{m+1}$ and $\mathbf{F}_{\ell+1}$. The factorings depend heavily on the symmetries and redundancies among the $e^{2\pi ik/(n+1)}$. In the simplest case, if $n+1$ is even, one can perform two DFTs of half the size and with only a little extra work, combine them to get what was desired. Suppose $n+1=2^{\ell}$ is a pure power of 2. Then if the cost to perform an FFT of size $n+1$ is C_n, we have

$$C_n = 2C_{n/2} + \text{small change}. \tag{9.40}$$

Let's say $C_n = 2C_{n/2} + \mu n$, for some constant μ. Can we solve this recurrence relation? Phrase it in terms of $\ell : n = 2^{\ell}$. Then $C_{\ell} = 2C_{\ell-1} + \mu 2^{\ell}$. MAPLE solves this instantly giving $C_{\ell} = 2^{\ell}C_0 + \mu\ell 2^{\ell}$ (which we can verify). That is,

$$C_n = \mu(\log_2 n)n + O(n)\,. \tag{9.41}$$

This analysis shows that (if the decomposition can be done recursively) we save a huge amount. Even if we can't, just one step saves a lot:

$$C_{2N} = 2C_n + \mu N \tag{9.42}$$

and if $C_N = cN^2$, done by the direct method, then $C_{2N} = 2cN^2 + O(N)$, whereas the direct method on $C_{2N} = c(2N)^2 = 4cN^2$, costing twice as much.

Other important developments in FFT technology include a huge collection of advances in hardware (multiprocessor, multithreads, fast memory, all kinds of things) and is increasingly harder to keep up with. The SPIRAL project (`www.spiral.org`) tries to automatically generate FFT programs (and others) that take significant advantage of particular hardware configurations. Taken all together, their automatically tuned algorithms are very impressive; if you have the hardware, you ought to look at what they can do for you.

Accuracy issues are discussed in Higham (2002). We have already pointed out that since $\mathbf{F}_{n+1}/\sqrt{n+1}$ is unitary, the problem of computing $\mathbf{F}_{n+1}\mathbf{f}$ is well-conditioned. Are these fast algorithms stable? In short, the answer is "yes," provided a little care is taken with the trigonometric functions.[4] The FFT is a wonderful algorithm.

9.2 Chebfun and the FFT

The Chebfun package makes consistent use of the FFT to effect its change-of-basis from Lagrange interpolation on the Chebyshev–Lobatto points $\eta_k = \cos(\pi k/n)$, for $0 \le k \le n$, and the Chebyshev polynomial basis $T_j(x) = \cos(j\cos^{-1}x)$. Calling this

[4] Of course, it is only *normwise* well-conditioned. If you want a relatively small Y_k to be accurate, you'll have to work, and if there's data error, you may be out of luck.

change-of-basis matrix $\mathbf{T}$, Exercise 8.15 in Chap. 8 asked you to show that $\mathbf{T}$ was well-conditioned even for large n, with a 2-norm condition number asymptotically $\sqrt{2}$ as $n \to \infty$. This shows that changing bases in this way is not, a priori, a bad thing to do numerically (unlike changing to the monomial basis from any purely real set of interpolation points, which is exponentially ill-conditioned even from the Chebyshev–Lobatto points).

And as we will see shortly, by using the FFT, one can do it quickly as well, at a cost of $O(n\log n)$ flops. The key observation is that the entry $T_{j,k} = \cos(\pi jk/n)$ is just the real part of $\exp(2\pi ijk/2n)$, which is the j,k entry of the Fourier matrix of order $2n$. That is, $\mathbf{T}_n$ is the real part of a block of $\mathbf{F}_{2n}$. Thus, it's all over but the bookkeeping. To do that, consider the following lines of MATLAB code (which we will meet in Chap. 10 when we consider Clenshaw–Curtis quadrature), which we take from Trefethen (2008a):

```
x  = cos(pi*(0:n)'/n);
fx = feval(f,x)/(2*n);
g  = real(fft(fx([1:n+1 n:-1:2])));
a  = [g(1); g(2:n)+g(2*n:-1:n+2); g(n+1)];
```

The first line defines the η_k. The second line evaluates a function f at those points and divides by $2n$ (note there are $n+1$ points). The third line creates a vector of length $n+1+n-1=2n$ by copying the vector of f values (except the first one) *backwards*, takes the FFT of that vector, and takes the real part of the answer. The final line sorts that FFT out, computing the coefficient of $T_0(x)$ as g_1, the coefficient of T_n as g_{n+1}, and all the intermediate coefficients of $T_j(x)$ as $g_j + g_{2n-j+1}$. It is left as an exercise for the reader to verify that this procedure is correct (in MATLAB; of course, with other programs and FFT conventions, one has to do the bookkeeping again). One can speed this program up, and perform computations entirely over the reals (in which case, what we are doing is called the discrete cosine transform, or fast cosine transform); but one could hardly write a shorter program than the two lines above that do the work.

Example 9.4. When we try this out on, say, $f = \exp(x)$ with $n = 14$, we get

```
f = @(x) exp(x);
n = 14;
x   = cos(pi*(0:n)'/n);                     % Cheby points
%x = -chebpts( n+1 );
fx  = feval(f,x)/(2*n);                     % evaluate f
g   = real(fft(fx([1:n+1 n:-1:2])));        % FFT
a   = [g(1); g(2:n)+g(2*n:-1:n+2); g(n+1)]; % Cheby coeffs
format long e
a
%a =
%
%     1.266065877752008e+000
%     1.130318207984970e+000
%     2.714953395340766e-001
%     4.433684984866384e-002
%     5.474240442093672e-003
%     5.429263119140365e-004
%     4.497732295430186e-005
%     3.198436462376053e-006
%     1.992124805968780e-007
%     1.103677167604671e-008
%     5.505896164340052e-010
%     2.497962253711350e-011
%     1.039224262200378e-012
%     4.007905118896815e-014
%     1.443289932012704e-015
```

and the coefficients are very similar to those we saw in the Chebfun example in Chap. 2. ◁

Indeed, we are now in a position to explain the differences between the Chebyshev series coefficients in Eq. (2.118) and what is produced by Chebfun and similarly by the code above: The difference is again due to *aliasing*. As explained in (Trefethen 2013 Chap. 4), the polynomials $T_m(z)$, $T_{2n-m}(z)$, $T_{2n+m}(z)$, $T_{4n-m}(z)$. $\ldots$, take exactly the same values on the $(n+1)$-point Chebyshev grid. For instance, if $n = 7$ and $m = 3$, then at $z = \tau_2 = \cos({}^{\pi\cdot 2}/_{7}) = 0.9009688679\ldots$, all of $T_3(z)$, $T_{2\cdot 7-3}(z) = T_{11}(z)$ and $T_{2\cdot 7+3}(z)$ (and infinitely many others) take the value $0.22252093\ldots$. Thus, sampling as done above or by Chebfun must lump contributions from higher-degree terms in the series into the same bin that certain lower-degree terms use. Because the series converges so quickly, the aliased terms are usually small enough that they have little effect.

9.3 Notes and References

There are a great many books on the FFT. A good general presentation is the book by Briggs and Henson (1995). We recommend Strang (1986) in particular for applied mathematicians. The older work Hamming (1973) contains an excellent nitty-gritty introduction that clarifies many issues, especially about the Nyquist frequency. Van Loan (1992) is also a useful resource. For a thorough and entertaining view from the analytic side, consult Körner (1989). Circulant matrices are treated by Davis (1994) and more on them can be found in Horn and Johnson (1990). See Kahaner et al. (1989) for a few more details on applications of convolution, including Duhamel integrals.

Moler (2004) (more properly, the code repository for that book) contains several programs for interaction with the FFT. We particularly recommend `fftgui` and `fftmatrix`. Finally, the use of the FFT in the Chebfun package is described in Battles and Trefethen (2004).

Problems

9.1. Show that the matrix $\mathbf{F}_3$ of our first example can be obtained in MATLAB by asking for `3*ifft(eye(3))`.

9.2. Show that the analysis of the condition number for evaluation of $f(z) = \sum_{k=0}^{n} c_k \phi_k(z)$, namely, the computation of $B(z) = \sum_{k=0}^{n} |c_k||\phi_k(z)|$, applies directly if the $\phi_k(z)$ are trigonometric polynomials $\sin jz$ or $\cos \ell z$. For real z, then, deduce that the condition number for evaluation of a trigonometric polynomial or Fourier series is bounded by $\sum_{k=0}^{n} |c_k|$.

9.3. In MATLAB issue the command `doc fft` and look at the example therein, with identification of the intended frequencies in a noisy signal. Modify the example first by plotting the power spectral density (periodogram) on a semilogy scale. Then rerun the code several times, changing the amplitude of the noise, making it both larger on some runs and smaller on others. Can the signal frequencies be reliably identified with a much higher noise level than that in the documentation (that is, much higher than 2)?

9.4. Using code similar to that of Problem 9.3, but this time replacing the random noise with the logistic iteration $L_{n+1} = rL_n(1 - L_n)$ for the value $r = 4$, which is guaranteed to produce a chaotic signal starting with $L_0 =$ some random number in $(0, 1)$, use the FFT to identify the frequencies of the nonchaotic part of the signal. Is the FFT able to identify the signal for higher amplitudes of logistic chaos than it is for random noise as used in previous signal?

9.5. Write (or find on the Internet and test) a MATLAB routine to trigonometrically interpolate equispaced data using the FFT.

9.6. Write (or find on the internet and test) a MATLAB routine to trigonometrically interpolate *two-dimensional* data (on a tensor product grid, equispaced in both directions) using the two-dimensional FFT (see `doc fft2` in MATLAB).

9.7. Go to `http://www.spiral.net/` and follow the link there to Püschel et al. (2011), which is an overview of the SPIRAL project. This gives a description of an interesting approach to providing high-performance, architecture-tuned code for digital signal processing (and possibly better than doing it yourself).

9.8. Suppose you have a polynomial $f(z)$ given by values ρ_k on nodes $\tau_k = \exp(2\pi ik/(n+1))$ for $0 \le k \le n$. Write a short program in MATLAB that uses the FFT and the inverse FFT to accept as input the vector $\boldsymbol{\rho}_{k,0}$ and return the vector $\boldsymbol{\rho}_{k,1}$, where $\rho_{k,1} = f'(\tau_k)$; that is, your routine should return the vector of derivatives of the polynomial $f(z)$ at the roots of unity. Test your routine. This in effect uses a differentiation matrix such as we will meet in Chap. 11, although no matrix should be formed explicitly.

9.9. The following algorithm for solving confluent Vandermonde systems suggests itself: Interpret the Vandermonde system as a problem in conversion from a Hermite interpolational basis to the monomial basis. Simply compute the generalized barycentric weights $\beta_{i,j}$ from the nodes τ_i as usual, which costs $O(d^2)$ operations. Evaluate the interpolant at the $d+1$ roots of unity $\exp(2\pi ij/(d+1))$ for $0 \le j \le d$. This costs $O(d^2)$ operations. Interpret that as the DFT of a polynomial. Use the inverse FFT to compute the coefficients of that polynomial. At worst, this costs $O(d^2)$

operations, and if $d+1$ is highly composite, then the cost will be $O(d\log d)$. This algorithm therefore has the same asymptotic complexity as Algorithms 22.2 and 22.3 of Higham (2002), up to constant factors. Work out the details of this algorithm and implement it and test it. Show that it can be numerically stable, but that it can often be terrible. In particular, for equally spaced nodes τ_i on the interval $[-1,1]$ for modestly large n, say $n=25$ or so, the accuracy of the results can be poor. Use the condition number of evaluation of polynomials to predict the size of the errors in $p(\exp({}^{2\pi ik}/_{(d+1)}))$, and show that the difficulty is not the fault of the Fourier transform. Compare with the condition number $\kappa(\mathbf{V})$ of the generalized Vandermonde matrix.

9.10. Do the same as in Problem 9.9, but using Chebyshev–Lobatto points η_k.

9.11. If n is even, then the trigonometric interpolant after folding at the Nyquist frequency is

$$F(z) = c_0 + \sum_{k=1}^{n/2} c_k z^k + \sum_{\ell=1}^{n/2} c_{-\ell} z^{-\ell}\,,$$

where $c_{-\ell}$ is defined to be $c_{n+1-\ell}$ from the FFT. If n is odd, then the interpolant is

$$F(z) = c_0 + \sum_{k=1}^{(n-1)/2} c_k z^k + \sum_{\ell=1}^{(n-1)/2} c_{-\ell} z^{-\ell} + \frac{1}{2} c_{(n+1)/2} z^{(n+1)/2} + \frac{1}{2} c_{-(n+1)/2} z^{-(n+1)/2},$$

and $n+1-{}^{(n+1)}/_2 = {}^{(n+1)}/_2$ shows that the bookkeeping is ok for $c_{n+1-\ell} = c_{-\ell}$.

These are *rational functions* in z, with known denominators $z^{n/2}$ if n is even and $z^{(n+1)/2}$ if n is odd. Compute the partial fraction decompositions of $z^{n/2}/(z^{n+1}-1)$ if n is even and $z^{(n+1)/2}/(z^{n+1}-1)$ if n is odd and thereby write the second barycentric forms for

$$F(z) = \frac{\displaystyle\sum_{k=0}^{n} {}^{\alpha_k \rho_k}/_{(z-\tau_k)}}{\displaystyle\sum_{k=0}^{n} {}^{\alpha_k}/_{(z-\tau_k)}}\,.$$

That is, find formulæ for the α_k explicitly. This gives an expression for the trigonometric interpolant that costs only $O(n)$ flops to construct. Implement your formulæ and test them. What can't you do with these formulæ at a cost less than $O(n^2)$ that you can do with the c_k from the FFT?

9.12. Write the complex Fourier coefficients as $c_k = a_k + ib_k$ and thus write the complex discrete Fourier expansion (9.26) in real form. Do the same for the case when n is odd, and the resulting highest-frequency terms are multiplied by ${}^1/_2$ in order to maintain symmetry.

Chapter 10
Numerical Integration

Abstract We show that in an absolute sense, *integration is well-conditioned* but that in a relative sense, *integration can be ill-conditioned*. We interpret standard algorithms as *finding the exact integrals of nearby functions*. We discuss several methods, including *adaptive* methods, *Gaussian quadrature*, and methods for *oscillatory integrands*. We pay some attention to the issue of *singularity*. ◁

Numerical integration (also known as *quadrature*[1]) consists in using numerical methods to approximate the value of a definite integral:

$$I = \int_a^b f(x)dx\,. \tag{10.1}$$

The interval (a,b) is usually finite, but is sometimes infinite. We begin our discussion by assuming that $f(x)$ is at least continuous and may be smoother, perhaps infinitely differentiable or even analytic. Discontinuities introduce difficulties, some of which we take up in Sect. 10.6. There exist methods (of *great* practical interest) to evaluate definite integrals of higher dimensions numerically, but we will focus on the one-dimensional case. Accordingly, this chapter will present some basic quadrature rules and their respective error analyses.

Here are some of the reasons for which you would want to do numerical integration:

1. Perhaps f is known only at a limited number of points [such as in data gathering, where we only know the function through a set of points $(x_i, f(x_i))$].
2. It might be very difficult or impossible to find a formula for an exact antiderivative to f. Indeed, the majority of definite integration problems fall into this category. For example, if $f(x) = x\tan x$, then its antiderivative contains a dilogarithm and is not expressible in terms of elementary functions. (This particular integral

[1] From the empirical method of drawing small squares underneath the curve in question, and counting them to estimate the area.

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_10,

may not be an issue if you have access to a numerical routine for evaluating dilogarithm functions over the complex plane, but trust us, there are plenty of other examples.)

3. The evaluation of the given analytical expression for $F(x)$ may be less numerically stable than the quadrature of $f(x)$. We will see an example of this later, where the analytical expression needs thousands of digits (temporarily, because they cancel).
4. Even if a stable analytical expression for $F(x)$ is known, it may be expensive to evaluate numerically. In this case, a numerical quadrature may be much more cost-effective. One common situation for this is if $F(x)$ needs to be evaluated at a long series of nearby points x_k, a situation practically tailor-made for quadrature.

The basic idea of numerical quadrature is to *replace* $f(x)$ with a slightly different function, call it $f(x) + \Delta f(x)$ or $(f + \Delta f)(x)$, and integrate the second function instead. This is our *engineered problem* (see Chap. 1). We will choose Δf so that it's not too large, and so that $f + \Delta f$ is simple to integrate exactly. That this idea handles all cases above will be seen in the examples that follow.

In fact, the simplest execution of this idea can be grasped from the usual first-year explanation of definite integrals (see Fig. 10.1). From this point of view, we

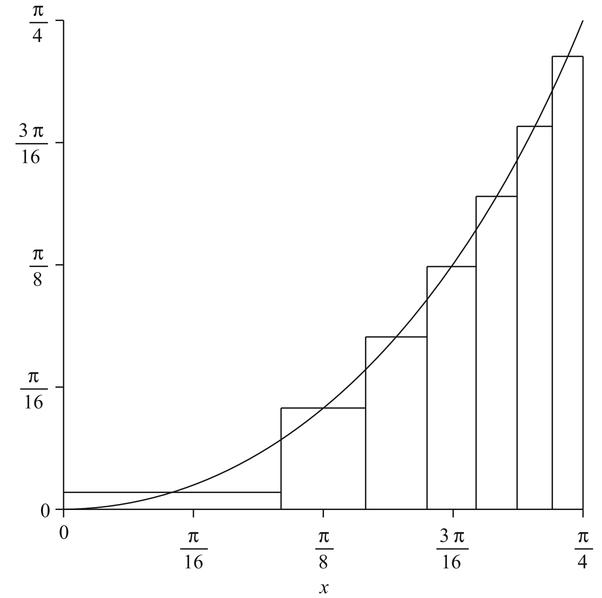


Fig. 10.1 An approximation to a definite integral, $\int_0^{\pi/4} x \tan x \, dx$, by a finite Riemann sum

evaluate a definite integral by replacing the curve using a finite number of horizontal lines—that is, a piecewise constant approximation. Since piecewise constant functions have analytically known areas, this satisfies our requirements. One could instead use a piecewise linear approximation, giving trapezoids.[2] The figures' heights are evaluated by means of the integrand $f(x)$, for example, at the middle point of each rectangle, as in the figure, thereby providing us with the width and height of each rectangle. We can then find the area of each rectangle, add them up, and obtain an approximate value of the definite integral. In a first-year calculus class, there is more interest in the process of taking the limit as the partition is refined (letting the maximum rectangle width go to zero) and in making a connection to antiderivatives, which works for the majority of the integrands encountered in the benign environment of a calculus class. Even if the antiderivative is not elementary, though, we obtain the definition of the "exact" integral in that limit. For example, with an equally spaced partition $x_k = a + (b-a)^k/_n$, giving widths $\Delta x = {}^{(b-a)}/_n$, we have the limit

$$\lim_{n\to\infty} \sum_{k=1}^{n} f(x_k)\Delta x = \int_a^b f(x)dx\,. \tag{10.2}$$

The points where the function is evaluated are called *nodes* (or integration points). The widths can be thought of as the weights attributed to the value of the function, and the sum is then an average. In general, the weights appearing in the weighted sum can be different, depending on what quadrature rule we are using. The resulting accuracy will be determined by the method for the selection of integration points, which then determines the weights.

Suppose that the interval $[a,b]$ is finite and has been partitioned as $a = x_0 < x_1 < x_2 \cdots < x_{n-1} < x_n = b$. Then we can give three simple Riemann-sum–based methods, which we assume that the reader remembers from first-year calculus—the left-hand Riemann sum, the right-hand Riemann sum, and the midpoint rule:

$$L_n = \sum_{k=0}^{n-1} f(x_k)\,(x_{k+1} - x_k) \tag{10.3}$$

$$R_n = \sum_{k=0}^{n-1} f(x_{k+1})\,(x_{k+1} - x_k) \tag{10.4}$$

$$M_n = \sum_{k=0}^{n-1} f(\frac{1}{2}(x_k + x_{k+1}))\,(x_{k+1} - x_k)\,. \tag{10.5}$$

[2] For an interesting discussion in the literature on numerical quadrature, try searching the web for information on "Tai's model," which should lead you to a highly cited paper in the journal *Diabetes Care*. The author, seemingly unaware of standard numerical methods (although one of the commenting articles points out the relevant one), reinvents the trapezoidal rule and applies it to problems relevant to the interests of the readers of the journal. It might seem humorous that someone could think reinventing a method that is at least hundreds of years old was publishable, but it seems less funny when you realize that she was correct, and count the number of citations. And, in the end, it's a useful lesson for students to learn that numerical computation of definite integrals is needed in diabetic care. It is a chastening lesson for educators to learn that numerical methods are taught so poorly in first-year calculus that this paper was needed, as evidenced by the number of citations of this paper.

We also give the trapezoidal (or trapezium) rule:

$$T_n = \sum_{k=0}^{n-1} \frac{1}{2} \left(f(x_k) + f(x_{k+1}) \right) (x_{k+1} - x_k) \; . \tag{10.6}$$

Alternatively, $T_n = (L_n + R_n)/2$ is the arithmetic mean of the left- and right-hand Riemann sums.

If f is monotonically increasing, and is continuous, then we obtain the inequality

$$L_n < \int_a^b f(x)dx < R_n. \tag{10.7}$$

Moreover, if the partition is equally spaced, so that $x_{k+1} - x_k = {}^{(b-a)}/_{n}$, then $R_n - L_n = (f(b) - f(a)) \cdot (b-a)/n \to 0$ and is $O({}^{1}/_{n})$ as $n \to \infty$. But the midpoint rule is better: If f is concave down, that is $f(x)$ lies below its tangent lines, then the ith piece of the trapezoidal rule and the midpoint rule are related by the inequality

$$T_i < \int_{x_i}^{x_{i+1}} f(x) < M_i \, , \tag{10.8}$$

and if $f(x)$ is instead convex upward, the sense of the inequality reverses; that is,

$$M_i < \int_{x_i}^{x_{i+1}} f(x) < T_i \, . \tag{10.9}$$

For our example $f(x) = x \tan x$, which is convex upward, that is, $f(x)$ lies above its tangent lines, the midpoint rule gives a lower bound, while the trapezoidal rule gives an upper bound. Evaluating $f(x)$ at 11 equally spaced nodes on $[0, {}^{\pi}/_{4}]$ (giving 10 panels between the nodes) gives us a midpoint rule estimate of 0.1851249174, whereas the trapezoidal rule estimate is 0.1871047441, so we know that the true area A satisfies $0.185 < A < 0.187$. Exercise 10.2 asks you to get more accuracy. In each case, note that the rule gives the exact (apart from roundoff) area under a different curve $f + \Delta f$, and that by taking enough panels in the mesh, we can make Δf as small as we please.

There are an infinite number of quadrature rules. Almost all of them look like

$$\int_a^b f(x)dx \doteq \sum_{k=1}^{n} w_k f(\tau_k) \, , \tag{10.10}$$

usually with $\tau_k \in (a,b)$ and with $\sum_{k=1}^{n} w_k = b - a$. Alternatively, they can be interpreted as replacing the integral or *true average* by a finite-sum *weighted average*:

$$(b-a)\overline{f} \doteq \sum_{k=1}^{n} w_k f(\tau_k) \, . \tag{10.11}$$

Reverse-engineering an $f+\Delta f$ for a given quadrature rule is usually simple if one knows the definition of the rule. As should be obvious, there are infinitely many possible such $f+\Delta f$ if we start with the value of the definite integral only.

We will analyze the error made by the Riemann sum rules in a later section, but we note for now that each of the first three rules is exactly correct for step functions with heights given by $f(x_k)$ as appropriate, and that the trapezoidal rule is exact for piecewise linear functions interpolating the data $(x_k, f(x_k))$. The midpoint rule is *also* exactly correct for the tangent line approximation $f(x) = f(x_{k+1/2}) + f'(x_{k+1/2})(x - x_{k+1/2})$ because the integral of the degree-1 term is zero. In all cases, these rules give the exact areas under functions slightly different that the one intended, namely, $f(x)$. This raises the question of how significant such changes are; this is the recurring question of conditioning.

10.1 Conditioning of Quadrature

Consider the intergral $I = \int_a^b f(x)\,dx$ of f and the integral

$$I + \Delta I = \int_a^b (f(x) + \Delta f(x))\,dx\,. \tag{10.12}$$

Then the relation

$$|\Delta I| = \left|\int_a^b \Delta f(x)\,dx\right| \le \int_a^b |\Delta f(x)|\,dx = 1\cdot\|\Delta f(x)\|_1 \tag{10.13}$$

holds, where the norm in the final term is the function 1-norm on this interval. In this norm, the absolute condition number is just 1! Therefore, in an absolute sense, *quadrature is well-conditioned.*

In a *relative* sense, things are harder:

$$\left|\frac{\Delta I}{I}\right| \le \frac{\displaystyle\int_a^b |\Delta f(x)|\,dx}{\left|\displaystyle\int_a^b f(x)\,dx\right|} = \frac{\|\Delta f\|_1}{|I|} = \frac{\|f\|_1}{|I|}\cdot\frac{\|\Delta f\|_1}{\|f\|_1}\,. \tag{10.14}$$

Therefore, the relative condition number is

$$\frac{\|f\|_1}{|I|} = \frac{\displaystyle\int_a^b |f(x)|\,dx}{\left|\displaystyle\int_a^b f(x)\,dx\right|}\,. \tag{10.15}$$

If f is large while its integral is small (i.e., f oscillates), this number can be very large.

Example 10.1. For instance, consider a perturbation of $\int_{-1}^{1} \cos(\omega t)/(1+t^2)\,dt$, such as

$$\int_{-1}^{1} \frac{\cos(\omega t)(1+\varepsilon \cos \omega t)}{1+t^2}\,dt\,, \tag{10.16}$$

which has a relatively small change to the integrand if ε is small. Detailed analysis shows that if $\varepsilon = 0$ and ω is large, the integral is ${}^{\sin(\omega)}/_{\omega} + O({}^{1}/_{\omega^2})$; but if ε is not zero, then the difference is, using the fact that $\cos^2\theta = {}^{(1+\cos 2\theta)}/_{2}$,

$$\varepsilon \int_{-1}^{1} \frac{{}^{1}/_{2} + {}^{\cos(2\omega t)}/_{2}}{1+t^2}\,dt = \frac{\pi}{4}\varepsilon + O(\frac{\varepsilon}{\omega})\,. \tag{10.17}$$

As a result, it follows that

$$\frac{\Delta I}{I} \doteq \frac{{}^{\omega\pi}/_{4}}{\sin \omega} \times \varepsilon\,, \tag{10.18}$$

and so the condition number goes to infinity as $\omega \to \infty$, even at values of ω where the integral isn't exactly zero. We return to this example in Sect. 10.8. ◁

We can say that quadrature of highly oscillatory integrands is ill-conditioned, in this relative sense. Small changes in the integrand will make large relative changes in the integral. Nonetheless, oscillatory integrals are of great practical interest, and we take them up in Sect. 10.8.

But there is a potentially worse problem than ill-conditioning. Can we do numerical integration at all?

Theorem 10.1 (Kahan (1980)). *Deterministic numerical integration of functions defined by procedures (”black boxes”) is impossible, even for continuous functions, unless the class of integrands represented by the black boxes is further constrained.*

Proof. Suppose to the contrary that someone has implemented a deterministic quadrature program that claims to integrate all continuous functions implemented as black boxes,[3] perhaps even just all polynomials. Call this program `Candidate`. Take the function $\mathrm{spy}(x)$ which *records its input* and returns 0, and call `Candidate` with it, asking to evaluate

$$\int_{0}^{1} \mathrm{spy}(x)\,dx = 0\,. \tag{10.19}$$

Of course, `Candidate` should tell you that the integral is zero. Now examine the list of values at which `Candidate` probed the ”spy” function. This list must necessarily be finite; suppose there are M numbers x_i in the list. We would expect

[3] A “black box” is a procedure of which one does not know the details; all one can do is call it with its input and receive its output. The presumption is that the black box here is deterministic: If one sends in an input τ and gets an output y, then that same y will occur every time the input τ is given to the box.

$0 \le x_i \le 1$ because it would be an unusual quadrature program[4] that evaluated its integrands outside the interval of integration, but this doesn't matter. What does matter is that the x_i are machine numbers. Now let

$$\text{malicious}(x) = K \prod_{i=1}^{M} (x - x_i)^2 \,, \tag{10.20}$$

for some constant K. Now ask `Candidate` to integrate "malicious," not "spy," over the interval $[0,1]$. Since `Candidate` was deterministic, it will sample "malicious" at the same x_i, at which "malicious" is zero, just as "spy" was. Notice that subtraction of two machine numbers $x_i - x_i$ is exactly zero. Therefore, `Candidate` will come to the same conclusion with "malicious" that it did with "spy," namely, that the integral is zero. But this is wrong:

$$\int_0^1 \text{malicious}(x)\, dx = K \int_0^1 \prod_{i=1}^{M} (x - x_i)^2 \, dx. \tag{10.21}$$

Since the integrand is nonnegative away from the nodes x_i, it can be made into any value one likes by an appropriate choice of constant K. ♮

This sounds like a contrived argument, but yet it happens. Consider the following example.

Example 10.2. In MATLAB, create the function `spyfn` (there is a built-in MATLAB function called `spy` that is useful for sparse matrices, so we don't want to shadow that) as follows:

```
function y = spyfn(x)
x
y=zeros(size(x));
```

Asking MATLAB to integrate this function by calling, say, `quad` gives us several values of x:

```
quad( @spyfn, 0, 1 )
```

yields 0, 0.1358, and several more. When we try to use these numbers in our "malicious" program, they don't work. Of course, they are only four-decimal-place approximations. Even when we use `format long` or `format long e`, they still don't work, though: because they are the results of converting the internal IEEE doubles to decimal. We have to use `format hex` to see the intermediate xs for ourselves! When we do this—or, better, if we use a persistent variable to record the numbers—we can write our malicious program:

```
function [ y ] = malice( x )
%MALICE nonzero fn pretending to be zero
%   output from spy stored in "secrets"
secrets = hex2num({ '0000000000000000',   '3fc16191148fd9fd', ...
   '3fd16191148fd9fd',   '3fe0000000000000', ...
```

[4] But not unprecedented; there are even methods that use complex nodes to evaluate real integrals.

```
    '3fe74f3775b81302',   '3feba79bbadc0981', ...
    '3ff0000000000000',   '3fb16191148fd9fd', ...
    '3fca12599ed7c6fc',   '3fd8b0c88a47ecfe', ...
    '3fe3a79bbadc0981',   '3fe97b69984a0e42', ...
    '3fedd3cddd6e04c0' });
y = zeros( size(x) );
n = length(y);
for i=1:n,
    y(i) = 1.0e15;
    for j=1:length(secrets),
       y(i) = y(i)*(x(i)-secrets(j))^2;
    end
end
end
```

Sure enough, asking for the area under malice on the interval $0 \le x \le 1$ using quad returns, as it did for spyfn, just 0. Yet when we plot malice we see that it is symmetric about $x = 1/2$, and has maximum value about $y = 50$. Indeed, if we ask quad to integrate this function on $0 \le x \le 1/2$, we get 1.50674 and a similar number for $1/2 \le x \le 1$, so the true area is larger than 3. This demonstrates that Kahan's proof of impossibility is constructive—or is it destructive? Anyway, numerical integration is technically impossible. Not that will stop us, and we largely ignore this difficulty. Not to worry; we have others. ◁

The way to rescue quadrature from this theorem is to restrict ourselves to integrating polynomials of not "too high" a degree. The rules that we show here will be adequate for polynomials of degree less than, say, $2M-1$. The theorem is something to be kept in mind, however.

10.2 Equally Spaced Partitions and Spectral Methods

What is the convergence behavior for the simple Riemann-sum rules? Consider the trapezoidal rule first. The trapezoidal rule is exact for integrating a piecewise linear $p(x)$, where $p(x)$ interpolates $f(x)$ at each x_k. The linear interpolant is, on $x_i \le x \le x_{i+1}$,

$$f(x) = f(x_i) + \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}(x - x_i) + \frac{f''(\Theta_i)}{2}(x - x_i)(x - x_{i+1}), \tag{10.22}$$

where $\Theta_i = \Theta_i(x)$ is some unknown quantity that depends on x, between x_i and x_{i+1}. Then, because the integral of the linear interpolant is just the area of the trapezoid, T_i, we have

$$\begin{aligned}\int_{x_i}^{x_{i+1}} f(x)\,dx &= T_i + \frac{1}{2}\int_{x_i}^{x_{i+1}} f''(\Theta_i)(x - x_i)(x - x_{i+1})dx \\ &= T_1 + \frac{1}{2}f''(\Phi_i)\int_{x_i}^{x_{i+1}} (x - x_i)(x - x_{i+1})dx,\end{aligned} \tag{10.23}$$

for some possibly different $\Phi_i \in [x_i, x_{i+1}]$ by the extended mean value theorem.[5] Then

$$\begin{aligned}\int_{x_i}^{x_{i+1}} f(x)\,dx &= T_i + \frac{1}{2}f''(\Phi_i)(-\frac{1}{6}(x_{i+1}-x_i)^3) \\ &= T_i - \frac{1}{12}f''(\Phi_i)(x_{i+1}-x_i)^3\,. \end{aligned} \tag{10.24}$$

Therefore,

$$\int_a^b f(x)\,dx = \left(\sum_{i=0}^{n-1} T_i\right) - \frac{1}{12}\left(\sum_{i=0}^{n-1} f''(\Phi_i)(x_{i+1}-x_i)^3\right), \tag{10.25}$$

where the first term is now seen to be just the (composite) trapezoidal rule, while the second is the total forward error. If we suppose for simplicity that $x_{i+1}-x_i = h = {}^{(b-a)}\!/_{n}$ for all i, then we get

$$\int_a^b f(x)\,dx = \left(\sum_{i=0}^{n-1} \frac{1}{2}(f(x_i)+f(x_{i+1}))h\right) - \left(\frac{h^2}{12}(b-a)\frac{1}{n}\sum_{i=0}^{n-1} f''(\Phi_i)\right)\,. \tag{10.26}$$

That is, the error in the trapezoidal rule is, for equally spaced partitions, $O(h^2)$ and proportional to an average value of the second derivative of the integrand, and to the width of the interval. We will see that the factor $-{}^1\!/_{12}$ will play a role as well. In Sect. 10.3, we return to formula (10.24) and its predecessors.

We remark that the backward error, that is, the error in piecewise linear interpolation, is slightly easier: from (10.22) we have that $|f(x)-L_i(x)| \le |f''(\Theta_i)|{}^{h_i^2}\!/_{8}$, with the obvious meaning for $L_i(x)$, and so one could phrase the error analysis that way: The trapezoidal rule gives the exact integral for a function only slightly different to $f(x)$.

For the midpoint rule, the analysis is still simpler. If we Taylor expand $f(x)$ about a typical midpoint, so that

$$f(x) = f(x_{i+1/2}) + f'(x_{i+1/2})(x-x_{i+1/2}) + \frac{1}{2}f''(\theta_i)(x-x_{i+1/2})^2\,, \tag{10.27}$$

the integral of the degree 1 term vanishes, so that by the extended mean value theorem as before, we get

$$\begin{aligned}\int_{x_i}^{x_{i+1}} f(x)\,dx &= M_i + \frac{1}{2}f''(\psi_i)\int_{x_i}^{x_{i+1}} (x-x_{i+1/2})^2 \\ &= M_i + \frac{1}{24}f''(\psi_i)(x_{i+1}-x_i)^3\,. \end{aligned} \tag{10.28}$$

[5] This can be found, for example, in Clark (1972 p. 160). The theorem states that if $f(x)$ and $g(x)$ are continuous, and $g(x)$ has one sign on $[a,b]$, then there exists a point $c \in [a,b]$ such that $\int_a^b f(x)g(x)\,dx = f(c)\int_a^b g(x)\,dx$. Here $g(x) = (x-x_i)(x_{i+1}-x)$ has one sign on the interval.

When we add up all the rectangles, we obtain

$$\int_a^b f(x)\,dx = \left(\sum_{i=0}^{n-1} M_i\right) + \frac{1}{24}\left(\sum_{i=0}^{n-1} f''(\psi_i)(x_{i+1}-x_i)^3\right), \tag{10.29}$$

which in the simpler case of equally spaced points gives

$$\int_a^b f(x)\,dx = M_n + \frac{1}{24}(b-a)\left(\frac{1}{n}\sum_{i=0}^{n-1} f''(\psi_i)\right)(x_{i+1}-x_i)^2 . \tag{10.30}$$

As before, we have that the error is $O(h^2)$, proportional to $b-a$, and proportional to an average value of the second derivative. Again, we could interpret this as the exact integral of the best piecewise linear approximation of $f(x)$ at the midpoints of each interval.

Notice that the error of the midpoint rule is twice as good as that of the trapezoidal rule, and of opposite sign, assuming that the two averages of f'' in each formula are approximately equal. This suggests that the combination $S = {}^{(2M+T)}\!/_3$ ought to be even better. This turns out to be Simpson's rule, which we derive in a different way, below.

10.2.1 Simpson's Rule

Consider two panels of equal width, $x_{2k} < x < x_{2k+1}$ and $x_{2k+1} < x < x_{2k+2}$. Suppose we wish to evaluate the integral

$$\int_{x_{2k}}^{x_{2k+2}} f(x)\,dx = F(x_{2k+2}) - F(x_{2k}), \tag{10.31}$$

where we know that $F'(x) = f(x)$ [although we may not be able to find an elementary expression for $F(x)$]. There is a related *interpolation problem*, namely, given $F(x_{2k})$, $f(x_{2k})$, $f(x_{2k+1})$, and $f(x_{2k+2})$, find $F(x_{2k+2})$. This is, properly speaking, a Hermite–Birkhoff interpolation problem,[6] because we are missing the datum $F(x_{2k+1})$, but let us proceed as if we knew that datum. Form the partial fraction expansion

$$\begin{aligned}\frac{1}{(\xi-x_{2k})^2(\xi-x_{2k+1})^2(\xi-x_{2k+2})^2} &= \frac{{}^3\!/_4}{h^5(\xi-x_{2k+1}+h)} + \frac{{}^1\!/_4}{h^4(\xi-x_{2k+1}+h)^2}\\ &\quad + \frac{1}{h^4(\xi-x_{2k+1})^2} - \frac{{}^3\!/_4}{h^5(\xi-x_{2k+1}-h)} + \frac{{}^1\!/_4}{h^4(\xi-x_{2k+1}-h)^2},\end{aligned}$$

[6] It is discussed in Problem 8.12.

where we have used the fact that $x_{2k} = x_{2k+1} - h$ and $x_{2k+2} = x_{2k+1} + h$, that is, that the two panels are of equal width. Notice that the coefficient of the $1/(\xi - x_{2k+1})$ term is zero; that is, in the Hermite interpolation problem, the datum $F(x_{2k+1})$ is *not used*. This is an inherent consequence of the symmetry of the situation. Proceeding as in Sect. 8.2 and taking a contour integral of $F(z)$ times this partial fraction expansion over a large enough contour to enclose all of x_{2k}, x_{2k+1}, and x_{2k+2}, we see that

$$\frac{3F(x_{2k})}{4h} + \frac{f(x_{2k})}{4} + f(x_{2k+1}) + \frac{f(x_{2k+2})}{4} - \frac{3F(x_{2k+2})}{4h} = 0\,, \tag{10.32}$$

and that because the denominator was degree 6, this equation is true for all polynomials $F(z)$ of degree at most 4 [and so true for all polynomials $F'(z) = f(z)$ of degree at most 3]. For such $F(z)$, this equation can be rearranged to give

$$F(x_{2k+2}) = F(x_{2k}) + \frac{h}{3}\left(f(x_{2k}) + 4f(x_{2k+1}) + f(x_{2k+2})\right)\,. \tag{10.33}$$

This is Simpson's rule across two panels. If we have an even number n of panels, which are pairwise of equal width, we may combine this rule to give a composite Simpson's rule. In the special case where all panels have width h, the rule becomes h times the "ends" plus twice the "evens" plus four times the "odds." As we have seen, it is exact for functions $f(x)$ that are piecewise cubic. Conventionally, the error term is reported as ${}^{(b-a)h^4}/_{90}$ times (a bound for) an average value of the fourth derivative of $f(x)$. That this rule is exact for polynomials with $\deg f \le 3$ can also be seen from the presence of the fourth derivative in the error bound.

10.2.2 Practical Error Estimation

In the previous subsections, theoretical error estimates containing unknown derivatives were used to get a qualitative idea of the behavior of various quadrature formulæ. We were satisfied there with asymptotic error estimates: $O(h)$, $O(h^2)$, and $O(h^4)$ for various methods using equally spaced partitions. Generally, for a smooth enough problem, a higher-order method is more accurate for the same amount of work (work being the number of function evaluations). In practice, we want more: We would like to know how accurate our answer is.

Given the earlier (absolute) conditioning analysis, this is equivalent to asking how accurately we have approximated $f(x)$ by our engineered problem $(f + \Delta f)(x)$. But quadrature is so easy (in the absolute sense) that we are often happy with a simple comparison of two different methods (perhaps the same method with different mesh sizes).

Example 10.3. Consider $f(x) = {}^1/_{(1+x^{64})}$ on $0 \le x \le 1$. If we now apply the midpoint rule using 100 evaluation points at $x_k = {}^1/_{100}(k + {}^1/_2)$, $0 \le k \le 99$, then our resulting approximation to $\int_0^1 f(x)\,dx$ is $M = 0.989434062837952$. The function is convex down, so we know that this is larger than the true integral. Using the trapezoidal rule instead, on $x_k = {}^{k}\!/\!{}_{100}$ for $0 \le k \le 1$ (that is, 101 points, each of which is different to what we had before), this gives us an approximation $T = 0.989233192109316$. By our previous analysis, we know that $T < \int_0^1 f(x)\,dx < M$. The difference $M - T \approx 2.0 \cdot 10^{-4}$, and this gives a narrow interval containing the true answer (rounding errors are too small to have affected the containment). This can be written as $\int_0^1 f(x)\,dx \approx 0.989^4_2$. But we can say more: We know that the approximation $S = {}^{(2M+T)}\!/\!{}_3 = 0.989367105928406$ is supposed to be $O(h^4)$ accurate, and indeed it differs by about $1.1 \cdot 10^{-7}$ from the value returned by MATLAB's recommended routine `quadgk`. ◁

That example shows that using two different methods can give a good idea of the error. The idea of using the *same* method with different mesh sizes leads also to an effective algorithm, which we sketch in the next section. The use of such practical error estimates is fundamental to adaptive methods, which we take up later.

10.2.3 Extrapolation Methods

For smooth problems, extrapolation is a powerful method. We have already seen the left- and right-hand Riemann sum methods, each $O(h)$, combined to form the $O(h^2)$ trapezoidal rule; and the $O(h^2)$ trapezoidal and midpoint rules averaged by $S = {}^{(2M+T)}\!/\!{}_3$ to get the $O(h^4)$ Simpson's rule. This idea can be systematically extended by what is called Richardson extrapolation; in the particular case of quadrature, this is called Romberg integration.

The key to Richardson extrapolation is the *existence* of an asymptotic series expansion of the error, that is, asymptotic as the panel width h goes to 0. The Euler–Maclaurin sum formula, in a form convenient for our use here, is

$$h\sum_{i=0}^{n} f(ih) \sim \int_0^1 f(x)\,dx - hB_1\left(f(1)+f(0)\right) + \sum_{k=1}^{m} \frac{h^{2k}B_{2k}}{(2k)!}\left(f^{(2k-1)}(1) - f^{(2k-1)}(0)\right), \tag{10.34}$$

where $h = {}^{1}\!/\!{}_{n}$, $B_1 = -{}^{1}\!/\!{}_{2}$, and B_{2k} are the Bernoulli numbers.[7] This is an asymptotic series, and so it makes sense as $n \to \infty$ and thus $h \to 0$ for any fixed number of terms m on the right-hand side. Note that $f^{(2k-1)}(1) - f^{(2k-1)}(0)$ is, by the mean value theorem, equal to $f^{(2k-2)}(\theta)$ for some $\theta \in [0,1]$. We don't use that here, but note it as a point of contact with our earlier error analysis.

[7] There is quite a nice article in Wikipedia on Bernoulli numbers. As a quick definition, expanding ${}^{t}\!/\!{}_{(\exp(t)-1)}$ in a Taylor series gives $\sum_{n \ge 0} {}^{B_m t^m}\!/\!{}_{m!}$, generating the "first" Bernoulli numbers. For a very beautiful discussion of the Euler–Maclaurin sum formula itself, see Hairer and Wanner (1996 p. 160).

Also, note that (10.34) can be rearranged to get

$$\begin{aligned} T &= -\frac{h}{2}f(0) + h\sum_{i=0}^{n} f(ih) - \frac{h}{2}f(1) \\ &\sim \int_0^1 f(x)\,dx + \sum_{k=1}^{m} \frac{h^{2k}B_{2k}}{(2k)!}\left(f^{(2k-1)}(1) - f^{(2k-1)}(0)\right), \end{aligned} \tag{10.35}$$

which gives an asymptotic error estimate for the trapezoidal rule. Thus, for the trapezoidal rule, the error expansion has only *even order* terms: The same is true for the midpoint rule. That is,

$$\int_a^b f(x)\,dx - T_n \sim \sum_{k=1}^{\infty} c_{2k}h^{2k}. \tag{10.36}$$

We saw explicitly that the first term c_2 can be interpreted to contain an average f''; similarly, the higher terms contain higher derivatives. But as we said, one does not have to know the c_{2k} explicitly.

The key idea to Richardson extrapolation is that we can take a combination of two separate evaluations of the trapezoidal rule, with different but related widths h, and combine them in such a way as to eliminate the c_2 term. For example, by computing a trapezoidal sum with width h and then another one with width $h/2$ (the usual choice), we would have

$$\begin{aligned} I &= T_h + c_2h^2 + c_4h^4 + \cdots \\ &= T_{h/2} + c_2(h/2)^2 + c_4(h/2)^4 + \cdots. \end{aligned} \tag{10.37}$$

Clearly, the combination $(4T_{h/2} - T_h)/3$ would satisfy

$$I = \frac{4T_{h/2} - T_h}{3} + (\frac{1}{4} - 1)\frac{c_4h^4}{3} + \cdots, \tag{10.38}$$

which, provided the higher derivatives are well behaved, gives us an $O(h)^4$ formula, which ought to be more accurate. Using the trapezoidal rule with width $h/4$ allows us to do it again, giving another fourth-order error estimate; we can then combine the two fourth-order integrals to get a sixth-order accurate integral, and so on. We do not pursue this idea further here, but note that it is quite practical.

Example 10.4. Let us continue with the same example as used previously: $f(x) = 1/(1+x^{64})$ on $0 \le x \le 1$. If we use the trapezoidal rule with 100 subintervals, we previously found that the integral was approximately $T_{100} = 0.989233192109316$. Now if we include all the points $x_k = (k + 1/2)/100$ for $0 \le k \le 99$, then we will be able to evaluate the trapezoidal rule with 200 subintervals (note that this is different than the combination of the midpoint and trapezoidal rules used in the previous example: Here we are using *only* the trapezoidal rule). This gives

$T_{200} = 0.989333627473634$. By the Euler–Maclaurin formula previously, the error in this answer should be about a quarter of the previous error. We form the new combination $(4T_{200} - T_{100})/3 = 0.989367105928406$. Again, this is $O(h^4)$ accurate (and happens to be Simpson's rule again, which ought not to be surprising since we're using the same information as previously, just combining it differently). ◁

Remark 10.1. Of course, we are left as usual with an accurate answer whose error we know only roughly. Using that accurate answer, we may accurately estimate the error in *less* accurate answers, and there are situations where we prefer to know how accurate the answer is, but by the same reasoning, we know that this answer is *at least* as accurate as the lower-order results, and so why wouldn't we use it? ◁

10.2.4 Spectral Accuracy for Smooth Periodic Functions

Suppose we integrate a function $f(x)$ with the trapezoidal rule. Then, as per our error analysis done previously, we expect that the error is $O(h^2)$. Sometimes, however, our trapezoidal estimate is *vastly* more accurate than this expectation!

Example 10.5. For instance, consider

$$f(v) = \frac{(1 - v\cot v)^2 + v^2}{z + v\csc v e^{-v\cot v}} \tag{10.39}$$

for some $z > -\exp(-1)$ and with $0 \le v \le \pi$. Then, using the trapezoidal rule on

$$\frac{W(z)}{z} = \frac{1}{\pi}\int_0^{\pi} f(v)dv \tag{10.40}$$

[here $W(z)$ is the Lambert W function of z] with $h = \pi/200$, say, one *expects* that the error would be $K(\pi/200)^2 \doteq 10^{-5}$. And this expectation is fulfilled:

$$\frac{1}{\pi}\int_0^{\pi} f(v)dv = T_{200} + \approx 10^{-4}. \tag{10.41}$$

But if instead we integrate this *even* function over a symmetric interval with the trapezoidal rule and 200 panels, we get the astonishing accuracy

$$\frac{W(z)}{z} = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(v)dv = T_{200} + \approx 8\cdot 10^{-17}. \tag{10.42}$$

Periodic extension (with, for example, very flat edges) that happens to be infinitely differentiable everywhere turns out to be the key. Note that for our example, the limit as $v \to \pi$ from the left of $f(v)$ is zero, as are all its derivatives. The periodic extension of $f(v)$ with period 2π is infinitely flat at the odd multiples of π. This is visible in the graph (see Fig. 10.2).

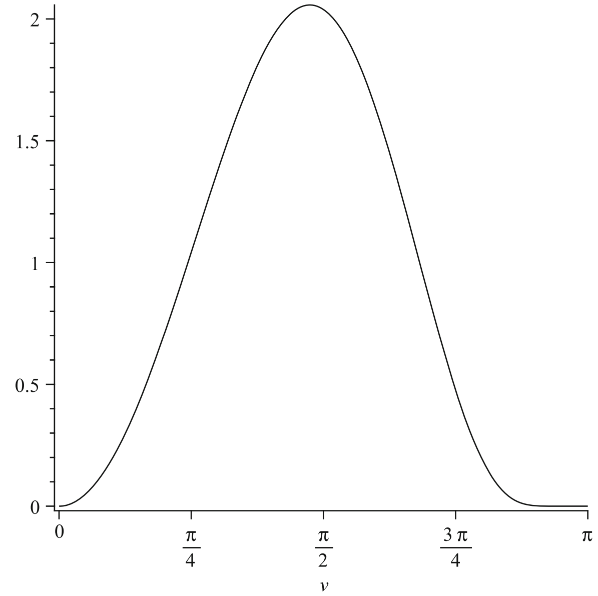


Fig. 10.2 The graph of Eq. (10.39) on $0 \le v \le \pi$. Notice the significant flattening at π, in contrast to the mere double zero at the origin. This even function, when extended periodically with period 2π, is infinitely differentiable (but not analytic). All order derivatives are zero (although the function is not identically zero in any neighborhood) at v equal to odd multiples of π

In effect, all the error terms arising from the Euler–Maclaurin asymptotic formula are zero, so only exponentially small error terms remain. When the error is smaller than all orders $O(h^p)$ as $h \to 0$, we say the convergence is called. For example, if the error is $\propto K \cdot 2^{-1/h} = O(2^{-n})$, then this is smaller than $O(h^p) = O(1/n^p)$ for any fixed p. ◁

10.3 Adaptive Quadrature

We have not yet examined the issue of adapting the integration method to the problem. Many practical codes—probably all practical codes that are in wide use—are adaptive in this sense. Instead of using a fixed number of panels of a mesh chosen a priori, the codes probe the function in an effort to decide where to sample further, and try to place more samples where more are needed.

Consider Fig. 10.3, showing our previous example $f(u) = 1/(1+u^{64})$, which (we now tell you) we borrowed from Kahan (1980). We wish to compute $F(x) = \int_0^x f(u)\,du$. As Kahan (1980 p. 25) points out, "[T]he extent to which $F(x)$ is here more complicated than $f(u)$ is atypically modest out of consideration for the typesetter." The symbolic expression referred to in this quote is

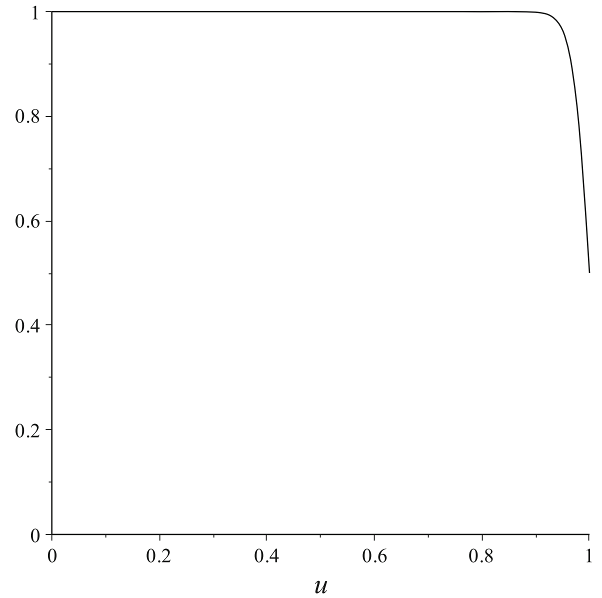


Fig. 10.3 The graph of ${}^1\!/\!{}_{(1+u^{64})}$ on $0 \le u \le 1$. Nearly all the variation in the function takes place in the last 10% of the interval

$$F(x) = \frac{1}{32}\sum_{k=1}^{16}\left(\sin\theta_k \arctan\left(\frac{2x\sin\theta_k}{1-x^2}\right) + \frac{1}{2}\cos\theta_k \ln\left(1+\frac{2}{\dfrac{x+x^{-1}}{2\cos\theta_k}-1}\right)\right) + \left(\frac{\pi}{64}\csc\left(\frac{\pi}{64}\right)\operatorname{signum}(x) \text{ if } x^2>1\right), \quad (10.43)$$

where $\theta_k = (k-{}^1\!/\!{}_2){}^{\pi}\!/\!{}_{32}$. As an aside, MAPLE produces a different-looking symbolic answer:

$$F(x) = -\sum_{\_R=\mathrm{RootOf}\left(2^{384}\,\_Z^{64}+1\right)} \_R\,(6\ln(2)+\ln(\_R)) + \sum_{\_R=\mathrm{RootOf}\left(2^{384}\,\_Z^{64}+1\right)} \_R\ln(x+64\_R)\,. \quad (10.44)$$

MAPLE's answer is actually a bit worse-looking than what we present here: MAPLE explicitly writes the 2^{384} as a multiplied-out integer. We prefer Kahan's expression for the answer.

Either way, numerical integration will generate more intelligible answers. MAPLE can evaluate its ugly symbolic answer at $x = 1$ to get $0.9893669892 + 0.0i$. Kahan's formula just gives 0.9893669892 without the zero imaginary part. But if all we want is the number, then there's no reason to go through either of the formulæ. Kahan uses this example to argue in favor of numerical quadrature.

For this integral, equally spaced partitions of the interval would cause us to do too much work. Rather, the computer would be doing too much work, which might matter.[8] For efficiency, we want to partition $[a,b]$ in a way that puts more points x_k where needed.

The MATLAB routine `quad` does so in a recursive fashion. If an integral over an interval is not estimated to be accurate enough, it refines the partition and tries again. For this example,

```
f = @(x) 1.0./(1+x.^64);
q = quad( f, 0, 1, 1.0e-6, 1 )
```

yields

```
 9      0.0000000000     2.71580000e-001      0.2715800000
11      0.2715800000     4.56840000e-001      0.4568399999
13      0.7284200000     2.71580000e-001      0.2599542310
15      0.7284200000     1.35790000e-001      0.1357887364
17      0.8642100000     1.35790000e-001      0.1254882693
19      0.8642100000     6.78950000e-002      0.0677373449
21      0.9321050000     6.78950000e-002      0.0574311353
23      0.9321050000     3.39475000e-002      0.0325601539
25      0.9660525000     3.39475000e-002      0.0248598823
27      0.9660525000     1.69737500e-002      0.0141626395
29      0.9830262500     1.69737500e-002      0.0106976902

q =

   0.989366564796066
```

and we can see that `quad` has used intervals of width from 2.7158×10^{-1} down to 1.697×10^{-2}; it used 9 function evaluations to start. The second line records an attempt to use a larger step, of width 0.45684, which succeeded, getting to $x = 0.72842$. The third line records an attempt to take a step of size 0.27158 again, but this time it *failed* and reduced the step further, after which it succeeded, getting to $x = 0.86421$. It had to reduce the width yet again, to get to 0.932105, and then again to get to 0.966, and the width of the penultimate step was 0.01697 to arrive at 0.983. In the end it used 29 function evaluations. In comparison to MAPLE we see that the final answer is accurate to about six figures (using `format long` to display the answer).

Remark 10.2. We are not commenting on how, exactly, this program is estimating the accuracy of the resulting answer. Of course, the details really matter, but for the present discussion we merely note that it is so doing. ◁

Example 10.6. Consider the more difficult problem

$$F(x) = \int_0^x \frac{dt}{1+t^{1024}}\,, \tag{10.45}$$

[8] It's true that it matters less and less as computers get more powerful—compare the discussion in Kahan (1980), where the machine takes minutes, hours, and even days to do one-dimensional integrals; more than 30 years later, those examples are dated indeed—but these ideas will matter greatly for the multidimensional case.

increasing the exponent from 64. Years ago, RMC attempted to find $F(1)$ on an HP 48S calculator; it was reported by the calculator as 1 to 12 places and no error flag indicated that there was a difficulty; however, the reference answer differs from 1 by about $\ln(2)/1024$, which was much more than the tolerance. The difficulty is that, for any $x_i < 0.977$ in 11-digit decimal arithmetic, $f(x_i)$ is computed as 1 to all places, and therefore the routine would have had to sample between 0.977 and 1, less than 3% of the interval; and it did not. This is exactly the impossibility proof for numerical integration!

Let us go further. The true value of $\int_0^1 (1+t^n)^{-1}\,dt$ is beautifully expressible in terms of a 2F1 hypergeometric function, $\mathscr{F}([1, 1/n], [1+1/n], -1) \sim 1 - \log(2)/n + O(1/n^2)$. Quite a lot is known about this function, and this analytic answer is very useful indeed. Thus, Kahan's point with this example, namely, that symbolic integration is often worse than numeric integration, becomes neatly reversed if one extends one's symbolic alphabet.

We hasten to emphasize that Kahan is indeed correct: Numerical integration is generally much to be preferred and works much more often than analytic methods do; the irony is that one of the examples he chose actually supports the opposite point, *and* that the numerical quadrature fell afoul of his own proof!. ◁

We now return to discussing `quad` (which succeeds on the $n = 1024$ case anyway). What is the program trying to do? Rather than answer this in detail, we try to present a heuristic that gives some of the flavor. We introduce here an idea that will be important for the numerical solution of initial and boundary value problems for ordinary differential equations, namely, the idea of *equidistribution*. For the midpoint rule, the error on each panel was approximately $f''(x_{k-1/2})h_k^2/24$, taking the central value as an estimate of the average value of the second derivative. If the error is *equidistributed*, then the error in each panel is roughly the same:

$$\frac{f''(x_{k-1/2})}{24}h_k^2 = \frac{\phi}{24}h^2, \tag{10.46}$$

where here $h = (b-a)/n$ is the mean panel width and ϕ is some constant (we will identify this constant later as a Hölder mean of the samples of the second derivative). To repeat: Equally distributing the error means that the local 2nd derivative times the local step width h_k squared is the *same* as some average second derivative times the *mean* step width h squared.

Using 10 panels and the midpoint rule on this example, if we use an equally spaced partition, we get the terrible answer 0.9963, wrong in the second decimal place, with a relative error about 7×10^{-3}. If we use 10 panels, but just roughly try to equidistribute the error according to the rule above, using the nodes

$$\left[0, \frac{3}{4}, \frac{13}{16}, \frac{7}{8}, \frac{29}{32}, \frac{15}{16}, \frac{61}{64}, \frac{31}{32}, \frac{125}{128}, \frac{63}{64}, 1\right], \tag{10.47}$$

we get the much better answer 0.98951, out by only 1.5×10^{-4}. That is, the same amount of work gets us better than 20 times the accuracy.

If instead of using 10 panels, we split each of those 10 panels into two equal-width subpanels, and use Simpson's rule on each of those, we get an excellent answer: 0.989365863, which has a relative error about 1.1×10^{-6}. This is comparable to the results from `quad` above, with a roughly comparable amount of work: It used about a third of its function evaluations to decide how to split the interval up, and we did a roughly comparable (nonscientific) amount of work to try to find an approximate equidistribution. By comparison, Simpson's rule with 20 equally spaced panels gives the terrible answer 0.9351, which is out by 0.05—not even as good as the 10-panel midpoint rule! This is because the large high-order derivatives nullify any advantage gained by the high power of h in the error formula, naively used. But, with equidistribution, the high-order accuracy of the method is recaptured. Adaptivity is an extremely powerful tool.

On a final note, with an excessive amount of work we can find a set of nodes that (nearly) exactly equidistributes the true midpoint rule error:

$$\left[0.0, 0.8888, 0.9173, 0.9346, 0.9472, 0.9574, 0.9662, 0.9741, 0.9817, 0.9896, 1.0\right].$$

With this set of nodes, the midpoint rule gives 0.989439. The error is $7.2 \cdot 10^{-6}$ on each interval, all the same (though notice that the widest interval has $h = 0.888$ and the smallest has $h = 0.0076$). Because the error also has the same sign on each interval, the total error is just the sum of those, or $7.2 \cdot 10^{-5}$, about 100 times better than the equally spaced rule, and 5 times better than our rough equidistribution (10.47). However, the cost of finding this optimal mesh was so high that we were long past the point of diminishing returns for this example. In general, we don't want to spend more on computing the optimal mesh than we save in computing a good answer! Most of the practical work in adaptive quadrature is an attempt to achieve something close to what rough equidistribution would do, but at a modest cost.

We will show in Chap. 12 that equidistribution is optimal, in a certain sense. Of course, we don't wish to have to work too hard to find the optimal equidistributing partition either; as we did here, we will be satisfied with a rough equidistribution. We end by repeating that quadrature methods don't actually try to equidistribute their error; they just keep sampling, and never throw anything away; but equidistribution is a kind of ideal goal.

10.4 Gaussian Quadrature

Gaussian quadrature is classically thought to be most appropriate for certain classes of integrand, for example, the following particular classes of integrals times weight functions:

$$\int_{-1}^{1} f(x)\,dx\,, \quad \int_{-1}^{1} \frac{f(t)}{\sqrt{1-t^2}}\,dt\,, \quad \int_{0}^{\infty} e^{-x} f(x)\,dx\,, \quad \int_{-\infty}^{\infty} e^{-x^2} f(x)\,dx.$$

Gauss solved all of these, and more, with an elegant and powerful method. We will begin more naively. Consider

$$\int_{-1}^{1} \frac{f(t)}{\sqrt{1-t^2}} dt = w_1 f(\tau_1) + w_2 f(\tau_2). \tag{10.48}$$

There are four unknowns here: the two τ_ks and the two w_ks. Can we make this *exact* for polynomials of degree no more than 3, which also have four degrees of freedom? The answer is "yes," and the general answer involves orthogonal polynomials, in this case, Chebyshev polynomials. We will examine the first two cases here, in two different ways. We leave a sketch of the general case to the Notes and References section.

10.4.1 Gauss–Legendre Integration (Case I)

We take the following discussion from Butcher et al. (2011). Consider the weight function $w(x) = 1$ on the interval $[-1, 1]$. That is, we wish to find accurate formulæ

$$\int_{-1}^{1} f(x)\,dx = \sum_{i=1}^{n} w_i f(\tau_i)\,. \tag{10.49}$$

We use interpolational ideas, and look for an approximation

$$F(b) - F(a) = \sum_{i=1}^{n} \beta_i f(\tau_i), \qquad F' = f\,. \tag{10.50}$$

We will use a denominator that evaluates $F(z)$ only at the endpoints, $z = -1$ and $z = 1$, but *in potentia* evaluates both $F(z)$ and $F'(z) = f(z)$ at all the interior nodes τ_i. We will try to choose the τ_i so that the residues multiplying the $F(\tau_i)$, as in the Simpson's rule computation in Sect. 10.2.1, are zero. Therefore, consider

$$0 = \frac{1}{2\pi i} \oint \frac{F(z)}{(z+1)(z-1)P(z)^2} dz\,. \tag{10.51}$$

Here, we have written $P(z)$ for the polynomial that is zero at these (as yet unknown) nodes. That is,

$$P(z) = \prod_{i=1}^{n} (z - \tau_i)\,. \tag{10.52}$$

If we can find such a $P(z)$, then the contour integral will be zero, and our formula exact, for all polynomials $F(z)$ with $\deg F \leq 2n$. So, choose one of the nodes, call it τ, and write

$$P(x) = (x - \tau)Q(x)\,. \tag{10.53}$$

Then, we also have

$$\begin{aligned} P' &= Q(x) + (x-\tau)Q'(x) & \Rightarrow \quad & P'(\tau) = Q(\tau) \\ P'' &= Q' + Q' + (x-\tau)Q'' & \Rightarrow \quad & P''(\tau) = 2Q'(\tau)\,. \end{aligned} \tag{10.54}$$

Now we are ready to compute the terms of the partial fraction expansion of

$$\frac{1}{(z+1)(z-1)P(z)^2} = \frac{1}{(z+1)(z-1)(z-\tau)^2 Q(z)^2}\,, \tag{10.55}$$

valid near $z=\tau$; this is equivalent to finding the Taylor series at $z=\tau$ of

$$\begin{aligned} \frac{1}{(z+1)(z-1)Q(z)^2} = {} & \frac{1}{(\tau+1)(\tau-1)Q(\tau)} \\ & - \frac{2\tau Q(\tau) + Q'(\tau)\left(\tau^2-1\right)}{(\tau+1)^2(\tau-1)^2(Q(\tau))^2}(z-\tau) + O(z-\tau)^2\,. \end{aligned} \tag{10.56}$$

The coefficient of $1/(z-\tau)$ will vanish, therefore, if

$$\begin{aligned} 0 &= 2\tau Q(\tau) + (\tau^2-1)Q'(\tau) \\ &= (\tau^2-1)P''(\tau) + 2\tau P'(\tau) \end{aligned} \tag{10.57}$$

(where we have used (10.54) to replace Q' and Q). This is to be zero for $\tau = \tau_1, \tau_2, \ldots, \tau_n$. That is, this degree-$n$ polynomial is a multiple of P! Remembering that P is monic, and noticing that the leading coefficients of x^2P'' and xP' are both known, we can identify the constant of proportionality and thus deduce that

$$(x^2-1)P''(x) + 2xP'(x) - n(n+1)P(x) = 0\,. \tag{10.58}$$

The polynomial solutions of this equation are the *Legendre polynomials*, the first few of which are (in nonmonic form) 1, x, $(3x^2-1)/2$, $x(5x^2-3)/2$, $(35x^4-30x^2+3)/8$, ... See `JacobiP(k,0,0,x)` in MAPLE. Also, the first eleven Legendre polynomials are plotted in Fig. 10.4.

Now, we have found the nodes, but what about the weights w_i? They come out of the remaining residues, that is, the rest of the partial fraction expansion, but in any case the hard part was the τ_i; a naive attack on the problem gives rise to equations that are *nonlinear*. For completeness, we note that the barycentric weight at $z=-1$ is $-1/(2(P'(-1))^2)$, the weight at $z=1$ is $1/(2(P'(1))^2)$, and at all interior nodes is it $1/((\tau_k^2-1)Q^2(\tau_k))$. One can use the differential equation to identify $P'(1) = n(n+1)P(1)/2$ and similarly for $P'(-1)$. This completely gives the weights in terms of the nodes.

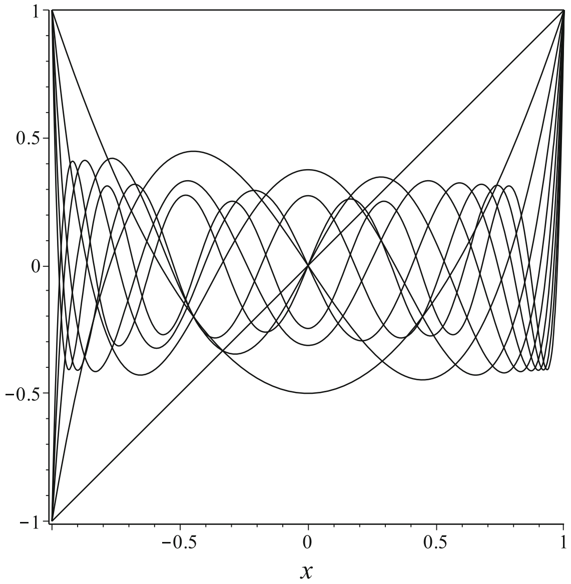


Fig. 10.4 The first 11 Legendre polynomials plotted on $-1 \le x \le 1$. Quadrature at the nodes given by the degree-n Legendre polynomial is exact, on this interval, for polynomials $f(x)$ of degree at most $2n-1$. Gauss was the first to deduce this

Example 10.7. If $n = 8$, then the nodes (computed in MAPLE to 30 digits[9]) are approximately

$$[-0.96029, -0.79667, -0.52553, -0.18343, 0.18343, 0.52553, 0.79667, 0.96029],$$

and the weight corresponding to node τ_i is, after a short partial fraction computation in MAPLE (which makes some algebraic simplifications[10])

$$w_i = \frac{71}{192} - \frac{50369}{235200}{\tau_i}^2 + \frac{7293}{78400}{\tau_i}^4 - \frac{429}{2240}{\tau_i}^6 \,. \tag{10.59}$$

Working to 30 digits in MAPLE, if we use this 8-node quadrature formula to evaluate $\int_{-1}^{1} {\sin(x)dx}/{x}$, we get the correct answer to 19 decimal places. If we replace the denominator with $x+2$, destroying the symmetry, the accuracy drops to 8 decimal places—still excellent for only 8 nodes used. ◁

Thus, as we see, Gaussian quadrature is a powerful tool.

[9] This was the hard way. We will see that roots of orthogonal polynomials are eigenvalues of a symmetric tridiagonal matrix, so really the nodes are not hard at all.

[10] Again this is the hard way: We will see an analytical formula that could have saved us this trouble.

10.4.2 Gauss–Chebyshev Integration (Case II)

Suppose now that we are interested in Chebyshev polynomials for quadrature. Note first that the Chebyshev polynomials are *orthogonal* with respect to this weight function on this interval:

$$\int_{-1}^{1} \frac{T_k(x)T_m(x)}{\sqrt{1-t^2}} dx = 0 \tag{10.60}$$

if $k \neq m$. Also,

$$\int_{-1}^{1} \frac{T_k^2(x)}{\sqrt{1-x^2}} dx = \begin{cases} \pi/2 & k \neq 0 \\ \pi & k = 0 \end{cases} \tag{10.61}$$

To see why this is true, let $x = \cos\theta$, with $0 \le \theta \le \pi/2$, so $dx = -\sin\theta d\theta$ and $\sqrt{1-x^2} = \sin\theta$. Also, $T_n(x) = \cos(n\arccos x) = \cos(n\theta)$, so

$$\int_{-1}^{1} T_k T_m w(x) dx = \int_{0}^{\pi} \cos(k\theta)\cos(m\theta) d\theta\,, \tag{10.62}$$

and we have reduced the problem to trigonometric integrals for which the orthogonality is well understood.

We take the following discussion from (Rivlin 1990, pp. 44–45). Suppose $p(t)$ is a polynomial of degree at most $2n-1$. The Lagrange interpolating polynomial to $p(t)$ at the nodes τ_k, which are the n zeros of $T_n(t)$, namely,

$$\tau_j = \xi_j = \cos\left(\frac{2j-1}{n}\frac{\pi}{2}\right) \qquad 1 \le j \le n, \tag{10.63}$$

is

$$L_{n-1}(p,T;t) = \sum_{j=1}^{n} p(\xi_j) \frac{T_n(t)}{T_n'(\xi_j)(t-\xi_j)} \tag{10.64}$$

[that is, the barycentric weight β_j is just the leading coefficient of $T_n(x)$ times $1/T_n'(\xi_j)$]. Since $p(t) - L_{n-1}(t) = 0$ for $t = \xi_1, \xi_2, \ldots, \xi_n$, we have

$$p(t) - L_{n-1}(t) = T_n(t) r(t) \tag{10.65}$$

for some polynomial $r(t)$ with $\deg r(t) \le n-1$. Now, $r(t) = \sum_{k=0}^{n-1} c_k T_k(t)$ for some c_k (because every polynomial may be expressed in terms of Chebyshev polynomials, that is what it means to form a basis). Therefore, by orthogonality,

$$\int_{-1}^{1} \frac{r(t)T_n(t)}{\sqrt{1-t^2}} dt = 0\,, \tag{10.66}$$

and as a result

$$\int_{-1}^{1} p(t)\frac{dt}{\sqrt{1-t^2}} = \int_{-1}^{1} L_{n-1}(p,T;t)\frac{dt}{\sqrt{1-t^2}}\,. \tag{10.67}$$

Hence, we may evaluate the right-hand side of the above by the weighted sum

$$\int_{-1}^{1} \frac{p(t)dt}{\sqrt{1-t^2}} = \sum_{j=1}^{n} w_j p(\xi_j)\,, \tag{10.68}$$

where the *w*s turn out to be explicitly calculable[11]:

$$w_j = \frac{1}{T_n'(\xi_j)}\int_{-1}^{1} \frac{T_n(t)dt}{(t-\xi_j)\sqrt{1-t^2}} = \frac{\pi}{n}\,. \tag{10.69}$$

Thus, the quadrature formula

$$\int_{-1}^{1} \frac{f(t)}{\sqrt{1-t^2}}dt \approx \frac{\pi}{n}\sum_{j=1}^{n} f(\xi_j) \tag{10.70}$$

is *exact* for all polynomials of degree $\le 2n-1$.

Other Gaussian quadrature formulæ are not quite so simple. Here both the nodes and the weights have simple analytical formulæ. In the general case, to find the nodes you have to solve a symmetric tridiagonal eigenvalue problem, and then evaluate a polynomial formula to find the weights.

Returning to the Chebyshev case, to understand its accuracy, approximate f *as best you can* by a degree-$(2n-1)$ polynomial $p^*(t)$ that agrees with f at the n Chebyshev points. This leaves some degrees of freedom. The error in this quadrature formula is thus

$$\int_{-1}^{1} (f(t)-p^*(t))\frac{dt}{\sqrt{1-t^2}}\,, \tag{10.71}$$

which is often *very* small.

Example 10.8. Taking $n = 48$, evaluating $f(t) = \frac{1}{1+25t^2}$ on 48 points, we get

$$\int_{-1}^{1} \frac{dt}{(1+25t^2)\sqrt{1-t^2}} \approx \frac{\pi}{48}\sum_{j=1}^{48} \frac{1}{1+25\xi_j^2} = 0.616117002990640, \tag{10.72}$$

while the true answer is $\pi/\sqrt{26} = 0.616117009400542$. We see an error that is about 6×10^{-9}, which is pretty good for only 48 points, on such a nasty example! ◁

One common situation where Gaussian quadrature is attractive is if you wish to perform many numerical integrations of a similar type. A prototype of that situation is the computation of orthogonal series, say Chebyshev series

[11] This is a remarkably simple formula for the weights, but it turns out that there is at least an analytical formula for all orthogonal polynomial weights; we will see this later.

$$f(z) = \sum_{k \geq 0} c_k T_k(z)\,. \tag{10.73}$$

By the orthogonality of the series, the coefficients are

$$c_k = \frac{2}{\pi} \int_{x=-1}^{1} \frac{f(x)T_k(x)}{\sqrt{1-x^2}}\,dx \tag{10.74}$$

unless $k = 0$ when the factor in front is $1/\pi$. If we are to do this for many functions f, and choose to use (say) 15 terms in each series, then the numerical integrations might efficiently be done if we set up Gauss–Chebyshev quadrature for each one.

Example 10.9. When we do this for $f(z) = \exp(-z)$ using (say) $n = 40$ points ξ_j, we get quite decent accuracy. The Chebyshev series that we compute are accurately evaluated by use of Algorithm 2.2. The error reported in this example is about 1.0×10^{-14}. This is similar to what Chebfun does, but not identical; there the Chebyshev–Lobatto points are used and instead of Gauss–Chebyshev quadrature, Clenshaw–Curtis quadrature, which we briefly sketch in the next section, is used.

```
function [p,c] = GaussChebyshevClenshaw( f_in, n_in, m_in )
% GAUSSCHEBYSHEVCLENSHAW Gauss-Chebyshev quadrature
%                              to find approximate Chebyshev series
% GaussChebyshevClenshaw( f, n, m )
% Find a degree m Chebyshev series expansion for f on -1 <= x <=
    1
% using n-poing Gauss-Chebyshev quadrature.
%
% Evaluate the resulting series using the Clenshaw algorithm at
    201 points
% on -1 <= x <= 1 and compare with the original function.

if nargin<2,
    n = 40;
else
    n = n_in;
end


xi = cos( pi*((1:n)-1/2)/n );
if nargin <1,
    f = exp( - xi );
else
    f = feval( f_in, xi );
end

if nargin< 3,
    m = 15; % m+1 terms in the series we compute
else
    m = m_in+1;
end

c = zeros(m+1,1);
c(1) = 1/n *sum(f);
```

```
for k=1:m,
    c(k+1) = 2/n*f*cos( k*pi*((1:n)-1/2)/n ).';
end

t = linspace(-1,1,201);

function p = ClenshawChebyshev( t, c )
% Clenshaw algorithm for evaluation of the series
m = length(c) - 1;
if m<0,
    % empty c ==> zero polynomial
    p = zeros(size(t));
elseif m==0,
    % degree 0 ==> constant polynomial
    p = c(1)*ones(size(t));
else
    % degree 1 or more, use Clenshaw algorithm
    y2 = zeros(size(t));  % y_{m+1}
    y1 = c(m+1)*ones(size(t));  % y_{m}
    %
    % p = sum( c(j)*T(j-1,t), j=1..m-1) + c(m)*T(m-1,t)
    %        + y1*T(m,t) - y2*T(m-1,t)  Loop invariant
        initialization
    for k=m:-1:2,
        % p = sum( c(j)*T(j-1,t), j=1..k-1) + c(k)*T(k-1,t) + y1*
            T(k,t)
        %  - y2*T(k-1,t)                   Loop invariant
        y0 = c(k) +2*t.*y1 - y2;   % y_{k-1}
        % p = sum( c(j)*T(j-1,t), j=1..k-1) + y0*T(k-1,t) +
        % y1*(T(k,t)-2tT(k-1,t)) [= -y1*T(k-2,t) ]
        y2 = y1;
        y1 = y0;
        % p = sum( c(j)*T(j-1,t), j=1..k-2) + c(k-1)*T(k-2,t) +
        % y1*T(k-1,t) - y2*T(k-2,t)     Proving correctness
    end
    % p = c(1)*T(0,t) + y1*T(1,t) - y2*T(0,t)
    p = c(1)-y2 + y1.*t;
end
end

p = ClenshawChebyshev( t, c );
if nargin<1,
    ft = exp(-t);
    errs = p - ft ;
else
    ft = feval(f_in,t);
    errs = p - ft;
end
figure(1), plot( t, errs, 'k'), set(gca,'fontsize',16),xlabel('t'
    ),ylabel('absolute␣error')
figure(2), plot( t, p, 'k', t, ft, 'k--'), set(gca,'fontsize',16)
    ,xlabel('t'),ylabel('Chebyshev␣series')
end
```

If $m > 1$, then a loop invariant for the Clenshaw algorithm is given by

$$p(z) = \sum_{j=0}^{k-1} c_j T_j(z) + c_k T_k(z) + y_1 T_{k+1}(z) - y_2 T_k(z) \tag{10.75}$$

for $k = m-1, m-2, \ldots, 0$. Because MATLAB does not allow indexing from 0 in arrays, that loop invariant must be translated (as it is in the comments) to index into c starting from 1. When the loop finishes with $k = 0$ so the sum is empty, $p(z) = c_0 T_0(z) - y_2 T_0(z) + y_1 T_1(z)$, which is easy to evaluate. ◁

10.5 Clenshaw–Curtis Quadrature

We have presented a lot of methods so far. Which one should you use for *your* problem? We can't answer that, of course, because that depends strongly on your context, but we now present an interesting candidate for smooth problems, namely, Clenshaw–Curtis quadrature.

The following MATLAB function, taken from Trefethen (2008a) and modified slightly to fit the format of this page, evaluates an integral in a nonadaptive fashion:

```
function I = clenshaw_curtis( f, n )
%CLENSHAW_CURTIS (n+1)-pt C-C quadrature of f
%   original program L.N. Trefethen, 2008
  x  = cos(pi*(0:n)'/n);                        % Cheby points
  fx = feval(f,x)/(2*n);                        % evaluate f
  g  = real(fft(fx([1:n+1 n:-1:2])));           % FFT
  a  = [g(1); g(2:n)+g(2*n:-1:n+2); g(n+1)];   % Cheby coeffs
  w  = 0*a'; w(1:2:end) = 2./(1-(0:2:n).^2);  % weight vector
  I  = w*a;                                     % the integral
end
```

This nice didactic program is intended to find quadratures of $\int_{-1}^{1} f(x)\,dx$ using the $(n+1)$-point formula

$$\int_{-1}^{1} f(x)\,dx \doteq \sum_{j=0}^{n} w_j f(\eta_j)\,, \tag{10.76}$$

where η_j are the $n+1$ Chebyshev–Lobatto points, namely, the *extrema* of the nth-degree Chebyshev polynomial, $\eta_j = \cos {}^{j\pi}\!/\!_{n}$ for $0 \le j \le n$, including the endpoints. The weights are as described in the penultimate line of code above and are very simple. In part, it is the simplicity of the method that recommends it first. It is actually very efficient, because it takes advantage of the FFT. It's hard to imagine a shorter program. We note that a full implementation of Clenshaw–Curtis is used, for example, as a workhorse in the MAPLE integrator and is especially useful at moderately high precision.

Example 10.10. Integrating

$$I = \int_{x=-1}^{1} x \tan x \, dx = 0.856176602730352043\ldots \tag{10.77}$$

using 10, 20, and 40 points with the Clenshaw–Curtis program above produces answers that have $I_{10}/I_{40} - 1 = -7.6 \times 10^{-7}$ and $I_{20}/I_{40} - 1 = -2.8 \times 10^{-12}$. This leads us to believe that the 40-point answer is accurate up to roundoff. Indeed, by comparison to the exact result from MAPLE (printed above), we find that I_{40} is the exactly rounded result. ◁

In fact, the method is nearly as accurate as Gaussian quadrature, and without all that bothersome computation of weights. The accuracy is something of a surprise. As the method is written, it is simply a weighted average and is exact only for polynomials of degree n, not $2n-1$ as the Gaussian rules are; but this turns out not to be a problem in a significant number of cases. Indeed on a large number of examples, it performs very nearly as accurately as Gaussian integration, and at a lower overall cost. We note that the method is *not* adaptive, as written here, but can be written in an adaptive fashion.

10.6 The Effect of Derivative Singularities and of Infinite Intervals

The quadrature of integrands whose derivatives are infinite at or near some point in the interval is quite problematic. After all, even Riemann integrals are defined only for continuous integrands, strictly speaking: Integrals of functions with discontinuities—so-called improper integrals—are defined as limits of proper integrals.

Because the error coefficients for quadrature rules involve average values of derivatives of the integrand—for example, the midpoint rule error increases with $f''(x)$—and because higher-order derivatives are more sensitive to nearby singularities, higher-order methods suffer more loss of accuracy than lower-order methods do when employed on singular integrands. There are a great many tricks to employ in trying to deal with these. Adaptive quadrature is some help; but changes of variables to eliminate the singularity, if possible, are better. As a last resort, one can try to subtract off an analytically evaluable integral with the same sort of singularity, and use numerical quadrature on the difference; this technique, sometimes called *acceleration*, is of great utility. It is used in the numerical quadrature routines in MAPLE.

Example 10.11. As an example of that, consider

$$\int_{0}^{\infty} \sin(x) \ln(x) e^{-x^3} \, dx \doteq -0.19578851584879975383905723314 6\,, \tag{10.78}$$

which has both a weak singularity at the origin and an infinite interval, although the function decays very rapidly as $x \to \infty$. However, MAPLE evaluates the integral without difficulty, numerically (the antiderivative is not expressible in terms of elementary functions). The method it uses, as stated above, is to analyze the singularity structure (by use of generalized series) and integrate singular approximations analytically. ◁

Example 10.12. For $n < 1$, consider the simple integral.

$$\int_0^1 x^{-n}\,dx = \frac{1}{1-n}\,. \tag{10.79}$$

If we ask MATLAB to compute this via `quad` for $n = 0.662672$, with tolerance $1.0 \cdot 10^{-8}$, all is well, and it returns the correct answer (with an error $2.0 \cdot 10^{-6}$, which is worse than the tolerance, but it's not bad). If we increase n to 0.662673, `quad` fails with an error message, "Warning: Minimum step size reached; singularity possible." Worse, it has taken over 600 function evaluations to tell us this.[12] This can be repaired in almost any of the ways previously mentioned; change of variables can help, which is what is done in `quadgk`, and done for strong singularities almost automatically by the new code `quadsas` available in the MATLAB Central File Exchange and described in Shampine (2010), which we believe has been incorporated into the newer MATLAB routine `integral`.

Below we try the technique of subtracting off an analytically known singularity. For example, suppose that we were really trying to integrate

$$\int_0^1 \cos(x)x^{-n}\,dx \tag{10.80}$$

for some n near 1. If we instead integrated

$$\int_0^1 \cos(x)x^{-n}\,dx = \frac{1}{1-n} + \int_0^1 (\cos(x)-1)x^{-n}\,dx\,, \tag{10.81}$$

then we have considerably better success: The integrand on the right has a much weaker (derivative) singularity at the origin. The command `quad` has no trouble with it, as

```
quad( @(x) (cos(x)-1)./x.^0.66273, 0,  1, 1.0e-8, 1 )
```

returns -0.2045. ◁

Example 10.13. Here is another example (from Kahan (1980)) where things *do* go wrong, in a bad way:

$$A(x) = \frac{1}{x}\int_0^x \frac{\sqrt{-2\log\cos u^2}}{u^2}\,du = 1 + \frac{x^4}{60} + \frac{x^8}{480} + \cdots\,. \tag{10.82}$$

[12] For functions with endpoint singularities, the routine `quadgk` is recommended instead of `quad`, which is intended for low accuracies for nonsmooth functions but with only weak singularities.

In MATLAB, we can implement this function with, for example,

```
A = @(x)  (quad(@(u)sqrt(-2*log(cos(u.^2)))./u.^2,0,x,1.0e-8))./x
```

Just as stated in Kahan (1980), this program reports answers impossibly less than 1, for small x, say 0.1 or smaller. For instance, $A(0.001)$ is reported as 0.89958, without a warning (that's the bad bit). As Kahan says, though, "Don't panic! The answers are wrong, but the [computer] is right." The problem is not with the quadrature, but with the evaluation of the function! We leave you with his article to read, but note in passing that his "cure" for this integral, namely, to write the function in the following curious way, which is mathematically but not numerically equivalent, does not quite work in MATLAB, but almost. See also problem 3.26. Define

```
B = @(x)  (quad('velvel',0,x,1.0e-8))./x,
```

where "velvel" is

```
function  [ f ] = velvel( u )
%VELVEL nasty quadrature example
% from "How to Tame A Wild Integral"
%    f(u)  = sqrt(-2*log(cos(u^2)))/u^2
y = cos(u.^2);
if y==1,
    f = 1;
else
    f  = sqrt(-2*log(y))./acos(y);
end
end
```

Then $B(0.01)$ succeeds with full accuracy [whereas $A(0.01)$ does not], but $B(0.001)$ yields an error message, complaining about a NaN—this does not happen on the old HP calculator. ◁

Infinite intervals seem to present a similar problem, but provided the decay is rapid enough, there is no difficulty when precision is high enough, and indeed the double-exponential method (also called tanh-sinh quadrature) takes advantage of the spectral accuracy of the trapezoidal rule for rapidly decaying integrals on infinite intervals to evaluate *finite interval* integrals by first transforming them to the infinite integral!

10.7 A Diversion with Chebfun

We return briefly to the simple singular integrands $\int_0^1 x^{-n}\,dx$ and $\int_0^1 \cos(x)x^{-n}\,dx$ and have a look at what Chebfun can do with them. The answer is (to us) quite shocking: it does them perfectly, even for $n = 1 - \varepsilon_M$, and blazingly fast. This beats MATLAB's built-in numerical quadrature for accuracy and robustness, and roundly tramples MAPLE's symbolic integration (which eventually returns a complicated answer to the second one involving Lommel S_1 functions) for speed. Execute the following:

```
% Quadrature of singular integrands with chebfun
% Testing by RMC 2011
x = chebfun('x',[0,1]);
a = 1-eps;
y = x.^(-a);
relerr = (1-a)*sum( y )-1
y2 = cos(x).*y;
sum(y2)
```

The relative error in the first integral is 2^{-52}, that is, one bit wrong (possibly because of the multiplication and not the quadrature), while the answer returned from the second (nonelementary) integral is correct to all bits: The relative error is zero, even though the value of the integral is larger than 10^{15}.

As of this writing, though, Chebfun is not (yet) perfect. Looking at our example integral for $W(z)/z$ (which was handled beautifully by the simple trapezoidal rule), we get the following:

```
function [ W ] = RealW( z )
%REALW Lambert W of z by chebfun integration
%   Testing chebfun quadrature
lim = pi;
v = chebfun( 'x', [-lim,lim] );
vcotv = chebfun( 'x*cot(x)', [-lim,lim], 'splitting', 'on', '
    blowup', 'on' );
vcscv = chebfun( 'x*csc(x)', [-lim,lim], 'splitting', 'on', '
    blowup', 'on' );
y = ((1-vcotv).^2 + v.^2)./( z + vcscv.*exp(-vcotv) );
W = z.*sum( y )/(2*pi);
% RealW(1) at one point yielded the error message:
%??? Error using ==> chebfun.exp at 9
%chebfun cannot handle exponential blowups
%
%Error in ==> RealW at 8
%y = ((1-v.*cotv).^2 + v.^2)./( z + v.*cscv.*exp(-v.*cotv) );

end
```

We expect that this particular case will be handled in a future release. Note also that MAPLE used to have trouble with this one too. While the essential singularities at the ends of the interval are, in fact, benign (for the integration), the fact that the singularities are there at all gives some numerical schemes “fits of anxiety.”

Example 10.14. Chebfun does very well on Kahan’s first example:

$$I_1 = \int_0^1 \left(\frac{\sqrt{u}}{u-1} - \frac{1}{\ln u} \right) du. \tag{10.83}$$

Executing

```
y = chebfun('sqrt(x)/(x-1)-1/log(x)',[0,1],'blowup','on')
sum(y)
res = ans / 0.0364899739785765205 9 - 1
```

returns $3.648997397857652 \cdot 10^{-2}$ and the residual 0. The reference answer we use is MAPLE's evaluation to 20 digits of $2 - 2\ln 2 - \gamma$. Chebfun got it correct to all places. ◁

Example 10.15. On Kahan's nastier example, Eq. (10.82), Chebfun does not do so well. The command

```
K2 = chebfun('sqrt(-2*log(cos(x^2)))/x^2', [0,1], 'blowup','on');
```

just goes away. When we try to use Kahan's trick of rewriting the function, using the `velvel` subroutine given earlier, we do somewhat better, but not perfectly. Executing

```
v = chebfun(@velvel, [0,1], 'blowup', 'on' )
```

yields a single degree-33 approximation to the function, which `cumsum`, as overloaded in Chebfun, is happy to integrate. If we then define a Chebfun by

```
a = cumsum(v)./chebfun('x',[0,1])
```

we get a degree-29 approximation, which plots very nicely: but while $a(0.01)$ is larger than 1, as it should be, all values of x that we tried at or below $x = 0.001$ give us numbers *just* below 1. For example, $a(0.001)$ yields 0.9999999999999044 and not $1 + {}^{0.001^4}\!/_{60} + \cdots$, as it should. ◁

Several other examples are explored on the Chebfun page.[13] But now, we turn to an argument *extending Kahan's impossibility proof to Chebfun*. Consider the following simple function:

```
function [ y ] = Aphra( x )
%APHRA A harmless function, that simply returns 0
%
global KingCharles;
global KingCharlesIndex;
n = length(x);
KingCharles(KingCharlesIndex:KingCharlesIndex+n) = x;
KingCharlesIndex = KingCharlesIndex + n;
y = zeros(size(x));
end
```

All it does is return 0, no matter what its input is. Well, like Aphra Behn herself,[14] this program *also* reports back to King Charles; here, she's just reporting the arguments she was called with. Now, enter the following program:

```
function [ y ] = Benedict( x )
%BENEDICT Another harmless function
%   But this function is not zero.
global KingCharles;
global KingCharlesIndex;
global Big;
s = ones(size(x));
```

[13] See the page http://www2.maths.ox.ac.uk/chebfun/examples/quad/html/BatteryTest.shtml.

[14] Aphra Behn (1640–1689) was one of the first female English playwrights. She was also a spy for Charles II, using the code name *Astrea*.

```
for i=1:KingCharlesIndex-1,
    s(:) = s(:).*(x-KingCharles(i)).^2;
end
y = Big*s;
end
```

This program, like the `Malicious` program we mentioned before, uses the results of Aphra's spying operation to define its function value. Of course, a function *could* have these zeros just by chance (admittedly a small chance), so it's not completely unreasonable. But `Benedict`'s function value is *not* identically zero—in fact, the function is zero *only* at the points known through Aphra via King Charles, and is positive everywhere else. So quite clearly, the function that Chebfun would approximate would be zero, an erroneous value. Here is how the treachery is carried out:

```
clear all
close all
global KingCharles
global KingCharlesIndex
global Big
KingCharles = zeros(10000,1);
KingCharlesIndex = 1;
A = chebfun( @Aphra, [-1,1], 'minsamples', 129 )
%Warning: Function failed to evaluate on array inputs;
    vectorizing the function may speed up its
%evaluation and avoid the need to loop over array elements. Use '
    vectorize' flag in the chebfun
%constructor call to avoid this warning message.
%> In @chebfun\private\vectorcheck at 36
%%  In @chebfun\private\ctor_adapt at 117
% In chebfun.chebfun at 204
% In treachery at 5
%A =
%   chebfun column (1 smooth piece)
%       interval          length    endpoint values
%(       -1,        1)        1          0         0
%
KingCharlesIndex
%
%KingCharlesIndex =
%
%    133
%
Big = 1.0e77;
B = chebfun( @Benedict, [-1,1], 'minsamples', 129 )
%B =
%   chebfun column (1 smooth piece)
%       interval          length    endpoint values
%(       -1,        1)        1          0         0
%vertical scale =   0
% Heh.  Same as Aphra.  She has done her job well!
t = linspace(-1,1,2012);
Bt = Benedict(t');
Bct = B(t');
plot( t', Bct, 'r--', t', Bt, 'k.' )
```

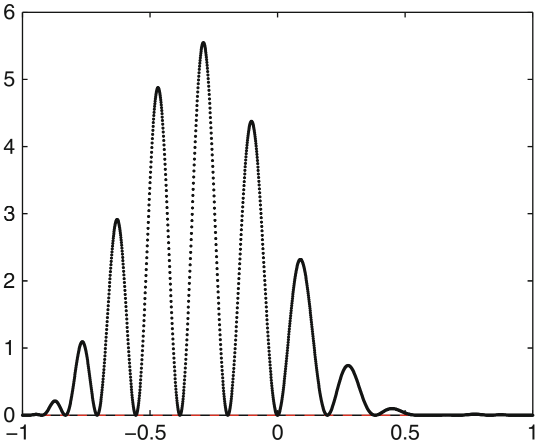


Fig. 10.5 The graph of the not-identically-zero function computed by `Benedict`, after the identically-zero function `Aphra` has reported on where Chebfun probes a zero function. The asymmetry demonstrates that Chebfun samples in an asymmetric fashion; in fact, it uses just one extra point

The result is plotted in Fig. 10.5. If the `minsamples` argument had been used, this second function would have been a higher-degree polynomial that is zero precisely at the places that Chebfun probes a zero function. However many samples are used, if the same argument is passed with `Benedict`, this will fool Chebfun into thinking that `Benedict`, like `Aphra` before him, is a cipher. This leads Chebfun into a serious error of approximation: The `chebfun` for `Benedict` is the identically zero `chebfun`, just as it is for `Aphra`. Curiously enough, `Benedict` is not symmetric on $[-1,1]$, which we find surprising.

In principle, it might happen in an application that Chebfun could be fooled this way, not through malice, as here, but simply by mischance. Any function that is passed to Chebfun might *just happen* to have its zeros exactly where Chebfun will probe it. This is unlikely and not a serious worry for real computation, although high-degree polynomial and rational functions, and exponential functions, can look remarkably constant over intervals. This is why Chebfun allows the user to override the default sampling behaviour, and to ask it to take more samples. The keyword to use is `minsamples`, as in the “spike” example in the quadratures page.[15] Of course, here, one could call `Aphra` first, with the same setting of `minsamples`, but that would be truly malicious.

[15] See the page at `http://www2.maths.ox.ac.uk/chebfun/examples/quad/pdf/SpikeIntegral.pdf`.

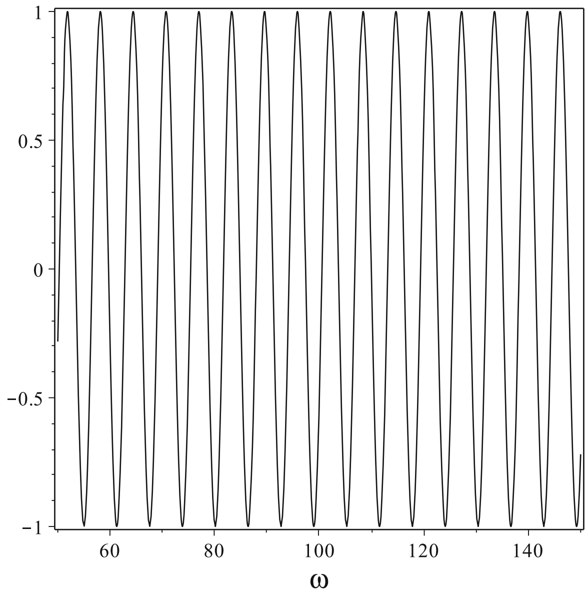


Fig. 10.6 A graph of $\omega F(\omega)$, showing clearly that $F(\omega)$ oscillates and decays like $O(1/\omega)$

10.8 Oscillatory Integrands

Oscillatory integrals, as previously mentioned, are ill-conditioned. This means that they are sensitive to data errors or to modeling errors; but if there are none of those, then they are still sensitive to numerical errors, and in effect one has to resort to semi-analytical techniques. There is an excellent survey of recent results in Iserles et al. (2006), but we will spend a little time here on an older technique known as Filon integration, which is the simplest one studied in Iserles et al. (2006).

Suppose we wish to evaluate

$$F(\omega) = \int_{-1}^{1} \frac{\cos(\omega t)}{1+t^2}\, dt\,. \tag{10.84}$$

MAPLE can find an analytical expression for this, namely,

$$\begin{aligned} F(\omega) = \tfrac{i}{2}\left(\mathrm{Ci}((1+i)\omega) - \mathrm{Ci}((1-i)\omega) + \mathrm{Ci}(-(1+i)\omega) - \mathrm{Ci}(-(1-i)\omega)\right)\cosh\omega \\ - \left(\mathrm{Si}((1+i)\omega) + \mathrm{Si}((1-i)\omega)\right)\sinh\omega\,, \end{aligned} \tag{10.85}$$

which contains hyperbolic sines, cosines, and the nonelementary sine and cosine integral functions Si and Ci. Evaluating that MAPLE expression for $\omega = 100$ turns out to be surprisingly difficult, requiring more than 100 digits of precision to draw the graph in Fig. 10.6. The difficulty is that the exponentially growing $\cosh(\omega)$

and $\sinh(\omega)$ terms present in the analytical answer cancel out, in nontrivial ways. One must use enough precision to evaluate things of size $e^{2\omega}$ and subtract them to get an answer of the size of $1/\omega$ accurately. This is an example of the third and fourth reasons we pointed out at the beginning of this chapter for doing numerical integration: Even though we have a formula, it's so unstable that it's more expensive to evaluate it than it is to do numerical quadrature!

Getting MAPLE to find an asymptotic series for that formula is also surprisingly difficult, although eventually one can use Taylor series (and a good bit of further work[16]) to find

$$F(\omega) = \int_{-1}^{1} \sum_{k=0}^{\infty} (-t^2)^k \cos(\omega t)\, dt \sim \frac{\sin\omega}{\omega} + O(\omega^{-2})\,. \tag{10.86}$$

Direct numerical integration in MAPLE for $\omega = 100$ succeeds, with about 500 function evaluations (for 200-decimal digits of answer), and the answer is accurate. For $\omega = 1000$, however, the quadrature takes about 1500 evaluations. A similar story holds in MATLAB. So this integral can be done, with standard tools, but it seems to be harder for larger ω. Indeed, MATLAB fails already for $\omega \doteq 2000$, in the sense that it hits its function evaluation count limit (this can be raised, but the fact that the limit is encountered is a sign of trouble; we're better off looking for a smarter way).

Filon[17] integration of

$$\int_{-1}^{1} f(t) \cos\omega t\, dt \tag{10.87}$$

tries to be a bit smarter. The method uses the analytically computable integrals

$$\int_{-1}^{1} \phi_{i,j}(t) \cos\omega t\, dt = \mu_{i,j}(\omega)$$

for basis polynomials $\phi_{i,j}(t)$ in much the same way that the Taylor series method did, but instead of using a Taylor series and the monomial basis to approximate the nonoscillating part of the function, it uses interpolation. The original Filon scheme used piecewise quadratic interpolation, but we (as did Iserles et al. (2006)) will use Hermite interpolation, although we will work directly in the Hermite interpolational basis. Thus, if $f(x)$ has Hermite basis coefficients $\rho_{i,j}$, that is,

$$f(t) = \sum_{i=1}^{n} \sum_{j=0}^{s_i-1} \rho_{i,j} \phi_{i,j}(t)\,,$$

[16] At least, RMC made heavy going of it, being led through Lommel S_1 functions and having to sum an infinite series just to get the leading coefficient. One hopes there is an easier way.

[17] Louis Napoleon George Filon (1875–1937) was an English applied mathematician (but yes, born in France). See his Wikipedia entry for more details. One of us is happy to claim distant cousinship. Of course, it's true also for the other of us, but probably even more distantly.

then

$$\int_{-1}^{1} f(t)\cos\omega t\,dt \doteq \sum_{i=1}^{n}\sum_{j=0}^{s_i-1} \rho_{i,j}\mu_{i,j}\,. \tag{10.88}$$

Precomputing the $\mu_{i,j}$, called the *moments*, allows one to evaluate integrals remarkably well, and it turns out that the accuracy increases as ω increases. The difficulty with cancellation is taken care of largely analytically, or even in the case of high-precision computation of the moments, seminumerically.

What follows is an idiosyncratic approach to Filon integration, specifically aimed at this example. The detailed technique (which is our own variation on the standard Filon integration) will probably work only for oscillatory integrands without stationary points—the easy cases, in other words. In fact, we have only tried it for the regular case of $\int f(t)\exp(ig(t))\,dt$, where $g(t) = t$. But we like the approach, because it shows off differentiation matrices (which we study in the next chapter) and Hermite interpolation and the method of undetermined polynomials, and is quite remarkably effective. We plan to investigate extensions of this, in the near future. But for now, let us consider only this simplest case.

The first thing to note is that the integral of a polynomial times a cosine is a polynomial combination of a sine and cosine:

$$\int p(t)\cos\omega t\,dt = P(t)\cos\omega t + Q(t)\sin\omega t\,. \tag{10.89}$$

It is true for the integral of $p(t)\sin\omega t$ as well. One can see that this must be so by integration by parts. However, we do not choose to use integration by parts here, or indeed the monomial basis at all. Instead, we differentiate both sides and equate trigonometric coefficients to find

$$\begin{aligned} 0 &= \omega P(t) - Q'(t) \\ p(t) &= P'(t) + \omega Q(t)\,. \end{aligned} \tag{10.90}$$

These (basis-free!) equations are equivalent to $\omega p(t) = (D^2 + \omega^2)Q(t)$. Notice that the degree of P is one less than the degree of Q, by the first equation, and that therefore the degree of Q is the same as the degree of p. Thus, if we interpolate p with enough information to recover it completely, then the corresponding information is also enough to specify both Q and P. Therefore, we may use the same set of nodes and confluencies to describe all three polynomials.

Finally, notice that if we know what P and Q are, then we can evaluate

$$\int_{-1}^{1} p(t)\cos\omega t\,dt = (P(1) - P(-1))\cos\omega + (Q(1) + Q(-1))\sin\omega \tag{10.91}$$

by the fundamental theorem of calculus—that is, we can analytically evaluate this kind of integral. Of course, our example has $f(t) = {}^{1}/_{(1+t^2)}$, which is not a polynomial of any sort, and so we may not evaluate this integral analytically. But what happens if we interpolate $f(t)$ at some convenient points? For example, take $\tau = [-1,0,1]$ and confluency $s = 2$. We then must solve the Eq. (10.90), which we do *without converting to any other basis* by use of the differentiation matrix for these nodes and this confluency. We examine this in more detail in Chap. 11; for now, we can use this code:

```
% A script to do Filon quadrature of int( f(t)*cos(omega*t), t =-1..1 )
%
% Hermite interpolation of f, differentiation matrices, and the
% method of undetermined polynomials are used:
% int( p*cos(omega*t), t ) = P*cos(omega*t) + Q*sin(omega*t)
% where Q'-omega*P=0 and P'+omega*Q=p.
%
% RMC December 30, 2010
%
f = @(t) 1.0./(1+t.^2);
df= @(t) -2.0*t./(1+t.^2).^2;
%f = @(t) 1 + 2*t + 3*t.^2 + 4*t.^3 + 5*t.^4;
%df= @(t) 2 + 6*t + 12*t.^2 + 20*t.^3;
tau = [-1,0,1];
%tau = cos( pi*(0:3)/3 );
%tau = [-1, -1/2, 1/2, 1];
[gam,D] = genbarywts(tau,2);
rho = [f(tau); df(tau)];
p = rho(:);
w = 200.3; % omega=2003, Maple = -0.485623502941671423e-3, quad fails
A = D^2 + w^2*eye(size(D)); % D^2
Q = A\(w*p);
P = D*Q/w;
% use the fact that tau(1)=-1 and tau(end)=1
Filon = (P(end-1)-P(1))*cos(w) + (Q(end-1)+Q(1))*sin(w)
% Now try quad, and see how it compares
osc = @(t) f(t).*cos(w*t);
qans = quad(osc,tau(1),tau(end),1.0e-9);
reler = Filon/qans-1
```

For the parameters in this code, the extremely cheap Filon integration technique gets an answer accurate to 4 places. If ω is increased to 2013, the Filon procedure (just exactly as cheap as before) gets *six* figures of accuracy—it is better for higher oscillations!—whereas quad fails with the message "Warning: Maximum function count exceeded; singularity likely." MAPLE needs *2,000-decimal digit arithmetic* to evaluate its analytical answer for $\omega = 2003$, and takes a great deal of time. If instead we take $\omega = 20,000$, then MAPLE needs *more than 9,000-digit* arithmetic (we got it to succeed with 18,000-digit arithmetic, but didn't look for the exact breaking point) to do so, whereas the Filon integration program above works just fine in IEEE double precision, getting $F(\omega) = -2.306441272337951 \times 10^{-5}$, which is accurate to about 10^{-11}.

So what about the ill-conditioning? We *proved* that a small change in the integrand would make a large change, a factor ω at least, in the integral! The trick is that small change had to be very high frequency, although it could have a tiny amplitude. If we make small *low*-frequency changes, then those errors are also essentially smoothed out by the integral of the oscillation. The trick is to arrange your numerical procedure to do just that. We have done so here by providing the exact integral of a function that differs from the original but has exactly the same oscillatory behavior. In some sense, then, we have only allowed changes to the integrand that did not affect the oscillation—it was, in other words, a method that gave a good *structured* backward error.[18]

Remark 10.3. The approach to Filon integration first discussed, namely precomputing the moments, can be done numerically. However, it is analogous to computing a full matrix inverse, and if only one or a few integrals need to be done for each ω, then the approach described second seems more efficient. ◁

10.9 Multidimensional Integration

High-dimensional quadrature is the "Mount Everest" of integration. It has various applications, particularly in financial mathematics. In this section, we confine ourselves to trying to show what the difficulty is. The difficulty is called "The Curse of Dimension," and in essence it becomes harder and harder to sample the domain of the integral with enough smarts to find the places where the integrand contributes something to the integral.

Suppose first that we are integrating $f(x_1,x_2,\ldots,x_d)$ over a d-dimensional cube,[19] $0 \leq x_k \leq 1$, for each variable x_k for $k = 1, 2, \ldots, d$. If we sample our integrand on a simple tensor-product grid that is at N_1 grid-lines in x_1, N_2 in x_2, and so on, then obviously we must choose $N = N_1 N_2 \cdots N_d$ points in all to evaluate our function. If each $N_k = n$, this is $N = n^d$ points. If $n = 1000$ and $d = 2$, this is only a million points, which is not so bad, and indeed bivariate quadrature is considered fairly easy (see `quad2d` in MATLAB). With $d = 3$, again while it is computationally intensive, there is much that can be done (see `triplequad` in MATLAB). If $n = 100$, and $d = 4$, this is 10^8 points, which is getting pretty bad. This is, in fact, an exponential cost increase as d increases. If d is several *hundred*, as it can be, then this is clearly hopeless. Tensor product grids must be abandoned.

This is not to give up hope. MAPLE uses ACM TOMS Algorithm 698 in its multiple integration as described in Berntsen et al. (1991) and claims to be able to handle dimensions up to 15. There are other alternatives, and some very promising work on lattice rules (see Cools et al. 2006). But this is a very hard problem indeed. We end this chapter by pointing you at the vast literature, and in particular at the

[18] There's a lot more to be said about oscillatory integrals, and we leave you with the survey by Iserles et al. (2006).

[19] The other major difficulty of higher-dimensional integration is the geometry of the region itself. Here we mention only rectangles or cubes, but there are many other shapes of practical interest, some of which can be mapped to a cube. Not only is the boundary then of concern, but the location of singularities on or near the boundaries becomes much more problematic.

work of Sloan and Joe (1994), Sloan and Wozniakowski (2001), and Hickernell and Wozniakowski (2001). Perhaps the most useful single reference is Kuo and Sloan (2005).

10.10 Notes and References

For an elementary but thorough introduction to the various tricks for using numerical methods on singular integrands, consult Kahan (1980). The advanced semi-analytical methods of quadrature used in MAPLE are described in Geddes and Fee (1992). Please see the beautiful and informative introductory paper Weideman (2002), which does a detailed asymptotic analysis of the error and gives many clarifying examples for spectral accuracy. More details of methods like these can be found in Bailey et al. (2005). Waldvogel (2011) discusses the spectral accuracy of the trapezoidal rule in depth. The notion of equidistribution of error is old in numerical analysis. It can be found, for example, in de Boor (1978) in a discussion of optimal placement of nodes for linear interpolation.

For an excellent survey of the current state of the art in adaptive quadrature, see Gonnet (2010). There are a lot of adaptive methods implemented in MATLAB, not just `quad`; there is also `quadl`, `quadgk`, and a vectorized version `quadv` (see Shampine 2008b). There is also the newer code `integral`. There is a great interest in comparing methods; see, for example, Gonnet (2010) for a discussion of adaptive methods, and Trefethen (2008a) for a comparison of Gaussian quadrature with Clenshaw–Curtis quadrature and a beautiful geometric explanation of its behavior. Wilf (1962) has (among many other useful things) a complete discussion of Gaussian quadrature and computation of nodes by eigenvalues of the symmetric tridiagonal so-called Jacobi matrix and the weights by either of a pair of general formulæ: In our current notation,

$$w_i = \frac{c_n}{c_{n-1}}\left[\frac{1}{\phi_{n-1}(\tau_i)\phi_n'(\tau_i)}\right] = \frac{1}{\sum_{k=0}^{n-1}\phi_k^2(\tau_i)}\,, \tag{10.92}$$

where $c_n = [z^n]\phi_n(z)$ is the leading coefficient of the nth orthogonal polynomial. The symmetric tridiagonal Jacobi matrix mentioned above and treated in Golub and Welsch (1969) is closely related to the companion matrix pencil for the polynomial $\phi_n(z)$ in the orthogonal basis $\{\phi_k(z)\}$. If we use the three-term recurrence relation, which we write as

$$z\phi_n(z) = s_n\phi_{n+1}(z) + u_n\phi_n(z) + v_n\phi_{n-1}(z)\,, \tag{10.93}$$

with the special case $z\phi_0(z) = u_0\phi_0(z) + s_0\phi_1(z)$, then it is clear that

$$z\begin{bmatrix}\phi_0(z)\\ \phi_1(z)\\ \vdots\\ \phi_{n-1}(z)\end{bmatrix} = \begin{bmatrix} u_0 & s_0 & & & \\ v_1 & u_1 & s_1 & & \\ & v_2 & u_2 & s_2 & \\ & & \ddots & \ddots & \ddots \\ & & & v_{n-1} & u_{n-1} & s_{n-2}\end{bmatrix}\begin{bmatrix}\phi_0(z)\\ \phi_1(z)\\ \vdots\\ \phi_{n-1}(z)\end{bmatrix} \tag{10.94}$$

because $z\phi_{n-1}(z) = s_{n-1}\phi_n(z) + u_{n-1}\phi_{n-1}(z) + v_{n-1}\phi_{n-2}(z)$ has only two terms in it if z is a zero of $\phi_n(z)$. Thus, roots of $\phi_n(z)$ are eigenvalues of this matrix. With this derivation, the Jacobi matrix is not usually symmetric, but it can always be symmetrized by a positive diagonal similarity transformation, and we presume this has been done. Thus, the zeros of $\phi_n(z)$ are the (necessarily real) eigenvalues of a symmetric tridiagonal matrix.

For a discussion of tanh-sinh integration, see Bailey et al. (2005), the original papers by Takahasi and Mori (1974), and the review Mori and Sugihara (2001). That these methods do not always work in IEEE double precision is discussed in Shampine (2010) but is of less importance in the arbitrary-precision works previously cited.

Numerical integration can be important in rootfinding for analytic functions by using the residue theorem on

$$N = \frac{1}{2\pi i}\oint_C \frac{f'(z)}{f(z)}\,dz\,, \tag{10.95}$$

where N counts the number of zeros of $f(z)$ inside the contour C. This is the basic method used by MAPLE's `RootFinding[Analytic]` command. This idea is also used in localized eigenvalue computation, for example in the paper Sakurai and Sugiura (2003).

A MAPLE implementation of `Aphra` and `Benedict` (called `spy` and `malicious`) can be found in Corless (2002).

Problems

Theory and Practice

10.1. Show that if $f(x)$ is convex up on an interval $a \le x \le b$, that is, $f(x)$ is above its tangent lines, then the midpoint rule gives a lower bound on the area $\int_a^b f(x)\,dx$, and that the trapezoidal rule gives an upper bound (this reverses, for convex down). Assume $f(x)$ is sufficiently differentiable. Hint: Draw the tangent line to $f(x)$ at the midpoint, and use similar triangles to compare the area under the trapezoid—with that tangent line as top—with the area predicted by the midpoint rule.

10.2. Use the midpoint rule, the trapezoidal rule, and Simpson's rule to estimate the area under $f(x) = x\tan x$, above the x-axis, and between $x = 0$ and $x = {}^{\pi}\!/4$. Use enough panels to guarantee six-decimal-place accuracy.

10.3. Show that the error in the composite Simpson's rule is ${}^{(b-a)h^4}\!/90$ times an average fourth derivative of the function being integrated.

10.4. Find (by brute force) real numbers τ_1, τ_2, w_1, and w_2 so that

$$\int_0^1 f(x)\,dx \doteq w_1 f(\tau_1) + w_2 f(\tau_2) \tag{10.96}$$

is exact for $f(x) = 1$, x, x^2, and x^3. What is the error for $f(x) = x^4$? This is the beginning of the Gaussian quadrature story.

10.5. Bessel functions have integral representations, for example,

$$J_0(x) = \frac{1}{\pi}\int_0^{\pi} \cos(x\sin(t))\,dt\,. \tag{10.97}$$

Use a quadrature program to evaluate $J_0(1)$, and to draw a graph of $J_0(x)$ on $0 \leq x \leq 10$. Use `fzero` to identify the first positive zero of $J_0(x)$. Compare your routine with the built-in `besselj`.

10.6. Suppose $F(\tau_1) = \rho_{1,0}$, $F'(\tau_1) = \rho_{1,1}$, and $F'(\tau_2) = \rho_{2,1}$. What is $F(\tau_2)$?

10.7. Using a computer algebra system (if you like) to do the partial fraction decomposition, find an integration formula for

$$\int_{\tau_1}^{\tau_3} F'(\tau)d\tau \tag{10.98}$$

that uses samples at τ_1, τ_2 between, and τ_3. How high a degree $F(\tau)$ is your formula exact for?

10.8. The monomial basis is, as we all know, convenient for differentiation of polynomials. Chapter 11 will show how to differentiate a polynomial, using a differentiation matrix. Integration of polynomials is also convenient in the monomial basis. How about Lagrange or Hermite? Invent a formula that allows you to easily evaluate

$$\int_{\tau_1}^{t} p(\tau)d\tau$$

when you know Hermite data at τ_1 through τ_n, with confluencies $s_k \geq 1$. Hint: Use the contour integral approach on a small example, first. For instance, you should be able to show that the formula

$$\int_{-1}^{1} f(x)\,dx = f(1) + f(-1) + \frac{1}{3}\left(f'(1) - f'(-1)\right) \tag{10.99}$$

is exact for polynomials of degree 3 or less. We remark that these formulæ aren't all that common, probably because differentiation of the integrand is more work, and you might as well just use more function evaluations.

10.9. Use `CompanionMatrix` in MAPLE (or work out using the recurrence relation which you will have to look up) and symmetrization to find a symmetric tridiagonal eigenvalue problem whose zeros are the roots of the polynomial $C_5^{1/4}(z)$, the fifth Gegenbauer C polynomial with parameter $^1\!/\!_4$. These are orthogonal on $[-1,1]$ with respect to the weight function $(1-z^2)^{a-1/2}$. These polynomials can be used for Gaussian quadrature with this weight function.

10.10. Work out the Gaussian quadrature nodes and weights for $n = 10$ with the weight function $w(z) = \exp(-z^2)$ on the interval $(-\infty,\infty)$.

10.11. Use Filon integration to evaluate

$$G(\omega) = \int_0^1 \frac{\sin \omega t}{1+t^2}\,dt \tag{10.100}$$

for large values of ω, say $\omega = 2013$.

10.12. Try Filon integration on the example given but with different nodes, specifically `tau=cos(pi*(0:n)/n)` for different n and large values of ω.

10.13. Find an optimal mesh for evaluating $\int_0^1 \sqrt{1-\sin^4(\pi x)}dx$ using the midpoint rule and 10 panels (11 nodes). Compare with the mesh found by `integral`.

10.14. Show that

$$\frac{7-4z^2}{(z+1)^2(z+1/2)^2(z-\frac{1}{2})^2(z-1)^2} = \frac{4/3}{(z+1)^2} + \frac{32/3}{(z+1/2)^2} + \frac{32/3}{(z-1/2)^2} + \frac{4/3}{(z-1)^2} + \frac{12}{z+1} - \frac{12}{z-1}$$

and that therefore

$$\int_{-1}^{1} f(z)dz = \frac{1}{9}f(-1) + \frac{8}{9}f(-1/2) + \frac{8}{9}f(1/2) + \frac{1}{9}f(1)$$

for all polynomials f of degree at most 3.

Investigations and Projects

10.15. Try to use the integral definition (1.22) of the Airy function and a quadrature routine to evaluate $\mathrm{Ai}(x)$ for several x, say $x = 1.2$ to start. Which routine, if any, is best? The MATLAB routine `quadgk` is supposed to be able to handle infinite intervals. Is this a hard integral to do? If so, why? See the discussion in Gil et al. (2002) for an alternative approach.

10.16. Using Filon integration on $\int_a^b f(t)\cos \omega t + g(t)\sin \omega t\, dt$ by the method of the text leads to the linear system

$$\begin{bmatrix} \omega\mathbf{I} & \mathbf{D} \\ \mathbf{D} & -\omega\mathbf{I} \end{bmatrix} \begin{bmatrix} Q \\ P \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix} . \tag{10.101}$$

The antiderivative is denoted $P(t)\cos\omega t + Q(t)\sin\omega t$. Show that two steps of Jacobi iteration gives the solution

$$\begin{aligned} Q &\doteq \frac{f}{\omega} + \frac{\mathbf{D}g}{\omega^2} \\ P &\doteq -\frac{g}{\omega} + \frac{\mathbf{D}f}{\omega^2} \end{aligned} \tag{10.102}$$

and that this solution has residual $O(1/\omega^3)$ as $\omega \to \infty$. This incidentally shows why an earlier version of the code of Sect. 10.8, which had a bug in the creation of the matrix **D**, got good answers anyway for large ω.

10.17. Investigate the MATLAB routines for double integration and triple integration.

10.18. Larry Shampine pointed out after reading a draft that this chapter really doesn't talk much about the critical issue of estimating the error in a computed numerical integration. We talk about equidistribution, but not of what. One possible method is to compare the result of one quadrature method with that of another (possibly the same one with a refined mesh). Discuss.

Chapter 11
Numerical Differentiation and Finite Differences

Abstract Taking derivatives of numerical functions is one of the most often performed tasks in computation. *Finite differences* are a standard way to approximate the derivative of a function, and *compact finite differences* are especially attractive. We study the *conditioning of differentiation*, including some *structured condition numbers* for differentiation of polynomials. We look at *differentiation matrices* for derivatives of polynomials expressed in a Lagrange or Hermite interpolational basis. We look at *regularization or smoothing* before taking derivatives, and briefly touch on *automatic differentiation*. ◁

Numerical differentiation can be described in nearly the same terms as we described quadrature, simply by replacing three words: The basic idea of numerical ~~quadrature~~ differentiation is to *replace* $f(x)$ with a slightly different function, call it $f(x)+\Delta f(x)$ or $(f+\Delta f)(x)$, and ~~integrate~~ differentiate the second function instead. This is our *engineered problem* (see Chap. 1). We will choose Δf so that it's not too large, and so that $f+\Delta f$ is simple to ~~integrate~~ differentiate exactly.

As we did in the previous chapter for integration, here we examine the conditioning of differentiation, and we examine some methods to numerically differentiate functions from the backward error point of view.

11.1 Conditioning

We begin this chapter on the numerical computation of derivatives with an answer to what, by now, should be a familiar question: What are the effects of data or modeling error on the computation of derivatives?

Theorem 11.1. *Differentiation is infinitely ill-conditioned.*

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_11,

Proof. Suppose $f(z)$ has derivative $f'(z)$ and that $\Delta f(z) = \varepsilon \cos \omega z$. Then

$$\frac{d}{dz}(f(z) + \Delta f(z)) = f'(z) + (\Delta f)'(z) = f'(z) - \varepsilon\omega \sin \omega z$$

and the *absolute condition number*

$$\frac{\|(\Delta f)'(z)\|}{\|\Delta f(z)\|} = \omega \tag{11.1}$$

can be arbitrarily large. ♮

In one sense this is a trivial theorem; of course, a change in f that is small but wiggly will change f' a lot.

In another sense, however, this theorem is not so trivial because every computer-instantiated function over $\mathbb{R}$ is *locally constant*: That is, $\hat{f}' = 0$ almost everywhere. So, computers (at their finest scale) do exactly this, replacing f by $f + \Delta f$, where $\|\Delta f\|$ is small but where $\|\Delta f'\|$ is not small. Let us put it another way. Consider

$$\hat{f}'(z) := \lim_{h \to 0} \frac{\hat{f}(z+h) - \hat{f}(z)}{h} . \tag{11.2}$$

If $|h| < \mu_M |z|$, then $z \oplus h \equiv z$ and thus $\hat{f}'(z) = 0$. This is quite reminiscent of another of Zeno's paradoxes; here we have a curve, but everywhere its derivative is zero—well, everywhere the derivative is defined (and it is only undefined at the boundaries between rounding up or down to adjacent machine numbers). See Fig. 11.1. Nonetheless we're going to try, anyway, to see if we can find $f'(z)$

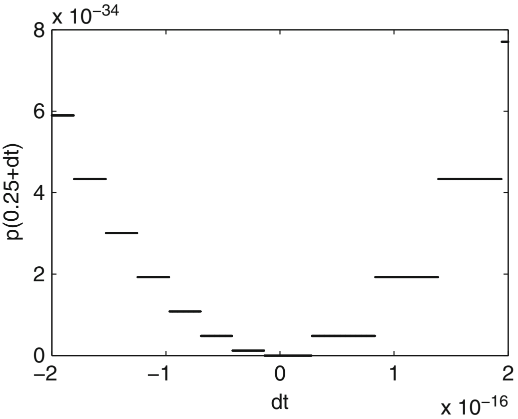


Fig. 11.1 Functions are flat when you zoom in close enough. Here, the function of Example 8.6 is plotted very near a double zero $z = 1/4$. Evaluation is by the second barycentric form. One sees that at this fine a scale, the approximation is locally flat, so its derivative is zero wherever it is defined

approximately, without running into $\hat{f}'$. One thing that helps is the idea of structured condition number, which we take up after we study differentiation matrices.

11.2 Polynomial Formulæ and Differentiation Matrices

The traditional way to begin is by differentiation of polynomials, and it's a good way. However, because of our emphasis in previous chapters, we begin by differentiating polynomials expressed in the Lagrange basis.

We give an example first and then a general theorem. Suppose that $f(z)$ is given by $\boldsymbol{\rho} = [1, -1, 1, -1]$ on $\boldsymbol{\tau} = [-1, -1/3, 1/3, 1]$. What is $f'(-1)$? To find the answer, form the integral

$$0 = \frac{1}{2\pi i}\oint_C \frac{f(z)}{(z+1)^2(z+1/3)(z-1/3)(z-1)}dz$$

about a contour C that encloses all of the nodes τ_k. The integral is then zero for all polynomials $f(z)$ of degree 3 or less. Expanding in partial fractions and using the residue theorem as usual, we have

$$0 = \frac{1}{2\pi i}\oint_C \frac{-9/16}{(z+1)^2}f(z) - \frac{99/64}{z+1}f(z) + \frac{81/32}{z+1/3}f(z) - \frac{81/64}{z-1/3}f(z) + \frac{9/32}{z-1}f(z)dz\,,$$

from which we obtain

$$-\frac{9}{16}f'(-1) - \frac{99}{64}f(-1) + \frac{81}{32}f(-\frac{1}{3}) - \frac{81}{64}f(\frac{1}{3}) + \frac{9}{32}f(1) = 0\,. \tag{11.3}$$

Solving for $f'(-1)$, we find

$$\begin{aligned} f'(-1) &= -\frac{11}{4}f(-1) + \frac{9}{2}f(-\frac{1}{3}) - \frac{9}{4}f(\frac{1}{3}) + \frac{1}{2}f(1) \\ &= -10\,. \end{aligned} \tag{11.4}$$

This type of formula—which expresses f' at a point in terms of function values at other points, and is exact in exact arithmetic for polynomials of a certain degree or less (here, 3 or less)—is called a *finite difference*.

Indeed, if we take the cases $f(x) \equiv 1$, $f(z) = z$, $f(z) = z^2$, and $f(z) = z^3$, we obtain, respectively,

$$f'(-1) = -\frac{11}{4} + \frac{9}{2} - \frac{9}{4} + \frac{1}{2} = 0\,, \tag{11.5}$$

$$f'(-1) = \frac{11}{4} + \frac{9}{2}\left(-\frac{1}{3}\right) - \frac{9}{4}\left(\frac{1}{3}\right) + \frac{1}{2} = 1\,, \tag{11.6}$$

$$f'(-1) = \frac{11}{4} + \frac{9}{2}\left(\frac{1}{9}\right) - \frac{9}{4}\left(\frac{1}{9}\right) + \frac{1}{2} = -2\,, \tag{11.7}$$

$$\text{and} \qquad f'(-1) = \frac{11}{4} - \frac{9}{54} - \frac{9}{108} + \frac{1}{2} = 3\,, \tag{11.8}$$

as it should be. However, for $f(z) = z^4$, we obtain $-20/9$, not -4; this is as expected, because our formula was only guaranteed to be correct for polynomials of degree 3 or less. Notice that at no time have we used the monomial basis, except in our check (and even there we evaluated our test functions at the nodes).

Quite clearly, we can use a similar trick to find $f'(-\frac{1}{3}), f'(\frac{1}{3})$, and $f'(1)$. When we do, we get

$$\begin{aligned}
f'(-\frac{1}{3}) &= -\frac{1}{2}f(-1) - \frac{3}{4}f(-\frac{1}{3}) + \frac{3}{2}f(\frac{1}{3}) - \frac{1}{4}f(1)\\
f'(\frac{1}{3}) &= \frac{1}{4}f(-1) - \frac{3}{2}f(-\frac{1}{3}) + \frac{3}{4}f(\frac{1}{3}) + \frac{1}{2}f(1)\\
f'(1) &= -\frac{1}{2}f(-1) + \frac{9}{4}f(-\frac{1}{3}) - \frac{9}{2}f(\tfrac{1}{3}) + \frac{11}{4}f(1)
\end{aligned}$$

and we now have the values of the derivative of $f(z)$ at all four nodes. This can be arranged in matrix form:

$$\begin{bmatrix} f'(-1) \\ f'(-1/3) \\ f'(1/3) \\ f'(1) \end{bmatrix} = \begin{bmatrix} -11/4 & 9/2 & -9/4 & 1/2 \\ -1/2 & -3/4 & 3/2 & -1/4 \\ 1/4 & -3/2 & 3/4 & 1/2 \\ -1/2 & 9/4 & -9/2 & 11/4 \end{bmatrix} \begin{bmatrix} f(-1) \\ f(-1/3) \\ f(1/3) \\ f(1) \end{bmatrix}. \tag{11.9}$$

At this point, we may use the computed derivative values at the nodes to compute the value of the derivative f' at *any* point z, by using the second barycentric form of the Lagrange interpolant:

$$f'(z) = \frac{\displaystyle\sum_{i=0}^{3} \frac{\beta_i f'(\tau_i)}{z-\tau_i}}{\displaystyle\sum_{i=0}^{3} \frac{\beta_i}{z-\tau_i}}\,. \tag{11.10}$$

The degree of this polynomial is necessarily one less than the degree of $f(z)$, but that doesn't bother the barycentric formula: It still costs $O(n)$ flops to evaluate, which is a bit too expensive (we could drop one node and update the others if we had to evaluate the derivative many times), but this is not very important. Thus, computation of the derivatives at the nodes then allows us to evaluate the derivative *anywhere*.

Perhaps surprisingly, this technique is effective in many circumstances. The contour integral approach need only be done theoretically to construct formulæ for the entries of the *differentiation matrix* $\mathbf{D} = [D_{ij}]$, which appears in (11.9) above, where

$$f'(\tau_i) = \sum_{j=0}^{n} D_{ij} f(\tau_j)\,. \tag{11.11}$$

Each row of the matrix $\mathbf{D}$ comes from the coefficients in the corresponding finite-difference formula.

Example 11.1. Suppose that $\boldsymbol{\tau} = [-1, -1/2, 1/2, 1]$ (which is slightly different than the introductory example) and, once again, $\boldsymbol{\rho} = [1, -1, 1, -1]$. Then the differentiation matrix is

$$\mathbf{D} = \begin{bmatrix} -19/6 & 4 & -4/3 & 1/2 \\ -1 & 1/3 & 1 & -1/3 \\ 1/3 & -1 & -1/3 & 1 \\ -1/2 & 4/3 & -4 & 19/6 \end{bmatrix} .$$

This can be computed with the MATLAB routine `genbarywts` or the MAPLE routine `bhip`, both of which are available in the code repository for this book. We will shortly see the algorithm that those codes use to generate $\mathbf{D}$. Applying this matrix to $[1, -1, 1, -1]^T$ produces the vector $[-9, 0, 0, -9]^T$. ◁

In general, a *differentiation matrix* is a matrix $\mathbf{D}$ such that

$$\mathbf{D} \cdot \mathbf{f} = \mathbf{f}' , \tag{11.12}$$

where $\mathbf{f}$ is a vector of *polynomial coefficients* and $\mathbf{f}'$ is a vector of coefficients for the derivative of that polynomial in the same basis. In the case where we are working in the Lagrange basis, the polynomial coefficients are just the values at the nodes. In the Hermite basis, the coefficients are the values and derivatives at the nodes. The notion of differentiation matrix is wholly independent of the basis.

In the monomial basis, differentiation matrices are, of course, particularly simple. If $f(z) = a_0 + a_1 z + a_2 z^2 + a_3 z^3$, and $f'(z) = b_0 + b_1 z + b_2 z^2 + b_3 z^3$, then

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} . \tag{11.13}$$

The generalization to the $n \times n$ case is immediately evident. Several features of this matrix are important. First, it is singular, because differentiation of a constant polynomial gives zero. Second, its singular values are just 0, 1, 2, ..., n.

Quite clearly, we could use instead an $(n-1) \times n$ matrix and express $f'(z)$ as a degree-$(n-1)$ polynomial, because the monomials unlike the Lagrange basis are a degree-graded basis. Indeed, there is no particular reason save convenience to express the derivative in the same basis as the original function.[1] Using this, and the definitional matrix $\mathbf{B}$ for the basis $\phi_k(z)$ from Eq. (2.2) connecting $\boldsymbol{\phi}$ to the monomial basis, we can write a formula for the differentiation matrix in any basis:

$$\mathbf{D}_\phi = \mathbf{B}^{-T} \mathbf{D}_{\text{monomial}} \mathbf{B}^T . \tag{11.14}$$

[1] Olver and Townsend (2013) adopt the technique of using a different basis for the derivative $f'(t)$ than for the function $f(t)$, which allows a diagonal differentiation matrix, attaining great efficiency.

In practice, this is too cumbersome (and in the case of Lagrange and Hermite interpolational bases, these involve inversions of ill-conditioned Vandermonde matrices). Special formulæ are advisable.

Coming back to the Lagrange basis, it turns out to be more numerically stable to treat D_{ii} (the diagonal elements) to preserve the following: When $f(z) \equiv 1$,

$$\sum_{j=0}^{n} D_{ij} \cdot 1 = 0\,, \tag{11.15}$$

and so one could isolate the diagonal entries to give

$$D_{ii} = -\sum_{\substack{j=0 \\ j \neq i}}^{n} D_{ij}\,. \tag{11.16}$$

This formula, which comes out naturally in the derivation below, is not as simple as some other formulæ in print; but as stated, it is more numerically stable than some of those "simpler" formulæ, which we don't give here.

11.2.1 Structured Condition Number for Differentiation of Polynomials

We saw that differentiation was in general arbitrarily ill-conditioned: The difference between $(f+\Delta f)'$ and f' can be arbitrarily large even if Δf is itself small. But what about the case in which f and $f+\Delta f$ are *both* polynomials? In this case, the absolute condition number turns out to be *bounded.*

Theorem 11.2. *If f and Δf are both polynomials of degree at most d expressed in a basis $\boldsymbol{\phi}$ defined in terms of the monomial basis by a matrix $\mathbf{B}$ and the vector of coefficients of Δf is called $\Delta\mathbf{a}$, then the norm of the vector of coefficients of $\Delta f'$ (call it $\Delta\mathbf{b}$) is bounded by*

$$\|\Delta\mathbf{b}\| \leq \|\mathbf{D}_\phi\|\, \|\Delta\mathbf{a}\|\,. \tag{11.17}$$

Moreover, the norm of the vector of coefficients of $\Delta f'$ expressed in the monomial *basis is bounded by $d\kappa(\mathbf{B})$.*

Proof. These follow directly from the discussion above. ♮

Remark 11.1. The main improvement over the general case is the reduction of this absolute condition number from infinite to something finite. In the cases where the change-of-basis matrix $\mathbf{B}$ is unitary, we will have an explicit bound d. However, the *relative* condition number is still infinite: f' can be zero (and must be if f is constant, in any basis). Small errors in differentiating constants will therefore be particularly detectable. This is the reason for insisting that Eq. (11.16) hold: It enforces exact differentiation of machine-representable constant vectors. ◁

11.2.2 Differentiation Matrices for Lagrange Bases

To find formulæ for the entries D_{ij} of the differentiation matrix $\mathbf{D}$ for a Lagrange basis, we apply the contour integral approach as before:

$$\frac{1}{w(z)(z-\tau_i)} = \frac{1}{(z-\tau_i)}\sum_{j=0}^{n}\frac{\beta_j}{(z-\tau_j)}\,, \tag{11.18}$$

where we have used the partial fraction expansion for ${}^{1}\!/\!{}_{w(z)}$ to give us the β_js. Notice that the $(z-\tau_i)$ term is repeated—it also occurs for one of the terms in the sum. We may rewrite that as

$$\begin{aligned}\frac{1}{w(z)(z-\tau_i)} &= \frac{\beta_i}{(z-\tau_i)^2} + \sum_{\substack{j=0\\ j\neq i}}^{n}\frac{\beta_j}{(z-\tau_i)(z-\tau_j)}\\ &= \frac{\beta_i}{(z-\tau_i)^2} + \frac{1}{(z-\tau_i)}\sum_{\substack{j=0\\ j\neq i}}^{n}\frac{\beta_j}{(\tau_i-\tau_j)} + O(1)\,,\end{aligned} \tag{11.19}$$

where we have done a Laurent expansion at $z=\tau_i$ to give that last line. Series expansion about each $z=\tau_j$ for $j\neq i$ gives the remaining terms as

$$\frac{1}{w(z)(z-\tau_i)} = \frac{\beta_i}{(z-\tau_i)^2} + \frac{1}{(z-\tau_i)}\sum_{\substack{j=0\\ j\neq i}}^{n}\frac{\beta_j}{(\tau_i-\tau_j)} + \sum_{\substack{j=0\\ j\neq i}}^{n}\frac{\beta_j(\tau_j-\tau_i)^{-1}}{(z-\tau_j)}\,. \tag{11.20}$$

Our contour integral then gives

$$0 = \beta_i f'(\tau_i) + \left(\sum_{\substack{j=0\\ j\neq i}}^{n}\frac{\beta_j}{(\tau_i-\tau_j)}\right) f(\tau_i) + \sum_{\substack{j=0\\ j\neq i}}^{n}\beta_j(\tau_j-\tau_i)^{-1}f(\tau_j)\,, \tag{11.21}$$

from which $f'(\tau_i)$ can be isolated. Therefore, if $i\neq j$,

$$D_{ij} = -\frac{(\tau_j-\tau_i)^{-1}\beta_j}{\beta_i} \tag{11.22}$$

and D_{ii} is given by (11.16). The cost of constructing this matrix is $O(n^2)$ flops. That is, the off-diagonal elements are easily found from the barycentric weights, and thence the entire differentiation matrix. A similar formula can be derived for Hermite interpolation (see the next section) and is implemented in `genbarywts`.

11.2.3 A Detailed Derivation of the Differentiation Matrix for the Hermite Interpolation Polynomial

This section can be skipped if you are not overly enamored of sums. The result at the end, however, is remarkably simple: a formula for the nontrivial entries of the differentiation matrix for confluent (Hermite) interpolation. The journey along the way, if you *are* enamored of sums, is rather nice.

The goal of this section is to find formulæ for the entries of the matrix **D**, which takes the vector of coefficients ρ_{ij} of a polynomial $f(t)$ expressed in a Hermite interpolational basis to the vector of coefficients $\text{þ}_{i,j}$ of the derivative $f'(t)$, *expressed in the same basis*.[2] That is, if

$$f(t) = w(t)\sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\sum_{k=0}^{j}\beta_{i,j}\rho_{i,k}(t-\tau_i)^{k-j-1}\,, \tag{11.23}$$

in (say) the first barycentric form, then $\text{þ} = \mathbf{D}\boldsymbol{\rho}$ gives us

$$f'(t) = w(t)\sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\sum_{k=0}^{j}\beta_{i,j}\text{þ}_{i,k}(t-\tau_i)^{k-j-1}\,, \tag{11.24}$$

with exactly the same $\beta_{i,j}$. Because the coefficients of the Hermite interpolational basis are scaled derivatives, $\rho_{i,j} = {}^{f^{(j)}(\tau_i)}\!/_{j!}$, computing these coefficients is equivalent to computing further derivatives at the nodes τ_i.

Let us begin. If the distinct nodes τ_i with confluencies $s_i \geq 1$ are given, first compute the generalized barycentric weights β_{ij} from the partial fraction decomposition

$$\frac{1}{w(z)} = \prod_{i=0}^{n}(z-\tau_i)^{-s_i} = \sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\beta_{ij}(z-\tau_i)^{-j-1}\,. \tag{11.25}$$

As stated, if

$$\boldsymbol{\rho} = [\rho_{00}, \rho_{01}, \ldots, \rho_{0,s_0-1}, \rho_{10}, \rho_{11}, \ldots, \ldots, \rho_{n,s_n-1}]^T \tag{11.26}$$

and

$$p(z) = w(z)\sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\sum_{k=0}^{j}\beta_{ij}\rho_{ik}(z-\tau_i)^{k-j-1} \tag{11.27}$$

[2] We puzzled for a while over what symbol to use for the vector of derivative values. If the function values $f(\tau_i)$ are denoted by ρ_i, then clearly the symbol for the derivatives $f'(\tau_i)$ should be related, though different. Finally, we chose the Norse rune þ (pronounced "thorn" with a "th" as in the word "thin") because it looks a bit like a ρ, and linguistically the symbol is equivalent to the atin D convenient for derivative. In some fonts it has an angular character, which fits with the loss of smoothness when derivatives are taken. We also had to worry about conflicts with other symbols in this book. So, "thorn" (þ) it is Elliot (1981).

interpolates the given data $\boldsymbol{\rho}$, where

$$\rho_{i,s_i} := \frac{p^{(s_i)}(\tau_i)}{s_i!}\,, \tag{11.28}$$

we want the values that we denote

$$\text{þ} = [\rho_{01}, 2\rho_{02}, \ldots, s_i\rho_{0,s_i}, \rho_{11}, 2\rho_{12}, \ldots, s_n\rho_{n,s_n}]^T \tag{11.29}$$

and the entries of a differentiation matrix $\mathbf{D}$ such that

$$\mathbf{D}\boldsymbol{\rho} = \text{þ}\,. \tag{11.30}$$

Obviously, the rows of $\mathbf{D}$ that simply shift ρ_{ij} up one are elementary: The first $s_0 - 1$ rows of $\mathbf{D}$, for example, are

$$\begin{bmatrix} 0 & 1 & & & \\ 0 & 0 & 2 & & \\ \vdots & \vdots & & \ddots & \\ 0 & 0 & \cdots & 0 & s_0 - 1 \end{bmatrix}, \tag{11.31}$$

which is an $(s_0 - 1) \times s_0$ matrix. The reason that the entries are not simply 1 is that $\rho_{i,j} = f^{(j)}(\tau_i)/j!$, and we wish to replace that with $\text{þ}_{i,j} = (f')^{(j)}(\tau_i)/j!$, and if we get this by shifting $\rho_{i,j+1}$, we have to adjust the factorial.

But the s_0th row is nontrivial, as are the $(s_0 + s_1)$th, $(s_0 + s_1 + s_2)$th, and so on. For these rows, we must have, for $0 \le i \le n$,

$$d^{(i)}_{00}\rho_{00} + d^{(i)}_{01}\rho_{01} + \cdots + d^{(i)}_{n,s_n-1}\rho_{n,s_n-1} = \text{þ}_{i,s_i-1} = \rho_{i,s_i}\,. \tag{11.32}$$

To find these nontrivial entries in $\mathbf{D}$, consider the following contour integral:

$$0 = \frac{1}{2\pi i}\oint_C \frac{f(z)}{(z-\tau_k)w(z)}dz\,, \tag{11.33}$$

which is zero if, first, $\deg f(z) \le d$, where $1 + d = \sum_{i=0}^{n} s_i$ and $\deg w(z) = 1 + d$, so that the degree of the denominator $(z - \tau_k)w(z)$ is $d + 2$, and, second, if the contour C encloses all τ_k. Now expand $1/w(z)$ as a partial fraction:

$$\frac{1}{z-\tau_k}\frac{1}{w(z)} = \frac{1}{z-\tau_k}\sum_{i=0}^{n}\sum_{j=0}^{s_i-1}\beta_{ij}(z-\tau_i)^{-j-1}\,, \tag{11.34}$$

and split the sum into the repeated term and the rest, as follows:

$$\frac{1}{(z-\tau_k)w(z)} = \frac{1}{z-\tau_k}\sum_{j=0}^{s_k-1}\frac{\beta_{kj}}{(z-\tau_k)^{j+1}} + \frac{1}{z-\tau_k}\sum_{\substack{i=0\\ i\neq k}}^{n}\sum_{j=0}^{s_i-1}\frac{\beta_{ij}}{(z-\tau_i)^{j+1}}$$

$$
\begin{aligned}
&= \sum_{j=0}^{s_k-1} \frac{\beta_{kj}}{(z-\tau_k)^{j+2}} + \frac{1}{z-\tau_k} \sum_{\substack{i=0\\ i\neq k}}^{n} \sum_{j=0}^{s_i-1} \beta_{ij}(\tau_k-\tau_i)^{-j-1} \\
&\quad + \sum_{\substack{i=0\\ i\neq k}}^{n} \frac{1}{z-\tau_k} \cdot \sum_{j=0}^{s_i-1} \beta_{ij} \left(\frac{1}{(z-\tau_i)^{j+1}} - \frac{1}{(\tau_k-\tau_i)^{j+1}} \right) .
\end{aligned}
$$

To complete the new partial fraction decomposition, which must look like

$$
\frac{1}{(z-\tau_k)w(z)} = \sum_{j=0}^{s_k-1} \frac{\beta_{kj}}{(z-\tau_k)^{j+2}} + \frac{1}{z-\tau_k} \sum_{\substack{i=0\\ i\neq k}}^{n} \sum_{j=0}^{s_i-1} \beta_{ij}(\tau_k-\tau_i)^{-j-1} \tag{11.35}
$$

$$
+ \sum_{\substack{i=0\\ i\neq k}}^{n} \sum_{j=0}^{s_i-1} \beta_{ij;k}(z-\tau_i)^{-j-1} , \tag{11.36}
$$

we need to compute the new residues $\beta_{ij;k}$. This can be done with Cauchy convolution as follows. Near $z = \tau_i$, $i \neq k$, the important terms are

$$
\frac{1}{z-\tau_k} \sum_{j=0}^{s_i-1} \frac{\beta_{ij}}{(z-\tau_i)^{j+1}} . \tag{11.37}
$$

Notice that

$$
\frac{1}{z-\tau_k} = \frac{1}{z-\tau_i+\tau_i-\tau_k} = \frac{1}{\tau_i-\tau_k} \frac{1}{1-\dfrac{z-\tau_i}{\tau_k-\tau_i}} \tag{11.38}
$$

$$
= -\sum_{\ell\geq 0} \frac{(z-\tau_i)^{\ell}}{(\tau_k-\tau_i)^{\ell+1}} \tag{11.39}
$$

for $|z-\tau_i| < |\tau_k-\tau_i|$. The important terms then become

$$
\begin{aligned}
&- \left(\sum_{\ell\geq 0} \frac{(z-\tau_i)^{\ell}}{(\tau_k-\tau_i)^{\ell+1}} \right) \left(\sum_{j=0}^{s_i-1} \beta_{ij}(z-\tau_i)^{-j-1} \right) \\
&= - \left(\sum_{\ell\geq 0} \frac{1}{(\tau_k-\tau_i)^{\ell+1}} (z-\tau_i)^{\ell} \right) \left(\sum_{m=0}^{s_i-1} \beta_{i,s_i-m-1}(z-\tau_i)^{-(s_i-m-1)-1} \right) \\
&= - \left(\sum_{\ell\geq 0} \frac{1}{(\tau_k-\tau_i)^{\ell+1}} (z-\tau_i)^{\ell} \right) \left(\sum_{m=0}^{s_i-1} \beta_{i,s_i-m-1}(z-\tau_i)^{m} \right) (z-\tau_i)^{-s_i} \\
&= -(z-\tau_i)^{-s_i} \cdot \sum_{j\geq 0} c_j (z-\tau_i)^{j} ,
\end{aligned}
$$

where

$$c_j = \sum_{\ell=0}^{j} \beta_{i,s_i-\ell-1} \frac{1}{(\tau_k - \tau_i)^{j-\ell+1}} \tag{11.40}$$

by Cauchy convolution. Simplifying, (11.37) then results in

$$\sum_{j\geq 0} \left(-\sum_{\ell=0}^{j} \beta_{i,s_i-\ell-1} \frac{1}{(\tau_k - \tau_i)^{j-\ell+1}} \right) (z-\tau_i)^{j-s_i} , \tag{11.41}$$

and only the terms $j = 0, 1, 2, \ldots, s_i - 1$ contribute to the residue:

$$\sum_{j=0}^{s_i-1} \left(-\sum_{\ell=0}^{j} \beta_{i,s_i-\ell-1} \frac{1}{(\tau_k - \tau_i)^{j-\ell+1}} \right) (z-\tau_i)^{j-s_i} . \tag{11.42}$$

Putting $m = s_i - 1 - j$, this is

$$\sum_{m=0}^{s_i-1} \left(-\sum_{\ell=0}^{s_i-1-m} \beta_{i,s_i-\ell-1} \frac{1}{(\tau_k - \tau_i)^{s_i-1-m+1-\ell}} \right) (z-\tau_i)^{-m-1} , \tag{11.43}$$

which is the desired form, $\beta_{i,m;k}$. We can simplify a bit further by writing

$$\beta_{i,m;k} = -\sum_{\ell=0}^{s_i-1-m} \beta_{i,s_i-\ell-1} \frac{1}{(\tau_k - \tau_i)^{s_i-m-\ell}} \tag{11.44}$$

and letting $\mu = s_i - \ell - 1$, so that

$$\begin{aligned} \beta_{i,m;k} &= -\sum_{\mu=s_i-1}^{s_i-(s_i-1-m)-1} \beta_{i,\mu} \frac{1}{(\tau_k - \tau_i)^{s_i-m-(s_i-\mu-1)}} && (11.45) \\ &= -\sum_{\mu=m}^{s_i-1} \beta_{i,\mu} (\tau - \tau_i)^{m-1-\mu} . && (11.46) \end{aligned}$$

That is, we obtain the simple expression

$$\beta_{i,j;k} = -\sum_{\mu=j}^{s_i-1} \beta_{i,\mu} (\tau_k - \tau_i)^{j-1-\mu} . \tag{11.47}$$

Finally, performing the contour integration gives

$$\sum_{j=0}^{s_k-1} \beta_{kj} \frac{f^{(j+1)}(\tau_k)}{(j+1)!} + \left(\sum_{\substack{i=0\\ i\neq k}}^{n} \sum_{j=0}^{s_i-1} \beta_{ij}(\tau_k - \tau_i)^{-j-1} \right) f(\tau_k) \\ + \sum_{\substack{i=0\\ i\neq k}}^{n} \sum_{j=0}^{s_i-1} \beta_{ij;k} \frac{f^{(j)}(\tau_i)}{j!} = 0\,, \qquad (11.48)$$

and the term $f^{(s_k)}(\tau_k)/s_k!$ corresponds to $j = s_k - 1$ in the first sum:

$$\beta_{k,s_k-1} \frac{f^{(s_k)}(\tau_k)}{s_k!} = -\sum_{j=0}^{s_k-2} \beta_{kj} \frac{f^{(j+1)}(\tau_k)}{(j+1)!} \\ - \left(\sum_{\substack{i=0\\ i\neq k}}^{n} \sum_{j=0}^{s_i-1} \beta_{ij}(\tau_k - \tau_i)^{-j-1} \right) f(\tau_k) - \sum_{\substack{i=0\\ i\neq k}}^{n} \sum_{j=0}^{s_i-1} \beta_{ij;k} \frac{f^{(j)}(\tau_i)}{j!}\,. \qquad (11.49)$$

At long last, if $\beta_{k,s_k-1} \neq 0$, the entries of the nontrivial row in $\mathbf{D}$ for τ_k can be read off that last line, when it is divided by β_{k,s_k-1}. For example, the entry that multiplies ρ_{ij} for $i \neq k$ is just $-\beta_{ij;k}/\beta_{k,s_k-1}$ (and $\beta_{ij;k}$ is computed in (11.47)). As in the Lagrange case, for stability reasons, it is best to compute the entry multiplying $\rho_{k,0}$ by insisting that the differentiation matrix be exact for $f(z) \equiv 1$. This implies that

$$\mathbf{D}_{k,0} = -\sum_{\substack{i=0\\ i\neq k}}^{n} \mathbf{D}_{i,0}\,, \qquad (11.50)$$

that is, only the elements multiplying $\rho_{i,0}$ will figure, because for $f(z) \equiv 1$, all other $\rho_{i,j} = 0$. Finally, we must multiply the row by s_k, because, as noted for the trivial rows, we have just computed $f^{(s_k)}(\tau_k)/s_k!$, and we by convention want $(f')^{(s_k-1)}/(s_k-1)!$, in order to match the existing barycentric form: That is, using the same weights as before, we may now evaluate $f'(t)$ for any t.

Note that the distinctness of the τ_i is used throughout. In particular, $\beta_{k,s_k-1} = 0$ if the nodes are not distinct.

Remark 11.2. We leave to Exercises 11.9 and 11.16 the important questions of *conditioning* of the computation of $\mathbf{D}$—and of the differentiation of polynomials that it is to effect—and the question of the numerical stability of the algorithms: Do they compute the exact matrix $\mathbf{D}$ for some nearby polynomial differentiation problem? In particular, do they have small componentwise backward error? ◁

11.2.4 Differentiation Matrix Examples

Example 11.2. For $\boldsymbol{\tau} = [-1, -1/3, 1/3, 1]$, (cf. equation (11.9)) the differentiation matrix is (using the MATLAB function `genbarywts.m`),

$$\mathbf{D} = \begin{bmatrix} -2.7500 & 4.500 & -2.2500 & 0.5000 \\ -0.5000 & -0.7500 & 1.5000 & -0.2500 \\ 0.2500 & -1.5000 & 0.7500 & 0.5000 \\ -0.5000 & 2.2500 & -4.5000 & 2.7500 \end{bmatrix} .$$

So, if the polynomial values at $\boldsymbol{\tau}$ are $\boldsymbol{\rho} = [1, -1, 1, -1]^T$, then

$$\begin{aligned} \mathbf{f}' &= \mathbf{D}\boldsymbol{\rho} = [-10, 2, 2, -10]^T \\ \mathbf{f}'' &= \mathbf{D}\mathbf{f}' = [27, 9, -9, -27]^T \\ \mathbf{f}''' &= \mathbf{D}\mathbf{f}'' = [-27, -27, -27, -27]^T \\ \mathbf{f}^{(4)} &= \mathbf{D}\mathbf{f}''' = [0, 0, 0, 0]^T . \end{aligned}$$

For the last one, as computed in MATLAB, the norm of the vector was $\approx 10^{-13}$, not zero as it should have been. Rounding errors do show up. ◁

Example 11.3. If we take $n = 4$ and the $n+1$ nodes $\tau_j = \cos(j\pi/n)$, $j = 0, 1, 2, 3$, and 4, then the barycentric weights can be scaled to be $\beta_0 = 1$, $\beta_1 = -\beta_2 = \beta_3 = -2$, and $\beta_4 = 1$. The differentiation matrix is

$$\mathbf{D} = \begin{bmatrix} 5.5000 & -6.8284 & 2.0000 & -1.1716 & 0.5000 \\ 1.7071 & -0.7071 & -1.4142 & 0.7071 & -0.2929 \\ -0.5000 & 1.4142 & -0.0000 & -1.4142 & 0.5000 \\ 0.2929 & -0.7071 & 1.4142 & 0.7071 & -1.7071 \\ -0.5000 & 1.1716 & -2.0000 & 6.8284 & -5.5000 \end{bmatrix}$$

if the confluency is just 1—that is, we are doing Lagrange interpolation. If the confluency is instead 2 at each node, then the differentiation matrix is instead $\mathbf{D} =$

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -147.5 & 22 & 131.9 & 27.31 & 8 & 8 & -3.882 & 4.686 & 11.5 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 24.61 & -1.707 & -31 & -2.828 & 4 & 2.828 & -1 & 1.414 & 3.393 & 0.292 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 6 & -0.5 & 0 & -2.828 & -12 & 0 & 0 & 2.828 & 6 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 3.393 & -0.2929 & -1 & -1.414 & 4 & -2.828 & -31 & 2.828 & 24.61 & 1.707 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 11.5 & -1 & -3.882 & -4.686 & 8 & -8 & 131.9 & -27.31 & -147.5 & -22 \end{bmatrix} . \qquad (11.51)$$

If, say, we interpolate the values and derivatives

$$\boldsymbol{\rho} = \begin{bmatrix} 0.5000 & 0.6667 & 1.0000 & 0.6667 & 0.5000 \\ -0.5000 & -0.6285 & -0.0000 & 0.6285 & 0.5000 \end{bmatrix} \tag{11.52}$$

at these points, where the first row gives $\rho_{i,0} = f(\tau_i)$ and the second row gives $\rho_{i,1} = f'(\tau_i)$, then application of the differentiation matrix $\mathbf{D}$ to the (straightened out) vector gives

$$\text{þ} = \begin{bmatrix} -0.5000 & -0.6285 & -0.0000 & 0.6285 & 0.5000 \\ 0.6111 & 0.3333 & -1.9444 & 0.3333 & 0.6111 \end{bmatrix} . \tag{11.53}$$

That is to say, for example, that the *second* derivatives of the original function $f(t)$ will be approximated at the nodes by the entries in the second row of þ, because they are the exact (up to roundoff) second derivatives of the polynomial that interpolates the given value and derivative data. See Fig. 11.2. ◁

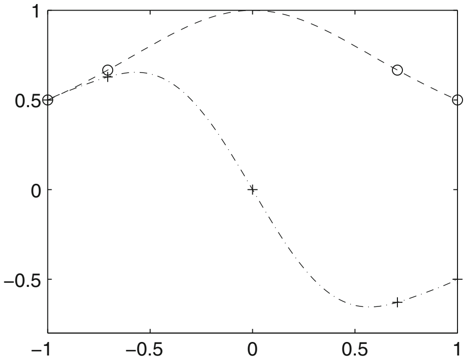


Fig. 11.2 The degree- 7 Hermite interpolant to the data (11.52), together with its derivative, computed *via* the differentiation matrix (11.51)

What about the ultimate flatness problem, the infinite ill-conditioning talked about at the beginning? Well, because we have constrained our differentiation to the space of polynomials, things work out well enough (in this case, the absolute condition number is, as stated earlier, bounded). For the problem in Fig. 11.1, the differentiation matrix produces a vector þ of derivative values that can then subsequently be used in `hermiteval` to plot the derivative quite acceptably; if we zoom in on that, we find that the straight line that we expect has a representation in terms of piecewise flat functions (as it must, too) that rise at an appropriate "slope." See Fig. 11.3. The derivative values are computed at the nodes as

$$\text{þ} = \begin{bmatrix} -0.1289 & 0 & 0 & 0.1289 \\ 1.5078 & 0.0312 & 0.0312 & 1.5078 \end{bmatrix} . \tag{11.54}$$

Evaluation of the derivative by the second barycentric form of the Hermite interpolant is straightforward and accurate, provided the mesh widths are not too widely varying or too small.

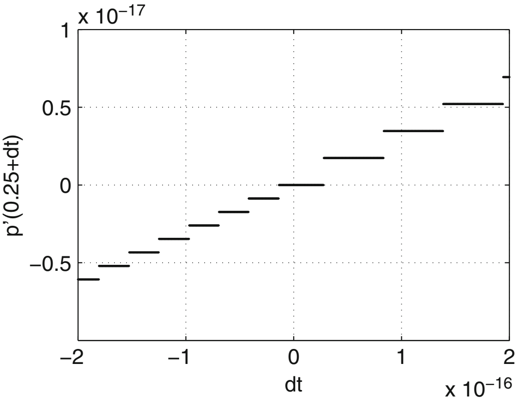


Fig. 11.3 Derivative of a polynomial in the Hermite interpolational basis

Remark 11.3. As previously stated, if an interpolant is $O(h^p)$ accurate as $h \to 0$, typically its derivative will only be $O(h^{p-1})$ accurate in the same limit. ◁

11.3 Complex Nodes

It is often necessary or advantageous to use complex nodes, and so we discuss this case here. To begin, notice that, if C is the circular contour $\zeta + re^{i\theta}$, the integral

$$f'(\zeta) = \frac{1}{2\pi i}\oint_C \frac{f(z)}{(z-\zeta)^2}dz \tag{11.55}$$

can be written explicitly as

$$f'(\zeta) = \frac{1}{2\pi i}\int_0^{2\pi} \frac{f(\zeta + re^{i\theta})rie^{i\theta}}{r^2e^{2i\theta}}d\theta = \frac{1}{2\pi r}\int_0^{2\pi} f(\zeta + re^{i\theta})e^{-i\theta}d\theta \tag{11.56}$$

and that the integrand is analytic and periodic in θ with period 2π and that we are integrating over a whole period. Therefore, an equi-spaced rule like the trapezoidal rule will be spectrally convergent. That is, let $\Delta\theta = {}^{2\pi}\!/_{n}$ and approximate the integral by

$$\frac{1}{2\pi r}\frac{2\pi}{n}\left(\frac{1}{2}f(\zeta + r) + \sum_{k=1}^{n-1} f(\zeta + re^{2\pi ik/n})e^{-2\pi ik/n} + \frac{1}{2}f(\zeta + re^{2\pi i})\right)$$
$$= \frac{1}{rn}\sum_{k=0}^{n-1} f(\zeta + re^{2\pi ik/n})e^{-2\pi ik/n}. \tag{11.57}$$

If $f(z)$ is polynomial, the rule is exact for degrees $\le n-1$. If $f(z)$ is not polynomial, then the error is exponentially small with n, in theory; and rounding errors are relatively harmless.

Example 11.4. If we take $n=7$ in that formula and then take its series expansion as $r \to 0$ (using MAPLE), we find that

$$\frac{1}{rn}\sum_{k=0}^{n-1} f(\zeta + re^{2\pi ik/n})e^{-2\pi ik/n} = f'(\zeta) + \frac{f^{(8)}(\zeta)}{8!}r^7 + \frac{f^{15}(\zeta)}{15!}r^{14} + O(r^{21})\,.$$

Indeed, the error terms are all of the form $r^{\ell n} \cdot f^{(\ell n+1)}(\zeta)/(\ell n+1)!$, confirming the statement above that the approximation is spectrally convergent if $r<1$, as $n \to \infty$. ◁

Remark 11.4. The alert reader who remembers the derivation in Chap. 9 of the finite Fourier transform from the barycentric form of the interpolant will recognize this derivative as the linear term c_1 from the FFT of the values of the polynomial interpolant to f at the roots of unity. Indeed, this computation of the derivative can be done by the FFT (in which case, we don't just get one derivative; we get n derivatives). See Problem 11.7. ◁

Example 11.5. Consider the short program

```
function [ dy ] = Lyness( f, zeta, r, n )
%LYNESS Differentiate f by complex evaluation around zeta
%   Spectrally convergent formula from Lyness and Moler 1967
%   dy  = Lyness( f, zeta, r, n ) gives dy = f'(zeta) + O(r^n)
tau = zeta + r* exp(2*pi*1i*(0:n-1)/n);
y = feval( f, tau );
dy = sum(y.*exp(-2*pi*1i*(0:n-1)/n))/n/r;
end
```

If we execute the commands

```
dy = Lyness( @(z) sin(z), 1, 0.1, 5 )
dy/cos(1) - 1
```

we find that the error in approximating the derivative with a five-term complex integral is about -2×10^{-8}. If instead we ask for 10 function evaluations, the answer is accurate to full machine precision.

We remark that this only works with functions that MATLAB knows how to evaluate over $\mathbb{C}$. Functions such as $1/\Gamma(z)$ cannot be differentiated in MATLAB by this method, because the implementation in MATLAB is limited to real z. If MATLAB were better at the complex-valued Gamma function, this would work for that function as well; in other environments, it does work. ◁

11.3.1 The Differentiation Matrices on Roots of Unity

If the interpolation nodes are the nth roots of unity, $\tau_k = \exp(2\pi ik/n)$, for $0 \le k \le n-1$, then because $w(z) = z^n - 1$, we have some particularly simple formulæ for the barycentric weights of the Lagrange basis:

$$\beta_k = \frac{\tau_k}{n} \qquad \mathbf{D}_{kk} = \frac{n/2}{\tau_k} \qquad \mathbf{D}_{jk} = \frac{\tau_k}{\tau_j(\tau_k - \tau_j)}. \tag{11.58}$$

In exact arithmetic, all differentiation matrices are singular (because $\mathbf{D1} = \mathbf{0}$ and indeed they are nilpotent, $\mathbf{D}^{n+1} = \mathbf{0}$), but the differentiation matrix for Lagrange interpolation on roots of unity is particularly simple, having singular values $n-1$, $n-2$, …, 1, and 0, just like the monomial basis.

Theorem 11.3. *The singular values of the Lagrange basis differentiation matrix on the nth roots of unity are $\sigma_j = n - j$, for $j = 1, 2, \ldots, n$.*

Proof. Consider the polynomial z^{j-1}, for $j = 1, 2, \ldots, n$ in succession, and the vectors of its values on the nodes $\tau_k = \exp(2\pi i k/n)$. Evaluating z^{j-1} on the vector of nodes gives the vector with entries $\exp(2\pi i(j-1)k/n)$. The 2-norm of each of the orthogonal vectors with entries $v_j = \exp(2\pi i(j-1)k/n)/\sqrt{n}$ is 1, independent of j. Applying the differentiation matrix to v_j gives

$$\mathbf{D}v_j = (j-1)v_{j-1} \tag{11.59}$$

(and one can take $v_{-1} = v_n$ by orthogonality) because the differentiation matrix is exact for polynomials of degree less than n. Therefore, $\mathbf{D} = \mathbf{U\Sigma V}^H$, where the jth column of $\mathbf{V}$ is v_j and where the jth column of $\mathbf{U}$ is v_{j-1}. ♮

Remark 11.5. This theorem seems surprising when we come at it (as we have) via polynomial interpolation. However, it's really a discrete Fourier transform, to evaluate or interpolate on the roots of unity, and that differentiation is simple in Fourier space is not at all a surprise (see Chap. 9). Since the differentiation matrices in different bases are related by the change-of-basis matrix, here the quasi-unitary Vandermonde matrix on the roots of unity, we find that the singular values are just the same as the singular values of the monomial basis differentiation matrix. ◁

11.3.2 A Surprisingly Simple Rule

As a final use of complex finite differences to approximate the derivative of a real-valued function, consider the following very simple formula, which is due to Squire and Trapp (1998):

$$f'(x_0) = \frac{\mathrm{Im}(f(x_0 + ih))}{h} + O(h^2). \tag{11.60}$$

It follows for real-valued f from

$$f(x_0 + ih) = f(x_0) + f'(x_0)ih - \frac{h^2}{2}f''(x_0) + O(h^3)\,. \tag{11.61}$$

A purely real-variable formula of the same theoretical order is the central difference formula

$$f'(x_0) = \frac{f(x_0 + {}^{h}\!/\!_{2}) - f(x_0 - {}^{h}\!/\!_{2})}{h} + O(h^2)\,. \tag{11.62}$$

You will derive this formula in Problem 11.4, following the method of the next section. This suffers severely from subtractive cancellation, which reveals rounding errors made in the computation of $f(x_0 + {}^{h}\!/\!_{2})$ and $f(x_0 - {}^{h}\!/\!_{2})$. A comparison of the two formulæ employed on

$$f(x) = \frac{e^x}{\sin^3 x + \cos^3 x} \tag{11.63}$$

can be seen in Fig. 11.4, which was generated by the following code:

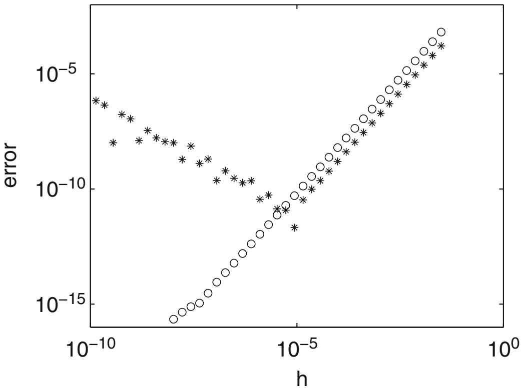


Fig. 11.4 An example showing that the complex formula (11.61) (*circles*) hardly suffers at all from rounding error, getting results accurate to machine epsilon, whereas the classical real central difference formula (11.62) (*dots*) shows the same theoretical order (and is a bit more accurate) for h bigger than about 10^{-4}, but suffers severely from catastrophic cancellation for h smaller than about 10^{-4}. For small enough h, the complex formula gets the exact answer for this problem, and the error is not representable on a log-scale graph

```
f = @(x) exp(x)./(sin(x).^3+cos(x).^3);
c = @(h) (f(h/2)-f(-h/2))./h;
st= @(h) imag(f(1i*h))./h;
phi =(1+sqrt(5))/2;
% Fibonacci spaced h
h   = 1.0./((2+2/sqrt(5))*(phi.^[5:50]));
ce  = c(h)-1;
se  = st(h)-1;
% Squire-Trapp formula gives exact result for small enough h
se(end)
loglog( h,abs(ce),'k*', h,abs(se),'ko' )
set(gca,'fontsize',16)
```

```
axis([1.0e-10,1,1.0e-16,1.0e-2])
xlabel('h')
ylabel('error')
```

Remark 11.6. In some sense, this formula cheats! If one can evaluate functions over $\mathbb{C}$, then one already knows a lot about the derivative. Consider

$$\begin{aligned}\sin(x+ih) &= \sin(x)\cos(ih)+\sin(ih)\cos(x)\\ &= \sin(x)\cosh(h)+i\sinh(h)\cos(x)\,,\end{aligned} \tag{11.64}$$

so that the imaginary part is

$$\operatorname{Im}(\sin(x+ih)) = \frac{\sinh(h)}{h}\cos(x)\,. \tag{11.65}$$

The true derivative, $\cos x$, is there in the limit, completely naturally. This also has some links with automatic differentiation (see Sect. 11.7). ◁

Remark 11.7. We should also point out that this method requires the imaginary part of $f(x+ih) = u(x,h)+iv(x,h)$, that is, $v(x,h)$, to be computed with good relative accuracy; that is, we need

$$fl(v(x,h)) = v(x,h)(1+\delta_1) \tag{11.66}$$

for some modestly small δ_1. If that is the case, then $f'(x)$ will be computed as

$$fl\left(\frac{v(x,h)}{h}\right) = f'(x)(1+\delta_1)(1+\delta_2)+O(h^2)\,, \tag{11.67}$$

where now $|\delta_2| \leq \mu_M$. This is what happened in Fig. 11.4. If we instead have only *absolute* accuracy,

$$fl\,(v(x,h)) = v(x,h)+\Delta\,, \tag{11.68}$$

where Δ is modestly small, then

$$fl\left(\frac{v(x,h)}{h}\right) = f'(x)(1+\delta_2)+\frac{\Delta}{h}(1+\delta_2)+O(h^2) \tag{11.69}$$

and the $O(^{\Delta}/_{h})$ term will go to infinity as $h \to 0$, exactly as errors in the real-valued finite-difference formulæ do. This happens for the Bessel function example in MATLAB if the derivatives are taken not at $x = 1$ as in the code here but rather at larger values of x, say $x = 22.3$. This demonstrates very clearly that the MATLAB routines do not evaluate Bessel functions with high relative accuracy in the imaginary part for arguments near the real axis if x is at all large. ◁

11.4 A Backward Error Interpretation of Simple Finite Differences

We now rejoin the mainstream of finite differences. In order to get there, recall the mean value theorem: If f is $\mathscr{C}^{(1)}(a,a+h)$, then (for real variables)

$$\frac{f(a+h)-f(a)}{h} = f'(a+\theta h), \tag{11.70}$$

for some $\theta \in (0,1)$. Remember that $\theta = \theta(h)$ depends on h. Here

$$\Delta f = f(a+h) - f(a) \tag{11.71}$$

is called the *forward difference*.[3] Here, note that $\Delta x = a+h-a = h$. The ratio of Δf to Δx is often called a forward divided difference. Sometimes the terminology gets sloppy and the word "divided" is left out. The important point to note here is that a forward divided difference gives you the *exact* value of the derivative at some point between a and $a+h$ (note h may be negative).

In contrast, the *central divided difference* is defined as the ratio on the left, below, while on the right, the mean value theorem asserts that this, too, is an exact derivative,

$$\frac{f\left(a+{}^{h}\!/_{2}\right)-f\left(a-{}^{h}\!/_{2}\right)}{h} = f'\left(a+\frac{h}{2}\theta_2\right), \tag{11.72}$$

for some $\theta_2 \in (-1,1)$. To avoid confusion with forward divided differences, we will denote this by ${}^{\delta(f)}\!/_{\Delta x}$.

Remark 11.8. Both of these methods for approximating derivatives, and a great many others, may be found by contour integration as we have been using it. For example, the integral

$$0 = \oint_C \frac{f(z)}{(z-t)^2(z-t-h)}\,dz \tag{11.73}$$

gives the forward difference formula via the partial fraction expansion

$$\frac{{}^{1}\!/_{h^2}}{z-t-h} - \frac{{}^{1}\!/_{h^2}}{z-t} - \frac{{}^{1}\!/_{h}}{(z-t)^2}\,. \tag{11.74}$$

If f is a polynomial of degree at most 1, it gives exactly the relation (11.70) (with $\theta = 0$) by Cauchy's theorem. If f is not a linear polynomial, then θ will not be 0, of course. ◁

[3] It may become clearer why this is called a *forward* difference if you think of $x_k = a$ and $x_{k+1} = a+h$ with $h > 0$. Then Δf and Δx both "look ahead" to compute the differences.

Can we say more about $\theta(h)$? How close to where we want to take the derivative does the finite (divided) difference get us? A series computation for forward differences shows that

$$\theta(h) = \frac{1}{2} + \frac{h}{24}\frac{f'''(a)}{f''(a)} + O(h^2) = \frac{1}{2} + O(h) \tag{11.75}$$

if $f''(a) \neq 0$. A more complicated formula holds if $f''(a) = 0$. Thus, if $f''(a) \neq 0$, then as $h \to 0$,

$$\frac{\Delta f}{\Delta x} = f'\left(a + h(\frac{1}{2} + O(h))\right) = f'(a) + \frac{h}{2}f''(a) + O(h^2)\,. \tag{11.76}$$

Therefore, both the forward error and the backward error are, in general, $O(h)$; if f'' is nearly zero, though, this requires a separate analysis.

For central differences, we find a better bound; namely, we have

$$\theta_2(h) = \frac{h}{12}\frac{f'''(a)}{f''(a)} + O\left(\frac{h^3}{(f''(a))^3}\right) \tag{11.77}$$

if $f''(a) \neq 0$. Recall that

$$\begin{aligned}\frac{\delta(f)}{\Delta x} &= \frac{f(a + {}^{h}\!/_{2}) - f(a - {}^{h}\!/_{2})}{h}\\ &= f'\left(a + \theta_2 \cdot \frac{h}{2}\right) = f'\left(a + \frac{h^2}{24}\frac{f'''(a)}{f''(a)} + O(h^4)\right)\\ &= f'(a) + \frac{h^2}{24}f'''(a) + O(h^4)\,, \end{aligned}\tag{11.78}$$

which shows that the formula is $O(h^2)$ accurate, both backward and forward. If $f''(a)$ is small, again we have to redo the analysis.

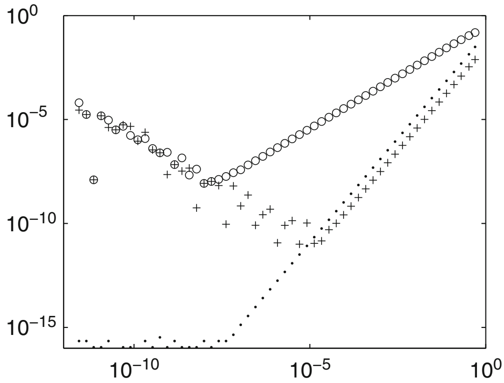


Fig. 11.5 Errors in three finite-difference formulæ: forward (*opencircle*), central (*plus*), and the Squire–Trapp complex formula (*dot*), for the Bessel function $J_0(x)$ at $x = 1$. Forward differences have $O(h)$ error, while the other two have $O(h^2)$ error

To compare different difference formulæ, we can execute the commands

```
[h,fe,ce,ie] = fordifferr(@(x)besselj(0,x),1, -besselj(1,1),55);
loglog( h, abs(fe), 'ko', h, abs(ce), 'k+', h, abs(ie), 'k.' )
```

to produce the graph in Fig. 11.5. These commands call the following short program:

```
function [h,ef,ec,ei] = fordifferr( f, a, dfa, k )
  function f = fibonacci(n)
    f = zeros(1,n);
    f(1) = 1;
    f(2) = 1;
    for i=3:n,
        f(i) = f(i-1)+f(i-2);
    end;
  end
  ns = fibonacci(k+2);
  h  = 1.0./ns(3:end); % Throw away 1's at start
  x  = a + h;
  ef = zeros(1,k);
  ec = zeros(1,k);
  ya = feval( f, a );
  ef = ( feval( f, x ) - ya )./h/dfa - 1 ;
  ec = ( feval( f, a+h/2 ) - feval( f, a-h/2 ) )./h/dfa - 1;
  ei = ( imag( feval( f, a+i*h ) )./h/dfa ) - 1;
end
```

Note that the program computes relative forward error, not the backward error discussed in this section, but up to a constant, namely, the value of a certain derivative, they are equivalent.

Now, what *rounding error effects* can be expected from finite differences' computation? We have seen several times now that finite-difference formulæ will reveal rounding errors made in the computation of $f(z)$, especially for small h. When this happens, the formula gives increasingly worse approximations to the derivative as h continues to get smaller. This is easy to understand intuitively, and to be expected because differentiation is ill-conditioned, but it can also be understood quantitatively by a rounding error analysis.

Suppose that $f(z+h)$ is computed with relative accuracy δ, that is,

$$fl(f(z+h)) = f(z+h)(1+\delta)\,, \tag{11.79}$$

where δ is some small number. For example, to be as charitable as we can, suppose that f is computed as best it can be, in the *correctly rounded* sense, and so $|\delta| < \mu_M$. Suppose also, to continue to make the case as favorable as possible for finite differences, that no further rounding errors are made *at all* and $f(z)$ is computed exactly and the division by h is done exactly [perhaps $f(z) = 0$ and h is a power of 2; thus only one single rounding error has occurred. This benign case can be hoped for but surely not expected]. Then

$$fl\left(\frac{f(z+h)-f(z)}{h}\right)=\frac{f(z+h)-f(z)}{h}+f(z+h)\frac{\delta}{h} \tag{11.80}$$

and now we see the problem. As $h \to 0$, the first term on the right approaches $f'(z)+O(h)$ as it should, but the second term goes to infinity. Thus, the subtractive cancellation reveals the sole rounding error δ in the limit as $h \to 0$.

11.5 Finite Differences for Higher Derivatives

The traditional higher-order formulæ are obtained by composition of the lower-order formulæ. For example,

$$\begin{aligned}\Delta^2 f &= \Delta(\Delta f) = \Delta(f(a+h)-f(a)) \\ &= f(a+2h)-f(a+h)-(f(a+h)-f(a)) \\ &= f(a+2h)-2f(a+h)+f(a)\,, \end{aligned} \tag{11.81}$$

so $\Delta^2 f/\Delta x^2 = \Delta^2 f/h^2$ should be like f''. Indeed, we may use the mean value theorem twice to capture this:

$$\begin{aligned} f(a+2h)-f(a+h) &= f'(a+h+\theta_1 h)h \\ f(a+h)-f(a) &= f'(a+\theta_2 h)h \end{aligned} \tag{11.82}$$

for some $\theta_1, \theta_2 \in (0,1)$. Thus,

$$\Delta^2 f = f'(a+(1+\theta_1)h)h - f'(a+\theta_2 h)h = f''(a+\theta_3 h)h^2 \tag{11.83}$$

for some $\theta_3 \in (0,2)$.

Alternatively, one can use contour integral methods to define higher-order finite differences, if we add an extra factor in the numerator in order to cancel unwanted (that is, unknown) terms. To find a formula that gives the third derivative at $x=a$, for example, given values of $f(x)$ at (say), $x=[a, a+rh, a+sh, a+h]$, for some values of r and s in $(0,1)$, we might begin by contemplating the partial fraction expansion of

$$\frac{1}{(x-a)^4(x-a-rh)(x-a-sh)(x-a-h)}\,, \tag{11.84}$$

but we would see very quickly that all we would get was a formula relating $f'''(a)$, $f''(a)$, $f'(a)$, and the function values $f(a)$, $f(a+rh)$, $f(a+sh)$, and $f(a+h)$. We do not know $f'(a)$ or $f''(a)$, and so this formula seems useless. A lovely idea of J. C. Butcher is to instead consider the partial fraction decomposition of the same thing, but with some unknown coefficients thrown into the numerator in order to allow us to set some unwanted residues to zero. Consider, therefore,

$$\frac{b_0+b_1x+x^2}{(x-a)^4(x-a-rh)(x-a-sh)(x-a-h)}, \tag{11.85}$$

where the numbers b_0 and b_1 remain to be chosen. We call the function $B(z) = b_0 + b_1z + b_2z^2$ (here $b_2 = 1$) the *Butcher factor*. We can simplify the algebra a decent amount if we use, instead of x, the variable θ, where $x = a+\theta h$. This leads us to the partial fraction decomposition

$$\frac{b_0+b_1\theta+\theta^2}{\theta^4\,(\theta-r)\,(\theta-s)\,(\theta-1)} = \frac{A_4}{\theta^4}+\frac{A_3}{\theta^3}+\frac{A_2}{\theta^2}+\frac{A_1}{\theta}+\frac{A_r}{\theta-r}+\frac{A_s}{\theta-s}+\frac{A_h}{\theta-1}\,.$$

Each of the A_i is a function of b_0, b_1, r, and s. When we set $A_3 = A_2 = 0$, we get two linear equations in the unknowns b_0 and b_1. Solving these, we get

$$b_0 = \frac{rs}{1+s+r} \quad \text{and} \quad b_1 = -\frac{rs+r+s}{1+s+r}\,.$$

With this condition, the partial fraction expansion becomes $(1+r+s)$ times the following:

$$-\frac{1}{s\,(-s+r)\,(-1+s)\,(\theta-s)}+\frac{1}{r\,(-s+r)\,(-1+r)\,(\theta-r)} \\ -\frac{1}{r\theta\,s}+\frac{1}{(-1+r)\,(-1+s)\,(\theta-1)}-\frac{1}{\theta^4}.$$

From this, we conclude that, after we have set our contour integral to zero and isolated the third derivative term,

$$\frac{f'''(a)}{3!} = \frac{1}{h^3}\left(\frac{1}{s(r-s)(1-s)}f(a+sh)\right. \\ \left.+\frac{1}{r(s-r)(1-r)}f(a+rh)-\frac{1}{rs}f(a)+\frac{1}{(1-r)(1-s)}f(a+h)\right)\,,$$

and that this formula is exact for all polynomials of degree at most $7-2-2=3$ (taking the degree of the numerator, subtracting two as usual, then two more for the degree of the new term $b_0+b_1x+x^2$). Taking the Taylor series of the right-hand side above, we get

$$\frac{f'''(a)}{6}+\frac{(1+s+r)f^{(iv)}(a)}{24}h+\frac{(1+s+s^2+rs+r+r^2)f^{(v)}(a)}{120}h^2+O(h^3)\,,$$

and we see that indeed the formula is exact if $f^{(iv)}(x)\equiv 0$, that is, if $f(x)$ is degree 3 or less.

11.6 Compact Finite Differences

In this section, we change our point of view significantly. The main idea is that instead of using a single explicit finite-difference formula to evaluate a derivative at a point, we have a whole mesh of function values and we wish to compute the derivatives at all the nodes. This is quite like the case where we had a global interpolating polynomial and we used a differentiation matrix, and indeed we will have the equivalent of a differentiation matrix here; but it will not be explicitly formed. Instead, we will solve a banded linear system.

We proceed by example. The following formula is known as a classical "Padé" scheme:

$$\frac{1}{6}f'(t)+\frac{2}{3}f'(t+h)+\frac{1}{6}f'(t+2h)=\frac{1}{h}\left(-\frac{1}{2}f(t)+0\cdot f(t+h)+\frac{1}{2}f(t+2h)\right).$$

As we will see, it gives us a tridiagonal (whence "compact") system of equations for the unknown derivatives. That is, instead of simply *applying a formula* to a vector of function values to get a vector of derivative values, we instead have to *set up and solve* a linear system of equations for the unknown derivatives. We have already seen this scheme in use for cubic splines. Having to solve equations instead of using a formula is more complicated, but it has several advantages, mostly stability.

To understand where the system of equations for this formula comes from, make the following simplifying assumptions. Suppose $f'(\tau_0)$ and $f'(\tau_n)$ are known (just to make it simple) and that $\tau_{k+1}-\tau_k=h$ is constant. Then fix attention on one particular node, say τ_k. The formula above becomes, when $t=\tau_{k-1}$, $t+h=\tau_k$, and $t+2h=\tau_{k+1}$,

$$f'(\tau_{k-1})+4f'(\tau_k)+f'(\tau_{k+1})=\frac{3}{h}(f(\tau_{k+1})-f(\tau_{k-1}))\,. \tag{11.86}$$

Now we let k vary over all the indices of the interior nodes, $1\le k\le n-1$. Each interior node gives us one equation. Each equation only contains at most three of the unknown derivatives [and the equation for $k=1$ touches the known derivative $f'(\tau_0)$, while the equation for $k=n-1$ touches the known derivative $f'(\tau_n)$]. This gives us a tridiagonal linear system of equations to solve for the unknown derivatives $f'(\tau_k)$, $1\le k\le n-1$. Call the tridiagonal matrix $\mathbf{M}$. Notice also that the right-hand side of the system involves linear combinations of the values of $f(\tau_k)$ at different nodes—these are supposed to be known. Call the (also tridiagonal) matrix that forms that combination $\mathbf{B}$. Note that $\mathbf{B}$ has a zero diagonal. The system $\mathbf{M}\text{þ}=\mathbf{B}\boldsymbol{\rho}$ needs to be solved computationally to get the vector þ of desired derivatives.

In *effect*, this computes the differentiation matrix $\mathbf{D}$ as $\mathbf{M}^{-1}\mathbf{B}$, but in practice, one never explicitly computes $\mathbf{M}^{-1}$ because it is a full matrix. Instead, of course, to compute $\mathbf{Dv}$, one solves $\mathbf{M}\text{þ}=\mathbf{B}\boldsymbol{\rho}$ for þ, which is formally $\mathbf{M}^{-1}\mathbf{B}\boldsymbol{\rho}$.

Example 11.6. Let $n=5$ and $\boldsymbol{\tau}=[0,0.2,0.4,0.6,0.8,1.0]$. Consider $f(t)=\cos(\pi t)$ sampled at these 6 points, and construct the matrices $\mathbf{M}$ and $\mathbf{B}$ as above. Notice that the derivatives at the endpoints are both 0. The matrix $\mathbf{M}$ is determined by the compact formula (11.86) being required to hold at all interior nodes. So we have

$$\mathbf{M} = \begin{bmatrix} 4 & 1 & & \\ 1 & 4 & 1 & \\ & 1 & 4 & 1 \\ & & 1 & 4 \end{bmatrix} \qquad \text{and} \qquad \mathbf{B} = 3 \begin{bmatrix} -1 & & 1 & & & \\ & -1 & & 1 & & \\ & & -1 & & 1 & \\ & & & -1 & & 1 \end{bmatrix} .$$

Of course, $\mathbf{B}$ times the 6-vector of function values gives a 4-vector of interior differences, suitable for solving our 4×4 system $\mathbf{M}\text{þ} = \mathbf{B}\boldsymbol{\rho}$. The solution of this linear system gives the approximate values

$$\begin{bmatrix} -1.844906087 & -2.985120747 & -2.985120747 & -1.844906087 \end{bmatrix}^T ,$$

which are in error by no more than $9.1 \cdot 10^{-4}$. This compares well with $h^4 = 1.6 \cdot 10^{-3}$. These are graphed with the exact derivative in Fig. 11.6. ◁

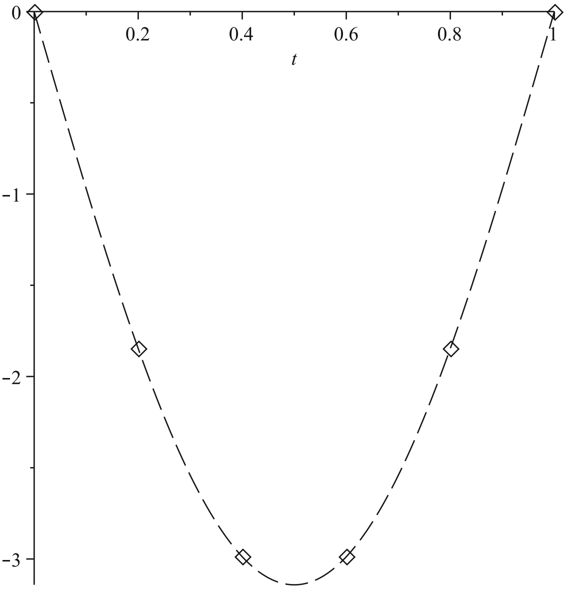


Fig. 11.6 Fourth-order uniform-mesh compact finite-difference derivative graphed with the exact derivative (*dashed line*). Even with just five subintervals, visual accuracy is achieved in this example

This formula is worth deriving from the contour integral approach. In what follows, we do so for a variable mesh. Consider

$$\frac{1}{(z+rh)^2 z^2 (z-sh)^2} = \frac{1}{r^2 h^4 (s+r)^2 (z+rh)^2} + \frac{4r+2s}{(z+rh)\, r^3 h^5 (s+r)^3}$$

$$+\frac{1}{r^2h^4s^2z^2}+\frac{2r-2s}{zr^3h^5s^3}+\frac{1}{h^4\left(s+r\right)^2s^2\left(z-sh\right)^2}+\frac{-4s-2r}{\left(z-sh\right)h^5\left(s+r\right)^3s^3},$$

from which we deduce the compact formula, exact for polynomials $f(x)$ of degree $6-2=4$ or less:

$$\begin{aligned}&\frac{1}{r^2\left(s+r\right)^2}f'(-rh)+\frac{1}{r^2s^2}f'(0)+\frac{1}{\left(s+r\right)^2s^2}f'(sh)\\&\qquad=-\frac{4r+2s}{r^3h\left(s+r\right)^3}f(-rh)-\frac{2r-2s}{r^3hs^3}f(0)-\frac{-4s-2r}{h\left(s+r\right)^3s^3}f(sh)\,.\end{aligned}\tag{11.87}$$

If we now have a mesh $\tau_0<\tau_1<\cdots<\tau_{n-1}<\tau_n$, we can lay this compact formula first on τ_0, τ_1, and τ_2, with $rh=\tau_1-\tau_0$ and $sh=\tau_2-\tau_1$, which, if we have a reference step width h, say $h=(\sum(\tau_{k+1}-\tau_k))/n$, gives us an equation relating the derivative values on these three mesh points to the function values on the mesh points. We then lay the formula over τ_1, τ_2, and τ_3, giving us another equation—one for each interior point, $\tau_1,\tau_2,\ldots,\tau_{n-1}$. The linear system for the unknown derivative values is tridiagonal; but we have $n-1$ equations and $n+1$ unknowns $f'(\tau_0)$, $f'(\tau_1)$, $\ldots$, $f'(\tau_n)$. We need two more equations. Since the error term in the (residual of) the right-hand side above is $h^4f^{(5)}(0)/5!+h^5f^{(6)}(0)(s-r)/360+O(h^6)$, we should look for fourth-order formulae at either end, giving equations involving τ_0 and its nearest mesh neighbors, and τ_n and its nearest neighbors. We will want the formulæ exact for polynomials of degree 4 or less. Since the matrix is so far tridiagonal, we try to keep it that way and we thus look for relations of the form

$$a_0f'(\tau_0)+b_0f'(\tau_1)=c_0f(\tau_0)+c_1f(\tau_1)+c_2f(\tau_2)+c_3f(\tau_3)\,.\tag{11.88}$$

This still qualifies as "compact," though, because we use only two extra mesh points at the left end, and similarly only two extra on the right, and these appear in the right-hand side and do not change the tridiagonality of the matrix. We suppose $n>4$ so that the formulæ make sense. This ansatz suggests looking at the partial fraction decomposition of

$$\frac{1}{\left(z-\tau_0\right)^2\left(z-\tau_1\right)^2\left(z-\tau_2\right)\left(z-\tau_3\right)},\tag{11.89}$$

from which we straightforwardly find (of course by using a computer algebra system) that

$$a_0=\frac{1}{\left(\tau_0-\tau_1\right)^2\left(\tau_0-\tau_2\right)\left(\tau_0-\tau_3\right)}\quad\text{and}\quad a_1=\frac{1}{\left(\tau_0-\tau_1\right)^2\left(\tau_1-\tau_2\right)\left(\tau_1-\tau_3\right)}$$

and that the c_ks are

$$c_0=-\frac{2\,\tau_2\tau_3-3\,\tau_0\tau_2-3\,\tau_0\tau_3+4\,\tau_0{}^2+\tau_1\tau_3-2\,\tau_0\tau_1+\tau_2\tau_1}{\left(\tau_1-\tau_0\right)^3\left(\tau_0-\tau_2\right)^2\left(\tau_0-\tau_3\right)^2}$$

$$
\begin{aligned}
c_1 &= -\frac{-2\,\tau_2\tau_3 + 3\,\tau_2\tau_1 + 3\,\tau_1\tau_3 - 4\,\tau_1{}^2 - \tau_0\tau_3 + 2\,\tau_0\tau_1 - \tau_0\tau_2}{(x_1-\tau_0)^3\,(\tau_1-\tau_2)^2\,(\tau_1-\tau_3)^2}\\
c_2 &= \frac{1}{(\tau_2-\tau_0)^2\,(\tau_2-\tau_1)^2\,(\tau_3-\tau_2)}\\
c_3 &= \frac{1}{(\tau_2-\tau_0)^2\,(\tau_2-\tau_1)^2\,(\tau_3-\tau_2)}\,.
\end{aligned}
$$

It turns out that the residual error in (11.88) is, as desired, $O(h^4)$. In detail, if $\tau_k = \tau_0 + r_k h$, for $k = 1, 2, 3$, then the residual error is $h^4 f^{(5)}(0)/120 + (2r_1 + r_2 + r_3)h^5 f^{(6)}(0)/720 + O(h^6)$, so even the error coefficient is the same at the end as it is for the interior nodes. A similar formula holds for the other end (indeed, simply reverse the labels, $\tau_k \leftrightarrow \tau_{n-k}$). This gives us closure in our search for a compact, variable-mesh fourth-order finite-difference formula.

So, how well does it work? If the mesh ratios are not "too large," that is, adjacent subintervals are not too different in width (so that the r_k factors in the edge formulæ and the r and s factors in the interior formula are not too large), then the formula is very effective for smooth functions. Note, however, that the formulæ above are quite susceptible to produce rounding errors, especially in the formulæ at the edges, and should be rewritten using $h_i = \tau_{i+1} - \tau_i$ and factored wherever possible. When this is done, the influence of rounding errors, while still felt, is significantly reduced. The program vcompact4 has been used to generate the data shown in Fig. 11.7:

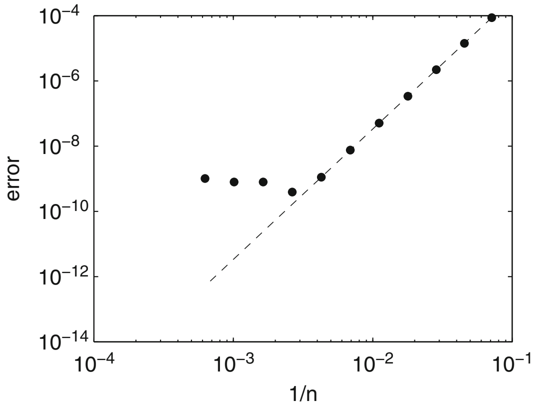


Fig. 11.7 Fourth-order variable-mesh compact finite-difference maximum error computing $(1/r)'(x)$ on $1 \le x \le 3$. The spatial mesh was Chebyshev points $x_j = 2 + \cos(\pi j/n)$ for $0 \le j \le n$, for various n. Theoretical fourth-order behavior is shown by the *dashed line*

11.7 Automatic Differentiation

Recall the Mandelbrot polynomials, defined by $p_0 = 1$ and $p_{n+1} = xp_n^2 + 1$. Symbolic expressions for the p_n grow exponentially (in the expanded monomial basis, since the degree is $2^n - 1$), and symbolic expressions for the derivatives of these polynomials also grow exponentially, being degree $2^n - 2$. In fact, symbolic derivative expressions grow remarkably quickly in general, unless serious care is taken not to reprint repeated subexpressions. Symbolic differentiation has its uses in numerical analysis, most notably to compute the Jacobian matrix of a set of algebraic equations—because such matrices are often sparse, and we typically only want one derivative—but for other contexts, and sometimes even for Jacobian matrices, symbolic differentiation is just too expensive. To add insult to injury, the resulting expressions are often not only large, but also numerically unstable.

Coming back to the Mandelbrot polynomials, a moment's thought shows that the derivatives $p'_n(x)$ can also be computed by a recurrence relation, because $p'_{n+1} = p_n^2 + 2xp_np'_n$. Consider, therefore, this algorithm:

```
$[p, dp] = \mathrm{mandelbrot}(x, n)$
$p_0 = 1$
$dp_0 = 0$
**for** $i$ from 1 by 1 to $n$ **do**
    $p_i = xp_{i-1}^2 + 1$
    $dp_i = p_{i-1}^2 + 2xp_{i-1}dp_{i-1}$
**end for**
```

Wouldn't it be nice if such a "differentiated" algorithm could be obtained without human intervention? It is indeed possible, and it is indeed nice. In fact, the material we have covered on formal power series generation of terms of Taylor series forms one part of such *automatic differentiation* (AD) algorithms.[4] We cannot do justice here to the refinements necessary to get automatic differentiation programs to be robust, efficient, and able to generate efficient code. Still, it's worth going over at least the simplest ideas in automatic differentiation, to distinguish it from symbolic differentiation and from approximate numerical differentiation methods such as finite differences.

The main idea of the forward mode of AD is nothing more than the formal power series algebra of Chap. 2. That is, when we add two truncated power series (expanded at the same point), we simply add their coefficients. When we *multiply* two such series, we use the Cauchy convolution. Division is handled similarly, as are most of the elementary functions, as described in that chapter. So how does this become automatic?

The key is *operator overloading*. In many modern computing languages (including MATLAB), one can redefine the operators $+$, $*$, $/$, and so forth, so that they do different things depending on the type of operand they encounter. If, say, they en-

[4] For a large collection of resources, including reference books, tutorials, lists of papers, and (perhaps most importantly) links to programs that can differentiate your programs in languages ranging from C through MATLAB to Python, see www.autodiff.org.

counter a pair of objects of type “truncated power series,” then they use the series algorithms. Otherwise, they perform the originally intended operations. An automatic differentiation program that provided series operations for plus, minus, times, divide, and power, together with most of the elementary functions such as sine, cosine, and exponential, could then take a user’s program and (without changing a line!) execute it in series instead! At the end, the result would be a series—which could be interpreted not only as the final result, but as all the derivatives of the final result. This approach works quite well if there are only a few inputs, and only one output.

As for the Mandelbrot example above, it was differentiated by hand, obviously. To better simulate automatic differentiation, we take instead the *original* algorithm, modify the constants so that they are now constant *series* (this can be done invisibly to the user, but we show it in the algorithm below for clarity), and modify occurrences of x in the program to be $[x,1]$ (because the derivative of x with respect to itself is 1). The algorithm becomes

```
$p = \mathrm{mandelbrot}(x,n)$
$p_0 = [1,0]$
**for** $i$ from 1 by 1 to $n$ **do**
    $p_i = [x,1] * p_{i-1}^2 + [1,0]$
**end for**
```

and we execute the algorithm with the understanding that the plus in $a+b$ really takes as input $[a,da]$ on the left, $[b,db]$ on the right, and produces $[a,da]+[b,db] = [a+b,da+db]$ as a final result. Similarly, $a*b$ produces $[ab, a\cdot db + da\cdot b]$ (which is just the first term in the Cauchy convolution).

All in all, it is a remarkably simple approach, and it is quite powerful even in the simplistic condition presented here. In comparison with numerical differentiation, one sees that there is no truncation error, although there is indeed rounding error. In comparison with symbolic differentiation, we see that at no point is a formula for the derivative generated—just a program (well, in this case, just an algorithm). However, automatic differentiation didn’t take off in applications or the literature until it became more powerful. The breakthrough was the *reverse mode*, which is effective for a large vector of inputs, that is, computation of gradients. We do not pursue this powerful idea here, but leave you to the references.

11.8 Smoothing

The discussion in this chapter has so far been confined to the case where we know the function to be differentiated very precisely. This is typical of functions defined by formulæ, or by programs that can be executed in high precision. Of course, this is useful, but it isn’t the only place where differentiation is needed. Indeed, the differentiation of noisy data in science is often desired. But as we have seen, differentiation is infinitely ill-conditioned. What, then, can be done?

There are several approaches used in practice. What we find simplest is the idea of "smoothing"—that is, we make some computational effort to filter out the smooth function to be differentiated. There are infinitely many ways to do this, however, and it would be nice if there were some theoretical basis underpinning the method used. One such theoretical underpinning is the idea of "regularization," that is, transforming the ill-conditioned problem into a well-conditioned one, in effect by restricting the class of allowable perturbations.

For the purposes of this chapter, we will examine a much older, "local," technique, due to Lanczos (1988). Lanczos' first smoothing technique is local because it considers only a few nearest neighbors in the sampled data and fits a *smooth* low-degree polynomial to that data. To be specific, we use a least-squares fit to the data, to fit a lower-degree and therefore smoother polynomial to the data, and differentiate that smoother polynomial. Given the method we favor in this book, though, we apply Lanczos' technique entirely in the Lagrange basis. In the end, we get formulæ very similar to Lanczos'. The formulation of the least-squares problem in the Lagrange basis is simple enough, and we take one approach below, by example.

Example 11.7. Suppose that we are given data y_k on $\tau_k = -1 + {}^{2k}/_{4}$, for $k = 0$, 1, 2, 3, and 4. We could fit a degree-4 Lagrange polynomial on that data easily enough, but if the data y_k contain errors, then we would be fitting the errors exactly, which doesn't help. So instead we try to fit a degree-2 polynomial on this data. Here we run into a snag with the Lagrange basis: The degree is not visible in the form. We can evaluate our degree-2 polynomial on the 5 pieces of data to get ρ_k, $0 \le k \le 4$, and indeed we will have to; and this will give us the barycentric forms of the interpolant, as usual.

But how do we specify the degree? It's contained implicitly in the ρ_k, to be sure, but it's not obvious. One way to do it is to impose the following linear constraints. First,

$$\beta_0\rho_0 + \beta_1\rho_1 + \beta_2\rho_2 + \beta_3\rho_3 + \beta_4\rho_4 = 0\,, \tag{11.90}$$

which ensures that the degree cannot be 4, but must be 3 or less, because this number is just the coefficient of $p(x)$ if we expand into the monomial basis. However, we are using only the given data and are not converting to the monomial basis; it would be the case *if we used this coefficient* (and the other monomial basis coefficients) to evaluate $p(x)$, and we would then be in trouble. There would be rounding errors in its computation, which would be worse than the original data errors. But we're *not* going to use the coefficient! In fact, we are constraining this coefficient to be *zero*, not computing it, and this makes all the difference. We will still find ρ_k on all 5 nodes; but by insisting on the constraint, we get ρ_k from a lower-degree polynomial.

We want to make it smoother yet as well. If we also apply the similar constraints that all of the data must be fittable by degree-2 polynomials on each of the five related sets of four points obtained by omitting one of our given τ_k, each in turn, then we get five more linear constraints. Of course, these are not linearly independent, and so we won't need to apply all of them. For this problem, all we need, in fact, is just one more, say that ρ_k on τ_k for $1 \le k \le 4$ must not be degree 3 but rather degree 2. We compute the barycentric weights on τ_1, τ_2, τ_3, and τ_4 (note that τ_0 has been omitted), and call these weights $\beta_{k\backslash 0}$ and number them from 1 to 4 corresponding to our old τ_k. Then we get the constraint

$$\beta_{1\backslash 0}\rho_1 + \beta_{2\backslash 0}\rho_2 + \beta_{3\backslash 0}\rho_3 + \beta_{4\backslash 0}\rho_4 = 0\,. \tag{11.91}$$

More, a short computation shows that $\beta_{k\backslash 0} = \beta_k(\tau_k - \tau_0)$, and so we needn't do much work in computing these related barycentric weights. (We see in Problem 11.14 a related approach.)

By applying these two constraints, we will ensure that our polynomial fit will have degree at most 2. Now, we wish to find the ρ_k that are the "best possible" fit to the given data, y_k. If we use least squares, then we wish to minimize

$$\sum_{k=0}^{4} (\rho_k - y_k)^2 \tag{11.92}$$

(assuming our data are real, for simplicity). But we must also satisfy the two constraints above: Denote them $\mathbf{b}_1\boldsymbol{\rho} = 0$ and $\mathbf{b}_2\boldsymbol{\rho} = 0$ for appropriate vectors $\mathbf{b}_j$. We may use Lagrange multipliers on our minimization problem to enforce the constraints. That is, we find $\boldsymbol{\rho}$ (and the multipliers λ_1 and λ_2) by minimizing

$$\sum_{k=0}^{4} (\rho_k - y_k)^2 + 2\lambda_1\mathbf{b}_1\boldsymbol{\rho} + 2\lambda_2\mathbf{b}_2\boldsymbol{\rho}\,. \tag{11.93}$$

Differentiating with respect to each ρ_k and to each λ_j gives us seven equations in the seven unknowns, with (for our $\boldsymbol{\tau}$) the symmetric positive-definite matrix

$$\mathbf{A} = \begin{bmatrix} 1 & & & & & 2/3 & 0 \\ & 1 & & & & -8/3 & -4/3 \\ & & 1 & & & 4 & 4 \\ & & & 1 & & -8/3 & -4 \\ & & & & 1 & 2/3 & 4/3 \\ 2/3 & -8/3 & 4 & -8/3 & 2/3 & & \\ 0 & -4/3 & 4 & -4 & 4/3 & & \end{bmatrix} \tag{11.94}$$

and right-hand side $[y_0, y_1, y_2, y_3, y_4, 0, 0]^T$. See Fig. 11.8 for a plot of a fit using this method.

Now, how do we take the derivative of this data? The usual contour integral method gives us that the derivative at $t = 0$ of the smoothed function is

$$f'(0) = \frac{1}{6}(\rho_0 - \rho_4) + \frac{4}{3}(\rho_3 - \rho_1)\,. \tag{11.95}$$

And now we note that the linear system described above *can be solved symbolically* for arbitrary data y_k, giving

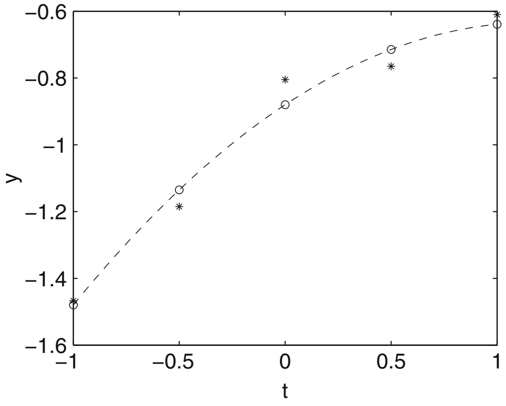


Fig. 11.8 The degree 2 least-squares fit to five samples of noisy data, done in the Lagrange basis. The smoothed data are $[-1.478, -1.137, -0.8812, -0.7103, -0.6245]$, while the original data are $[-1.467, -1.185, -0.8047, -0.7647, -0.6101]$, at nodes $\boldsymbol{\tau} = [-1, -1/2, 0, 1/2, 1]$. The *dashed line* is obtained by barycentric Lagrange interpolation. The derivative at $t = 0$ is, by this method, $f'(0) = 0.4203$

$$f'(0) = \frac{6}{35}y_0 - \frac{17}{35}y_1 - \frac{4}{7}y_2 - \frac{3}{35}y_3 + \frac{34}{35}y_4 . \tag{11.96}$$

Now, finally, we transport our set of nodes $\boldsymbol{\tau}$ onto a mesh of width h: $\tau = -1$ corresponds to $x = -2h$, $\tau = -1/2$ to $x = -h$, $\tau = 0$ to $x = 0$, and so on. The formula is $x/2h = \tau$, and therefore $d/dx = 1/(2h) d/d\tau$, so to get a *smoothing finite-difference formula*, we divide that formula by $2h$. This gives

$$f'(x) \doteq \frac{1}{2h}\left(\frac{6}{35}f(x-2h) - \frac{17}{35}f(x-h) - \frac{4}{7}f(x) - \frac{3}{35}f(x+h) + \frac{34}{35}f(x+2h)\right) .$$

Formally, this is exact only for linear polynomials and is only an $O(h)$ formula. It uses 5 points, not 2, which seems wasteful, but the extra information has been used to smooth the data. The same approach can be used to give formulæ accurate near the edges of a mesh (the above only works if one has two mesh points on either side). The approach can be extended to a variable mesh. ◁

Lanczos then goes on to point out a connection to quadrature, because this finite difference formula looks somewhat like a quadrature rule, apart from the division by h. The curious formula

$$f'(x) = \frac{3}{2\varepsilon^3}\int_{-\varepsilon}^{\varepsilon} t f(x+t)\,dt, \tag{11.97}$$

which is accurate to $O(\varepsilon^2)$, suggests that one can obtain an approximate derivative by instead computing an approximate integral. Of course, we have already seen this in the complex plane, but still, this entirely real integral is itself interesting.

Other methods of smoothing include trigonometric differentiation (see Eq. (9.29) in Chap. 9).

11.9 Multidimensional Finite Differences

Multidimensional finite differences require more bookkeeping. The short MAPLE program `findif` mentioned in Zhao et al. (2007) uses multivariate Taylor series in MAPLE to set up equations for the unknown coefficients in an ansatz for a desired differential expression. Unfortunately, this code has not been made widely available, but fortunately it is not hard to reimplement in a computer algebra system. Even FORTRAN implementations are rumored to exist!

Here, we only consider an example of a multivariate finite difference.

Example 11.8. Given information about the value of $f(x,y)$ at the five points $(x,y) = (0,0)$, $(x,y) = (\Delta_x, 0)$, $(x,y) = (-\Delta_x, 0)$, $(x,y) = (0,\Delta_y)$, and $(x,y) = (0,-\Delta_y)$, one might wish to evaluate, say, $f(r\Delta_x, s\Delta_y)$ for any $r \in [-1,1]$, $s \in [-1,1]$. This is an interpolation problem. The `findif` program generates the following formula:

$$\begin{aligned} f(r\Delta_x, s\Delta_y) &= \left(1 - r^2 - s^2\right) f(0,0) + \frac{1}{2} s(s+1) f(0,\Delta_y) \\ &\quad + \frac{1}{2} s(s-1) f(0,-\Delta_y) + \frac{1}{2} r(r+1) f(\Delta_x, 0) + \frac{1}{2} r(r-1) f(-\Delta_x, 0) \,. \end{aligned}$$

The principal term in the error for this formula is $rsD_{12}(f)(0,0)\Delta_x\Delta_y$.

If, on the other hand, one wanted a derivative, say $D_{12}(f)(r\Delta_x, s\Delta_y)$, then this is a finite-difference problem (and there may or may not be enough information on this "molecule" to specify it). ◁

11.10 Notes and References

Theorem 11.1, that differentiation is infinitely ill-conditioned, is well known. Several early computational discussions exist, for example, Lanczos (1988) and Cullum (1971), but it seems obvious that the phenomenon must have been known long before those. The study of finite differences is probably even older; by the time of Boole (1880), it was already very advanced and by Milne-Thomson (1951) must have seemed mature indeed. The coming of the digital computer completely revolutionized it, of course.

See Trefethen (2000 Ch. 6) for examples of differentiation matrices in action. In Trefethen (2013), we find a detailed discussion of the history of differentiation matrices, and there the first derivation of Eq. (11.22) is attributed to Bellman et al. (1972). The special care needed to get differentiation matrices accurate for representations of constant vectors was first noted by Don and Solomonoff (1995). Significant improvements can be made to formula (11.22) to protect it from rounding errors (and incidentally to improve its speed) in the case where the nodes come from a known family, such as the Chebyshev nodes (actually, in that case the weights are known analytically—see Problem 8.24). See Weideman and Reddy (2000) and their MATLAB codes at `http://www.mathworks.com/matlabcentral/fileexchange/29` and the earlier work Don and Solomonoff (1995). The use of complex nodes for differentiation was studied first in Lyness and Moler (1967).

Compact finite differences are studied in Lele (1992) using a Fourier approach, and their exceptional accuracy for minimal effort—nearly spectral accuracy, in fact—on uniform grids is explored in that paper. Compact schemes can be very accurate and very efficient in practice. See Zhao et al. (2007) for an application in option pricing, and see Rokicki and Floryan (1995) for an application in fluid mechanics. Compact schemes for variable meshes in fluids applications are studied, for example, in Pettigrew and Rasmussen (1996).

For a full introduction to automatic differentiation, consult Corliss et al. (2002).[5] For an introduction to AD in MATLAB, see Forth (2006).

The seminal use of regularization for smoothing in order to take derivatives is the work of Cullum (1971), where she converted the problem of finding the derivative $F'(x)$ from $F(x)$ into finding the solution $f(x)$ of the Volterra-type integral equation

$$A(f)(x) := \int_0^1 u(x-y)f(y)\,dy = F(x) - F(0)\,, \tag{11.98}$$

where $u(t)$ is the Heaviside unit step function, a problem that can be *regularized* by Tikhonov's procedure into the following minimization problem: Find f so that

$$\|A(f) - F\|^2 + \left(\int_0^1 f\right)^2 + \alpha\left(\|f\|^2 + \|f'\|^2\right) \tag{11.99}$$

is minimized. [As phrased here, this requires $F(0) = F(1) = 0$, but this restriction is easily overcome.] This problem is well-conditioned: Small changes in the data F produce small changes in the answer f, because the norms control the derivatives. One difficulty is that the problem contains an unknown *regularization parameter* α which must be estimated well, in order to get good answers. However, this approach has seeded a very large number of (dare we say it) derivative works, and overall the approach is very practical. One interesting refinement is to consider averages of finite differences taken at several different mesh widths, in such a way that the mesh widths used become the regularization parameter. For a discussion of this and alternative approaches, see Lu and Pereverzev (2006). A very interesting observation is that of Anderssen and Hegland (1999), who show that higher-dimensional problems are in some sense easier. These regularization techniques are "global" in the sense that they use all the data available in order to find the derivative at any one point. See Anderssen and Hegland (2010) and the references therein for a selection of applied problems requiring smoothing.

[5] Note the spelling of George F. Corliss' name.

Problems

Theory and Practice

11.1. Given the values of $y = f(x)$ at $x = 0$, $x = h$, and $x = 2h$, and the value of $y' = f'(x)$ at $x = 0$ only, find a formula for

$$f''(0) \doteq \frac{a_0 f_0 + a_1 f_1 + a_2 f_2}{h^2} + \frac{b_0 f_0'}{h} \tag{11.100}$$

that is as accurate as possible in the asymptotic limit as $h \to 0^+$. You may use either Taylor series or contour integrals to derive your formula. Test your formula on

1. $f(x) = \cos(\pi x)$
2. $f(x) = x^{5/2}$

for $h = 1/50$, $1/80$, $1/130$, $1/210$, $1/340$, $1/550$, $1/890$, $1/1440$, $1/2330$, $1/3770$, $1/6100$. What is the order of your formula? That is, your error should be $O(h^p)$ for some p. What is p? Explain your results.

11.2. It can be shown that if $c = 2 + \sqrt{3}$ and the tridiagonal matrix $\mathbf{M}_1$ has ones on the subdiagonal and superdiagonal, and the diagonal elements are 4 except for the top left corner, which is c, not 4, then $\mathbf{M}_1$ can be factored exactly (analytically) once and for all, independent of its dimension (so long as it's square) into $\mathbf{M}_1 = \mathbf{LDL}^T$.

1. Factor $\mathbf{M}_1$ as mentioned.
2. If (in MAPLE syntax), we write a matrix–vector product $\mathbf{B}_1\mathbf{u}$ as

```
c := 2.0 + sqrt(3.0);
alpha := 1.0/c;
b := Array(0..n); # same as u
b[0] := (-(25*c+3)*u[0] + (48*c-10)*u[1] + (18-36*c)*u[2]
     + (16*c-6)*u[3]
             +(1-3*c)*u[4] )/12/h;
for k to n-1 do
   b[k] := 3*(u[k+1]-u[k-1])/h;
end do;
b[n] := (11*u[n-4] - 58*u[n-3] + 126*u[n-2] - 182*u[n-1] +
      103*u[n])/12/h;
```

   then $\mathbf{M}_1\mathbf{v} = \mathbf{B}_1 f$ has $\mathbf{v} = f'(x) + O(h^4)$ if the vector being multiplied by $\mathbf{B}_1$ is, in fact, the value of a function f evaluated at a vector $\mathbf{x}$ of equally spaced points on an interval $a \le x \le b$. Translate that MAPLE syntax into MATLAB.
3. Use the factoring of 11.2 and the translation of 2 to write a MATLAB program that accepts as input a vector of function values (evaluations of f at equally spaced points) and a width $h = (b-a)/n$ and outputs a vector of derivative values. *Your program should not form any matrices explicitly.* Instead, you should have a loop for forward elimination, perhaps a loop for solving a diagonal system, and a loop for back substitution. Test your program on the smooth function $y = \sin(\pi x)$ on the interval $-1 \le x \le 1$, for the same Fibonacci scale h as in

Problem 11.1, and plot the maximum error versus h on a log–log plot. Is the error really $O(h^4)$?

11.3. Show that the norm of the differentiation matrix $\mathbf{D}$ of the Lagrange basis on equally spaced nodes on $[-1,1]$ grows faster than exponentially as the number of subintervals n grows, while the norm of the differentiation matrix on Chebyshev–Lobatto nodes $\cos{}^{k\pi}\!/_{n}$ grows only as $O(n^2)$. Compare to the norm of the differentiation matrix in the monomial basis.

11.4. Using the contour integral

$$\oint_C \frac{f(z)}{(z+{}^h\!/_2)z^2(z-{}^h\!/_2)}\,dz \tag{11.101}$$

or otherwise derive the central difference approximation to the derivative

$$f'(0) = \frac{f({}^h\!/_2)-f(-{}^h\!/_2)}{h} \tag{11.102}$$

and show that it is exact for polynomials of degree at most 2.

11.5. Show (perhaps by example) that Eq. (11.14) is in the case of Lagrange basis with Vandermonde matrix $\mathbf{V}$ as produced by MAPLE

$$\mathbf{D}_{\text{Lagrange}} = \mathbf{V}\mathbf{D}_{\text{monomial}}\mathbf{V}^{-1}\,. \tag{11.103}$$

Actually, using this formula to compute the differentiation matrix does not seem advisable since Vandermonde matrices are notoriously ill-conditioned for real interpolation nodes.

11.6. Verify that Eq. (11.61) suffers accuracy loss as $h \to 0$ for $f(z) = J_0(z)$ (`besselj(0,z)` in MATLAB). By using the Cauchy–Riemann equations and $J_0'(z) = -J_1(z)$, compute the separate condition number for the imaginary part $v(x,y)$ of $J_0(x+iy) = u(x,y)+iv(x,y)$ at $x = 22.3$, $y = h$. If perturbations in x happen, is the imaginary part ill-conditioned?

11.7. Explore the accuracy of computing n derivatives of various analytic functions by means of the FFT as in Remark 11.4. Give a formula for the order of accuracy you can expect.

11.8. Another reason not to discard the Newton basis (divided differences) completely is their convenience for differentiation. Show how to efficiently differentiate a polynomial expressed in a Newton basis.

Investigations and Projects

11.9. Is the algorithm given in the text for computing the differentiation matrix **D** on the Lagrange basis a numerically stable algorithm? What about the algorithm for computing **D** on a Hermite interpolational basis?

11.10. Get a copy of the paper (Lele 1992) and confirm the claims therein of accuracy of compact finite differences on uniform grids for trigonometric functions.

11.11. Suppose one has a polynomial given by Lagrange or Hermite data, $\rho_{i,j}$ at distinct nodes τ_i. One might not want to compute a large number of derivatives; perhaps one only needs a few at some points t not equal to any τ_i. In that case, the construction and use of an entire differentiation matrix might be overkill (convenient if one has the routines ready to hand, but still overkill). One could instead differentiate by constructing an explicit derivative from the partial fraction expansion of

$$\frac{1}{(z-t)^2 \prod_{i=0}^{n} (z-\tau_i)^{s_i}}, \tag{11.104}$$

which gives a formula connecting $f'(t)$, $f(t)$, and all the $\rho_{i,j}$; once one has evaluated $f(t)$ which presumably one needs anyway then the derivative $f'(t)$ becomes available. Construct this formula and test it.

An alternative that does not require the value of $f(t)$, apparently, is to choose a Butcher factor $B(z)$ so that the residue of $1/(z-t)$ is zero in

$$\frac{B(z)}{(z-t)^2 \prod_{i=0}^{n} (z-\tau_i)^{s_i}}. \tag{11.105}$$

This connects $f'(t)$ directly to the $\rho_{i,j}$. Which approach is better?

11.12. Construct a differentiation matrix for taking derivatives of the *rational* interpolant with denominator $q(z)$ where

$$\frac{q(z)}{w(z)} = \sum_{i=0}^{n} \sum_{j=0}^{s_i-1} \frac{\alpha_{i,j}}{(z-\tau_i)^{j+1}}. \tag{11.106}$$

Test your formula on some small examples. Hint: Does the derivation of the differentiation matrix for the Lagrange basis or Hermite interpolational basis actually use the fact anywhere that the barycentric weights define a polynomial interpolant, as opposed to a rational interpolant?

11.13. Write a MATLAB function to compute the differentiation matrix **D** according to Eqs. (11.22) and (11.16) for interpolation on a set of distinct nodes τ_i, $0 \le i \le n$, with a given set of barycentric weights β_i. Your program will also be useful for differentiation of *rational* interpolants simply by inputting nonpolynomial barycentric weights α_i. Test your program on several examples. The cost of your program should be at most $O(n^2)$ flops to construct **D**. Try to write a short program, avoiding loops as much as is reasonable.

11.14. Piers Lawrence pointed out to us a formula that allows the constraints used in Sect. 11.8 to be simplified. Specifically, if the leading coefficient $\sum_{k=0}^{n} \beta_k \rho_k = 0$, then the next coefficient is

$$[z^{n-1}](p) = \sum_{k=0}^{n} \beta_k \tau_k \rho_k \,, \tag{11.107}$$

and if *that* is zero, then the *next* coefficient is

$$[z^{n-2}](p) = \sum_{k=0}^{n} \beta_k \tau_k^2 \rho_k \,, \tag{11.108}$$

and so on. That is, if all of $[z^{n-j}](p) = 0$ for $0 \le j \le m-1$, then the next coefficient is

$$[z^{n-m}](p) = \sum_{k=0}^{n} \beta_k \tau_k^m \rho_k \,. \tag{11.109}$$

Use these formulæ to write a MATLAB function that fits a degree-$(n-m)$ polynomial to $n+1$ pieces of data (τ_k, y_k) in the least-squares sense. Either by updating the barycentric weights and throwing away m pieces of data or simply by using the original differentiation matrix on your computed ρ_k, include the ability to find the derivative of your smoothed polynomial. Will this process work well if m is at all large?

11.15. Suppose we have a differentiation matrix $\mathbf{D}$ available; for simplicity, take the differentiation matrix on the Chebyshev–Lobatto nodes. If the number of nodes is large enough to represent f but we know its derivative g on those nodes, can we solve $\mathbf{Df} = \mathbf{g}$ for $\mathbf{f}$, perhaps by using the SVD?

11.16. Compute the condition numbers $\tau_k \partial \beta_{i,j} / \partial \tau_k / \beta_{i,j}$ for a few explicit sets of nodes and confluencies. Are the entries of the Lagrange differentiation matrix at all ill-conditioned? Note that we are not talking about the condition number of $\mathbf{D}$ itself (which is infinite, because the matrix is singular).

11.17. Write an MATLAB program with the header `[vpp] = compact4second ( rho, tau )` that accepts as input a vector $\boldsymbol{\rho}$ of values of $f(z)$ on a vector of distinct values $\boldsymbol{\tau}$ and returns a vector of approximate second derivatives, where the second derivatives are computed by an $O(h^4)$ compact formula, which you will have to derive. Your program (and formula) should work on meshes that do not necessarily have constant step size. For simplicity, you may suppose that the second derivatives are known at τ_0 and at τ_n. Compare your results with using the program in the text twice. Hint: Use a contour integral but nondimensionalize first. If you consider the nodes $[\tau_{k-1}, \tau_k, \tau_{k+1}]$, then the nondimensionalization $z = \tau_k + \theta h$, where $h = \tau_k - \tau_{k-1}$ and $\tau_{k+1} = \tau_k + rh$ defines the mesh width h and the mesh ratio parameter r is very helpful. For the contour integral, consider

$$\frac{B(\theta)}{(\theta+1)^3 \theta^3 (\theta - r)^3}, \tag{11.110}$$

where $B(\theta) = b_0 + b_1\theta + \theta^2 + b_3\theta^3$ is the Butcher factor containing parameters that must be chosen to force the coefficients of ${}^1\!/\theta^2$, ${}^1\!/(\theta+1)^2$, and ${}^1\!/(\theta-r)^3$ to be zero. Note the somewhat unusual normalization of the quadratic term—this was chosen after some preliminary computations because it makes the algebra nicer (even in a computer algebra system, which we recommend you use).

11.18. In Chebfun, the MATLAB command `diff`, which ordinarily just computes forward differences $y_{k+1} - y_k$, takes on another meaning (by operator overloading). The command `diff(y)`, where y is a chebfun, produces another chebfun for the derivative of the function represented by the chebfun y. There are two natural methods it could do this: It could form the differentiation matrix $\mathbf{D}$ on the Chebyshev–Lobatto nodes from the analytically known barycentric weights and use that; or it could convert to the Chebyshev series with the FFT, use the differentiation matrix in the Chebyshev basis, and then convert back. Which is a better method? How does Chebfun actually do it?

11.19. During the writing of this book, an engineering PhD student came and asked how to differentiate noisy data; in fact, a second derivative was needed. The application was to soil engineering. The data were given on a nonuniform grid. Write a program that uses Lanczos' approach as in Example 11.7 to take in a vector of nodes $\boldsymbol{\tau}$ and a vector of values $\boldsymbol{\rho}$ and produce a smoothed second derivative at the interior nodes. You will need to use a higher degree and more neighboring points than was used in that example. If the average width between nodes is h, what order p in $O(h^p)$ is the accuracy of your method? The data are as below: The left-hand column is position x and the right-hand column is the strain-gauge reading $\boldsymbol{\rho}$. The function is assumed to be *even*.

$$\begin{array}{lr} 1.725 & 73.430 \\ 1.035 & -86.515 \\ 0.975 & -90.475 \\ 0.405 & -138.670 \\ 0.015 & -154.984 \end{array} \tag{11.111}$$

11.20. Consider applying the compact scheme

$$f'(\tau_{k-1}) + 4f'(\tau_k) + f'(\tau_{k+1}) - \frac{3}{h}(f(\tau_{k+1}) - f(\tau_{k-1})) = 0 \tag{11.112}$$

to the problem $f(t) = \cos(\omega t)$ on some interval, with equally spaced nodes having $\tau_k - \tau_{k-1} = h$. The exact derivative is, of course, $f'(t) = -\omega \sin(\omega t)$. By substituting the *exact* solution into the compact scheme, we get a kind of "dual residual" [whereas if we could interpolate the approximate solution and put that into an equation, say $\int y(t)\,dt - \cos(\omega t) = 0$, we would be computing a residual as we normally do]. Specifically,

$$r(t;h) = f'(t-h) + 4f'(t) + f'(t+h) - 3(f(t+h) - f(t-h))/h\,. \tag{11.113}$$

The dual residual can be considered a kind of residual itself, by thinking of the reference solution $-\omega\sin(\omega t)$ as an approximate solution to the difference equations in the compact scheme, and its nearness to zero is also measure of accuracy, albeit less physically interpretable. The condition number of the compact difference equations is also less understood than the condition number of the differentiation problem. This type of analysis is actually quite common in the study of numerical methods.

Here, show that the dual residual is small compared to the reference solution as $h \to 0$ and in fact is $O(\omega h)^4$. Show that the dual residual for the central difference formula is, in contrast, $O(\omega h)^2$.

Part IV
Differential Equations

This book takes the somewhat unusual perspective that numerical methods for the solution of differential equations produce continuous, even continuously differentiable, solutions and not merely a discrete set or ‘skeleton’ of solution values. This habit of thought, which we encourage, is possible nowadays because professional-quality solvers already provide access to accurate interpolants together with the solution values.

Codes do this for several reasons. The first reason to provide interpolants came from a desire to improve graphical output: accurate codes can often take time-steps so large that plotting just discrete values (the ‘skeleton’) gives too sparse a plot for easy interpretation (and connecting the bones of the skeleton with straight lines gives inaccurate intermediate points and an incorrect impression of poor accuracy at the skeleton) (Fig. IV.1).

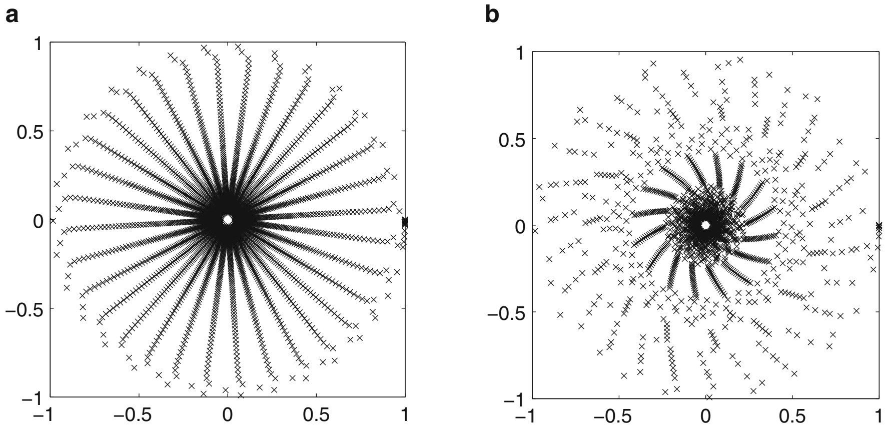


Fig. IV.1 Solving differential equations without interpolating the numerical solutions sometimes makes the output hard to interpret. See problem 12.1. (**a**) `ode45`. (**b**) `ode113`

The second reason came from a desire to allow codes to locate events, such as when one component of the solution becomes zero, or when the solution becomes singular. A time-step could be so large as to miss the event, and interpolants seem necessary. Yet another reason comes from the desire to solve not just initial-value problems for ordinary differential equations, but also to solve delay differential equations; interpolants into the past history of the solution then become necessary.

Enright and others then pointed out that if codes were providing interpolants anyway, it might be sensible to use them for error control. In a series of papers, Enright and co-workers developed defect (residual) control strategies for Runge–Kutta methods, and implemented them in high-quality Fortran codes. Once that was done, the advantage of *thinking continuously* about the solution became evident: a good numerical method gives you the exact solution of a nearby (a *quantitatively* nearby) problem. This provides a cheap and reliable alternative to so-called “shadowing” techniques for chaotic problems, for instance.[6] Important works about the development of residual control methods include Hanson and Enright (1983), Higham (1989a), Enright (1989a,b, 2000b, 2006a,b), Enright and Higham (1991), Enright and Muir (1996), Kierzenka and Shampine (2001) and Enright and Hayes (2007).

Interesting issues with structured backward error, that is, taking into account the kind of perturbation that the numerical method induces, are still being investigated. However, the main advantage—for a large class of initial-value problems for ODEs—of thinking continuously is that numerical methods can be seen to deliver an exact solution, and in many cases precisely as good a solution as an analytic exact solution of the original problem, when physical perturbations of the original problem are taken into account. In other words, thinking about continuously differentiable solutions allows us to use backward error analysis. This way of thinking allows one to conceptually unify the error analysis of numerical solutions of differential equation with other areas of numerical analysis to a much higher degree than what could have otherwise been done.

Chapter 12 looks at the MATLAB codes as exemplars of good numerical methods that provide continuous solutions. We use those to introduce in detail the ideas discussed in the previous paragraphs, together with the crucial idea of the *conditioning* of an IVP. This is called 'stability' in the differential equations literature, but to a numerical analyst 'stability' means numerical stability: an algorithm is numerically stable if the residual is small and hence the backward error is small; a problem is well-conditioned if small changes in the problem lead to small changes in the solution.

In Chap. 13, we look in some detail at methods for constructing the skeleton of a solution, and some details of interpolation. We take a brief look at geometric integration methods, which preserve special properties. We also consider Taylor series methods, which provide the fundamental theory for all numerical methods for ODE. In the early days, Taylor series methods were eschewed in practice even for smooth problems, apparently because of their relative complexity of programming. Yes, it is true that it is easier to construct a discrete stepping method than to construct a program-generating Taylor series method. However, once one adds interpolants, variable-stepsize methods using error control, and variable-order the level of programming complexity becomes comparable; that is, the extra programming complexity needed for Taylor series methods buys free interpolants, essentially free variable order, free error estimates (either local error or residual error), and as they are one-step methods they are perfect to adapt stepsizes with. In addition, implicit Taylor series methods have been studied since the work of Barton (1980), and Taylor series methods are very successful for differential-algebraic equations

[6] A numerical orbit is 'shadowed' by a reference orbit if there is an orbit of the reference problem starting from some possibly slightly different initial condition that uniformly follows the computed orbit. This, too, is a kind of backward error analysis, although mixed in that a small forward error is also allowed (if it's uniformly small for all time $t > t_0$); note that the only kind of backward error that is allowed is in the initial conditions. This may be more appropriate than a residual-type analysis if one is supremely confident that all terms and parameters have been included correctly in the model equations, which are not allowed to change. It is certainly more expensive than the type of analysis advocated in this book.

(e.g., the early work by Chang and Corliss (1994), and later work by Nedialkov and Pryce (2005, 2007) and co-workers). However, codes for numerical methods for the solution of initial-value problems for ODEs have evolved based on the design decisions of their first workers, and so most professional codes nowadays are multistep methods, extrapolation methods, or Runge–Kutta methods, with Taylor series methods confined to being used only for ultra-high-accuracy solutions for very smooth problems. Indeed the efforts of so many smart people working on the mainstream methods has produced a large number excellent codes that use either multistep, extrapolation, or Runge–Kutta methods by preference.[7]

Which method is best for your problem? Well, it's a bit like the game of rock-paper-scissors; there are problems for which each of the known methods is best, and others for which it is worst. The perspective taken in this book is that you will learn how to asses the solution afterwards, to see whether or not the code did a good job.

We go on from IVP and methods for IVP to the study of numerical methods for Boundary Value Problems for ODE (BVP, or BVPODE) in Chap. 14; here the influence of the boundary conditions is seen to be crucial. With Delay Differential Equations as studied in Chap. 15 we meet our first infinite-dimensional problems, but we will find the ideas of residual and conditioning just as useful. The book ends with Chap. 16, which uses the same techniques as are used to understand numerical methods for ODE and DDE to understand some simple methods for partial differential equations.

[7] You may find many of these high-quality codes, which are usually freely available, by consulting the Guide to Available Mathematical Software (GAMS) at `http://gams.nist.gov/` or directly at `http://www.netlib.org/ode/`.

Chapter 12
Numerical Solution of ODEs

Abstract We use a *continuously differentiable representation* of the numerical solution in order to define a *residual*, also known as a defect or deviation. This allows the use of *backward error analysis* on the numerical solution of *initial-value problems (IVP) for ordinary differential equations*. Of course, this means that we should also examine the *conditioning or sensitivity of an IVP*. We use the MATLAB ODE Suite as an example of high-quality software. We pay some attention to *chaotic problems*, *stiff problems*, and *singular problems*. ◁

This chapter has two objectives. The first is to explain the use of some simple but high-quality codes that are available in MATLAB for solving differential equations. We take this explanatory approach both because the codes can conveniently be used to solve scientific and engineering problems of genuine interest and because these problem-solving environment codes provide a relatively gentle introduction to general-purpose codes, which are used to solve very large-scale scientific problems efficiently and accurately.

The second objective is to introduce the concepts and the perspective by which we determine when to trust (and when to distrust) numerical solutions of ODE, in general. These lessons will hold true for large-scale codes as well as the MATLAB codes. Since numerical solution is the principal tool we have for nonlinear problems, and sometimes the *only* tool, this occupies an important place in the more general practice of scientific modeling. To understand these lessons, we adopt a backward error perspective centered on the concept of *residual*, as outlined in Chap. 1.

The residual of a differential equation is also called the *defect* and the *deviation* in the literature. We will sharpen and standardize those variant terms in order to make a useful distinction among classes of residuals. The use of residuals for ODE is old, going back at least to Cauchy. However, the notion of residual has not been used as much as it could have been in numerical analysis, in spite of its distinguished

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_12,

history. Within the last few decades, however, this has begun to change, and we believe that the right approach to the pedagogy and the usage of numerical methods for the solution of ODE includes the residual.

But before we talk about the residual, we should ask why one would use numerical methods at all. After all, analytical techniques are well studied and can produce great insight into a problem. Similarly valuable, perturbation series solutions and asymptotic series solutions of differential equations are also occasionally available.[1] The results of these analytic or semianalytic methods can sometimes be better—especially at singular points—and give more insight than numerical methods can, sometimes with greater ease; when these analytical and semianalytical techniques work at all, that is. In fact, perturbation and asymptotics are *complementary* to numerical methods: They work well when numerical methods have trouble at singular points, and they are cumbersome and inefficient at regular points.

Example 12.1. As a modest but typical example, consider the following simple-looking initial-value problem (we use x' and $\dot{x}$ to denote ${}^{dx}/_{dt}$):

$$\dot{x}(t) = t^2 + x(t) - \frac{x^4(t)}{10}, \qquad x(0) = 0, \tag{12.1}$$

on, say, the interval $0 \le t \le 5$. Differential equations of this kind can easily be found in applications; although this is a made-up problem, one could imagine that it had to do with population growth, where t is the time, and where x represents a population with spontaneous generation and a power-law death rate. If one tries to find the exact solution of this equation, then difficulties arise:

- The exact solution of this problem is not expressible in terms of elementary functions or known special functions.
- High-order power series solutions (see Sect. 13.4) have numerical evaluation difficulties for large $(t-a)$ caused by catastrophic cancellation, similar to the difficulties suffered by other functions we have seen before such as AiryAi(x) or e^{-x}, due to "the hump" phenomenon (see Chaps. 1 and 2). This can be repaired by the use of analytic continuation, but that is exactly the paradigmatic numerical method, as we shall see.
- Perturbation methods applied to this problem are explored in Problem 12.3, and they can indeed be used to predict the behavior for large times t: $x \sim (10t^2)^{1/4}$, although the higher-order terms are awkward. But this approximation is not good for $0 \le t \le 5$, say; it is only for "large" t.

Even for the problem posed in Eq. (12.1), which is vastly simpler than problems that occur in real models, the classical solution techniques essentially fail us. In contrast, numerical solution on (for example) $0 \le t \le 5$ is simplicity itself when we use state-of-the-art codes such as MATLAB's `ode45` to get a reliable numerical solution. For this generality and ease of use for many applications, numerical solution will be considered the solution method *par excellence*. ◁

[1] Incidentally, the notion of residual and backward error analysis applies equally well to series solutions, perturbation series solutions, and asymptotic series solutions of ODE.

12.1 Solving Initial-Value Problems with ode45 in MATLAB

MATLAB's ode45 is one of four IVP codes that we will use routinely in this book. The others are ode15s, ode113, and occasionally ode23. We use them to introduce the reader to the idea of using professional codes.

MATLAB's ode45 requires three arguments:

1. a *function handle* corresponding to $\mathbf{f}(t,\mathbf{x})$ in $\dot{\mathbf{x}} = \mathbf{f}(t,\mathbf{x})$; ($t$ is a scalar and $\mathbf{x}$ is a *column* vector, and the function must also return a column vector).
2. a *time span*, that is, a 1×2 vector corresponding to the interval over which we will solve the equation numerically;
3. an *initial value* (which is also a column vector).

Numerical initial-value problems (IVP) are in practice quite different to theoretical ones. One important difference is the need for an interval of computation, the time span: Theoretically, one could integrate forever, or until one hit a singularity—but in practice, the time span can make the difference between a "stiff" (and difficult) problem and an easy one, and the direction of integration can make the difference between a well-conditioned problem and an ill-conditioned one. These issues will be discussed as they arise (see, e.g., Problem 12.4).

For the problem described in Eq. (12.1), the dimension of the column vector is just 1, and we can simply execute

```
f = @(t,x) t^2 + x - x^4/10;
ode45( f, [0,5], 0 );
```

These commands[2] will dynamically generate the plot displayed in Fig. 12.1. See Problem 12.2, which asks you to solve this same problem again using two other MATLAB solvers, ode113 and ode15s. This shows how simple the use of these solvers can be.

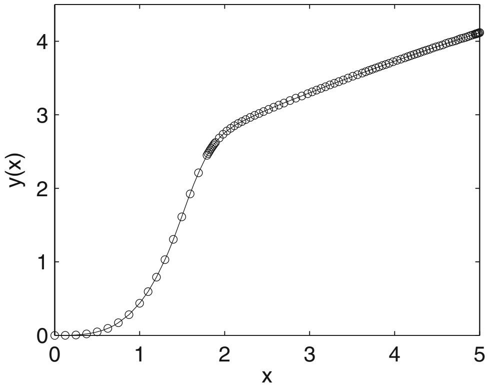


Fig. 12.1 Numerical solution of Eq. (12.1)

[2] Note the conventional ordering: (t,x). We have seen people get incorrect answers because they mixed the order up.

Note that the solver has apparently produced a *continuously differentiable* function as a solution and plotted it (see Problem 12.24). In fact, this graph has been generated step by step, using discrete values of time and values of the solution at each time unit. This discrete set is called the *skeleton* of the solution. Between each pair of points in the skeleton lies a separate function, each of which are collected and pieced together to produce a piecewise continuous function on the interval of integration.

This leads us to an important *notational convention*. We denote this continuous approximate solution of our initial-value problem by $z(t)$ or in the case of a vector system $\mathbf{z}(t)$. We will call the reference solution $x(t)$ or $x_{\text{ref}}(t)$ for emphasis; in the case of vector systems, $\mathbf{x}(t)$ or $\mathbf{x}_{\text{ref}}(t)$. For convenience and consistency with other works, for particular low-dimensional problems, we occasionally use scalar variables $x(t)$, $y(t)$, and $z(t)$ for individual components of either the reference solution or of the computed solution; hopefully no confusion will arise.

For many purposes, one will instead want to execute `ode45` in MATLAB with a left-hand side, for instance,

```
sol = ode45( @odefun, tspan, x0 );
```

For the problem in Eq. (12.1), the resulting solution `sol` will then be a structured *object* of this kind:

```
sol =
     solver: 'ode45'
    extdata: [1x1 struct]
          x: [1x25 double]
          y: [1x25 double]
      stats: [1x1 struct]
      idata: [1x1 struct]
```

Note that one could also write `[t,z] =` **ode45**`(@odefun,tspan,x0)` in order to put the values of `sol.x` and `sol.y` directly into other variables `t` and `z`, which is sometimes useful. The MATLAB command `odeexamples` can be considered a good go-to reference for the use of basic ODE codes.

We will see that this "solution object" can be used in the same way that a formula can be used, in that the computed solution $z(t)$ and its derivative $\dot{z}(t)$ can be evaluated at any desired point in the interval `tspan`. For now, we note that the points contained in the array `sol.x` are called the "steps," "nodes," or "mesh" t_k for (here) $1 \leq k \leq 25$, and the points in the array `sol.y` are the corresponding values of $z(t_k)$ (these were plotted with the circles in Fig. 12.1). In what follows we occasionally refer to the nodes `sol.x` together with the values `sol.y` as the *skeleton* of the solution, as stated previously.

We also note immediately that the value of z and $\dot{z}$ are available at off-mesh values of t, by use of the `deval` function, which automatically provides accurate interpolants and their derivatives for all solutions provided by the built-in solvers of MATLAB.

In other books, especially older books, a numerical solution to an IVP is considered to consist merely of a discrete mesh of times (i.e., `sol.x`), together with the corresponding values of x at those times (i.e., `sol.y`). That is, the numerical solution in other books is usually just what we call the skeleton of the solution, here. In those books, while in contrast an analytic solution to an IVP is a function for which we know the rule or formula, a numerical solution is merely a discrete graph. Nowadays it is different: Since there are interpolation methods widely implemented for evaluating the numerical solution at any point, and its derivative if we choose to ask for it, the distinction between an analytic solution and a numerical solution is not so great. A numerical solution need not just be a skeleton, and will not be in this book.

In the above example, the use of MATLAB's `ode45` required no preparation of the problem. However, it often happens that if one wants to use standard codes to solve a problem numerically, one has to rearrange the problem so that it is in a form that the code can process. The *standard form of an initial-value problem* is

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}(t)), \qquad \mathbf{x}(t_0) = \mathbf{x}_0, \tag{12.2}$$

where $\mathbf{x}: \mathbb{R} \to \mathbb{C}^n$ is the vector solution as a function of time, $\mathbf{x}_0 \in \mathbb{C}^n$ is the initial condition, and $\mathbf{f}: \mathbb{R} \times \mathbb{C}^n \to \mathbb{C}^n$ is the function describing the vector field. In terms of dynamical systems, $\mathbf{f}$ is a velocity vector field and $\mathbf{x}$ is a curve in phase space that is tangent to the vector field at every point. Equation (12.2) can thus be expanded as a system of coupled initial-value problems as follows:

$$\begin{aligned}
\dot{x}_1 &= f_1(t, x_1(t), x_2(t), \ldots, x_n(t)), & x_1(t_0) &= x_{1,0} \\
\dot{x}_2 &= f_2(t, x_1(t), x_2(t), \ldots, x_n(t)), & x_2(t_0) &= x_{2,0} \\
&\vdots & &\vdots \\
\dot{x}_n &= f_n(t, x_1(t), x_2(t), \ldots, x_n(t)), & x_n(t_0) &= x_{n,0}.
\end{aligned}$$

It is also sometimes convenient to write the system in matrix–vector notation,

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}(t)) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{b}(t)\,, \tag{12.3}$$

when dealing with linear or linearized systems, since the stability properties will then be partially explained in terms of the eigenvalues or the pseudospectra of $\mathbf{A}(t)$. In this case, the entries $a_{ij}(t)$ of $\mathbf{A}$ are usually assumed to be continuous in t and, for linear systems, do not depend on any of the x_i. For nonlinear systems, the notation is sometimes abused to let the a_{ij}s depend on the x_is. The vector $\mathbf{b}(t)$ corresponds to the nonhomogeneous part of the system; namely, it is a function of t only.

Example 12.2. In Shonkwiler and Herod (2009), we find an extended discussion of the following three-compartment model of the takeup of the toxic metal lead (Pb) into an organism's blood, soft tissue, and bones. Although for constant transmission rates between compartments and constant lead input for the environment the model can be solved explicitly by using a matrix exponential, numerical solution is still valuable. For this system, the matrix is 3 × 3:

$$\mathbf{A} = \begin{bmatrix} -(a_{01}+a_{21}+a_{31}) & a_{12} & a_{13} \\ a_{21} & -(a_{02}+a_{12}) & 0 \\ a_{31} & 0 & -a_{13} \end{bmatrix} \tag{12.4}$$

and the inhomogeneity $\mathbf{b} = [I_L, 0, 0]'$. The values of the parameters used there are $a_{0,1} = 0.0211$ (all parameters are in units of micrograms per day, denoted $^{\mu g}/_{d}$), $a_{12} = 0.0124$, $a_{13} = 0.000035$, $a_{21} = 0.0111$, $a_{02} = 0.0162$, $a_{31} = 0.0039$, and the environmental input is $I_L = 49.3$. The matrix is nonsymmetric (there is no reason that the rate of transfer of lead from soft tissue to bones should be the same as the other way round, for example) and all eigenvalues are negative. Problem 12.11 asks you to solve this system on $0 \le t \le 10^8$ using `ode15s`. ◁

Finally, we observe that the system can always be modified so that it becomes an autonomous system [where $\mathbf{f}$ does not depend on t, that is, where $\mathbf{f}(t, \mathbf{x}(t)) = \mathbf{f}(\mathbf{x}(t))$]. In order to do so, we simply add an $(n+1)$st component to $\mathbf{x}$ if necessary, so that the f_i are now of the form $f_i(x_1(t), \ldots, x_n(t), x_{n+1}(t))$, and add an $(n+1)$st equation

$$\dot{x}_{n+1} = f_{n+1}(\,x_1(t), x_2(t), \ldots, x_n(t), x_{n+1}(t)\,) = 1 \qquad x_{n+1}(t_0) = t_0\,. \tag{12.5}$$

It is often convenient to deal with the autonomous form of systems (see, e.g., Sect. 13.5.5), and we will freely do so. In this notation, we will often simply write $\mathbf{x}$ instead of $\mathbf{x}(t)$ and $\mathbf{A}$ instead of $\mathbf{A}(t)$.

Example 12.3. As a first nonlinear example, we show how to express *systems of first-order equations* in this manner. Consider the Lorenz system, in which we have a system of three first-order differential equations:

$$\begin{aligned} \dot{x} &= yz - \beta x & x(0) &= 27 \\ \dot{y} &= \sigma(z-y) & y(0) &= -8\,. \\ \dot{z} &= y(\rho - x) - z & z(0) &= 8 \end{aligned} \tag{12.6}$$

We use Saltzman's values of the parameters: $\sigma = 10, \rho = 28$, and $\beta = {}^{8}/_{3}$. To express this system in a way that MATLAB can process, we simply make the trivial relabeling of variables $x_1(t) = x(t)$, $x_2(t) = y(t)$, and $x_3(t) = z(t)$:

$$\begin{aligned} \dot{x}_1 &= x_2 x_3 - \beta x_1 \\ \dot{x}_2 &= \sigma(x_3 - x_2) \\ \dot{x}_3 &= x_2(\rho - x_1) - x_3\,. \end{aligned}$$

We can then, if we like, rewrite the system in matrix–vector notation:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} x_2x_3 - \beta x_1 \\ \sigma(x_3 - x_2) \\ x_2(\rho - x_1) - x_3 \end{bmatrix} = \begin{bmatrix} -\beta & 0 & x_2 \\ 0 & -\sigma & \sigma \\ -x_2 & \rho & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \mathbf{A}\mathbf{x}.$$

Moreover, our initial conditions $x(0)$, $y(0)$, and $z(0)$ now form a vector

$$\mathbf{x}(0) = \begin{bmatrix} x_1(0) \\ x_2(0) \\ x_3(0) \end{bmatrix} = \begin{bmatrix} 27 \\ -8 \\ 8 \end{bmatrix} = \mathbf{x}_0\,.$$

We remark that our chosen "standard" notation for the reference solution $\mathbf{x}(t)$ has a clash with the first scalar variable $x(t)$, which is $x_1(t)$ in the vector notation. The chosen standard notation $\mathbf{z}(t)$ for the numerical solution has a clash with the third scalar variable $z(t)$. Such is life. ◁

For the problem of Example 12.3, we also remark that the reference solution $\mathbf{x}(t)$ or $\mathbf{x}_{\text{ref}}(t)$ is unavailable for the Lorenz system for long times, because the solution is chaotic. We will examine this aspect in more detail later, but for now we note that the computed solution $\mathbf{z}(t)$ is quite acceptable for many purposes, as we will see. To tackle this problem, one can use the code below:

```
function lorenzsys
  % Three-dimensional plot of solution of Lorenz equations
  rho   = 28;
  sigma = 10;
  beta  = 8/3;
  tspan = linspace(0, 100, 1e4 );
  y0    = [ 27, -8, 8 ];

  [ tplot, yplot ] = ode45( @(t,y)lorenzeqs(t,y,rho,sigma,beta),
      tspan, y0 );
  % first few points of the time history
  plot( tplot(1:200), yplot(1:200,1),'k-', tplot(1:200), yplot
      (1:200,2), 'k--', tplot(1:200), yplot(1:200,3), 'k-.' )
  set(gca,'fontsize',16);
  figure
  % phase diagram
  plot3( yplot(:,1), yplot(:,2), yplot(:,3), '-k' )
  set(gca,'fontsize',16);


function ydot=lorenzeqs(t,y,rho,sigma,beta)
  ydot(1,:) = -beta*y(1,:) + y(2,:).*y(3,:);
  ydot(2,:) = sigma*( y(3,:)-y(2,:) );
  ydot(3,:) = -y(2,:).*y(1,:) + rho*y(2,:)-y(3,:);
end

end
```

Note the use of a “curried” in-line function call[3]

```
@(t,y)lorenzeqs(t,y,rho,sigma,beta)
```

in line 9 to create a function for `ode45` that does not have explicit reference to the parameters σ, ρ, and β. The first function simply provides the parameters and executes `ode45` (lines 3–9). The second function defines $\mathbf{f}(t,\mathbf{x})$ as described in Eq. (12.6) (lines 19–23).

We might have used the `sol = ode45(...)` construct instead of demonstrating the direct computation of values of the solution at the given 10^4 points in the tspan. If we had done that, `sol` would have been a structure containing the $3 \times 1{,}491$ array `sol.y`; the row `sol.y(i,:)` would have been the vector of computed values of $z_i(t_n)$, which then could be used to evaluate the solution at the 10,000 desired points. By doing it the way we did, we obtained instead a $10{,}000 \times 3$ array of values.[4]

However one does it, one can then easily obtain useful plots, such as time histories and phase portraits, using `deval` if necessary. See Fig. 12.2. We will often explicitly use `deval` to evaluate the numerical solution, which can then be fed into the functions `plot` and `plot3` to obtain graphical results. By asking for the output `[tplot,yplot]`, we are implicitly using `deval`, by the way.

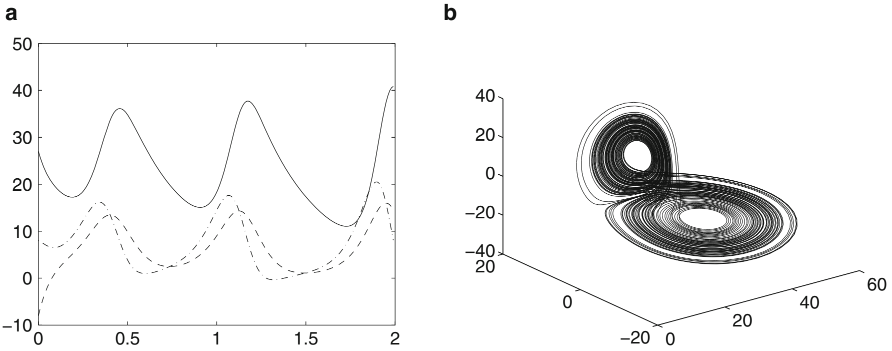


Fig. 12.2 Plots of the numerical solutions of the Lorenz system. (**a**) Time history of $x(t), y(t)$, and $z(t)$. (**b**) Phase portrait for all three components of the solution

Another trivial change of variables can be used to solve systems of *higher-order differential equations* in MATLAB. This time, the change of variables is used to transform a higher-order differential equation into a system of first-order differen-

[3] A curried function, named after Haskell Curry (see, e.g, Curry and Feys 1958) but first developed by Schönfinkel (1924), is a transformation of a multivariate function to treat it as a sequence of univariate functions.

[4] The transposition is a bit of a headache, to be honest—we are always having to debug errors caused by not remembering which way round the vectors are to go; however, it is what it is, for backward compatibility reasons. Again, we remember Admiral Grace Hopper: “The nice thing about standards is that there’s so many to choose from.”

tial equations[5]; we can then write it in vector notation as above if we like. In this case, a solution to the nth-order initial-value problem is a column vector $\mathbf{x}(t)$ whose components are

$$x_i(t) = \frac{d^{i-1}}{dt^{i-1}}x(t) = x^{(i-1)}(t).$$

Example 12.4. Suppose we are given the differential equation

$$\ddot{x} + 2\zeta\omega_0\dot{x} + \omega_0^2 x = 0 \tag{12.7}$$

for a damped harmonic oscillator. We first form the vector

$$\mathbf{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix},$$

so that

$$\begin{aligned} \dot{\mathbf{x}} &= \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} \dot{x}(t) \\ \ddot{x}(t) \end{bmatrix} = \begin{bmatrix} x_2 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -2\zeta\omega_0\dot{x} - \omega_0^2 x \end{bmatrix} \\ &= \begin{bmatrix} x_2 \\ -2\zeta\omega_0 x_2 - \omega_0^2 x_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\omega_0^2 & -2\zeta\omega_0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \mathbf{A}\mathbf{x} = \mathbf{f}(t,\mathbf{x}). \end{aligned}$$

We can then use the MATLAB routine `ode45` to find a numerical solution to this initial-value problem. Again, we create an `m`-file similar to the one below:

```
function dampedharmonicoscillator

tspan = [ 0, 10 ];
y0    = [ 0, 1 ]; %this is [x(0),x'(0)]
opts = odeset('Refine', 8);
[ tplot, yplot ] = ode45( @odefun, tspan, y0, opts );
figure(1), plot( tplot, yplot(:,1), '-k', ...
                 tplot, yplot(:,2), '--k' )
set(gca,'fontsize',16);
%phase portrait
figure(2), plot( yplot(:,1), yplot(:,2), '-k' )
set(gca,'fontsize',16);

function f = odefun(t,y)
omega = 2*pi;
zeta  = 0.1;   % parameters hard-coded
f = [ 0, 1; -omega^2, -2*zeta*omega ]*y;
end
end
```

Here, `sol.y(:,1)` contains the computed values of $x(t_n)$ and `sol.y(:,2)` contains the computed values of $\dot{x}(t_n)$. This is an example of using the `Refine`

[5] Instead of this trivial change of variable, it is sometimes better to use physically relevant variables (see Ascher et al. 1988). In this book, we usually just use the trivial new set of variables.

option in order to produce a reasonably smooth phase portrait. If we just plotted the `sol.x` and `sol.y` data, the points are too far apart in the phase portrait for linear interpolation to produce a good plot (try it—see Problem 12.12). By default, `ode45` produces a plot with a refinement factor of 4, and this is normally enough (it would be enough in this example, too) to make a smooth plot. Here we use `nRefine = 8`, just because. Note that the variable `yplot` has two columns and $8n+1$ rows, where the length of `tplot` is also $8n+1$. The length of `sol.x` would have been just $n+1$. The graphical results are displayed in Fig. 12.3. ◁

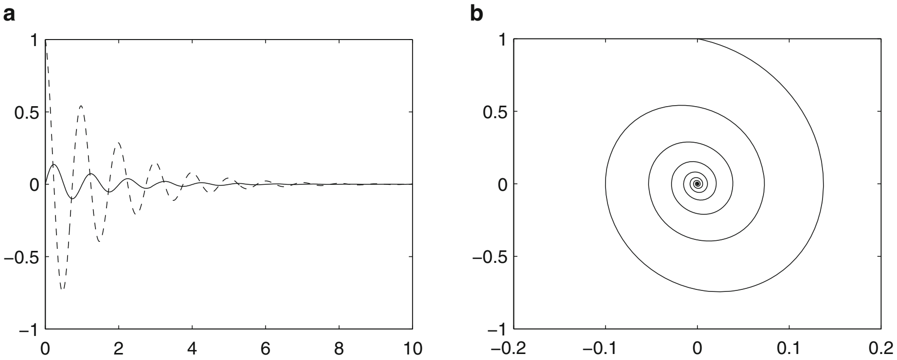


Fig. 12.3 Damped harmonic oscillator (12.7) with $\omega = 2\pi$ and $\zeta = 0.1$ and zero initial conditions. (**a**) Time history for $x(t)$ and $\dot{x}(t)$. (**b**) Phase portrait of $\mathbf{x}$

The question we usually start with when attempting to solve a differential equation is: What do the solutions look like, for various initial conditions and parameter values? In the numerical context, we might have various methods returning various solutions. If the solutions differ, or even if they don't, a question presents itself: *Are the numerical solutions faithful to the mathematical model?* One possible, and indeed natural, answer to that question involves comparing the numerical solutions to the reference solution, somehow. But in the usual cases of interest, the reference solution is unavailable. What, then, to do? In the next section, we look at an effective and efficient answer that uses the residual.

12.2 The Residual

We show how to assess the quality of a solution by examining its *residual*. The word "defect" is a standard name (in the field of numerical methods for the solution of differential equations) for what we have called the residual so far. Other works call this quantity the *deviation* (e.g.. Birkhoff and Rota 1989). We will clarify the use of those terms, and then show how to compute the residual.

12.2.1 Residual, Defect, and Deviation

For a given initial-value problem $\dot{\mathbf{x}}(t) = \mathbf{f}(t,\mathbf{x}(t))$, if we knew the reference solution $\mathbf{x}(t)$ and its derivative, we would obviously find that $\dot{\mathbf{x}}(t) - \mathbf{f}(t,\mathbf{x}(t)) = 0$. However, numerical methods do not return the reference solution $\mathbf{x}(t)$ and its derivative $\dot{\mathbf{x}}(t)$, but rather some other function $\mathbf{z}(t)$ and its derivative $\dot{\mathbf{z}}(t)$ (we have not yet seen an example where `deval` returns the derivative of the interpolant as well as the value, but we will). But then, $\dot{\mathbf{z}}(t) - \mathbf{f}(t,\mathbf{z}(t))$ will not in general be zero; rather, we will have

$$\Delta(t) = \dot{\mathbf{z}} - \mathbf{f}(t,\mathbf{z}(t)) \,, \tag{12.8}$$

where $\Delta(t)$ is what we call the *absolute* defect or residual, as opposed to *relative* defect or residual, which we define next. The *relative* defect $\delta(t)$ is defined componentwise [provided each component of $\mathbf{f}(t,\mathbf{z}(t)) \neq 0$] so that

$$\delta_i(t) = \frac{\dot{z}_i - f_i(t,z)}{f_i(t,z)} = \frac{\dot{z}_i}{f_i(t,z)} - 1 \,. \tag{12.9}$$

As before, we can express the original problem in terms of a modified, or perturbed, problem, so that our computed solution is an *exact* solution to this modified problem:

$$\dot{\mathbf{z}} = \mathbf{f}(t,\mathbf{z}) + \Delta(t) \,. \tag{12.10}$$

Similarly in the relative case $\dot{z}(t) = f(z(t))(1+\delta(t))$, with a suitable modification in meaning for a vector system. The residual vector Δ is then a nonhomogenous term added to the function $\mathbf{f}$, and Eq. (12.10) represents a reverse-engineered problem.

Definition 12.1. If the computed solution (typically a piecewise polynomial interpolant) is continuously differentiable, we will call the residual a *defect*. If the computed solution is only piecewise continuously differentiable, so that the residual is only defined piecewise and may have jump discontinuities, we will call it a *deviation*. See Problem 12.24. ◁

Let us look at the residual from the point of view of dynamical systems. As we have seen, in an ODE of the form $\dot{\mathbf{x}} = \mathbf{f}(t,\mathbf{x}(t))$, the function $\mathbf{f}$ determines a velocity vector and a solution $\mathbf{x}(t)$ is then a curve in the phase space that is tangent to the vector field at every point (see Fig. 12.4). By computing the residual $\Delta(t) = \dot{\mathbf{z}}(t) - \mathbf{f}(t,\mathbf{z}(t))$, we are in effect measuring how far from satisfying the differential equation our computed trajectory $\mathbf{z}(t)$ is, that is, how close it is to being tangent to the vector field.[6] Alternatively, we can then say that the computed trajectory $\mathbf{z}$ is tangent to a perturbed vector field $\mathbf{f}(t,\mathbf{x}) + \Delta(t)$.

Note that the residual is easily computed in MATLAB. The ODE solver returns a structure `sol` containing the evaluation points τ_k as well as the values of $\mathbf{z}(t_k)$, that

[6] This really only makes sense for systems that had originally been nonautonomous, and then embedded in an $(n+1)$-dimensional autonomous system; in that case, the residual $\Delta(t)$ can be interpreted as $\Delta(x_{n+1})$ and the perturbed vector field will be the same dimension as the original. Otherwise, the perturbed vector field is really one dimension higher.

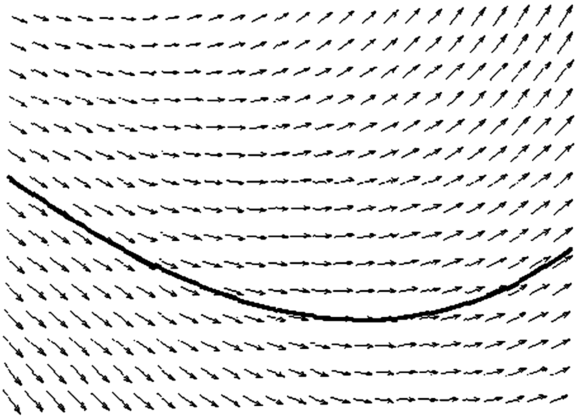


Fig. 12.4 A vector field with a nearly tangent computed solution

is, the skeleton. Using `deval`, which knows a suitable interpolant for each solver's output, we can then find a continuous and differentiable function $\mathbf{z}(t)$, as well as its derivative $\dot{\mathbf{z}}(t)$. But this is all one needs to compute the defect, since the function $\mathbf{f}$ is known from the beginning, being the definition of our initial-value problem.[7]

For a given selection of points t_k (defined, for instance, by `linspace`), the MATLAB command

```
[ z, dz ] = deval( sol, t )
```

returns the values $\mathbf{z}(t)$ and $\dot{\mathbf{z}}(t)$ at the point(s) t in the interval. Computation of the defect is straightforward after that. As an example, consider again the problem in Eq. (12.1). One can obtain the residual at a lot of points[8] as follows:

```
f   = @(t,x) t.^2 + x - x.^4/10;
sol = ode45( f, [0,5], 0 );
% Compute and plot the relative residual on a lot of points
t = RefineMesh( sol.x, 40 );
[ z, dotz ] = deval(sol,t);
deltat = dotz./f(t,z)-1;
figure(1),semilogy(t,abs(deltat),'k.'),set(gca,'fontsize',24)
xlabel('t'),ylabel('relative␣residual'),axis([0,5,1.0e-8,1.0])
```

In Fig. 12.5a, we show these values of the residual for this problem. Observe that the maximum residual over the interval is quite large, namely, about 0.5.

To reduce the size of the residual, MATLAB offers the user the possibility of specifying a tolerance. That is, it allows the user to specify both a (scalar) *relative* tolerance and an *absolute tolerance* (which might be a vector of absolute tolerances). The above example can be modified as follows in order to specify a tolerance:

[7] This isn't perfect. We will later encounter examples where the built-in interpolant and its derivative have some trouble. But for the problems these codes were designed for, difficulties are rare.

[8] At this moment, it's not clear how many we should take; we will explain the program `RefineMesh` later.

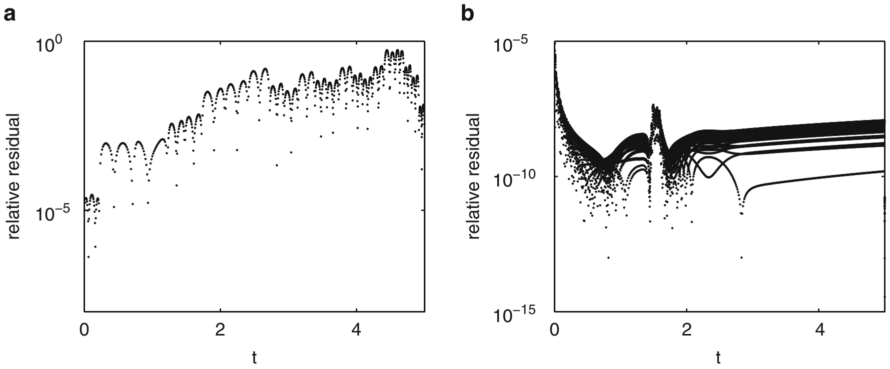


Fig. 12.5 Easily computed residual of the solution of (12.1) with `ode45`. Note the scale difference. Note also that the size of the residual is much larger than the tolerance in each case, although the solution with a tighter tolerance has a smaller residual. (**a**) With default tolerance. (**b**) With tolerance `1e-11`

```
opts = odeset( 'Reltol', 1.0e-6, 'Abstol', 1.0e-11 );
sol = ode45( f, [0,5], 0, opts );
```

The default absolute tolerance in MATLAB for `ode45` is 1.0×10^{-6}. If we run our example above instead at this tighter tolerance, we obtain the residual displayed in Fig. 12.5b. Observe that the relative residual is *not* actually smaller than the tolerance everywhere, but it's small anyway, and it gets smaller with tighter tolerances. The tolerance and the relative residual are related, but still different quantities. The important point is that one can *control* the size of the residual by *tightening* the tolerance.

Something important has happened here. Notice that the numerical method gives you the exact solution to a nearby problem and that you get to control just how nearby it is (we will examine this relationship in more detail in Sect. 13.2.2). One can look at the residual if one chooses, although it is necessary to do more computations. As we do throughout this book, from our a posteriori backward error analysis point of view, we then interpret the residual as a backward error, and henceforth say that our numerical solution gives us the exact solution of a nearby problem. We will look at more sophisticated backward errors later.

In our first example, we can then claim that the tight-tolerance (10^{-11}) computation provided an exact solution to

$$\dot{x} = \left(t^2 + x - \frac{x^4}{10}\right)\left(1 + 10^{-6}v(t)\right), \qquad 0 \le t \le 5,$$

where $v(t)$ is some smooth but otherwise unidentified function such that $|v(t)| \le 1$. The value $\varepsilon = 10^{-6}$ has been chosen based on the maximum computed value of the relative residual on the interval $[t_0, t_f]$.

In the example of the Lorenz system, we solve it again at tighter tolerances but otherwise with the same parameters and use the solution structure `sol`. The overall attractor looks no different from the loose tolerance solution presented earlier. We zoom in on the residual over the last few points as follows (see Fig. 12.6 for a scaled version; apart from dividing by dz, the code is identical):

```
tzoom = RefineMesh( sol.x(end-3:end), 80 );
[z,dz] = deval( sol, tzoom );
Delta = dz - lorenzeqs( tzoom, z, rho, sigma, beta );
figure(4), plot( tzoom, Delta, 'k' )
axis([tzoom(1),tspan(end),-7.5e-6,7.5e-6]),grid('on')
```

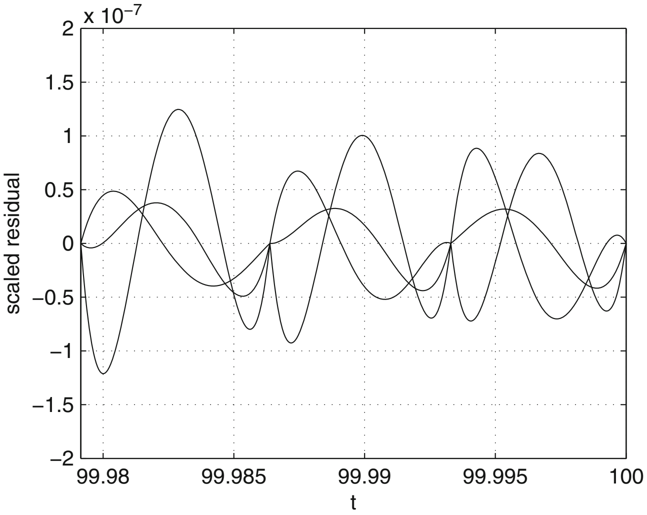


Fig. 12.6 Scaled residual components of the Lorenz system with relative tolerance 1×10^{-9} and absolute tolerance 1×10^{-12} using ode45. The maximum is only slightly larger than 1×10^{-7} times the magnitude of the derivative of the interpolant

As a result, we say that our numerical solution is the exact solution of the nearby problem

$$\dot{\mathbf{x}} = \text{lorenzeqs}(\mathbf{x})\left(1 + 1.2 \cdot 10^{-7}\mathbf{v}(t)\right),$$

where each component of $\mathbf{v}(t) = [v_1(t), v_2(t), v_3(t)]^T$ is less than 1; that is, $\|\mathbf{v}(t)\|_\infty \leq 1$.

12.2.2 A Closer Look at Computing the Residual

The numerical methods used to compute the values of the solution of the IVP at the internal mesh points t_k are studied in Chap. 13. For the moment, we note that as usual the mesh is returned in the solution structure field ending in '.x'. For example, if our solution is stored in the variable `sol`, then the discrete mesh is contained in the structure field `sol.x`. The rest of the skeleton, namely, the values of the solution at the mesh `sol.x`, is contained in the structure field `sol.y`.

Example 12.5. Consider the simple ODE $\dot{x}(t) = \frac{x(t)}{(1+tx(t))}$, with initial condition $x(0) = 0.7$. We can solve this on $0 \le t \le 100$ in MATLAB by executing

```
f = @(t,x) x./(1+t.*x);
sol = ode113( f, [0,100], 0.7 );
```

The size of the discrete mesh returned by the MATLAB routine `ode113` is found by executing `length(sol.x)`, which returns 38. This isn't a terribly fine mesh; more importantly, it is not a *uniform* mesh. One way to fill in the graph is to ask to interpolate the solution at (say) 1000 intermediate points.

```
t = linspace( 0, 100, 1000 );
[xt dxt ] = deval( sol, t );
plot( sol.x, sol.y, 'ko', t, xt, 'k-')
```

(That figure is not shown here—it's pretty boring.) However, using this number of points might be quite wasteful, and—more subtly—may in some cases miss important regions, in spite of taking so many points (roughly 25 per subinterval, but the difficulty is that not every subinterval actually receives 25 points). The MATLAB codes, as all good general-purpose codes do, try to adjust the mesh so that the spacings $h_k = t_k - t_{k-1}$ are small when important things are happening; if we just then grossly sample (even oversample) with a uniform mesh, it may well be possible that the uniform mesh will completely miss very narrow but important regions of interest.

This is especially problematic if we wish to measure the residual, and therefore we do not want to use a gross uniform mesh with a large number of points (possibly both too large a number and still yet missing out on important behavior). Instead, what we want to do is to *refine* the mesh that the solver gives us. One short MATLAB program to do this is given by

```
function [ refinedMesh ] = RefineMesh( coarseMesh, nRefine )
%REFINEMESH Insert more points into each subinterval of a mesh
%   refinedMesh = RefineMesh( coarseMesh, nRefine )
%                                       default nRefine = 4
    if nargin == 1,
        nRefine = 4;
    end
    n = length( coarseMesh );
    [m1,m2] = size( coarseMesh );
    h = diff( coarseMesh );
    refinedMesh = repmat( coarseMesh(1:end-1).', 1, nRefine );
    refinedMesh = (refinedMesh+(h.')*[0:nRefine-1]/nRefine).';
    refinedMesh = [refinedMesh(:);coarseMesh(end)]; % column
        vector
    if m1<m2,
        refinedMesh = refinedMesh.'; % row vector input ==> also
            output
    end
end
```

That short program puts `nRefine` (-1) new points into each subinterval in the supplied mesh; it preserves the overall distribution and does not miss any narrow regions that the solver decided were important. As we saw in previous examples, there are provisions for asking the solvers themselves to produce a refined mesh; however, one can't (apparently) do that and return a solution structure at the same time—hence this little program.

As a short example, if $\boldsymbol{\tau} = [-1, -1/2, 1/2, 1]$ (which is not equally spaced) and we call `RefineMesh` asking for 3 subintervals in each interval, we get

$$\mathbf{t} = [-1.0000, -0.8333, -0.6667, -0.5000, -0.1667, \\ 0.1667, 0.5000, 0.6667, 0.8333, 1.0000]\,,$$

as expected. The subintervals have widths $1/6$ at the edges and $1/3$ in the middle, again as expected.

The code `RefineMesh` can be used as follows:

```
f = @(t,x) x./(1+t.*x);
opts = odeset( 'RelTol', 1.0e-3, 'AbsTol', 1.0e-6 );
sol = ode113( f, [0,100], 0.7, opts );
length( sol.x )
h = diff( sol.x );
t = RefineMesh( sol.x, 20 );
np = length( t )
[zt dzt ] = deval( sol, t );
figure(1), plot( sol.x, sol.y, 'ko', t, zt, 'k-' ),set(gca,'
    fontsize',16)
xlabel('t'),ylabel('x')
figure(2), semilogy( sol.x(2:end), h, 'k.' ),set(gca,'fontsize'
    ,16)
xlabel('t'),ylabel('step␣size␣h')
residual = zeros(size(t));
for i=1:np,
    residual(i) = dzt(i) - f(t(i),zt(i));
end
mx = max( abs( residual ) );
figure(3), plot( sol.x, zeros(size(sol.x)), 'ko', t, residual, 'k
    .' ),
xlabel('t','fontsize',16),ylabel('residual','fontsize',16),set(
    gca,'fontsize',16),set(gca,'YTick',-5E-4:2.5E-4:5E-4)
axis([0,10,-mx, mx] )
```

An examination of the `figure(2)` produced in that script (but not reproduced here) shows that the mesh widths h_k vary quite widely, getting to be $O(10)$ as the solution settles down. In the wider subintervals, a very fine mesh as used initially would be wasteful. On the other hand, using 20 points per subinterval gives rise to a refined mesh with 740 points—this is plenty, and more than plenty, to see what is happening in this example. With the default relative error tolerance (explicitly coded in at 10^{-3}, as a setup for Problem 12.24; similarly, the defaults for `ode113` for the absolute tolerance are also there), the code has produced the exact solution to

$$\frac{dx}{dt} = \frac{x}{1+tx} + 5 \cdot 10^{-4} v(t) \tag{12.11}$$

with $|v(t)| \leq 1$ on $0 \leq t \leq 100$.

In Fig. 12.7, we see the results of `figure(3)` from that script. This zooms in on the interesting part of the computed residual for this example. ◁

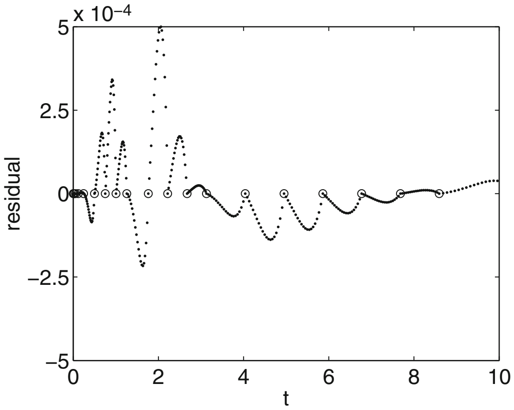


Fig. 12.7 The (absolute) residual of the equation in Example 12.5 computed on a mesh refined 20 times (and thus with less work overall than the 1000-point mesh used crudely the first time), and displayed only on $0 \leq t \leq 10$. On the remainder of the interval, the residual is much smaller

The MATLAB codes `ode45`, `ode15s`, `ode113`, and the rest each come with their own interpolant, which the evaluation routine `deval` knows about. To be specific, the name of the solver is stored in the solution structure returned, and so when the solution structure is passed to `deval`, it knows which interpolant to use to evaluate the solution and its derivative. This is a very convenient system.

Remark 12.1. The interpolant used by `ode45` is sometimes not quite accurate enough and usually overestimates the actual residual. For the purposes of this book, this is harmless. ◁

12.3 Conditioning of IVP

In the previous section, we have defined the residual and we have seen how to compute it without pain.[9] We then use this computed value to estimate a backward error. Now, from the general perspective developed in Chap. 1, an initial-value problem can be seen as a functional map

[9] Of course, the computer has to do some work! Each point at which we evaluate the residual requires a new evaluation of $f(t,x(t))$, and thus reassurance comes at a price.

$$\varphi : \Big(\mathbf{f}(t,\bullet), t_0, \mathbf{x}_0\Big) \to \Big\{\mathbf{x}(t) : \dot{\mathbf{x}} = \mathbf{f}(t,\mathbf{x}(t)) \quad \& \quad \mathbf{x}(t_0) = \mathbf{x}_0\Big\}, \tag{12.12}$$

where $\mathbf{f}$ is a functional $\mathbb{R}\times\mathbb{C}^n \to \mathbb{C}^n$ (the tangent vector field) and $\mathbf{x}_0$ is the initial condition. We can then study the effects of three cases of backward errors: where we perturb $\mathbf{f}$, where we perturb $\mathbf{x}_0$, and where we perturb both. In the previous section, we have given two examples of perturbation of $\mathbf{f}$ with $\varepsilon\mathbf{v}$, where the magnitude of ε is the maximum computed residual and $\mathbf{v}$ is a noisy function with $\|\mathbf{v}\|_\infty \le 1$ ($\mathbf{v}$ will sometimes be assumed to be a function of t only, as in Sect. 12.3.2, and will sometimes be allowed to be a nonlinear function of t and $\mathbf{x}$, as in Sect. 12.3.3). This situation is represented in Fig. 12.8.

$$\begin{tikzcd}
\text{input space} & & \text{output space} \\
(\mathbf{f}(t,\mathbf{x}),\mathbf{x}_0) \arrow[rr, "\varphi"] \arrow[d, dash, "\text{backward error (b.e.) } \varepsilon\mathbf{v}"'] \arrow[drr, dashed, "\widehat{\varphi}\text{, } \textit{i.e.}\text{, num. method}"] & & y=\{\mathbf{x}(t):\dot{\mathbf{x}}=\mathbf{f}(t,\,\mathbf{x}),\mathbf{x}(0)=\mathbf{x}_0\} \arrow[d, dash, "\text{forward error} \approx cond\times b.e."] \\
(\mathbf{f}(t,\mathbf{x})+\varepsilon\mathbf{v},\mathbf{x}_0) \arrow[rr, "\varphi"'] & & \mathbf{z}(t)
\end{tikzcd}$$

Fig. 12.8 Commutative diagram for the backward error analysis of initial value problems. Note that we can also perturb $\mathbf{x}_0$, or both $\mathbf{x}_0$ and $\mathbf{f}$. In some cases, this diagram will be implicitly replaced by an "almost commutative diagram," as defined in Chap. 1

The question we ask in this section is: What effects do perturbations of $\mathbf{f}$, $\mathbf{x}_0$, or both have? That is what the *conditioning* of the initial-value problem tells us. We begin by examining the effects of a perturbation of the initial condition only. Next, we examine the effects of a perturbation of the functional $\mathbf{f}$.

12.3.1 Lipschitz Constants

We first need to look at the classical theory of Lipschitz constants. Consider the initial-value problem

$$\dot{\mathbf{x}} = \mathbf{f}(t,\mathbf{x}(t)) \qquad \mathbf{x}(0) = \mathbf{x}_0\,. \tag{12.13}$$

The function $\mathbf{f}(t,\mathbf{x})$ is said to be Lipschitz continuous in $\mathbf{x}$ with respect to a norm $\|\cdot\|$ over the interval $t\in[a,b]$ if there is a constant L such that for any $t\in[a,b]$ and for any two $\mathbf{x}_1(t)$ and $\mathbf{x}_2(t)$,

$$\|\mathbf{f}(t,\mathbf{x}_1)-\mathbf{f}(t,\mathbf{x}_2)\| \le L\|\mathbf{x}_1-\mathbf{x}_2\|. \tag{12.14}$$

L is called a *Lipschitz constant*. Moreover, if the function $\mathbf{f}$ in the initial-value problem above is continuous in t and Lipschitz continuous in $\mathbf{x}$, then there exists a unique solution $\mathbf{x}(t)$ to the IVP, at least in a small neighborhood of the initial condition.

Observe that when $\mathbf{x}_1 = \mathbf{x}_2$, (12.14) is trivially satisfied. Also, if $\mathbf{x}_1 \neq \mathbf{x}_2$, we have

$$\frac{\|\mathbf{f}(t,\mathbf{x}_1) - \mathbf{f}(t,\mathbf{x}_2)\|}{\|\mathbf{x}_1 - \mathbf{x}_2\|} \leq L\,.$$

Thus, L provides an upper bound on the effect that changes in $\mathbf{x}$ can have on $\mathbf{f}(t,\mathbf{x})$.

However, we are not so much interested in the effects of a perturbation of $\mathbf{x}(t_0)$ on $\mathbf{f}$ as in its effect on $\mathbf{x}(t)$, the solution of the initial-value problem. It can be shown that the forward error $\mathbf{z}(t) - \mathbf{x}(t)$ satisfies the following inequality:

$$\|\mathbf{z}(t) - \mathbf{x}(t)\| \leq \|\Delta(t)\| \frac{e^{L(t-t_0)} - 1}{L}\,. \tag{12.15}$$

For a proof, see Hairer et al. (1993 Chap. I.10). As nice as this bound is, however, it allows for exponential growth of the forward error even if the residual remains bounded. Sometimes this does indeed happen, but in a great many cases of practical interest the actual forward error does *not* grow exponentially, even though the bound does. Hence, we need a sharper bound, in some important cases.

For a large class of problems, one can replace L with a so-called logarithmic norm, which can be negative. We will not do that here. Instead, we give the tools that allow computation of a condition number specific to the problem at hand. This requires more work on any given problem, but the work gives more information about the problem at hand.

12.3.2 Condition via the Variational Equation

Consider the autonomous initial-value problem

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}(t)), \qquad \mathbf{x}(0) = \mathbf{x}_0 \tag{12.16}$$

and the corresponding nonautonomously perturbed autonomous initial-value problem

$$\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}(t)) + \varepsilon\mathbf{v}(t), \qquad \mathbf{z}(0) = \mathbf{z}_0\,. \tag{12.17}$$

We denote their solutions $\mathbf{x}(t)$ and $\mathbf{z}(t)$, respectively,[10] and we suppose for simplicity here that $\mathbf{x}_0 = \mathbf{z}_0$. The perturbation $\mathbf{v}(t)$ is assumed to be continuous, which is enough to ensure that $\mathbf{z}(t)$ is continuously differentiable.

Notice that since $\mathbf{z}(t)$ is the solution of the perturbed problem, we can also write $\mathbf{z}(t) = \mathbf{x}(t;\varepsilon)$ as a regular perturbation series, using the notation of Sect. 2.8. In

[10] The notation here echoes the use of $\mathbf{x}$ for the reference solution and $\mathbf{z}$ for the computed solution. In this section, though, it is important that both are reference solutions for their respective equations—as indeed the numerical solution $\mathbf{z}(t)$ genuinely is.

this context, ε is a small number and $\varepsilon\mathbf{v}$ a small perturbation, and we will accordingly investigate $\|\mathbf{x}(t) - \mathbf{z}(t)\|$ as $\varepsilon \to 0$. One question is then: As $\varepsilon \to 0$, does $\mathbf{x}(t;\varepsilon)$ converge to $\mathbf{x}(t)$ as $t \to \infty$? We will examine the Gröbner–Alexeev approach in the next section, which shows that it does. However, we take an easier approach here: We simply *assume* convergence, and linearize the problem about the reference solution. In effect, we will study the relation between $\mathbf{x}$ and $\mathbf{z}$ in the *tangent space*. The information we obtain will be valid only insofar as the tangent space trajectories represent well the original trajectories (i.e., in a small neighborhood).

Consider the asymptotic expansion of $\mathbf{z}(t)$—which, remember, is just $\mathbf{x}(t;\varepsilon)$—in powers of the perturbation ε:

$$\mathbf{z}(t) = \mathbf{x}_0(t) + \varepsilon\mathbf{x}_1(t) + O(\varepsilon^2)\,. \tag{12.18}$$

By formula (2.117), since the limit of $\mathbf{x}(t;\varepsilon)$ as $\varepsilon \to 0$ is just $\mathbf{x}(t)$, we have $\mathbf{x}_0(t) = \mathbf{x}(t;0) = \mathbf{x}(t) = \mathbf{x}_{\text{ref}}(t)$, giving us

$$\mathbf{z}(t) = \mathbf{x}(t) + \mathbf{x}_1(t)\varepsilon + O(\varepsilon^2)\,. \tag{12.19}$$

We want to solve for $\mathbf{x}_1(t)$ to determine the first-order effect of the perturbation on the solution, since $\mathbf{z}(t) - \mathbf{x}(t) \doteq \varepsilon\mathbf{x}_1(t)$.

The derivative of Eq. (12.19) is

$$\begin{aligned}\dot{\mathbf{z}}(t) &= \dot{\mathbf{x}}(t) + \dot{\mathbf{x}}_1(t)\varepsilon + O(\varepsilon^2)\\ &= \mathbf{f}(\mathbf{x}(t)) + \dot{\mathbf{x}}_1(t)\varepsilon + O(\varepsilon^2)\,.\end{aligned}$$

Since it follows from Eq. (12.19) that $\mathbf{x}(t) = \mathbf{z}(t) - \mathbf{x}_1(t)\varepsilon - O(\varepsilon^2)$, we can substitute and expand $\mathbf{f}$ about the computed solution $\mathbf{z}(t)$:

$$\begin{aligned}\dot{\mathbf{z}}(t) &= \mathbf{f}\Big(\mathbf{z}(t) - \mathbf{x}_1(t)\varepsilon - O(\varepsilon^2)\Big) + \dot{\mathbf{x}}_1(t)\varepsilon + O(\varepsilon^2)\\ &= \mathbf{f}(\mathbf{z}(t)) + \mathbf{f}'(\mathbf{z}(t))\Big((\mathbf{z}(t) - \varepsilon\mathbf{x}_1(t) - O(\varepsilon^2)) - \mathbf{z}(t)\Big) + \varepsilon\dot{\mathbf{x}}_1(t) + O(\varepsilon^2)\\ &= \mathbf{f}(\mathbf{z}(t)) + \mathbf{f}'(\mathbf{z}(t))(-\varepsilon\mathbf{x}_1(t)) + \varepsilon\dot{\mathbf{x}}_1(t) + O(\varepsilon^2).\end{aligned}$$

Now, by Eq. (12.17) and writing $\mathbf{f}'$ as $\mathbf{J_f}$, the Jacobian matrix, we obtain

$$\dot{\mathbf{z}}(t) - \mathbf{f}(\mathbf{z}(t)) = \varepsilon\mathbf{v}(t) = \varepsilon\dot{\mathbf{x}}_1(t) - \varepsilon\mathbf{J_f}(\mathbf{z}(t))\mathbf{x}_1(t)\,,$$

where the partial derivatives in the Jacobian $\mathbf{J_f}$ are evaluated at the computed solution $\mathbf{z}$.

Therefore, neglecting the higher powers of ε and rearranging the terms, we finally obtain

$$\dot{\mathbf{x}}_1(t) = \mathbf{J_f}(\mathbf{z}(t))\mathbf{x}_1(t) + \mathbf{v}(t), \qquad \mathbf{x}_1(t_0) = \mathbf{0}\,. \tag{12.20}$$

This is the *first variational equation*. The exact, analytic solution of this equation involves the machinery for linear nonhomogeneous equations with (possibly) variable coefficients. In what follows, we only briefly sketch how to solve the equation for $\mathbf{x}_1$ in the matrix–vector notation. This is an aside that does not have much to do with numerics, but it will help to fix the notation and to explain the mathematical objects we're dealing with in the codes. We will see at the end of this section how to implement all of this numerically.

Consider the homogeneous part of the variational equation (12.20), $\dot{\mathbf{x}}_1 = \mathbf{J}_\mathbf{f}(\mathbf{z})\mathbf{x}_1$ (we'll drop the subscript "1" below since the method applies generally to linear systems). If $\mathbf{x}_1(t), \mathbf{x}_2(t), \ldots, \mathbf{x}_n(t)$ are solutions (not to be confused with the terms $\mathbf{x}_k$ in the perturbation series above—especially $\mathbf{x}_1$, for which we dropped the subscript) of $\dot{\mathbf{x}} = \mathbf{J}_\mathbf{f}(\mathbf{z})\mathbf{x}$ and the Wronskian is nonzero, that is, if

$$W(\mathbf{x}_1, \ldots, \mathbf{x}_n) = \det\left[\mathbf{x}_1 \; \cdots \; \mathbf{x}_n\right] \neq 0\,,$$

then the solutions are linearly independent, so that the general solutions of $\dot{\mathbf{x}} = \mathbf{J}_\mathbf{f}(\mathbf{z})\mathbf{x}$ are

$$\mathbf{x}_h(t) = \mathbf{x}_1(t)c_1 + \mathbf{x}_2(t)c_2 + \ldots + \mathbf{x}_n(t)c_n = \mathbf{X}(t)\mathbf{c}\,,$$

and $\mathbf{X}(t)$ is then called a *fundamental solution matrix*. As one can easily verify from the above definition, the fundamental solution matrix satisfies the matrix differential equation $\dot{\mathbf{X}}(t) = \mathbf{J}_\mathbf{f}(\mathbf{z})\mathbf{X}(t)$. Moreover, we can always choose a fundamental matrix whose initial conditions will be $\boldsymbol{\xi}(0) = \mathbf{I}$ by applying a transformation $\mathbf{X}(t) = \boldsymbol{\xi}(t)\mathbf{C}$ ($\mathbf{C}$ constant), so that $\mathbf{I} = \mathbf{X}^{-1}(0)\mathbf{C}$. Then,

$$\dot{\boldsymbol{\xi}}(t) = \mathbf{J}_\mathbf{f}(\mathbf{z})\boldsymbol{\xi}(t), \qquad \boldsymbol{\xi}(0) = \mathbf{I} \tag{12.21}$$

is called the *associated matrix variational equation*. Note that there is no perfectly general symbolic method to identify the fundamental solutions filling up the matrix $\mathbf{X}(t)$, when the components of $\mathbf{J}_\mathbf{f}(\mathbf{z})$ are not constant, in terms of elementary functions, but methods are known for some classes of problems.[11]

Whenever $\mathbf{X}(t)$ is a fundamental matrix, it is nonsingular, and as a result the coefficients c_i are uniquely identifiable for a given initial-value problem as $\mathbf{c} = \mathbf{X}^{-1}(t_0)\mathbf{x}(t_0)$. Therefore, we find that the solution of the homogeneous system is

$$\mathbf{x}_h(t) = \mathbf{X}(t)\mathbf{X}^{-1}(t_0)\mathbf{x}(t_0)\,.$$

Once we know a set of fundamental solutions for the homogeneous part of the system, we can find a particular solution $\mathbf{x}_p(t)$ to the nonhomogeneous system by variation of parameters, obtaining a general solution with the superposition principle $\mathbf{x}(t) = \mathbf{x}_h(t) + \mathbf{x}_p(t)$. The solution of the inhomogeneous system, $\dot{\mathbf{x}} = \mathbf{J}_\mathbf{f}(\mathbf{z})\mathbf{x} + \mathbf{v}$, is obtained by letting the coefficients c_i be functions of t, so that $\mathbf{x}_p(t) = \mathbf{X}(t)\mathbf{c}(t)$.

[11] That is, there *are* algorithms that will find elementary expressions for the solutions when they are findable, and prove that they are not when they are not. See, for example, Bronstein and Lafaille (2002). These algorithms do not cover all classes of entries in $\mathbf{A}(t)$ and can be expensive even when they do work. Finally, if, as is usually the case, they simply prove that no elementary expressions are available, then we're back to numerical methods anyway.

Note that the derivative of $\mathbf{x}_p(t)$ is

$$\dot{\mathbf{x}}_p(t) = \dot{\mathbf{X}}(t)\mathbf{c}(t) + \mathbf{X}(t)\dot{\mathbf{c}}(t)\,,$$

so that, by substituting in $\dot{\mathbf{x}} = \mathbf{J_f}(\mathbf{z})\mathbf{x} + \mathbf{v}$, we obtain

$$\dot{\mathbf{X}}(t)\mathbf{c}(t) + \mathbf{X}(t)\dot{\mathbf{c}}(t) = \mathbf{J_f}(\mathbf{z})\mathbf{X}(t)\mathbf{c}(t) + \mathbf{v}(t).$$

Since, as we observed, the fundamental solution matrix satisfies $\dot{\mathbf{X}}(t) = \mathbf{J_f}(\mathbf{z})\mathbf{X}(t)$, we find that $\dot{\mathbf{c}}(t) = \mathbf{X}^{-1}(t)\mathbf{v}(t)$ (since $\mathbf{X}$ is invertible). By integration of $\dot{\mathbf{c}} = \mathbf{X}^{-1}\mathbf{v}$, we finally identify the variable coefficients and the particular solution:

$$\mathbf{x}_p(t) = \mathbf{X}(t)\int_{t_0}^{t} \mathbf{X}^{-1}(\tau)\mathbf{v}(\tau)d\tau\,.$$

Therefore, the general solution of the variational equation is (we reintroduce the subscript “1”):

$$\mathbf{x}_1(t) = \mathbf{X}_1(t)\mathbf{X}_1(t_0)\mathbf{x}_1(t_0) + \int_{t_0}^{t} \mathbf{X}_1(t)\mathbf{X}_1^{-1}(\tau)\mathbf{v}(\tau)d\tau\,. \qquad (12.22)$$

But since $\mathbf{x}_1(t_0) = 0$, the homogeneous term is just zero. This gives us the following expression for $\mathbf{z}(t) - \mathbf{x}(t)$:

$$\mathbf{z}(t) - \mathbf{x}(t) = \varepsilon\mathbf{x}_1(t) = \varepsilon\int_{t_0}^{t} \mathbf{X}_1(t)\mathbf{X}_1^{-1}(\tau)\mathbf{v}(\tau)d\tau\,,$$

where again we have ignored higher powers of ε. Now, it follows that

$$\|\mathbf{z}(t) - \mathbf{x}(t)\| = \varepsilon\left\|\mathbf{X}_1(t)\int_{t_0}^{t} \mathbf{X}_1^{-1}(\tau)\mathbf{v}(\tau)d\tau\right\| \le \varepsilon\|\mathbf{X}_1(t)\|\int_{t_0}^{t} \|\mathbf{X}_1^{-1}(\tau)\|\|\mathbf{v}(\tau)\|d\tau$$

by the triangle inequality.

Therefore, we obtain the inequality

$$\|\mathbf{z}(t) - \mathbf{x}(t)\| \le \|\mathbf{X}_1(t)\| \max_{t_0\le\tau\le t} \|\mathbf{X}_1^{-1}(\tau)\| \int_{t_0}^{t} \varepsilon\|\mathbf{v}(\tau)\|d\tau\,, \qquad (12.23)$$

where

$$\kappa(\mathbf{X}_1) = \|\mathbf{X}_1(t)\| \max_{t_0\le\tau\le t} \|\mathbf{X}_1^{-1}(\tau)\| \qquad (12.24)$$

acts as a condition number and the integral of $\varepsilon\|\mathbf{v}\|$ is a norm of $\varepsilon\mathbf{v}$. Thus, if the fundamental solution matrix $\mathbf{X}_1$ of the variational equation is well-conditioned, then we can expect an accurate numerical solution since the term $\mathbf{v}(t)$ will be damped, or at least won’t grow too much.

As we have seen in Chap. 4, the norm of a matrix $\mathbf{A}$ equals the largest singular value σ_1 of $\mathbf{A}$, and the norm of $\mathbf{A}^{-1}$ is the inverse of the smallest singular value of $\mathbf{A}$, denoted σ_n^{-1}. Accordingly, we can write the condition number as

$$\kappa(\mathbf{X}) = \frac{\sigma_1(t)}{\min\limits_{t_0 \le \tau \le t} \sigma_n(\tau)} .$$

It is not an accident that the singular value decomposition plays once again a role in this context. A standard tool for the evaluation of the effect of $\mathbf{X}$ on $\|\mathbf{z}(t) - \mathbf{x}(t)\|$ is the computation of the Lyapunov exponents λ_i of $\mathbf{X}(t)$. These λ_i are defined as the *logarithms of the eigenvalues of* $\boldsymbol{\Lambda}$, where

$$\boldsymbol{\Lambda} = \lim_{t \to \infty} \left(\mathbf{X}^T \mathbf{X}\right)^{1/2t} . \tag{12.25}$$

In other words, the Lyapunov exponents are closely related to the eigenvalues of $\mathbf{X}^T\mathbf{X}$. But as we have seen in Sect. 4.6.3, the eigenvalues of $\mathbf{X}^T\mathbf{X}$ are just the squares of the singular values of $\mathbf{X}$ (since $\mathbf{X}^T\mathbf{X} = (\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H)^H\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H = \mathbf{V}\boldsymbol{\Sigma}^2\mathbf{V}^H$). Note that we use a variation on the SVD, namely, the "analytic SVD," where the singular values can be negative and need not be ordered from largest to smallest. If we take a small displacement in $\mathbf{x}$ (so far, that's what we labeled $\varepsilon\mathbf{x}_1$), the singular value decomposition gives us a nice geometrical interpretation of Eq. (12.25). The SVD factoring $\mathbf{X} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H$ guarantees that $\mathbf{X}\mathbf{V} = \boldsymbol{\Sigma}\mathbf{U}$ for some unitary (in the real case, orthogonal) set of vectors $\{\mathbf{v}_i\}$ and $\{\mathbf{u}_i\}$. We can apply it to the displacement vector $\varepsilon\mathbf{x}_1$ to get $\mathbf{X}\mathbf{V}\varepsilon\mathbf{x}_1 = \boldsymbol{\Sigma}\mathbf{U}\varepsilon\mathbf{x}_1$; since the unitary transformations $\mathbf{V}\varepsilon\mathbf{x}_1$ and $\mathbf{U}\varepsilon\mathbf{x}_1$ of $\varepsilon\mathbf{x}_1$ preserve 2-norm, we see that the singular values show how much the matrix $\mathbf{X}(t)$ stretches the displacement vector $\varepsilon\mathbf{x}_1$ (here, since the original equation is nonlinear, the singular values will be a function of t). Now, if we take the logarithmic average of the singular values when $t \to \infty$ (because we are interested in average exponential growth), we get

$$\lim_{t \to \infty} \frac{1}{t} \ln \sigma_i(t) = \lim_{t \to \infty} \ln \left(\sigma_i^2(t)\right)^{1/2t} = \lim_{t \to \infty} \ln \operatorname{eig}_i(\mathbf{X}^T\mathbf{X})^{1/2t} = \ln \operatorname{eig}_i(\boldsymbol{\Lambda}) = \lambda_i .$$

Thus, if any Lyapunov exponent of $\mathbf{X}$ is positive, the average exponential growth of the displacement $\varepsilon\mathbf{x}_1$ will be *positive*, and so our initial-value problem will be exponentially ill-conditioned. However, remember that this is the behavior in the tangent space, since we linearized the initial value-problem. Thus, this does *not* imply that the displacement $\mathbf{z}(t) - \mathbf{x}(t)$ is unbounded, since the nonlinear terms might have an effect as we move away from the point about which we linearized. If one exponent is positive but the solution remains bounded, then the solution must behave unpredictably, though, and this is often termed *chaos*. We will return to this issue in Sect. 12.6.

Now, all of these tools somehow require us to compute $\mathbf{X}(t)$ in some way, or at least a computation of $\mathbf{X}(t_k)$ on a sufficient number of nodes t_k. An alternative to solving for $\mathbf{X}(t)$ analytically consists of solving the variational equation numerically.

However, the reader might want to ask: How do we know the numerical solution of the variational equation is reliable? It may seem that in order to determine whether or not it is, we would need to find *its* variational equation, and solve it numerically. Does it not lead to a regress *ad infinitum*? No. The first observation we need to make is that whereas the reference initial-value problem is in general nonlinear, the variational equation is linear. As a result, the variational equation is its own variational equation, and so there is no regress.

Moreover, our numerical solution of the variational equation can show that the reference problem is either ill-conditioned or not. On the one hand, if the computed solution shows that the components of $\mathbf{X}_1$ are large, then the variational equation is claimed to be ill-conditioned. This must be the case: If it were well-conditioned, then because we are using a stable method, we would find a small forward error and thus a small $\mathbf{X}_1$. We may well not identify $\mathbf{X}_1$ very accurately, but we will see that it is large. On the other hand, if the computed solution shows that the entries of $\mathbf{X}_1$ are not large, then the variational equation is claimed to be well-conditioned. However, obtaining a computed solution with a small $\mathbf{X}_1$ is actually (potentially) consistent with an ill-conditioned solution if all the errors just happened to cancel. How likely is it that an ill-conditioned $\mathbf{X}_1$ will have all its errors arranged in such a way that the computed answer is small and thus the equation appears well-conditioned? Nothing forbids this from happening, but it doesn't seem likely; in our experience, it has never happened.

> Anyone unlucky enough to encounter this sort of calamity has probably already been run over by a truck.[12]

Example 12.6. Consider Airy's differential equation $y'' = ty$ on the interval $0 \leq t \leq 10$, with the initial conditions $y(0) = \mathrm{Ai}(0)$ and $y'(0) = \mathrm{Ai}'(0)$. Clearly, the reference solution is $y(t) = \mathrm{Ai}(t)$. Is this a well-conditioned IVP or an ill-conditioned one?

Consideration of the known properties of the Airy functions $\mathrm{Ai}(t)$ and $\mathrm{Bi}(t)$ shows that this problem is quite ill-conditioned. We can go ahead and use the analysis of the previous section to show that the condition number grows faster than the exponential function, but because the answer is known, there is a simpler way: The general solution of the ODE is $y(t) = \alpha\mathrm{Ai}(t) + \beta\mathrm{Bi}(t)$, and even just a rounding error in the initial conditions (surely a milder disturbance than a residual, even a relative residual, of about 10^{-6}) will mean that our computed solution is something like $\mathrm{Ai}(t) + \varepsilon\mathrm{Bi}(t)$. By consulting reference books (or by use of MAPLE), we find that the asymptotics of the Airy functions are $\mathrm{Ai}(t) \sim c_1 t^{-1/4}\exp((-2/3)t^{3/2})$ and $\mathrm{Bi}(t) \sim c_2 t^{-1/4}\exp((+2/3)t^{3/2})$, so we see that the $\mathrm{Bi}(t)$ function grows faster than exponentially, while $\mathrm{Ai}(t)$ decays faster than exponentially; their ratio grows very quickly indeed and by $t = 10$, it is larger than 10^{18}.

Another way to see this is from the matrix DE for the fundamental solution of the variational equation in the first-order form (which has a slightly different, and larger, condition number than the second order equation, by the way). If

[12] This comment is from a prominent analyst quoted by W. Kahan, as reported by Higham (2002 p. 242) in a different context.

$$\dot{\boldsymbol{\xi}} = \begin{bmatrix} 0 & 1 \\ t & 0 \end{bmatrix} \boldsymbol{\xi}\,, \tag{12.26}$$

with $\xi(0) = \mathbf{I}$, then one can solve this explicitly to get

$$\boldsymbol{\xi} = \begin{bmatrix} f(t) & g(t) \\ f'(t) & g'(t) \end{bmatrix}\,, \tag{12.27}$$

with $f(t) = \alpha_1 \mathrm{Ai}(t) + \beta_1 \mathrm{Bi}(t)$ and $g(t) = \alpha_2 \mathrm{Ai}(t) + \beta_2 \mathrm{Bi}(t)$. The four constants are known explicitly but don't matter much for the present purpose (you can use MAPLE to find them if you like, but they are within reach of hand computation as well if you have a reference on Airy functions handy). A simple lower bound for the largest singular value of $\boldsymbol{\xi}(t)$ follows from noting that $\boldsymbol{\xi}(t)[\alpha_2, -\alpha_1]^T$ contains only $\mathrm{Bi}(t)$ and its derivative, and a similar upper bound for the smallest singular value of $\boldsymbol{\xi}(t)$ follows from noting that $\boldsymbol{\xi}(t)[\beta_2, -\alpha_2]^T$ contains only $\mathrm{Ai}(t)$ and its derivative. This shows that the condition number of the first-order system is bounded below by a constant times

$$\frac{\|\mathrm{Bi}(t), \mathrm{Bi}'(t)\|_2}{\|\mathrm{Ai}(t), \mathrm{Ai}'(t)\|_2}\,, \tag{12.28}$$

which gets very large indeed as $t \to \infty$.

However, the IVP that we obtain by integrating *backward* from $y(10) = \mathrm{Ai}(10)$ and $y'(10) = \mathrm{Ai}'(10)$ is very well-conditioned instead! This is sometimes called a terminal-value problem, by the way. We leave the details to Problem 12.7, but in essence the disturbances are damped in this direction, which is the opposite the growth of $\mathrm{Bi}(t)$. This shows up in the constants (which were left out) in the analytical computation sketched above.

Contrariwise, integrating backward from $y(10) = \mathrm{Bi}(10)$, $y'(10) = \mathrm{Bi}'(10)$ to try to compute $\mathrm{Bi}(t)$ is ill-conditioned because now the $\mathrm{Ai}(t)$ term is (in the parlance) parasitic and grows in this direction; to compute $\mathrm{Bi}(t)$, we are better off to integrate forward, as we tried initially to do for $\mathrm{Ai}(t)$. ◁

Example 12.7. Let us examine another example, in detail, and use numerical computation of the condition number this time. Consider this initial-value problem:

$$\begin{aligned} \dot{x}_1 &= -x_1^3 x_2, & x_1(0) &= 1 \\ \dot{x}_2 &= -x_1 x_2^2, & x_2(0) &= 1 \end{aligned}\,. \tag{12.29}$$

We may solve this simple nonlinear autonomous system analytically to find that

$$x_1(t) = \frac{1}{1 + W(t)} \qquad \text{and} \qquad x_2(t) = e^{-W(t)}\,,$$

where W is the principal branch of the Lambert W function. This function has a derivative singularity at $t = -1/e \doteq -0.3679$. We don't integrate here all the way to that singularity; see Sect. 12.11 for a discussion of what might happen if we did.

Now, if we didn't know the exact solution (and perhaps we wouldn't if we hadn't started with the solution and worked out a differential equation for it), we would solve the problem numerically. As we show, we can also use the theory presented in this section to track the condition number of the problem numerically. Since we can also track the residual numerically as shown before, we have all the ingredients needed for an a posteriori error analysis of our solution.

The trick is to simultaneously solve the associated matrix equation of the variational equation (12.21), $\dot{\boldsymbol{\xi}}(t) = \mathbf{J_f}(\mathbf{z})\boldsymbol{\xi}(t)$ with $\boldsymbol{\xi}(0) = \mathbf{I}$, and the system above. So we first compute the Jacobian by hand (or with MAPLE),

$$\mathbf{J} = \begin{bmatrix} -3z_1^2 z_2 & -z_1^3 \\ -z_2^2 & -2z_1z_2 \end{bmatrix}$$

and then expand the matrix product $\mathbf{J_f}(\mathbf{z})\boldsymbol{\xi}(t)$. We can then create an m-file solving for all the components of

$$\mathbf{y} = \begin{bmatrix} x_1 & x_2 & \xi_{11} & \xi_{12} & \xi_{21} & \xi_{22} \end{bmatrix}^T$$

simultaneously. The function $\mathbf{f}(t, \mathbf{y})$ will then be as follows:

```
function yp=bothodes(t,y)
    yp=zeros(size(y));
    %yp=[x1;x2;xi11;xi12;xi21;xi22];
    yp(1,:)=-y(1,:).^3.*y(2,:);
    yp(2,:)=-y(1,:).*y(2,:).^2;
    yp(3,:)=-3*y(1,:).^2.*y(2,:).*y(3,:)-y(1,:).^3.*y(5,:);
    yp(4,:)=-3*y(1,:).^2.*y(2,:).*y(4,:)-y(1,:).^3.*y(6,:);
    yp(5,:)=-y(2,:).^2.*y(3,:)-2*y(1,:).*y(2,:).*y(5,:);
    yp(6,:)=-y(2,:).^2.*y(4,:)-2*y(1,:).*y(2,:).*y(6,:);
end
```

The use of colon (:) in the code above allows the DE to be evaluated at a vector of t-values. That is, the input y will be a $6 \times N$ matrix corresponding to an N-vector of t values (although t does not appear explicitly in the DE). The reason we "vectorize" this function is so that, unlike previous examples, the residual $\Delta(t)$ can be computed in a single statement instead of in a loop. This practice might confuse the reader (and, to be honest, the writer after leaving the code alone for a while) and isn't really recommended. Indeed, one friendly reviewer of this book (Larry Shampine) suggested that we not do it at all, and stick to using loops to evaluate the residual. However, we decided to show the technique once, anyway. It is efficient if you can make it work. This technique is used in `odeexamples[brussode]` if you wish to see another example.

With the function bothodes defined, we can then use the code below to do all these things:

1. Solve our initial value problem for x_1 and x_2 (lines 2–11);
2. Solve for the components of the fundamental solution matrix ξ_{ij} (lines 2–11 again, since they are solved simultaneously);
3. Find the absolute and relative residual (lines 21–25);
4. Estimate the condition number of the problem (lines 34–44);
5. Plot the results on the appropriate scale (lines 27–32 and 45–46).

In addition, we use the routine RefineMesh discussed earlier to refine the mesh size to obtain clearer graphical output (the reader is invited to use deval on a mesh ignoring adaptive step-size selection to see that the result would not look right).

```
1  function variationaleq_ex
2  % Simple nonlinear autonomous example
3  % dotx1 = -x1^3x2  and dotx2 = -x1x2^2, with x1(0)=x2(0)=1.
4  % J = Jacobian matrix, so the associated matrix variational
       equation is dotXI = J XI, XI(0) = eye(2).
5  tspan = [0,-exp(-1)+5.0e-3];
6  % Initial conditions for x_1, x_2, and xi = eye(2)
7  Y0 = [1,1,1,0,0,1];
8  % Integrate to reasonably tight tolerances
9  opts = odeset( 'reltol', 1.0e-8, 'abstol', 1.0e-8 );
10 % Put the solution in a structure, to compute the residual.
11 sol = ode45( @bothodes, tspan, Y0, opts );
12
13 % We refine the mesh ourselves so as to be sure that our residual
        computation reflects the actual changes in the solution as
       found by ode45.
14 nRefine = 9;
15 n = length( sol.x );
16 size( sol.x )
17 h = diff( sol.x );
18 tRefine = RefineMesh( sol.x, nRefine );
19 numpoints = length(tRefine);
20
21 % Now compute the residual.
22 [yhat,yphat] = deval(sol,tRefine);
23 Resid = yphat - bothodes(tRefine,yhat);
24 % The residual relative to the size of the rhs is also of
       interest.
25 RResid = yphat./bothodes(tRefine,yhat) - 1;
26
27 figure(1),plot(tRefine,Resid(1,:),'k-')
28 set(gca,'fontsize',16);
29 title('Residual␣in␣x_1')
30 figure(2),plot(tRefine,Resid(2,:),'k-')
31 set(gca,'fontsize',16);
32 title('Residual␣in␣x_2')
33 figure(3), semilogy(tRefine,abs(RResid), 'k-')
34 axis([-0.4,0,10E-12,10E-6]);
```

```
set(gca,'fontsize',16);
title('Relative␣Residual')

% Now look at the condition number
sigma1 = zeros(1,n);
sigma2 = zeros(1,n);
cond  = zeros(1,n);
for k=1:n,
    Xt=[sol.y(3,k),sol.y(4,k);sol.y(5,k),sol.y(6,k)];
    sigma = svd(Xt);
    sigma1(k) = sigma(1);
    sigma2(k) = sigma(2);
    cond(k) = sigma1(k)/min(sigma2(1:k));
end
figure(4), semilogy(sol.x,cond,'k')
set(gca,'fontsize',16);
title('Condition␣number')
end
```

The results of this numerical a posteriori analysis of the numerical solution of the initial-value problem are presented in Fig. 12.9. As we observe in Fig. 12.9a–c, the relative residuals of all the components on this interval of integration are smaller than 10^{-6}. So, we have computed the exact solution of

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})\left(\mathbf{I} + 10^{-6}\mathbf{v}(t)\right), \qquad \|\mathbf{v}\|_\infty \leq 1.$$

Moreover, as we observe in Fig. 12.9, the condition number increases significantly as we approach the singularity $1/e$ (reaching $\approx 10^3$ on this scale), which is as expected from what a condition number does. Consequently, as we approach the singularity, we can expect that the reference solution and the computed solution differ as

$$\|\mathbf{z} - \mathbf{x}\| \leq 10^3 \int_{t_0}^{t} 10^{-4}\mathbf{v}(\tau)d\tau = 10^{-1}\int_{t_0}^{t} \mathbf{v}(\tau)d\tau.$$

Naturally, we would get a value better than 10^{-1} for tighter tolerances, or if we looked at the relative forward error. ◁

12.3.3 Condition Analysis Based on the Gröbner–Alexeev Approach

In the previous section, we have assumed that ε was small enough that linearization allowed a good approximation. Now, we consider a full nonlinear perturbation. We also explicitly allow cases when the residual is correlated with the solution of the initial-value problem, as indeed it is usually for numerical methods.

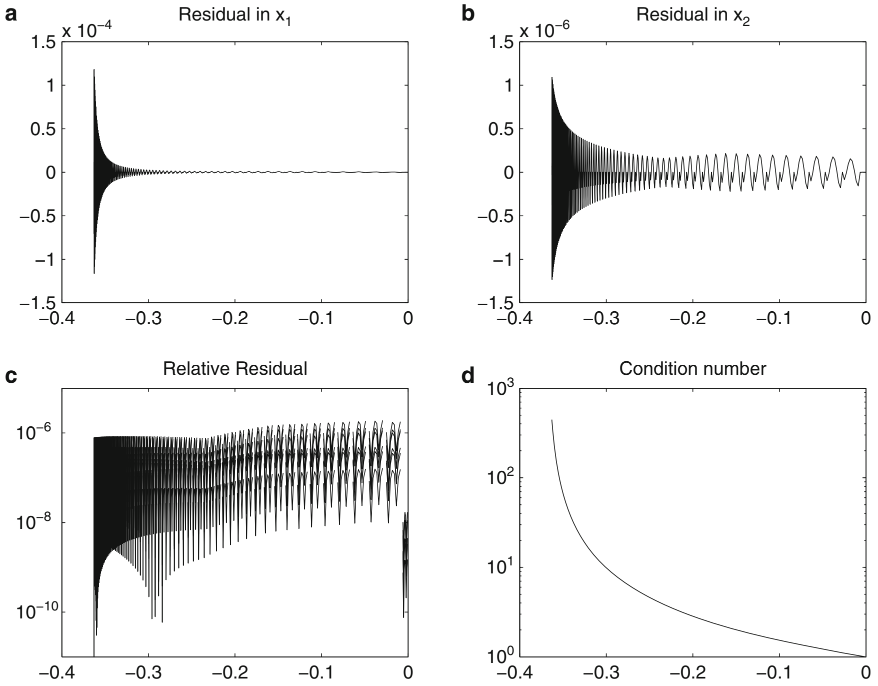


Fig. 12.9 Numerical, a posteriori analysis of the numerical solution of the initial-value problem (12.29) with `ode45`. (**a**) Absolute residual in x_1. (**b**) Absolute residual in x_2. (**c**) Relative residual in the six components of **y**. (**d**) Condition number of the problem

Theorem 12.1 (Gröbner–Alexeev nonlinear variation-of-constants formula). *Let* $\mathbf{x}(t)$ *and* $\mathbf{z}(t)$ *be the solutions of*

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{f}(t,\mathbf{x}(t)), & \mathbf{x}(t_0) &= \mathbf{x}_0 \\ \dot{\mathbf{z}}(t) &= \mathbf{f}(t,\mathbf{z}(t)) + \varepsilon\mathbf{v}(t,\mathbf{z}) & \mathbf{z}(t_0) &= \mathbf{x}_0, \end{aligned}$$

respectively. If $\mathbf{J_f}$ *exists and is continuous, then*

$$\mathbf{z}(t) - \mathbf{x}(t) = \varepsilon \int_{t_0}^{t} \mathbf{G}(t,\tau,\mathbf{z}(\tau))\mathbf{v}(\tau,\mathbf{z}(\tau))d\tau \tag{12.30}$$

where the matrix $\mathbf{G}$ *is given by*

$$G_{ij}(t,\tau,\mathbf{z}(\tau)) := \frac{\partial x_i}{\partial x_{0,j}}(t,\tau,\mathbf{z}(\tau))$$

and acts as a condition number, that is, as a quantity dictating how $\varepsilon\mathbf{v}(t,\mathbf{z})$ *will be magnified over the interval of integration* $[t_0,t_f]$.

We will not give a proof of this theorem. The reader is referred instead to the excellent discussion in Hairer et al. (1993 Chap. I.14).

If we happen to know the analytic solution of the differential equation, then it is possible to compute $\partial x/\partial x_0$ directly (but, of course, this is an artificial situation, and we know the answer in either case).

Example 12.8. Consider this simple (scalar) problem:

$$\dot{x}(t) = x^2(t) \qquad x(t_0) = x_0 \tag{12.31}$$

$$\dot{z}(t) = z^2(t) - \varepsilon z(t) \qquad z(t_0) = x_0\,. \tag{12.32}$$

The solution of this particular nonlinear problem is known analytically:

$$x(t,t_0,x_0) = \frac{x_0}{1 - x_0(t-t_0)}\,,$$

which clearly has a singularity at $t^* = t_0 + 1/x_0$ if $x_0 \neq 0$. This is an example of a *movable pole*, and we consider such cases further in Sect. 12.11. For now we suppose that $x_0 > 0$ and $t_0 \leq t < t^*$. Remark that we have so far written explicitly the dependence of x on t_0 and y_0. This allows us to adopt a convenient notation for $\partial x/\partial x_0$, which is to indicate differentiation with respect to a parameter. In what follows, we will use instead the notation $\partial x/\partial x_0 = D_3 x(t,t_0,x_0)$. Then, we directly find that

$$D_3 x(t,t_0,x_0) = \frac{1}{(1 - x_0(t-t_0))^2} = \frac{1}{(1 - z(t_0)(t-t_0))^2}\,.$$

By Theorem 12.1, the difference between z and x can be written as

$$z(t) - x(t) = -\varepsilon \int_{t_0}^{t} \frac{z(\tau)}{(1 - z(\tau)(t-\tau))^2}\, d\tau.$$

Here again, our life is easy since we can solve $\dot{z} = z^2(t) - \varepsilon z$ analytically, and we find that

$$z(t) = \frac{\varepsilon x_0}{x_0 - (x_0 - \varepsilon)e^{\varepsilon(t-t_0)}}.$$

We can then compute the resulting integral:

$$\begin{aligned}\int_{t_0}^{t} \frac{z(s)}{(1 - z(s)(t-s))^2}\, ds &= \left. \left(\frac{-x_0}{(\varepsilon - x_0)\, e^{\varepsilon\,(s-t_0)} + x_0\,(1 - \varepsilon t + \varepsilon s)} \right) \right|_{t_0}^{t} \\ &= \frac{-x_0}{(\varepsilon - x_0)\, e^{\varepsilon\,(t-t_0)} + x_0} + \frac{x_0}{\varepsilon - x_0 + x_0\,(1 - \varepsilon t + \varepsilon t_0)}.\end{aligned}$$

It turns out to be just the same as $z(t) - y(t)$, as it should be, according to the theorem. ◁

This example (and Theorem 12.1) shows that *if we can differentiate the solution of our differential equation with respect to the initial conditions*, then we can account for perturbations to the differential equation. This raises the question of how to, in general, differentiate a solution $\mathbf{x}(t)$ of a differential equation with respect to an initial condition when we cannot find a formula for $\mathbf{x}(t)$. As shown in Hairer et al. (1993 Chap. I.14), with a simple substitution ($\mathbf{x} = \mathbf{z} + \mathbf{x}_0$), this problem reduces to the easier question of differentiating with respect to a *parameter*. Suppose $\mathbf{x}(t) = \mathbf{x}(t, t_0, \mathbf{x}_0, p) \in \mathbb{C}^n$ is the solution to

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}, p), \qquad \mathbf{x}(t_0) = \mathbf{x}_0,$$

where $p \in \mathbb{C}$ is a parameter, constant for the duration of the solution process. We want to find what $D_4 x_j(t, t_0, \mathbf{x}_0, p) = {}^{\partial x_j}/_{\partial p}$ is.

Suppose $\mathbf{y} = \mathbf{x}(t, t_0, \mathbf{x}_0, p)$ solves the given problem, and $\mathbf{z} = \mathbf{x}(t, t_0, \mathbf{x}_0, p + \Delta p)$ solves the same problem with a slightly different parameter. Then, by taking the Taylor expansion about $\mathbf{y}$ and $p + \Delta p$, we find that the difference $\dot{\mathbf{z}} - \dot{\mathbf{y}}$ satisfies

$$\begin{aligned} \dot{\mathbf{z}} - \dot{\mathbf{y}} &= \mathbf{f}(t, \mathbf{z}, p + \Delta p) - \mathbf{f}(t, \mathbf{y}, p) \\ &= \mathbf{J_f}(t, \mathbf{y}, p)(\mathbf{z} - \mathbf{y}) + D_3(\mathbf{f})(t, \mathbf{y}, p)\Delta p + O(\Delta p)^2. \end{aligned} \tag{12.33}$$

As a result, if

$$\phi_i = \frac{\partial x_i(t, t_0, \mathbf{x}_0, p)}{\partial p} = \lim_{\Delta p \to 0} \frac{x_i(t, t_0, \mathbf{x}_0, p + \Delta p) - x_i(t, t_0, \mathbf{x}_0, p)}{\Delta p},$$

then taking the limit of Eq. (12.33) divided by Δp, as $\Delta p \to 0$, gives

$$\dot{\phi}(t) = \mathbf{J_f}(t, \mathbf{x}, p)\phi(t) + \mathbf{f}_p(t, \mathbf{x}, p)\,. \tag{12.34}$$

Examining the initial solutions of $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{x}, p)$, for example by Euler's method or a higher-order method, shows that $\phi(0) = 0$. That is, in order to take the derivative of $\mathbf{x}(t, t_0, \mathbf{x}_0, p)$ with respect to p, we *differentiate the differential equation* and then solve that resulting linear differential equation. Of course, we may solve this differential equation numerically, along with the solution of the original equation. This is sometimes called the variational method of computing sensitivity.

Now let us adapt this idea to the differentiation of $\mathbf{x}(t)$ with respect to the initial condition, which does not appear in the differential equation (usually). A little thought shows that, instead of Eq. (12.34), we can formulate an equation for $\boldsymbol{\Phi}(t)$, which is a matrix, each entry of which is ${}^{\partial x_i(t, t_0, x_0)}/_{\partial x_{0,j}}$, the derivative with respect to *one component* of the initial condition. It is convenient to package them all at once:

$$\dot{\boldsymbol{\Phi}}(t) = \mathbf{J_f}(t, \mathbf{x})\boldsymbol{\Phi}(t), \tag{12.35}$$

where now the initial condition is $\boldsymbol{\Phi}(t_0) = \mathbf{I}$, the identity matrix. This is, in fact, exactly the first associated matrix variational equation.

In more complicated examples, of course, one must solve for $\mathbf{x}(t)$ and $\Phi(t)$ simultaneously, by some numerical scheme. It has been our experience that MATLAB is perfectly satisfactory for the process, once a Jacobian $\mathbf{J_f}(t,\mathbf{x})$ has been coded; of course, computer algebra systems help with that task, as we show in the extended example in Sect. 12.4.

12.3.4 A Crude Practical Estimate of the Condition Number

We may find a crude estimate—in fact, a lower bound, and unfortunately often not a very good lower bound—for the condition number merely by solving the problem twice, at different tolerances. The trick is to solve the problem first at a loose tolerance, giving a solution that we call $\mathbf{z}_1(t)$ satisfying $\dot{\mathbf{z}}_1 = \mathbf{f}(t,\mathbf{z}_1) + \Delta_1(t)$ and another solution at a tighter tolerance giving a solution that we call $\mathbf{z}_2$(t), satisfying $\dot{\mathbf{z}}_2 = \mathbf{f}(t,\mathbf{z}_2) + \Delta_2(t)$. By assumption, $\|\Delta_2\| \ll \|\Delta_1\|$. Therefore, the difference between $\mathbf{z}_1$ and $\mathbf{z}_2$ is almost entirely due to the larger perturbation Δ_1, and the condition number must be at least $\|\mathbf{z}_1 - \mathbf{z}_2\|/\|\Delta_1\|$. After the condition number has been estimated, one can use it together with the size of the computed residual in the best solution to estimate the forward error in the best solution. We illustrate this by an example.

Example 12.9. Consider again the differential equation $\dot{x}(t) = x^2(t) - t$ with initial condition $x(0) = -1/2$, this time on the interval $0 \le t \le 5$. We solve this using `ode45` using tolerances 10^{-8}, and again with tolerances 10^{-10}, using this code:

```
West = @(t,x) x.^2 - t ;  % vectorized right-hand side of the DE
b = 5;
opts = odeset('reltol',1.0e-8,'abstol',1.0e-8);
sol1 = ode45( West, [0,b], [-1/2], opts );
opts = odeset('reltol',1.0e-10,'abstol',1.0e-10);
sol2 = ode45( West, [0,b], [-1/2], opts );
t = RefineMesh( sol2.x, 20 );
[z1,dz1] = deval( sol1, t );
[z2,dz2] = deval( sol2, t );
res1 = dz1 - West( t, z1 );
res2 = dz2 - West( t, z2 );
e = z1 - z2;
CrudeK = max(abs(e))/max(abs(res1))
figure(1),plot( t, res1, 'k.'),set(gca,'fontsize',16),xlabel('t')
    ,ylabel('Delta_1')
figure(2),plot( t, e, 'k.'),set(gca,'fontsize',16),xlabel('t'),
    ylabel('z_1␣-␣z_2')
figure(3),plot( t, res2, 'k.'),set(gca,'fontsize',16),xlabel('t')
    ,ylabel('Delta_2')
Maple_reference_x5 = -2.1827854817673358944;
forward_error = deval( sol2, 5 ) - Maple_reference_x5
```

This gives a solution with a residual $\|\Delta_1\|$ about 3×10^{-7} and an estimated forward error $\|z_1 - z_2\|$ about 6×10^{-9}; this then gives an estimated lower bound for the condition number of 0.0165. Since the residual of the tighter tolerance solution is about $\|\Delta_2\| = 1 \times 10^{-8}$, using this condition number suggests that the forward error in the tighter tolerance solution (compared to the at-this-moment-unknown reference solution) will be about $0.0165 \cdot 10^{-8}$, that is, 1.65×10^{-10}. Using MAPLE, we can in fact find the reference solution for this problem in terms of Airy functions. This, of course, is very unusual, but this time we can do it and evaluate the reference solution at, say, $t = 5$ to get $x(5) = -2.18278548176733589\ldots$. Comparison of the tighter solution with this reference value shows a forward error of about 2.9×10^{-11}, which is satisfactorily smaller than the estimate just obtained. Though crude, this technique can supply useful information, and it is much easier than solving the variational equation. ◁

12.4 An Extended Example: The Restricted Three-Body Problem

What follows is a discussion of a much-used example of numerical solution of initial-value problems for ordinary differential equations. The model in question dates back to Arenstorf (1963), where it was derived as a *perturbation* model intended to analyze part of the three-body problem in the limit of when one of the masses goes to zero. Further discussions of this model can be found in Hairer et al. (1993) and in Shampine and Gordon (1975), and in MATLAB it is taken up in the `orbitode` demo.

In spite of its simplistic nature (or perhaps because of it), the example is a good traditional one, in that it shows the value of adaptive step-size strategies for efficiency, and in `orbitode` makes a good example of event location (see Sect. 12.8) being used to detect a periodic orbit. Since the example has been well studied, there are a great many solutions existing in the literature to which the reader's computation can be compared. Furthermore, it is interesting and produces surprising pictures. However, it is (nowadays) a curious problem, in that there seems little point in the idealizations that give rise to the model: One might as well integrate Newton's equations of motion directly, and indeed direct simulations of the solar system are of great interest and are highly advanced.[13] Nonetheless, we use the model equations and extend the discussion a bit to show that the example is also excellent for showing how residual (defect) computations can be interpreted in terms of the physical modeling assumptions: In particular, we will see that if we have the solution of a model with a small residual, then we have just as good a solution as the reference solution would be. We also take the discussion a bit further and exhibit the conditioning of the initial-value problem.

In the Arenstorf model there are three bodies moving under their mutual gravitational influence.[14] Two of the bodies have nontrivial mass and move about

[13] See, for example, the JPL simulator, at `http://space.jpl.nasa.gov/`.

[14] We do not claim to be computational astronomers. This discussion is by no means complete and should not be considered authoritative. The discussion is intended only to motivate the model and

their mutual center of gravity in *circular* orbits. If we are thinking of these bodies as the Earth and the Moon, then already this is an idealization: The Earth–Moon system more nearly has the Moon moving elliptically, with eccentricity about 0.05, about the center of gravity of the Earth–Moon system. If we are thinking about the Sun–Jupiter pair, then again Jupiter's orbit is not circular but rather eccentric. So, again, a circular orbit is an idealization. The third body in the model is taken to be so small that its influence on the two larger bodies can be supposed negligible. One thinks of an artificial satellite, with mass less than 1000 tonnes (10^6 kg). The mass of the Moon is $M_M = 7.3477 \times 10^{22}$ kg, so the satellite's mass is less than 10^{-16} of that (coincidentally, not too different from the unit roundoff in MATLAB). Therefore, supposing that this body does not affect the other two bodies is a reasonable assumption. By making this assumption, the actual mass of the satellite drops out of the computation.

Another assumption, common in gravitational models since Newton, is that the bodies act as *point masses*. Neither the Earth, the Moon, the Sun, nor Jupiter, nor even the satellite, is a point mass. In fact, the radius of the Earth is about $1/60$ the Earth–Moon distance, and while the gravitational effects of a uniform spherical body are indeed, when outside the body, identical in theory to those of a point mass, the Earth is neither uniform nor exactly spherical (it's pretty close, though, with a radius of $6,378.1$ km at the equator and $6,356.8$ km at the poles). Similarly for Jupiter, which departs from sphericity by more than the Earth does ($71,492$ km radius at the equator and $66,854$ km radius at the poles). Most importantly, the bodies are inhomogeneous (that is, lumpy inside with pieces of differing density) and rotating. This departure from ideal point-mass gravity has a potentially significant effect on the satellite's orbit.

Finally, we neglect the influence of the other bodies in the Solar system. For the Earth–Moon system, neglecting the influence of the Sun, which differs at different points of the satellite's orbit around the Earth–Moon pair, means that we are neglecting forces about 10^{-11} of the base field; this seems to be a smaller influence than the eccentricity of the orbit, and smaller than the effects of departure from the point-mass idealization, but larger than the effects of the trivial mass of the satellite. For the Sun–Jupiter pair, this is not a bad assumption—the most significant other body is Saturn, and Saturn is far enough away that its effects are detectable only cumulatively.

Once these assumptions are made, then we put M equal to the mass of the largest body, and m equal to the mass of the smaller nontrivial body; the total mass of the system is then $M+m$, and we place the origin of our coordinate system at the center of gravity. The larger mass is treated as a point at distance

$$-\mu = -\frac{m}{M+m} \tag{12.36}$$

to remind the reader of some of the idealizations made, for comparison with the numerical effects of simulation.

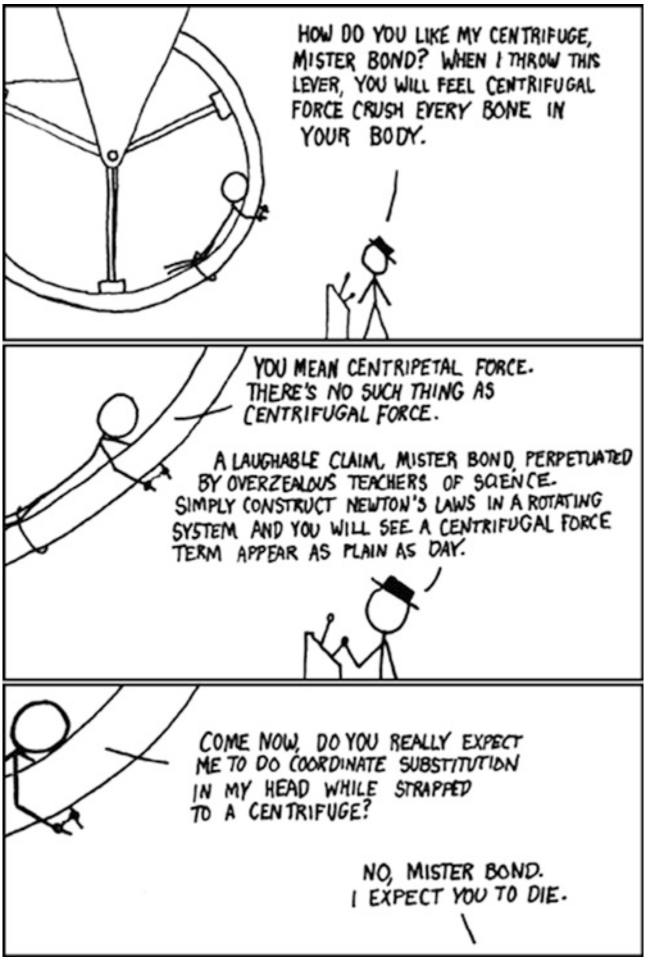

HOW DO YOU LIKE MY CENTRIFUGE, MISTER BOND? WHEN I THROW THIS LEVER, YOU WILL FEEL CENTRIFUGAL FORCE CRUSH EVERY BONE IN YOUR BODY.
YOU MEAN CENTRIPETAL FORCE. THERE'S NO SUCH THING AS CENTRIFUGAL FORCE.
A LAUGHABLE CLAIM, MISTER BOND, PERPETUATED BY OVERZEALOUS TEACHERS OF SCIENCE. SIMPLY CONSTRUCT NEWTON'S LAWS IN A ROTATING SYSTEM AND YOU WILL SEE A CENTRIFUGAL FORCE TERM APPEAR AS PLAIN AS DAY.
COME NOW, DO YOU REALLY EXPECT ME TO DO COORDINATE SUBSTITUTION IN MY HEAD WHILE STRAPPED TO A CENTRIFUGE?
NO, MISTER BOND. I EXPECT YOU TO DIE.

Fig. 12.10 http://xkcd.com/123/. For the alt-text, see the original comic. Reproduced under the blanket permission http://xkcd.com/license.html

from the origin, in units of the diameter of the orbit, and the smaller mass is then at $\mu^* = 1-\mu$.

In the *rotating coordinate system* (see Fig. 12.10) that fixes the large body at $-\mu$ and the small body at $1-\mu$, the equations of motion of the tiny satellite are given in Arenstorf (1963) as follows:

$$\begin{aligned}\ddot{x} &= 2\dot{y} + x - \mu^* \frac{x+\mu}{R_{13}^3} - \mu \frac{x-\mu^*}{R_{23}^3} \\ \ddot{y} &= -2\dot{x} + y - \mu^* \frac{y}{R_{13}^3} - \mu \frac{y}{R_{23}^3}, \end{aligned} \tag{12.37}$$

where the distances $R_{13} = \sqrt{(x+\mu)^2 + y^2}$ and $R_{23} = \sqrt{(x-\mu^*)^2 + y^2}$ between the tiny satellite and the massive body and the minor body, respectively, are consequences of Newton's law (and an assumption of smallness of μ). The problem parameters used by the MATLAB demo `orbitode` are

```
% Problem parameters
mu = 1 / 82.45;
mustar = 1 - mu;
y0 = [1.2; 0; 0; -1.04935750983031990726];
tspan = [0 7];
```

Note that ${}^{1}\!/_{82.45} \doteq 0.01213$. The value of μ used by Hairer et al. (1993) is, however, $\mu = 0.012277471$, and they use 30 decimals in their initial conditions for their periodic orbits. On the other hand, if we take the values of the Earth's mass and the Moon's mass given in Wikipedia, we get $\mu = 0.0121508\ldots$, different in the fourth significant figure from either of these two. While these differences in parameters are alarming, if we take the model equations at all seriously, they are not terribly significant given that the model's derivation has neglected things like the eccentricity of size about 0.05. Another modification we made was to use `ode113` instead of `ode45`, which for this problem made for smoother plots of the residual (but was otherwise fairly similar). Finally, Arenstorf's derivation has at one point replaced ${}^{\mu}\!/_{(1-\mu)}$ with just μ, a simplification from the point of view of perturbation theory but no simplification at all for numerical solution; this makes a further difference of about $0.012^2 = 1.4 \times 10^{-4}$ in the terms of the equation.

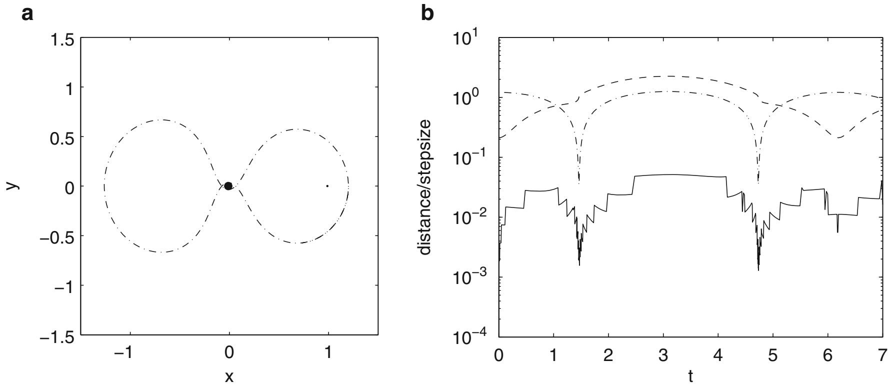


Fig. 12.11 Solution of the Arenstorf model restricted three-body problem for the same parameters and initial conditions as are in `orbitodedemo`, but using `ode113` instead of `ode45`, intended to be similar to a small satellite orbiting the Earth–Moon system in a coordinate system rotating with the Earth and Moon. When distances to either massive body are small, step sizes (computed as speed times delta t) also get small. (**a**) Restricted three-body problem orbit. (**b**) Distances and stepsizes

Computing the solution to this problem with the parameters given above and analyzing the results, we find Figs. 12.11, b, and 12.12a. We see that the step sizes

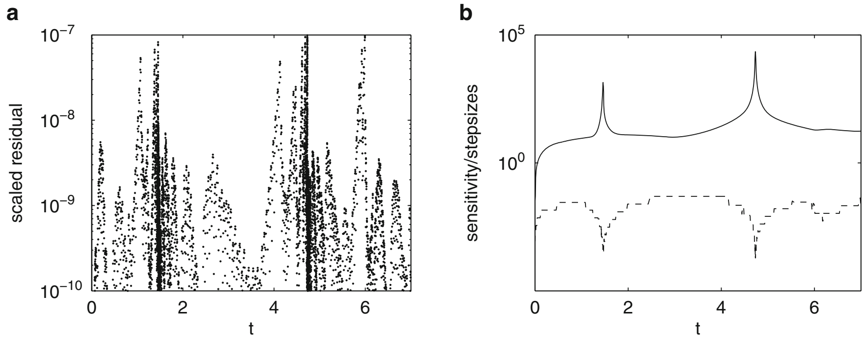


Fig. 12.12 Measured residual and sensitivity in the Arenstorf restricted three-body problem. (**a**) Residual for solution by `ode113` with relative tolerance 1.0×10^{-6}. (**b**) Sensitivity by solving the variational equations (12.35) using `ode113` with relative tolerance 1.0×10^{-6}

are not uniform—indeed, they change over the orbit by a factor of nearly 1000. As seems physically reasonable, the steps are small when the satellite is near one of the massive bodies (and thus experiencing relatively large forces and accelerations). We also see, from Fig. 12.12a, that the residual is small, never larger than 10^{-5}, approximately. Since we have neglected terms in the equations whose magnitude is 10^{-2} or so, we see that we have found a perfectly adequate model solution: This plot tells us precisely as much as the (unobtainable) reference solution to the Arenstorf restricted three-body problem would.

We really also need to know how sensitive these orbits are to changes in μ or to the initial conditions are. Using the first variational equation and plotting the norm of the fundamental solution matrix (i.e., based on results from Chap. 4, the largest singular value) in Fig. 12.11b, we see that the orbit gains sensitivity as the satellite plunges toward the Earth but shows a rapid decrease in sensitivity as the satellite moves away; both of these observations agree with intuition.

In detail, here is how it works. We can use our variational equation

$$\dot{\boldsymbol{\Psi}} = \mathbf{J}_{\mathbf{f}}(\mathbf{x})\boldsymbol{\Psi} + \mathbf{f}_{\mu},$$

where $\boldsymbol{\Psi}(0) = 0$ and the Jacobian matrix $\mathbf{J}_{\mathbf{f}}(\mathbf{x})$ and the partial derivative $\mathbf{f}_{\mu}$ are easily computed by a computer algebra system. Here are the MAPLE commands used to do so for this example.

```
f[1] := y[3];
f[2] := y[4];
mustar := 1-mu;
r13 := ((y[1]+mu)^2+y[2]^2)^(3/2);
r23 := ((y[1]-mustar)^2+y[2]^2)^(3/2);
f[3] := 2*y[4]+y[1]-mustar*(y[1]+mu)/r13-mu*(y[1]-mustar)/((1-mu)
     *r23);
f[4] := -2*y[3]+y[2]-mustar*y[2]/r13-mu*y[2]/((1-mu)*r23);
with(LinearAlgebra);
with(VectorCalculus);
J := Jacobian([seq(f[k], k = 1 .. 4)], [seq(y[k], k = 1 .. 4)]);
```

```
CodeGeneration[Matlab](J, optimize);
fp := map(diff, [seq(f[k], k = 1 .. 4)], mu);
CodeGeneration[Matlab](Vector(fp), optimize);
```

The output of these commands are then bundled up into MATLAB m-files and used in the call to `ode113`. We explain below how we created a modified version of MATLAB's `orbitode`. We first define the parameters of the problem:

```
function t=orbitode113jac

% Problem parameters
mu = 1 / 82.45; % Shampine & Gordon
mustar = 1 - mu;
y0 = [1.2; 0; 0; -1.04935750983031990726; 0; 0; 0; 0]; % Shampine & Gordon
tspan = [0 7]; % Shampine & Gordon
options = odeset('RelTol',1e-8,'AbsTol',1e-10,'Events',@events);
```

As before, we solve the problem with the command

```
[t,y,te,ye,ie] = ode113(@f,tspan,y0,options);
```

The command is slightly different than in the previous examples, since we use the "event location" feature. We then plot the results using the following list of commands:

```
% Plotting circles representing earth and moon
rade = 6371/384403;
radm = 1737/384403;
th=linspace(-pi,pi,101);
esx = rade*cos(th)-mu;
esy = rade*sin(th);
msx = radm*cos(th)+1-mu;
msy = radm*sin(th);
close all
figure
% Plot timesteps scaled by speed so they are "distance-steps"
avevel = (y(1:end-1,3:4)+ y(2:end,3:4) )/2;
avespeed = sqrt( avevel(:,1).^2 + avevel(:,2).^2 );
%Plots Below: 'k--' line is the distance of the satellite to the moon; 'k' line is the time-steps scaled by average speed; 'k-.' line is the distance of the satellite to the earth
semilogy( t, sqrt( (y(:,1)-mustar).^2 + y(:,2).^2 ), 'k--',t(2:end), diff(t).*avespeed, 'k', t, sqrt( (y(:,1)+mu).^2 + y(:,2).^2 ), 'k-.' ),set(gca,'fontsize',16)
axis([t(1),t(end),1.0e-4,1.0e1]),xlabel('t','fontsize',18),ylabel('distance/stepsize','fontsize',18)
figure
plot(y(:,1),y(:,2),'k-.',esx,esy,'k',msx,msy,'k'),set(gca,'fontsize',18)
hold on
% Estimate of the size of filled circle was obtained by trial and error on actual radius of earth and moon figures, later not needed.
scatter( -mu, 0, 50, 'k', 'filled');
scatter( 1-mu, 0, 50*(radm/rade)^2, 'k', 'filled' );
axis([-1.5,1.5,-1.5,1.5])
axis('square')
xlabel('x','fontsize',18),ylabel('y','fontsize',18)

hold off
figure
semilogy(t(2:end),diff(t), 'k--', t,sqrt(y(:,5).^2+y(:,6).^2+y
    (:,7).^2+y(:,8).^2),'k'),set(gca,'fontsize',18)
xlabel('t','fontsize',18),ylabel('sensitivity/stepsizes','
    fontsize',18)
axis([t(1),t(end),1.0e-5,1.0e5])
figure

% Inefficiently, solve it again, so we may compute the residual (
    should have done this the first time)
sol = ode113(@f,tspan,y0,options);
tt = RefineMesh( sol.x, 10 );
np = length(tt);
[yb,ypb] = deval( sol, tt );
res = zeros(1,np);
speed = zeros(size(res));
for i=1:np,
    rr = ypb(:,i) - f(tt(i), yb(:,i) );
    res(i) = norm(rr,inf); speed(i) = norm( ypb(:,i), inf );
end;
```

It remains to describe the code for the function handle `@f` passed to the command `ode113`. The code begins as

```
  function dydt = f(t,y)
  % Derivative function -- mu and mustar shared with the outer
      function.
    r13 = ((y(1) + mu)^2 + y(2)^2) ^ 1.5;
    r23 = ((y(1) - mustar)^2 + y(2)^2) ^ 1.5;

    % Jacobian computed by Maple
    cg0 = zeros(4,4);
    cg2 = zeros(4,1);
    t1 = 1 - mu;
```

and then it is followed by the MAPLE-generated MATLAB code (which we do not display here—computers don't tend to generate readable code). Using these, we finally define our function `dydt=f(t,y)`:

```
    dydt = [ y(3)
              y(4)
              2*y(4) + y(1) - mustar*((y(1)+mu)/r13) - mu/mustar
                  *((y(1)-mustar)/r23)
             -2*y(3) + y(2) - mustar*(y(2)/r13) - mu/mustar*(y(2)/
                 r23)
             cg2(1)
             cg2(2)
             cg2(3)
             cg2(4)];
```

Thus, the use of numerical solutions in combination with computer algebra systems is a very effective way to estimate the condition of an initial-value problem (or, in other words, the "sensitivity" of the orbits).

Remark 12.2. Note that the residual as computed in that example was in the first-order system, not in the second-order original equations. This is potentially misleading: For a true residual, we would have had to interpolate $x(t)$ and $y(t)$ by a twice continuously differentiable interpolant and then substitute that interpolant into the *original* equations. However, this seems to lose a bit of accuracy and the resulting residual, while physically interpretable, is quite a bit larger. See Problem 13.18. ◁

12.5 Structured Backward Error for ODE

In this section, we apply the Oettli–Prager backward error idea to linear ordinary differential equations. This will show how to find a structured backward error, given a solution with a small residual. We work by an example. Suppose that we are interested in solving the linear IVP

$$\mathbf{E}(t)\dot{\mathbf{x}}+\mathbf{F}(t)\mathbf{x}=\mathbf{b}(t)\,, \tag{12.38}$$

subject to the initial condition $\mathbf{x}(t_0)=\mathbf{x}_0\in\mathbb{C}^n$. Both $\mathbf{E}$ and $\mathbf{F}$ are complex matrices, and for the moment we assume that $\mathbf{E}$ is nonsingular, although perhaps we don't wish to invert it explicitly (it may be tridiagonal, for example, and the inverse would cause more trouble than it was worth). The matrix $\mathbf{E}$ is often termed a "mass matrix" because such matrices in ODE occur in spring-mass systems, but there are other contexts in which they occur as well. Many of the MATLAB codes have a provision for solving systems with mass matrices. To be concrete, let

$$\mathbf{E}=\begin{bmatrix} 2 & 1 & & \\ 1 & 2 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & 2 \end{bmatrix} \tag{12.39}$$

be a familiar constant tridiagonal matrix. Let $\mathbf{F}$ be a similar matrix, but with -1s on the sub- and superdiagonals instead. Take the first entry of $\mathbf{b}(t)$ to be $\sin(t)$ and zero otherwise.

Now suppose that we have solved the system using, say, `ode15s`. We will exhibit code that does so, shortly. Suppose that the solution called $\mathbf{x}(t)$ mathematically is contained in the variable `sol` and that as usual we may evaluate it, and its derivative, anywhere using `deval`. The question that arises now is, can we find

$$\begin{aligned}\min\Big\{\varepsilon : (\mathbf{E}+\Delta\mathbf{E})\dot{\mathbf{x}}(t)+(\mathbf{F}+\Delta\mathbf{F})\mathbf{x}(t)=\mathbf{b}+\Delta\mathbf{b},\\ |\Delta\mathbf{E}|\leq\varepsilon|\mathbf{E}|,|\Delta\mathbf{F}|\leq\varepsilon|\mathbf{F}|,|\Delta\mathbf{b}|\leq\varepsilon|\mathbf{b}|\Big\}\,?\end{aligned} \tag{12.40}$$

The answer is yes we can, for each fixed t, by using the Oettli–Prager theorem. For fixed t, this is just a matrix–vector product! The computed quantities $\dot{\mathbf{x}}(t)$ and $\mathbf{x}(t)$ are available via `deval`, and so we may simply write down a formula for the structured backward error:

$$\varepsilon = \max_i \frac{|r_i(t)|}{|b_i| + \sum_{j=1}^{n} \left(|e_{i,j}||\dot{x}_j(t)| + |f_{i,j}||x_j(t)|\right)} . \tag{12.41}$$

As usual, if the numerator is zero, we may take the fraction to be zero, but if the denominator is zero while the numerator is not, the structured backward error is infinite.

We use the code

```
function [ sol, tresid, resid, eback ] = OPExample( n )
% OPExample --- solve Ey' = -Fy + b, dimension n, and
%               return the structured backward error with
%               the solution
%
    D = sparse(1:n,1:n,2*ones(1,n),n,n);
    S = sparse(2:n,1:n-1,ones(1,n-1),n,n);
    E = S+D+S';
    opts = odeset( 'Mass', E, 'MStateDependence', 'none', 'MassSingular', 'no', 'RelTol', 1.0e-7, 'AbsTol', 1.0e-7 );
    function [DYP] = defun(DT,YD)
        DYP = -2*YD + [YD(2:end);0] + [0; YD(1:end-1)] + [sin(DT); zeros(n-1,1)] ;
    end

    sol = ode15s(@defun, [0,5], zeros(n,1), opts );
    [tresid, resid, yhat, yphat ] = ODEResidualMass( sol, E, @defun, 5 ); % 5 pts per step
    nump = length(tresid);
    eback = zeros( size(tresid) );
    lambda = zeros(n,1);
    for j=1:nump,
        if resid(j)~=0,
          lambda(1) = abs(resid(j))/(2*abs(yphat(1,j)) + abs(yphat(2,j)) ...
                            + 2*abs(yhat(1,j)) + abs(yhat(2,j)) + 1);
          for i=2:n-1,
              lambda(i) = abs(resid(j))/(abs(yphat(i-1,j))+2*abs(yphat(i,j)) + ...
                                abs(yphat(i+1,j)) + abs(yhat(i-1,j)) + 2*abs(yhat(i,j)) ...
                                + abs(yhat(i+1,j)) );
          end
          lambda(n) = abs(resid(j))/(2*abs(yphat(n,j)) + abs(yphat(n-1,j)) ...
                            + 2*abs(yhat(n,j)) + abs(yhat(n-1,j)) );
```

```
            end;
            eback(j) = max( lambda ) ;
        end

    end
```

which relies on

```
    function [ t, resid, yhat, yphat ] = ODEResidualMass( sol, M, f,
        nRefine )
    %Compute the residual of the solution sol in the de My' = f(t,y)
    %   ODEResidualMass( sol, M, f, nRefine )
        if nargin < 4,
            nRefine = 4;
        end
        t = RefineMesh( sol.x, nRefine );
        nump = length(t);

        [yhat,yphat] = deval( sol, t );
        resid = zeros( size(yphat) );
        yphat = M*yphat;
        for i=1:nump,
            resid(:,i) = yphat(:,i) - f(t(i), yhat(:,i));
        end

    end
```

The commands

```
[ sol, t, del, eback ] = OPExample( 8 );
figure(3), plot( sol.x, sol.y, 'k' )
figure(2), semilogy( t, abs(del), 'k.' )
figure(1), semilogy( t, eback, 'k' )
```

produce figures showing that the structured backward error, which is now a function of t, behaves (for $n = 8$) quite well, being smaller than the residual (Fig. 12.13). That is, we have the exact solution of a (weakly) time-varying problem of the same type as the original. It should be clear that this idea applies to many kinds of linear problems—and indeed to any kind of problem in which the parameters appear linearly. A similar idea, the method of modified equations, is studied in Sect. 13.7.

12.6 What Good are Numerical Solutions of Chaotic Problems?

We begin with an example.

Example 12.10. Consider the Rössler system

$$\begin{aligned}\dot{x} &= -y - z\\ \dot{y} &= x + ay\\ \dot{z} &= b + z(x - c)\end{aligned} \tag{12.42}$$

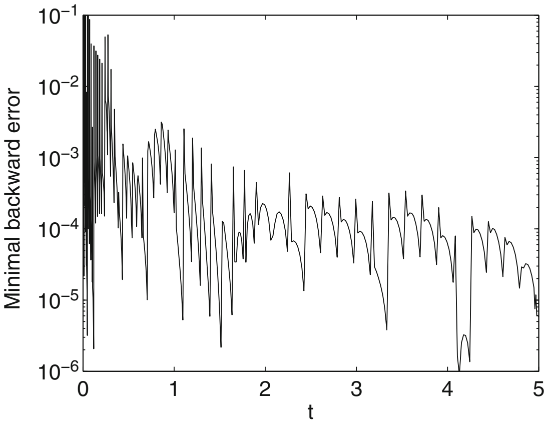


Fig. 12.13 Size of the componentwise backward error $\varepsilon(t)$ in solving the case $n = 8$ of (12.38) on $0 \le t \le 5$ by formula (12.41)

with the commonly used parameter values $a = b = 0.1$ and $c = 14$. We take an essentially arbitrary initial condition, $[x, y, z] = [0, 1, 1]$. Using tight tolerances and the solver `ode113`, we solve this on the interval $0 \le t \le 300$, to get Fig. 12.14:

```
function RoesslerDefect
%ROESSLERDEFECT  Defect in Roessler equations.
%  Chaotic values of parameters

tspan = [0 300];
y0 = [0; 1; 1];

% solve the problem using ODE45
opts = odeset('RelTol',1.0e-10,'AbsTol',[1.0e-11,1.0e-11,1.0e
    -11]);
Roesslersol = ode45(@f,tspan,y0,opts);
tref = RefineMesh( Roesslersol.x );
[y,dy] = deval( Roesslersol, tref );
residual = zeros( size(y) );
numt = length(tref);
for i=1:numt,
    residual(:,i) = dy(:,i) - f(tref(i),y(:,i));
end;
max(max(abs(residual)))
figure(1), plot3( Roesslersol.y(1,:), Roesslersol.y(2,:),
    Roesslersol.y(3,:), 'k' );
set(gca,'fontsize',16)
axis([-20,30,-40,20,0,40]);
set(gca,'XTick',-20:10:30);
figure(2), semilogy( tref, abs( residual ), 'k.' )
set(gca,'fontsize',16)
h = diff( Roesslersol.x );
figure(3), semilogy( Roesslersol.x(2:end), h, 'k.' )
```

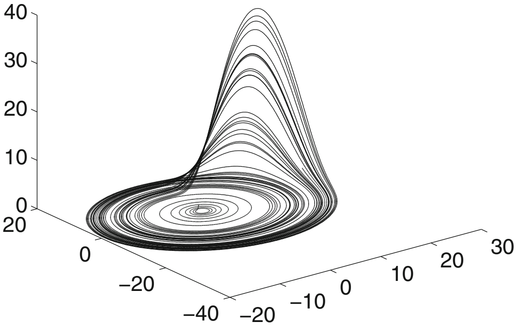


Fig. 12.14 The Rössler attractor for $a = b = 0.1$, $c = 14$, computed to tight tolerances using `ode113`

```
set(gca,'fontsize',16)

% ---------------------------------------
function dydt = f(t,y)
a = 0.1;
b = 0.1;
c = 14;  % Wikipedia values
dydt = [   -y(2)-y(3)
           y(1)+a*y(2)
           b + y(3)*(y(1)-c) ];
```

This system is one of the simplest continuous dynamical systems that generate *chaos*. The trajectories are exponentially sensitive (in a way that we will make clearer later) to perturbations in the parameter values or the initial conditions, to truncation errors, or to rounding errors. Therefore, the one thing that we know about the computed solution presented in this figure is that it must differ in detail, by $O(1)$, from the reference solution.

But we *have* computed an exact solution: an exact solution to the equations

$$\begin{aligned}\dot{x} &= -y - z + \varepsilon v_x(t)\\ \dot{y} &= x + ay + \varepsilon v_y(t)\\ \dot{z} &= b + z(x - c) + \varepsilon v_z(t),\end{aligned} \tag{12.43}$$

where each of v_x, v_y, and v_z is less than 1 in magnitude and $\varepsilon = 2 \cdot 10^{-7}$. Since the magnitudes of the solution components can be as large as 40, and the final equation contains a product term, then we suppose that the perturbations introduced by the numerics are indeed very small. ◁

The key thing, though, is whether or not these perturbations are small *compared to physical perturbations*. The model was not constructed as a physical model, however, but rather as a deliberate attempt to create continuous chaos (unlike the Lorenz system). So, in this case, one can quite legitimately question whether or not a small residual gives us what we want. A further subtlety is that the perturbations depend on t, and thus our numerical solution is the exact solution of a four-dimensional model, not a three-dimensional model.

Suppose, however, that we built a mechanism purposely to behave in a way that is predicted by these equations (this can be done; essentially building an analog computer). Then, if that machine is sitting in a lab on a table, it would feel external vibrations due to earthquakes, or trucks passing by on the highway nearby, or even just the vibrations caused by students walking down the hallway between classes.[15] Would such perturbations invalidate the computations of the mechanism? We contend that they would not.

Chaotic systems can be studied from many perspectives and, accordingly, chaotic motion can be defined in many ways.[16] In all cases, some intuitions drawn from physics motivate the various approaches:

> The concepts of "order" and "determinism" in the natural sciences recall the predictability of the motion of simple physical systems obeying Newton's laws: The rigid plane pendulum, a block sliding down an inclined plane, or motion in the field of a central force are all examples familiar from elementary physics. In contrast, the concept of "chaos" recalls the erratic, unpredictable behavior of elements of a turbulent fluid or the "randomness" of Brownian motion as observed through a microscope. For such chaotic motions, knowing the state of the system at a given time does not permit one to predict it for all later times. (Campbell and Rose 1983 vii)

The idea is that a chaotic motion $\mathbf{x}(t)$ satisfying a deterministic nonlinear differential equation $\dot{\mathbf{x}}(t) = \mathbf{f}(t, \mathbf{x}(t))$ is *bounded* [i.e., $\mathbf{x}(t)$ does not go to ∞ as $t \to \infty$], *aperiodic* [i.e., for no T does $\mathbf{x}(t) = \mathbf{x}(t+T)$], and extremely *sensitive to initial conditions*. Now, if two trajectories $\mathbf{x}(t)$ and $\mathbf{z}(t)$ were uniformly diverging (i.e., if the distance between the two trajectories were continuously increasing exponentially with t), at least one of them would be unbounded. But because of the nonlinearity of the equation, the distance between the two curves varies in very erratic ways. It is thus practically impossible to track how close our two trajectories are from one another in the long run (and often even in the short run!). To establish the sensitivity to initial conditions, the important thing is that, *on average*, for finite time, the trajectories diverge from each other exponentially quickly. This is exactly what positive Lyapunov exponents (as defined in Eq. (12.25), and computed by the SVD) show. But then, it also follows that solutions of chaotic initial-value problems are exponentially ill-conditioned.

[15] One cannot imagine such a machine being built anywhere but at a place where they teach dynamical systems. Of course, such a place would have hordes of students.

[16] Martelli et al. (1998 p. 112) claim amusingly, "[W]ith a bit of exaggeration, that there are as many definitions of chaos as experts in this new area of knowledge." Concerning some of the conceptual issues involved with different definitional attempts, see Batterman (1993).

This situation raises two related questions regarding the reliability of numerical methods:

1. Are the computed trajectories satisfactory, and in what sense are they to be regarded as being satisfactory?
2. Are the numerical methods introducing spurious chaos or suppressing actual chaos?

These questions have generated a substantial amount of debate. We take only the simplest, least computationally expensive point of view and justify the computations by using the residual.

The first and foremost observation to make is that, because chaotic problem are exponentially ill-conditioned (they have positive Lyapunov exponents), one cannot hope to obtain numerical solutions with a small forward error. Accordingly, errors in the initial conditions, discretization error, and even rounding error will be exponentially magnified as time increases. As a result, one can obtain a huge forward error, that is, an important lack of correlation between the projected motion and the actual motion. But is this a reason to claim that our algorithm is a bad one, that is, that it gives results that are unfaithful to the model?

No. The solutions can be satisfactory in the backward sense, since it is unreasonable to ask of a numerical method more than the problem allows for. For chaotic problem, good methods such as the one implemented in `ode45` *verifiably solve a problem near the reference problem.* As we explained in Chap. 1, for genuine (as opposed to artificial) problems, we will have to consider physical perturbations anyway.

This means there are other aspects of chaotic systems that are *required* by the backward error perspective. Even if the *trajectory* $\mathbf{x}(t)$ is sensitive to initial conditions, some features of chaotic systems *must not be*—for instance, the dimension of the attractor, possibly, or the measure on the attractor, or some other such quantity of physical interest. If *no* quantity of physical interest were robust under physical perturbations, then the behavior of the system would be unpredictable by simulation or modeling at all! But if at least one physically interesting quantity is insensitive to physical perturbations, then a numerically stable method of solution can be expected to give us good information about that quantity.

We repeat: For the model to be useful at all in view of physical perturbations, there must be *something* that is insensitive to perturbations. But in this case, the decision of whether or not the numerical method is stable and the computed solution is useful really depends on the modeling context of the problem. This is in some sense outside the purview of numerics, except insofar as numerics can provide an explicit residual that can be explicitly interpreted in the physical terms of the model.

For the Lorenz system, which is already a truncation of a much more complex fluid model, physical perturbations are important, and the detailed trajectory is very sensitive to perturbations. Yet the picture of the Lorenz attractor is itself stable: It is immediately recognizable, and its dimension (for example) is extremely insensitive to perturbations of the system. This explains why and in what sense the results of our simulations of chaotic systems agree with experiments.

To sum up, if we concern ourselves primarily with (for instance) probability measure on the attractor in phase space, the computed trajectories of chaotic systems can be satisfactory in the backward sense (if computed with a backward-stable method), because such measures are usually quite robustly stable under perturbations of the model. However, if one is asking whether the computed trajectories are satisfactory in the forward sense, the answer is that they are not, because of the ill-conditioning of the problem. If you really need the reference trajectory, then you need to specify the initial condition with exponential accuracy and do the integration with an exponential amount of precision. If you do not know the initial condition well but you insist on knowing *some* reference solution that corresponds to *some* initial condition, things are easier: You can use so-called shadowing techniques as discussed in the preface to this part of the book. Shadowing techniques are more expensive than residual control or residual assessment, but they do gain you the assurance of having found an accurate representation of *some* reference solution. We do not discuss them further in this book; we believe that they are usually unwarranted.

12.7 Solution of Stiff Problems

Stiffness is an important concept in scientific computation. So far, we have seen that standard methods such as the one implemented in MATLAB's `ode45` can be relied upon to accurately and efficiently solve many problems. Even for chaotic problems, we have seen that such methods can be relied upon to give faithful results, in the backward sense. With stiff problems, however, we are facing a different computational phenomenon that does not fall under the themes treated so far. As Shampine and Gear (1979 p. 1) put it,

> The problems called "stiff" are too important to ignore, and they are too expensive to overpower. They are too important to ignore because they occur in many physically important situations. They are too expensive to overpower because of their size and the inherent difficulty they present to classical methods, no matter how great an improvement in computer capacity becomes available.

In the literature, authors often discuss "stiff problems" and "stiff methods"; this is, however, somewhat misleading, since stiffness is a property that only makes sense when applied to a problem (in its context) and a method taken in combination. Thus, it will be important to keep in mind this accurate, semipragmatic characterization from A. Iserles:

> Stiffness is a confluence of a problem, a method, and a user's expectations.

Even if there is no generally accepted rigorous definition of stiffness, there is a widely shared consensus as to what stiffness involves in practice. In this section, we explain the problem-specific aspect of stiffness, and delay the method-specific aspects to Chap. 13, insofar as this is possible.

When a problem is moderately well-conditioned and we try to solve it with a reasonably tight tolerance, we will usually observe that the step size automatically

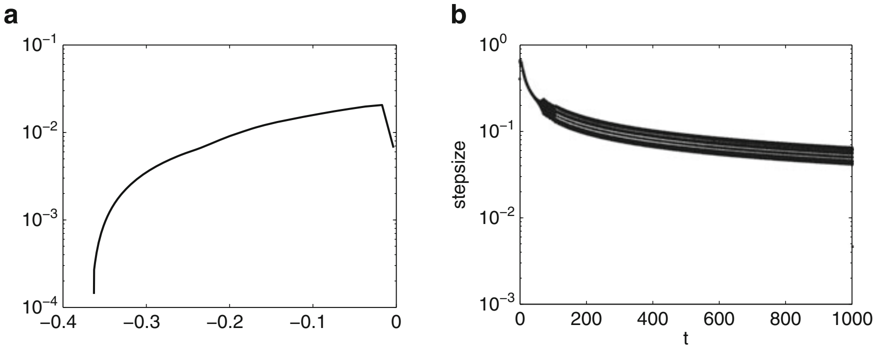


Fig. 12.15 Step-size adaptation of ill-conditioned and well-conditioned problems with respect to step number. (**a**) Reduction of the step size in the solution of (12.29) as the problem becomes increasingly ill-conditioned. (**b**) Reduction of the step size in the solution of (12.44) when the problem is very well-conditioned

selected by the program decreases as t increases. This is because state-of-the-art codes ultimately control the residual of the computed solution by reducing the step size when necessary.

Let us reconsider the example from Eq. (12.29), but let us now turn our attention to the step size as we approach the singular point $-1/e$. As we can see in Fig. 12.15a, `ode45` takes 71 steps to solve the problem. The largest step size, near the beginning, is 2×10^{-2}. The smallest step size, occurring at the very end when the problem is the most ill-conditioned, is 3×10^{-4}. The difference would be even larger if we required a tighter tolerance (or got closer to the singularity). As we see, the amount of work required increases with the condition number.

Now, what would happen if we had a very well-conditioned problem instead? Our *user expectation* would be that the step size would remain large and that we would obtain a cheap solution to the problem. Even if this will often be the case, stiff problems are such that this is precisely what fails. A stiff problem is extremely well-conditioned; loosely speaking, its Jacobian matrix has *no* eigenvalue with large positive real part, but it has *at least one* eigenvalue with large negative real part. Accordingly, it is not an accident that we examine stiff problems immediately after chaotic problems since, in a sense, they are opposite on the same spectrum.[17] Chaotic problems are badly conditioned: Since they have at least one positive real Lyapunov exponent, initially nearby trajectories diverge very quickly from the reference trajectory of the initial-value problem. The difficulty with stiff problems comes from the fact that they are *too well-conditioned*.

[17] But, yes, problems can be both stiff *and* chaotic. The attracting set containing the chaotic attractor can be very well-conditioned, and only the details of the flow on the attractor might be ill-conditioned.

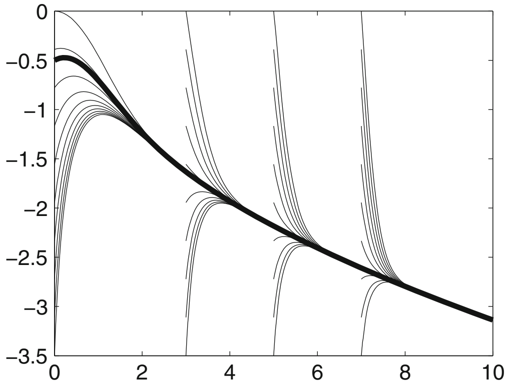


Fig. 12.16 Extreme well-conditioning of a stiff problem

Example 12.11. Consider the following:

$$\dot{x} = x^2 - t, \qquad x(0) = -\frac{1}{2}. \tag{12.44}$$

We will look in detail at two intervals: $I_1 = 0 \leq t \leq 1$ and $I_2 = 0 \leq t \leq 10^3$. The solution of this problem is displayed in Fig. 12.16, together with many other solutions using different initial values. We see that the trajectories all converge to the same one extremely fast. Our expectation would thus be that our program would find the numerical solution without being too strict about step size, since even relatively large errors on each step would quickly be damped. In fact, for the first time span I_1, `ode45` takes 11 steps only and has residual $\leq 8 \cdot 10^{-7}$ (see Fig. 12.17a). This agrees with our expectations: `ode45` does just fine. However, as we see in Fig. 12.15b, the step sizes required to go over I_2 remain very small, the minimum one being 3×10^{-4}. For this interval, `ode45` takes 12,718 steps and the residual is huge (see Fig. 12.17b, which only shows the beginning of it). This goes against what we would expect, since as t increases, the problem becomes increasingly better conditioned. On the other hand, on the same problem for the interval I_2, `ode15s` takes only 57 steps! The residual (not shown here) is everywhere less than about $10^{-2}|x|$, which is quite satisfactory for the low tolerance used. If we examine the step sizes (see Fig. 12.18), we see that as t increases and the problem becomes better conditioned, `ode15s` takes bigger and bigger steps; in fact, six of the last seven steps have $h = 100$. ◁

This situation is not unique to this problem. In practice, a common diagnostic tool to identify stiff problems consists of trying to solve the problem with a standard nonstiff ODE solver, for instance, `ode45`. If the nonstiff solver has a hard time,

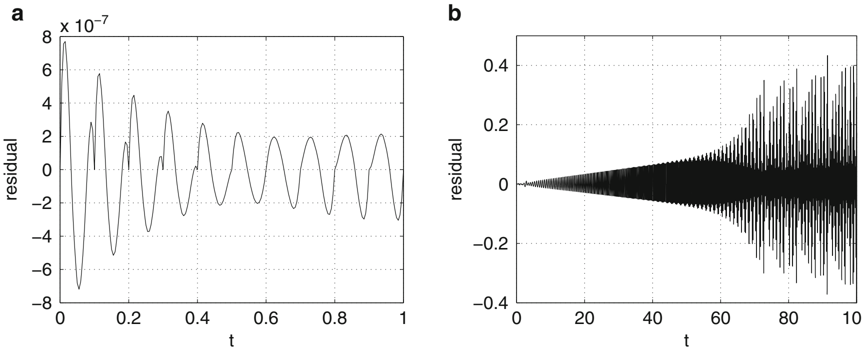


Fig. 12.17 Residual for the solution of (12.44) on two intervals using `ode45`. This method finds the problem harder as t increases. On I_2 instead of $^{I_2}/_{10}$, the graph of the residual looks like a solid triangle, reaching as large as 5 by $t = 1000$. Even by $t = 100$, as here, the size of the residual is obviously too large. (**a**) Absolute residual on I_1, 11 steps. (**b**) Absolute residual on $^{I_2}/_{10}$, 409 steps

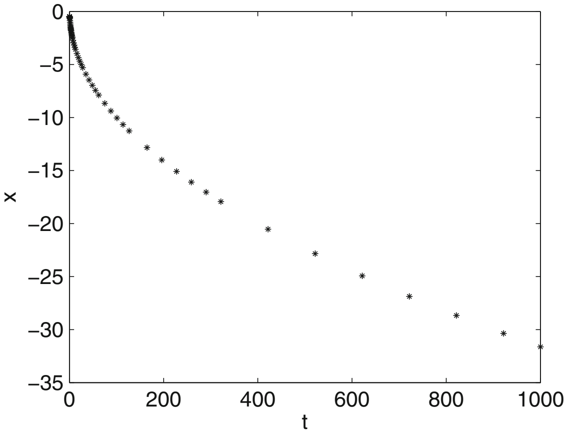


Fig. 12.18 The skeleton of the `ode15s` solution to Eq. (12.44). Note that the step size increases with respect to t, using `ode15s`. This method finds the problem easier as t increases. The residual is everywhere less than about $10^{-2}|x|$

try solving instead with `ode15s`. This test brings some authors to define stiffness pragmatically as follows:

> A problem is "stiff" if, in comparison, `ode15s` mops the floor with `ode45` on it.[18]

In fact, as we will see in Chap. 13, when a problem is stiff, the so-called implicit methods (such as `ode15s`) will mop the floor with the so-called explicit methods (such as `ode45`). This is because, when the problem is very well-conditioned,

[18] This is from L. F. Shampine.

explicit methods become "unstable"[19] for large step sizes. In this spirit, Higham and Trefethen (1993) claim that

> It is generally agreed that the essence of stiffness is a simple idea,
>
> *Stability is more of a constraint than accuracy.*

More precisely, an initial-value problem is stiff (for a given interval of very good condition) whenever the step size required to maintain the stability of the method is much smaller than the step size required to maintain a small residual. However, to explain this statement in more detail, we will need to look at the method-specific aspects of the numerical solutions of IVPs for ODEs. See Chap. 13.

12.8 Event Location

Many applications of initial-value problems for differential equations involve the *location of events*. This means that we integrate the differential equation from the given starting conditions, not for a time interval fixed before we begin, but rather until the solution or its derivative reaches a certain state, if it ever does. Examples include the trajectory of a ball falling under gravity—the differential equation $y'' = -g$ will hold for the duration of the fall, but will require restarting should the ball hit the floor, say $y = 0$. Another application is the location of the maximum height of a ball thrown, which occurs when $y' = 0$. If there is no air resistance, then, of course, this can be done analytically. Air resistance, however, makes the problem more interesting.

Example 12.12. In a beautiful study (though sadly unpublished), G. V. Parkinson started with the following equations for the trajectory of the motion of a ball in the air, which were first used in Bearman and Harvey (1976):

$$\begin{aligned}\ddot{x} &= -\frac{\rho S}{2m}\left(\dot{x}^2 + \dot{y}^2\right)(C_D\cos\alpha + C_L\sin\alpha)\\ \ddot{y} &= \frac{\rho S}{2m}\left(\dot{x}^2 + \dot{y}^2\right)(C_D\cos\alpha - C_L\sin\alpha) - g.\end{aligned} \tag{12.45}$$

These equations are approximate models of the flight path of a spinning golf ball or baseball of mass m and cross-sectional area $S = \pi d^2/4$ through a fluid of density ρ. The fluid-mechanical effects are assumed to be captured in the drag and lift coefficients, C_D and C_L. The trajectory is assumed two-dimensional here, that is, normal to the axis of spin, which is not really a bad assumption. Parkinson went on to make further approximations to this model, enough to allow an analytical solution: He took the drag coefficient C_D to be constant, and the lift coefficient to be approximately

[19] The stability of an IVP method is an important technical concept that is not (quite) the same as the generic sense of the word "stability" of a numerical method as we have been using so far, and which we have not yet discussed. It will form an important part of the discussion in Chap. 13.

$$C_L = p\sqrt{\frac{\omega d}{2V}}\,. \tag{12.46}$$

This assumption about C_L is particularly important: It means that an essential characteristic of the fluid mechanics and its interaction with the rough (either dimpled or seamed) surface of the ball, namely, the lift, can be measured empirically on a spinning ball held otherwise steady in a wind tunnel, and summarized by the single semiempirical coefficient p. Note that $V^2 = \dot{x}^2 + \dot{y}^2$ (assuming the ball is flying through still air). Here ω is the angular velocity of the ball's spin, which was assumed to be approximately constant throughout the flight of the ball.

He also assumed that the angle of attack α, that is, $\tan\alpha = \dot{y}/\dot{x}$, was "small," small enough to make the approximations $\cos\alpha = 1$ and $\sin\alpha = \tan\alpha = \dot{y}/\dot{x}$, together with $\dot{y}^2 \ll \dot{x}^2$. You will be asked in Problem 12.26 to solve the differential equations without this assumption, so that $\cos\alpha = \dot{x}/v$ and $\sin\alpha = \dot{y}/v$.

Finally, he noted that the lift was much less than the drag (around a spherical body this is certainly true) and so took $|C_L\dot{y}| \ll |C_D\dot{x}|$. By letting

$$\begin{aligned} k &= \frac{\rho S C_D}{2m} \\ q &= \frac{\rho S p}{2\sqrt{2}m}\sqrt{\omega d}\,, \end{aligned} \tag{12.47}$$

he arrived at the pair of ordinary differential equations for the velocities $u = \dot{x}$ and $v = \dot{y}$

$$\begin{aligned} \frac{du}{dt} &= -ku^2 \\ \frac{dv}{dt} &= -kuv + qu^{3/2} - g\,, \end{aligned} \tag{12.48}$$

which he then integrated by hand to get

$$u(t) = \frac{u_0}{1 + ku_0 t} \tag{12.49}$$

$$v(t) = \frac{(v_0 - gt - gku_0t^2/2)}{1 + ku_0 t} + \frac{2q\sqrt{u_0}}{k}\left\{\frac{1}{\sqrt{1 + ku_0 t}} - \frac{1}{1 + ku_0 t}\right\}. \tag{12.50}$$

Put $x(0) = y(0) = 0$. Each of these can again be integrated by hand to get

$$\begin{aligned} x(t) &= \frac{\ln(1 + ku_0 t)}{k} \\ y(t) &= \frac{g}{4k^2u_0^2}\left(2\left(1 + 2\frac{v_0\,ku_0}{g}\right)\ln(ktu_0 + 1) - (ktu_0 + 1)^2 + 1\right) \\ &\quad + 4\frac{q\sqrt{u_0}}{k^2}\left(\sqrt{ktu_0 + 1} - \frac{1}{2}\ln(ktu_0 + 1) - 1\right). \end{aligned} \tag{12.51}$$

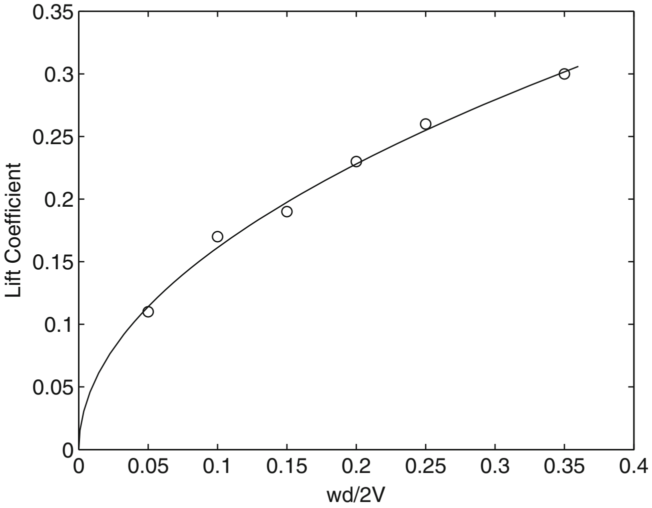


Fig. 12.19 Experimental fit of empirical data relating lift coefficient C_L with angular speed ω, diameter d, and velocity V by $C_L = p\sqrt{\omega d/2V}$. These data give $p = 0.51$ approximately

This gives a complete analytical solution to the problem, under the assumption that ω and thus q is constant.

Taking the diameter d to be 1.68 inches (for an American golf ball—the British golf balls are slightly smaller, at 1.62 inches in diameter), with a weight of 1.62 American ounces, an initial angle of 10 degrees (corresponding to the lift angle of the club typically used to hit a golf ball off a tee), and assuming an initial velocity of $V_0 = 130$ miles per hour with an angular speed of about 600rpm, and a drag coefficient $C_D = 0.3$, all that is needed is a numerical estimate of p. From an old-school plot constructed again by hand by Parkinson, we read the data and fit the parabola and display the results in in Fig. 12.19. We find that $p = 0.51$ fits the data well enough.

We now come (at last) to the question of *events*. We are interested in the event $y = 0$, although not the first one where the ball is hit and lifts off. We want to know how far the ball travels—that is, $x(T)$, where T is the time it takes for y to hit the ground. In a real simulation, one would want to know also how far the ball bounced again, but that depends on the characteristics of the ground it hits, of course (discounting sand and water traps, or the bogs one might find occasionally), and because the spin would change unpredictably, the subsequent flight then depends much less on the aerodynamics in any case: A simple ballistic trajectory might do nicely for its simulation. After all that analytical work, this boils down to solving the nonlinear Eq. (12.51) by a numerical method, such as Newton's method.

We pause to let that sink in for a minute. All that analytical work, and then we have to do some numerics anyway. Parkinson was reasonably happy with the result, because the solution of the nonlinear equation could be done easily on a calculator. But there is no analytical formula for the time t^* at which the ball hits the ground again. Even further, if we choose to *change the model* by allowing (say) the rate of spin ω to decay over time, as is surely physical (a typical flight of a golf ball takes a few seconds, and the air resistance experienced by a spinning golf ball in flight must slow it down at least a little), then the analytical solution of the differential equations no longer obtains.

But a purely numerical solution would be trivial to adapt to the new circumstances. Further, using the idea of the residual, we see that an additional term of the form $\varepsilon\Delta_x(t)$ and $\varepsilon\Delta_y(t)$ to the original pair of second-order equations would be very easy to interpret as fluctuations in the still air that was assumed for the model. So a numerical solution would *be* an exact solution of a similar model. Hence, finding events (such as the point of maximum height, or finding the time at which the ball strikes the ground again) numerically makes perfect sense.

In Exercise 12.26, you are asked to modify `ballode` of `odeexamples` in MATLAB in order to simulate a single trajectory, from being struck to first impact, of the flight of a ball modeled by these equations. ◁

In the history of numerical methods for the solution of initial-value problems, techniques for the location of events are discussed early on. Indeed, the need for locating events was one of the original motivators for the development of interpolatory schemes, because an event would almost always occur between time steps, and it turns out to be important to locate the very first event. That is, we are looking for a t^* that makes some function

$$g(t, y(t), \dot{y}(t)) = 0\,, \tag{12.52}$$

and moreover looking for the first such t^* in the interval $t_n < t^* \le t_{n+1}$. This is a scalar equation involving the unknown vector $y(t)$, but of course one may be looking for any of several events at once. The function g might itself be highly nonlinear—this is not an easy problem.

What is done in practice is to use polynomial interpolants, typically of fairly low degree (after all, the time step is presumed to be small if the solution is varying rapidly), and use polynomial rootfinders. In Corless et al. (2008), barycentric Hermite interpolation such as discussed in this book, and rootfinding by the generalized eigenvalue problem as discussed in this book, are advocated as a particular method to apply polynomial interpolants and rootfinding for just this problem. However, that is not what is done at present in MATLAB. Instead, the local monomial basis used internally by the codes `ode45` and similar are used instead.

Example 12.13. We dig into the details for a bit, to clarify that last paragraph. Suppose that we are solving a differential equation, say $y'(x) = \cos(\pi x y(x))$, with initial condition $y(0) = 1$. Suppose also that we want to locate the very first x value for which $y'(x) = 0$, where we will find a local maximum (or minimum—we'll have to check). This is, of course, equivalent to $\pi xy = \pi/2$, or $y = 1/(2x)$. Suppose we have solved the problem using (say) `ode45`, up until the current step, and have not yet located a place where $y'(x) = 0$, but that we suspect that it will happen on this current step. All codes, of course, must be "paranoid"—they have to check *every* step to see if the events happen. The code ode45 comes equipped with an interpolant (of degree 4, although the details don't matter), and thus what we will do is see if the polynomial interpolant satisfies $p'(x) = 0$ anywhere in the step; alternatively, we could be looking for places where the interpolant took the value $p(x) = {}^{1}/_{(2x)}$. The solution of this equation—which is a polynomial rootfinding problem—will provide an approximate location of the event.

```
function [T,Y,TE,YE,IE] = wavevent

    function [val, ist, dir] = crest( x, y )
        val = y - 0.5/x;
        ist = 1;
        dir = 0;
    end
    f = @(x,y) cos(pi*x*y);
    opts = odeset ('Events', @crest );
    [T,Y,TE,YE,IE] = ode45( f, [0, 5], 1, opts );

end
```

Executing the above code finds an event location at $x = 0.3972$. ◁

Example 12.14. We end this section with a well-known chaotic example, the Hénon–Heiles model. This set of model equations was originally motivated by the motion of stars in a galaxy; however, the gravitational potential was chosen for simplicity, not detailed realism. It is now used as an interesting example in several texts, for example, Bender and Orszag (1978 p. 187) and Hairer et al. (2006). We use the initial conditions given in Channell and Scovel (1990) at first: $p_1(0) = p_2(0) = q_1(0) = q_2(0) = 0.12$. The equations of motion are

$$
\begin{aligned}
\dot{p}_1 &= -q_1 - 2q_1q_2 \\
\dot{p}_2 &= -q_2 - q_1^2 + q_2^2 \\
\dot{q}_1 &= p_1 \\
\dot{q}_2 &= p_2,
\end{aligned}
\tag{12.53}
$$

and we take a long(ish) time span of integration: $0 \le t \le 10^5$. Since this system is four-dimensional, it is awkward to present the results visually and we use a common device in the theory of dynamical systems, namely, the *first return map* or Poincaré map. That is, we take a two-dimensional section in the four-dimensional phase space; it turns out to be usual to take the section $q_1(t) = 0$, and to plot only $q_2(t)$ against $p_2(t)$ since the fourth quantity $p_1(t)$ can be recovered from the total energy of the system (which is conserved, in theory) and the other three coordinates. We only plot $p_2(t)$ against $q_2(t)$ at those points where $q_1(t) = 0$ and $q_1(t)$ is *increasing*; in the case of periodic orbits, this only plots a point when the orbit passes through the section in the positive direction.

This is easy to do with the event location facility of the MATLAB ode solvers, as shown below. The following function uses ode113 at very tight tolerances to solve the system; for the initial conditions given in Channell and Scovel (1990), this tight a tolerance is not necessary, but if we take instead initial conditions twice as large (which correspond to a higher energy level), we do need these tight tolerances to have any confidence in the results. The routine used is this:

```
function [ t, y, te, ye, E ] = Henon( y0, tf )
%HENON  Integration of the Henon-Heiles model locating q[1] = 0
%   solve the model on 0 \le t \le tf, locating all places q[1]=0

    function dy = Hf( t, y )
        dy = zeros(4,1);
        p = y(1:2);
        q = y(3:4);
        dy(1:2) = -[q(1) + 2*q(1)*q(2); q(2) + q(1)^2-q(2)^2 ];
        dy(3:4) = [p(1); p(2)];
    end

    function [z, isterm, dir] = Poincare(t,y)
        z = y(3);
        isterm = 0;
        dir = 1;
    end
    % Tell solver about events function, and make tolerances
        TIGHT.
    opts = odeset('Events', @Poincare, ...
                  'Reltol', 100*eps, 'Abstol', 1.0e-14 );

    [ t, y, te, ye ] = ode113( @Hf, [0,tf], y0, opts );
    E = (y(:,1).^2+y(:,2).^2+y(:,3).^2+y(:,4).^2)/2 ...
        + y(:,3).^2.*y(:,4)-y(:,4).^3/3;

end
```

This routine was called with the following commands:

```
st = clock; [t,y,te,ye,E] = Henon( 2*[0.12, 0.12,0.12, 0.12], 1.0
    e5 ); et = etime(clock,st);
```

This returns the clock time `et` $4.9979 \cdot 10^2$, i.e. about 500 seconds of computation, on the tablet PC used. We will discuss this example further in Sect. 13.8. The Poincaré sections for the two cases are shown in Figs. 12.20 and 12.21. They have been generated with the following commands (the first uses the results from above):

```
figure(3),plot( ye(:,2), ye(:,4), 'k.','Markersize', 3 ),axis('
    square'),set(gca,'Fontsize',16),xlabel('p_2'),ylabel('q_2')
st = clock; [t,y,te,ye,E] = Henon( [0.12, 0.12,0.12, 0.12], 1.0e5
     ); et = etime(clock,st);
figure(4),plot( ye(:,2), ye(:,4), 'k.','Markersize', 3 ),axis('
    square'),set(gca,'Fontsize',16),xlabel('p_2'),ylabel('q_2')
```

We do not plot the residuals here, but in both cases they are small and `ode113` has in each case produced the exact solution to a time-dependent but tiny perturbation of this (Hamiltonian) system of equations. Over the long term, such a perturbation will have a large effect; but then, so will physical perturbations and errors and omissions in the model equations. ◁

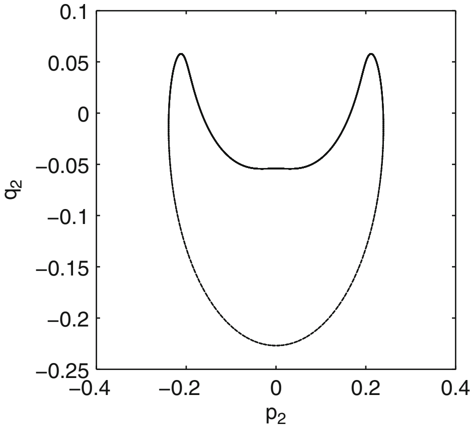


Fig. 12.20 Solving the Hénon–Heiles equation for the same initial conditions as used in Channell and Scovel (1990), namely, $p_1(0) = p_2(0) = q_1(0) = q_2(0) = 0.12$ with an energy $E = 0.02995$, gives an apparently smooth curve in the Poincaré section. Event location is used to identify the times and values of p and q when $q_1 = 0$ which are needed to draw the Poincaré section. This smooth-looking resulting curve filled out by the return map corresponds to a quasiperiodic orbit on a 3-torus. Note that the figure here, unlike that in Channell and Scovel (1990), plots only those points for which $p_1(t) > 0$. Computation took just over 6 minutes at these tight tolerances (which were doubtless unnecessarily tight)

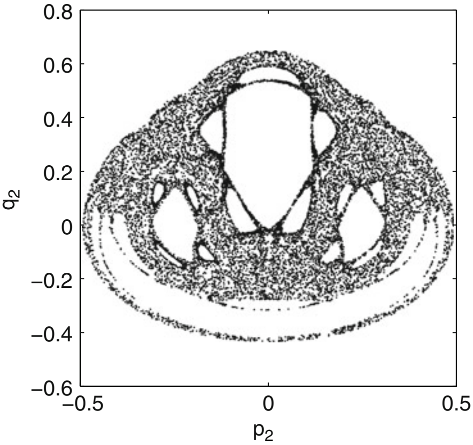


Fig. 12.21 The first-return map (Poincaré map) for the Hénon–Heiles equation with energy $E = 0.1244$ and initial conditions $p_1(0) = p_2(0) = q_1(0) = q_2(0) = 0.24$. The map shows chaotic behavior outside calm "islands"

12.9 More on Mass Matrices: DAE and Singular Perturbation Problems

Differential-algebraic equations (DAE) occur frequently in practice and in some sense can be thought of as being infinitely stiff. Mass matrices, as occurred in Sect. 12.5, often arise in the natural formulation of DAE, although in this case the rank of the matrix is generally not full. This requires initial conditions to be consistent and also requires that certain constraints not be violated. DAE are always harder to solve than IVP are, and some DAE are even harder than most. This is usually described by means of an "index" (there are several flavors of index to choose from, but higher-index problems are harder than index-1 problems).

Example 12.15. As a beginning example, consider the singular perturbation problem

$$\begin{aligned} \dot{x} &= -xy^2 \\ \varepsilon\dot{y} &= 1 - x^2 - y^2 \,. \end{aligned} \tag{12.54}$$

For very small ε, we expect the solution of this problem to look like the solution of the DAE that is obtained when we substitute $\varepsilon = 0$ into the equations, which forces $x(t)$ and $y(t)$ to lie on the unit circle. In this simple example, we can see directly what happens in the DAE case: We can analytically solve the second equation for y^2 and, knowing this, the first equation reduces to the ordinary differential equation $\dot{x} = -x(1-x^2)$. In larger DAE, finding such an explicit solution to the algebraic constraint(s) is almost always impossible, and in essence what one wants to do is solve the algebraic part of the equation numerically. More, one wants to have this done automatically as part of the numerical solution of the DAE itself. It is almost always true that the constraint needs to be satisfied to a much higher accuracy than the differential equation part needs to; in other words, the residual in the constraint needs to be *much* smaller than the residual in the differential equation part. The reason for this is that the *conditioning* of DAE is usually such that the forward error is highly sensitive to errors in the algebraic parts of the equation. We do not discuss conditioning of DAE further in this book.

For this particular DAE, we can see (by having reduced it explicitly to an ODE) that there are equilibria at $x = 0$ and at $x = \pm 1$. We can also see, by multiplying the ODE by x to get $^{d}/_{dt}(^{x^2}/_{2}) = -x^2y^2 \leq 0$, that the magnitude of the solutions are nonincreasing. Finally, the solutions are symmetric about the t-axis because the equation is invariant under $x \to -x$. Any good solution of this problem ought to reproduce those qualitative features.

The following function uses `ode15s` to solve this DAE. The DAE is encoded as a singular perturbation problem, with a possible parameter ε (called `mu` in the code to avoid confusion with ε_M):

```
function [t, y, res] = singpertex1( mu, u0 )

%
% A singular perturbation problem
```

```
%
% Leave the 'MassSingular' property at its default 'maybe' to
    test the
% automatic detection of a DAE.

    function dy = f( t, y )
        dy = [ -y(1)*y(2)
               1 - y(1)^2 - y(2)^2 ];
    end

M = [ 1, 0
      0, mu ];

options = odeset('Mass',M);

sol = ode15s(@f,[0 5],u0,options);

t = RefineMesh( sol.x, 5 );
[y, dy ] = deval( sol, t );
np = length( t );
res = zeros(size(dy));
for i=1:np,
    res(:,i) = dy(:,i) - f(t(i), y(:,i));
end

end
```

Just with the default tolerances, the computed solution satisfies the constraint to within 2.5×10^{-4} (when $\mu = 0$) and has a satisfactorily small residual. ◁

Remark 12.3. Many DAE can be naturally embedded into singular perturbation problems; that is,

$$\begin{aligned} \dot{x}(t) &= f(t,x(t),y(t)) \\ \varepsilon\dot{y}(t) &= g(t,x(t),y(t)) \,. \end{aligned} \tag{12.55}$$

One might even start with the $\varepsilon = 0$ case and introduce the $\varepsilon\dot{y}(t)$ term, out of the blue, just so as to use a stiff IVP solver on the problem if one doesn't have a DAE-capable code handy. There is a duty of care here, to make sure that the constraint is *stable*. In this example, the constraint is stable if $\varepsilon > 0$ (but small). If instead you take $\varepsilon < 0$, then the constraint is unstable, and the solution of the IVP doesn't look at all like the solution of the DAE. See Problem 12.16. ◁

12.10 Which Interpolant, Which Residual?

Up until now, we have simply used the polynomial interpolants made available by the codes. This has the benefit of convenience, but in many cases (especially outside of MATLAB), the interpolants used may have been chosen for other reasons than accuracy. In that case, the computation of the residual will be "suboptimal," in that we could do better.[20]

The first way in which we might do better is to try to use what are called *shape-preserving* interpolants. The idea is to use a rational interpolant that contains a parameter that can be tuned so as to preserve qualitative features such as convexity of the solution or monotonicity of the solution. This can also be handled quite easily within the framework of barycentric interpolants, as discussed in Sect. 8.7. Other possibilities also come to mind. One might try to find the "best possible" interpolant. For example, one might try to find an interpolant that minimized $\Delta(t)$ in some manner. A natural thing to try to do would be to minimize, over each step, the function 2-norm of $\Delta(t)$. Specifically, on a subinterval $t_k \le t \le t_{k+1}$, this means trying to find an interpolant, call it $\mathbf{z}(t)$, satisfying $\mathbf{z}(t_k) = \mathbf{x}_k$ and $\mathbf{z}(t_{k+1}) = \mathbf{x}_{k+1}$ and at the same time minimizing

$$\|\Delta\|_2^2 = \int_{t_k}^{t_{k+1}} \Delta^H(\tau)\Delta(\tau)\,d\tau\,, \tag{12.56}$$

where $\Delta(\tau) = \dot{\mathbf{z}}(\tau) - \mathbf{f}(\tau,\mathbf{z}(\tau))$. This is a classical problem in the calculus of variations and leads to a two-point boundary value problem (BVP) by means of what are called the Euler–Lagrange equations. In this case, the equations work out to be

$$\begin{aligned} \dot{\mathbf{z}}(t) - \mathbf{f}(t,\mathbf{z}(t)) &= \Delta(t) \\ \dot{\Delta}(t) + \mathbf{J}_f^H(\mathbf{z})\Delta(t) &= 0\,. \end{aligned} \tag{12.57}$$

We might call the solution of this BVP (if it exists) the "optimal" interpolant and the Δ that results the "optimal" residual. You will be asked to pursue this idea in the exercises, once you have learned how to solve boundary value problems in Chap. 14. Of course, it will be useful theoretically before that if one can solve the resulting BVP analytically. See Problem 13.36.

12.11 Singularity

Consider the harmless-looking problem

$$\dot{x}(t) = t^2 + x^2 \tag{12.58}$$

on the interval $0 \le t \le 1$, subject to the initial condition $x(0) = \alpha$ for various α. When we attempt to solve this in MATLAB as in the script below, we encounter a surprise:

[20] In fact, the interpolant used in `ode45` seems to be not quite as good as we would want: The solution at the mesh points is roughly speaking $O(h^5)$ accurate, but the interpolant seems to be only $O(h^4)$ accurate at off-mesh points. This is good enough for many purposes, but for us it seems to overestimate the residual.

```
% An example with a moveable pole
f = @(t,x) t.^2 + x.^2;
singsol = ode45( f, [0,1], 0.5 );
%singsol = ode45( f, [0,1], 0 );
figure(1), plot( singsol.x, singsol.y, 'k.' ) % No trouble seen

singsol = ode45( f, [0,1], 1.0 ); % Initial condition important!
```

MATLAB complains that it can't complete the integration![21] The difficulty is, of course, that the solution of this problem contains a *movable pole*. That is, the location of the pole depends on the initial condition. The convergence of numerical methods to a solution needs regularity of the solution; having a Lipschitz constant guarantees solvability, albeit only locally. Here $f(t,x) = t^2 + x^2$ and so $f(t,x) - f(t,y) = x^2 - y^2 = (x+y)(x-y)$, which means that so long as the solution is finite and bounded there will be a Lipschitz constant (and vice versa)—but obviously, any possible constant would have to grow without bound since the solution does.[22]

In practice, this is reflected in the mesh size control: If the solution goes singular, the mesh width must go to zero. Indeed, the correlation is so strong that until recently, when the MATLAB routines discovered the step size getting so small that $t_n + \Delta t$ rounded to t_n (so no possible progress could be made), they declared that there was a "singularity likely." Nowadays they are more conservative and say merely that they are "unable to continue."

However, `ode45` does a pretty good job of locating the singularity in this example, at least! Exact solution of the problem in MAPLE gives a somewhat complicated (but not too bad) rational function of Y and J (i.e., Bessel functions); we can then isolate the denominator and (for a given α) numerically solve for the location of the singularity.[23] In contrast, with its default tolerances, `ode45` located the singularity accurately to four decimal places for $\alpha = 2$.

The reason it works is that the underlying theory, that of analytic continuation, allows one to locate a nearby singularity from the coefficients of the Taylor series of the solution, using a method originally due to Daniel Bernoulli and formalized by Darboux: The ratio of successive Taylor coefficients is related to the location of the nearest singularity (this is why the ratio test of elementary calculus works as well). All of the numerical methods discussed here are (as we will see in Chap. 13) based on analytic continuation, and the step-size control can be related to the convergence or lack thereof of local Taylor series.

[21] Afterwards, we could pretend it wasn't a surprise after all, by noting that solution of the related but easily solved problems $\dot{x} = x^2$ and $\dot{x} = 1 + x^2$, which trap the solution of our "harmless" problem between, are themselves both singular: ${}^{\alpha}/_{(1-\alpha t)}$ in the first case and $\tan(t + \arctan(\alpha))$ in the second. Therefore, our apparently harmless problem is *of course* not harmless, and everyone ought to have known that straight away.

[22] Obviously, we mean that on a fixed interval $[0,T]$. Short of the singularity, there could be a constant L, but that as T approached the singularity we would have to take L larger and larger.

[23] This of course requires accurate methods to evaluate the Bessel functions, but these are available and good to have anyway.

The important point is this: *Singularities* (even singularities in higher derivatives, which appear smooth to the human eye) *cause difficulties for numerical methods*. One reason for this is that the accuracy of many underlying numerical formulae is proportional to bounds for higher derivatives; and if a problem is singular, these can be infinite. Even just approaching a singularity can mean that these bounds get quite large. We examine below some strategies to handle singular problems.

12.11.1 Pole-Vaulting

If one encounters a singularity, then one often wants to somehow get past the singularity. A lovely idea is to replace the real variable of integration (here thought of as "time" t) by a path in the complex t-plane. This is called, amusingly, "pole-vaulting."

For our example problem, we do the simplest possible thing: We back off slightly from the pole, and then go in a semicircular arc in the complex t-plane. Let $p = 0.4989$ be our approximate location of the pole. We look at the solution generated thus far and find that, at the 10th mesh point, `singsol2.x(10)` (which is about 0.4907, not so far from the pole), the value of the solution is larger than 100. We choose a semicircular path in the t-plane: $t = p + \rho\exp(i\theta)$, with $\rho = (0.4989 - 0.4907) = 0.0083$. An interesting subtlety is whether we should hop *over* the pole by taking $\pi \geq \theta \geq 0$, or duck under the pole and take $-\pi \leq \theta \leq 0$. If the residue at the pole is nonzero, the answers will be different; we will be on different branches of the solution. This will certainly matter in some applications! Here we choose to go over the pole.

The differential equation in the new variable is

$$\frac{dx}{d\theta} = \frac{dt}{d\theta}\frac{dx}{dt} = (i\rho e^{i\theta})\big((p+\rho e^{i\theta})^2 + x^2\big)\,, \tag{12.59}$$

and this is perfectly simple to solve using `ode45`, because that code allows complex-valued solutions. Integrating this from $\theta = \pi$ down to $\theta = 0$, we find that the solution is $-1.2095 \times 10^2 - 3.9379 \times 10^{-3}i$, and of course that near-zero imaginary part tells us roughly how accurate our solution is (this is not too inconsistent with the default tolerances of 10^{-6}). We then drop the imaginary part and use the real part as the initial condition to restart the integration on the next interval, again using `ode45`. A new singularity is reported further on, near 2.518. See Fig. 12.22. In Problem 12.13, you are asked to follow this idea.

12.11.2 Other Kinds of Singularities

The other kinds of singularities that you will have to worry about, from time to time, include singularities or nondifferentiabilities explicitly in the problem, such as $\dot{x} = {}^{x}\!/\!_{(t-1)}$ or $\dot{x} = \sqrt{x-1}$ or $\dot{x} = |1-x^2|$; one often encounters singularities of

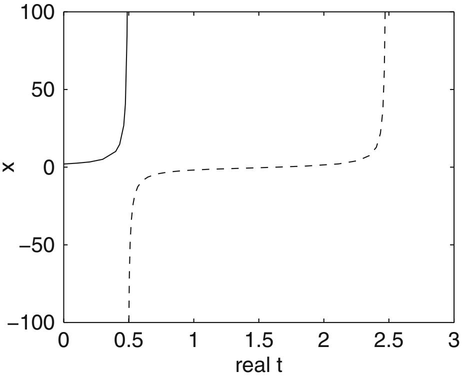


Fig. 12.22 Integrating past a singularity using the technique of "pole-vaulting"

an Euler type in Sturm–Liouville and other problems, where the leading derivative is multiplied by a term that goes to zero: $x^2 {}^{d^2x}\!/_{dt^2} + f(t,x) = 0$, for example. In that case, even getting a numerical method started requires using something extra, a series about the singular point (possibly containing logarithmic or algebraic powers) for example, to find the values of the solution near, but not at, the singular point; from there the solution will be regular and a normal method can then proceed.

There is a large class of problems where the input contains known singularities—the Dirac delta function, or the Heaviside unit step function, for example—and these are important in applications. Because numerical methods can have a hard time locating such singularities, it is important to tell the method you are using about any such singularities, so it can restart the problem there. This essentially breaks the singular problem up into a sequence of nonsingular problems, each of which is easier for a numerical method. You will also have to worry about infinite intervals, which are a kind of singularity. Numerical methods can be surprisingly good in this case if the solutions tend to equilibria. A good method can even take an infinite-length time step!

But there are worse things to worry about as well. Moveable essential singularities are true headaches (see the next example); and natural boundaries (moveable or not) prevent analytic continuation of any kind. Since all numerical methods for initial-value problems that we have been discussing here are based on analytic continuation, this effectively limits further application of methods such as these.

Example 12.16. Consider the differential equation

$$\frac{d^2y}{dx^2} = \frac{\left(1 - {}^3\!/_2(ax-1)^2\right)\left(\frac{dy}{dx}\right)^2}{y(x)} \tag{12.60}$$

subject to the initial conditions $y(0) = {}^1\!/_e$ and $y'(0) = {}^{-2a}\!/_e$. This isn't quite in the category we have been discussing, because a appears both in the equation and in the initial condition. However, the reference solution, $y(x) = \exp(-{}^1\!/_{(1-ax)^2})$, has an essential singularity at $x = {}^1\!/_a$; mind you, it's infinitely differentiable along the real axis and all derivatives are zero there. Solving this with `ode45` seems to notice the singularity—the step size gets small as the solver approaches the singularity—but it goes right through and produces the solution displayed in Fig. 12.23. That looks

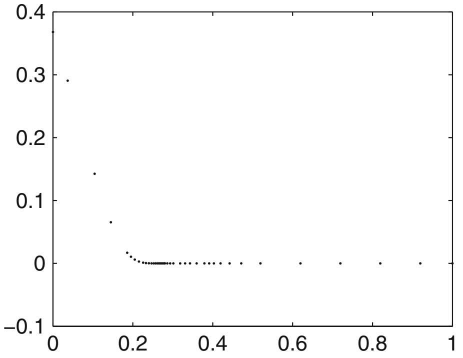


Fig. 12.23 Numerical solution of a problem with an essential singularity, although approached in a direction where the solution is infinitely differentiable at the singularity

perfectly fine, of course, until one realizes that the solution, which is apparently 0 after $1/a$, involves a division by zero in the right-hand side of the differential equation, so it must be not quite zero, possibly even so that $y(x)$ is small but nonzero, but the derivative is so small that its square underflows to zero! Also, the computed solution is very different from the reference solution $\exp(-1/(1-ax))$. Essential singularities are a problem!

Now let us consider a slightly different ODE:

$$\frac{d^2}{dx^2}y\,(x) = \frac{(2\,\pi x - 1)\,({d}/{dx}y\,(x))^2}{y\,(x)} \tag{12.61}$$

subject to $y(0) = 1/e$ and $y'(0) = -a/e$. Choose $a = \pi$ this time. Now we get

```
Warning: Failure at t=3.394010e-001.  Unable to meet integration
    tolerances without reducing the step size below the smallest
    value allowed (8.881784e-016) at time t.
```

This is in some sense acceptable, in that the reference solution $y(x) = \exp(-1/(1-ax))$ is not finite or finitely differentiable to the right of $x = 1/a$. However, the point $1/\pi = 0.318309886\ldots$ was not located very well—the integration went *past* this point (by about 0.02). One sees the potential difficulty: Stepping past a singularity by so much risks missing it altogether. Indeed, for this example, had we changed the differential equation for $t > 0.32$ to something that was smoother, the numerics would have missed the singularity (at these tolerances). ◁

Finally, here is what has been described as the “simplest ODE whose solution has a movable natural boundary” in Clarkson and Olver (1996), the Chazy equation:

$$\frac{d^3y}{dx^3} = 2y\frac{d^2y}{dx^2} - 3\left(\frac{dy}{dx}\right)^2 . \tag{12.62}$$

See also Ablowitz and Clarkson (1991). When faced with a natural boundary, analytic continuation (and hence all the numerical methods we have used so far) fails.

12.12 Notes and References

The MATLAB ODE Suite is described in Shampine and Reichelt (1997). There are similar codes available in MAPLE via `dsolve` with its `numeric` option described in Shampine and Corless (2000), although we do not use those explicitly in this book.

For methods to determine when certain differential equations can be solved in terms of elementary functions, see, for instance, Geddes et al. (1992); von zur Gathen and Gerhard (2003); Bronstein (2005).

A nice proof of the existence and uniqueness of the solution of IVP for ODE, essentially due to Cauchy originally and which uses the deviation, is given in Birkhoff and Rota (1989 Chap. 6). The Lipschitz constant is always positive, but for some problems (as we will discuss in Chap. 13), there exists a negative bound that allows converging trajectories. This has been studied, for example, by Söderlind (1984). The use of the variational equation to study the condition number of an IVP is explored further in Mattheij and Molenaar (1996); that book is especially useful for its integration of the theory of differential equations with numerical practice. The type of reasoning used in Sect. 12.3.2 is standard in the theory of dynamical systems (see, e.g., Nagle et al. 2000; Lakshmanan and Rajasekar 2003). An excellent presentation, with many examples, is given by Bender and Orszag (1978). For the computation of Lyapunov exponents, see Geist et al. (1990). A recent thorough survey is Skokos (2010). The analytic SVD is studied by Bunse-Gerstner et al. (1991). The reference solution to the Lorenz system is studied with interval methods in Tucker (2002).

The fidelity of numerical solutions to chaotic problems by backward error analysis is discussed in Corless (1992, 1994a,b). Wayne Enright, in a conversation with RMC in the early 1990s, suggested that a reference problem could be called chaotic (even if it wasn’t “really” chaotic in the theoretical sense) if generic nearby problems *were* chaotic (in the theoretical sense). In such a case, in view of physical perturbations and modeling errors, there would be no distinction possible in practice. See also the extensive discussion of backward error for dynamical systems in Moir (2010). Example 12.11 was taken originally from Hubbard and West (1991), although they used it for different purposes, namely, to show how unreliable fixed time-step numerical methods were.

Taylor series methods to solve DAE have been shown to have cost polynomial in the number of bits of residual accuracy, even for higher-index problems; but they are still hard. See Corless and Ilie (2008) and Nedialkov and Pryce (2005, 2007). Nonetheless, MATLAB (and the repository at GAMS) have facilities for solving DAE, at least for index-1 problems. See `odeexamples('dae')` in MATLAB, and see Hairer and Wanner (2002) for an extensive discussion of DAE methods. For an interesting example of high-index problems in chemical process control, see Qin et al. (2012). With regard to the sensitivity or conditioning of DAE, see Cao et al. (2003), Li and Petzold (2004) and Petzold et al. (2006).

Shape-preserving interpolants are studied, for example, in Brankin and Gladwell (1989). Barycentric versions are looked at briefly in Butcher et al. (2011).

Detection of singularities in numerical methods for ODE has a long history. See, for example, Suhartanto and Enright (1992). For pole-vaulting, see Corliss (1980) or Tourigny (1996), or Fornberg and Weideman (2011) for an alternative method that uses Padé approximation to deal with singularities. The PhD thesis by Orendt (2011) may also be of interest.

The problem of *parameter estimation* for nonlinear DE models is much studied and relies on the numerical solution of such problems (although see Ramsay et al. (2007) for an interesting variation).

Problems

Theory and Practice

12.1. Show that the damped harmonic oscillator (12.7) can be written in first-order form as

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} 0 & -\omega \\ \omega & -2\,\zeta\,\omega \end{bmatrix} \mathbf{x}(t)\,. \tag{12.63}$$

Let the initial conditions be $x_1(0) = 1$ and $x_2(0) = 0$. Take $\omega = 1$ and $\zeta = 0.005$, and use `ode45` and `ode113` with default tolerances to solve the problem on $0 \le t \le 200\pi$. Plot your solutions in the phase plane [that is, $x_1(t)$ versus $x_2(t)$]. Plot them again without the interpolants—that is, just plot the discrete values in `sol.x`—and comment on the value or lack thereof of interpolation for graphical interpretation. Compare your results with Fig. IV.I.

12.2. Solve Eq. (12.1) on $0 \le t \le 5$ with the initial condition $x(0) = 0$ using not `ode45` but rather `ode113` and `ode15s`. What differences do you note?

12.3. The first example problem of this chapter, $\dot{x} = t^2 + x - {}^{x^4}/_{10}$, can be embedded in a family of problems with a parameter ε that is assumed small: $\dot{x} = t^2 + x + \varepsilon x^4$, and our particular case is $\varepsilon = -{}^1/_{10}$. Assume that there is a solution of this family of the form

$$x(t) = x_0(t) + \varepsilon x_1(t) + \varepsilon^2 x_2(t) + \cdots \tag{12.64}$$

and use this assumption and the idea of equating like powers of ε in the expansion to find $x_0(t)$ and $x_1(t)$ [you may wish to use a computer algebra system for $x_1(t)$]. Put $z = x_0 + \varepsilon x_1$. Show explicitly that the residual $r(t) := \dot{z} - f(t,z)$ is $O(\varepsilon^2)$. Thus, the ideas of this chapter can be used to understand perturbation methods for solving differential equations, as well. In this particular example, the perturbation solution is really too ugly to be genuinely useful, but in other situations the method can give results that are better than numerics. The *asymptotics* of the solution of this equation as $t \to \infty$ can be studied by the methods of Bender and Orszag (1978), and indeed it is useful to discover that to leading order $x(t) \sim (10t^2)^{1/4}$, which can be confirmed by numerics.

12.4. Consider the numerical solution of the following initial-value problem, first on the interval $0 \le t \le 1$, and then on the interval $-10 \le t \le 0$, but still starting at $t = 0$; that is, in the last case we will be integrating backward in time. To do this for (say) `ode45`, just put the tspan as `[0,-10]`. The initial conditions are $y(0) = 1$ and $y'(0) = -1$, and the differential equation is $y'' + 11y' + 10y = 0$ in both cases, and of course you will have to transform it to a first-order system in order to use `ode45`. The reference solution is in both cases (of course) $y(t) = \exp(-t)$. Comment on your solutions. Note that $\exp(-(-10)) \approx 22{,}000$. What is the residual? Is your second numerical solution a good one? If so, in what sense?

12.5. Suppose we solve a scalar autonomous nonlinear IVP numerically, and get a computed solution with a relative defect $\delta(t)$; that is, $\dot{x}(t) = f(x(t))(1+\delta(t))$ is satisfied exactly by the computed solution $x(t)$. By using separation of variables, argue that the computed solution is the exact solution of the reference problem evaluated at a slightly perturbed time. It would be nice if this worked for autonomous systems, too, but it doesn't—explain why not.

12.6. Express the initial-value problem

$$y^{iv}(t) + a_3 y'''(t) + a_2 y''(t) + a_1 y'(t) + a_0 y(t) = 0$$

in standard form, and argue thereby that the eigenvalues of the Frobenius companion matrix are the roots of the polynomial

$$\lambda^4 + a_3\lambda^3 + a_2\lambda^2 + a_1\lambda + a_0 = 0\,.$$

12.7. This problem can be done numerically, or if manual computation is preferred, then bounds can be used instead. Either way, explicitly compute the condition numbers (or good bounds thereof) of the following initial-value problems:

1. Airy's differential equation $y'' = ty$ with initial values $y(0) = \mathrm{Ai}(0)$, $y'(0) = \mathrm{Ai}'(0)$, on the interval $0 \le t \le 10$;
2. Airy's differential equation $y'' = ty$ with initial values $y(0) = \mathrm{Bi}(0)$, $y'(0) = \mathrm{Bi}'(0)$, on the interval $0 \le t \le 10$;

3. Airy's differential equation $y'' = ty$ with initial values $y(10) = \mathrm{Ai}(10)$, $y'(10) = \mathrm{Ai}'(10)$, on the interval $0 \le t \le 10$, backward;
4. Airy's differential equation $y'' = ty$ with initial values $y(10) = \mathrm{Bi}(10)$, $y'(10) = \mathrm{Bi}'(10)$, on the interval $0 \le t \le 10$, backward.

12.8. Simulate a perfect ball bouncing on an infinite floor, with air resistance and tilt. Use the "events" feature in your simulation.

12.9. Consider the initial-value problem $\dot{x} = |1 - x^2|$, with $x(0) = 0$, apparently originally discussed by K. Nickel. Show that the reference solution to this is $x(t) = \tanh(t)$, which has $-1 < x(t) < 1$, and so the absolute value signs make no difference to the reference solution. Solve the problem on the interval $0 \le t \le 100$ using `ode45` three times, each with different relative tolerances. You should find that the numerical solution goes singular in finite time, but that the location of the singularity depends on the tolerance. Compute the residual in each of the three cases, and show that it is often positive. Now consider the solution of $\dot{x} = |1 - x^2| + \varepsilon$ and argue that eventually this $x(t) > 1$. Consider also the reference solution of the equation $\dot{x} = x^2 - 1$ with the initial condition $x(t_0) = 1 + \delta > 1$ and show that it goes singular in finite time, and that the location of the singularity depends on δ. Now: Has `ode45` done a reasonable job of solving the original problem? Justify your answer. Remark: The answers "yes" and "no" can each receive full credit if the justifications are argued well. This isn't an easy question.

12.10. Use `ode45` or `ode15s` to find the roots of $x^5 + x - 1 = 0$ by *homotopy* or *continuation*, as follows. Let $X(t)$ be the roots of $X(t)^5 + tX(t) - 1 = 0$. Clearly, $X(0)$ is one of the roots of $x^5 - 1 = 0$, or $\exp(2\pi ik/5)$ for $k = 0, \ldots, 4$. We wish to follow the roots of this polynomial as t ranges from $t = 0$ to $t = 1$, when the roots of our original ("hard") problem will be the five values of $X(1)$. By differentiating, we find

$$5X(t)^4\dot{X}(t) + X(t) + t\dot{X}(t) - 0 = 0 \tag{12.65}$$

or

$$\dot{X}(t) = -\frac{X(t)}{5X(t)^4 + t}. \tag{12.66}$$

How accurate are your final answers? Of course, this is only the beginning of the story of continuation methods....

12.11. Use `ode15s` to solve the lead uptake model of Example 12.2 on $0 \le t \le 10^8$. Compute the residuals, and show that the problem gets progressively easier as t increases. On such a long time interval, the problem is "stiff," and the use of a nonstiff solver such as `ode45` is contraindicated. You may solve the problem on a shorter time interval (say $0 \le t \le 1000$; remember that the time is in units of days, and 1000 days is not so far off 3 years) using a nonstiff solver, and perhaps this is a more realistic picture anyway. Discuss.

12.12. Consider the damped harmonic oscillator (12.7). Take $\omega_0 = 1$ and $\zeta = 0$, to begin with. Solve the model with $x(0) = 1$, $\dot{x}(0) = 0$ on the interval $0 \le t \le 10\pi$ using `ode23`. Use the Oettli–Prager theorem to estimate the "numerical damping" the code has introduced into the solution.

12.13. Solve the differential equation $\dot{x}(t) = t^2 + x^2$ with the initial condition $x(0) = 2$ on the interval $0 \le t \le 10$. Use the technique of pole-vaulting to jump over the singularities.

12.14. Now, let us consider a simple potentially stiff differential equation, sometimes called the Prothero–Robinson DE and used as a test for stiff methods: Choose a slowly varying function $g(t)$ and create the IVP

$$\dot{y} = \lambda(y - g(t)) + \dot{g}(t), \qquad y(0) = g(0) + 2\,. \tag{12.67}$$

The true solution is $y(t) = g(t) + 2e^{\lambda t}$. Stiffness occurs if $\mathrm{Re}(\lambda) \ll 0$ and models the situation when the person doing the solving is interested in the behavior of the slowly varying attractor [here the graph of $g(t)$], and not so interested in the transient region $0 \le t \le O(-1/\lambda)$. Thus, tolerances are set rather loosely, meaning that we can accept a fairly large residual. Take $g(t) = \sin(t)$, $y(0) = 2$, $\lambda = -1000$, and solve the problem on $0 \le t \le 10$ using both `ode45` and `ode15s`. Discuss.

12.15. Consider the problem $\dot{x} = -\mathrm{signum}(x)$, with the variant convention that $\mathrm{signum}(0) = 1$. That is, $\dot{x} = -1$ if $x \ge 0$, and $\dot{x} = 1$ if $x < 0$. This problem is apparently originally due to John Butcher. The right-hand side can be implemented in MATLAB with the command `f = @(t,x) -sign(x+realmin)`, near enough. Show analytically that this problem fails to have a solution for $t > |x_0|$, although the machine-implemented problem does have an equilibrium at `-realmin` that doesn't matter since we'd never see it as the result of a computation unless lightning struck the computer or something. Show also that none of `ode45`, `ode23`, or `ode113` finds the singularity where the solution ceases to exist, but that their residuals go from $O(\text{tolerance})$ to $O(1)$ at that point. Show that `ode15s` does find the singularity. Use $x_0 = 5$ or $x_0 = -5$.

12.16. Solve the problem in Example 12.15 for various initial conditions and various μ.

12.17. Consider the differential equation

$$\varepsilon^2 \frac{d^2}{dt^2} y(t) = \left(1 + t^2\right)^2 y(t)\,, \tag{12.68}$$

which is discussed in Bender and Orszag (1978 p. 488), with various initial conditions, including $y(0) = 0$ and $y'(0) = 1$. Solve this equation with these initial conditions on $0 \le t \le 1$ for $\varepsilon = 0.1$, 0.01, and 0.001. MAPLE can apparently solve this analytically in terms of the `HeunT` function, but evaluating the result requires high precision for large t. The asymptotic formula derived in Bender and Orszag (1978) is

$$y(t) = \frac{\varepsilon}{\sqrt{1+t^2}} \sinh\left(\frac{1}{\varepsilon}\left(t + \frac{t^3}{3}\right)\right)\,, \tag{12.69}$$

which shows that the solution grows very rapidly.

12.18. The linear homogeneous second-order differential equation

$$(1-z^2)\frac{d^2y}{dz^2} - z\frac{dy}{dz} + n^2 y = 0 \tag{12.70}$$

has as one of its solutions $y(z) = AT_n(z)$, where $T_n(z)$ is the nth Chebyshev polynomial. One expects, then, that the initial-value problem with $y(0) = T_n(0)$ and $y'(0) = T_n'(0)$ would have as its unique solution on $-1 \le z \le 1$ just $y(z) = T_n(z)$. Is this true, numerically? Even for very large n? What might go wrong? What is the condition number of this differential equation? To make the computation of the condition number easier, if you change variables so that $z = \cos\theta$ with $-1 \le z \le 1$ corresponding to $\pi \ge \theta \ge 0$, then the differential equation becomes just $y'' + n^2 y$.

12.19. Add the lines

```
% Linear interpolation does not look good
skeleton = ode45( @odefun, tspan, y0 );
figure(3), plot( skeleton.y(1,:), skeleton.y(2,:), 'k' )
```

to the code from Example 12.4 and run it. The skeleton of the solution is exactly as accurate as before, but the plot which uses linear interpolation looks misleadingly inaccurate. This is one reason good interpolants are wanted.

12.20. The problem $\dot{y}_1 = -2y_1 + y_2$, $\dot{y}_k = y_{k-1} - 2y_k + y_{k+1}$ for $2 \le k \le n-1$, and $\dot{y}_n = y_{n-1} - 2y_n$, with $y_1(0) = 1$ and all other components $y_k(0) = 0$, is one of the test problems in the DETEST suite. Solve this problem on $0 \le t \le 20$ for $n = 100$ and $n = 1000$ using `ode45`, `ode113`, `ode15s`, and `ode23`. In the game of Rock–Paper–Scissors, Rock breaks Scissors, Paper wraps Rock, and Scissors cuts Paper; on this problem you should find that one of the four methods is clearly the most efficient for a given accuracy (or equivalently the most accurate for a given amount of work). Yet we will see other problems where each of the other three methods is the best.

12.21. The problem $\dot{y} = 1 - y^4$, $y(0) = 0$ has an analytical solution of sorts, but an *implicit* one:

$$t = \frac{1}{2}\arctan(y) + \frac{1}{4}\ln(y+1) - \frac{1}{4}\ln(1-y)\,. \tag{12.71}$$

Solve the differential equation numerically using for instance `ode45` and see how nearly the implicit Eq. (12.71) is satisfied. Is this a kind of forward error?

12.22. The problem

$$\begin{aligned}
{y_1}' &= 2xy_4y_1 \\
{y_2}' &= 10xy_4y_1^5 \\
{y_1}' &= 2xy_4 \\
{y_1}' &= -2x(y_3-1)
\end{aligned} \tag{12.72}$$

with initial conditions $y_k(0) = 1$ has a known reference solution $y_1 = \exp(\sin(x^2))$, $y_2 = \exp(5\sin(x^2))$, $y_3 = 1 + \sin(x^2)$, and $y_4 = \cos(x^2)$ (indeed the equations are constructed in Hairer et al. (1993) from the solutions). Solve this problem on $0 \le x \le 5$ using `ode45` and `ode113` at various tolerances and compute the residuals. Compute also the forward error and compare with the residuals.

Investigations and Projects

12.23. Euler's quadratic differential equation $(1-x^2)(y'(x))^2 = 1 - y(x)^2$ has various singularities. Explore them numerically, and compare with the analytic solution.

12.24. We have claimed that interpolation of a numerical solution of a differential equation gives a *continuously differentiable* (piecewise) solution. We investigate this claim in more detail here. Assume that the nodes t_k, $0 \le k \le n$ and the corresponding values x_k are machine-representable numbers. Suppose that the interpolation scheme uses the polynomial bases $\phi_{k,j}(t)$ for $0 \le j \le m_k$. In practice, one can take all the m_k to be the same, but you will see that you need $m_k \ge 1$. For `ode45` all $m_k = 5$, for instance.

1. Show that if exact arithmetic is used, then the Hermite interpolation conditions $x(t_k) = x_k$ and $\dot{x}(t_k) = f(x_k)$ ensure that the interpolant is continuously differentiable [and that the residual (defect) is zero at each t_k].
2. Show that rounding errors will in general destroy the continuity, and therefore also the differentiability, of the interpolant if we use the forward notion of error of evaluation.
3. Show nonetheless that there exists a continuously differentiable interpolant near the computed interpolant. That is, by using backward error, one can claim that the defect is continuous. [Hint: Use induction on k. Consider first the node t_1. Bound how accurately the interpolant on the left (i.e., $t_0 \le t \le t_1$) reproduces x_1 and $f(x_1)$, and then use backward error only on the interpolant on the *right*, that is, $t_1 \le t \le t_2$. Complete the induction. It is simpler, by the way, to specialize to local monomial bases $\phi_{k,j} = (t-t_k)^j$, but not necessary.]
4. Use `ode113` at much tighter tolerances to see how accurately you can solve the given IVP in MATLAB. At what point does the code begin to fail to be able to achieve the tolerance? Be careful to adjust the absolute tolerance as you tighten the relative tolerance. (This is simply an exercise in running the given code with different tolerances and reporting on what you see.)

12.25. The solutions to our favorite IVP for ODE, namely,

$$\dot{x}(t) = \cos(\pi t x(t)), \tag{12.73}$$

give nice pictures on the square $0 \le x \le 5$, $0 \le t \le 5$, when the solutions from several initial conditions are plotted at once. See Fig. 12.24. This problem is discussed

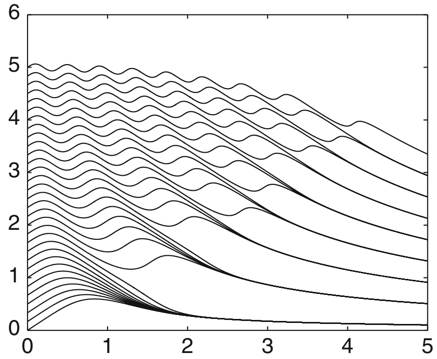


Fig. 12.24 What your solution to Exercise 12.25 should look like

in Bender and Orszag (1978). For this exercise, choose 31 equally spaced initial values $x(0) = {}^{5k}\!/_{30}$, for $0 \le k \le 30$, solve the problem using `ode45`, and plot all the curves on one graph as in the figure. Indeed, if one vectorizes the MATLAB code, all 31 initial-value problems can be solved at once. Also compute and plot the residuals (not shown here). What tolerances do you have to use to ensure that the absolute residuals are all less than about 2.0×10^{-6}? Finally, are these initial-value problems well-conditioned? Are there initial conditions for which the problem is *ill*-conditioned? If so, find some. In detail, what is the condition number of the initial-value problem (12.73)? This is a nonautonomous problem, but just consider the scalar version, assume the residual $\Delta(t) = \varepsilon v(t)$, put the difference $z - x \approx \varepsilon u_1(t)$, and ignore terms of order ε^2. Can you explain the curious "bunching" of the graphs?

12.26. Use the parameter values given in Example 12.12 to solve Eq. (12.45) (not the analytically solvable Eq. (12.48)). That is, do not approximate $\cos\alpha = {}^{\dot{x}}\!/_{v}$ or $\sin\alpha = {}^{\dot{y}}\!/_{v}$. Use the events feature of `ode45` (or `ode23`) to find how far the ball carries. Be careful to make your units consistent (you may keep them in the given medieval units if you choose; after all, golf is an ancient sport). Compare your numerical solution with GVP's analytic solution.

Curiously enough, the dimple patterns of the smaller British golf balls give more lift; this translates into about 15% larger p. What difference does this make to the distance the ball travels (be careful, the ball is also smaller)? What if they were the same size? What if the lift and drag are instead zero as would happen if one played golf in a vacuum? What happens, for either kind of ball, if the axis of spin is not exactly horizontal, so the lift vector is not directed upward? How accurately do you have to integrate these equations to come to your conclusions? What physical interpretation could you place on the residual in your computations?

12.27. The DETEST suite described in Enright and Pryce (1987) contains a collection of test problems, some stiff and some nonstiff. Choose a subset of those problems (possibly the whole set) and compare the performance of `ode23`, `ode45`, `ode113`, and `ode15s` as appropriate. Measure the size of the residuals, not the "global error," so that you don't have to worry about the exact solutions. All of these problems are supposed to be fairly well-conditioned; check the condition number of at least one problem, to be sure.

12.28. One can use numerical methods for ODE to plot curves defined implicitly by equations $f(x,y) = 0$. The technique used in Aruliah and Corless (2004) is to assume the existence of a smooth parameterization $x(s)$, $y(s)$ such that $f(x(s),y(s)) = 0$. Taking derivatives, we see that if

$$x'(s) = \alpha(s,x,y) f_y(x,y)$$
$$y'(s) = -\alpha(s,x,y) f_x(x,y)\,, \tag{12.74}$$

then ${}^{df}/_{ds} = f_x x' + f_y y' = 0$, no matter what choice is made for $\alpha(s,x,y)$. Indeed, it can usually be chosen so that $(x')^2 + (y')^2 = 1$, so that s represents arc length along the curve. Consider the curve

$$f(x,y) = \left(x^2+y^2\right)^2 + 3x^2y - y^3 = 0\,, \tag{12.75}$$

which goes through the point $(0,1)$, but has a singular point (where $f_x = f_y = 0$) at the origin. See if you can use the differential equation method [or a DAE method, adding the equation $0 = f(x,y)$ to the mix] to draw all three lobes of the curve. This is the technique used in the MAPLE routine `plot_real_curve` in the `algcurves` package, by the way; that routine also does Puiseux series expansions at singular points in order to get the numerics started.

12.29. Consider the linear homogeneous second-order differential equation

$$(1-t)\frac{d^2y}{dt^2} + \frac{dy}{dt} + (1-t)y = 0 \tag{12.76}$$

with the initial conditions $y(0) = y'(0) = 1$. This is solved in series in Braun (1992). The computer algebra system MAPLE gets a symbolic expression containing Bessel functions for the solution of this initial-value problem. Solve the problem numerically on the interval $0 \le t \le 2$, with the default tolerances in `ode45` and `ode113`. Solve it again using tight tolerances (say 10^{-12}). Does there appear to be any difficulty? Compute and plot the residual. Now does there appear to be any difficulty? Should we have believed the numerical solution without checking it? What happens with the Bessel function expression given by MAPLE on this interval?

12.30. The well-known *error function* is defined as

$$\operatorname{erf}(t) := \frac{2}{\sqrt{\pi}} \int_0^t e^{-\tau^2}\, d\tau\,. \tag{12.77}$$

The *inverse* error function is what you get when you solve the equation $z = \operatorname{erf}(y)$ for y as a function of z. By implicit differentiation, $1 = \operatorname{erf}'(y)y'$ and by definition $\operatorname{erf}'(y) = \frac{2}{\sqrt{\pi}}\exp(-y^2)$, so we wind up with a differential equation

$$y' = \frac{\sqrt{\pi}}{2} e^{y^2} . \tag{12.78}$$

Because $\mathrm{erf}(0) = 0$, we have also that $y(0) = 0$. Because $\mathrm{erf}(\infty) = 1$, we see that the domain of this differential equation is $0 \le z < 1$. Solve this differential equation numerically. Compute the *condition number* of this differential equation by simultaneously solving the variational equation. Compute also the condition number of $y(z)$ as a function of z (by the techniques of Chap. 3). Compare the results. See also the MATLAB function `erfinv`.

12.31. Computation of Lyapunov exponents isn't quite straightforward, because of the numerical dependence of the columns of $\boldsymbol{\xi}$, where $\dot{\boldsymbol{\xi}} = \mathbf{J}_f(\mathbf{z})\boldsymbol{\xi}$ and $\boldsymbol{\xi}(t_0) = \mathbf{I}$. What we mean by that is that although the matrix $\boldsymbol{\xi}$ is mathematically guaranteed to be nonsingular, the exponential growth of the largest singular value means that the corresponding singular vector grows parasitically in each of the other columns. There are a variety of ways to compute Lyapunov exponents; we will explore just one. Suppose the system we wish to solve is $\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z})$. Initial conditions may vary. We will need to compute the n^2 elements of the fundamental solution matrix $\boldsymbol{\xi}$ that satisfies $\boldsymbol{\xi}(t_0) = \mathbf{I}$ and $\dot{\boldsymbol{\xi}} = \boldsymbol{J}_f(\mathbf{z})\boldsymbol{\xi}$. However, because of the eventual numerical dependence difficulty, we must *renormalize* occasionally. Instead of computing the SVD of $\boldsymbol{\xi}(t)$ and tracking the exponential growth (or decay) of $\sigma_k(t)$, what we do is take a typical normalization time (say $T = 1$) and solve the problem $\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z})$, $\mathbf{z}(t_k) = \mathbf{z}_k$, and $\dot{\boldsymbol{\xi}} = \mathbf{J}_f(\mathbf{z})\boldsymbol{\xi}$ subject to $\boldsymbol{\xi}(t_k) = \mathbf{Q}_k$. That is, we integrate over the interval $t_k \le t \le t_{k+1} = t_k + T = t_k + 1$. At the end of the step, we perform a QR factoring on $\boldsymbol{\xi}(t_{k+1}) = \mathbf{Q}_{k+1}\mathbf{R}_{k+1}$. The absolute values of the diagonal elements of $\mathbf{R}_{k+1}$ contribute to estimates of the growth of the $\sigma_j(t)$, and the orthogonal matrix $\mathbf{Q}_{k+1}$ is used as the initial condition for the next step. Figure out how the elements of $\mathbf{R}$ are related to the Lyapunov exponents, and implement this method. Try the method out on the Rössler system (12.42).

12.32. We have stated in this book that the built-in interpolant for `ode45` isn't quite satisfactory for computing a residual, because it isn't quite of high enough order of accuracy: It should be $O(h^5)$ not $O(h^4)$ accurate, and usually therefore it overestimates the residual. This problem asks you to explore an alternative (more expensive) interpolant. You may do this in one of two ways. First, as discussed in Hairer et al. (1993), one might interpolate over *two* intervals and use the data $(t_k, x_k, f(t_k, x_k))$ at each of the three points; this gives a piecewise quintic interpolant that should have the correct order. This has the disadvantage of extra bookkeeping, but it can be expected to be more accurate. A second alternative is to use just one interval at a time as usual, but provide the second derivative $\ddot{\mathbf{x}}$ at each end, again giving a piecewise quintic interpolant. This has the advantage of less bookkeeping, and you may easily modify your `pqhip` program to compute the interpolant and its derivatives, but of course has the disadvantage of requiring you to provide the second derivative. We point out that the second derivative can be computed by $\ddot{\mathbf{x}} = \mathbf{J}_f(\mathbf{x})\dot{\mathbf{x}}$ and the Jacobian is sometimes available anyway (and always helpful when it is).

Use either method, and take the `ode45` solution of $\dot{x} = x^2 - t$, $x(0) = -1/2$ on $0 \le t \le 5$ for various tolerances and show that your higher-order interpolant does indeed produce a smaller residual. Show also that this technique does not improve the residual if `ode113`, `ode23`, or `ode15s` is used instead of `ode45`.

Chapter 13
Numerical Methods for ODEs

Abstract This chapter gives a very brief survey of *Runge–Kutta methods*, including *continuous explicit Runge–Kutta methods*. We also discuss *multistep methods* and the *Taylor series method* (i.e., analytic continuation). We talk about *implicit methods for stiff problems* and backward error by the *method of modified equations*. We mention some of the several flavors of *stability* of a numerical method for solving IVP that are sometimes useful. We talk about *interpolants* and the *residual* and compare the residual with *local error per unit step*. We sketch methods for *adaptive step-size control*. ◁

In the previous chapter, we investigated what a numerical solution to an initial-value problem *is*, as well as how to use high-quality codes to obtain one. We showed how to compute a residual and interpret it as a backward error, and extended the concept of condition number to the context of IVPs. As is true for all mathematical models, we saw that conditioning is an intrinsic property of a problem and its formulation rather than a property of a particular numerical method to solve it.

Now, it's time to investigate what's under the hood, in other words, what the codes are actually doing to construct the skeleton of the solution, and why they work or fail for certain types of problems. We will begin our investigation with a venerable method, namely, *Euler's method*. Through the presentation of this method, we will introduce some key concepts such as the distinction between *implicit* and *explicit* methods, *local*, *global*, and *residual error*, and discuss just how to do *adaptive step-size selection*, and why. We also discuss, as you might expect by now, the important issue of interpolation, which is needed to flesh out the skeleton!

Euler's method turns out to be a particular case of more general methods, and all of these concepts can be demonstrated first for Euler's method. Then, generalizing in one way, we will obtain the *Taylor series methods* (also called Obreschkoff methods), which are in some sense fundamental. Another generalization, which we take in order to avoid the effort of taking derivatives, gives us the *Runge–Kutta methods*. Yet another generalization gives us *multistep methods*, which are also important

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_13,

in practice. A third class of useful methods, *extrapolation methods*, are not covered here, except in Exercise 13.31. As already mentioned, these methods will each come in two flavors: explicit and implicit (the latter being used for the solution of stiff problems).

13.1 Euler's Method and Basic Concepts

Euler's method is in some sense the most fundamental method for numerically solving initial-value problems. Although simple to program, it is of limited use because it is not very accurate for a given amount of computation time; equivalently, it is not very efficient for a given accuracy.[1] Nonetheless, since it is theoretically and pedagogically important, it is worth looking at in some detail. We will use it to introduce a number of fundamental concepts that will reappear in our study of more sophisticated methods. We will also see that such methods correspond to different ways of generalizing Euler's method.

The basic idea of Euler's method is one shared by many important methods of calculus. Consider the initial-value problem $\dot{x}(t) = f(t,x(t))$, with f having Lipschitz constant L on the region of interest; also, suppose $x(t_0) = x_0$, and let $x(t)$ be the unique reference solution. Since we are given the values t_0, x_0, and $\dot{x}(t_0)$ [since we can compute $f(t_0,x(t_0))$], we can make the construction shown in Fig. 13.1. Now,

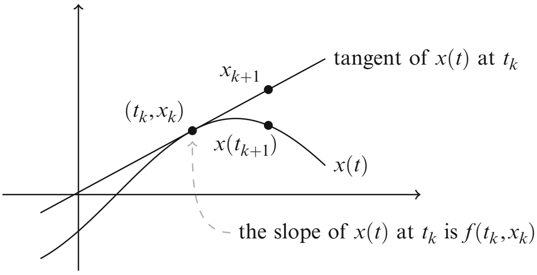

tangent of $x(t)$ at t_k
x_{k+1}
(t_k,x_k)
$x(t_{k+1})$
$x(t)$
the slope of $x(t)$ at t_k is $f(t_k,x_k)$

Fig. 13.1 A step with Euler's method, where $x_k = x(t_k)$ is assumed to be known exactly

consider a forward time step h with the tangent line approximation so that $t_1 = t_0 + h$ and $x_1 = x_0 + hf(t_0,x_0)$. In general, we won't have the exact equality $x(t_1) = x_1$. Nonetheless, if h is small, we will have the approximation $x(t_1) \approx x_1$. Euler's method precisely consists in replacing the computation of $x(t_1)$ by the computation of x_1. Since it is enough to have the point (t_0,x_0) and the value of $f(t_0,x_0)$ to find the equation of the tangent of $x(t)$ at (t_0,x_0), we have

[1] It should be noted, however, that if *robustness* is paramount and the code is not allowed to return the error message "did not converge," then until recently Euler's method was the most efficient method known for some classes of problems. This is no longer the case (see Christlieb et al. 2010).

$$x_1 = x_0 + hf(t_0, x_0)\,.$$

In other words, we treat the problem based on the fact that the value of x changes approximately linearly as $h \to 0$, with f as its slope.

Then, if x_1 is close to $x(t_1)$, we can *pretend* that this point is on the curve $x(t)$, and repeat the same operation, with $f(t_1, x_1)$ as the slope. We will thus get a point (t_2, x_2), which will hopefully satisfy $x(t_2) \approx x_2$ to a sufficient degree. Using the map

$$x_{n+1} = x_n + hf(t_n, x_n)\,,$$

we can then generate a sequence of values $x_0, x_1, x_2, \ldots, x_N$ at the mesh points $t_0, t_1, t_2, \ldots, t_N$ that approximates $x(t_0), x(t_1), x(t_2), \ldots, x(t_N)$. This is, of course, the skeleton of the numerical solution. This iterative process gives rise to Algorithm 13.1. Analytically, as we approach the limit $h \to 0$, it is expected that the approximation will become arbitrarily good. An example is presented in Fig. 13.2. Numerically, however, it is important to be careful, since for very small values of h, rounding error may prevent us from obtaining the desired convergence.

Algorithm 13.1 Fixed-step-size explicit Euler method

```
Require: The right-hand side $f(t, \mathbf{x}(t))$ of an IVP in standard form, an initial value $x_0$, a time span
    $[t_0, t_f]$, and a step size $h$
  $n = 0$
  while $t_n < t_f$ do
    $x_{n+1} = x_n + hf(t_n, x_n)$
    $t_{n+1} = t_n + h$
    $n := n + 1$
  end while
  Obtain a continuous function $z(t)$ by somehow interpolating on $\mathbf{t}, \mathbf{x}$, typically piecewise.
  return $z(t)$
```

Euler's method is not limited to scalar problems. For an IVP posed in standard form, we have the vector map

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{f}(t_n, \mathbf{x}_n) \qquad\qquad \mathbf{x}_0 = \mathbf{x}(t_0)\,. \tag{13.1}$$

The method then produces a sequence of vectors $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k, \ldots, \mathbf{x}_N$ whose ith components x^i_k approximate the value of the ith component of the solution at time k, that is, $x^i_k \approx x^i(t_k)$. We can interpolate those points to obtain a continuous approximation $\mathbf{z}(t)$ to $\mathbf{x}(t)$. Algorithm 13.1 can be rewritten in vector form in an obvious way.

Observe the notation x^i_k. In this chapter, we use superscript i to denote the ith component of a vector—typically $\mathbf{x}$ or $\mathbf{f}$—and subscript k to denote the time step t_k. In context, there should not be confusion with exponents. We introduce this notation as a matter of necessity since, later in this chapter, we will have to introduce tensor notation that naturally builds on the one we introduce here.

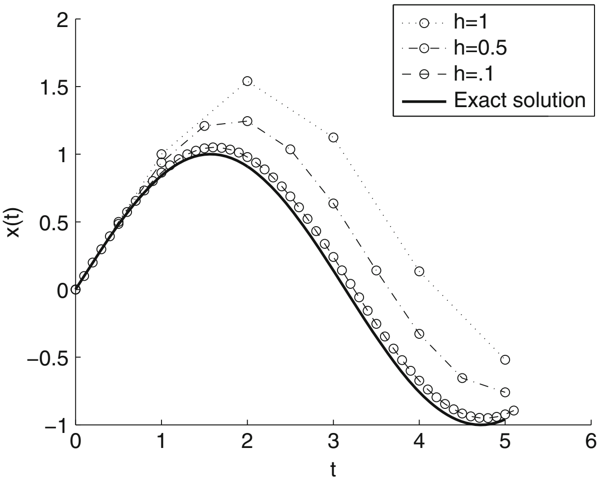


Fig. 13.2 Effect of reducing the step size in Euler's method

Because $\mathbf{z}(t)$ is iteratively generated by a discrete-time variable in order to approximate $\mathbf{x}(t)$, Euler's method (and all the later explicit methods) is often called a *marching method*. In general, such methods will produce the value of $\mathbf{x}_{k+1}$ by means of the values of $\mathbf{x}_k, \mathbf{x}_{k-1}, \ldots, \mathbf{x}_1, \mathbf{x}_0$. (Once the skeleton of the solution has been constructed by marching, we may interpolate to produce a continuous or even continuously differentiable solution. Indeed, for IVP, the interpolation can proceed simultaneously with the marching.) Thus, we can introduce the notation

$$\mathbf{x}_{k+1} = \Phi(t_k; \mathbf{x}_{k+1}, \mathbf{x}_k, \mathbf{x}_{k-1}, \ldots, \mathbf{x}_0; h; \mathbf{f}), \tag{13.2}$$

to represent an arbitrary method. If Φ does not depend on $\mathbf{x}_{k+1}$, the method is said to be *explicit*. Euler's method is an explicit method. However, if Φ depends on $\mathbf{x}_{k+1}$, then we will have to solve for $\mathbf{x}_{k+1}$ in some way (perhaps using a variation of Newton's method); in this case, the method is said to be *implicit*.

13.2 Error Estimation and Control

From now on, we assume that the methods can have varying step sizes. So, we will add a subscript k to h to indicate $h_k = t_{k+1} - t_k$. This is important since most high-quality general-purpose methods adapt the step size, often resulting in lower-cost solutions than otherwise. In fact, the programs implementing such methods will automatically increase or decrease the step sizes on the basis of some assessment of the error, not merely for efficiency but more importantly to provide some assessment of the *quality* of the resulting solution.

As we have seen for quadrature in Sect. 10.3 (and we will see more of in Sect. 14.4), the basic technique in error control is to try to equidistribute the estimated error: In an ideal world, we would have $\mathrm{err}_i = \varepsilon$ on every step. Asymptotically, we model the error on the ith step as $\mathrm{err}_i = \phi_i h_i^p$. If we use an *estimate* of the error on the ith interval (that is, we somehow estimate err_i), we may use it to compute an estimate of ϕ_i. If we *assume* that ϕ_{i+1} won't be much different than ϕ_i (we will say in this case that the error constants are "weakly dependent on the mesh" in Sect. 14.4), then in order to have $\phi_{i+1}h_{i+1}^p = \varepsilon$, we should then *choose*

$$h_{i+1} = \left(\frac{\varepsilon}{\mathrm{err}_i}\right)^{1/p} h_i\,, \tag{13.3}$$

because then $\phi_{i+1}h_{i+1}^p \approx \phi_i h_i^p (\varepsilon/\mathrm{err}_i) = \varepsilon$. This idea doesn't *quite* work, and one has to include some safety factors to be conservative; after all, the ϕ_i will change a little from step to step; and then there is the issue of how accurate an estimate of the error err_i actually is. It has been found by experience that using certain "magic constants" such as 0.8 or 0.9 will make the code work fairly well and fairly reliably on various classes of test problems. For example, the relevant line in MATLAB's `ode45` is

```
absh = max(hmin, absh * max(0.1, 0.8*(rtol/err)^pow));
```

Not only is there a factor 0.8 there, but the new step is not allowed to be smaller than one tenth the size of the previous one and is never allowed to go below a certain minimum step size. However, it must be said that these magic constants are based on experience and not strict theorems, and because the problems we are interested in are nonlinear, there is room for numerical methods to fail.[2] But no matter how the error estimates are used to predict the next step size, we still need a good error estimate. Accordingly, let us turn to error estimation.

13.2.1 The Residual

We have seen in Chap. 12 that the residual of a numerical solution is

$$\Delta(t) = \dot{\mathbf{z}} - \mathbf{f}(t, \mathbf{z}(t))\,,$$

where $\mathbf{z}(t)$ is *differentiable* (or at least piecewise differentiable). That the numerical solution be differentiable is very important for the use of the concept of residual in error control, since its evaluation requires the derivative $\dot{\mathbf{z}}$ of the numerical solution.

What is the residual of a solution produced with Euler's method? As we have seen, the Euler method generates a sequence of point $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ based on the

[2] One prominent ODE researcher of our acquaintance reminds us that "there is no place for Authority in mathematics." It is one thing if people report experimental claims—and indeed the effectiveness of these "magic" constants has been tested by many thousands of problems solved and so reported—but independent assessment is justified for *your* problem, which may never have been solved before. That is why we take a "trust, but verify" approach using the residual in this book.

rule $\mathbf{x}_{k+1} = \mathbf{x}_k + h_k\mathbf{f}(t_k, \mathbf{x}_k)$. The numerical solution is then generated from those points by interpolation. We then say that the numerical solution $\mathbf{z}$ is the interpolant of those points on the given mesh, that is, of the skeleton of the solution. The residual may be different depending on the choice of interpolant.

Given the rudimentary character of the Euler method, it makes sense to choose the equally rudimentary piecewise linear interpolant, which is the one used in Fig. 13.2. Between the point t_k and t_{k+1}, the interpolant will then be

$$\mathbf{z}_k(t) = \mathbf{x}_k + (t - t_k)\frac{\mathbf{x}_{k+1} - \mathbf{x}_k}{t_{k+1} - t_k} = \mathbf{x}_k + (t - t_k)\mathbf{f}(t_k, \mathbf{x}_k)\,. \tag{13.4}$$

Moreover, if we let

$$\theta_k = \frac{t - t_k}{t_{k+1} - t_k} = \frac{t - t_k}{h_k}\,,$$

we can write the pieces of the interpolant as

$$\mathbf{z}_k(\theta_k) = \mathbf{x}_k + \theta_k(\mathbf{x}_{k+1} - \mathbf{x}_k) = \mathbf{x}_k + h_k\theta_k\mathbf{f}(t_k, \mathbf{x}_k)\,. \tag{13.5}$$

Then, the argument θ_k ranges over the interval $[0, 1)$. As a result, for the mesh points $t_0, t_1, \ldots, t_N$ and the generated points $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_N$, the interpolant is then defined piecewise as

$$\mathbf{z}(t) = \begin{cases} \mathbf{z}_0(t) & t_0 \le t < t_1 \\ \mathbf{z}_1(t) & t_1 \le t < t_2 \\ \vdots & \\ \mathbf{z}_k(t) & t_k \le t < t_{k+1} \\ \vdots & \\ \mathbf{z}_{N-1}(t) & t_{N-1} \le t \le t_n \end{cases}\,,$$

or it can equivalently be written as a function of θ.

Remark 13.1. With this interpolant, the solution is only piecewise differentiable (although it is continuous). Therefore, the residual that we will compute will contain jump discontinuities; we have termed such a residual a “deviation” and reserved the word “defect” for what results from using a continuously differentiable interpolant. If we used cubic Hermite interpolation here, matching not just the values $\mathbf{x}_k$ but also the slopes $\mathbf{f}(\mathbf{x}_k)$, then we would have a continuous residual. However, the extra complication is not worth it for the purpose of the present discussion. ◁

Theorem 13.1. *The Euler method has an $O(h)$ residual.*

We will give a detailed proof that includes some typical series manipulations and introduces notation that we will use later in the chapter.

Proof. Without loss of generality, we choose an interval $t_k \le t < t_{k+1}$. We can substitute the explicit expression of (13.4) for $\mathbf{z}$ in the definition of residual:

$$\begin{aligned}\Delta(t) &= \frac{d}{dt}\left(\mathbf{x}_k + (t-t_k)\mathbf{f}(t_k,\mathbf{x}_k)\right) - \mathbf{f}\big(t,\mathbf{x}_k + (t-t_k)\mathbf{f}(t_k,\mathbf{x}_k)\big)\\ &= \mathbf{f}(t_k,x_k) - \mathbf{f}\big(t,\mathbf{x}_k + (t-t_k)\mathbf{f}(t_k,\mathbf{x}_k)\big)\,. \qquad (13.6)\end{aligned}$$

Now, we will expand $\mathbf{f}(t,\mathbf{x}_k + (t-t_k)\mathbf{f}(t_k,\mathbf{x}_k))$ in a Taylor series about the point $(t_k,\mathbf{x}_k)$:

$$\begin{aligned}&\mathbf{f}\big(t,\mathbf{x}_k + (t-t_k)\mathbf{f}(t_k,\mathbf{x}_k)\big) =\\ &\qquad \mathbf{f}(t_k,x_k) + (t-t_k)\mathbf{f}_t(t_k,\mathbf{x}_k) + (t-t_k)\mathbf{J}_{\mathbf{f}}(t_x,\mathbf{x}_k)\mathbf{f}(t_k,\mathbf{x}_k) + O((t-t_k)^2).\end{aligned}$$

Here, $\mathbf{f}_t(t_k,\mathbf{x}_k)$ is the vector of partial derivatives of $\mathbf{f}$ with respect to t and $\mathbf{f}_{\mathbf{x}}(t_k,\mathbf{x}_k)$ is the Jacobian matrix with partial derivatives with respect $\mathbf{x}$, both evaluated at $(t_k,\mathbf{x}_k)$. From now on, we will not explicitly write the point of evaluation of $\mathbf{f}$ and its derivatives when it is $(t_k,\mathbf{x}_k)$, and simply write $\mathbf{f},\mathbf{f}_t$, and $\mathbf{J}_{\mathbf{f}}$, for otherwise the expressions would quickly become unreadable. Adding the fact that $\theta_k h_k = t - t_k$, we obtain the much neater expression

$$\mathbf{f}\big(t,\mathbf{x}_k + \theta_k h_k \mathbf{f}\big) = \mathbf{f} + \theta_k h_k(\mathbf{f}_t + \mathbf{J}_{\mathbf{f}}\mathbf{f}) + O(h_k^2)\,.$$

Putting this in Eq. (13.6), we find that

$$\begin{aligned}\Delta(t) &= \mathbf{f} - \big(\mathbf{f} + \theta_k h_k(\mathbf{f}_t + \mathbf{J}_{\mathbf{f}}\mathbf{f}) + O(h_k^2)\big)\\ &= -\theta_k h_k(\mathbf{f}_t + \mathbf{J}_{\mathbf{f}}\mathbf{f}) + O(h_k^2),\end{aligned}$$

which is $O(h)$ since k was arbitrarily chosen.[3] ♮

Example 13.1. Take $\dot{x} = \cos(\pi t x)$ and $x(0) = 2$. We take two Euler steps with $h = 0.03$. The skeleton of the solution is $[2.0000, 2.0300, 2.0595]$ at $t = 0$, $t = h$, and $t = 2h$. Moreover, the slopes $\dot{x}$ are $[1.0000, 0.9818, 0.9256]$ at those times.

We interpolate by straight lines, and also by cubic Hermite interpolants, to see if the residual is any smaller with the smoother interpolant. As it turns out, it is, but not significantly: Both are $O(h)$. See Fig. 13.3.

◁

Note that, in this proof, we have also given an explicit expression for the first term of the residual of the numerical solution on an arbitrary subinterval. It can be expanded in vectors and matrices as follows:

$$\Delta(t) \doteq -\theta_k h \left(\begin{bmatrix} \partial f_1/\partial t \\ \partial f_2/\partial t \\ \vdots \\ \partial f_n/\partial t \end{bmatrix}_{\substack{t=t_k\\ \mathbf{x}=\mathbf{x}_k}} + \begin{bmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \cdots & \partial f_1/\partial x_n \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \cdots & \partial f_2/\partial x_n \\ \vdots & \vdots & \ddots & \vdots \\ \partial f_n/\partial x_1 & \partial f_n/\partial x_2 & \cdots & \partial f_n/\partial x_n \end{bmatrix}_{\substack{t=t_k\\ \mathbf{x}=\mathbf{x}_k}} \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}_{\substack{t=t_k\\ \mathbf{x}=\mathbf{x}_k}} \right).$$

[3] In Chap. 14, we will see a justification for using $\overline{h}$, the arithmetic mean of the step sizes, instead of a maximum h_k, but for now we may assume that the maximum h_k is h and that this goes to zero.

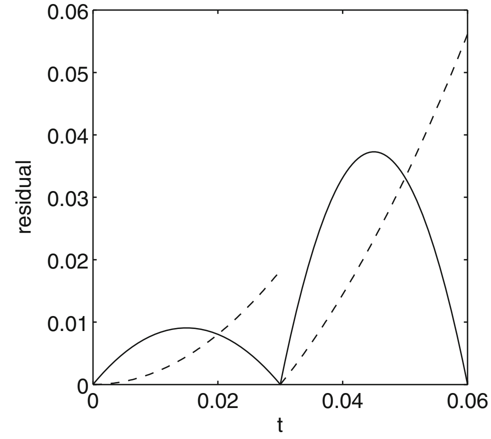


Fig. 13.3 Residuals in Euler's method, with a linear interpolant (*dashed line*) and cubic Hermite interpolation (*solid line*). As we see, cubic Hermite interpolation is smoother, so that the residual is a defect and not just a deviation and is somewhat smaller although still $O(h)$

Note also that, instead of giving an explicit expression for the residual in terms of its Taylor expansion, there are situations in which we are satisfied with only a bound for it. Suppose that $\mathbf{f}$ satisfies a Lipschitz condition with Lipschitz constant L. Moreover, without loss of generality, suppose $\mathbf{f}$ is autonomous. The residual of a numerical solution generated with the Euler method is then

$$\begin{aligned}\Delta(t) &= \dot{\mathbf{z}}(t) - \mathbf{f}(\mathbf{z}) = \frac{d}{dt}(\mathbf{x}_k + (t-t_k)\mathbf{f}(\mathbf{x}_k)) - \mathbf{f}(\mathbf{x}_k + (t-t_k)\mathbf{f}(\mathbf{x}_k)) \\ &= \mathbf{f}(\mathbf{x}_k) - \mathbf{f}(\mathbf{x}_k + (t-t_k)\mathbf{f}(\mathbf{x}_k)),\end{aligned}$$

and so

$$\begin{aligned}\|\Delta(t)\| &\leq L\|\mathbf{x}_k - \mathbf{x}_k - (t-t_k)\mathbf{f}(\mathbf{x}_k)\| \\ &\leq Lh_k\|\mathbf{f}(\mathbf{x}_k)\|\,.\end{aligned}$$

This is a natural inequality given the connection between the Jacobian and the Lipschitz constant.

We can now define the important concept of the order of a method.

Definition 13.1 (Order of a method Φ). If a fixed time-step method Φ with step size h has residual $O(h^p)$ as $h \to 0$, then it is said to be a pth-order method. ◁

In general, the higher the order of a method is, the more accurate it is for smooth problems. In the literature on the numerical solutions of ODEs, the concept of accuracy is usually formulated using the following definition:

Definition 13.2 (Global error). The global error $\mathbf{ge}(t)$ for the numerical solution from t_0 to t is simply

$$\mathbf{ge}(t) = \mathbf{z}(t) - \mathbf{x}(t)\,. \tag{13.7}$$

As we see, the global error is simply what we have so far called *forward error*. ◁

In fact, we find it preferable to use the term "forward error" rather than "global error," since it makes explicit the uniformity of the methods of error analysis about differential equations and about other things, such as matrix equations, roots of polynomials, interpolation, and so on.

Note that since the global error is nothing but the forward error, we can use the formulæ of Sect. 12.3 for it, such as the Gröbner–Alexeev formula

$$\mathbf{z}(t) - \mathbf{x}(t) = \int_{t_0}^{t} \mathbf{G}(t,\tau,\mathbf{z}(\tau))\Delta(\tau)d\tau\,.$$

We can also use the approximate inequality

$$\|\mathbf{z}(t) - \mathbf{x}(t)\| \leq \kappa(\mathbf{X}_1)\|\Delta(t)\|\,,$$

where $\mathbf{X}_1$ is a fundamental solution of the first variational equation; this approximate inequality holds in the limit as $\|\Delta\| \to 0$ whether or not we write $\Delta(t)$ or $\Delta(t,x(t))$, explicitly noting the correlations between the computed solution and the residual. Sometimes, as we wrote in Chap. 12, changing notation and writing the residual as some function $\varepsilon\mathbf{v}$ with $\|\mathbf{v}\|_\infty \leq 1$ makes things clearer. Now, if we know what method has been used for the computation of the numerical solution, we can actually find an expression for $\varepsilon\mathbf{v}$, if we wish to do so.

This shows that the order of the residual and the order of the global error are the same, so the order of a method can be characterized by one or the other interchangeably.

13.2.2 Local Error

A standard way to work with the error in the numerical solution of ODEs is based on the concept of local error. We think this concept should be replaced by the use of the residual, but that opinion is by no means universal, and local error is used in many codes. To define the local error, we first need the following:

Definition 13.3 (Local reference solution). If $\mathbf{x}_k(t)$ satisfies the reference problem $\dot{\mathbf{x}} = \mathbf{f}(t,\mathbf{x})$ with the local initial condition $\mathbf{x}(t_k) = \mathbf{x}_k$, it is called the *local reference solution*. ◁

Remark 13.2. We will often use $\mathbf{x}_k$ to represent a constant vector; a piece of the skeleton of the solution, in fact. We will use $\mathbf{x}_k(t)$ to represent the local reference solution as above; note that $\mathbf{x}_k(t_k) = \mathbf{x}_k$ is then satisfied. However, $\mathbf{x}_k(t_{k+1})$ will not in general be $\mathbf{x}_{k+1}$; that is the point at which we will change to the next local reference solution, so that $\mathbf{x}_{k+1}(t_{k+1}) = \mathbf{x}_{k+1}$. ◁

Definition 13.4 (Local truncation error). The *local truncation error* or, for short, *local error*, of a computed solution is the error **le** incurred by the method $\boldsymbol{\Phi}$ over a single time step $[t_k, t_{k+1}]$ of length h_k, as compared to the local reference solution. Accordingly, the local error is

$$\mathbf{le} = \mathbf{z}(t) - \mathbf{x}_k(t)\,, \tag{13.8}$$

for $t_k \leq t < t_{k+1}$. Note that the local error is always zero at $t = t_k^+$, but is likely to be maximum at $t = t_{k+1}^-$ at least asymptotically as $h_k \to 0$. If the order of the method is $O(h^p)$, the local error will be $O(h^{p+1})$. Associated with **le** is the local error per unit step, **lepus**, which is just **lepus** $= \mathbf{le}/h_k$. ◁

The concept of "local error" has traditionally been used in construction of numerical methods for the solution of differential equations, but has been used nowhere else. We strongly believe (following many other researchers) that the concept of *residual* is more general, more physically intuitive, and more understandable. In fact, we can imagine no reasonable interpretation of the local error in terms of the modeling context in which the computation occurs.[4] However, since because the algebra with local error is reasonably simple and because the concept is commonly used, we will explain it briefly here, and show how it relates to our approach in terms of residual.

To begin with, let us find the local error of Euler's method.

Theorem 13.2. *The local error in Euler's method is* $O(h^2)$ *as* $h \to 0$.

Proof. If we expand the local reference solution $\mathbf{x}_k(t)$ about t_k, we obtain

$$\mathbf{x}_k(t) = \sum_{\ell=0}^{\infty} \frac{\mathbf{x}^{(\ell)}(t_k)}{\ell!}(t-t_k)^{\ell} = \mathbf{x}_k(t_k) + \dot{\mathbf{x}}_k(t_k) + \frac{\ddot{\mathbf{x}}_k(t_k)}{2}(t-t_k)^2 + O\left((t-t_k)^3\right).$$

Because $\dot{\mathbf{x}}_k = \mathbf{f}(t, \mathbf{x}_k)$, we also have

$$\mathbf{x}(t) = \mathbf{x}(t_k) + \mathbf{f}(t_k, \mathbf{x}_k)(t-t_k) + \frac{\dot{\mathbf{f}}(t_k, \mathbf{x}_k)}{2}(t-t_k)^2 + O\left((t-t_k)^3\right)$$

[note that the derivative of $\dot{\mathbf{f}}$ at (t_k, x_k) is just $\dot{\mathbf{f}} = \mathbf{f}_t + \mathbf{J_f}\dot{\mathbf{x}} = \mathbf{f}_t + \mathbf{J_f}\mathbf{f}$, by the chain rule], and so we can rewrite our series for $\mathbf{x}_k(t)$ in this way:

$$\mathbf{x}_k(t) = \mathbf{x}_k(t_k) + (t-t_k)\mathbf{f} + \frac{(t-t_k)^2}{2}\left(\mathbf{f}_t + \mathbf{J_f}\mathbf{f}\right) + O\left((t-t_k)^3\right).$$

Now, we evaluate $\mathbf{x}_k(t)$ at $t = t_{k+1}$ using this series:

$$\begin{aligned}\mathbf{x}_k(t_{k+1}) &= \mathbf{x}_k(t_k) + (t_{k+1}-t_k)\mathbf{f} + \frac{(t_{k+1}-t_k)^2}{2}\left(\mathbf{f}_t + \mathbf{J_f}\mathbf{f}\right) + O\left((t_{k+1}-t_k)^3\right)\\ &= \mathbf{x}_k(t_k) + h\mathbf{f} + \frac{h^2}{2}\left(\mathbf{f}_t + \mathbf{J_f}\mathbf{f}\right) + O\left(h^3\right).\end{aligned}$$

[4] Perhaps you can. We're admitting defeat here, not claiming that no one can find such an interpretation and believe it reasonable.

So, the local error at the end of the step (where it asymptotically is maximum) is $\mathbf{le} = \mathbf{x}_k(t_{k+1}) - \mathbf{x}_{k+1}$, which is

$$\mathbf{le} = \mathbf{x}_k(t_{k+1}) - \mathbf{x}_{k+1} = \frac{h^2}{2}(\mathbf{f}_t + \mathbf{J}_\mathbf{f}\mathbf{f}) + O\left(h^3\right) = O\left(h^2\right). \tag{13.9}$$

Therefore, the local error is $O(h^2)$. ♮

In our reasoning about local truncation error, we compared the computed solution to the local reference solution. Somehow we were pretending that the local reference solution was something like the global reference solution $\mathbf{x}(t)$. But this need not be true. We are only guaranteed that $\mathbf{x}(t_0) = \mathbf{x}_0$; the further points $\mathbf{x}_1$, $\mathbf{x}_2$, ..., $\mathbf{x}_n$ will not, in general, be exactly on the curve $\mathbf{x}(t)$. Thus, for each step $k > 0$, we based our calculation on approximate initial points, and we must ask how the error accumulates as we go through the interval. This cumulative local truncation error gives another expression for the global error:

$$\begin{aligned}\mathbf{ge} &= \mathbf{x}(t_k) - \mathbf{x}_k \\ &= \mathbf{x}(t_k) - (\mathbf{x}_0 + \Phi(t_0;\mathbf{x}_1,\mathbf{x}_0;h;\mathbf{f}) + \ldots + \Phi(t_{k-1};\mathbf{x}_k,\mathbf{x}_{k-1},\ldots,\mathbf{x}_0;h;\mathbf{f})).\end{aligned}$$

This formula, together with its many simplifications, is to a large extent why, traditionally, the error analysis of numerical solution to differential equations has focused on the control of local error. By controlling the local error, we obtain a way to keep a certain control on the global error, which is what we often really want to control. This formula, however, involves a lot of bookkeeping and obfuscates the relation between error control in numerical analysis for ODEs and other fields of numerical analysis.[5]

Now, controlling the residual gives a more direct control of accuracy and, in fact, also characterizes the order of the method.[6] However, as an error-control strategy, controlling the local error provides mostly satisfactory results because it gives an indirect control on the residual. We make this precise below.

Theorem 13.3. *Controlling the local error per unit step indirectly controls the residual. Specifically, if $h_k \le h$, $h_k L_k \le B$, and* **lepus** $\le \varepsilon$*, then*

$$\|\Delta(t)\| \le \left(h + \frac{3}{2} + \frac{62}{27}B + \frac{4}{27}B^2\right)\varepsilon. \tag{13.11}$$

[5] The concept of local error is also potentially deceitful in another way. Many users erroneously think that setting `rtol=1.0e-6` in MATLAB means that the code will attempt to guarantee that

$$\|\mathbf{z}(t) - \mathbf{x}(t)\| \approx 10^{-6}\|\mathbf{x}(t)\|. \tag{13.10}$$

Instead, codes only try to make the difference small to the *local reference solution* $\mathbf{x}_k(t)$. The relationship to global error and the residual is often more remote than many users think. This potential confusion adds to the difficulty of interpretation of the quality of the numerical solution.

[6] Traditionally, the order of a method is said to be p when the local error is of order $p+1$. This is slightly awkward; the definition in terms of the residual seems to us more natural.

Proof. Without loss of generality, we assume that the Problem $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ is autonomous. We also suppose that the underlying numerical solution method has used the mesh $t_0 < t_1 < \ldots < t_k < t_{k+1} < \ldots < t_N$ and that, as per the hypothesis, the mesh has been chosen in such a way as to ensure that the **lepus** is less than or equal to a given tolerance $\varepsilon > 0$ on each subinterval.

As before, let $\mathbf{x}_k(t)$ be the *local reference solution* on the interval $t_k \le t < t_{k+1}$, so that

$$\dot{\mathbf{x}}_k(t) = \mathbf{f}(\mathbf{x}), \quad \mathbf{x}(t_k) = \mathbf{x}_k, \quad t_k \le t < t_{k+1}\,.$$

As defined above, we also have asymptotically $\mathbf{le} = \mathbf{x}_{k+1} - \mathbf{x}_k(t_{k+1})$ and $\mathbf{lepus} = {}^{\mathbf{le}}/_{h_k}$.

Now, consider the following theoretical interpolant satisfying the conditions $\mathbf{z}(t_k) = \mathbf{x}_k$, $\dot{\mathbf{z}}(t_k) = \mathbf{f}(\mathbf{x}_k)$, $\mathbf{z}(t_{k+1}) = \mathbf{x}_{k+1}$, and $\dot{\mathbf{z}}(t_{k+1}) = \mathbf{f}(\mathbf{x}_{k+1})$:

$$\mathbf{z}(t) = \mathbf{x}_k(t) + \frac{(t-t_k)^2}{h_k^2}\mathbf{le} + \left(\mathbf{f}(\mathbf{x}_{k+1}) - \mathbf{f}(\mathbf{x}_k(t_{k+1})) - \frac{2}{h_k}\mathbf{le}\right)\frac{(t-t_k)^2(t-t_{k+1})}{h_k^2}\,.$$

We take a moment to verify that the interpolant matches the conditions: $\mathbf{z}(t_k) = \mathbf{x}_k(t_k) = \mathbf{x}_k$ as specified; $\mathbf{z}(t_{k+1}) = \mathbf{x}_k(t_{k+1}) + \mathbf{le} = \mathbf{x}_{k+1}$ as specified; $\dot{\mathbf{z}}(t_k) = \dot{\mathbf{x}}_k(t_k) = \mathbf{f}(\mathbf{x}_k)$; and finally,

$$\begin{aligned}\dot{\mathbf{z}}(t_{k+1}) = \mathbf{f}(\mathbf{x}_k(t_{k+1})) + 2(t_{k+1} - t_k)\frac{\mathbf{le}}{h_k^2}& \\ + \left(\mathbf{f}(\mathbf{x}_{k+1}) - \mathbf{f}(\mathbf{x}_k(t_{k+1})) - 2\frac{\mathbf{le}}{h_k^2}\right)\frac{(t_{k+1}-t_k)^2}{h_k^2}&,\end{aligned}$$

which simplifies to $\mathbf{f}(x_{k+1})$, as it should. This might not be the best possible interpolant; what we get from this choice is a guaranteed bound on the residual from the local error bound, which is not necessarily the best possible bound on the residual (which may well be smaller).

Note that the derivative of this interpolant is

$$\begin{aligned}\dot{\mathbf{z}}(t) &= \mathbf{f}(\mathbf{x}_k(t)) + 2(t-t_k)\frac{\mathbf{le}}{h_k^2} \\ &+ 2\left(\mathbf{f}(\mathbf{x}_{k+1}) - \mathbf{f}(\mathbf{x}_k(t_{k+1})) - 2\frac{\mathbf{le}}{h_k^2}\right)\frac{(t-t_k)(t-t_{k+1})}{h_k^2} \\ &+ \left(\mathbf{f}(\mathbf{x}_{k+1}) - \mathbf{f}(\mathbf{x}_k(t_{k+1})) - 2\frac{\mathbf{le}}{h_k^2}\right)\frac{(t-t_k)^2}{h_k^2}\,.\end{aligned} \tag{13.12}$$

So, the residual $\Delta(t) = \dot{\mathbf{z}} - \mathbf{f}(\mathbf{z})$ is

$$\begin{aligned}\Delta(t) = &\frac{(t-t_k)(3t-2t_{k+1}-t_k)}{h_k^2}\left(\mathbf{f}(\mathbf{x}_{k+1}) - \mathbf{f}(\mathbf{x}_k(t_{k+1}))\right) \\ &+ \mathbf{f}(\mathbf{z}(t)) - \mathbf{f}(\mathbf{x}_k(t)) \\ &+ 6\frac{(t-t_k)(t_{k+1}-t)}{h_k^3}\mathbf{le}\,.\end{aligned} \tag{13.13}$$

Therefore, using the fact that ${}^{(t-t_k)}\!/_h = \theta \leq 1$, we find that

$$\begin{aligned} \Delta(t) = \theta(3\theta-2)\left[\mathbf{f}(\mathbf{x}_{k+1}) - \mathbf{f}(\mathbf{x}_k(t_{k+1}))\right] \quad &(13.14)\\ + \mathbf{f}(\mathbf{z}(t)) - \mathbf{f}(\mathbf{x}_k(t)) \quad &(13.15)\\ + 6\theta(1-\theta)\frac{\mathbf{le}}{h_k}\,. \quad &(13.16) \end{aligned}$$

We may bound each of the terms (13.14)–(13.16) in turn, using the definition of local error and the Lipschitz constant for $\|\mathbf{f}(\mathbf{a}) - \mathbf{f}(\mathbf{b})\| \leq L\|\mathbf{a}-\mathbf{b}\|$. Note also that $|\theta(3\theta-2)| \leq 1$ on $0 \leq \theta \leq 1$ and that $0 \leq \theta(1-\theta) \leq {}^1\!/_4$ on the same interval, and so

$$\begin{aligned} \|\Delta(t)\| \leq L\|\mathbf{x}_{k+1} - \mathbf{x}_k(t_{k+1})\| & \\ + L\|\mathbf{z}(t) - \mathbf{x}_k(t)\| & \\ + \frac{3}{2}\frac{\|\mathbf{le}\|}{h_k}\,. & \quad (13.17) \end{aligned}$$

By the definition of $\mathbf{z}(t)$, we also have

$$\mathbf{z}(t) - \mathbf{x}_k(t) = \frac{(t-t_k)^2}{h_k^2}\mathbf{le} + \left(\mathbf{f}(\mathbf{x}_{k+1}) - \mathbf{f}(\mathbf{x}_k(t_{k+1})) - \frac{2}{h_k}\mathbf{le}\right)\frac{(t-t_k)^2(t-t_{k+1})}{h_k^2}\,.$$

from which, using the Lipschitz condition and ${}^{\|\mathbf{le}\|}\!/_{h_k} \leq \varepsilon$ and $\theta^2(1-\theta) \leq {}^4\!/_{27}$, it follows that

$$\begin{aligned} \|\mathbf{z}(t) - \mathbf{x}_k(t)\| &\leq \|\mathbf{le}\| + \frac{4h_k}{27}\left(L\|\mathbf{x}_{k+1} - \mathbf{x}_k(t_{k+1})\| + 2\frac{\mathbf{le}}{h_k}\right) \\ &\leq \|\mathbf{le}\| + \frac{4}{27}(B+2)\|\mathbf{le}\|\,. \qquad (13.18) \end{aligned}$$

As a result, we finally obtain

$$\begin{aligned} \|\Delta(t)\| &\leq L\mathbf{le} + L\left(\mathbf{le} + \frac{4}{27}h_k\left(L\mathbf{le} + 2\frac{\mathbf{le}}{h_k}\right)\right) + \mathbf{le} + \frac{3}{2}\frac{\mathbf{le}}{h} \\ &\leq \left(h + \frac{3}{2} + \frac{62}{27}B + \frac{4}{27}B^2\right)\varepsilon, \end{aligned}$$

as desired. This completes the proof. ♮

This bound is apt to be pessimistic in most cases; admittedly the bound $h_n L_n \leq B$ might be inconvenient in practice as well. Nonetheless, it gives a clear rationale for expecting that controlling the local error per unit step will also control (possibly up to a problem-and-method-dependent constant B) the residual. In addition, note that this analysis is independent of the method used to generate the mesh or the solutions $\mathbf{x}_k$ at the mesh points or the method used to guarantee that $\|\mathbf{lepus}\| \leq \varepsilon$. This analysis works for one-step methods and for multistep methods (indeed for Taylor series methods and general multistep methods too).

Remark 13.3. If the local error is so deceitfully remote from what users want, why is it popular? The answer seems to be that the notion is convenient analytically. To see this, suppose, for instance, that we have an interpolant $\mathbf{x}(\theta)$ on $t_k \leq t \leq t_{k+1}$, where $0 \leq \theta \leq 1$ and $t = t_k + h_k\theta$. Suppose also that the interpolant has local error function $h^{p+1}\mathbf{E}_k(\theta)$; that is, $\mathbf{x}(\theta) = \mathbf{x}_k(\theta) + h^{p+1}\mathbf{E}_k(\theta)$. Strictly speaking, the local error is just the final value $h^{p+1}\mathbf{E}_k(1)$. Now Exercise 13.19 asks you to show that the residual is

$$\Delta(t) = \frac{1}{h_k}\mathbf{x}'(\theta) - \mathbf{f}(\mathbf{x}(\theta)) = h_k^p\mathbf{E}_k'(\theta) + O(h_k^{p+1})\,, \tag{13.19}$$

to leading order (the next term is also fairly simple to derive). Thus, if we have a formula for the asymptotics of the local error, then we can quite easily find a similar formula for the residual. It turns out that formulæ for the leading terms of the local error, while tedious, are usually not that hard to work out, whereas working directly with asymptotic formulæ for other quantities is less straightforward. ◁

Remark 13.4. As we have discussed, there is a connection between the global error (forward error) and the residual via the Gröbner–Alexeev nonlinear variation-of-constants formula. This relationship is easily adapted to find a similar relationship between the local error and the residual:

$$\varepsilon(t) = \int_{t_k}^{t} \mathbf{G}(t,\tau;\mathbf{x}_k)\Delta(\tau)\,d\tau\,. \tag{13.20}$$

For the simplest possible problem, $\dot{x}(t) = 1$, we have in fact that the local error is just

$$\varepsilon(t) = \int_{t_k}^{t} \Delta(\tau)\,d\tau, \tag{13.21}$$

and for this problem since $\|\varepsilon(t_{k+1})\| \leq h_k\|\Delta\|$, we have immediately that

$$\|\Delta\| \geq \frac{\mathbf{le}}{h_k}\,. \tag{13.22}$$

This shows that while the bound from the theorem can be tightened in many cases, the residual cannot, in general, be smaller than the local error per unit step. Thus, controlling the residual is in some sense equivalent to controlling the local error per unit step. ◁

13.2.3 Convergence and Consistency of Methods

We begin our discussion of convergence with a simple-looking initial-value problem, namely, the *Dahlquist test problem*:

$$\dot{x} = f(t,x) = \lambda x, \qquad x(0) = x_0\,. \tag{13.23}$$

We integrate on a definite time interval, say $0 \leq t \leq 1$. The analytic solution is $x(t) = x_0 e^{\lambda t}$. This apparently simple problem tells us a lot, as we will see. Certainly, any good method must necessarily do well on this problem. Of course, because this is a *linear* problem, it's not sufficient for a method to do well on this problem to be a "good" method, but it is necessary.

If we use Euler's method (with fixed step size) to tackle this problem, we can then give a general expression for the x_k:

$$x_{k+1} = x_k + hf(t_k, x_k) = x_k + h\lambda x_k = (1+h\lambda)x_k = (1+h\lambda)^k x_0 \,. \tag{13.24}$$

We take n steps, giving values x_k for $0 \leq k \leq n$. There are several parameters in that formula, and at least two "natural" limits to take: $h \to 0$, which means that $n \to \infty$ simultaneously because we must partition the interval $[0,1]$ into $0 = t_0 < t_1 < \cdots < t_n = 1$. We might also wish to worry about $\lambda \to \infty$ as well, which we postpone until the next section. We begin with the "most" natural, $h \to 0$. This corresponds to doing the computation over again, but with a smaller step size. As we said, if we wish to provide a solution on a finite interval (say $0 \leq t \leq 1$), then this also requires the second natural limit, $n \to \infty$, simultaneously. It does not materially affect the discussion to suppose that $h = {}^1\!/\!{}_n$, in fact.

We can now talk about the convergence of this process for fixed λ. We have

$$\lim_{n\to\infty} x_k = \lim_{n\to\infty} (1 + \frac{\lambda}{n})^k x_0 = \qquad x_0 \tag{13.25}$$

for any fixed k; this is clearly unsatisfactory. However, we are not limited to fixed k: We may look at $k = [tn]$, where t is some fixed fraction and $[a]$ means the greatest integer less than or equal to a, and this limit has $x_k \to \exp(\lambda t)x_0$ as desired.

Under what circumstances does the Euler method converge to the reference solution as $h \to 0$ and $n \to \infty$? The standard formal theory uses the following concept:

Definition 13.5 (Consistency of a method). A method is said to be *consistent* if

$$\lim_{h\to 0} \Delta(t) = 0 \,,$$

that is, if the residual of the method tends to 0 as $h \to 0$. ◁

Obviously, any method whose residual is the product of some positive power of h by some factor independent of h will be consistent (this is also true for other gauge functions in asymptotic series, of course). As we have seen in Theorem 13.1, this is the case for Euler's method, where $\|\Delta(t)\| \leq Lh_k$ on each interval.

The concept of consistency is not to be confused with the following concept:

Definition 13.6 (Convergence of a method). A method is said to be *convergent* if

$$\lim_{h\to 0} \|\mathbf{x}(t) - \mathbf{z}(t)\| = 0 \,,$$

that is, if the forward error of the method tends to 0 as $h \to 0$. ◁

Note that, as we have explained in Chap. 12, the forward error is bounded by the product of the condition number and the norm of the residual; that is,

$$\|\mathbf{x}(t) - \mathbf{z}(t)\| \leq \kappa \|\Delta(t)\| \, .$$

Therefore, if the method is consistent, then it will also be convergent, provided that the problem has a finite condition number.

Remark 13.5. Having a finite condition number is a weaker requirement than being "well-conditioned." This theorem says that Euler's method converges (theoretically) even for ill-conditioned problems, and so it does. In the literature, there is the related notion of a problem being "well-posed," meaning that the solution exists, is unique, and depends continuously on the initial data (and parameters). A well-conditioned problem is certainly well-posed, but some well-posed problems are not well-conditioned. To be well-conditioned, we need not only well-posedness, but also a condition number that is not only finite but also "small" or of "moderate size," relative to the errors in the problem context. ◁

13.3 Stiffness and Implicitness

As discussed in Sect. 12.7, stiff problems are, typically, well-conditioned problems for which explicit numerical methods surprisingly have to take small step sizes in order to maintain something referred to as "stability," even if the easy accuracy of a well-conditioned problem seems as though it should allow large step sizes. Consider again the example of Eq. (13.23) in the last section, but now think about $\lambda \to \infty$: When λ has negative real part, the reference solution $\exp(\lambda t)x_0$ will be monotonically decreasing for $t > 0$, more quickly decreasing if $\mathrm{Re}(\lambda)$ is more negative; and thus we would want our numerical solution $(1+h\lambda)^k x_0$ also to be monotonically decreasing as k increases, in this case. This requires $|1+h\lambda|$ to be smaller than 1, at the very least. This, in turn, requires $z = h\lambda$ to be in the interior of the disk of radius 1 centered at $z = -1$ (see Fig. 13.4), and in particular requires that $|h| < {}^{2Re(\lambda)}/_{|\lambda|^2}$ (see Problem 13.20). The larger $|\lambda|$ is [if $\mathrm{Re}(\lambda)$ is negative], the smaller the step size has to be, just to get a monotonic decrease (a qualitative feature of the correct solution; we are not even worried about quantitative accuracy). Because this restricts how large h can be, the method becomes extremely inefficient.

In Sect. 12.7, we also suggested that the cure to this problem is the use of implicit methods, which use the derivative at the as-yet unknown point $\mathbf{x}_{k+1}$, which then has to be solved for. So, let us consider a first-order implicit method, namely, Euler's implicit method:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{f}(t_{k+1}, \mathbf{x}_{k+1}) \, . \tag{13.26}$$

Note the appearance of $\mathbf{x}_{k+1}$ on both sides of the equation. Because of that, if $\mathbf{f}$ is nonlinear, then to find the unknown $\mathbf{x}_{k+1}$, we will have to solve nonlinear equations.

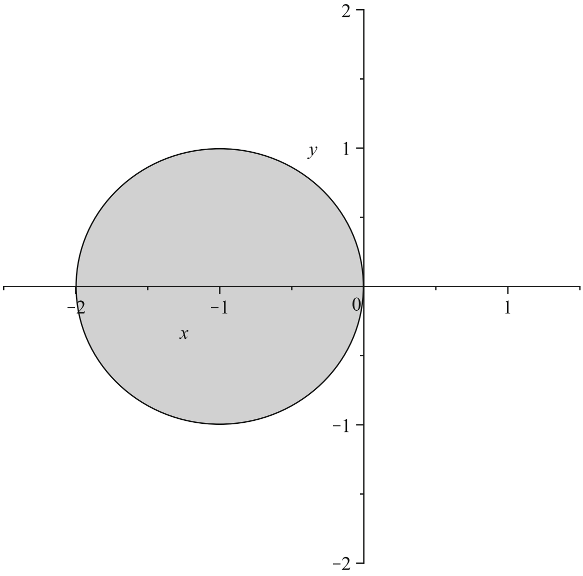


Fig. 13.4 The region in the $z = h\lambda$ plane (shaded) where forward Euler with fixed time step $h > 0$ applied to the scalar test problem $y' = \lambda y$ with $y(0) = y_0$ gives a computed solution that has a monotonic decrease as $k \to \infty$. The reference solution $\exp(\lambda t)y_0$ has a monotonic decrease for $t > 0$ if $\mathrm{Re}\lambda < 0$, which corresponds to the entire left half-z-plane, whereas the forward Euler method solution has monotonic decrease only for step sizes such that $z = h\lambda$ lies inside the shaded disk, or $h < 2^{|\mathrm{Re}(\lambda)|}/_{|\lambda|^2}$

To start with an easier problem, if we again consider the scalar linear example from Eq. (13.23), we obtain

$$x_{k+1} = x_k + hf(t_{k+1}, x_{k+1}) = x_k + h\lambda x_{k+1} \,.$$

Because this scalar equation is actually linear, we may solve for the unknown x_{k+1}:

$$x_{k+1} = \frac{1}{1-h\lambda} x_k = \left(\frac{1}{1-h\lambda}\right)^k x_0 \,.$$

The region of monotonic decrease for *this* method requires $|1 - h\lambda| > 1$. This is then the entire complex $z = h\lambda$ plane minus a disk of radius 1 centered at 1 (see Fig. 13.5).

Outside that disk, however large $\mathrm{Re}\lambda < 0$ is, and however large $h > 0$ is, we have monotonic decrease in the numerical solution. Indeed, we have a monotonic decrease even for a lot of $\mathrm{Re}\lambda > 0$, which is also a difficulty; we don't usually want unwarranted decay any more than we want unwarranted growth. Still, for well-conditioned problems, implicit Euler (also called backward Euler) produces the correct qualitative behavior.

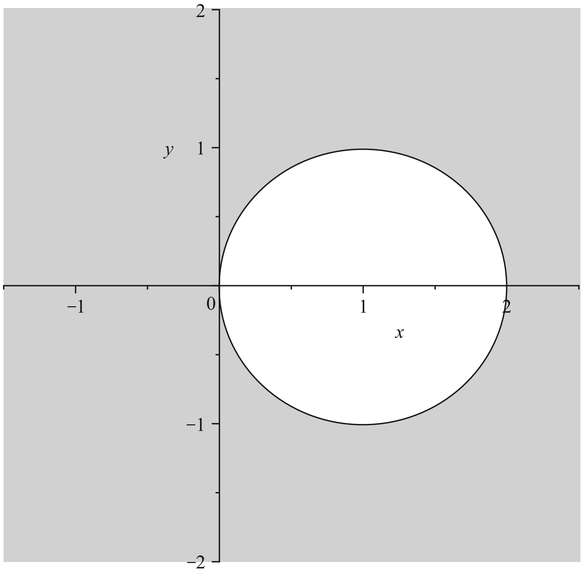


Fig. 13.5 The region in the $z = h\lambda$ plane (shaded) where backward Euler with fixed time step $h > 0$ applied to the scalar test problem $y' = \lambda y$ with $y(0) = y_0$ gives a computed solution that has a monotonic decrease as $k \to \infty$. The reference solution $\exp(\lambda t)y_0$ has a monotonic decrease for $t > 0$ if $\mathrm{Re}\lambda < 0$

However, we should note that the much larger region of stability does not justify using implicit methods by default on generic problems. For nonlinear problems with vector functions **f**, the cost of solving the system for the implicit value will be very high (in fact, solving the nonlinear systems for large h may be impossible in practice). Only seriously stiff problems justify the use of implicit methods from the point of view of efficiency. Hence, we need both explicit methods (for nonstiff problems) and implicit methods (for stiff problems).

Note that this example gives us grounds to draw somewhat general conclusions since, in practice, the numerical solution of a problem involves linearizing the equation, fixing the variable coefficients to some value of interest, and finally diagonalizing the system to obtain decoupled differential equations. But then, each of the decoupled equations will have the form of the equation in this example. Hence we may expect that this idea of preserving decrease will apply to more problems than just this simple scalar test problem.

The story above, explaining the success of fixed time-step implicit methods in terms of regions of stability is, however, not the whole story. Let us once again assume that we have linearized our problem, that we have fixed the coefficients, and that we have decoupled the equations. In this common context, fundamental solutions of differential equations have the form $e^{\lambda t}$, and the general solutions of the homogeneous part are linear combinations of such terms (as we have seen in Sect. 12.3.2). Now, we have seen that the explicit Euler method, in fact, corresponds to the two leading terms of a Taylor expansion. Moreover, as we will see later in the chapter, higher-order methods are also constructed so that they match the leading terms of the Taylor series. Accordingly, we will examine the accuracy of Taylor polynomials to approximate the exponential function.

To begin with, we observe the asymmetric relative accuracy δ_{exp} of a particular truncated Taylor series for e^x:

$$\begin{aligned} p(x) &= 1+x+\frac{x^2}{2}+\frac{x^3}{6}+\frac{x^4}{24}+\frac{x^5}{120} \\ \delta_{exp}(x) &= \frac{p(x)-e^x}{e^x} = p(x)e^{-x}-1\,. \end{aligned} \tag{13.27}$$

Note that $\delta_{exp}(-2.0) \doteq 0.508$ is more than 30 times as large as $\delta_{exp}(2.0) \doteq 0.016$. Accuracy is quite a bit better to the right, with a relative error of less than 2%, but more than 50% on the left. By the time we reach $x = 5$, the ratio is about 10,000. See Fig. 13.6.

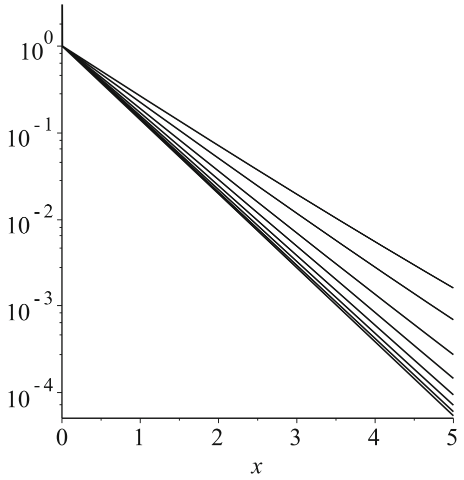


Fig. 13.6 Approximation of e^x by a Taylor polynomial has a larger relative error for negative x than it does for positive x. What is plotted here are the ratios of the relative errors on the right to the relative errors on the left, for degrees 2, 3, 5, ..., 34, and 55 Taylor approximations. *The smaller the ratio, the better the approximation is on the right and the more asymmetrical the quality of polynomial approximation.* The lower curves correspond to higher-degree approximations. As the degree increases, the relative benefit nearly stabilizes on this range, but independent of degree, the approximation is relatively far better for $x > 0$ than for $x < 0$. Notice that by $x = 5$, the high-order polynomial approximations are *four orders of magnitude better at representing growth than they are at representing decay*

This is simply a fact; a fact in the relationship between Taylor polynomials, which we use for numerical computation, and the exponential function, which we often use in applications. The reader is strongly urged to spend some time appreciating this asymmetry, which is simple but has far-reaching effects.

Figure 13.6 was generated by the MAPLE commands below:

```
ratiorelerr := n -> abs((convert(series(exp(x), x, n), polynom)*
    exp(-x)-1)/(convert(series(exp(-x), x, n), polynom)*exp(x)-1)
    ) ;
Digits := 200;
plots[logplot]([seq(ratiorelerr(k), k = [2, 3, 5, 8, 13, 21, 34,
    55])], x = 0 .. 5, colour = BLACK)
```

The more terms we keep in the series, the bigger the asymmetric factor is. Lower-order approximations are not quite as asymmetric; if we only keep $p(x) = 1 + x + x^2/2 + x^3/3!$, we have a factor of 24 difference. However, the asymmetry persists at all orders, as can be seen by series expansion of $\delta_{exp}(x)$, showing that the series has alternating signs and thus must be larger on the left, with negative x, when all the terms have the same sign.

What does this asymmetry mean? It means that Taylor polynomials are good at growing, but that they are not so good at flattening out. Now, e^x grows to the right (faster than polynomials can, it's true), and flattens out very quickly to the left. Fitting a polynomial at one point only, as we are doing with Taylor series, encounters this asymmetry. This phenomenon can be observed no matter about what point we expand e^x, as one can easily check. The polynomial approximations are going to have a better *relative* error on the growing side.

Naturally, if we examine e^{-x} and its corresponding Taylor polynomials, the "left" and "right" above will be interchanged. What does that mean for the example we have been examining? Suppose we integrate forward for positive time. On the one hand, if the problem is ill-conditioned, then λt is going to be positive. In this case, the Taylor polynomial will have a better relative residual on the right-hand side, as shown in Fig. 13.6. Thus, explicit methods based on Taylor series will do better than implicit methods. On the other hand, if the problem is well-conditioned, then λt will be negative. Accordingly, the situation will be reversed and the Taylor polynomial will have a better relative residual on the left-hand side. In this case, implicit methods based on Taylor series, generalizing Eq. (13.26), will do better than explicit methods.

Taylor series methods use information at z_n to predict, by polynomial approximation, z_{n+1}; and we will see that all other numerical methods are based on the Taylor series method. Foreshadowing a bit more, we will see that Runge–Kutta methods choose the coefficients in the tableau so that Taylor expansion of the stepping function agrees with Taylor expansion of the solution, for example. Multistep methods are also designed using Taylor series, as we will see. Thus, the fundamental choice of implicit methods or explicit methods seeks to exploit the asymmetric relative error behavior of approximation of the exponential function by polynomials. If we base our Taylor series on t_n, z_n, we are using polynomials to estimate exponential decay, which we noted doesn't work very well. If we base our Taylor expansion on (t_{n+1}, t_{n+1}), then, in compensation for the difficulty in finding z_{n+1}, we are using Taylor polynomials to estimate decay backward, that is, growth forward—which polynomials are better at.

The bottom line is that implicit methods, when the solution of the generally nonlinear equations defining z_{n+1} converges, gain a large factor improvement in the local error; this translates into a smaller "optimal" residual and hence to smaller forward error.

Other schemes for getting smaller residuals without having to solve nonlinear equations have been tried, for example, the use of rational functions as approximations, not just Taylor polynomials. However, such schemes have so far failed, owing to their own difficulties, such as introduction of spurious movable poles.

In many applications, the program is not allowed to return the error message "did not converge." The program is required to give you an answer. If you have a stiff system, and cannot use implicit methods because the nonlinear equations are too hard to solve, then Fig. 13.6 suggests that *low*-order methods will suffer less and should be considered. Indeed, if we scale by the number of function evaluations, forward Euler becomes competitive and can be made adaptive and effectively parallelized. See Christlieb et al. (2010) for a further improvement on that.

Remark 13.6. The previous discussion considered only fixed time-step methods. However, stiffness really shows up in adaptive methods when the step-size control forces the step size to be (in some sense) overly small, making a nonstiff method inefficient. That is not the only effect of variable step size. It has been observed that step-size control can *stabilize* methods, and indeed special step-size controls can be designed with this in mind. See Stuart and Humphries (1995). ◁

13.4 Taylor Series Method

We now look at the details of a natural generalization of Euler's method, namely, the use of local Taylor series, or what is known theoretically as *analytic continuation*. Let us first consider how Taylor series methods are used analytically to find solutions in terms of series, and then we will examine how they are implemented numerically. Consider the scalar initial-value problem

$$\dot{x}(t) = x(t)^2 - t, \qquad x(1) = 2\,. \tag{13.28}$$

Apart from the initial condition, this is the characteristic example of Hubbard and West (1991). Now, we suppose the existence of a solution $x(t)$ that can be written as a Taylor series about $t_0 = 1$:

$$\begin{aligned} x(t) &= x(t_0) + \dot{x}(t_0)(t-t_0) + \frac{\ddot{x}(t_0)}{2!}(t-t_0)^2 + \frac{\dddot{x}(t_0)}{3!}(t-t_0)^3 + \cdots \\ &= \sum_{k=0}^{\infty} \frac{x^{(k)}(t_0)}{k!}(t-t_0)^k. \end{aligned}$$

The Taylor series method consists of determining the coefficients $x^{(k)}(t_n)/k!$—we will denote them by $x_{n,k}$—in a recursive way. That is, given that we know $x(t_0)$

[henceforth denoted $x_{0,0}$] and that we know how to differentiate the differential equation, we can find all the coefficients $x_{n,k}$ automatically. Coming back to our example, we obtain $\dot{x}(t_0) = \dot{x}(1)$ by direct substitution in (13.28):

$$\dot{x}(1) = x(1)^2 - 1 = 2^2 - 1 = 3\,.$$

We can then differentiate the differential equation as many times as needed and then evaluate at $t_0 = 1$, for example,

$$\begin{aligned} \ddot{x}(t) &= 2x(t)\dot{x}(t) - 1 & \qquad \ddot{x}(1) &= 2\cdot 2\cdot 3 - 1 = 11 \\ \dddot{x}(t) &= 2(x(t)\ddot{x}(t) + \dot{x}(t)^2) & \qquad \dddot{x}(1) &= 2(2\cdot 11 + 3^2) = 62\,. \end{aligned}$$

Accordingly, the solution can be written as

$$x(t) = 2 + 3(t-1) + \frac{11}{2}(t-1)^2 + \frac{62}{6}(t-1)^3 + \cdots\,. \tag{13.29}$$

Now, associated with the Taylor series about t_n is a radius of convergence. Within the radius of convergence, we can use a Taylor polynomial containing the first N terms of the series to approximate $x(t)$ with a residual that is at most $O((t-t_n)^N)$, but outside the radius of convergence, we cannot make the error arbitrarily small. Accordingly, any time step we make must be no larger than the radius of convergence to give valid approximating results. We therefore let t_{n+1} be inside the circle of convergence, in order that we may find $x(t_{n+1})$ to the accuracy desired.

Moreover, we can then use this computed $x(t_{n+1})$ as a *new* initial value and expand $x(t)$ in a Taylor series about this point using the same procedure as last time. If this new expansion has a convergence disk overlapping with the original one [which it must if t_{n+1} was actually inside the previous circle of convergence], then the new disk can even allow us to advance outside the original convergence disk. See Fig. 13.7.

More generally, for an initial-value problem

$$\dot{x} = f(t, x(t))\,, \qquad x(t_0) = x_0\,, \qquad t_0 \leq t \leq t_N\,, \tag{13.30}$$

let us denote by

$$x_n(t) = \sum_{k=0}^{\infty} x_{n,k}(t-t_n)^k \tag{13.31}$$

the Taylor expansion of $x(t)$ about t_n [then, notice that $x_{0,0} = x_0(t_0) = x_0$]. For any t_{n+1} in the disk of convergence of this series, we may then find (in theory)

$$x_n(t_{n+1}) = \sum_{k=0}^{\infty} x_{n,k}(t_{n+1}-t_n)^k. \tag{13.32}$$

The *analytic continuation* idea is just to repeat this procedure: Let $x_n(t_{n+1}) = x_{n+1,0}$ be the new initial value and find a new Taylor series of the problem

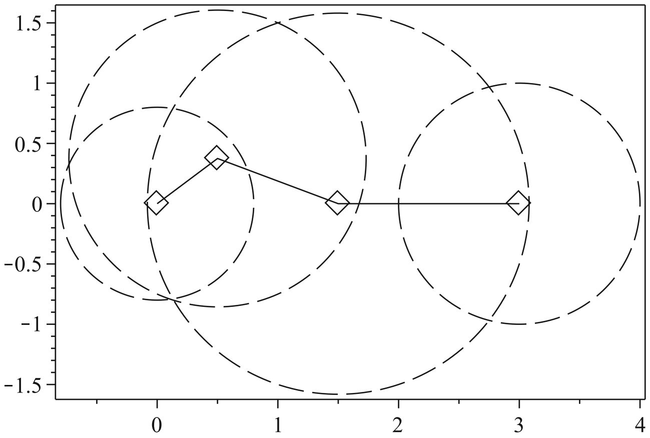


Fig. 13.7 Analytic continuation, namely, successive application of the Taylor series method. Each new circle's center must lie within the old circle, and disks must successively overlap; but provided there are not too many poles in the way, eventually one may break free and continue the knowledge of the function past the original radius of convergence. The successive centers of the circles, marked with diamond shapes, are at $[0, 1/2 + 3i/8, 3/2, 3]$. Poles were placed at -0.8, $1 + 3i/2$, and $3 + i$; the radius of convergence is the distance to the nearest pole

$$\dot{x} = f(t, x(t)), \qquad x(t_{n+1}) = x_{n+1,0} \tag{13.33}$$

and repeat. By piecing together these series, we find a complete solution along the path from t_0 to t_N.

In numerical practice, the Taylor series method uses not series but polynomials:

$$z_n(t) = \sum_{k=0}^{N} x_{n,k}(t - t_n)^k . \tag{13.34}$$

The resulting (absolute) residual about $t = t_n$ is then

$$\Delta_n(t) = \dot{z}_n - f(z_n) \tag{13.35}$$

$$= \sum_{k=1}^{N} k x_{n,k}(t - t_n)^{k-1} - f\left(\sum_{k=0}^{N} x_{n,k}(t - t_n)^k\right) \tag{13.36}$$

$$= r_N(t - t_n)^N + r_{N+1}(t - t_n)^{N+1} + \cdots . \tag{13.37}$$

To obtain an automatic numerical procedure, we need a way to find an iterative numerical substitute to repeatedly differentiating the differential equation. This is done by using the algebra of power series studied in Chap. 2.

Observe that we could think about this method in a dual way, which is the one used for computation. The presentation above *assumes* that the coefficients in the series representation of the solution are Taylor series coefficients (as in Eq. (13.31)), from which it results as a *matter of fact* that the first N coefficients of the residual are zero. Instead, we can merely assume that the solution is represented as a formal power series, and we make no assumption as to whether they are Taylor coefficients. Then, we can impose the *constraint* that the first N coefficients of the residual are zero, and conclude that the coefficients of the formal power series for the solution are indeed Taylor coefficients. This dual perspective suggests that the method might be called a "minimal residual formal power series method," where the "minimal" term means minimal in a norm that uses power series. But in any case, the notions are equivalent.

Algorithm 13.2 Sketch of explicit Taylor series algorithm for numerical analytic continuation to solve IVP

```
Require: A procedural description for the function $\mathbf{f}$ in $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$, a desired time span of integration $[t_0, t_F]$, an integration tolerance $\varepsilon$, and a vector of initial conditions.
  Use automatic differentiation tools to construct a procedure TaylorPoly that computes $N+1$ terms $\mathbf{c}_j$, $0 \le j \le N$, of the Taylor series of $\mathbf{x}(t)$ about $t = t_k$, given $\mathbf{c}_0$.
  Choose an initial stepsize $h_0$ and set $k = 0$.
  Choose an initial order $N$.
  while $t_k + h_k > t_k$ and $t_k + h_k \le t_F$ do do
    Construct $N+1$ terms $\mathbf{c}_j$ of the Taylor series about $t = t_k$.
    Put $\mathbf{x}_k(t) = \sum_{j=0}^{N} \mathbf{c}_j (t - t_k)^j$ .
    Estimate the residual (deviation), perhaps by evaluating $\Delta = \dot{\mathbf{x}}_k(t_k + h_k) - \mathbf{f}(\mathbf{x}(t_k + h_k))$.
    if $\|\Delta\| \le \varepsilon$ then {Accept the step}
      Predict a new step size, perhaps by $h_{k+1} = (\varepsilon / \|\Delta\|)^{1/N} h_k$, possibly with safety factors
    else {Reject the step}
      Reduce the step size $h_k = (\varepsilon / \|\Delta\|)^{1/N} h_k$ and try again {(notice that the Taylor coefficients need not be recomputed)}
    end if
    Adjust the order $N$. For example, if step sizes have been cycling or chattering, reduce the order; otherwise, increase it.
    Set $t_{k+1} = t_k + h_k$ and increment $k$.
  end while{If $h_k < h_{min}$, then we may have encountered a singularity.}
  return The piecewise function $\mathbf{x}(t) = \mathbf{x}_k(t)$ on each of $t_k \le t \le t_{k+1}$.
```

Example 13.2. Let us return to an example we discussed in Chap. 12:

$$\begin{aligned} \dot{S} &= -S^3 I, \qquad & S(0) &= 1 \\ \dot{I} &= -S I^2, \qquad & I(0) &= 1\,. \end{aligned} \tag{13.38}$$

We may solve this simple nonlinear autonomous system analytically to find that

$$S(t) = \frac{1}{1 + W(t)} \qquad \text{and} \qquad I(t) = e^{-W(t)}\,, \tag{13.39}$$

where W is the principal branch of the Lambert W function. Let us pretend that we don't know this and use the series method. To begin with, we let

$$S_n(t) = \sum_{k=0}^{N} S_{n,k}(t-t_n)^k \qquad \text{and} \qquad I_n(t) = \sum_{k=0}^{N} I_{n,k}(t-t_n)^k \tag{13.40}$$

be truncated power series approximating $S(t)$ and $I(t)$. We define auxiliary quantities to deal with SI, SI^2, S^2, and S^3I, since we will encounter these quantities in the evaluations of the derivatives $\dot{S}(t_n), \ddot{S}(t_n), \dddot{S}(t_n), \ldots$, and $\dot{I}(t_n), \ddot{I}(t_n), \dddot{I}(t_n), \ldots$ required to find the series coefficients. So, let

$$\begin{aligned} C = SI &\doteq \sum_{k=0}^{N} C_{n,k}(t-t_n)^k \\ D = SI^2 = CI &\doteq \sum_{k=0}^{N} D_{n,k}(t-t_n)^k \\ E = S^2 &\doteq \sum_{k=0}^{N} E_{n,k}(t-t_n)^k \\ F = S^3I = CE &\doteq \sum_{k=0}^{N} F_{n,k}(t-t_n)^k , \end{aligned}$$

where the coefficients satisfy the conditions for series multiplication:

$$\begin{aligned} C_{n,k} &= \sum_{j=0}^{k} I_{n,j} S_{n,k-j} \\ D_{n,k} &= \sum_{j=0}^{k} C_{n,j} I_{n,k-j} \\ E_{n,k} &= \sum_{k=0}^{k} S_{n,j} S_{n,k-j} \\ F_{n,k} &= \sum_{j=0}^{k} C_{n,j} E_{n,k-j} . \end{aligned}$$

Observe that these sums are just inner products, so they pose no numerical difficulty, as we will see below. The residuals in S and I, denoted Δ_S and Δ_I, are defined to be

$$\Delta_S = \dot{S} + S^3 I = \sum_{k=0}^{N} \Delta_{S,k}(t-t_n)^k + O(t-t_n)^{N+1} \tag{13.41}$$

$$\Delta_I = \dot{I} + SI^2 = \sum_{k=0}^{N} \Delta_{I,k}(t-t_n)^k + O(t-t_n)^{N+1} , \tag{13.42}$$

where the coefficients then satisfy

$$\Delta_{S,k} = (k+1)S_{n,k+1} + F_{n,k} \tag{13.43}$$

$$\Delta_{I,k} = (k+1)I_{n,k+1} + D_{n,k} \tag{13.44}$$

for all k, $0 \le k \le N$. Note that $S_{n,N+1} = I_{n,N+1} = 0$ because we truncate this series. As a result, for $0 \le k \le N-1$, we may set both $\Delta_{S,k}$ and $\Delta_{I,k}$ to zero as follows: Because both D_k and F_k are known once all $S_{n,j}$ and $I_{n,j}$ with $0 \le j \le k$ are known, we may simply set

$$S_{n,k+1} = -\frac{F_{n,k}}{k+1} \qquad \text{and} \qquad I_{n,k+1} = -\frac{D_{n,k}}{k+1}\,. \tag{13.45}$$

Then the first N coefficients of Δ_S and Δ_I are zeros and, consequently, the $S_{n,k}$ and $I_{n,k}$ in the truncated power series are the coefficients of the Taylor polynomials. So we have a recursive method in terms of power series to find the coefficients of the Taylor series without having to differentiate the differential equation directly. Note also that, since

$$\Delta_{S,k} = F_{n,k} \qquad k \ge N \tag{13.46}$$

$$\Delta_{I,k} = D_{n,k} \qquad k \ge N\,, \tag{13.47}$$

our procedure for computing Taylor series also automatically computes Taylor series for the residual. Thus, with little extra effort we will have an error estimate at hand. Of course, we may evaluate both $\Delta_S(t)$ and $\Delta_I(t)$ directly, just as easily, and this is usually best.

For our example, we can easily implement the scheme numerically in MATLAB. First, the coefficients are computed in a straightforward way with this compact code:

```
function [I,S] = tsw(I0,S0,N,sg)
% Taylor series coeffs of solution of S' = -S^3*I, I'= -S*I^2
% S(tn)=S0, I(tn)=I0; sg is the direction of integration.
% if tn=0, I0=S0=1, then I=exp(-W(t)) and S = 1/(1+W(t)).
    I = zeros(1,N+1);
    S = I; C = I; D = I; E = I; F = I;
    I(1) = I0; S(1) = S0;
    for k=1:N,
        C(k) = S(1:k)*I(k:-1:1).';
        D(k) = C(1:k)*I(k:-1:1).';
        E(k) = S(1:k)*S(k:-1:1).';
        F(k) = C(1:k)*E(k:-1:1).';
        S(k+1) = -sg*F(k)/k;
        I(k+1) = -sg*D(k)/k;
    end
end
```

Then the residual is also easily computed:

```
function [I,S,r,dt,It,St]=tswresid4text(In,Sn,N,sg)
[I,S]=tsw(In,Sn,N,sg); %Find the coefficients.
t=linspace(0,.5,256);
It=polyval(I(end:-1:1),t);
St=polyval(S(end:-1:1),t);
Std=polyval([N:-1:1].*S(end:-1:2),t);
Itd=polyval([N:-1:1].*I(end:-1:2),t);
r=[Itd+sg*St.*It.^2;Std+sg*St.^3.*It];
rsq = sqrt(r(1,:).^2+r(2,:).^2);
```

```
semilogy(t,abs(r(1,:)),'k-',t,abs(r(2,:)),'k-.')
%Find the numerical value dt at which the residual start to be bigger than the tolerance, here 1.0e-6, and evaluate the Taylor polynomial at this point to have new initial values.
ii=find(abs(rsq)>1.0e-6);
if numel((ii))>0,
    ig=ii(1)-1;
    if ig==0,
        error('failure',rsq(1))
    end
    dt = t(ig);
    It = It(ig);
    St = St(ig);
else
    dt = 1.0;
    It = It(end);
    St = St(end);
end
end
```

Let us take two steps explicitly with this method. For no particular reason, take $N = 7$ and start with $n = 0$ and $t_n = t_0 = 0$. Here $S(0) = s_{0,0} = 1$ and $I(0) = I_{0,0} = 1$. The program above gives

$$S_0(t) = 1 - t + 2t^2 - 4.5t^3 + 10.6667t^4 - 26.0417t^5 + 64.8t^6 - 163.4014t^7$$
$$I_0(t) = 1 - t + 1.5t^2 - 2.667t^3 + 5.2083t^4 - 10.8t^5 + 23.3431t^6 - 52.0127t^7,$$

where, as usual, we have printed only the default MATLAB representation of the coefficients; more decimal places are used internally than shown here.

By inspection of the graphs of (Fig. 13.8)

$$\Delta_S(t) = \dot{S} + S^3 I \qquad \text{and} \qquad \Delta_I(t) = \dot{I} + SI^2, \tag{13.48}$$

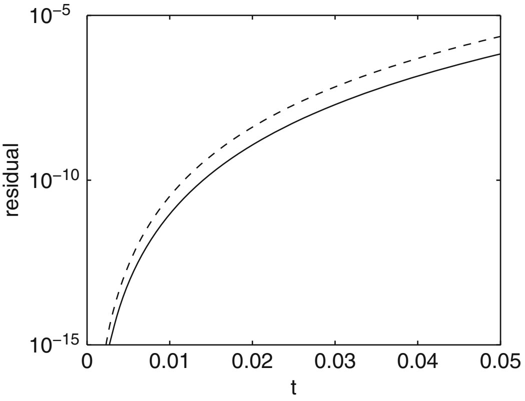


Fig. 13.8 Residual of the Taylor polynomials of S (*dotted line*) and I (*hard line*)

we see that $\sqrt{\Delta_S^2+\Delta_I^2} \le 10^{-6}$ if $0 \le t \le 0.0431$. We thus take $t_1 = 0.0431$ and evaluate the Taylor polynomials at this point, so that

$$S_{1,0} = S_0(0.0431) = 0.9603 \tag{13.49}$$

$$I_{1,0} = I_0(0.0431) = 0.9595 \tag{13.50}$$

are our initial values for the second step (these are the `St` and `It` in the program). We can then generate new truncated Taylor series about t_1:

$$S_1(t) = 0.9603 - 0.8495(t-t_1) + 1.5188(t-t_0)^2 + \ldots - 71.184(t-t_0)^7$$

$$I_1(t) = 0.9595 - 0.8840(t-t_1) + 1.2085(t-t_0)^2 + \ldots - 24.6896(t-0.0545)^7.$$

Again, the program indicates that $\sqrt{\Delta_{I_1}^2+\Delta_{S_1}^2}$ is smaller than 10^{-6} if $0 \le t - 0.0431 \le 0.0490$.

This process can obviously be repeated. *Provided that we can always take t_{n+1} so that $h_n = t_{n+1} - t_n$ is bounded below* by some minimum step size, say $\varepsilon_M t_n$, so that $t_{n+1} > t_n$ in floating-point arithmetic, we may integrate $\dot{x} = f(x)$ from t_0 to some fixed t_N by taking a finite number of steps of this method. At the end, we will have a mesh $t_0 < t_1 < t_2 < \ldots < t_{N-1} < t_N$ and a collection of polynomials $z_k(t)$ with residuals $\Delta_k(t)$ on $t_k \le t \le t_{k+1}$, where $\|r_k\|$ is at most our tolerance ε. All together, we will have a continuous (but not continuously differentiable) piecewise function $z(t)$ solving $\dot{x} = f(x) + \varepsilon v(t)$ and $x(t_0) = y_0$, with $\|v\|_\infty \le 1$.

The caveat, that we must be able to make progress, that is, $t_{n+1} > t_n$ in floating-point arithmetic, turns out to be interesting. In the exercises, you will be asked to solve (13.38). You should find that for tight tolerances you will *not* be able to get much past $t = 1/e \doteq 0.3679$. This is because the solution is singular there (more precisely, it has a derivative singularity). Location (or merely just detection) of singularities is an interesting topic, and useful in and of itself. We remark that the Taylor series method offers a way to detect such singularities essentially for free; if there is one, then keeping $\|\Delta_n\| \le \varepsilon$ ensures $h_n \to 0$. ◁

13.4.1 Implicit Taylor Series Methods

If we expand the solution of $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ about the known value x_n at $t = t_n$, we get the explicit Taylor series method just described. If, on the other hand, we expand the solution about the as-yet *unknown* x_{n+1} at $t = t_{n+1}$, and use that series to “retrodict” the known value x_n at $t = t_n$, we get an implicit method. We leave the development of this idea to the exercises, but we do one example to give the idea.

Example 13.3. Consider solving the Dahlquist test problem $y' = \lambda y$, $y(0) = y_0$, by using an implicit second-order Taylor series method. The series expansion

$$y(t) = y(t_{n+1}) + y'(t_{n+1})(t - t_{n+1}) + \tfrac{1}{2}y''(t_{n+1})(t - t_{n+1})^2/2 + \cdots$$

requires the computation of $y'(t_{n+1}) = \lambda y_{n+1}$ and of $y''(t_{n+1}) = \lambda y'(t_{n+1}) = \lambda^2 y_{n+1}$ (this is very easy for this example; for nonlinear problems and for systems, automatic differentiation is needed to generate the programs to evaluate all the derivatives). Thus, the approximate equality

$$\begin{aligned} y(t_n) &= y(t_{n+1}) + y'(t_{n+1})(t_n - t_{n+1}) + \tfrac{1}{2}y''(t_{n+1})\frac{(t_n - t_{n+1})^2}{2} \\ &= y_{n+1} - h\lambda y_{n+1} + h^2\frac{\lambda^2 y_{n+1}}{2} \end{aligned} \tag{13.51}$$

gives us an equation to solve for the unknown y_{n+1}. Again, this is easy in this case, because the equation is linear and scalar. We get, with $z = h\lambda$,

$$y_{n+1} = \frac{1}{1 - z + {}^{z^2}\!/_2} y_n \tag{13.52}$$

and it is no coincidence that this is one of the Padé approximants to the exponential function $\exp(z)$.

We may examine the stability region of this method—that is, for which z will the solution $y_n = (1 - z + {}^{z^2}\!/_2)^{-n} y_0$ decay monotonically as $n \to \infty$? Quite clearly, this will happen when $|1 - z + {}^{z^2}\!/_2| > 1$; this region is easy to draw (see Fig. 13.9). ◁

13.4.2 Final Remarks on Taylor Series Methods

There are freely available and commercial Taylor series codes. They are used especially for applications that require high accuracy or high dependability (in particular, they are used in interval methods, which provide guarantees of forward numerical accuracy). They are also used in noninterval computation. For instance, MAPLE offers `dsolve/numeric` with the `taylorseries` optional method, written by Allan Wittkopf, which works very well. A code for DAE, called DAETS, by Nedialkov and Pryce, is available for industrial applications. The package ATOMFT by Corliss and Chang is freely available. There is the interval ("Taylor model") code COSY by Berz and Makino (see `http://bt.pa.msu.edu/index_cosy.htm`). There is also a new second edition of the code TIDES available at `http://gme.unizar.es/software/tides`. Nonetheless, most industrial or other high-quality codes use other methods. We speculate that a set of historical, evolutionary traps locking in the results of earlier decisions have likely played a role, as follows.

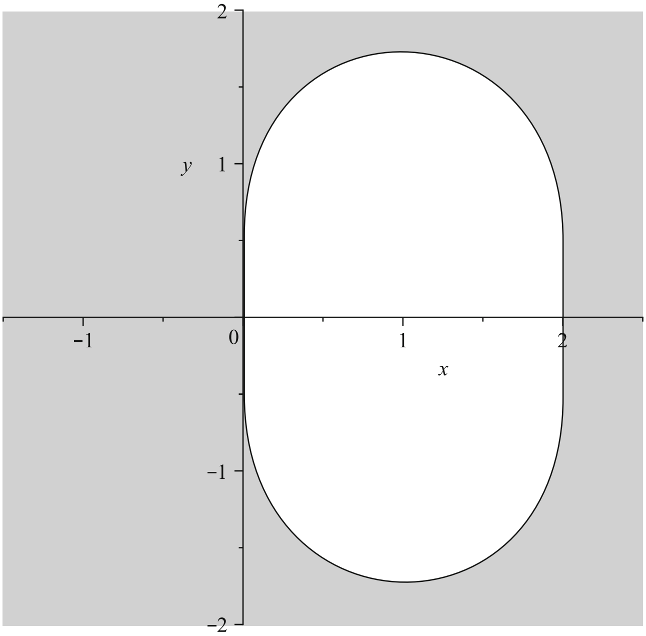


Fig. 13.9 The region in the $z = h\lambda$ plane (shaded), where the second-order implicit Taylor series method with fixed time step $h > 0$ applied to the scalar test problem $y' = \lambda y$ with $y(0) = y_0$, gives a computed solution that has a monotonic decrease as $k \to \infty$. The reference solution $\exp(\lambda t)y_0$ has a monotonic decrease for $t > 0$ if $\mathrm{Re}\lambda < 0$

In the early days of computation, there was neither computing time nor memory available to compute or represent interpolants: Numerical methods for ODE were expected to produce only a table of values at selected points (and that was hard enough, given limited hardware). There was also little understanding of the code generation needed for what is now called automatic differentiation—and symbolic differentiation, done badly, generates exponential growth in the length of expressions. A more interesting objection is that not all interesting functions have differential equations associated with them. For example, solving

$$\dot{y}(t) = \Gamma(t) + \frac{y(t)}{1 + \Gamma(t)} \tag{13.53}$$

by Taylor series needs special treatment because the derivatives of Γ are themselves special. However, in practice, this seems not to be an issue. Finally, and perhaps most important, many problems are not smooth, such as

$$\dot{x} = |1 - x^2|\,, \qquad x(0) = 0\,, \tag{13.54}$$

and so derivatives were set aside, as being too much work for too little gain.

Nonetheless, before moving on, let us recount the advantages of the Taylor series method:

1. It provides a free piecewise interpolant on the whole interval of integration.
2. It provides an easy estimate of the residual from the leading few terms in $\dot{x} - f(x)$ and an asymptotically accurate free estimate via $\dot{\mathbf{z}}(t_{k+1}^{-}) - \mathbf{f}(\mathbf{x}_{k+1})$.
3. It is a good tool for singularity detection and location.
4. It is flexible as to order of accuracy $N \geq 1$ and adaptive step size $h_n = t_{n+1} - t_n$.
5. Finally, but perhaps most importantly, it has been implemented.

All this is by the way. The fact is that people *did* turn away from Taylor series methods, not realizing their advantages, and invented several classes of beautiful alternatives. Using Taylor series methods as the underlying standard, we now discuss one such class of alternatives, the Runge–Kutta methods.

13.5 Runge–Kutta Methods

Marching methods, including Euler's method, Taylor series methods, and Runge–Kutta (RK) methods, share the following structure: Start at $\mathbf{x}_k$ and move to $\mathbf{x}_{k+1}$ by making a step of length h_k along a line whose slope approximates the slope of the secant connecting $\mathbf{x}_k$ and the desired $\mathbf{x}(t_{k+1})$. In the case of Euler's method, we simply used $\mathbf{f}(t_k, \mathbf{x}_k)$ as our approximate value for the slope of the secant. The idea of Taylor series methods was to improve the prediction by use of higher-order derivatives. In contrast, the idea of Runge–Kutta methods is not to take a second or higher derivative but rather to evaluate the derivative function $\mathbf{f}(t, \mathbf{x})$ more than once, at different points, in a manner reminiscent of finite differences; and from there to use a weighted average of the values thus obtained as an approximation of the slope of the secant. Then, depending on how many evaluations of $\mathbf{f}$ the method uses and on how well the weights of the average have been chosen, the methods so constructed will have a higher or lower order of accuracy. Let us consider a few examples.

13.5.1 Examples of Second-, Third-, and Fourth-Order RK Methods

If we consider adding a second evaluation of the function $\mathbf{f}$, then the natural thing to do is to compute it at the point $(t_{k+1}, \mathbf{x}(t_{k+1}))$, or rather at the best approximation to that point that we have available. This is exactly what the *Improved Euler method* does: Let

$$\mathbf{Y}_1 = \mathbf{x}_k + h\mathbf{f}(t_k, \mathbf{x}_k) \tag{13.55}$$

be the approximation we would get by a simple Euler step, and then take

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\left(\frac{1}{2}\mathbf{f}(t_k, \mathbf{x}_k) + \frac{1}{2}\mathbf{f}\big(t_{k+1}, \mathbf{Y}_1\big)\right). \tag{13.56}$$

This method just takes the arithmetic mean (i.e., the weights are $1/2$) of the slope at the beginning of the interval and at the end of it. Here rather than using an implicit method, we replace the exact point $(t_{k+1}, \mathbf{x}(t_{k+1}))$ by the approximation $(t_{k+1}, \mathbf{x}_k + h\mathbf{f}(t_k, \mathbf{x}_k))$. An alternative is to leave it as

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{h}{2}\left(\mathbf{f}(t_k, \mathbf{x}_k) + \mathbf{f}(t_{k+1}, \mathbf{x}_{k+1})\right), \tag{13.57}$$

which gives us an implicit system of equations to solve for the unknown $\mathbf{x}_{k+1}$; this implicit version of the improved Euler method has some advantages. For now, though, we consider the simple explicit version given previously.

Once we generate the skeleton of the solution, we need to find a way to interpolate the data generated in an appropriate manner. We not only need the interpolant for graphical output or for events, but also to make it possible to define and compute the residual. Because (as we will see) the method is second-order, simple linear interpolation as we used for the Euler method is not accurate enough.

Theorem 13.4. *The improved Euler method has second order accuracy.*

We will examine later how to construct *continuous* Runge–Kutta methods, and we will then be able to find the order of methods based on their residual. But since we do not have this available yet, we will show instead that the order of the local error of the improved Euler method is $O(h^3)$, which implies that this is an order-2 method.

Moreover, in this section, we will usually assume without loss of generality that the functions $\mathbf{f}$ are autonomous, since we can always rewrite a nonautonomous system as an autonomous system of higher dimension using the trick presented in Chap. 12. This assumption simplifies very much the Taylor series expansion. Occasionally, we will use the nonautonomous form simply for emphasis. Moreover, we will continue to simply write $\mathbf{f}$, $\mathbf{J}_\mathbf{f}$, and so forth, to denote the evaluation of those derivatives of $\mathbf{x}$ at $\mathbf{x}_k$.

Proof. First, notice that

$$\mathbf{f}(\mathbf{x}_k + h\mathbf{f}) = \mathbf{f} + h\mathbf{J}_\mathbf{f}\mathbf{f} + O(h^2),$$

so that the solution computed by the improved Euler method can be expanded as

$$\begin{aligned}\mathbf{x}_{k+1} &= \mathbf{x}_k + \frac{h}{2}\left(\mathbf{f} + \mathbf{f}(\mathbf{x}_k + h\mathbf{f})\right) = \mathbf{x}_k + \frac{h}{2}\left(\mathbf{f} + h\mathbf{J}_\mathbf{f}\mathbf{f} + O(h^2)\right)\\ &= \mathbf{x}_k + h\mathbf{f} + \frac{h^2}{2}\mathbf{J}_\mathbf{f}\mathbf{f} + O(h^3).\end{aligned}$$

Moreover, the local reference solution $\mathbf{x}_k(t_{k+1}) = \mathbf{x}_k(t_k + h)$ can be expanded about t_k as

$$\begin{aligned}\mathbf{x}_k(t_{k+1}) = \mathbf{x}_k(t_k + h) &= \mathbf{x}_k(t_k) + h\dot{\mathbf{x}}_k(t_k) + \frac{h^2}{2}\ddot{\mathbf{x}}_k(t_k) + O(h^3)\\ &= \mathbf{x}_k + h\mathbf{f} + \frac{h^2}{2}\mathbf{J_f}\mathbf{f} + O(h^3).\end{aligned}$$

As, a result, the local error $\mathbf{le} = \mathbf{x}_k(t_{k+1}) - \mathbf{x}_{k+1}$ is $O(h^3)$. Therefore, the method is second order. ♮

When we examine higher-order methods, we will need more machinery to deal with the Taylor series. Expanding vector-valued scalar functions in Taylor series does not require much notation beyond what is found in vector and matrix analysis. However, doing so for vector-valued vector functions requires the introduction of tensors. Even for this simple second-order method, if one tries to find an explicit expression for the first term of the local truncation error, we see that the matrix–vector notation is cumbersome (although it can be used, with a lot of patience). For higher-order terms, however, it becomes essential to use tensor notation. So, for now, we will simply present a third- and a fourth-order method without providing any proof.

Here's how a standard third-order RK method works. We use $\mathbf{k}_i$ for the various values of $\mathbf{f}$ we compute:

$$\begin{aligned}t_1 &= t_0 + h\\ \mathbf{k}_1 &= \mathbf{f}(t_0, \mathbf{x}_0)\\ \mathbf{k}_2 &= \mathbf{f}(t_0 + h, \mathbf{x}_0 + h\mathbf{k}_1)\\ \mathbf{k}_3 &= \mathbf{f}\left(t_0 + \frac{h}{2}, \mathbf{x}_0 + \frac{h}{2}\left(\frac{\mathbf{k}_1 + \mathbf{k}_2}{2}\right)\right)\\ x_1 &= \mathbf{x}_0 + \frac{h}{6}\left(\mathbf{k}_1 + \mathbf{k}_2 + 4\mathbf{k}_3\right).\end{aligned} \tag{13.58}$$

An alternative notation that is sometimes better is to use $\mathbf{Y}_j$ for the arguments to $\mathbf{f}$: namely, $\mathbf{k}_j = \mathbf{f}(\mathbf{Y}_j)$. In that notation, the above method can be written (for an autonomous problem this time)

$$\begin{aligned}\mathbf{Y}_1 &= \mathbf{x}_0 + h\mathbf{f}(\mathbf{x}_0)\\ \mathbf{Y}_2 &= \mathbf{x}_0 + h\mathbf{f}(\mathbf{Y}_1)\\ \mathbf{Y}_3 &= \frac{1}{2}\left(\mathbf{Y}_1 + \mathbf{Y}_2\right)\\ \mathbf{Y}_4 &= \mathbf{x}_0 + h\mathbf{f}(\mathbf{Y}_3)\\ \mathbf{x}_1 &= \frac{1}{3}\mathbf{Y}_3 + \frac{2}{3}\mathbf{Y}_4\,.\end{aligned} \tag{13.59}$$

The evaluations of $\mathbf{f}$ are called *stages*. In either the k-notation or the $\mathbf{Y}$-notation, each time step takes three stages with this method. Each stage of this method is illustrated in Fig. 13.10. Note that in the case in which $\mathbf{f}$ depends only on t, and not on $\mathbf{x}$ (i.e., the case of a simple quadrature), the problem amounts to $\mathbf{x}(t) = \int_{t_0}^{t_1} \mathbf{f}(t)dt$, so that the method in effect is the very same as Simpson's rule.

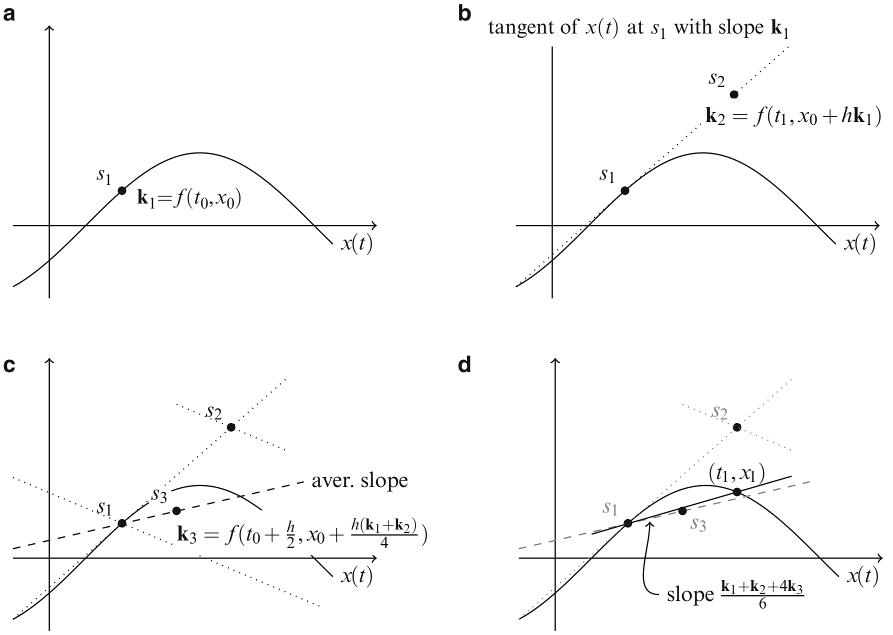

a
s_1
$\mathbf{k}_1{=}f(t_0,x_0)$
$x(t)$
b
tangent of $x(t)$ at s_1 with slope $\mathbf{k}_1$
s_2
$\mathbf{k}_2 = f(t_1, x_0 + h\mathbf{k}_1)$
s_1
$x(t)$
c
s_2
aver. slope
s_3
s_1
$\mathbf{k}_3 = f(t_0 + \frac{h}{2}, x_0 + \frac{h(\mathbf{k}_1+\mathbf{k}_2)}{4})$
$x(t)$
d
s_2
(t_1, x_1)
s_1
s_3
slope $\frac{\mathbf{k}_1+\mathbf{k}_2+4\mathbf{k}_3}{6}$
$x(t)$

Fig. 13.10 A graphical interpretation of RK3, a third-order Runge–Kutta method. (**a**) Stage 1. We compute $\mathbf{k}_1 = f(t_0, x_0)$ at the point $s_1 = (t_0, x_0)$. (**b**) Stage 2. We move h along the tangent, whose slope is $\mathbf{k}_1$, to the point $s_2 = (t_1, x_0 + h\mathbf{k}_1)$. Then, we compute $\mathbf{k}_2 = f(t_1, x_0 + h\mathbf{k}_1)$. (**c**) Stage 3. The slope at s_2 is $\mathbf{k}_2$. We come back to s_1 and move $h/2$ on a straight line whose slope is $(\mathbf{k}_1 + \mathbf{k}_2)/2$, the average of $\mathbf{k}_1$ and $\mathbf{k}_2$ to the point $s_3 = (t_0 + h/2, x_0 + (h/2)((\mathbf{k}_1 + \mathbf{k}_2)/2))$. (**d**) We make the step h with the weighted average of $\mathbf{k}_1, \mathbf{k}_2$ and $\mathbf{k}_3$. The resulting point is (t_1, x_1)

Here is a fourth-order Runge–Kutta method:

$$
\begin{aligned}
t_1 &= t_0 + h \\
\mathbf{k}_1 &= \mathbf{f}(t_0, \mathbf{x}_0) \\
\mathbf{k}_2 &= \mathbf{f}(t_0 + \frac{h}{2}, \mathbf{x}_0 + \frac{h}{2}\mathbf{k}_1) \\
\mathbf{k}_3 &= \mathbf{f}(t_0 + \frac{h}{2}, \mathbf{x}_0 + \frac{h}{2}\mathbf{k}_2) \\
\mathbf{k}_4 &= \mathbf{f}(t_0 + h, \mathbf{x}_0 + h\mathbf{k}_3) \\
\mathbf{x}_1 &= \mathbf{x}_0 + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)
\end{aligned}
\tag{13.60}
$$

In fact, this method is one of the most popular[7] Runge–Kutta methods. It is often simply referred to as RK4 or "the classical Runge–Kutta method." See Problem 13.7 for an example of where it fails.

With these representative examples, it should be clear what strategy Runge–Kutta methods exploit. To continue our investigation, we will now introduce a general notation for the Runge–Kutta methods, and then return to the mechanics of constructing the methods.

13.5.2 Generalization and Butcher Tableaux

We have seen two examples of Runge–Kutta methods above, in (13.58) and (13.60). Other simple methods include the following:

$$\begin{aligned}
&\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{f}(t_k, \mathbf{x}_k) = \mathbf{x}_k + h\mathbf{k}_1 && \text{Euler}\\
&\mathbf{x}_{k+1} = \mathbf{x}_k + h(\frac{1}{2}\mathbf{k}_1 + \frac{1}{2}\mathbf{k}_2) && \text{Trapezoidal}\\
&\qquad \text{with} \quad \mathbf{k}_2 = \mathbf{f}(t_k + h, \mathbf{x}_k + h\mathbf{k}_1) &&\\
&\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{k}_2 && \text{Midpoint}\\
&\qquad \text{with} \quad \mathbf{k}_2 = \mathbf{f}(t_k + \frac{1}{2}h, \mathbf{x}_k + \frac{1}{2}h\mathbf{k}_1). &&
\end{aligned}$$

The trapezoidal method is the one we called "improved Euler" before. We observe that each of these five methods consists of computing a number of slopes at different points and taking a weighted average of them, where the number s of slopes $\mathbf{k}_i$ computed corresponds to the number of stages of the method. Thus, if we let the weights be b_i, $i = 1, 2, \ldots, s$, the general form for these rules is

$$\mathbf{x}_{k+1} = \mathbf{x}_0 + h(b_1\mathbf{k}_1 + \ldots + b_s\mathbf{k}_s) = \mathbf{x}_k + h\sum_{i=1}^{s} b_i\mathbf{k}_i\,. \tag{13.61}$$

Moreover, we observe that the computation of some of the $\mathbf{k}_i$ depends on other values $\mathbf{k}_j$ (in the explicit cases considered here, only for $j < i$). For instance, in the trapezoidal rule, $\mathbf{k}_2$ depends on $\mathbf{k}_1$ for the value of the second variable in $\mathbf{f}(t, \mathbf{x}(t))$. The parameters indicating how much weight the previous steps j have in finding the new point of evaluation of f to determine $\mathbf{k}_i$ are denoted a_{ij}. Moreover, as we have seen in the midpoint rule, for instance, there is a constant dictating how big a time step we take. Thus, for explicit methods, we get the general form:

[7] Not because it *should* be used, and in fact there are good reasons not to use it; but it's usually the first fourth-order method people learn.

$$\mathbf{k}_1 = \mathbf{f}(t_k, \mathbf{x}_k) \tag{13.62}$$

$$\mathbf{k}_2 = \mathbf{f}(t_k + c_2 h, \mathbf{x}_k + a_{21}\mathbf{k}_1) \tag{13.63}$$

$$\mathbf{k}_3 = \mathbf{f}(t_k + c_3 h, \mathbf{x}_k + a_{31}\mathbf{k}_1 + a_{32}\mathbf{k}_2) \tag{13.64}$$

$$\vdots$$

$$\mathbf{k}_s = \mathbf{f}(t_k + c_s h, \mathbf{x}_k + a_{s1}\mathbf{k}_1 + a_{s2}\mathbf{k}_2 + \ldots + a_{s,s-1}\mathbf{k}_{s-1}). \tag{13.65}$$

Using indices, for explicit methods, we generally have

$$\mathbf{k}_i = \mathbf{f}(t_k + c_i h, \mathbf{x}_k + h\sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j)\,. \tag{13.66}$$

Since we always begin with an evaluation of the slope at $(t_k, \mathbf{x}_k)$, we have $c_1 = 0$. Moreover, since the evaluation of $\mathbf{k}_i$ cannot depend on previously computed values of $\mathbf{k}_i$ in an explicit method, we have $a_{1,i} = 0, i = 1, 2, \ldots, s$. In the trapezoidal rule, we have $c_2 = 1$ and $a_{21} = 1$. In the midpoint rule, we have $c_2 = {}^1\!/\!{}_2$ and $a_{21} = {}^1\!/\!{}_2$.

As we see, the weights b, the size of time steps c, and the weights a_{ij} of previously computed values of $\mathbf{k}_j$, $j < i$, fully determine an explicit Runge–Kutta method. This information for explicit methods can conveniently be summarized in a tableau, called a *Butcher tableau*, having the following form:

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^T \end{array} \quad = \quad \begin{array}{c|ccccc} 0 & 0 & 0 & 0 & \cdots & 0 \\ c_2 & a_{21} & 0 & 0 & \cdots & 0 \\ c_3 & a_{31} & a_{32} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ c_2 & a_{s1} & a_{s2} & \ldots & a_{s,s-1} & 0 \\ \hline & b_1 & b_2 & b_3 & \cdots & b_s \end{array}\,, \tag{13.67}$$

where $\mathbf{A}$ is a lower-triangular matrix with zeros as diagonal entries. We typically leave the 0s out, leaving the cells containing zero blank.

As one might expect, an implicit Runge–Kutta method (called an IRK method) of the form

$$\mathbf{k}_1 = \mathbf{f}(t_k + c_1 h, \mathbf{x}_k + a_{11}\mathbf{k}_1 + a_{12}\mathbf{k}_2 + \ldots + a_{1,s-1}\mathbf{k}_{s-1} + a_{1,s}\mathbf{k}_s) \tag{13.68}$$

$$\mathbf{k}_2 = \mathbf{f}(t_k + c_2 h, \mathbf{x}_k + a_{21}\mathbf{k}_1 + a_{22}\mathbf{k}_2 + \ldots + a_{2,s-1}\mathbf{k}_{s-1} + a_{2,s}\mathbf{k}_s) \tag{13.69}$$

$$\mathbf{k}_3 = \mathbf{f}(t_k + c_3 h, \mathbf{x}_k + a_{31}\mathbf{k}_1 + a_{32}\mathbf{k}_2 + \ldots + a_{3,s-1}\mathbf{k}_{s-1} + a_{3,s}\mathbf{k}_s) \tag{13.70}$$

$$\vdots$$

$$\mathbf{k}_s = \mathbf{f}(t_k + c_s h, \mathbf{x}_k + a_{s1}\mathbf{k}_1 + a_{s2}\mathbf{k}_2 + \ldots + a_{s,s-1}\mathbf{k}_{s-1} + a_{ss}\mathbf{k}_s) \tag{13.71}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\sum_{i=1}^{s} b_i\mathbf{k}_i \tag{13.72}$$

would have a full Butcher tableau:

$$\begin{array}{c|cccc}
c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\
c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}\,. \tag{13.73}$$

Let us illustrate this notation by giving the Butcher tableau of some methods we have encountered. The explicit midpoint rule has the tableau

$$\begin{array}{c|cc}
0 & & \\
{}^1\!/_2 & {}^1\!/_2 & \\
\hline
 & 0 & 1
\end{array}\,. \tag{13.74}$$

The *implicit* midpoint rule $\mathbf{x}_{k+{}^1\!/_2} = \mathbf{x}_k + {}^{h\mathbf{f}(\mathbf{x}_{k+1/2})}\!/_2$, or equivalently, $\mathbf{k}_1 = \mathbf{f}(\mathbf{x}_k + {}^{h\mathbf{k}_1}\!/_2)$, on the other hand, needs only a one-by-one matrix $\mathbf{A}$:

$$\begin{array}{c|c}
{}^1\!/_2 & {}^1\!/_2 \\
\hline
 & 1
\end{array}. \tag{13.75}$$

The explicit trapezoidal rule has the tableau

$$\begin{array}{c|cc}
0 & & \\
1 & 1 & \\
\hline
 & {}^1\!/_2 & {}^1\!/_2
\end{array}\,. \tag{13.76}$$

Finally, the classical Runge–Kutta method RK4 has the following Butcher tableau:

$$\begin{array}{c|cccc}
0 & & & & \\
{}^1\!/_2 & {}^1\!/_2 & & & \\
{}^1\!/_2 & 0 & {}^1\!/_2 & & \\
1 & 0 & 0 & 1 & \\
\hline
 & {}^1\!/_6 & {}^1\!/_3 & {}^1\!/_3 & {}^1\!/_6.
\end{array} \tag{13.77}$$

13.5.3 How to Construct a Discrete RK Method

This brings us at last to the beautiful theory of order conditions for Runge–Kutta methods. What follows, here, is only a gentle introduction to the primary references and is not intended to summarize or replace any of that material. Our goal is simply motivation and introduction. In this section, we consider, without loss of generality because of Eq. (12.5), only autonomous problems $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ because it makes the algebra simpler.

We now introduce the traditional strategy to construct a discrete Runge–Kutta method, that is, a method that returns a discrete set of points $\mathbf{x}_0, \mathbf{x}, \ldots, \mathbf{x}_N$ and the corresponding mesh points (what we have called the "skeleton" of the solution). However, this is just a transition stage we make in order to introduce the strategy for the construction of a *continuous* Runge–Kutta method in a didactically acceptable way.

The traditional strategy goes as follows. We begin by specifying the order p of the method we want to build, as well as the number s of stages we allow (there is, however, a minimum number of stages required to build a method of given order). Next, Taylor expanding the local reference solution $\mathbf{x}_k(t)$ about t_k and putting $h = t_{k+1} - t_k$, we get

$$\mathbf{x}_k(t_{k+1}) = \mathbf{x}_k + h\mathbf{f} + \frac{h^2}{2}(\mathbf{f}_t + \mathbf{J}_\mathbf{f}\mathbf{f}) + O(h^3) \tag{13.78}$$

up to (one more than) the desired order. Then, we Taylor expand the proposed Runge–Kutta method; this requires Taylor expansion of each of the stages $\mathbf{k}_j$ as follows:

$$\begin{aligned}\mathbf{k}_j &= \mathbf{f}\left(t_k + c_j h, \mathbf{x}_k + h\sum_{\ell=0}^{s} a_{j,\ell}\mathbf{k}_\ell\right) \\ &= \mathbf{f}(t, \mathbf{x}_k) + h \cdot () + \cdots .\end{aligned} \tag{13.79}$$

These Taylor coefficients involve the c_j and the $a_{i,j}$. We also Taylor expand the Runge–Kutta step itself. These coefficients involve the b_i, linearly.

We then equate the Taylor series: If our RK solution is to have local error of the desired order, the local Taylor series of the RK solution must be the same as the Taylor series of the local reference solution. This constraint gives us a sequence of equations containing the unknown as, bs, and cs. Then we solve for the b_i, c_i, and $a_{i,j}$. For higher-order methods, this procedure is very much more easily said than done.

Let us give the simplest nontrivial example: $p = 2$ and $s = 2$, that is, a second-order method with two stages. For $s = 2$, the general explicit Runge–Kutta method is given by

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\sum_{i=1}^{s} b_i\mathbf{k}_i = \mathbf{x}_k + h(b_1\mathbf{k}_1 + b_2\mathbf{k}_2)\,, \tag{13.80}$$

where the stages are $\mathbf{k}_1 = \mathbf{f}(t_k, \mathbf{x}_k) = \mathbf{f}$ and

$$\mathbf{k}_2 = \mathbf{f}(t_k + c_2 h, \mathbf{x}_k + ha_{2,1}\mathbf{k}_1)\,. \tag{13.81}$$

The first step in the construction is to expand the $\mathbf{k}_i$ (here, we have only one nontrivial one) in Taylor series about t_k:

$$\mathbf{k}_2 = \mathbf{f} + hc_2\mathbf{f}_t + ha_{21}\mathbf{J}_\mathbf{f}\mathbf{k}_1 + O(h^2) = \mathbf{f} + hc_2\mathbf{f}_t + ha_{21}\mathbf{J}_\mathbf{f}\mathbf{f} + O(h^2)\,. \tag{13.82}$$

We then substitute in Eq. (13.80):

$$\begin{aligned}\mathbf{x}_{k+1} &= \mathbf{x}_k + h\left(b_1\mathbf{f} + b_2\left(\mathbf{f} + hc_2\mathbf{f}_t + ha_{21}\mathbf{J}_\mathbf{f}\mathbf{f} + O(h^2)\right)\right) \\ &= \mathbf{x}_k + h\mathbf{f}(b_1 + b_2) + h^2(b_2c_2\mathbf{f}_t + b_2a_{21}\mathbf{J}_\mathbf{f}\mathbf{f}) + O(h^3)\,. \end{aligned} \tag{13.83}$$

Finally, we match the coefficients of the local reference solution (13.78) and of the numerical solution (13.83) [remembering that $\mathbf{x}_k(t_k) = \mathbf{x}_k$], to get a set of constraints for the b_i, c_i, and a_{ij}. In this case, we find that

$$1 = b_1 + b_2 \tag{13.84}$$

$$\frac{1}{2} = b_2c_2 \tag{13.85}$$

$$\frac{1}{2} = b_2a_{21}\,. \tag{13.86}$$

These equations are called *order conditions*. Now, since we have only three equations for four unknowns, there is a free degree of freedom; that is, there is an infinite family of two-stage order-2 Runge–Kutta methods.[8] The popular second-order methods we have examined before fall in this category. Letting b_2 be our free parameter, we find that $c_2 = a_{2,1} = {}^1\!/_{2b_2}$ and $b_1 = 1 - b_2$.

Setting $b_2 = {}^1\!/_2$ gives the trapezoidal method. Setting $b_2 = 1$ gives the midpoint method. Setting $b_2 = {}^1\!/_{2\varepsilon}$ and letting $\varepsilon \to 0^+$ (a silly thing to do numerically) gives the method

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{k}_1 + h\frac{\mathbf{k}_2 - \mathbf{k}_1}{2\varepsilon}\,, \tag{13.87}$$

which looks strange until you realize that in the limit as $\varepsilon \to 0^+$, because $\mathbf{k}_1 = \mathbf{f}(\mathbf{x}_k + \varepsilon h\mathbf{k}_1)$, the second term approaches ${}^{h^2\ddot{\mathbf{x}}}\!/_2$, which is the second-order Taylor series method.

13.5.4 Investigation of Continuous Explicit Runge–Kutta Methods

We now switch gears. The idea of a continuous explicit Runge–Kutta method, CERK for short, is quite natural. We already have the idea of generating a discrete set of points $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n$ on a mesh $t_0, t_, \ldots, t_N$ with a Runge–Kutta method. Moreover, we have already seen the importance of somehow interpolating our discrete set

[8] We won't discuss it much in this book, but as the desired order grows, the required number of order conditions grows *faster*. This means that we have to add more stages, that is, more unknowns, just to keep up. There is some complicated redundancy, however, so predicting the exact number of stages needed to get a Runge–Kutta method of a desired order is not an easy problem; for high orders, only bounds are known—see Butcher (2008b).

of points in order to be able to evaluate and interpret the results. The idea of CERKs is then: Instead of doing this in two separate operations, why not combine the ideas and do it in one?

How are we to do this? Following Hairer et al. (1993), we simply do as we did in the last subsection, namely, expand in Taylor series and match coefficients to find order conditions restricting the parameters of the method, except that we let the *weights* b_i be variable (in fact, polynomial) and range over a subinterval $[t_k, t_{k+1}]$. We have already used the notation $\theta = {}^{(t-t_k)}\!/_{h_k}$ before, in order to describe Euler's method interpolated piecewise linearly in Eq. (13.5). Adapting this equation to our current Runge–Kutta notation, we find

$$\mathbf{z}_k(t) = \mathbf{x}_k + h_k\theta_k\mathbf{f}(t_k, \mathbf{x}_k) \quad \xrightarrow{CERKing} \quad \mathbf{z}(\theta) = \mathbf{x}_k + hb_1(\theta)\mathbf{k}_1\,,$$

where $b_1(\theta) = \theta = {}^{(t-t_k)}\!/_{h_k}$. In general, for the construction of a CERK, as opposed to a discrete Runge–Kutta method, the rule generating the points $\mathbf{x}_i$ will have the form

$$\mathbf{z}(\theta) = \mathbf{x}_k + h\sum_{i=1}^{s} b_i(\theta)\mathbf{k}_i\,. \tag{13.88}$$

Note also that where we used the notation $\mathbf{z}_k(t)$, a function of t indexed for subintervals, we now use the variable θ and drop the index, since it is already built in $\theta = {}^{(t-t_k)}\!/_{h_k}$. Also, note that in addition to the order conditions found by the process described in the last section, we also impose other helpful constraints on the constants a_{ij}, such as

$$c_i = \sum_{j=1}^{i-1} a_{ij}\,, \tag{13.89}$$

where, remember, $c_1 = 0$. If this is true, the differential equation $\dot{x} = 1$ will be integrated exactly, and we assume it is true henceforth.

When constructing a continuous Runge–Kutta method, we will again choose an order p for the method and a number of stages s. In this context, the residual is

$$\Delta(t) = \frac{d}{dt}\mathbf{z}(\theta) - \mathbf{f}(\mathbf{z}(\theta))\,. \tag{13.90}$$

Since $\theta = {}^{(t-t_n)}\!/_{h}$, this gives

$$\Delta(\theta) = \frac{1}{h}\mathbf{z}'(\theta) - \mathbf{f}(\mathbf{z}(\theta))\,, \tag{13.91}$$

where the prime now denotes differentiation with respect to θ. By definition of order, the choice of order imposes a constraint on the residual of the method defined in (13.88), namely, is required that $\Delta(t) = O(h^p)$.

As an example, consider again the case of a second-order two-stage method:

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(\mathbf{x}_k) \\
\mathbf{k}_2 &= \mathbf{f}(t_k + c_2 h, \mathbf{x}_k + h a_{21}\mathbf{k}_1) \\
\mathbf{z}(\theta) &= \mathbf{x}_k + h\left(b_1(\theta)\mathbf{k}_1 + b_2(\theta)\mathbf{k}_2\right) .
\end{aligned} \tag{13.92}$$

To begin with, note that the Taylor series of the local reference solution is

$$\mathbf{x}_k(t_k + \theta h) = \mathbf{x}_k + h\theta\mathbf{k}_1 + \frac{h^2\theta^2}{2}(\mathbf{f}_t + \mathbf{J}_\mathbf{f}\mathbf{f}) + O(h^3)\,. \tag{13.93}$$

In order to match the coefficients of $\mathbf{z}(\theta)$ with the local reference solution, we first expand $\mathbf{k}_2$:

$$\mathbf{k}_2 = \mathbf{f} + c_2 h\mathbf{f}_t + h a_{21}\mathbf{J}_\mathbf{f}\mathbf{k}_1 + O(h^2). \tag{13.94}$$

As a result, the computed continuous solution has the form

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{k}_1(b_1(\theta) + b_2(\theta)) + h^2(b_2(\theta)c_2\mathbf{f}_t + b_2(\theta)a_{21}\mathbf{J}_\mathbf{f}\mathbf{k}_1) + O(h^3).$$

We can now match the coefficients, thus finding the order conditions:

$$\begin{aligned}
b_1(\theta) + b_2(\theta) &= \theta \\
\frac{\theta^2}{2} &= b_2(\theta)c_2 \\
\frac{\theta^2}{2} &= b_2(\theta)a_{21}.
\end{aligned} \tag{13.95}$$

Consequently, we can use $c_2 = a_{21} = \alpha$ as a free parameter, so that

$$\begin{aligned}
b_2(\theta) &= \frac{\theta^2}{2\alpha} \\
b_1(\theta) &= \theta - b_2(\theta) = \theta - \frac{\theta^2}{2\alpha}\,.
\end{aligned} \tag{13.96}$$

We have accordingly generated a family of two-stage second-order continuous explicit Runge–Kutta methods

$$\mathbf{z} = \mathbf{x}_k + h\theta\mathbf{k}_1 - \frac{h\theta^2}{2\alpha}\mathbf{k}_1 + \frac{h\theta^2}{2\alpha}\mathbf{k}_2 \tag{13.97}$$

whose Butcher tableaux are

$$\begin{array}{c|cc}
0 & & \\
\alpha & \alpha & \\
\hline
 & b_1(\theta) & b_2(\theta)
\end{array}
\quad = \quad
\begin{array}{c|cc}
0 & & \\
\alpha & \alpha & \\
\hline
 & \theta - \theta^2/2\alpha & \theta^2/2\alpha
\end{array}\,. \tag{13.98}$$

This shows how we can solve for the order condition to construct continuous Runge–Kutta methods. If we let $\alpha = 1$, then we have a continuous trapezoidal method. If we let $\alpha = 1/2$, then we have a continuous midpoint method.

The theoretical foundation for what we just did, and are about to do in general—that is, replace a direct analysis of the residual by instead matching Taylor series coefficients—is helped in practice by the following theorem.

Theorem 13.5. *Suppose the local reference solution to* $\mathbf{y}' = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(t_k) = \mathbf{y}_k$ *is denoted by* $\mathbf{y}_k(t)$. *Suppose that the degree-*$(p+1)$ *local Taylor approximation to* $\mathbf{y}_k(t)$ *is denoted by* $\mathbf{y}_T(t)$, *so* $\mathbf{y}_T(t) - \mathbf{y}_k(t) = O((t-t_k)^{p+2})$. *Let the interpolant to the numerical solution be denoted by* $\mathbf{z}(t)$. *Suppose also that* $\theta = (t-t_k)/h_k$ *as usual, and that we may flip back and forth between* $\mathbf{z}(t)$ *and* $\mathbf{z}(\theta)$ *for convenience. Then if* $\mathbf{z}(\theta) - \mathbf{y}_T(\theta) = (\theta h)^{p+1}\mathbf{E}_{p+1}(\theta) + O((\theta h)^{p+2})$ *for some polynomial* $\mathbf{E}_{p+1}(\theta)$ *then, as* $\theta h \to 0$,

$$\frac{1}{h}\mathbf{z}'(\theta) - \mathbf{f}(\mathbf{z}) = (\theta h)^p \mathbf{D}_p(\theta) + O(h^{p+1}), \tag{13.99}$$

for some function $\mathbf{D}_p(\theta)$, *which is polynomial in* θ.

Proof. We start by looking at the residual in the local Taylor expansion and use the fact that the local reference solution has zero residual (also, note that $d/dt = (1/h)d/d\theta$):

$$\begin{aligned}\frac{1}{h}\mathbf{y}_T' - \mathbf{f}(\mathbf{y}_T) &= \frac{1}{h}\left(\mathbf{y}_T' - \mathbf{y}_k'\right) - \left(\mathbf{f}(\mathbf{y}_T) - \mathbf{f}(\mathbf{y}_k)\right)\\ &= \frac{1}{h}\left(\mathbf{y}_T' - \mathbf{y}_k'\right) - O((\theta h)^{p+2})\\ &= O((\theta h)^{p+1}).\end{aligned} \tag{13.100}$$

The second equation follows from the Lipschitz continuity of $\mathbf{f}$, that is,

$$\|\left(\mathbf{f}(\mathbf{y}_T) - \mathbf{f}(\mathbf{y}_k)\right)\| \le L\|\mathbf{y}_T - \mathbf{y}_k\|,$$

and the final equation follows from direct computation; say, in fact,

$$= \mathbf{T}_{p+1}(\theta)\theta^{p+1}h^{p+1} + O(\theta h)^{p+2}), \tag{13.101}$$

where $\mathbf{T}_{p+1}(\theta)$ is polynomial in θ. Now

$$\mathbf{z}(\theta) - \mathbf{y}_T(\theta) = h^{p+1}\theta^{p+1}\mathbf{E}_{p+1}(\theta) + O((\theta h)^{p+2}) \tag{13.102}$$

by hypothesis. So

$$\begin{aligned}\frac{1}{h}\mathbf{z}'(\theta) - \frac{1}{h}\mathbf{y}_T(\theta) &= h^p\theta^p(p+1)\mathbf{E}_{p+1}(\theta) + h^p\theta^{p+1}\mathbf{E}_{p+1}'(\theta) + \cdots\\ &= h^p\theta^p\left((p+1)\mathbf{E}_{p+1} + \theta\mathbf{E}_{p+1}'\right) + O(h^{p+1}\theta^{p+1}),\end{aligned} \tag{13.103}$$

whereas, as can also be seen by the Lipschitz continuity of **f**,

$$\mathbf{f}(\mathbf{z}) - \mathbf{f}(\mathbf{y}_T) = \mathbf{f}(\mathbf{y}_T + O(h^{p+1}\theta^{p+1})) - \mathbf{f}(\mathbf{y}_T) = O(h^{p+1}\theta^{p+1}) \,, \tag{13.104}$$

which is one order higher. Therefore,

$$\Delta(\mathbf{z}) = O(h^p\theta^p), \tag{13.105}$$

as desired. ♮

This can simplify our computations. Instead of forcing $\Delta(\theta)$ to be small directly (in the sense of having a high-order factor $\theta^p h^p$ in front), we may more easily insist that our Runge–Kutta method agree in series with the Taylor series method. Again, we emphasize that the reason this works is that the method produces a small residual, in the end.

13.5.5 Order Conditions with Trees

We begin again with Taylor series, but this time we pay closer attention to efficiency in notation since, as we hinted before, it becomes crucial. Let us being with a three-dimensional autonomous system; we write

$$\frac{d\mathbf{y}}{dt} = \dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \qquad \mathbf{y}(t_0) = \mathbf{y}_0 \,. \tag{13.106}$$

Since subscripts are taken for time steps, we use *superscripts* to indicate components:

$$\dot{y}^1(t) = f^1(y^1, y^2, y^3) \tag{13.107}$$

$$\dot{y}^2(t) = f^2(y^1, y^2, y^3) \tag{13.108}$$

$$\dot{y}^3(t) = f^3(y^1, y^2, y^3), \tag{13.109}$$

where

$$\mathbf{y} = \begin{bmatrix} y^1 \\ y^2 \\ y^3 \end{bmatrix} \qquad \text{and} \qquad \mathbf{f} = \begin{bmatrix} f^1 \\ f^2 \\ f^3 \end{bmatrix} \,. \tag{13.110}$$

Now consider the second derivative $\ddot{\mathbf{y}}$. Direct computation using the chain rule gives

$$\ddot{y}^1 = f^1_1\dot{y}^1 + f^1_2\dot{y}^2 + f^1_3\dot{y}^3 \tag{13.111}$$

$$\ddot{y}^2 = f^2_1\dot{y}^1 + f^2_2\dot{y}^2 + f^2_3\dot{y}^3 \tag{13.112}$$

$$\ddot{y}^3 = f^3_1\dot{y}^1 + f^3_2\dot{y}^2 + f^3_3\dot{y}^3 \tag{13.113}$$

or, expressed in matrix–vector notation as we did before,

$$\ddot{\mathbf{y}} = \mathbf{J}_{\mathbf{f}}(\mathbf{y})\dot{\mathbf{y}} = \mathbf{J}_{\mathbf{f}}(\mathbf{y})\mathbf{f}, \tag{13.114}$$

where the Jacobian matrix is

$$\mathbf{f}' = \mathbf{f}^{(1)} = \mathbf{J}_{\mathbf{f}} = \begin{bmatrix} f^1_1 & f^1_2 & f^1_3 \\ f^2_1 & f^2_2 & f^2_3 \\ f^3_1 & f^3_2 & f^3_3 \end{bmatrix}. \tag{13.115}$$

In tensor notation, this takes an extremely compact form:

$$\ddot{y}^i = \sum_{j=1}^{3} f^i_j f^j. \tag{13.116}$$

Even better, if we use Einstein's summation convention, according to which repeated indices are summed over automatically, we can drop the $\sum$ altogether and simply write

$$\ddot{y}^i = f^i_j f^j. \tag{13.117}$$

Just as we wrote f^i_j for $\partial f^i/\partial y^j$ in the Jacobian matrix, we will now use the notation f^i_{jk} for $\partial^2 y^i/\partial y^j \partial y^k$, $f^i_{jk\ell}$ for $\partial^3 y^i/\partial y^j \partial y^k \partial y^\ell$, and so on. Using this notation, we can then write compact expressions for higher derivatives of $\mathbf{y}$. Let us consider $\dddot{\mathbf{y}}$. By direct computation, we find

$$\dddot{y}^1 = (f^1_{1,1}\dot{y}^1 + f^1_{1,2}\dot{y}^2 + f^1_{1,3}\dot{y}^3)\dot{y}^1 + (f^1_{2,1}\dot{y}^1 + f^1_{2,2}\dot{y}^2 + f^1_{2,3}\dot{y}^3)\dot{y}^2 \tag{13.118}$$

$$+ (f^1_{3,1}\dot{y}^1 + f^1_{3,2}\dot{y}^2 + f^1_{3,3}\dot{y}^3)\dot{y}^3 + f^1_1\ddot{y}^1 + f^1_2\ddot{y}^2 + f^1_3\ddot{y}^3 \tag{13.119}$$

by the product rule and the chain rule. This long expression is just the first component! But let us again move to tensor notation with the summation convention, and it becomes a compact expression fitting on one line:

$$\dddot{y}^1 = f^1_{jk}\dot{y}^j\dot{y}^k + f^1_j f^j_k f^k. \tag{13.120}$$

The other components have the same form, as well, so that we can simply write

$$\dddot{y}^i = f^i_{jk} f^j f^k + f^i_j f^j_k f^k, \tag{13.121}$$

which is a significant economy of notation. Just remember these are each double sums:

$$\dddot{y}^i = \sum_{j=1}^{3}\sum_{k=1}^{3} f^i_{jk} f^j f^k + \sum_{j=1}^{3}\sum_{k=1}^{3} f^i_j f^j_k f^k, \tag{13.122}$$

and, of course, $1 \leq i \leq 3$ also. Now, consider $^{d^4}/_{dt^4}y^i$:

$$\frac{d^4}{dt^4}y^i = f^i_{jk\ell}f^jf^kf^\ell + f^i_{jk}(f^i_\ell f^\ell)f^k + f^i_{jk}f^j(f^k_\ell f^\ell) \tag{13.123}$$

$$= (f^i_{j\ell}f^\ell)f^j_k f^k + f^i_j(f^j_{k\ell}f^\ell)f^k + f^i_j f^j_k(f^k_\ell f^\ell). \tag{13.124}$$

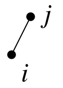
Notice now that the second, third, and fourth terms are somehow the same:

$$f^i_{jk}f^j_\ell f^\ell f^k = f^i_{jk}f^k_\ell f^j f^\ell = f^i_{j\ell}f^j_k f^\ell f^k \tag{13.125}$$

when summed (these are all triple sums, of course). Now comes the beautiful bit. Associate f^i with the rooted, labelled tree

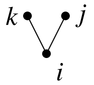
,

$f^i_jf^j$ with the tree

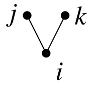


,

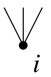
and $f^i_{jk}f^jf^k$ with the tree



(or equivalently

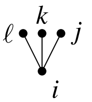


which is isomorphic). The root of the tree is labeled with the index of the component we're working on, i. If the f^i term has subscripts, say, j, k, and ℓ, put an arc out from $\bullet i$ for each subscript; so $f^i_{jk\ell}$ starts with



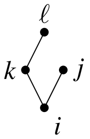
.

But we typically have terms that contain more than just $f^i_{jk\ell}$, as above, where we had $f^i_{jk\ell}f^jf^kf^\ell$. For terms like that, for each f^j, put a dot at the end:

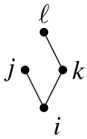


,

and expand the terms whose form is not simply f^j along the other arcs, in a recursive way. We then find that the term $f^i_{jk}(f^k_\ell f^\ell)f^j$ gives

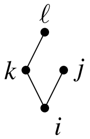


or



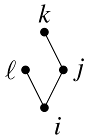
(these are again isomorphic). But there's something different between the pair of terms and trees

$$f^i_{jk} f^j f^k_\ell f^\ell$$



and

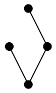
$$f^i_{j\ell} f^\ell f^j_k f^k$$



in that in order to map them onto each other, they must be *relabeled* (whereas the two equivalent trees for $f^i_{jk} f^k_\ell f^\ell f^j$ can simply be reflected to make them look identical).

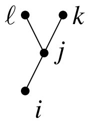
We can then count the number of ways of labeling a tree. Following Butcher (2001 p. 92), we say that "the number of ways of labeling a tree t with a given totally ordered set V (here, $V = \{i, j, k, \ell\}$ with $i < j < k < \ell$) with $|V| = r(t)$ (the number of nodes in the tree t) in such a way that if (m, n) is an arc, then $m < n$ is called $\alpha(t)$."[9] If

$\tau =$



then $\alpha(\tau) = 3$, as can be seen by inspection. We now continue with the other expressions. The term $f^i_j f^j_{k\ell} f^k f^\ell$ corresponds to

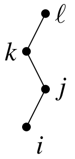
$$f^i_j f^j_{k\ell} f^k f^\ell$$



and $\alpha(\tau) = 1$ for this tree. Below, $f^i_j f^j_k f^k_\ell f^\ell$ is an example of what is called a *tall tree*:

[9] We note that here we run into an unfortunate conflict of notation: Butcher (2008b), Hairer et al. (1993), and Hairer et al. (2006) all use t to denote a tree, but we use t as the independent variable. We will use τ instead to denote a particular tree (but note that Butcher (2008b) uses τ for another purpose).

$f^i_j f^j_k f^k_\ell f^\ell$



Using this notation, derivatives for Taylor series are easy:

$$\mathbf{y}^{(n)} = \sum_{r(\tau)=n} \alpha(\tau)\mathbf{F}(\tau)(\mathbf{y}), \tag{13.126}$$

where $\mathbf{F}(\tau)(\mathbf{y})$ is the *elementary differential* corresponding to the tree τ, and the sum is taken over all rooted, labeled trees with $r(\tau) = n$, that is, with n vertices. An elementary differential is a vector with the components

$$f^i_{jk\ell\cdots} f^{j\cdots}, \tag{13.127}$$

that is, the associated expression of the tree.

These expressions, which are given informally here, can be formally defined in a recursive manner, as follows (definition 301a in Butcher (2008b)):

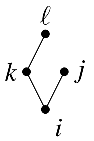
Definition 13.7. For a function $\mathbf{f} : \mathbb{C}^n \to \mathbb{C}^n$, the elementary differential $\mathbf{F}(\tau) : \mathbb{C}^n \to \mathbb{C}^n$ corresponding to the rooted tree τ is recursively defined by the base case

$$\text{Base} : \begin{cases} \mathbf{F}(\emptyset)(\mathbf{y}) = \mathbf{y} \\ \mathbf{F}([\cdot])(\mathbf{y}) = \mathbf{f}(\mathbf{y}) \end{cases} \tag{13.128}$$

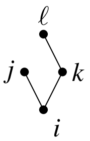
and the following recursion condition: If $\tau = \begin{bmatrix}\tau_1 & \tau_2 & \cdots & \tau_s\end{bmatrix}$, $s \geq 1$ (that is, the tree τ can be cut into the trees $\tau_1, \tau_2, \ldots, \tau_2$ by removing the root of τ and all arcs leading from it), then

$$\mathbf{F}(\tau)(\mathbf{y}) = \mathbf{f}^{(s)}(\mathbf{F}(\tau_1)(\mathbf{y}), \mathbf{F}(\tau_2)(\mathbf{y}), \ldots, \mathbf{F}(\tau_s)(\mathbf{y})). \tag{13.129}$$

The notation for terms such as $\mathbf{f}^{(3)}(\mathbf{f}, \mathbf{f}'(\mathbf{f}))$ appearing in the recursive definition above would also be associated with the tree



or



,

and to the previous notation $f^i_{jk} f^j f^k_\ell f^\ell$. ◁

While we are about it, let us give the recurrence relations for $r(\tau)$ and $\alpha(\tau)$. We also need a measure of density $\gamma(\tau)$ and symmetry $\sigma(\tau)$. First, we have

$$\sigma([\cdot]) = \alpha([\cdot]) = \gamma([\cdot]) = r([\cdot]) = 1\,. \tag{13.130}$$

Moreover, if

$$\begin{aligned}\tau &= [\tau_1\ \tau_2\ \cdots\ \tau_s] \\ &= [\hat{\tau}_1^{n_1}\ \hat{\tau}_2^{n_2}\ \cdots\ \hat{\tau}_\ell^{n_\ell}]\end{aligned} \tag{13.131}$$

where the $\hat{\tau}_i$ are all distinct subtrees, we have

$$r(\tau) = 1 + \sum_{i=1}^{s} r(\tau_i) \tag{13.132}$$

$$\gamma(\tau) = r(\tau) \prod_{i=1}^{s} \gamma(\tau_i) \tag{13.133}$$

$$\sigma(\tau) = n_1!n_2!\cdots n_\ell! \prod_{i=1}^{\ell} \sigma(\tau_i)^{n_i} \tag{13.134}$$

and

$$\alpha(\tau) = \frac{r(\tau)!}{\gamma(\tau)\sigma(\tau)}\,. \tag{13.135}$$

For proofs, see Butcher (2008b). This seems a lot of work for Taylor series (which we would generate by recursive relations anyway). We have, now,

$$\begin{aligned}\mathbf{y}(t) &= \sum_{k\geq 0} \frac{\mathbf{y}^{(k)}(t_n)}{k!}(t-t_n)^k \\ &= \sum_{k\geq 0} \left(\sum_{r(\tau)=k} \frac{\alpha(\tau)}{k!} \mathbf{F}(\tau)(\mathbf{y}_n) \right) (t-t_n)^k,\end{aligned} \tag{13.136}$$

but (and this is the point) it turns out we can use these graphs and elementary differentials for Runge–Kutta methods too.

Consider the Runge–Kutta method specified by

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}\,. \tag{13.137}$$

This corresponds to the method

$$\mathbf{k}_1 = \mathbf{f}\left(\mathbf{y}_n + h \sum_{j=1}^{s} a_{1,j} \mathbf{k}_j \right) \tag{13.138}$$

$$\mathbf{k}_2 = \mathbf{f}\left(\mathbf{y}_n + h \sum_{j=1}^{s} a_{2,j} \mathbf{k}_j \right) \tag{13.139}$$

$$\vdots$$

$$\mathbf{k}_s = \mathbf{f}\left(\mathbf{y}_n + h\sum_{j=1}^{s} a_{s,j}\mathbf{k}_j\right), \tag{13.140}$$

which is sometimes more conveniently expressed in the following notation:

$$\mathbf{Y}_i^J = \mathbf{y}_n^J + h\sum_{j=1}^{s} a_{ij}\mathbf{f}^J(\mathbf{Y}_j^1, \mathbf{Y}_j^2, \ldots, \mathbf{Y}_j^n) \tag{13.141}$$

for $1 \le i \le s$,and $1 \le J \le n$, and n is the number of components of $\mathbf{f}$. Then, the value of the solution at the next step is

$$\mathbf{Y}_{n+1}^J = \mathbf{y}_n^J + h\sum_{j=1}^{s} b_j\mathbf{f}^J(\mathbf{Y}_j^1, \cdots, \mathbf{Y}_j^n). \tag{13.142}$$

In the k-notation, the unknowns were like derivatives; here the unknown $\mathbf{Y}$ are like function values. Replacing h by θh and allowing $b_i = b_i(\theta)$ give the continuous Runge–Kutta (CRK) case.

We now define the *elementary weights* of this method recursively by

$$\Phi([\cdot]) = c_i \tag{13.143}$$

for the empty tree, and if $\tau = [\tau_1\ \tau_2\ \cdots\ \tau_m]$, then

$$\Phi_i(\tau) = \sum_{j=1}^{s} a_{ij}\prod_{k=1}^{m}\Phi_j(\tau_k). \tag{13.144}$$

Finally, we have

$$(\mathbf{Y}_{n+1}^J)^{(\ell)}(t_n) = \sum_{r(\tau)=\ell}\alpha(\tau)\gamma(\tau)\sum_{j=1}^{s} b_j\Phi_j(\tau)\mathbf{F}^J(\tau)(\mathbf{y}_n) \tag{13.145}$$

and for the Taylor series of the numerical method to agree with that of the Taylor series method to order p, we must have

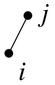
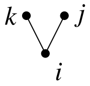
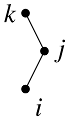
$$\sum_{j=1}^{s} b_j\Phi_j(\tau) = \frac{\theta^{r(\tau)}}{\gamma(\tau)} \tag{13.146}$$

for all trees τ with $r(\tau) \le p$. To prove this, compare the two series and equate coefficients. Since the elementary differentials are independent, for the difference

$$\mathbf{y}_{n+1} - \mathbf{y}_{TSM}(t_n + \theta h) = O((\theta h)^{p+1}) \tag{13.147}$$

to hold, each pair of coefficients must be separately identical. These are the order conditions.

The first few order conditions for continuous Runge–Kutta methods are given below.

r	τ	γ	α	$\Phi_j = \theta^{r(\tau)}/\gamma(\tau)$
0	$\emptyset$	1	1	
1		1	1	$\sum b_i = \theta$
2		2	1	$\sum b_i c_i = \theta^2/2$
3		3	1	$\sum b_i c_i^2 = \theta^3/3$
		6	1	$\sum b_i a_{ij} c_j = \theta^2/6.$

(13.148)

These are taken from tables such as table 307 from Butcher (2008a) or Table 2.7 from Hairer et al. (1993).

The number of trees and therefore the order conditions rise very quickly: For order 10, there are 1205 order conditions to satisfy (just in the $\theta = 1$ case), although, because of redundancy, we do not need 1205 unknowns. *Solving* these multivariate polynomials, on the other hand, remains a significant challenge for high orders.

13.5.6 Solving the Order Conditions

In solving the order conditions, we search for the parameters in a Butcher tableau such as

$$\begin{array}{c|cccc} c_2 & a_{21} & & & \\ c_3 & a_{31} & a_{32} & & \\ c_4 & a_{41} & a_{42} & a_{43} & \\ \hline & b_1(\theta) & b_2(\theta) & b_3(\theta) & b_4(\theta) \end{array} \tag{13.149}$$

so that

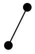
$$\mathbf{z}(\theta) = \mathbf{y}_k + h(b_1(\theta)\mathbf{k}_1 + b_2(\theta)\mathbf{k}_2 + b_3(\theta)\mathbf{k}_3 + b_4(\theta)\mathbf{k}_4) \tag{13.150}$$

has a residual [with $\theta = (t - t_k)/h$]:

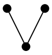
$$\Delta(\theta) = \frac{1}{h}\frac{d\mathbf{y}}{d\theta} - \mathbf{f}(\mathbf{y}(\theta)) = O(h^3)\,. \tag{13.151}$$

The order conditions are

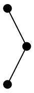
[•]
$$b_1(\theta)+b_2(\theta)+b_3(\theta)+b_4(\theta) = \theta$$



$$c_2b_2(\theta)+c_3b_3(\theta)+c_4b_4(\theta) = \tfrac{1}{2}\theta^2$$



$$c_2^2b_2(\theta)+c_3^2b_3(\theta)+c_4^2b_4(\theta) = \tfrac{1}{3}\theta^3 \qquad (13.152)$$



$$a_{32}c_2b_3(\theta)+(a_{42}c_2+a_{43}c_3)b_4(\theta) = \tfrac{1}{6}\theta^3$$

and, of course, $c_2 = a_{21}$, $c_3 = a_{31}+a_{32}$, and $c_4 = a_{41}+a_{42}+a_{43}$. Thus, as a linear system, we have

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & c_2 & c_3 & c_4 \\ 0 & c_2^2 & c_3^2 & c_4^2 \\ 0 & 0 & a_{32}c_2 & a_{42}c_2+a_{43}c_3 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = \begin{bmatrix} \theta \\ \theta^2/2 \\ \theta^3/3 \\ \theta^3/6 \end{bmatrix}. \qquad (13.153)$$

Lemma 13.1. *If the matrix* **A** *above is singular, then no solution exists for any values of* $a_{21}, a_{31}, a_{32}, a_{41}, a_{42}$*, or* a_{43}.

Proof. Assume to the contrary that $b_1(\theta), b_2(\theta), b_3(\theta)$, and $b_4(\theta)$ exist, with **A** singular, so that

$$\mathbf{Ab} = \begin{bmatrix} \theta \\ \theta^2/2 \\ \theta^3/3 \\ \theta^3/6 \end{bmatrix}. \qquad (13.154)$$

Now, since **A** is singular, then there exists a nonzero vector

$$\mathbf{L} = \begin{bmatrix} L_1 \ L_2 \ L_3 \ L_4 \end{bmatrix} \qquad (13.155)$$

such that $\mathbf{LA} = \begin{bmatrix} 0\ 0\ 0\ 0 \end{bmatrix}$. Thus, we have

$$\mathbf{LA}\begin{bmatrix} b_1 \\ \vdots \\ b_4 \end{bmatrix} = L_1\theta + \frac{L_2}{2}\theta^2 + (\frac{L_3}{3}+\frac{L_4}{6})\theta^3 \equiv 0\,. \qquad (13.156)$$

Because θ, θ^2, and θ^3 are independent, $L_1 = L_2 = 0$ and $L_3 = {}^{-L_4}\!/2$. Now, observe that

$$\begin{bmatrix} 0\ 0\ -\frac{L_4}{2}\ L_4 \end{bmatrix}\mathbf{A} = \begin{bmatrix} 0\ -\frac{c_2^2}{2}\ -\frac{c_3^2}{2}+a_{32}c_2\ -\frac{c_4^2}{2}+a_{32}c_2+a_{43}c_3 \end{bmatrix} = \begin{bmatrix} 0\ 0\ 0\ 0 \end{bmatrix}.$$

Hence, $c_2 = 0, c_3 = 0$, and $c_4 = 0$. However, it is obvious that in this case

$$\mathbf{A}\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = \begin{bmatrix} b_1 + b_2 + b_3 + b_4 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{13.157}$$

cannot be $\left[\theta \;\, {}^{\theta^2}\!/2 \;\, {}^{\theta^3}\!/3 \;\, {}^{\theta^3}\!/6\right]^T$. The contradiction invalidates the hypothesis that $\mathbf{b}$ exists. $\natural$

Thus, we may restrict our search for third-order CERK methods to the case $\det(\mathbf{A}) \neq 0$. A short computation (using MAPLE) gives

$$\det \mathbf{A} = c_2\big(c_2c_4(c_2 - c_4)a_{32} + (c_3 - c_2)c_3(a_{42}c_2 + a_{43}c_3)\big)\,. \tag{13.158}$$

Inspection shows that $c_2 = a_{21} \neq 0$ is necessary. There are three cases to consider: the generic case $c_3 \neq 0$, $c_2 \neq c_3$; the second case $c_2 = c_3$ with $c_2 \neq c_4$, $c_4 \neq 0$, and $a_{32} \neq 0$; and the third case $c_3 = 0$ with $c_2 \neq c_4$, $c_4 \neq 0$, and $a_{32} \neq 0$. These are the only conditions under which $\mathbf{A}$ is nonsingular.

It turns out that to get a CERK of order 3, we need not just three stages but also one more, as we have been using here. For a proof that four stages are really needed, see Owren and Zennaro (1991), where they extend the pioneering work of Butcher. This is serious: An extra stage means an extra function evaluation every step, and f may have a million components. We'd really like to make an economization here. It turns out that there is one possible economization, known as "stage reuse," or FSAL for "first same as last": If the last stage you use on this step, $\mathbf{k}_4$, turns out to give you the $\mathbf{k}_1$ you need next time, then on all but the very last step the fourth stage is essentially free. This gives us a condition to desire, for efficiency. We impose our efficiency condition here as follows: $b_1(1) = a_{41}$, $b_2(1) = a_{42}$, $b_3(1) = a_{43}$, and $b_4(1) = 0$. This ensures that computation of $\mathbf{y}_{k+1}$ does not require $\mathbf{f}(\mathbf{y}_{k+1})$, that is, $\mathbf{k}_4$.

All this gives four polynomial equations in the six unknowns $a_{21}, a_{31}, a_{32}, a_{41}, a_{42}$, and a_{43}. We examine the real solution of these equations under each of our three scenarios [generic, second ($c_2 = c_3$), and third ($c_3 = 0$)]. In the generic case, MAPLE gives a solution with two free parameters, a_{31} and a_{32}, except that a_{32} cannot be zero. Given these two parameters, compute α so that

$$3a_{32}\alpha^2 - (3a_{32} + 1)\alpha + (a_{32} + a_{31})^2 = 0\,. \tag{13.159}$$

Then the other parameters are determined by

$$a_{21} = \alpha \tag{13.160}$$

$$a_{41} = \frac{1}{6}\left(\frac{6a_{32}\alpha^2 - (3a_{32} + 1)\alpha + (a_{31} + a_{32})}{a_{32}\alpha^2}\right) \tag{13.161}$$

$$a_{42} = \frac{1}{6}\,\frac{3a_{32}\alpha - (a_{32} + a_{31})}{a_{32}\alpha^2} \tag{13.162}$$

$$a_{43} = \frac{1}{6a_{32}\alpha}\,. \tag{13.163}$$

Finally, we solve the linear system of Eq. (13.154) for the $b_i(\theta)$. This is a two-parameter family of solutions.

Example 13.4. The MAPLE solution above is for the generic case with $c_2 \neq c_3$. That two-parameter family does not exhaust the possible solutions. Consider the following Butcher tableau:

$$\begin{array}{c|cccc}
2/3 & 2/3 & & & \\
2/3 & & 2/3 & & \\
1 & 1/4 & 3/8 & 3/8 & \\
\hline
 & b_1(\theta) & b_2(\theta) & b_3(\theta) & b_4(\theta),
\end{array}$$

with

$$b_1 = -\frac{5}{4}\theta^2 + \frac{1}{2}\theta^3 + \theta \tag{13.164}$$

$$b_2 = \frac{9}{8}\theta^2 - \frac{3}{4}\theta^3 \tag{13.165}$$

$$b_3 = \frac{9}{8}\theta^2 - \frac{3}{4}\theta^3 \tag{13.166}$$

$$b_4 = -\theta^2 + \theta^3\,. \tag{13.167}$$

You will be asked to investigate this method in Problem 13.28 and prove that it is a FSAL third-order continuous explicit Runge–Kutta method. ◁

To solve the order conditions for higher-order methods, one cannot use brute force as we have here, even assisted by computer algebra. As previously stated, there are too many symmetries in the solution; as a result, the naive use of Gröbner bases runs into combinatorial growth in the number of possible roots. One must use a more nimble approach, possibly with simplifying assumptions. See Butcher (2008b) for an introduction to how it is done by humans; for a promising new approach that may lead to automatic solution methods, see Khashin (2009) and Khashin (2012).

13.5.7 Implicit RK Methods

John Butcher began the systematic study of implicit methods in 1964; prior to that, some isolated methods were known, and known to be effective for certain problems, but fundamental difficulties (the existence of high-order methods, for example) had not really been addressed. Following Butcher (1964), the Butcher tableau of an s-stage implicit method is

$$\begin{array}{c|ccccc}
c_1 & a_{11} & a_{12} & a_{13} & \cdots & a_{1s} \\
c_2 & a_{21} & a_{22} & a_{23} & \cdots & a_{2s} \\
\vdots & \vdots & \vdots & & \ddots & \vdots \\
c_s & a_{s1} & a_{s2} & a_{s3} & \cdots & a_{ss} \\
\hline
 & b_1 & b_2 & b_3 & \cdots & b_s,
\end{array} \tag{13.168}$$

and the stage values $\mathbf{Y}_i$ satisfy

$$\mathbf{Y}_i = \mathbf{y}_n + h \sum_{j=1}^{s} a_{ij} \mathbf{f}(\mathbf{Y}_j) \tag{13.169}$$

or, equivalently with $\mathbf{k}_i = f(\mathbf{Y}_i)$,

$$\mathbf{k}_i = \mathbf{f}(\mathbf{y}_n + h \sum_{j=1}^{n} a_{ij} \mathbf{k}_j) \,. \tag{13.170}$$

If $\mathbf{f}$ is a linear function, then these are linear equations for the NS unknowns; moreover, the linear system is quite highly structured. Let us consider two examples, first with $N = 1$ (a scalar autonomous problem) and then a vector problem.

Let us first consider the function $f(y) = cy + d$, and suppose we have the three stages

$$\mathbf{Y}_1 = \mathbf{y}_n + h \sum_{j=1}^{3} a_{ij}(c\mathbf{Y}_j + d) \tag{13.171}$$

$$\mathbf{Y}_2 = \mathbf{y}_n + h \sum_{j=1}^{3} a_{2j}(c\mathbf{Y}_j + d) \tag{13.172}$$

$$\mathbf{Y}_3 = \mathbf{y}_n + h \sum_{j=1}^{3} a_{3j}(c\mathbf{Y}_j + d) \,, \tag{13.173}$$

which can be written as

$$\left[\mathbf{I} - hc\mathbf{A}\right] \begin{bmatrix} \mathbf{Y}_1 \\ \mathbf{Y}_2 \\ \mathbf{Y}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{y}_n \\ \mathbf{y}_n \\ \mathbf{y}_n \end{bmatrix} + h\mathbf{A} \begin{bmatrix} d \\ d \\ d \end{bmatrix} \,. \tag{13.174}$$

We see that we may solve this linear system provided $|hc|$ is smaller than the reciprocal of the largest eigenvalue of $\mathbf{A}$. Even for linear scalar problems, implicit methods thus seem to have a step-size restriction, but for “small enough” step sizes, the matrix approaches $\mathbf{I}$ and solution becomes possible. Of course, vector problems and nonlinear problems inherit this behavior.

Let us now consider the vector problem

$$\mathbf{f}(\mathbf{y}) = \mathbf{C}\mathbf{y} + \mathbf{D} \tag{13.175}$$

with $\mathbf{C} \in \mathbb{C}^{4\times 4}$ and $\mathbf{D} \in \mathbb{C}^4$. Again, we use three stages, so that $s = 3$. Then our stage values are

$$\mathbf{Y}_i = \mathbf{y}_n + h \sum_{j=1}^{s} a_{ij}\mathbf{f}(\mathbf{Y}_j)\,, \tag{13.176}$$

and for this particular problem they are such that

$$\mathbf{Y}_i = \mathbf{y}_n + h \sum_{j=1}^{s} a_{ij}[\mathbf{C}\mathbf{Y}_j + \mathbf{D}] = \mathbf{y}_n + h \sum_{j=1}^{s} a_{ij}\mathbf{C}\mathbf{Y}_j + h \sum_{j=1}^{s} a_{ij}\mathbf{D}\,. \tag{13.177}$$

This can be written in the form

$$(\mathbf{I}_N - ha_{ii}\mathbf{C})\mathbf{Y}_i - h \sum_{\substack{j=1\\ j\neq i}}^{s} a_{ij}\mathbf{C}\mathbf{Y}_j = \mathbf{y}_n + h \sum_{j=1}^{s} a_{ij}\mathbf{D}\,, \tag{13.178}$$

or, using the tensor product,

$$(\mathbf{I} - h(\mathbf{A}\otimes\mathbf{C})) \begin{bmatrix} \mathbf{Y}_1 \\ \mathbf{Y}_2 \\ \vdots \\ \mathbf{Y}_s \end{bmatrix} = \begin{bmatrix} \mathbf{y}_n \\ \mathbf{y}_n \\ \vdots \\ \mathbf{y}_n \end{bmatrix} + h(\mathbf{A}\otimes\mathbf{D})\,. \tag{13.179}$$

Again, this matrix is nonsingular for “small enough” h.

This generalizes to nonlinear problems via the implicit function theorem and gives some reassurance that IRK methods can be used in practice (which is good, because indeed they are). But the theorem alluded to above masks an interesting problem: Implicit methods are of interest *precisely* when eigenvalues of $\mathbf{C}$ are large (with negative real part), and if the step-size restriction of the theorem always constrained the methods, IRK would be useless. We do not pursue further this technical but important matter. Instead, we consider some low-dimensional examples, in order to give insight.

Example 13.5. Consider the scalar nonlinear problem

$$\dot{x} = x^2 \qquad x(0) = 1 \tag{13.180}$$

and solve this with the implicit midpoint rule:

$$k_1 = f(x_n + \frac{h}{2}k_1) = (x_n + \frac{h}{2}k_1)^2. \tag{13.181}$$

Quite obviously this has two solutions for any $h \neq 0$, though only one if $h = 0$. The solutions of

$$\frac{h^2}{4}k_1^2 + (x_nh - 1)k_1 + x_n^2 = 0 \tag{13.182}$$

by the quadratic formula are

$$k_1 = \frac{1 - hx_n \pm \sqrt{(1-hx_n)^2 - h^2x_n^2}}{h^2/2}\,; \tag{13.183}$$

that is,

$$k_1^+ = \frac{2}{h^2}(1 - hx_n + \sqrt{1-2hx_n}) \quad \text{and} \quad k_1^- = \frac{2x_n^2}{1 - hx_n + \sqrt{1-2hx_n}}\,.$$

Notice that $k_1^- \to x_n^2 = f(x_n)$ as $h \to 0$, but that $k_1^+ \to \infty$. Even for this simple problem, the solution of the IRK equations is *not unique*, although the one we want does behave well as $h \to 0$. Notice also that if $h > 1/2x_n$, then both solutions are complex—that is, for large enough h, there are no real solutions. If x_n is large, this step-size restriction begins to bite hard. And since x_n *does* get large, this matters: As we saw in Chap. 12, the reference solution has a singularity, and forcing h to zero is a crude detection mechanism for singularities. ◁

Example 13.6. Consider instead

$$\dot{x} = 1 - x^2 \tag{13.184}$$

with (again) the implicit midpoint rule:

$$k_1 = (1 - (x_n + \frac{h}{2}k_1)^2) \tag{13.185}$$

$$x_{n+1} = x_n + hk_1. \tag{13.186}$$

Solving

$$\frac{h^2}{4}k_1^2 + (hx_n + 1)k_1 + x_n^2 - 1 = 0 \tag{13.187}$$

for k_1, we find

$$k_1 = \frac{-(1 + hx_n) \pm \sqrt{(1+hx_n)^2 + h^2(1-x_n^2)}}{h^2/2}\,. \tag{13.188}$$

Thus, we find the expression

$$k_1^+ = \frac{-2(1-x_n^2)}{-(1+hx_n) - \sqrt{1 + 2hx_n + h^2}} \tag{13.189}$$

for the root of interest. Now, we have $1 + 2hx_n + h^2 \geq 0$ if

$$x_n \geq -\frac{1+h^2}{2h}\,, \tag{13.190}$$

and since we can separately prove that $x_n \to 1$ as $n \to \infty$ for any $x_0 \geq 0$, we see that in this case there is no problem. There always is a solution to the IRK equations. $\triangleleft$

As a final remark, once the existence of a solution to the IRK has been established, one could compute the solution as a Taylor series in h—but up to order $O(h^{p+1})$, for a pth-order method, the coefficients would be the same (by definition) as those of the Taylor series method; to see what distinguishes the IRK solution from the Taylor series solution, one must think differently.

13.5.7.1 *L*-Stable RK Methods

We have already briefly examined the stability regions for forward (explicit) Euler and backward (implicit) Euler; in the exercises, you have been asked to draw the stability regions for a few Taylor series methods (both explicit and implicit). We have also just discussed the difficulties with implicit methods in general, which also affect implicit Runge–Kutta methods: The solution of the nonlinear equations may be difficult or expensive, and indeed the iterations may not converge at all.

So why use implicit methods if they are so awkward? Well, it turns out that for a large class of practical (but stiff) problems, these difficulties can be overcome, and the advantage of a stiffly stable method is that the time step h is not restricted (as much) by those difficulties as the time step would be by the lack of stiff stability for an explicit method.

Normally, as we have seen previously, the advantages of implicit methods are presented by examining how they work on the Dahlquist test problem $y' = \lambda y$, for $\mathrm{Re}(\lambda) \ll 0$. There is value in the standard approach, and our presentation of implicit Runge–Kutta methods would be incomplete without drawing a few stability regions (that is, regions in the $z = h\lambda$ plane for which the skeleton of the numerical solution y_n exhibits monotonic decay, just as the reference solution does for $\mathrm{Re}\lambda < 0$). We should also mention the notion of A-stability,[10] where the method preserves decay when $\mathrm{Re}\lambda < 0$—note that forward Euler is not A-stable, but backward Euler is A-stable: In the first case, sometimes the numerical solution grows when it should decay, and in the second case, while sometimes the numerical solution decays when it should grow, at least it always decays when it should decay. Some methods are “exactly” A-stable, having decay if and only if $\mathrm{Re}\lambda < 0$ although this is less useful than it might appear, as we now show.

For the next example of the stability region for an implicit RK method, consider the implicit midpoint rule:

$$\mathbf{k}_1 = \mathbf{f}(\mathbf{x}_n + \frac{h}{2}\mathbf{k}_1) \qquad (13.191)$$

[10] This powerful notion is due to Dahlquist (1963), who in that paper establishes an important negative result: No explicit linear multistep method can be A-stable if its order is higher than 2.

with $\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{k}_1$. Applying this to $y' = \lambda y$ gives

$$k_1 = \frac{\lambda x_n}{1 - z/2}\,, \tag{13.192}$$

and therefore we have

$$\begin{aligned} x_{n+1} = x_n + h\frac{\lambda x_n}{1 - z/2} &= \left(1 + \frac{z}{1 - z/2}\right) x_n \\ &= \left(\frac{1 + z/2}{1 - z/2}\right) x_n\,. \end{aligned} \tag{13.193}$$

The relevant qualitative behaviour is then described by the stability region $|(1 + z/2)/(1 - z/2)| < 1$. We may describe this region parametrically by solving the equation for the boundary of this region, namely,

$$\frac{1 + z/2}{1 - z/2} = e^{i\theta}\,, \tag{13.194}$$

to get

$$z = 2\frac{\exp(i\theta) - 1}{\exp(i\theta) + 1} = \frac{2\sin(\theta)}{1 + \cos(\theta)}\, i\,. \tag{13.195}$$

That is, the boundary is exactly the imaginary z-axis. Indeed, this can be deduced more elegantly from certain theorems in complex analysis, as is done, for example, in Hairer and Wanner (2002).

This numerical method gives a solution to $y' = \lambda y$ that decays when $\mathrm{Re}\lambda < 0$, and grows when $\mathrm{Re}\lambda > 0$, just as the reference solution $\exp(\lambda t)$ does. This is exact A-stability. The implicit midpoint rule is a nice method for other reasons, too. First, it is second-order accurate. Also, it preserves fixed points of autonomous systems; that is, an equilibrium of the system $\dot{x} = f(x)$ is exactly an equilibrium of the discrete dynamical system (13.191). Moreover, it preserves the *stability* of the equilibria (which not all methods do) and it preserves both period-doubling bifurcation points and flip bifurcation points.

But it isn't perfect. Perhaps its biggest flaw is that as $z \to \infty$, the stability function tends to -1. This induces a potentially spurious numerical oscillation into the solution—we still have decay, but it's *very slow* decay—that can be cured only by taking small time steps, which again might be excessively small for efficiency's sake.

Other exactly A-stable methods exist: For example, all the IRK methods based on Gauss quadrature points (which have order $2s$ if they have s stages) share this property. But sometimes the more useful property is "*L*-stability," which asks whether the stability function goes to zero as $z \to -\infty$; this reflects the qualitative feature that decay should be *faster* if $\mathrm{Re}\lambda$ is more negative.

There are also B-stability (Dahlquist 1976) and, equivalently for RK methods, "algebraic stability" for contractive nonlinear problems $\dot{x} = f(t,x)$, where f satisfies a one-sided Lipschitz condition

$$\langle f(t,x) - f(t,y), x-y\rangle \leq \nu \|x-y\|^2, \tag{13.196}$$

where, crucially, the number ν can be negative. This condition ensures that the distance between two reference solutions contracts as t increases. Of course, we would like for numerical solutions to share that feature. Not all RK methods do, but the ones that do are called B-stable. For a discussion of this, and the equivalent algebraic stability conditions, see Hairer and Wanner (2000).

Example 13.7. Consider again the problem $\dot{x} = x^2 - t$ from Hubbard and West (1991). Since

$$\begin{aligned}\langle x^2 - t - (y^2 - t), x-y\rangle &= \langle (x+y)(x-y), x-y\rangle \\ &= (x+y)(x-y)^2\,,\end{aligned} \tag{13.197}$$

this problem is contractive if $x(t)$ and $y(t)$ are both negative. If the initial conditions for x and y are negative enough, we can see that indeed $x(t) < 0$ for all time, and similarly for $y(t)$. Thus, we would like to see if, say, the implicit midpoint rule was contractive. By looking at the conditions for algebraic stability (given, for example, in Hairer and Wanner (2002) but not here), we can see that the method is predicted to be B-stable. To see this directly, we carry out one step of the method, symbolically. If we start at (t_n, x_n), then

$$x_{n+1/2} = x_n + \frac{h}{2}(x_{n+1/2}^2 - (t_n + \frac{h}{2}))\,, \tag{13.198}$$

$k_1 = x_{n+1/2}^2 - (t_n + h/2)$, and

$$x_{n+1} = x_n + hk_1 = x_n + 2(x_{n+1/2} - x_n) = 2x_{n+1/2} - x_n\,. \tag{13.199}$$

Thus, if we consider starting at *two* nearby points (t_n, x_n) and (t_n, y_n) with both $x_n < 0$ and $y_n < 0$, we want to show that the distance between x_{n+1} and y_{n+1} has diminished (as it would for the solutions of the reference equation). This is equivalent to showing that

$$\begin{aligned}|x_{n+1} - y_{n+1}| &= |2(x_{n+1/2} - y_{n+1/2}) - (x_n - y_n)| \\ &\leq |x_n - y_n|\,.\end{aligned} \tag{13.200}$$

The detailed symbolic algebra is a bit convoluted, but we can convince ourselves that this works at least for small h by writing the desired solution of the quadratic equation for $x_{n+1/2}$ as

$$x_{n+1/2} = \frac{x_n - h/2(t_n + h/2)}{1 - (h/2)x_{n+1/2}}\,, \tag{13.201}$$

and noticing that if $x_n < 0$, then for small h, so will be $x_{n+1/2}$, and then the denominator in the above will be larger than 1; hence, the magnitude of $x_{n+1/2}$ will be smaller than the magnitude of x_n. Similarly for y_n, and this means that the average used for the difference between x_{n+1} and y_{n+1} above will also be smaller than the difference between x_n and y_n. Thus, the implicit trapezoidal rule does seem to exhibit contractivity, as it should for this problem. See Problem 13.5. ◁

13.5.7.2 DIRKs and SDIRKs

The acronym "DIRK" refers to a diagonally implicit Runge–Kutta method. An SDIRK is a singly diagonally implicit Runge–Kutta method. The DIRK and SDIRK methods are important special cases of implicit RK methods that, in general, have easier linearizations to solve. The key is what is meant by "diagonally implicit," which we will demonstrate.

The difficulty that these methods address is that the Jacobian for a nonlinear system $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ gets blended into the nonlinear system for the stage values $\mathbf{k}_j$ in the RK method as follows, which gives rise to a system of ns equations in ns unknowns; the DIRK approach reduces the complexity of the system for the stage values to a sequence of s systems each of size $n \times n$. This is a considerable reduction, making the cost of solving the linearized systems not $O(s^3n^3)$ per step but rather $O(sn^3)$ per step, or perhaps even $O(n^3)$, as we will see.

The observation on which this technique rests is simple. If we have an implicit RK method with, say, $s = 3$ stages, then the *general* scheme is

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(\mathbf{x}_m + h(a_{1,1}\mathbf{k}_1 + a_{1,2}\mathbf{k}_2 + a_{1,3}\mathbf{k}_3)\\
\mathbf{k}_2 &= \mathbf{f}(\mathbf{x}_m + h(a_{2,1}\mathbf{k}_1 + a_{2,2}\mathbf{k}_2 + a_{2,3}\mathbf{k}_3)\\
\mathbf{k}_3 &= \mathbf{f}(\mathbf{x}_m + h(a_{3,1}\mathbf{k}_1 + a_{3,2}\mathbf{k}_2 + a_{3,3}\mathbf{k}_3)\,.
\end{aligned} \tag{13.202}$$

Each of those three equations is a *vector* equation with n equations, making $3n$ in total. Each of the $3n$ equations depends on the $3n$ unknowns $\mathbf{k}_j^i$ for $1 \le i \le n$ and $1 \le j \le s$. In this example, $s = 3$ of course. Once linearized, the Jacobian will be $3n \times 3n$ and the cost is, as stated, $O((3n)^3)$ to factor the Jacobian each time (unless, of course, it has a special structure, but that depends on the nonlinear function $\mathbf{f}$).

In contrast, a diagonally implicit Runge–Kutta (DIRK) method has its first nonlinear system only depending on $\mathbf{k}_1$:

$$\mathbf{k}_1 = \mathbf{f}(\mathbf{x}_m + ha_{1,1}\mathbf{k}_1)\,. \tag{13.203}$$

Its second equation depends only on $\mathbf{k}_2$ and on the (now known in principle from the first set of n equations) $\mathbf{k}_1$:

$$\mathbf{k}_2 = \mathbf{f}(\mathbf{x}_m + h(a_{2,1}\mathbf{k}_1 + a_{2,2}\mathbf{k}_2))\,. \tag{13.204}$$

Finally, the third equation depends on $\mathbf{k}_3$ and all the previously solved-for components of $\mathbf{k}_1$ and $\mathbf{k}_2$:

$$\mathbf{k}_3 = \mathbf{f}(\mathbf{x}_m + h(a_{3,1}\mathbf{k}_1 + a_{3,2}\mathbf{k}_2 + a_{3,3}\mathbf{k}_3))\,. \tag{13.205}$$

This reduces the problem to a sequence of 3 problems each only of size $n \times n$. If there are more than two stages, this is already a significant reduction in the amount of work done in solving the nonlinear equations on each step (and remember, there may be a *lot* of steps to take in the solution of the differential equation).

A further refinement happens if each of the diagonal elements $a_{1,1}$, $a_{2,2}$, $a_{3,3}$ is equal to the same thing [call it γ, in accordance with the traditional notation used, for example, in Hairer and Wanner (2002)]. If that is true, then the Jacobians of each system have the same structure, $\mathbf{I} - h\gamma\mathbf{J}$, and there is some hope of reusing the factoring itself; of course, the evaluation of the Jacobian would happen at different places if an exact Newton method was being used, but even so the factoring at one place can be used as a preconditioner for the solution at another. Moving to a DIRK will save a factor of s^2 in the linear algebra, and moving to an SDIRK might save a further factor of s, and amortize a factor of n across several steps.

Chapter IV.6 of Hairer and Wanner (2002) discusses in detail the construction of SDIRK methods and, in particular, gives a lovely method using the trees that you have now seen used for *explicit* methods (plus an elegant trick) to construct SDIRKs. They give an example of a five-stage fourth-order method and show how to solve the order conditions to get a three-parameter family of such methods. They give a specific good choice of a method that is A- and L-stable and has a small local error coefficient; they give more details of another choice, which is also A- and L-stable but has “nicer” coefficients rather than a small local error coefficient. They also make an implementation in Fortran freely available under the name SDIRK4. For that method, they also give a continuous extension; however, this extension is only third-order accurate (which isn't totally clear from the text). To be concrete, the SDIRK that they describe has the Butcher tableau

$$\begin{array}{r|ccccc}
1/4 & 1/4 & & & & \\
3/4 & 1/2 & 1/4 & & & \\
11/20 & 17/50 & -1/25 & 1/4 & & \\
1/2 & 371/1360 & -137/2720 & 15/544 & 1/4 & \\
1 & 25/24 & -49/48 & 125/16 & -85/12 & 1/4 \\
\hline
 & b_1(\theta) & b_2(\theta) & b_3(\theta) & b_4(\theta) & b_5(\theta)
\end{array}, \qquad (13.206)$$

where the b_i are constructed to be identical to the final stage values. More, they give degree-4 polynomials $b_i(\theta)$ such that the continuous extension $x(\theta) = x_m + h\sum_{j=1}^{5} b_j(\theta)\mathbf{k}_k$ is third-order accurate in the interval $0 < \theta < 1$ but interpolates the fourth-order-accurate x_{m+1}. We do not give that interpolant here, in part because there seems to be an error in four of the coefficients printed in the 1991 edition of that book, but mostly because for our purposes even when the misprints are corrected, the resulting residual is not continuous. Naively, cubic Hermite interpolation instead seems attractive. It turns out, however, that there can be bad behavior for stiff problems in that the slopes at either endpoint encourage spurious oscillation in the interpolant (this is discussed in Shampine and Reichelt (1997), for example); an alternative using a shape-preserving rational interpolant using the second barycentric form may also be satisfactory.

13.5.7.3 Rosenbrock Methods

We now come to Rosenbrock methods, which in some sense are even simpler than SDIRK methods. The MATLAB ODE Suite has a Rosenbrock method implemented in the routine `ode23s`.

A useful way to look at a Rosenbrock method is to think of it as a DIRK method where we take exactly one Newton step in order to approximately solve the nonlinear equations for the stage values. That is, we write

$$\mathbf{k}_i = h\mathbf{f}\left(x_n + \sum_{j=1}^{i-1} a_{i,j}\mathbf{k}_j + a_{i,i}\mathbf{k}_i\right) \tag{13.207}$$

for $1 \leq i \leq s$. Linearizing, this gives

$$\mathbf{k}_i = h\mathbf{f}(\mathbf{g}_i) + ha_{i,i}\mathbf{J}_{\mathbf{f}}(\mathbf{g}_i)\mathbf{k}_i\,, \tag{13.208}$$

where the vector $\mathbf{g}_i$ is the explicit stage value

$$\mathbf{g}_i = x_n + \sum_{j=1}^{i-1} a_{i,j}\mathbf{k}_j\,. \tag{13.209}$$

Solving this linear system of equations for the vector $\mathbf{k}_i$ is (as stated) equivalent to one step of Newton's method applied to the original DIRK.

Once we start looking at this method, though, several opportunities for efficiency present themselves. The main point of attack is the Jacobian; we wish to evaluate that as few times as necessary, and certainly to factor the matrix $\mathbf{I} - ha\mathbf{J}$ as few times as necessary. A Rosenbrock method is therefore *defined* as a method in the following class.

Definition 13.8. An s-stage Rosenbrock method is an explicit method of the form

$$\mathbf{k}_i = h\mathbf{f}(\mathbf{g}_i) + h\mathbf{J}_{\mathbf{f}}(\mathbf{x}_n)\sum_{j=1}^{i} \gamma_{i,j}\mathbf{k}_j\,, \tag{13.210}$$

for $1 \leq i \leq s$, and the step is advanced with $x_{n+1} = x_n + \sum_{j=1}^{s} b_j\mathbf{k}_j$. $\triangleleft$

The standard notation uses $\gamma_{i,j}$ differently than what we have heretofore been using—these are not bounds for rounding errors—but this hopefully will not add to the confusion caused by the use, different again, of $\gamma(\tau)$ as the parameter in the order condition for the tree τ.

Trees and elementary differentials can be used to write down order conditions for Rosenbrock methods, including higher-order methods. Solution of the resulting equations can give parametric families of solutions. Embedded methods can be chosen and used for error control. There are several tricks for efficient implementation discussed in Hairer and Wanner (2002). The method implemented in `ode23s` is low-order and has a free interpolant.

This is somehow a natural way to come at the Rosenbrock methods, but once you have thought of them this way, you realize that they are a kind of *explicit* method, except that they take into account the Jacobian matrix and so they are a kind of explicit RK method that has been tailored to the specific problem (and more than that, to the specific point from which the solution is proceeding). As such, they could benefit from study in their own right.

They turn out to be relatively easy to implement and quite remarkably effective for stiff problems even when numerical approximations to the Jacobian matrices are used (for example, by finite differences, as the Rosenbrock code `ode23s` uses by default). We look at a simple example.

Example 13.8. Consider solving the van der Pol equation

$$y'' - \mu(1-y^2)y' + y = 0 \tag{13.211}$$

with a parameter value $\mu = 10^6$ and initial conditions $y(0) = 2$ and $y'(0) = -0.66$. These are the parameter values used in Hairer and Wanner (2002) for demonstrating some curious step-control difficulties with a higher-order Rosenbrock method than is implemented in `ode23s`. When we solve this problem using `ode23s` on $0 \le t \le 3 \times 10^6$, we get the results in Fig. 13.11.

Instead of using the built-in interpolant, we compute the second derivatives at each mesh point from the returned solution values and use PQHIP from Chap. 8; this allows us to compute the second derivative of the piecewise quintic Hermite interpolant and substitute it in to get a residual in the second-order equation itself, not the mathematically equivalent first-order form. This gives $\Delta(t)$ in

$$z'' - \mu(1-z^2)z' + z = \Delta(t)\,. \tag{13.212}$$

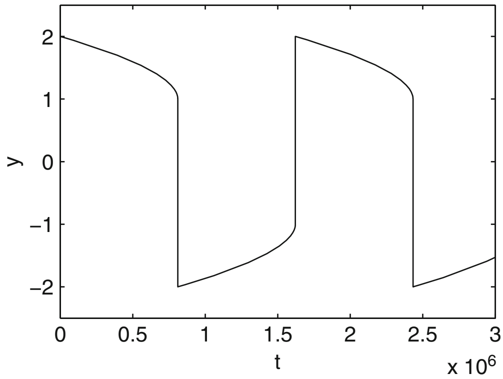


Fig. 13.11 The solution to the (very stiff) van der Pol equation (13.211) with $\mu = 10^6$ by the low-order Rosenbrock method implemented in `ode23s`, using default tolerances

This is graphed in Fig. 13.12, scaled by the maximum of 1, $|z|$, $|z''|$, and $\mu|1-z^2||z'|$. We see that the residual seems quite large in the regions of rapid change, and that perhaps this merits further investigation (Fig. 13.13). ◁

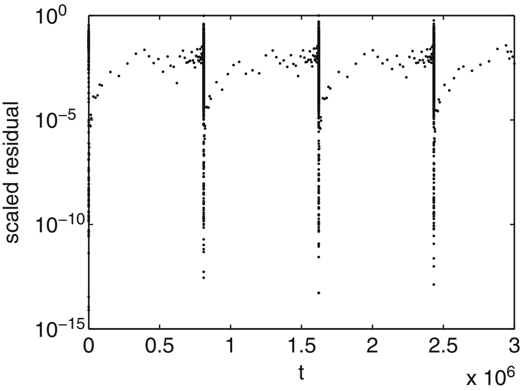


Fig. 13.12 Scaled residual in the solution to the (very stiff) van der Pol equation (13.211) with $\mu = 10^6$ by the low-order Rosenbrock method implemented in `ode23s`, using default tolerances. The residual appears to be too large in the regions of rapid change and may deserve further investigation

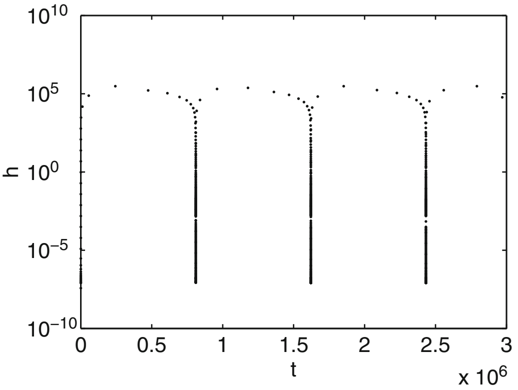


Fig. 13.13 Step sizes used in the solution to the (very stiff) van der Pol equation (13.211) with $\mu = 10^6$ by the low-order Rosenbrock method implemented in `ode23s`, using default tolerances. The step-size control appears to be doing a reasonable job. Step sizes change by 10 orders of magnitude over the course of integration

13.6 Multistep Methods

Multistep methods (and their relatives) are among the most commonly used in high-quality implementations. The MATLAB code `ode113`, for instance, is efficient and accurate. The stiff code `ode15s`, which we have frequently used, uses a modification of the BDF methods discussed below. Both of these codes deserve some comment, and this section is meant to provide grounds for that, from the point of view developed so far. Thus, once again, the presentation in this section is not quite standard. To preserve harmony with the rest of this book, it uses Hermite interpolation in barycentric form in order to develop the formulas and to convey the main ideas.

The barycentric formulæ method used here, while not standard in textbooks, has its advantages. The method seems to have first been used in Butcher (1967). We like it because it allows great flexibility in generating formulæ: It is not restricted to constant step size, it provides an interpolating polynomial in barycentric form, and it is easily programmed in a computer algebra system—all that is needed is the ability to compute partial fractions. As we will see below, there may be some theoretical advantages, too, but these have not yet been explored (and may only be marginally helpful there anyway, if at all).

One-step methods such as Runge–Kutta methods (or Taylor series methods, when it comes to that) seem inefficient when one realizes that previously computed values and derivatives of the solution are not being used. It seems a waste. For many problems, much greater efficiency can be obtained if that information is retained and reused. Indeed, the oldest methods of this very popular type predate Runge and Kutta, apparently going as far back as Bashforth's work in 1835.[11] The MATLAB exemplar of this method is `ode113`, which is used in much the same way as the other MATLAB codes we have discussed so far. More details can be found in Shampine and Reichelt (1997).

The basic idea of multistep methods is simple: Fit a polynomial to some of the data available, perhaps the values and/or derivatives at the last k steps,

$$y(t_{n-1}), y(t_{n-2}), \ldots, y(t_{n-k}), y'(t_{n-1}), y'(t_{n-2}), \ldots, y'(t_{n-k}),$$

and use the fitted polynomial to predict the value of $y(t_n)$. A useful (indeed, essential) variation on this idea is to include the unknown $y(t_n)$ and $y'(t_n) = f(t_n, y(t_n))$ in the polynomial fit—this gives an implicit equation to solve for $y(t_n)$, as happens also with one-step implicit methods.

Once the polynomial has been found, it is a simple matter to compute the residual and verify that a good reverse-engineered problem has been solved.

Example 13.9. For example, suppose we start by looking for a polynomial that fits y_n, y_{n-1}, y_{n-2} and f_n. To find our formula, consider the partial fraction expansion

$$\frac{1}{(z-t_n)^2(z-t_{n-1})(z-t_{n-2})} = \frac{1}{w(z)} \tag{13.213}$$

or, equivalently, consider the contour integral

$$0 = \frac{1}{2\pi i}\oint_C \frac{y(z)}{w(z)}dz = \beta_{00}y_n + \beta_1 y_{n-1} + \beta_2 y_{n-2} + \beta_{01} f_n, \tag{13.214}$$

where

$$\beta_{00} = \frac{\tau_{n-2} - 2\tau_n + \tau_{n-1}}{(\tau_n - \tau_{n-1})^2(\tau_n - \tau_{n-2})^2}$$

[11] See Hairer and Wanner (2002) for historical details.

$$\beta_{01} = \frac{1}{(\tau_n - \tau_{n-1})(\tau_n - \tau_{n-2})}$$

$$\beta_1 = \frac{1}{(\tau_n - \tau_{n-1})^2(\tau_{n-1} - \tau_{n-2})},$$

and

$$\beta_2 = \frac{1}{(\tau_n - \tau_{n-2})(\tau_{n-2} - \tau_{n-1})}\,.$$

Solving for y_n, we find, with $h_n = \tau_n - \tau_{n-1}$ and similarly for h_{n-1},

$$y_n \;=\; \frac{(h_n + h_{n-1})h_n}{2h_n + h_{n-1}} f_n \;+\; \frac{(h_n + h_{n-1})^2}{h_{n-1}(2h_n + h_{n-1})} y_{n-1} \;-\; \frac{h_n^2}{h_{n-1}(2h_n + h_{n-1})} y_{n-2}\;.$$

If it so happens that $h_{n-1} = h_n = h$, this simplifies to

$$y_n = \frac{2}{3}hf_n + \frac{4}{3}y_{n-1} - \frac{1}{3}y_{n-2}\,. \tag{13.215}$$

This is the well-known formula called the backward differentiation formula (BDF) of order 2. This is an implicit formula because f_n involves the unknown y_n. The standard way of deriving this formula (which is different than the method used here) uses backward difference formulæ (hence the name) and gives a nice recurrence relation and a collection of results on these lovely formulae.

The reader may notice some familiarity in this argument, and be reminded of the construction of the differentiation matrices for polynomials expressed in the Lagrange basis. There is indeed a connection, which we will touch on later. ◁

We continue with our Lagrange/Hermite interpolation program as usual. Given $y_n, f_n, y_{n-1}, y_{n-2}$ (and possibly even f_{n-1} and f_{n-2}), we have the interpolant

$$z(t) = \frac{\displaystyle\sum_{i=0}^{2}\sum_{j=0}^{1}\sum_{k=0}^{j} \beta_{n-i,j}\rho_{n-i,k}(t - t_{n-i})^{k-j-1}}{\displaystyle\sum_{i=0}^{2}\sum_{j=0}^{1} \beta_{n-i,j}(t - t_{n-i})^{-j-1}}, \tag{13.216}$$

and, if we use the differentiation matrix, its derivative vector $= \mathbf{D}\boldsymbol{\rho}$ gives

$$\dot{z}(t) = \frac{\displaystyle\sum_{i=0}^{2}\sum_{j=0}^{1}\sum_{k=0}^{j} \beta_{n-i,jn-i,k}(t - t_{n-i})^{k-j-1}}{\displaystyle\sum_{i=0}^{2}\sum_{j=0}^{1} \beta_{n-i,j}(t - t_{n-i})^{-j-1}}\,. \tag{13.217}$$

These can be analytically simplified, if desired, to get

$$\begin{aligned} z(t) = &-\frac{(\theta-1)^2\,\theta\, y_{n-2}}{(1+r)^2\, r} + \frac{(\theta-1)^2\,(\theta+r)\, y_{n-1}}{r} \\ &- \frac{(2\,\theta-3+r\theta-2\,r)\,(\theta+r)\,\theta\, y_n}{(1+r)^2} + \frac{h\,(\theta-1)\,\theta\,(\theta+r)\, f_n}{1+r}, \end{aligned} \tag{13.218}$$

where $t = \tau_{n-1} + \theta h$, $h = \tau_n - \tau_{n-1}$, and $rh = \tau_{n-1} - \tau_{n-2}$. This way, $-r \leq \theta \leq 1$ covers both subintervals. The differentiation matrix from which we get the vector of derivative values is

$$\mathbf{D} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -2\,\frac{3\,r+3+r^2}{h^2(1+r)^2} & 2\,\frac{2+r}{h(1+r)} & 2\,\frac{1+r}{rh^2} & -2\,\frac{1}{h^2(1+r)^2\,r} \\ \frac{2\,r^2+3\,r}{(1+r)^2 h} & -\frac{r}{1+r} & -\frac{-1+2\,r}{rh} & -\frac{1}{(1+r)^2 rh} \\ -\frac{r(r+3)}{h(1+r)} & r & \frac{(1+r)^2}{rh} & -\frac{3\,r+1}{hr(1+r)} \end{bmatrix}, \tag{13.219}$$

and this allows direct use of the barycentric form. Alternatively, an analytically simplified derivative can be used:

$$\begin{aligned} \dot{z}(t) = &-\frac{(\theta-1)\,(3\,\theta-1)\, y_{n-2}}{(1+r)^2\, rh} + \frac{(\theta-1)\,(3\,\theta+2\,r-1)\, y_{n-1}}{rh} \\ &- \frac{(\theta-1)\left(3\,r\theta+6\,\theta+2\,r^2+3\,r\right) y_n}{(1+r)^2\, h} + \frac{\left(3\,\theta^2+2\,r\theta-2\,\theta-r\right) f_n}{1+r}. \end{aligned} \tag{13.220}$$

Using either form, the residual $\Delta(t) = \dot{z}(t) - f(z(t))$ may be computed at any point. Doing this for this formula in series using MAPLE, we find

$$\Delta(t) = \frac{2}{2+r}(1-\theta)\theta(\theta+r)(\mathbf{f}_{z,z}(\mathbf{f},\mathbf{f}) + \mathbf{J_f}\cdot\mathbf{J_f}\cdot\mathbf{f}))h^2 + O(h^3)\,, \tag{13.221}$$

denoting the Jacobian matrix by $\mathbf{J_f}$ and the second bilinear form by $\mathbf{f}_{z,z}$. This series result is only obtainable if the starting value

$$y_{n-2} = y_{n-1} - rhf(y_{n-1}) + \frac{(rh)^2}{2}\mathbf{J_f}\cdot\mathbf{f} \tag{13.222}$$

is used; this is asymptotically the local reference solution, predicted backward from y_{n-1}, correct to $O(h^2)$. If a different value is used, the residual is *not* $O(h^2)$ small.

This highlights an important point: To get a linear multistep method (LMM) such as this one to be second-order, the starting values $y_0, y_1, \ldots, y_{k-2}$ must be accurate enough that a polynomial (or other interpolant) may be fit to the data while maintaining the correct order residual. That is, the polynomial fit of the starting values alone must already nearly satisfy the differential equation. If the starting data are only $O(h)$-accurate, then (unless the problem is so well-conditioned that errors are damped out as the integration proceeds) the overall method will also be just $O(h)$.

We remark that modern codes accept this limitation, but they try to ensure that the constant hidden by the O-symbol is small enough that it doesn't matter.

Definition 13.9. A linear multistep method of the form

$$\begin{aligned} 0 = \alpha_0 y_n + \alpha_1 y_{n-1} + \alpha_2 y_{n-2} + \ldots + \alpha_k y_{n-k} \\ + h(\beta_0 f(y_n) + \beta_1 f(y_{n-1}) + \ldots + \beta_k f(y_{n-k})) \end{aligned} \tag{13.223}$$

is called a linear k-step method if $|\alpha_k| + |\beta_k| \neq 0$ and is called implicit if $\beta_0 \neq 0$ (because then to find y_n, one must solve (13.223), which is nonlinear in general). The method is of order p if, when there is an interpolant $s(t)$ to the starting values (t_{n-j}, y_{n-j}), $1 \leq j \leq k$, with residual $\Delta_s(t) = s'(t) - f(s(t))$ of size $\|\Delta_s(t)\| \leq Kh^p$ for some constant K, then there exists a solution y_n of (13.223) such that an interpolant $z(t)$ can be found for $(t_{n-j}, y_{n-j}), 0 \leq j \leq k$ (note the wider interval now, going down to $j = 0$) with

$$\|\Delta_z(t)\| \leq \hat{K}h^p \tag{13.224}$$

for some $\hat{K}$ only at most modestly larger than K. Here h is taken to be a representative mesh width, and the relevant limit is as $h \to 0$. ◁

This presentation and definition differ from Butcher (2008b), Hairer and Wanner (2002), and Hairer et al. (1993). With these definitions, we could say here that an LMM was *convergent in residual* for an ODE with Lipschitz continuous f on an interval $a \leq t \leq b$ if, for every $\varepsilon > 0$, there existed a mesh $a = t_0 < t_1 < \cdots < t_n = b$ with $\|\Delta(t)\| \leq \hat{K}h^p \leq \varepsilon$ for every consecutive $k+1$ mesh points (because n is finite, we may take K to be the largest that appears in the definition). Proving LMM convergent (even with the standard definitions) is somewhat technical. See chapters V.7 and V.8 of Hairer and Wanner (2002) for an impressive collection of results (including for some nonlinear problems). See also Shampine (2002).

Here we do *not* do this. Instead, we give the reader tools to verify that the particular solution just computed (by any LMM) to the reader's special problem of interest has been solved to the reader's satisfaction: Namely, just compute the residual, and the condition number. It would be nice if LMM were always convergent in residual; it would be nice if we could prove it. But beyond translating some of the standard results into the notions used here, we have not done so. Moreover, there are problems (e.g., chaotic problems) that do not satisfy the hypotheses of the standard theorems, which nonetheless appear to be convergent in residual. So it appears that there is some work to do here.

13.6.1 Stability of LMM

We now look at the crucial notion of instability in LMM. We do not need a complicated example to show this. Let us look at the equal h formula relating $y_n, y_{n-1}, y_{n-1}, y_{n-3}$, and y'_{n-1} but not y'_n (so the formula is explicit). This means that the natural rational function to consider is

$$\frac{1}{(z-\tau_n)(z-\tau_{n-1})^2(z-\tau_{n-2})(z-\tau_{n-3})}, \tag{13.225}$$

where $\tau_k = \tau_0 + kh$. After the usual partial fraction expansion and simplification, we get

$$y_n = 3hy'_{n-1} - \frac{3}{2}y_{n-1} + 3y_{n-2} - \frac{1}{2}y_{n-3}, \tag{13.226}$$

which we expect to be exact for polynomials of degree 2 or less. Indeed, when we try it out in exact arithmetic on the problems $y' = 0$, $y' = 1$, $y' = t$, and $y' = t^2$, we do indeed get the exact solutions, but we do not for $y' = t^3$. Hence, this method is second-order accurate.

Now, let's consider the exact solution of $y' = 0$ again, but this time taking into account rounding errors. For this equation, we can solve the recurrence relation exactly, as we had claimed above. The recurrence equation is

$$y_n = -\frac{3}{2}y_{n-1} + 3y_{n-2} - \frac{1}{2}y_{n-3}\,. \tag{13.227}$$

As usual for linear recurrence relations, we look at solutions that are powers. Putting $y_n = \xi^n$ and dividing the result by ξ^{n-2} give

$$\xi^3 = -\frac{3}{2}\xi^2 + 3\xi - \frac{1}{2}\,. \tag{13.228}$$

This cubic equation has roots 1 and $-5/4 \pm \sqrt{33}/4$. It turns out to be a crucial fact that one of the 'extra' roots, namely, $-5/4 - \sqrt{33}/4$, is larger than 1 in magnitude.

The *general* solution of the recurrence relation is a linear combination of powers of all three roots:

$$y_n \doteq c_1 1^n + c_2(-5/4 + \sqrt{33}/4)^n + c_3(-5/4 - \sqrt{33}/4)^n\,. \tag{13.229}$$

If exact starting conditions and exact arithmetic are used, then the correct solution $y_n = c_1$ comes out; this is the exact solution of $y' = 0$. If, however, a tiny error is made in starting the integration, we don't get $c_2 = 0$ and $c_3 = 0$. For instance, we might start with $y_0 = c_1, y_1 = c_1 + \varepsilon, y_2 = c_1$; then neither of the coefficients c_2 or c_3 will be zero. Because $c_3\xi^n$ grows exponentially, eventually this so-called parasitic root dominates, swamping the true solution. This method, although accurate to as high an order as possible, is *unstable*.

A linear multistep method that can solve $y' = 0$ without introducing parasitic roots larger than 1, or multiple roots with magnitude 1, so that rounding or other errors are not amplified as the iteration proceeds, is called 0-stable, or alternatively D-stable (in honor of Germund Dahlquist). Methods that introduce parasitic roots that amplify errors are called unstable.

Let us now look at an implicit method, fitting $y_n, y_{n-1}, y_{n-2}, y_{n-3}$, and y'_n (but not y'_{n-1} this time). We get

$$y_n = \frac{6}{11}hy'_n + \frac{18}{11}y_{n-1} - \frac{9}{11}y_{n-2} + \frac{2}{11}y_{n-3}\,. \tag{13.230}$$

This turns out to be a well-known method in another guise; this is the third-order BDF method. This time, when we solve $y' = 0$, we get

$$y_n = c_1 1^n + c_2\left(\frac{7+i\sqrt{39}}{22}\right)^n + c_3\left(\frac{7-i\sqrt{39}}{22}\right)^n\,. \tag{13.231}$$

These parasitic roots are smaller than 1 in magnitude ($= \sqrt{22}/11 \doteq 0.42640$). Hence, the influence of small errors will be damped. As a result, the implicit formula is to be preferred if we can find an acceptable way to solve the (in general nonlinear) equation (13.230) when $y'_n = f(t_n, y_n)$ is used.

A standard and effective method is to use both methods together! That is, we use an explicit method to get an initial guess for the solution of the implicit method. In the vernacular, we *predict* y_n using the unstable method, evaluate $f(t_n, \hat{y}_n)$ using the predicted value, and then *correct* using the implicit formula:

$$\hat{y}_n = 3hf(t_{n-1}, y_{n-1}) - \frac{3}{2}y_{n-1} + 3y_{n-2} - \frac{1}{2}y_{n-3} \tag{13.232}$$

$$y_n = \frac{6}{11}hf(t_n, \hat{y}_n) + \frac{18}{11}y_{n-1} - \frac{9}{11}y_{n-2} + \frac{2}{11}y_{n-3}\,. \tag{13.233}$$

Naturally enough, this is called a *predictor–corrector method*. This automatically fixes the zero-stability problem [even with just one PEC cycle—that is, predict, evaluate, correct—or we could correct again, if we wished ($P(EC)^2$ mode)—and, of course, we will need $f(t_n, y_n)$ on the next step anyway, so $P(EC)^nE$ mode is the usual way to proceed]. Any predictor–corrector method is an explicit method, however, even though it is influenced by a fully implicit method; hence, it may suffer some difficulty with stiff problems.

13.6.2 More Stability Issues

Let us use a corrector to improve the (very bad) method (13.226), and try it out on one step of the Dahlquist test problem $y' = \lambda y$. This, of course, includes the previous problem, when $\lambda = 0$, so this is a generalization: Methods that are stable for this problem will be 0-stable too. Here $\mu = h\lambda$, and we will think about stiff stability (no longer about 0-stability) in the complex μ-plane. We have

$$\hat{y}_n = 3\mu y_{n-1} - \frac{3}{2}y_{n-1} + 3y_{n-2} - \frac{1}{2}y_{n-3}$$
$$y_n = \frac{6}{11}\mu(3\mu y_{n-1} - \frac{3}{2}y_{n-1} + 3y_{n-2} - \frac{1}{2}y_{n-3}) + \frac{18}{11}y_{n-1} - \frac{9}{11}y_{n-2} + \frac{2}{11}y_{n-3}\,.$$

Supposing $y_n = r^n$, we get

$$r^3 = \frac{18}{11}\mu^2 r^2 - \frac{9}{11}\mu r^2 + \frac{18}{11}\mu r - \frac{3}{11}\mu + \frac{18}{11}r^2 - \frac{9}{11}r + \frac{2}{11} \tag{13.234}$$

by dividing by r^{n-3}. Unlike with RK methods, where we use $z = h\lambda$, the convention for multistep methods is to use another letter, $\mu = h\lambda$, so z is available should we desire to use the z-transform. Of course, there is also the notational conflict with the interpolant $z(t)$, but that is less confusing.

The analysis of this stability polynomial is possible by considering a kind of bifurcation study. Suppose that all three roots of the polynomial (as a polynomial in r) are less than 1 in magnitude, for some special value of μ. Then consider altering μ, moving it away from that special value. The magnitudes of the roots will (in general) change; it's possible that one or more of the roots will increase in magnitude until the magnitude is $|r| = 1$. Now, the curve parameterized by $r = e^{i\theta}$ is a description of the locus of roots of critical magnitude, and we are thus interested in just when (if ever) any of the three root paths cross this locus. On substituting $r = e^{i\theta}$ into (13.234), we find that the equation

$$0 = \frac{18}{11}e^{2i\theta}\mu^2 + \left(-\frac{9}{11}e^{2i\theta} + \frac{18}{11}e^{i\theta} - \frac{3}{11}\right)\mu + \left(-e^{3i\theta} + \frac{18}{11}e^{2i\theta} - \frac{9}{11}e^{i\theta} + \frac{2}{11}\right)$$

defines the boundary of the region inside which $r^n \to 0$. See Fig. 13.14 and Problem 13.33.

This example method is, in fact, pathetic—its stability region is not even as large as the stability region for the explicit fourth-order Adams method, which we will see later. The problem is that the predictor is not 0-stable. Certainly, the implicit method, if used by itself, is very good (this is BDF3, as we said). Its stability polynomial is

$$\mu = \frac{11}{6} - 3r^{-1} + \frac{3}{2}r^{-2} - 2r^{-3}\,. \tag{13.235}$$

If we let $r = \cos(\theta) + i\sin(\theta)$, we find the locus in the μ-plane at which one of the parasitic roots has magnitude 1. When you do this (see Exercise 13.15), you find that the implicit method is stable *outside* a bulgy apple-shaped region mostly in the right half-plane (this method is not A-stable, but nearly). If you plot the region together with the region for the predictor–corrector method, you find that the stability region has been greatly reduced, indeed.

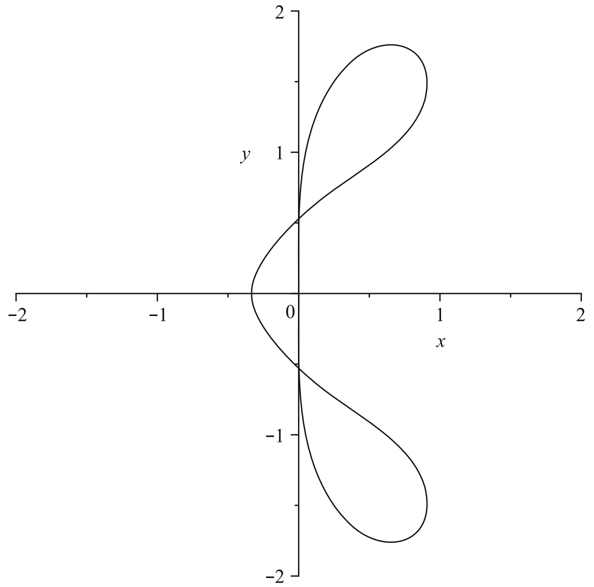


Fig. 13.14 The boundary of the stability region for the nonstandard (and really bad) predictor–corrector method in the complex μ-plane ($\mu = x + iy$). Since this boundary crosses itself, a separate investigation of each of the loops (and the exterior) is needed. A simple way is to take a sample μ value from each region, and compute the zeros of Eq. (13.234). When this is done, we find that only the small quasitriangular piece touching $(x, y) = (-1/3, 0)$ and bordered by a curve that is nearly straight up and down—but not quite, see Problem 13.33—contains any μ values for which *all three* of the roots of (13.234) are less than 1 in magnitude

13.6.3 Variable Step-Size Multistep Methods, Derived Using Contour Integral Methods

One standard derivation of the Adams methods, which is characterized by

$$y_n = \alpha_1 y_{n-1} + h \sum_{i=0}^{k} \beta_i f(y_{n-i}) \,, \tag{13.236}$$

and several variations thereof, uses quadrature rules on the integral form

$$y(t) = y_{n-1} + \int_{t_{n-1}}^{t} f(y(\tau)) d\tau \tag{13.237}$$

of $y' = f(y)$ on polynomial interpolants through $(\tau_{n-i}, f(\tau_{n-i}))$, for $0 \le i \le k$. Similarly, a standard derivation of the BDF methods (backward difference formulæ) instead uses polynomial interpolation of y_{n-i} and insists that the differential equation be satisfied at one point.

It is instructive to consider an alternative derivation using contour integrals (since we have the machinery readily at hand in this book). BDF methods turn out to be easier, and so we start with them. We have already done this for the second-order BDF method, in Example 13.9. It is very simple to use a CAS to derive BDF for higher orders just by doing a partial fraction expansion of

$$\frac{1}{(z-t_n)^2 \prod_{j=1}^{k}(z-t_{n-j})}\,, \tag{13.238}$$

but it's not even hard by hand. In fact, we've done it already when we computed the elements of the differentiation matrix for polynomials expressed in the Lagrange basis (see Eq. 11.22). For didactic purposes, we do the computation again in a manner specialized to the application at hand. We continue by writing the right-hand side:

$$= \frac{\beta_{n,0}}{z-t_n} + \frac{\beta_{n,1}}{(z-t_n)^2} + \sum_{j=1}^{k} \frac{\beta_{n-j}}{z-t_{n-j}}\,, \tag{13.239}$$

where

$$\beta_{n-j} = (t_{n-j}-t_n)^{-2} \prod_{\substack{\ell=1\\ \ell\neq j}}^{k} (t_{n-j}-t_{n-\ell})^{-1}\,, \tag{13.240}$$

for $1 \le j \le k$, and

$$\beta_{n,1} = \prod_{j=1}^{k} (t_n - t_{n-j})^{-1}\,. \tag{13.241}$$

The only "difficult" one, $\beta_{n,0}$, succumbs to a simple derivative computation:

$$\begin{aligned} u &= \prod_{j=1}^{k} (z-t_{n-j})^{-1} \\ \ln u &= -\sum_{j=1}^{k} \ln(z-t_{n-j}) \\ \frac{u'}{u} &= -\sum_{j=1}^{k} \frac{1}{z-t_{n-j}}. \end{aligned}$$

Then, because

$$\frac{u(z)}{(z-t_n)^2} = \frac{u(t_n)}{(z-t_n)^2} + \frac{u'(t_n)}{(z-t_n)} + O(1)\,,$$

we have

$$u'(t_n) = \beta_{n,0} = -\beta_{n,1} \sum_{j=1}^{k} \frac{1}{t_n - t_{n-j}}\,. \tag{13.242}$$

It follows that

$$0 = \frac{1}{2\pi i} \oint_C \frac{p(z)}{(z-t_n)^2 \prod_{j=1}^{k} (z - t_{n-j})}\, dz \tag{13.243}$$

for all polynomials with $\deg p \leq k$, and so we have

$$0 = \beta_{n,1} p'(t_n) + \beta_{n,0} p(t_n) + \sum_{j=1}^{k} \beta_{n-j} p(t_{n-j}), \tag{13.244}$$

or, rewriting this expression,

$$\left(\sum_{j=1}^{k} \frac{1}{t_n - t_{n-j}} \right) y_n = f_n + \sum_{j=1}^{k} \frac{\beta_{n-j}}{\beta_{n,1}} y_{n-j}\,. \tag{13.245}$$

These formulæ are equivalent, but seem to us to be rather simpler than those derived using divided differences. For example, one sees immediately that none of the α_{n-j} is zero.

The connection with the differentiation matrix is that the formulæ are exactly the same: however, we solve for y_n and divide by its coefficient, instead of solving for f_n as we did to derive the entries in the differentiation matrix. The lesson is the same: If you can compute partial fractions, you can develop variable-step BDF methods.

These methods have nontrivial regions of stability for $1 \leq k \leq 6$, but do become unstable for $k > 6$ in the equally spaced case; see Hairer and Wanner (2002) for a proof.

13.6.4 Adams' Methods

Now, let us go on to the more interesting problem of using contour methods to derive Adams' methods. Using this technique, one may derive a great many related formulæ. We use John Butcher's idea of choosing $B(z)$ to make certain residues zero. Put

$$w(z) = \prod_{i=0}^{k} (z - t_{n-i})^2 \tag{13.246}$$

and

$$\frac{B(z)}{w(z)} = \sum_{i=0}^{k} \left(\frac{\alpha_{i,0}}{z - t_{n-i}} + \frac{\alpha_{i,1}}{(z - t_{n-1})^2} \right) . \tag{13.247}$$

By scaling, we may insist, say, that $B(t_{n-1}) = 1$. Since the Adams formulæ do not contain $y_{n-2}, y_{n-3}, \ldots, y_{n-k}$, we must have $\alpha_{i,0} = 0$ for $2 \leq i \leq k$. These $k-1$ constraints, plus the constraint $B(t_{n-1}) = 1$, mean that we can take $B(z)$ to be of degree $k-1$ (it may happen that $\deg B < k-1$, but we discount this possibility). If the barycentric weights for

$$\frac{1}{w(z)} = \sum_{i=0}^{k} \left(\frac{\beta_{i,0}}{z - t_{n-i}} + \frac{\beta_{i,1}}{(z - t_{n-i})^2} \right) \tag{13.248}$$

are first computed, then by resummation, we have

$$\alpha_{i,0} = \beta_{i,0} B(t_{n-i}) + \beta_{i,1} B'(t_{n-i}) . \tag{13.249}$$

Thus, for $2 \leq i \leq k$, we must have

$$B'(t_{n-i}) = \left(2 \sum_{\substack{\ell=0 \\ \ell \neq i}}^{k} \frac{1}{t_{n-i} - t_{n-\ell}} \right) B(t_{n-i}) . \tag{13.250}$$

This is a curious kind of interpolation condition and, who knows, it might even be analytically soluble. But the following computational procedure suffices to find $B(t_{n-i})$ for $2 \leq i \leq k$ and thus to specify B uniquely. After that,

$$\alpha_{i,1} = \beta_{i,1} B(t_{n-i}) \tag{13.251}$$

gives us the coefficients for $1 \leq i \leq k$, while a separate Lagrange evaluation gives us $B(t_n)$ and a separate use of the differentiation matrix $\mathbf{D}$ on $t_{n-1}, t_{n-2}, \ldots, t_{n-k}$ gives us $B'(t_n)$, completing the method.

But first we need the $B(t_{n-i})$. Let $\mathbf{D}$ be the differentiation matrix mentioned above. Then, we remind you that

$$D_{ij} = -\frac{(t_{n-j} - t_{n-i})^{-1} \beta_j}{\beta_i} \tag{13.252}$$

if $i \neq j$, while

$$D_{ii} = -\sum_{i \neq j} D_{ij} . \tag{13.253}$$

Here, the single-indexed β_is are the Lagrange barycentric weights on the nodes $t_{n-1}, t_{n-2}, \ldots, t_{n-k}$:

$$\beta_i = \prod_{\substack{j=1\\ j\neq i}}^{k} (t_{n-i} - t_{n-j})^{-1} \,. \tag{13.254}$$

The Adams methods can then be found from

$$\begin{aligned} 0 &= \frac{1}{2\pi i} \oint \frac{B(z)}{w(z)} p(z) dz \\ &= \alpha_{n,0} p(t_n) + \alpha_{n,1} p'(t_n) + \alpha_{n-1,0} p(t_{n-1}) + \alpha_{n-1,1} p'(t_{n-1}) + \sum_{j=2}^{k} \alpha_{n-j,1} p'(t_{n-j}) \end{aligned}$$

(here we have switched notation to $\alpha_{n-i,0}$ instead of $\alpha_{i,0}$). However, to find the $\alpha_{n-j,\ell}$, we must find $B(t_{n-j})$, $2 \leq j \leq k$, then $B'(t_{n-1})$, then $B(t_n)$, and $B'(t_n)$.

After forming the differentiation matrix $\mathbf{D}$, note that

$$\mathbf{D} \begin{bmatrix} 1 \\ B(t_{n-2}) \\ B(t_{n-3}) \\ \vdots \\ B(t_{n-k}) \end{bmatrix} = \begin{bmatrix} B'(t_{n-1}) \\ B'(t_{n-2}) \\ \vdots \\ B'(t_{n-k}) \end{bmatrix} \tag{13.255}$$

and that the last $k-1$ entries of the vector on the right are known, since

$$B'(t_{n-i}) = -\frac{\beta_{i,0}}{\beta_{i,1}} B(t_{n-i}) = \left(2 \sum_{\substack{\ell=0\\ \ell\neq i}}^{k} \frac{1}{t_{n-i} - t_{n-\ell}} \right) B(t_{n-i}) \,. \tag{13.256}$$

Form the $(k-1) \times (k-1)$ matrix

$$\mathbf{A} = \mathbf{D}(2:k, 2:k) + \operatorname{diag}(\frac{\beta_{i,0}}{\beta_{i,1}}, i = 2..k) \,. \tag{13.257}$$

Then,

$$\mathbf{A} \begin{bmatrix} B(t_{n-2}) \\ B(t_{n-3}) \\ \vdots \\ B(t_{n-k}) \end{bmatrix} = \left[\mathbf{D}(2:k, 1) \right] \tag{13.258}$$

defines the $B(t_{n-k})$ uniquely if $\mathbf{A}$ is nonsingular. Curiously, the matrix $\mathbf{A}$ sometimes *is* singular, for certain arrangements of steps $t_{n-1}, t_{n-2}, \ldots, t_{n-k}$, even when the interpolation can be done after all; this is a problem with the process, not the problem. This means that this technique may also suffer from ill-conditioning on occasion.

Take, for example, $k = 2$. In this case, only one residue must be set to zero, and we find that we can do this only if

$$t_n + 2t_{n-1} - 3t_{n-2} \neq 0. \tag{13.259}$$

If the t_k are monotonic, this is no problem because this quantity is always positive: But if the t_k are not monotonic, for instance, $t_n - t_{n-1} = h$ and $t_{n-1} - t_{n-2} = rh$ and $r = -1/3$ (a strange thing to do—this means $t_n > t_{n-2} > t_{n-1}$), then we are apparently in trouble. However, if $t_n > t_{n-1} > t_{n-2}$ as usual, this determinant is positive. Following the procedure outlined above, we get

$$y_n = y_{n-1} + h\left(\frac{2+3r}{6(1+r)}f_n + \frac{1+3r}{6r}f_{n-1} - \frac{1}{6(1+r)r}f_{n-2}\right), \tag{13.260}$$

which reduces to

$$y_n = y_{n-1} + h\left(\frac{5}{12}f_n + \frac{2}{3}f_{n-1} - \frac{1}{12}f_{n-2}\right) \tag{13.261}$$

in the equally spaced case.

For $k = 3$, $t_n - t_{n-1} = h, t_{n-1} - t_{n-2} = rh$ and $t_{n-2} - t_{n-3} = sh$, we find by this method (solving a 2×2 symbolic system)

$$y_n = y_{n-1} + h(\beta_0 f_n + \beta_1 f_{n-1} + \beta_2 f_{n-2} + \beta_3 f_{n-3}) \tag{13.262}$$

with

$$\beta_0 = \frac{1}{12}\frac{3+6rs+8r+4s+6r^2}{(1+r)(1+r+s)} \tag{13.263}$$

$$\beta_1 = \frac{1}{12}\frac{1+6rs+4r+2s+6r^2}{r(r+s)} \tag{13.264}$$

$$\beta_2 = -\frac{1}{12}\frac{1+2r+2s}{(1+r)rs} \tag{13.265}$$

$$\beta_3 = \frac{1}{12}\frac{1+2r}{(1+r+s)(r+s)s}. \tag{13.266}$$

It reduces to $3/8, 19/24, -5/24, 1/24$ when $r = s = 1$. The process *fails* if

$$1 + 6rs + 4r + 2s + 6r^2 = 0 \tag{13.267}$$

(which it never is if $r > 0, s > 0$, but still). One can recover a solution afterward by taking limits; but did we divide by zero, or what? What actually happens in this strange case? Take $k = 2$ and $r = -1/3$; let us do a direct fit. We wish to have

$$y_n = y_{n-1} + h(\beta_0 f_n + \beta_1 f_{n-1} + \beta_2 f_{n-2}), \tag{13.268}$$

where now $t_{n-1} = t_{n-2} - \frac{h}{3}$ and $t_n = t_{n-1} + h$. This is to be accurate for polynomial y of degree at most 3. Letting $y = y_0 + y_1 + y_2 t^2 + y_3 t^3$, we find

$$y_n = y_{n-1} + h\left(\frac{1}{4}f_n + 0f_{n-1} + \frac{3}{4}f_{n-2}\right) , \tag{13.269}$$

which seems quite strange until you remember that t_{n-2} is closer to t_n than is t_{n-1}, and so it makes sense that it should be weighted more accurately. But we also see that the restriction $r \neq -1/3$ of the previous method was artificial—it's not a real restriction. This is seen in Butcher et al. (2011) also—the constraint of the form of the partial fraction, which is used as a method of solution, plays a role. Here the "problem" is that the coefficient of f_{n-1} needs to be zero, and this cannot happen in the partial fraction expansion: The coefficient of $1/(z-t_{n-1})^2$ cannot be zero, for any choice of $B(t_{n-1}) \neq 0$. Thus, the method fails in this case, even when there is a solution.

We leave this section having established a method for deriving multistep methods. Using the methods of Chap. 8, it is easy to construct an interpolant to the discrete solution thus generated. Once that is done, it is straightforward to compute and examine the residual.

13.7 The Method of Modified Equations

13.7.1 Correlations and Structure in the Residual

Now that we have defined various numerical methods, we can return to the point made in the previous chapter about the residual being correlated with the solution. We have written the reverse-engineered equation as

$$\dot{x}(t) = f(t,x) + \Delta(t) \tag{13.270}$$

and indeed any particular numerical method generates a reverse-engineered equation of this kind. But the numerical value of $\Delta(t)$ actually depends on the computed solution $x(t)$ and its derivative, and it might be important or interesting to understand the correlation. The so-called method of modified equations lets us construct at least the first terms in an asymptotic series for $\Delta(t)$ in terms of $x(t)$; asymptotic, that is, as $h \to 0$, where h is a representative step size. The analysis is possible for variable-step-size methods, but we confine ourselves here to the case of fixed-time-step methods. The analysis relies on two things: one, some algebraic manipulation that we will detail below, and two, some physical understanding of the differential equation form itself, and what certain modifications to it might mean physically. We will pursue this point with an example, shortly.

Constructing a modified equation for such a method is not hard, technically, but it is a bit slippery conceptually in that at some points in the process we think of the numerical solution as being discrete, and then we replace that with a continuous version. We thereby get a *functional* equation, which we solve in series by differentiation; during that process, we think of h as being allowed to vary. At the end, we get a modified equation with h terms in it, which we then think of as fixed. Griffiths and Sanz-Serna (1986) take some care to show how to make this process rigorous, by later verification that an answer has been found. In this section we provide an alternative method of verification, by using the residual. We will show that the numerical solution can be interpolated in such a way that the residual in the *modified* equation is asymptotically smaller than the residual in the original equation. That is, this process will allow us to explain part of the residual as arising from correlations with the solution. It is a surprisingly powerful idea, but is clearest by example.

We begin by simply writing down a modified equation for the fixed-time-step forward Euler method (indeed this is the same approach Griffiths and Sanz-Serna (1986) take at first, and it is pedagogically effective). If we are trying to solve $\dot{x}(t) = \mathbf{f}(x(t))$ by the fixed-time-step forward Euler method, producing iterates y_n by means of the rule $y_{n+1} = y_n + h\mathbf{f}(y_n)$, then we will now show that the iterates y_n more nearly solve the modified problem

$$\dot{z}(t) = \left(\mathbf{I} - \frac{h}{2}\mathbf{J}_f(z(t))\right)\mathbf{f}(z(t))\,. \tag{13.271}$$

Notice that the h that is used in the method appears in the equation. Suppose that we are solving the simple harmonic oscillator; that is, suppose that

$$\mathbf{f}(z) = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} z\,. \tag{13.272}$$

A short computation shows that the *modified* problem above is

$$\dot{z}(t) = \begin{bmatrix} h/2 & 1 \\ -1 & h/2 \end{bmatrix} z\,. \tag{13.273}$$

Call the matrix occurring here $\mathbf{A}(h)$, which is $\mathbf{A}(h) = \mathbf{A}(0) - \frac{h}{2}\mathbf{A}^2(0)$. We now construct an interpolant tailored to *this* problem. (We know that if we interpolate the numerical solution however we like, the best possible residual we can obtain from the original problem will be $O(h)$ in size—that's because forward Euler is a first-order method.) We use piecewise Hermite interpolation, matching the numerical solution values y_n and y_{n+1}, but we choose the derivatives at either endpoint to match the modified problem. Our conditions are, then,

$$\rho_{0,0} = y_n = \begin{bmatrix} s \\ c \end{bmatrix} \tag{13.274}$$

$$\rho_{1,0} = y_{n+1} = y_n + h\mathbf{A}(\mathbf{0})y_n = \begin{bmatrix} s + hc \\ c - hs \end{bmatrix} \tag{13.275}$$

$$\rho_{0,1} = \mathbf{A}(h)y_n \tag{13.276}$$

$$\rho_{1,1} = \mathbf{A}(h)y_{n+1}\,, \tag{13.277}$$

and we use our standard piecewise Hermite interpolant. We have chosen a symbolic representation for a typical vector y_n above: The first component s, the second c (for sine and cosine). Remember that, asymptotically, the residual will have maximum size at $t_n + h/2$; because it is zero at either end, we can show that the residual in (13.271) is $O(h^2)$ everywhere by sampling it at that one point (if we do the arithmetic in a computer algebra system, it's easy enough to do it for all t in the interval, but for hand computation, it's tedious enough to do it just at that one point—luckily, as argued, this suffices to show what we want). A short computation shows that at the midpoint, the residual is

$$\Delta(t_n + \frac{h}{2}) = \frac{h^2}{4}\begin{bmatrix} c \\ -s \end{bmatrix} + O(h^3)\,. \tag{13.278}$$

If we phrase this another way, what we have shown is that the forward Euler solution to $\dot{x} = \mathbf{A}x$ can be interpolated in such a way that the residual in

$$\dot{z} = \mathbf{A}z + \frac{h}{2}\mathbf{I}z + h^2 v(t) \tag{13.279}$$

has $\|v(t)\| = O(1)$. That is, the forward Euler solution to $\dot{x} = \mathbf{A}x$ is actually a *better* solution to a problem with altered dynamical behavior, in fact one with terms like $\exp(ht/2)\cos(t)$ and $\exp(ht/2)\sin(t)$ in its solution—that is, solutions that grow exponentially on the timescale $O(1/h)$. This is indeed what we see: Plotting a phase diagram shows that the forward Euler solution to this problem spirals outward. This allows us to interpret the numerical error of forward Euler on the simple harmonic oscillator, at least partially, as introducing some *negative damping*.

At this point, we stress that we are not trying to *improve* the numerical method (here, forward Euler), but rather to *understand what it does*. This requires us to be able to look at a modified (here, linear) system $\dot{y}(t) = \mathbf{A}(h)y(t)$ and realize that the real parts of the eigenvalues (here $h/2$) determine the asymptotic growth or decay of the solution. People who are familiar with simple harmonic oscillators (people such as mechanical engineers, for whom the vibration of structures is a serious matter) are well skilled in interpreting the pure imaginary parts of the eigenvalues as frequencies, and the real parts as damping (positive, meaning the vibrations decay away over time, or negative, meaning that they grow—as happens, for example, when energy is fed to the structure by flow of a surrounding fluid, such as wind on a bridge deck). For such people, the modified equation tells an immediate story about what forward Euler is doing to the original problem. Similarly, for an applied mathematical modeler, who is investigating a particular model, terms in the equation can usually be interpreted in terms of what is really happening. This technique, finding a modified equation, is of much potential value to the modeler who is investigating the fidelity of the numerical method used.

It is a simple matter to do a similar analysis for backward Euler on this problem, and one finds that the residual to the problem with $\mathbf{A}(-h)$ is now $O(h^2)$; that is, backward Euler introduces some positive damping, with terms like $\exp(-ht/2\cos(t)$ and $\exp(-ht/2)\sin(t)$. See Exercise 13.17.

There is an interesting and subtle point hiding here, to do with the use of the (apparently trivial) change of a second-order scalar problem, $y'' + y = 0$, to a first-order system by the usual embedding of variables $y_1 = y$, $y_2 = y'$. The residual above is a *vector* residual. If we worked directly with the second-order scalar problem, we would want a *scalar* residual. That is, we absolutely would *not* want any defect in the equation $y_1' = y_2$, in the (artificial) embedding of the scalar second-order problem into the first-order system. In the above analysis, that artificial equation is indeed perturbed, and we found that the modified equation was $y_1' = y_2 + (h/2)y_1 + O(h^2)$. What does this mean? For this *linear* problem, there is no great consequence: We can make a near-identity change of variables, $u = \mathbf{B}y$, with (for example)

$$\mathbf{B} = \begin{bmatrix} 1 - h^2/8 & h/2 \\ -h^3/16 & 1 + h^2/8 \end{bmatrix} \tag{13.280}$$

and the modified equation becomes, in the new variables,

$$u' = \begin{bmatrix} 0 & 1 \\ -1 - h^2/4 & h \end{bmatrix} u + O(h^2)\,. \tag{13.281}$$

Because the first row is $[0, 1]$, this equation can be interpreted as an embedding of the scalar equation $w'' - hw' + (1 + h^2/4)w = O(h^2)$ into a two-dimensional first-order system. This scalar equation has negative damping (again $h/2$), leading to terms like $\exp(ht/2)$ times sines and cosines. For nonlinear problems, this is more of an issue, or can be, and may be analyzed by a procedure akin to the construction of so-called normal forms. We leave you with the thought that the size of the residual really does depend on the problem formulation. If one is interested in the scalar residual, one should be careful to construct a method that is easily interpreted as a scalar higher-order method.

13.7.2 Finding a Modified Equation

The procedure for *finding* a modified equation is, when carried out, fairly simple. One need not be rigorous, as was pointed out in Griffiths and Sanz-Serna (1986), provided one is careful to verify afterward that the numerical solution does indeed follow the solution of the modified equation more closely. Since rigor is not needed in the derivation of modified equations, the task is simplified. A little thought and experimentation show that modified equations are by no means unique.

For forward Euler, begin with the definition

$$x_{n+1} = x_n + h\mathbf{f}(x_n)\,, \tag{13.282}$$

and now interpret this as a *functional equation* by replacing x_n by $v(t)$, and x_{n+1} by $v(t+h)$. We get

$$v(t+h) = v(t) + h\mathbf{f}(v(t))\,, \tag{13.283}$$

which can now be expanded in a Taylor series:

$$v(t) + h\dot{v}(t) + \frac{h^2}{2}\ddot{v}(t) + O(h^3) = v(t) + h\mathbf{f}(v(t)). \tag{13.284}$$

On removing the $v(t)$ from each side and dividing both sides by h, we obtain

$$\dot{v}(t) = \mathbf{f}(v(t)) - \frac{h}{2}\ddot{v}(t) + O(h^2)\,. \tag{13.285}$$

This is a singular perturbation of the original equation, and not very satisfactory, so we try to remove the second derivative term, by differentiating this equation once to get

$$\ddot{v}(t) = \mathbf{J}_f(v(t))\mathbf{f}(v(t)) + O(h), \tag{13.286}$$

which, when substituted into (13.285), gives

$$\dot{v}(t) = \mathbf{f}(v(t)) - \frac{h}{2}\mathbf{J}_f(v(t))\mathbf{f}(v(t)) + O(h^2), \tag{13.287}$$

as previously claimed. We see that indeed the process is quite simple; it's only when one tries to think about the answer, or tries to apply the process to a nonstandard method, that difficulties seem to arise. Once a candidate modified equation is found, it is quite important to verify, either by the method of the residual, or by the local error bound method of Griffiths and Sanz-Serna (1986), that the modified equation really does explain some features of the numerical solution.

Example 13.10. Suppose we wish to solve $\ddot{x} + x = 0$ using an "exotic" method, the second-order Taylor series method, *without* first transforming this equation to a first-order system. By this we mean that our method is a second-order Taylor series method for $\dot{x}(t)$, with $x(t) = x_n + \int_{\tau=t_n}^{t} \dot{x}(\tau)\,d\tau$ following naturally exactly. That is, we denote our computed values approximating $\dot{x}(t_n)$ as $\dot{x}_n$, and use the method

$$\dot{x}_{n+1} = \dot{x}_n - hx_n - \frac{h^2}{2}\dot{x}_n\,, \tag{13.288}$$

where we have used the original differential equation to replace $\ddot{x}(t_n)$ with $-x(t_n)$, that is, x_n, and taken one derivative of the equation to get $x^{(3)}(t) = -\dot{x}(t)$ so that the next term contains $-\dot{x}_n$. Then exact integration of the second-order Taylor interpolant gives

$$x_{n+1} = x_n + h\dot{x}_n - \frac{h^2}{2}x_n - \frac{h^3}{3!}\dot{x}_n\,. \tag{13.289}$$

A little thought should convince you that this formula really just "follows along" the second-order formula above that generates successive approximations to $\dot{x}(t_n)$.

Finding a modified equation for this method is quite simple. We follow the same general method sketched above. The first step is to replace the discrete equation with a functional equation:

$$\dot{v}(t+h) = \dot{v}(t) - hv(t) - \frac{h^2}{2}\dot{v}(t)\,. \tag{13.290}$$

We now wish to approximate that functional equation[12] with a differential equation. To do this, we expand $\dot{v}(t+h)$ in a Taylor series. Working to $O(h^5)$, we get

$$\dot{v}(t) + h\ddot{v}(t) + \frac{h^2}{2}v^{(3)}(t) + \frac{h^3}{3!}v^{(4)}(t) + \frac{h^4}{4!}v^{(5)}(t) + O(h^5) = \dot{v}(t) - hv(t) - \frac{h^2}{2}\dot{v}(t)\,.$$

The two leading $\dot{v}(t)$ on either side cancel, and then we may divide both sides by h. Isolating $\ddot{v}(t)$ in what remains, we find

$$\ddot{v}(t) = -v - \frac{h}{2}\left(\dot{v} + v^{(3)}\right) - \frac{h^2}{3!}v^{(4)} - \frac{h^3}{4!}v^{(5)} + O(h^4)\,. \tag{13.291}$$

Differentiating (13.291) with respect to time and keeping only terms to $O(h^3)$, we get

$$v^{(3)} = -\dot{v} - \frac{h}{2}\left(\ddot{v} + v^{(4)}\right) - \frac{h^2}{3!}v^{(5)} + O(h^3)\,. \tag{13.292}$$

Differentiating again gives

$$v^{(4)} = -\ddot{v} - \frac{h}{2}\left(v^{(3)} + v^{(5)}\right) + O(h^2)\,. \tag{13.293}$$

Next, we have

$$v^{(5)} = -v^{(3)} + O(h)\,. \tag{13.294}$$

Using (13.294) in (13.293), we see that

$$v^{(4)} = -\ddot{v} + O(h^2), \tag{13.295}$$

and, using these in (13.292), we find (after using the equation to simplify itself as well)

$$v^{(3)} = -\dot{v} - \frac{h^2}{3!}\dot{v} + O(h^3)\,. \tag{13.296}$$

[12] This functional equation is actually a *neutral delay differential equation*, a kind of equation that we will study in Chap. 15. Solving such equations by expanding in a series in the delay is known to be a potentially misleading technique; however, for this specialized application, all is well and we wind up with a modified equation that explains the numerics nicely.

Finally, we may now use all these to simplify (13.291). If we have done our algebra correctly, then what we get is

$$\ddot{v} = -v + \frac{h^3}{12}\dot{v} - \frac{h^2}{3!}(-\ddot{v}) - \frac{h^3}{4!}\dot{v} + O(h^4) \tag{13.297}$$

or

$$\ddot{v} = -v + \left(\frac{h^3}{12} - \frac{h^3}{24}\right)\dot{v} - \frac{h^2}{3!}(v) + O(h^4) \tag{13.298}$$

or, at last,

$$\ddot{v} - \frac{h^3}{24}\dot{v} + \left(1 + \frac{h^2}{3!}\right)v = O(h^4)\,. \tag{13.299}$$

We compare this to the well-known schema for damped harmonic oscillation,

$$\ddot{x} + 2\zeta\omega\dot{x} + \omega^2 x = 0, \tag{13.300}$$

which has analytic solution

$$x(t) = \left(\alpha\cos\omega\sqrt{1-\zeta^2}\,t + \left(\frac{\beta+\alpha\zeta\omega}{\omega\sqrt{1-\zeta^2}}\right)\sin\omega\sqrt{1-\zeta^2}\,t\right)e^{-\zeta\omega t}, \tag{13.301}$$

satisfying $x(0) = \alpha$, $\dot{x}(0) = \beta$. We thus identify the frequency as

$$\omega = 1 + \frac{h^2}{12} + O(h^4) \tag{13.302}$$

and the damping as

$$\zeta = -\frac{h^3}{48} + O(h^5) \tag{13.303}$$

[a higher-order computation is needed, in fact, to show that ζ is accurate to $O(h^5)$]. Therefore, we would *expect* that using this numerical method would give us an $O(h^2)$ error in the *frequency*, and introduce negative damping of $O(h^3)$, with negligible *detuning* $\sqrt{1-\zeta^2} = 1 + O(h^6)$. Other numerical effects would also be smaller than $O(h^3)$.

Now, carry out the numerical solution with the following code:

```
% Method of modified equations exotic example
% x'' + x = 0
%
% 2nd order Taylor series method (for x')
% exact integration of x' to get x
%
a = 0;
```

```
b = 5*pi;
n = 30;
h = (b-a)/n;
x = zeros(1,n+1);
x(1) = 1;
dx = zeros(1,n+1);
dx(1) = 0;

% Integration steps
for i=2:n+1,
    dx(i) = dx(i-1) - h*x(i-1) - h^2*dx(i-1)/2;
    x(i) = x(i-1) + h*dx(i-1) - h^2*x(i-1)/2 - h^3*dx(i-1)/6;
end
% Integration complete.

tc = linspace(a,b,n+1);
tf = linspace(a,b,5*n+1);
xref = cos( tf );
xrefc = cos( tc );

% Analytical solution of modified equation
w = 1 + h^2/12;
zet = -h^3/48;
alf = x(1);
bet = (dx(1) + zet*w*alf)/w/sqrt(1-zet^2) ;

% fine and coarse
modxref = exp(-zet*w*tf).*(alf*cos(w*sqrt(1-zet^2)*tf) ...
                         + bet*sin(w*sqrt(1-zet^2)*tf) );
modxrefc = exp(-zet*w*tc).*(alf*cos(w*sqrt(1-zet^2)*tc) ...
                          + bet*sin(w*sqrt(1-zet^2)*tc) );
% Second plot shows the errors clearly; first plot is nicer.
figure(1),plot( tc, x, 'k+', tf, xref, 'k--', tf, modxref, 'k-.' );
set(gca,'fontsize',16);
axis([0,16,-1.5,1.5]);
set(gca,'XTick',0:2:16);
gure(2),semilogy( tc, abs(x-xrefc), 'k--', tc, abs(x-modxrefc), 'k-.')
set(gca,'fontsize',16);
axis([0,16,1E-4,1]);
set(gca,'XTick',0:2:16);
```

When we actually carry out the numerical solution with this code, we obtain Figs. 13.15 and 13.16. We find that the modified equation does indeed explain the numerics quite well: We see that the phase is shifted by $O(h^2)$, as predicted, and that there is negative damping (exponential growth) of about the correct order as well.

Carrying out the analysis to two higher orders gives instead

$$\ddot{v} - (h^3/24 + 7h^5/480)\dot{v} + (1 + h^2/6 + 19h^4/720)v = O(h^6)\,, \tag{13.304}$$

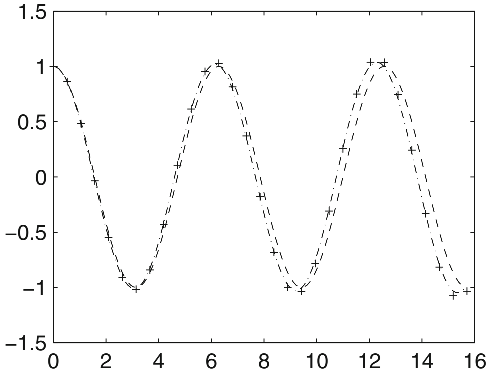


Fig. 13.15 Crosses represent second-order Taylor series solution with $h = 5\pi/30$. The dashed line is $\cos(t)$, the reference solution. The dash-dot line is the exact solution to the modified equation, and one sees that this solution explains the numerics quite well: The phase shift and the negative damping both appear the same on this graph

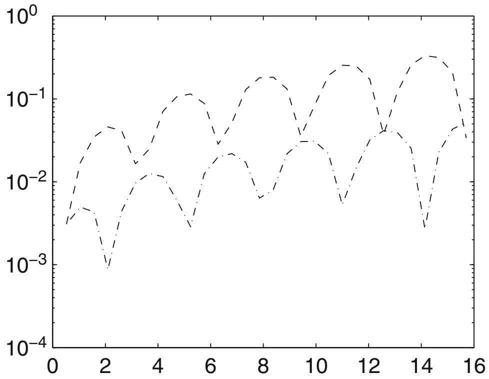


Fig. 13.16 The dashed line is the difference between the Taylor series solution and $\cos(t)$, the reference solution. The dash-dot line is the difference between the exact solution and the modified equation. This figure shows more clearly than the previous figure that the solution of the modified equation is closer to the results of the numerics

and the solution of this equation is noticeably different for the (not asymptotically small) values of h that we have used, and this raises another point: The resemblance of the solution of the modified equation to the numerical solution computed previously is to be expected only for a compact interval of time, and agreement will be better as $h \to 0$. Nonetheless, we see here that the modified equation has explained the numerics, in at least a qualitative sense. ◁

We end this section by pointing out that the most important benefit from this approach, namely, the explanation of the largest contribution to the residual from correlations with the solution, can be received by computing the first term in the asymptotic series. Higher-order terms return diminishing value for rather more effort for each term. In certain cases (see, e.g., Corless 1994a), one can find an infinite-order expansion (which often happens to be divergent, but no matter) and learn a surprising amount of information; but this situation is rather rare. For the simple case of $\dot{y}(t) = \lambda y(t)$, one can even find an exactly correct differential equation, not just an infinite-order series expansion, whose solution interpolates the numerical solution of the original equation (see Exercise 13.21).

13.8 Geometric Integration

We have largely considered methods for solving quite general initial-value problems for ODEs. For special classes of problems with special properties, we may be able to do better. Certain applications such as molecular dynamics, computational astronomy, mechanical and electrical systems, and others use models that have useful invariants, such as conservation of energy or momentum. A large and useful class of such models is exemplified by Hamiltonian mechanics problems, where the dynamics are governed by a Hamiltonian H. In canonical coordinates, the equations of motion are

$$\begin{aligned}\frac{d\mathbf{p}}{dt} &= -H_q(\mathbf{p},\mathbf{q})\\ \frac{d\mathbf{q}}{dt} &= H_p(\mathbf{p},\mathbf{q})\,,\end{aligned} \tag{13.305}$$

where the vector $\mathbf{p}$ typically represents momentum variables and the vector $\mathbf{q}$ represents position. Often the Hamiltonians are of special form, $H = T(\mathbf{p}) + U(\mathbf{q})$, where T represents kinetic energy and U the potential. This is called a *separable* Hamiltonian.

In keeping with the backward error point of view, we would say that a numerical method for solving (13.305) was a good one if it gave you (nearly) the reference solution to a nearby *Hamiltonian* problem. The so-called symplectic methods do just this.

One of the simplest such methods goes by the name of the "leapfrog" method,which is also called the Störmer method (if you do computational geophysics or climate modeling) and also called the Verlet method (if you do molecular dynamics). Following Preto and Tremaine (1999) and also Hairer et al. (2006), we mean the "drift–kick–drift" version of leapfrog:

$$\begin{aligned}\mathbf{q}^{(1)} &= \mathbf{q}_n + \frac{h}{2}H_p(\mathbf{p}_n,\mathbf{q}_n) \quad &,\quad \mathbf{p}^{(1)} &= \mathbf{p}_n\\ \mathbf{q}^{(2)} &= \mathbf{q}^{(1)} \quad &,\quad \mathbf{p}^{(2)} &= \mathbf{p}^{(1)} - hH_q(\mathbf{p}^{(1)},\mathbf{q}^{(1)})\\ \mathbf{q}_{n+1} &= \mathbf{q}^{(2)} + \frac{h}{2}H_p(\mathbf{p}^{(2)},\mathbf{q}^{(2)}) \quad &,\quad \mathbf{p}_{n+1} &= \mathbf{p}^{(2)}\,.\end{aligned} \tag{13.306}$$

That is, we first drift with constant momentum for half a step, then we kick the system with the potential, and then we drift another half a step.

For second-order problems where the Hamiltonian is separable and furthermore $T(p) = {}^{(p_1^2+p_2^2+\cdots p_N^2)}/_2$, the equations have $\dot{q}_i = p_i$ and so are equivalent to a system of second-order ODEs for the q_i. Then this method is equivalent to

$$\begin{aligned} \mathbf{p}_{n+1} &= \mathbf{p}_n - hU_q(\mathbf{Q}_n) \\ \mathbf{Q}_{n+1} &= \mathbf{Q}_n + h\mathbf{p}_{n+1}\,, \end{aligned} \tag{13.307}$$

as you can see by putting $\mathbf{Q}_n = \mathbf{q}_n + {}^{h\mathbf{p}_n}/_2$. In either formulation, the method only requires one evaluation of the force term U_q per step, just as Euler's method does. For this class of problems, however, the method is second-order and *symplectic* in that it gives samples at equal step sizes $t_k = t_0 + kh$ of vector functions $\tilde{\mathbf{p}}(t)$ and $\tilde{\mathbf{q}}(t)$ that (nearly) satisfy the equations of motion from a *perturbed* Hamiltonian system

$$\tilde{H} = H(\mathbf{p},\mathbf{q}) + h^2H_2(\mathbf{p},\mathbf{q}) + h^4H_4(\mathbf{p},\mathbf{q}) + \cdots\,. \tag{13.308}$$

That is, we can (spectrally nearly) interpolate the fixed-step-size skeleton by using the solution of a nearby Hamiltonian problem. The smaller h is, the more work we have to do, but the nearer the perturbed Hamiltonian is to the one we wanted. See Calvo et al. (1994) as well as the previously mentioned references for how to compute these perturbed Hamiltonians in general, but note that the method of modified equations as discussed in Sect. 13.7 will work for particular examples and methods.

For a variety of reasons, this is a more satisfactory backward error analysis than our usual interpretation of the defect by a polynomial interpolant. First, the perturbation of the problem is autonomous (if the original problem is autonomous—symplectic methods work for some time-dependent Hamiltonian problems as well). This means that the perturbed differential equation has the same dimension as the original. Second, many physical perturbations of, for example, computational astronomy problems, are themselves Hamiltonian—think of neglecting the influence of other planets, for example. Some, of course, are not, such as tidal friction or minor collisions, but these may be smaller than the Hamiltonian perturbations. Third, the preservation of symplecity[13] may better preserve certain statistical measures. Given the chaotic nature of many Hamiltonian systems, of course, there is no hope of ensuring small global forward error in trajectories, and so accurate statistics are all that can be computed.

Example 13.11. When we solved the Hénon–Heiles problem in Example 12.14, we mentioned only in passing that the problem was Hamiltonian. The Hamiltonian is

$$H = \frac{1}{2}\left(p_1^2 + p_2^2 + q_1^2 + q_2^2\right) + q_1^2q_2 - \frac{1}{3}q_2^3\,. \tag{13.309}$$

[13] The standard term is "symplecticity," which seems awkward to our ears. The alternative in the text is our own coining, which we warn you has not caught on. There is also "symplecticness" as an equally awkward standard word. However, the comparative simplicity of "symplecity" has not made any inroads to the usage of either term. One can hope, though.

We mentioned the computing times for the solutions, which were moderately long because we used very tight tolerances. Indeed, the easy case took nearly two million time steps, and the hard case at the higher energy took about two and a half million time steps (the average time step was $h = 0.0406$, and we went out to $t = 10^5$). By using so much work, we ensured that the energy was conserved over the whole range of integration to better than one part in 10^{11}, in both cases.

Can a low-order symplectic method such as the leapfrog method produce results that are "just as good" with less effort? Here is a fragment of the code `HenonLeapfrog.m`:

```
    for i=2:N+1,
        q(1) = y(3,i-1) + h*y(1,i-1)/2;
        q(2) = y(4,i-1) + h*y(2,i-1)/2;
        y(1,i) = y(1,i-1) - h*(q(1) + 2*q(1)*q(2));
        y(2,i) = y(2,i-1) - h*(q(2) + q(1)^2-q(2)^2);
        y(3,i) = q(1) + h*y(1,i)/2;
        y(4,i) = q(2) + h*y(2,i)/2;
        if y(3,i)*y(3,i-1) <= 0 && y(1,i) > 0,
        % Look for events  q1(t) = 0, p1(t) >= 0
```

The results are ambiguous, but encouraging. If we take $h = {}^1\!/_{25}$, so we use 2.5×10^6 time steps, just as `ode113` did for the hard problem, we find that because this method is only second-order, the accuracy is not as good: We get an energy error better than one part in 10^5, whereas the higher-order `ode113` got one part in 10^{11}. But the solution is faster with this simpler method, taking about 22 seconds instead of 8 minutes. To be fairer to `ode113`, we loosen the tolerances to 10^{-6} and run it again; we find that it's *still* quite a bit slower, taking 90 seconds, which is more than four times as long.

Part of the problem is that we have not yet mentioned how we solved the event location problem for the Poincaré map when we use the leapfrog method. Because it's such a simple event, we just look at sign changes in q_1, and if there is one between t_{i-1} and t_i, then we fit a cubic Hermite polynomial to $q_1(t_{i-1})$, $p_1(t_{i-1})$, $q_1(t_i)$, and $q_1(t_i)$,$p_1(t_i)$ and use the companion matrix pencil to find the root accurately in that interval. Of course, we have to write and debug this code ourselves (an alternative is simply to use the barycentric formula to *evaluate* this cubic and use `fzero` to find its roots). In contrast, once "events" are turned on, `ode113` must check every interval for events, and this takes more time since that routine must be prepared for more complicated events than we are looking at here. Still, it's easier to use the built-in event-location features of the built-in ode solvers than to write our own. If we are going to write our own, we would hope for clear advantages for our special method.

Here the results are ambiguous at first. At looser tolerances, `ode113` still does quite a decent job of computing a good first-return map. The result is virtually indistinguishable from the tighter tolerance version in Example 12.14. The energy error, however, is now about $1.5 \cdot 10^{-4}$, which is worse than the leapfrog result, where the energy is nearly constant to about $\pm 2 \times 10^{-6}$. The leapfrog result for the Poincaré map, though, seems not to be as good even though the energy is better. See Fig. 13.17, where quite a bit of phase drift can be seen.

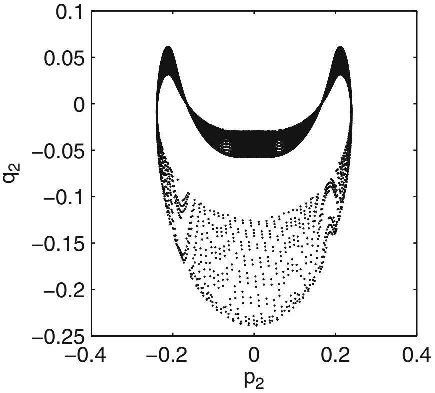


Fig. 13.17 First-return map (Poincaré map) for the Hénon–Heiles model solved by the leapfrog method with $h = 1/25$ on $0 \le t \le 10^5$. The energy error is less than one part in 10^5. The sections were computed by first checking for sign changes in q_1 and then fitting a cubic Hermite polynomial in barycentric form to the known values of q_1 at either end and using the known slopes p_1. The roots of the cubic were located using the eigenvalue method of Sect. 8.2.1 and confirmed by an alternative run using `fzero`. The leapfrog solution shows phase drift compared to the four times as costly non-symplectic method, but keeps much better energy error

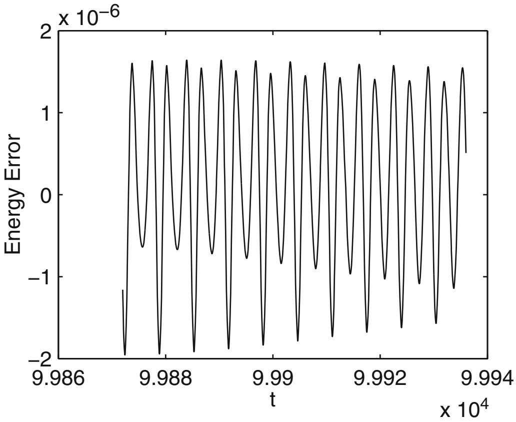


Fig. 13.18 The energy error in a leapfrog integration. So long as rounding errors play no role, the leapfrog method nearly conserves energy; more, its departures from the conserved energy are bounded, because the numerical solution is nearly the reference solution to a nearby Hamiltonian problem

The *real* benefit of this symplectic scheme is seen on much longer time intervals of integration. The energy error of the scheme is not just lower than the loose-tolerance run of `ode113`, the energy is *nearly constant* and does not grow on long time scales; there is no “secular growth,” in other words. We graph a typical interval of energy error for the leapfrog method on this problem in Fig. 13.18. ◁

The energy oscillates around the original energy level, with tiny $O(h^2)$ amplitude, but *does not grow* as long as rounding errors play no role. In contrast, the energy error for the non-symplectic method grows at a secular rate: By experiment, $E - E_{\text{reference}} = O(t\varepsilon)$, where ε is the tolerance for integration. Over very long intervals, this advantage of symplectic integration methods acquires considerable value.

Explicitly, the modified equation that the leapfrog method more nearly solves is, to $O(h^4)$, as we see from these equations:

$$\begin{aligned}
\frac{dq_1}{dt} &= p_1 + \left(-\frac{1}{12}p_1 - \frac{1}{6}q_1p_2 - \frac{1}{6}p_1q_2\right)h^2 + O(h^4) \\
\frac{dq_2}{dt} &= p_2 + \left(-\frac{1}{6}q_1p_1 - \frac{1}{12}p_2 + \frac{1}{6}q_2p_2\right)h^2 + O(h^4) \\
\frac{dp_1}{dt} &= -q_1\,(1+2q_2) + \left(\frac{1}{6}p_1p_2 - \frac{1}{6}q_1 - q_1q_2 - \frac{1}{3}q_1q_2^2 - \frac{1}{3}q_1^3\right)h^2 + O(h^4) \\
\frac{dp_1}{dt} &= -q_2 + q_2^2 - q_1^2 \\
&\qquad + \frac{1}{12}\left(p_1^2 - p_2^2 + 6q_2^2 - 6q_1^2 - 4q_1^2q_2 - 2q_2 - 4q_2^3\right)h^2 + O(h^4)\,. \qquad (13.310)
\end{aligned}$$

These are Hamilton's equations of motion for a perturbed Hamiltonian $H = H_0 + h^2H_2 + O(h^4)$, as can be seen by taking all the requisite cross-derivatives and seeing that they match. Specifically, the value of H_2 is

$$\begin{aligned}
&-\frac{1}{6}p_1p_2q_1 + \frac{1}{12}q_1^2 + \frac{1}{2}q_1^2q_2 + \frac{1}{6}q_1^2q_2^2 + \frac{1}{12}q_1^4 + \frac{1}{12}q_2^2 - \\
&\qquad \frac{1}{12}p_1^2q_2 + \frac{1}{12}p_2^2q_2 - \frac{1}{6}q_2^3 + \frac{1}{12}q_2^4 - \frac{1}{24}p_1^2 - \frac{1}{24}p_2^2\,. \qquad (13.311)
\end{aligned}$$

Notice that the perturbed Hamiltonian is *not* separable. However,, since the variables remain bounded, we see that the perturbation in the energy remains bounded, although energy is not strictly speaking conserved.

13.9 Other Methods

Probably the most important general class of practical methods not covered in this book is *extrapolation methods*. These are discussed in detail in Hairer et al. (1993) and strongly advocated in Press et al. (1986). But there are many other methods. The so-called Enright methods, which use evaluations of Jacobians of f to advance the step, are something like a mixture of second-order Taylor series methods and multistep methods, but use derivatives, as do Rosenbrock methods, such as that implemented in `ode23s`. In MAPLE the `method=stiff` option for `dsolve` (in numerical mode) also uses a Rosenbrock method. The grand unified theory of these methods is the study of *general linear methods*, for which you may consult Butcher (2008b).

The Lanczos τ-method (which we briefly looked at in Chap. 3 already) uses Chebyshev polynomials (see Lanczos 1988) and a residual-based approach. In fact, now that we have looked at methods for solving IVP, it may be worthwhile for the reader to go back to that chapter and reread the relevant sections, and to look again at Problem 3.17. The Hermite–Obreschkoff methods generalize Taylor series methods (Nedialkov and Jackson 1999). Picard iteration (explored in Problem 13.37) is a well-known theoretical technique that can sometimes be used for computation.

We gave a very brief introduction to symplectic integration, mentioning the low-order leapfrog method. There are higher-order methods, indeed methods of arbitrarily high order. Explicit symplectic methods are possible only for separable Hamiltonians, however, so implicit methods must be used if the Hamiltonian is not separable. Geometric integration methods, in general, include symplectic methods, energy-conserving methods, time-reversible methods, and others. Integration on manifolds—that is, making sure that the computed solutions of your ODE continue to satisfy the geometric constraints of your model automatically—is of great importance in practice: A good numerical method there will solve a nearby problem whose solutions *also* satisfy the constraints. Using a variable step size for efficiency with these methods is harder than it is if you don't need to stay symplectic or preserve constraints. See Hairer et al. (2006).

Which method is the best method? Unfortunately, there is no satisfactory answer. It's kind of like the game of Rock–Paper–Scissors; each of the methods described or mentioned above is "best" for at least one problem, maybe for yours. Most of the methods can be made to work satisfactorily on many problems, though, and now you know how to verify a posteriori that your chosen method has done so.

However, the goal of these two chapters was not to get you ready to write your own specialized code for solving initial-value problems for ordinary differential equations. The main goals were to teach you how to recognize a good solution given by a standard implementation when you saw it (using the theory of backward error and of conditioning), and to have a basic understanding of how the main methods worked, which is essentially by a numerical variant of analytic continuation. There were several secondary goals, one of which was to show how to develop formulæ and another of which was to show some aspects of the standard theory of stability for stiff systems; this, although based entirely on fixed-time-step methods, gives a rough guide as to what to expect when solving a stiff problem.

13.10 Notes and References

Theorem 13.3 is based on one in Stetter (1973). The relationship of local error to residual has therefore been known a long time; controlling LEPUS gives an indirect unfortunately problem-dependent control on the residual. Even so, in our opinion, this is the only reason that numerical methods based on local error control give high-quality solutions.

Taylor series methods and analytic continuation are very old. Some early modern work includes Gibbons (1960), which implements the series algorithms discussed in Chap. 8 for a "Ferranti Mercury computer with a floating-point unit and a high-speed store of $2,048$ twenty-bit words." The program would solve up to six simultaneous differential equations of order up to 15, and used up to 30-term Taylor series. Neither interpolants nor the residual were mentioned, however. Implicit Taylor series are studied in Barton et al. (1971). We have high accuracy if the solution is smooth and N is large, at a reasonable cost. Indeed, it has been shown that the Taylor series method has cost polynomial in the number of bits of accuracy requested (Ilie et al. 2008; Corless et al. 2006).

The modern theory of order conditions for Runge–Kutta methods was initiated in Butcher (1963). Streamlined explications can be found in Butcher (2008b), Hairer et al. (1993), and Hairer et al. (2006). Chapter 6 of that last is *extremely* streamlined but still yet complete. In this book, we have barely started an introduction to the use of trees for order conditions. The theory of trees and the corresponding B-series is now quite advanced: See Chartier et al. (2010), which presents and unifies many numerical results; in particular, a method is given for computing B-series of the residual of a Runge–Kutta method.

The multivariate polynomial equations that arise in the order conditions are themselves quite interesting. Their solutions can exhibit a great deal of symmetry, and there are often multiparameter families of solutions, as well as solutions that do not fall into those families. Sometimes there are no solutions. Currently, the best solvers of these systems are certain humans: Automatic solution by Gröbner basis computation, for example, has until now not been as successful (in part because the combinatorial symmetries mean that the degrees of the Gröbner bases grow faster than exponentially). Still, there are many families of solutions now known, and indeed there are formulæ known of arbitrarily high order. One very interesting development along these lines is presented in Khashin (2009) and in Khashin (2012). At the ANODE 2013 meeting, Jim Verner reported using these methods to give a new exact family of solutions, which he called the Runge–Kutta–Khashin methods, that needed fewer stages for high order than any previously known methods; this represented the first reduction in required stages in 38 years of research.

For a description of the Rosenbrock method used in `ode23s`, see Shampine and Reichelt (1997). For a discussion of Rosenbrock methods in general, see Hairer and Wanner (2002 chapter IV.7). The method used here to introduce them was modeled heavily on that work.

See Griffiths and Sanz-Serna (1986) for an extensive discussion and bibliography of the method of modified equations. This method was originally used for PDE in Warming and Hyett (1974).

For a technical description of the usefulness of variable stepsize for dynamical systems, see Higham and Stuart (1998). For an interesting view of stiffness and systems, see Higham and Trefethen (1993). Adaptive step-size control using control-theoretic ideas is explored in Söderlind (2003).

Problems

Theory and Practice

13.1. Prove that if the right-hand side of the differential equation $\dot{x} = f(t,x)$ satisfies the one-sided Lipschitz condition (13.196), then for any two solutions $x(t)$ and $y(t)$, we have

$$\|x(t) - y(t)\| \leq \|x(t_0) - y(t_0)\| e^{\nu(t-t_0)} . \tag{13.312}$$

13.2. Show that $p(\theta) = (1-\theta^2)\rho_{0,0} + \theta(1-\theta)\rho_{0,1} + \theta^2\rho_{1,0}$ satisfies $p(0) = \rho_{0,0}$, $p'(0) = \rho_{0,1}$, and $p(1) = \rho_{1,0}$. This is the piecewise quadratic interpolant needed for the improved Euler method (explicit trapezium, or trapezoidal rule). Show that this interpolant has a residual of $O(h^2)$, and thereby give another proof that the improved Euler method is a second-order method.

13.3. Consider applying the backward (also called implicit) Euler method to the problem in Example 13.5, $\dot{x} = x^2$. That is, let

$$x_{n+1} = x_n + hx_{n+1}^2 . \tag{13.313}$$

Analytically isolate the solution x_{n+1} of this iteration; show that the solution is not unique, and that time steps h cannot be arbitrarily large.

13.4. Repeat Problem 13.3, but this time for the ODE $\dot{x} = x^2 - t$ that we used in the previous chapter to demonstrate stiffness. Discuss the probability of encountering difficulties with the solution of nonlinear systems for larger problems, and the potential restrictions on the time step.

13.5. Implement and use the fixed-step-size implicit midpoint rule to solve $\dot{x} = x^2 - t, x(0) = -1/2$ on $0 \leq t \leq 10$. Vary the initial conditions, and verify that the numerical solution by this B-stable method is contractive.

13.6. Draw the region bounded by the curve defined by Eq. (13.235).

13.7. The "classical" fourth-order Runge–Kutta method used with fixed time step h in Eq. (13.60) is sometimes called "RK dumb." Sometimes it is indeed a dumb idea to use it, at least without assessing its solutions independently. Solve the problem $\dot{y} = y^2 - t$ on $0 \leq t \leq 20.5$ with the initial condition $y(0) = -1/2$. Use a time step $h = 20.5/55$, so you take 55 steps. The solution should look perfectly reasonable, but it will be quite wrong at $t = 20.5$.

13.8. Consider using explicit Taylor series methods with fixed time step $h > 0$ on the Dahlquist test problem $y'(t) = \lambda y(t)$, $y(0) = y_0$. Show that an explicit Nth-order Taylor series method applied to this problem gives the iterative formula

$$y_{k+1} = \left(1 + z + \frac{z^2}{2} + \cdots + \frac{z^N}{N!}\right) y_k , \tag{13.314}$$

where as usual $z = h\lambda$. For $N = 1, 2, 3, \ldots, 6$, draw the regions in the z-plane where the numerical solution has monotonic decay and as such captures the correct qualitative information if $\mathrm{Re}\lambda < 0$. Plot them on the same graph. Note that for higher N, the regions are not connected.

13.9. Repeat the previous problem, but for implicit Taylor series methods (see Example 13.3 for the second-order case).

13.10. Consider now using an explicit Nth-order Taylor series method on the m-dimensional linear system $\dot{\mathbf{y}} = \mathbf{A}\mathbf{y}$, $\mathbf{y}(0) = \mathbf{y}_0$. If m is large, then the cost of evaluating the Taylor series is about the same as doing N matrix–vector multiplications; thus, one step of the Taylor series method costs about the same as N steps of Euler's method. If the problem is stiff, that is, if there are many eigenvalues of $\mathbf{A}$ that have large negative real part, show that the high-order Taylor series method is really no cheaper (or more expensive) than simple forward Euler.

13.11. Show that the stability function for a general Runge–Kutta method is

$$R(z) = 1 + zb^T \left(\mathbf{I} - z\mathbf{A}\right)^{-1} \mathbf{e}, \tag{13.315}$$

where the vector $\mathbf{b}$ and the matrix $\mathbf{A}$ are from the Butcher tableau, $z = h\lambda$ is the product of the step size and the eigenvalue in the Dahlquist test problem $y' = \lambda y$, and the vector $\mathbf{e}$ is all 1s.

13.12. The 2-stage IRK based on Gauss points, with Butcher tableau

$$\begin{array}{c|cc} 1/2 - \sqrt{3}/6 & 1/4 & 1/4 - \sqrt{3}/6 \\ 1/2 + \sqrt{3}/6 & 1/4 + \sqrt{3}/6 & 1/4 \\ \hline & 1/2 & 1/2 \end{array}\,, \tag{13.316}$$

is of fourth order and exactly A-stable. Verify both these facts.

13.13. As discussed in the text, Taylor series methods can also be *implicit*. This amounts to doing the Taylor series expansion about the new point $t = t_{k+1}$, guessing (or solving via Newton's method) the required y_{k+1} so that Taylor series about that point allows matching back to the known value y_k at $t = t_k$. Use the implicit Taylor series method with fixed step size $h = 1/20$ and $N = 8$ to solve Airy's differential equation on $0 \le t \le 2$; that is, $y'' = ty$, with $y(0) = 3^{-2/3}/(\Gamma(2/3))$ and $y'(0) = -3^{1/6}\Gamma(2/3)/(2\pi)$. Since this is a linear equation, you will be able to write an explicit iteration. Decide on a reasonable interpolant for your numerical solution and compute the residual of your solution. Recall that this problem is very ill-conditioned on longer time intervals, although both this method and `ode45` work well enough on this short interval.

13.14. Consider solving the first-order system

$$\frac{dp}{dt} = -q$$

$$\frac{dq}{dt} = p \tag{13.317}$$

by the following mixture of forward Euler and backward Euler, sometimes known as the leapfrog method:

$$\begin{aligned} p_{n+1} &= p_n - hq_n \\ q_{n+1} &= q_n + hp_{n+1} . \end{aligned} \tag{13.318}$$

Find a modified equation showing that this method is *second*-order accurate. In order to do this, you will have to put $q_n \approx q(t_n + h/2)$, not $q(t_n)$. This example is perhaps the simplest problem solved by a *symplectic* method.

13.15. Draw the stability region in the $\mu = h\lambda$-plane for the implicit LMM (13.230). Compare this with the stability region for the predictor–corrector method (13.233).

13.16. Show that the stability polynomial for a general fixed-time-step LMM (13.223) is $r^k = p(r,q)$, with

$$p(r,q) = \rho(r) - q\sigma(q) \,, \tag{13.319}$$

where

$$\rho(r) = \sum_{j=1}^{k} \alpha_{k-j+1} r^j \quad \text{and} \quad \sigma(r) = \sum_{j=0}^{k} \beta_{k-j} r^j \,. \tag{13.320}$$

Show also that the stability polynomial of a predictor–corrector method in P(EC)^{n}E mode is

$$p_c(r,\mu) + \frac{q^n(1-q)}{1-q^n} p_p(r,\mu), \tag{13.321}$$

where p_c is the stability polynomial of the corrector and p_p is the stability polynomial of the predictor. For the predictor–corrector pair in (13.233), but now with two iterations of the corrector, that is, $n = 2$, find and draw the stability region.

13.17. Embed the simple harmonic oscillator $y'' + y = 0$ in a two-dimensional first-order system $\dot{\mathbf{y}} = \mathbf{A}\mathbf{y}$ as usual. Consider the backward Euler method applied to this system, $\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{A}\mathbf{y}_{n+1}$. Suppose that $\mathbf{y}_n = [s + hc, c - sh]^T$ and show that $\mathbf{y}_{n+1} = [s,c]^T$. Find a modified equation $\dot{\mathbf{z}} = \mathbf{A}(-h)\mathbf{z}$ such that the residual for this equation is $O(h^2)$. Show that the eigenvalues of $\mathbf{A}(-h)$ have real part $-h/2$ and thereby explain why backward Euler introduces positive numerical damping into this system. Carry out the integration over a suitable time interval, and verify that the backward Euler solution of $\dot{\mathbf{y}} = \mathbf{A}\mathbf{y}$ is actually a better fit to the reference solution of $\dot{\mathbf{z}} = \mathbf{A}(-h)\mathbf{z}$.

13.18. One irritating problem with embedding higher-order differential equations into first-order systems of equations is the resulting meaning of the residual as computed by the code. To be specific, consider a scalar second-order equation $\ddot{x} = f(t,x,\dot{x})$ with initial conditions $x(t_0) = x_0$ and $\dot{x}(t_0) = \dot{x}_0$. Suppose we integrate this with a Runge–Kutta method, say `ode45`, after converting to the first-order system $y_1 = x$, $y_2 = \dot{x}$, so our equations are $\dot{y}_1 = y_2$, $\dot{y}_2 = f(t,y_1,y_2)$. When we use `deval` to evaluate the interpolant and its derivative, we get a polynomial interpolant $z_1(t)$ for $y_1(t)$ satisfying certain constraints: namely, that $z_1(t)$ takes on the values `sol.y(1,:)` at the points `sol.x(:)`. We can also evaluate its derivative exactly, up to roundoff, because the interpolant is a polynomial. We also get an interpolant $z_2(t)$, which takes on the values `sol.y(2,:)` at the mesh points.

We engineered the problem so that $z_2(t) \doteq \dot{z}_1(t)$, but unfortunately it's not exactly equal; instead, $\dot{z}_1(t) = z_2(t) + \Delta_1(t)$, which contains the first component of the residual. So if we want the second derivative of $z_1(t)$ to put into the original second-order equation, it seems we must in effect differentiate $\Delta_1(t)$. Of course, this is possible, but it apparently loses an order of accuracy: If $z_1(t)$ is $O(h^{p+1})$ accurate, then $\dot{z}_1(t)$ will only be $O(h^p)$ accurate, and $\ddot{z}_1(t)$ will only be $O(h^{p-1})$ accurate, and so the residual in our second-order equation will be $O(h^{p-1})$, not $O(h^p)$.

Another approach is to start with the polynomial $\dot{z}_2(t)$, which is the exact derivative of the polynomial $z_2(t)$ and is returned by using `deval`. We can then attempt to construct a polynomial, call it $z(t)$, which is the exact integral of $z_2(t)$, and which satisfies $z(t_k) = z_1(t_k)$ for some fixed t_k. This will give an $O(h^p)$ residual in the second-order equation. Unfortunately, it won't quite be the case that $z(t_{k+1}) = z_1(t_{k+1})$. Why not?

13.19. Suppose we are solving the differential equation $\dot{x} = f(x)$. If the local error function is $\varepsilon(t)$, so that the computed solution is $z(t) = x_k(t) + \varepsilon(t)$, then the residual $\Delta(t)$ can be shown to be

$$\Delta(t) = \dot{\varepsilon}(t) + f(z-\varepsilon) - f(z) \tag{13.322}$$

and hence that $\|\Delta(t)\| \leq \|\dot{\varepsilon}\| + L\|\varepsilon\|$, where L is a Lipschitz constant for f. Fill in the details, and show that using $t = t_k + h\theta$ to write the leading term in $\varepsilon(t)$ as $h^{p+1}E(\theta) + O(h^{p+2})$ gives $\Delta(t) = E'(\theta)h^p + O(h^{p+1})$.

13.20. Show that $z = h\lambda$ is inside the disk $|z+1| < 1$ with $h > 0$ if and only if $h < -2\mathrm{Re}(\lambda)/|\lambda|^2$ and $\mathrm{Re}(\lambda) < 0$.

13.21. Consider solving $\dot{y} = \lambda y$ by the forward Euler method, so that $y_{n+1} = y_n + h\lambda y_n$. Show that $y_n = (1+h\lambda)^n y_0$. Rewrite this as

$$(1+h\lambda)^n = e^{nh\left(\ln(1+h\lambda)/h\right)} \tag{13.323}$$

and interpret this as the samples at $t_n = nh$ of the *reference* solution of $\dot{y}(t) = \Lambda y(t)$ for

$$\Lambda = \frac{\ln(1+h\lambda)}{h} = \lambda\left(1 - \frac{1}{2}\lambda h + \frac{1}{3}\lambda^2 h^2 - \frac{1}{4}\lambda^3 h^3 + O\left(h^4\right)\right). \tag{13.324}$$

This is one of the few examples where an infinite-order modified equation can be found for a numerical method, valid for all step sizes h. Repeat the exercise, using the backward Euler method.

13.22. Consider the second-order problem $\ddot{x}(t) = f(t,x(t),\dot{x}(t))$ with initial values $x(t_0) = x_0$, $\dot{x}(t_0) = \dot{x}_0$. Apply Euler's method to the usual transformed first-order system with $y_1 = x$ and $y_2 = \dot{x}$. We know that the residual will be $O(h)$. Now consider applying a modification of Euler's method directly to the second-order system: Advance the *derivative* of the solution by $\dot{x}_{k+1} = \dot{x}_k + hf(t_k,x_k,\dot{x}_k)$, that is, Euler's method, but advance the solution itself by the integral of the linear interpolant, that is, $x_{k+1} = x_k + h\dot{x}_k + {}^{h^2 f(t_k,x_k,\dot{x}_k)}/_2$. Note that this is no more expensive than Euler's method [counting only evaluations of $f(t,x,\dot{x})$, as is usual]. Interpolate the solution by replacing h with $(t-t_k)$. Show that the residual in the second-order equation is still $O(h)$. Does this variation make much difference? Discuss.

(We have taken to calling this method the *OCD Euler method*, where OCD stands for Obsessive-Compulsive Disorder. After all, it's still a first-order method and it's not immediately evident that the extra work (trivial as it is) gets you anything. However, it has some advantages, as you will doubtless discover.)

13.23. Use the OCD Euler method (see Problem 13.22) to solve the IVP $\ddot{x}(t) + \sin x(t) = 0$, $x(0) = 2$, and $\dot{x}(0) = 0$, on the interval $0 \le t \le 8\pi$. Compute the residual and plot it. Use a fixed time step, but take $h = {}^{8\pi}/_n$ small enough that the residual is everywhere smaller than 1.5×10^{-3}. Compare the OCD Euler method with the ordinary Euler method on this example.

13.24. Find a modified equation whose solution more nearly tracks the RK2 solution of $\dot{x} = x^2 - t$, $x(0) = {}^{-1}/_2$. RK2 is the method defined by

$$\begin{aligned}
\mathbf{k}_1 &= f(t_n, x_n) \\
\mathbf{k}_2 &= f(t_n + {}^{h}/_2, x_n + {}^{h\mathbf{k}_1}/_2) \\
x_{n+1} &= x_n + h\mathbf{k}_2\,.
\end{aligned}$$

Find an $O(h^4)$ accurate modified equation. See the discussion in Corless (1994a).

13.25. If you have access to MAPLE or another CAS, use it to construct an explicit Taylor series method that is $O(h^5)$ accurate for solving $\dot{x} = x^2 - t$, $x(0) = {}^{-1}/_2$ on $0 \le t \le T$. Use the natural interpolant and monitor the residual. Does the stiffness of the problem bother the method for large T? Is a lower-order method happier with the stiffness? Implement the method in an implicit manner as well: that is, to compute x_{k+1}, use Taylor expansion about $t = t_{k+1}$ and choose x_{k+1} so that Taylor series gives x_k at $t = t_k$ (you may use a simple initial guess such as $x_{k+1}^{(0)} = x_k$ to start your Newton iteration). Does the implicit method work better for large T?

Investigations and Projects

13.26. In Problem 13.7, we have encountered a difficulty with the fixed-step-size RK4 method. We hasten to add that the difficulty isn't with the Runge–Kutta part, it's with the fixed-time-step part: If a mechanism to estimate errors and adjust step size is added, then RK4 can get a good answer for this problem on this interval. Over longer intervals, as we have seen, the problem is stiff and RK4 needs unreasonably small step sizes to solve it, though. Make some numerical experiments with simple step-size adaptation rules and show your results. (Of course, when you add an error-estimation mechanism and adaptive step size, and modify the coefficients for efficiency, you get something like `ode45`.)

13.27. Write a MATLAB program to solve first-order initial-value problems for ODE $\dot{y}(t) = f(t, y(t))$ on an interval $a \le t \le b$ with given initial conditions $y(a) = y_a$ to a given tolerance by a *defect-controlled Euler method*. Specifically, put

$$\mathbf{k}_0 = f(t_n, y_n) \qquad \text{and} \qquad y_{n+1} = y_n + h_n \mathbf{k}_0$$

and accept this step only if the following estimate of the defect (residual) is smaller than the user's input tolerance ε:

$$\|\Delta(t)\|_\infty \approx \frac{3}{4}|\mathbf{k}_1 - \mathbf{k}_0| \le \varepsilon\,,$$

where

$$\mathbf{k}_1 = f(t_n + h_n, y_{n+1})$$

will be needed for the next step anyway, if the step is successful. Use the size of your estimated defect to predict the step size h_{n+1} to take on the next step. Your code should have the header `ode1d`, and have the same interface that `ode23tx` does. (Don't work too hard on the interface.) For adapting the step size, you should either modify the heuristics of `ode23tx` appropriately, or use the more advanced method of Söderlind (2003). Your code will therefore give the *exact* solution to $\dot{y}(t) = f(t, y(t)) + \Delta(t)$, with $\|\Delta(t)\| \le \varepsilon$.

In addition, to complete the investigation, do the following:

1. Show that interpolating your solution by the piecewise cubic function [with the variable $\theta = {}^{(t-t_n)}/_{h}$]

$$y(t) = y_n + h\left(1 - \theta - \theta^2\right)\theta\mathbf{k}_0 + \theta^2 h(\theta - 1)\mathbf{k}_1 \tag{13.325}$$

   gives a defect $\Delta(t) = \dot{y}(t) - f(t, y(t))$ that has its approximate maximum (for small h) at $\theta = {}^1/_2$, and that maximum is approximately $\frac{3}{4}|\mathbf{k}_1 - \mathbf{k}_0|$ as used above. Note that this is $O(h)$ as $h \to 0$.
2. Test your program on $\dot{y}(t) = -y(t)$, $y(0) = 1$ on $0 \le t \le 1$. Use $\varepsilon = 5 \cdot 10^{-2}$.
3. Test your program on

$$\dot{\mathbf{y}}(t) = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{y}(t) \tag{13.326}$$

with initial conditions $\mathbf{y}(0) = [1,0]^T$. Integrate over $0 \le t \le 4\pi$. Use $\varepsilon = 5 \cdot 10^{-2}$ again, and plot $y_1^2 + y_2^2 - 1$ on $[0, 4\pi]$. Repeat with $\varepsilon = 10^{-2}$.

4. How accurately does your code locate the singularity in the solution of $\dot{x}(t) = x(t)^2 + t^2$, $x(0) = 1$, on $0 \le t <$ singularity?
5. Compare your code `ode1d` to fixed-Δt Euler's method on the problems above.
6. The "flame problem" describes combustion of a match, in simplistic terms. The differential equation is

$$\dot{x} = x^2 - x^3, \tag{13.327}$$

and the initial condition is $x(0) = \varepsilon$, a small positive number. We wish to integrate this on the interval $0 \le t \le 2/\varepsilon$. This problem is described in Moler (2004 section 7.9), and also in Corless et al. (1996), where it is solved exactly in terms of the Lambert W function: $x(t) = 1/(1 + W(u\exp(u-t)))$, where $u = -1 + 1/\varepsilon$. Use your code to solve this problem. Compare the solution with that of `ode45` and of `ode15s`, and with the reference solution.

7. Solve $\dot{x} = \cos(\pi t x)$, for a vector of initial conditions $x_0 = 0 : 0.2 : 2.8$, on the interval $0 \le t \le 5$. Plot all your solutions on the same graph.
8. Solve the Lorenz system with your code for some chaotic values of the parameters. In view of "sensitive dependence on initial conditions," what good is your numerical solution?

13.28. Defect-controlled RK(2)3 pair investigation. In the text, the defect-controlled method

$$\begin{array}{c|cccc}
2/3 & 2/3 & & & \\
2/3 & & 2/3 & & \\
1 & 1/4 & 3/8 & 3/8 & \\
\hline
 & b_1(\theta) & b_2(\theta) & b_3(\theta) & b_4(\theta)
\end{array}$$

was mooted; here the $b_i(\theta)$ are

$$b_1(\theta) = -\frac{5}{4}\theta^2 + \frac{1}{2}\theta^3 + \theta \tag{13.328}$$

$$b_2(\theta) = \frac{9}{8}\theta^2 - \frac{3}{4}\theta^3 \tag{13.329}$$

$$b_3(\theta) = \frac{9}{8}\theta^2 - \frac{3}{4}\theta^3 \tag{13.330}$$

$$b_4(\theta) = -\theta^2 + \theta^3, \tag{13.331}$$

and the stage $\mathbf{k}_4$ can be reused as the first k-value on the next step. This method has another advantage built-in: There are two embedded *second*-order methods, namely,

$$\hat{\mathbf{y}}_2(\theta) = \mathbf{y}_n + h((\theta - \frac{3}{4}\theta^2)\mathbf{k}_1 + \frac{3}{4}\theta^2\mathbf{k}_2) \tag{13.332}$$

and

$$\hat{\mathbf{y}}_3(\theta) = \mathbf{y}_n + h((\theta - \frac{3}{4}\theta^2)\mathbf{k}_1 + \frac{3}{4}\theta^2\mathbf{k}_3)\,. \tag{13.333}$$

The residuals of $\hat{\mathbf{y}}_2$ and $\hat{\mathbf{y}}_3$ are each $O((\theta h)^2)$, and this will allow us to monitor the error and adapt the step size. Notice that the residual of

$$\mathbf{y}(\theta) = \mathbf{y}_n + h\sum_{k=1}^{4} b_k(\theta)\mathbf{k}_k \tag{13.334}$$

is $O(h^3)$ and therefore ought to be smaller, but might not be.

In the following questions, we investigate this method.

1. Show that this interpolant is mathematically the same as cubic Hermite interpolation on $t_n \le t \le t_{n+1} = t_n + h$. Recall $\theta = {}^{(t-t_n)}/_{h}$.
2. Show that the defect estimate in $\mathbf{y}_3$ is asymptotically accurate to $O(h^3)$ as $h \to 0$; that is,
$$\Delta(\theta) = \theta(1 - \frac{3\theta}{2})\mathbf{k}_1 - \frac{9}{4}\theta(1-\theta)\mathbf{k}_2 + \frac{3\theta}{4}(3\theta - 1)\mathbf{k}_3 + \theta(2 - 3\theta)\mathbf{k}_4 + O(h^3).$$
3. Show (by using a CAS or otherwise) that this $\Delta(\theta)$ is, to leading order in h, the same as $\Delta_3(\theta) = {}^{\hat{y}_3'}/_{h} - f(\hat{y}_3)$, where
$$\hat{y}_3(\theta) = y_n + h((\theta - \frac{3}{4}\theta^2)\mathbf{k}_1 + \frac{3}{4}\theta^2\mathbf{k}_3)\,, \tag{13.335}$$
namely,
$$\Delta_3(\theta) = -\frac{h^2}{6}\theta((3\theta - 2)f^{(2)}(f_1 f) + (3\theta - 4)f^{(1)}(f^{(1)}(f)) + O(h^3)\,.$$
4. Argue that, except in the unlucky case when $f^{(2)}(f_1 f) \doteq -f^{(1)}(f^{(1)}(f))$, putting $\theta = 1$ in (13.335) gives a decent asymptotic estimate of the maximum defect. What happens if you're unlucky? Do the same analysis for $\Delta_2(\theta) = {}^{\hat{y}_2'}/_{h} - f(y_2)$ and show that you can't be unlucky for both Δ_2 and Δ_3 at the same time, for $\theta = 1$, unless both $f^{(2)}(f_1 f)$ and $f^{(1)}(f^{(1)}(f))$ are very small (i.e., $O(h)$), and even then this just usually means that both Δ_2 and Δ_3 are small at this point in the integration.
5. Copy to a directory of your own the MATLAB m-files `ode23.m`, `ntrp23.m`, `deval.m`, `private/odearguments.m`, and `odefinalize.m`. Modify these files so that the method used is the $2/3$ pair discussed here. Test your code on a variety of examples and compare its performance to that of `ode23`. (We get comparable performance for similar residual accuracy, but with `deode23` giving predictably better residuals for a given tolerance—but when the tolerance for `ode23` is tightened further, similar amounts of work produce similar residuals.)

13.29. This is a supplement to Problem 13.28, in which we ask: Where did those c_i, a_{ij}, and $b_i(\theta)$ come from in the method RK(2)3? It's all very well to simply point to a tableau, such as

$$\begin{array}{c|cccc} u & u & & & \\ v & \frac{v(-3u+3u^2+v)}{u(-2+3u)} & \frac{v(v-u)}{u(2-3u)} & & \\ 1 & \frac{6uv-3u+2-3v}{6uv} & \frac{3v-2}{6u(v-u)} & & \frac{2-3u}{6v(v-u)} \\ \hline & b_1 & b_2 & b_3 & b_4 \end{array}\,, \tag{13.336}$$

where

$$b_1(\theta) = -\frac{1}{3}\frac{(-3u+2+3uv-3v)\theta^3}{3uv} + \frac{(-3u-3v+2uv+2)\theta^2}{2uv} + \theta \tag{13.337}$$

$$b_2(\theta) = \frac{(3v-2)\theta^3}{3u(-v+u)} - \frac{(3v-2)\theta^2}{2u(-v+u)} \tag{13.338}$$

$$b_3(\theta) = -\frac{(-2+3u)\theta^3}{3v(-v+u)} + \frac{(-2+3u)\theta^2}{2v(-v+u)} \tag{13.339}$$

$$b_4(\theta) = \theta^3 - \theta^2\,, \tag{13.340}$$

which, we are told, is a third-order CERK independent of the choice of parameters u and v so long as $u \neq 0, v \neq 0, u \neq v$ and $u \neq 2/3$. This family was taken from Owren and Zennaro (1991). But how do we obtain this?

Show that for a scalar f (the vector case is more notationally involved), the residual $\varDelta = y'/h - f(y)$ has a series expansion

$$\varDelta = h^3\theta(1-\theta)\Big(A(u,v)f^{(3)}(f,f,f) + B(u,v)f^{(2)}(f^{(2)}(f),f) + \frac{1}{6}(\theta-2)f^{(1)}(f^{(1)}(f^{(1)}(f)))\Big) + O(h^4)\,, \tag{13.341}$$

where

$$A = -\frac{1}{2}uv + \frac{1}{6}\theta - \frac{1}{3} + \frac{1}{3}u + \frac{1}{3}v \tag{13.342}$$

$$B = -\frac{4}{3} + \frac{1}{2}u + \frac{2}{3}\theta + v\,. \tag{13.343}$$

No choice of u or v can eliminate all the error, but $\theta(1-\theta)$ is maximum at $\theta = 1/2$, which may be useful because one does not want to underestimate the error.

13.30. This problem explores adaptive step-size selection for the simple second-order linear multistep method discussed in 13.9. This book has not discussed how existing codes actually adapt step sizes. What this investigative problem does is ask you to explore some nonstandard but suggestive ideas. There's no guarantee that these ideas will work as well as the standard ideas do; we're not advocating them as being practical. What we do hope that you get out of this problem is a sense of what kinds of thoughts were going through the heads of the people who originally came up with the methods for adapting time steps for linear multistep methods. There is also some contact with other branches of mathematics, especially time series analysis and control theory.

What we ask you to do is to model the residual error on each time step $t_{k-1} \leq t \leq t_k$ as $|\Delta_k| = \phi_k h_k^p$, where $h_k = t_k - t_{k-1}$, and for the third-order method we are considering, $p = 3$. You might *measure* the residual error on the current step by comparing the difference between the derivatives of the interpolant at $t = t_{k-1}$ and at $t = t_{k-2}$ and the (remembered) computed values of $f(t, x(t))$ at those points. These are "free" estimates of the size of the residual (although of possibly unknown quality). The interpolant, of course, matches the values x_{k-2}, x_{k-1}, x_k, and $f(t_k, x_k)$ at the three nodes t_{k-2}, t_{k-1}, and t_k, so it is degree 3. Working out those values of the derivatives gives rise essentially to two finite-difference formulas. These measurements allow you to estimate the residual over the step $t_{k-1} \leq t \leq t_k$, which you can then compare to the tolerance; if the step is accepted, then you can use your estimated error to estimate ϕ_k. By using more function evaluations than you need to advance the solution, you can improve the estimate of the residual, of course.

For reference, here are the finite-difference estimates of the derivatives at $t = t_{k-1}$ and $t = t_{k-2}$, in which we assume that $t = t_k - \theta h$ and $h_k = rh$ and $h_{k-1} = sh$:

$$\begin{aligned}
\frac{1}{h} f'(t_{k-1}) &= \frac{s(3r+2s)\rho_{0,0}}{r(r+s)^2} - \frac{\rho_{0,1} s}{r+s} + \frac{(r-2s)\rho_{1,0}}{rs} - \frac{r^2 \rho_{2,0}}{s(r+s)^2} \\
\frac{1}{h} f'(t_{k-2}) &= -\frac{\rho_{0,0} s(3r+s)}{(r+s)r^2} + \frac{\rho_{0,1} s}{r} + \frac{(r+s)^2 \rho_{1,0}}{r^2 s} - \frac{\rho_{2,0}(r+3s)}{s(r+s)} . \qquad (13.344)
\end{aligned}$$

You can remove one parameter r or s by, for example, taking $r + s = 2$, so that h then becomes the average step length over the last two steps; alternatively, you could take $r = 1$ and $h = h_k$. The fitting values are $\rho_{0,0} = x_k$, $\rho_{0,1} = hf(t_k, x_k)$, $\rho_{1,0} = x_{k-1}$, and $\rho_{2,0} = x_{k-2}$.

Now, in order to take a step, you must *predict* the value of ϕ_{k+1}. One way to do this is to use a linear prediction of the form

$$\log \phi_{k+1} = \alpha \log \phi_k + \beta \left(\frac{\log \phi_k - \log \phi_{k-1}}{h_k + h_{k-1}} \right) , \qquad (13.345)$$

where the constants α and β are chosen in order to fit recent integration history in some fashion (for instance, by least-squares fit to the last three, four, or five steps). Once ϕ_{k+1} has been predicted, then to take the next step you have to solve

$$\phi_{k+1} h_{k+1}^p = \varepsilon \qquad (13.346)$$

for the unknown step size h_{k+1}; but this is easy, and $\log h_{k+1} = {}^{(\log \varepsilon - \log \phi_{k+1})}/_p$ or, equivalently,

$$h_{k+1} = \left(\frac{\varepsilon}{\phi_{k+1}} \right)^{1/p} . \qquad (13.347)$$

This equation can doubtless be rearranged for greater efficiency.

Your assignment is to try this method out. Use some convenient one-step method (e.g., `ode23`) to start the integration. This will get your solution on track quickly, and ought to give good starting values for the step sizes. Try your method out on some simple problems to start with, and then give it a workout on the `orbitode` problem. Discuss the performance—robustness, accuracy, and efficiency—of the method.

13.31. Extrapolation methods use Richardson's idea: Construct a skeleton of a rough solution on a coarse mesh, and then do it again on a finer mesh. Next, use the two solutions together to construct a more accurate solution. This is best demonstrated by example, which you can do yourself by following the outline below.

Solve the problem $\dot{x} = x^2 - t$ with $x(0) = -1/2$ on the interval $0 \le t \le 2$ using the explicit trapezoidal rule with fixed time step h, first by using $h = 1/8$ (call the solution e_k for $0 \le k \le 16$) and then again using $h = 1/16$ (half the size). Call that solution s_k for $0 \le k \le 32$. Because the residual is $O(h^2)$, so is the forward error in the solution: So $e_k = x(t_k) + c_k h^2 + \cdots$ and $s_\ell = x(t_\ell) + C_\ell h^2/4 + \cdots$. Matching up the times in the two solutions (when we take steps of half the size, every other step aligns the refined solution with the coarse solution) also turns out to match up the error coefficients, and so $S_k := (4s_{2k} - e_k)/3 = x(t_k) + (4c_k/4 - c_k)h^2/3 + O(h^4)$. Thus, the error terms cancel, and the averaged quantity $S_k = (4s_k - e_k)/3$ is a better approximation to the desired solution.

1. Use your trapezoidal rule solutions with $h = 1/8$ and $h = 1/16$ to estimate $x(2)$ to fourth-order accuracy.
2. Do this again with $h = 1/16$ and $h = 1/32$.
3. Do this again with $h = 1/32$ and $h = 1/64$. Of course, you can reuse your earlier work. Now you should have two columns containing estimates of $x(2)$, which can be arranged like so:

h	e_k	S_k
$1/8$	$x.xxxx$	$---$
$1/16$	$x.xxxx$	$x.xxxx$
$1/32$	$x.xxxx$	$x.xxxx$
$1/64$	$x.xxxx$	$x.xxxx$

(13.348)

   and the bottom-right entry should be the most accurate (so far).
4. Of course, this idea can be repeated. The entries S_k can be shown to have an error expansion of the form $S_k = x(t_k) + a_4(t_k)h^4 + a_6(t_k)h^6 + \cdots$. Show that the averages $T_k := (8S_{2k} - S_k)/7$ have error $O(h^6)$, and add a third column to your table.
5. Repeating again, the desired average is $U_k = (16T_{2k} - T_k)/15$. Add a fourth column. Compare your best answer with a value of $x(2)$ computed using (say) `ode113` at tight tolerances. Examine also the error estimate obtained by comparing the diagonal entries in the (now-triangular) table.

Notice that the work doubles as you add another row, but that completion of the triangle is rather trivial. This is the power of extrapolation. A full description of extrapolation methods can be found in, for example, Hairer et al. (1993).

13.32. The leapfrog method applied to the Hénon–Heiles Hamiltonian H_0 gave a better solution to a perturbed Hamiltonian $H_0 + h^2 H_2 + O(h^4)$. A formula for H_2 was given in Eq. (13.311). What happens if you apply the leapfrog method to the perturbed Hamiltonian $H_0 - h^2 H_2$? Note the minus sign. Do some computations to verify or disprove your contention.

Since the computation of H_2 required partial derivatives of H_0 (there is a formula in Preto and Tremaine (1999) using Poisson brackets), this in effect uses a second-derivative method on the original problem. This leads to the theory of Lie series methods; see Channell and Scovel (1990) and Chartier et al. (2007).

13.33. The predictor–corrector method of Eq. (13.233) has a cubic stability polynomial (13.234). Instead of making a bifurcation argument as in the text, and considering just the boundary where the magnitude of one single root must be 1, one could use a more mechanical (and automatable) argument based on so-called semi-algebraic sets, and use the Hurwitz criterion (see Levinson and Redheffer 1970) that is implemented in the MAPLE package `PolynomialTools [Hurwitz]`. That routine returns a sequence of polynomials that must be strictly positive for all the roots of the original polynomial to lie in the left half-plane. Of course, here we want the unit circle, but that presents no difficulty: If and only if r is inside the unit circle, then $z = {}^{(r+1)}\!/_{(r-1)}$ will lie in the left half-plane. Transform the polynomial into a rational function in z using this change of variables, throw away the denominator, and find the Hurwitz conditions (don't write them down, they're rather messy; but MAPLE handles them quite well). Plot them—you should get something like Fig. 13.19. Put $\mu = \sigma + i\tau$. Show that near $\sigma = 0$, that is the imaginary axis, $\sigma = -{}^{\tau^4}\!/_4 + {}^{125\tau^6}\!/_{24} + \cdots$, showing that the right-hand boundary of the quasitriangular region is not actually straight up and down. Thus, there are small regions where a large, nearly imaginary eigenvalue λ would have spurious numerical growth in the solution.

13.34. Start with a CERK of order 3 for first-order systems. By specializing to embedded second-order systems, $\dot{\mathbf{x}}_1 = \mathbf{x}_2$, $\dot{\mathbf{x}}_2 = \mathbf{f}(t, \mathbf{x}_1, \mathbf{x}_2)$, show that you can construct a CERK that is order 3 for the $\mathbf{x}_2$ components and order 4 for the $\mathbf{x}_1$ components by simply integrating the CERK step for the $\mathbf{x}_2$ components exactly, and that therefore the residual in the *second*-order system $\ddot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}, \dot{\mathbf{x}})$ is explicitly $O(h^3)$. Try your method out on a scalar problem, say $\ddot{x} + 2\zeta\dot{x} + \sin(x) = 0$, where $\zeta = 10^{-2}$, $x(0) = 0$, $\dot{x}(0) = 1$. Then try your method on the Arenstorf problem (as in `orbitode` in the `odeexamples` demo). Is the apparent extra accuracy in the residual worth the effort?

13.35. Implicit methods applied to $\dot{y} = \lambda y$ produce rational Padé approximants to $\exp(\lambda h)$. It is tempting to try an explicit Taylor series method with an extra step of converting from a polynomial to a rational Padé approximant. That is, use the Taylor series method to start to solve $\dot{x} = f(t, x)$ in one step, but instead of defining x_{k+1}

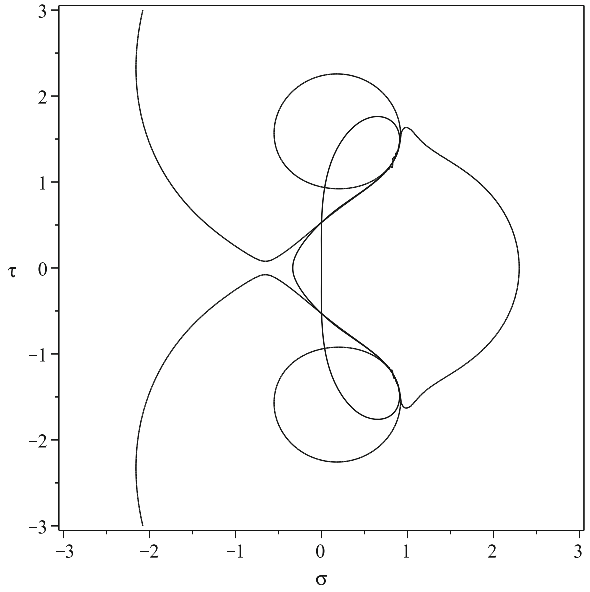


Fig. 13.19 The boundaries of all the Hurwitz conditions ensuring that all three roots of (13.234) are less than one in magnitude. The polynomial has all its roots inside the unit circle if and only if all Hurwitz conditions are satisfied, which in this case means that μ must lie outside the two ovals from the first condition, to the left of the curious "bent-wire" shape opening up to the left that comes from the second, and inside the loops of the third. This boils down to just that of Fig. 13.14

as the value of the Taylor polynomial at $z(t_k+h)$, one first finds an equivalent Padé approximant, a function

$$\frac{1}{a_0+a_1h+\cdots a_nh^n}, \tag{13.349}$$

which has the same Taylor series. This is easily done and is implemented in `convert(s,ratpoly)` in Maple, for example. This works well for scalar equations and in particular works well for $\dot{y}=\lambda y$ and seems to cure stiffness; one is tempted to say that we have an explicit method for stiff systems. This would be a major advantage, because solving nonlinear systems is the hard part for an implicit method.

Show that this is a pipe dream, by showing that the method introduces moveable poles into the solution even for *linear* systems of equations $\dot{\mathbf{x}}=\mathbf{f}(t,\mathbf{x})$. A general proof is not necessary; all you have to do is find an example. Excluding the moveable poles requires a limitation on step size h that is comparable to the limitation needed by explicit methods.

13.36. Consider the best possible interpolant of the solution to $y'=\lambda(y-g(t))+g'(t)$ on a step from y_n at $t=t_n$ to y_{n+1} at $t=t_{n+1}$. We say that it is the "best possible" in that the interpolant minimizes the 2-norm of the residual

$$\min_z \int_{t_n}^{t_{n+1}} \Delta^*(\tau)\Delta(\tau)d\tau\,, \tag{13.350}$$

where $\Delta(\tau) = \dot{z}(\tau) - f(z(\tau))$, taking the autonomous form (one dimension higher, if necessary) for convenience. The Euler–Lagrange equations for this (nonconvex if f is nonlinear) functional are

$$\dot{z} - f(z) = \Delta \tag{13.351}$$

$$\dot{\Delta} + J_f^H(z)\Delta = 0\,, \tag{13.352}$$

where $J_f^H(z)$ is the conjugate transpose of the Jacobian matrix of f. These $2(n+1)$ equations, called the *adjoint equations*, are subject to the boundary conditions

$$z(t_n) = \begin{bmatrix} y_n \\ t_n \end{bmatrix} \quad \text{and} \quad z(t_{n+1}) = \begin{bmatrix} y_{n+1} \\ t_{n+1} \end{bmatrix}. \tag{13.353}$$

This will lead to an interpolant in $\mathscr{C}^2(t_n, t_{n+1})$, which is continuous but not differentiable at t_n and t_{n+1}. This lack of differentiability is of little consequence: Our residual is called a "deviation" in this case. When we set up and solve this BVP for our one-dimensional system, taking $g(t) = 0$ to simplify the algebra, we get

$$\dot{z} - \lambda z = \Delta \tag{13.354}$$

$$\dot{\Delta} + \overline{\lambda}\Delta = 0 \tag{13.355}$$

or $\Delta = c_1 e^{-\overline{\lambda}(t-t_n)}$, and provided $\lambda \notin i\mathbb{R}$ (in which case we would have to introduce a secular term in z),

$$z = c_2 e^{\lambda(t-t_n)} - \frac{c_1}{\lambda + \overline{\lambda}} e^{-\overline{\lambda}(t-t_n)}\,. \tag{13.356}$$

Using $z(t_n) = z_n$ and $z(t_{n+1}) = z_{n+1}$ to identify c_1 and c_2, we have

$$c_2 = \frac{z_{n+1} - e^{-\overline{\lambda}h} z_n}{e^{\lambda h} - e^{-\overline{\lambda}h}}, \qquad \text{with } h = t_{n+1} - t_n \tag{13.357}$$

$$c_1 = \frac{\lambda + \overline{\lambda}}{e^{\lambda h} - e^{-\overline{\lambda}h}} (z_{n+1} - e^{\lambda h} z_n)\,. \tag{13.358}$$

Since $\Delta = c_1^{-\overline{\lambda}(t-t_n)}$, the size of the best possible residual is proportional to $z_{n+1} - e^{\lambda h} z_n$, for any method producing z_{n+1}. The proportionality constant,

$$\frac{\lambda + \overline{\lambda}}{e^{\lambda h} - e^{-\overline{\lambda}h}} e^{-\overline{\lambda}(t-t_n)}\,, \tag{13.359}$$

is independent of the method, but is $O(1/h)$ as $h \to 0$. But $z_{n+1} - e^{\lambda h} z_n$ is the *local error* of the method: If z_n were exact, then z_{n+1} should be $e^{\lambda(t_{n+1}-t_n)} z_n$ if the equation $\dot{y} = \lambda y$ were solved exactly. Thus, for this simple equation, the best possible residual is proportional to the local error per unit step. This, incidentally, supports the idea that codes that control the size of the local error per unit step have a small residual.

Your assignment for this question is to take $\lambda = -10$ on the interval $0 \le t \le 1$, solve the problem with $g(t) = 0$ using `ode23`, and compute the local errors made on each step. Then, using the formulæ above, compute the optimal residual on each step and plot it. Also, plot the residual that is obtained when using the built-in interpolant, and compare the two.

13.37. The venerable technique of *Picard iteration* is mostly useful as a theoretical technique for proving the existence and uniqueness of solutions to initial-value problems, but it can be used computationally (it's just not very efficient, when it works at all). Here we try implementing it in `chebfun`. Picard iteration for solving $\dot{x} = f(t,x)$, $x(t_0) = x_0$, consists of first rewriting this as the integral

$$x(t) = x_0 + \int_{t_0}^{t} f(\tau, x(\tau))\, d\tau\,. \tag{13.360}$$

Then, one takes an initial guess that is assumed to be valid on an interval, say (t_0, t_1); we take $x^{(0)}(t) = x_0$ constant, here, because this satisfies the initial condition. Then, we iterate

$$x^{(k+1)}(t) = x_0 + \int_{t_0}^{t} f(\tau, x^{(k)}(\tau))\, d\tau \tag{13.361}$$

for $k = 0, 1, 2, \ldots$ until we are happy with the answer.

What goes wrong with this in exact computation is that sooner or later you run into an integral that you can't express in terms of elementary functions. One might try series, or other approximations at that point. What you are asked to do is to use `chebfun` right from the start, because the integration of chebfuns is easy using `cumsum`: The integral of a chebfun is another chebfun.

Consider the problem $\dot{x} = \cos(\pi t x)$, $x(0) = 1$, on the interval $0 \le t \le 2$. How many iterations does it take to get a `chebfun` solution with residual smaller than 5×10^{-13}? (Our answer: 32 iterations, with a maximum residual less than 2.5×10^{-13}. Moreover, if we used Taylor series instead, then even having 100 iterations doesn't give us anything like this accuracy, although the solution is very accurate on, say, $0 \le t \le 1/4$.)

13.38. At several places in the book we have mentioned the Mandelbrot polynomials defined by $p_0 = 1$ and $p_k = z p_{k-1}^2 + 1$ for $k \ge 1$. We have shown how to compute zeros by eigenvalue techniques. This problem explores a potentially much faster method.

Suppose $p_k(z)$ and its zeros are known for $0 \le k \le N-1$. Consider now the polynomial

$$p_{N,t}(z) = z p_{N-1}^2(z) + t^2 = p_N(z) - 1 + t^2\,.$$

When $t = 0$, the roots are known: 0 and double roots at $\xi_{N-1,j}$, $1 \leq j \leq 2^{N-1} - 1$. When $t = 1$, the roots are the desired roots of $p_N(z)$.

1. Find a series expansion of z_t, where $0 = p_{N,t}(z_t)$ beginning $z_t = \xi_{N-1,t} + \beta_1 t + \beta_2 t^2 + \cdots$ (you will find two series).
2. Show that z_t satisfies the initial-value problem

$$\frac{d}{dt} z_t = -\frac{2t}{p_N'(z_t)}$$
$$z_t|_{t=\varepsilon>0} = \xi_{N-1,j} + \beta_1 \varepsilon + \beta_2 \varepsilon^2 + \cdots .$$

3. Either use a prewritten solver with singularity detection, together with the technique of pole-vaulting, to find all the roots of $p_N(z)$, or write your own defect-controlled Taylor series solver to do the same. For how large an N can you solve $p_N(z) = 0$?

Chapter 14
Numerical Solutions of Boundary Value Problems

Abstract Boundary value problems (BVP) for ordinary differential equations are more than a simple extension of initial-value problems; indeed, it may be more fairly said that *initial-value problems are a degenerate case of BVP*. The MATLAB codes for solving BVPs are *remarkably similar to call and assess* although their *detailed behavior and algorithms are quite different*. In this chapter, we look at linear problems, *quasilinearization to solve nonlinear problems*, and the principle of *equidistribution* on an optimal mesh. ◁

14.1 Introduction

Boundary value problems for ordinary differential equations (BVPODE, or simply BVP for short) are qualitatively different than initial-value problems for ordinary differential equations (IVPODE, or just IVP). Where a simple marching algorithm succeeds for IVP, and the temporal mesh can be adapted essentially optimally using only local information as one proceeds with the integration, the case is different for BVPs. Conditions ensuring the uniqueness of the solution are typically separated: One part of the solution is known at one place, and another part at another place. Indeed, it is often more natural to think of the independent variable for BVP as a spatial variable, although when the independent variable *can* be thought of as "timelike," there may be memory savings in treating the problem in a special way. We do not pursue such methods here, beyond a passing mention below.

Instead, we consider "spacelike" problems, where there is no preferred direction for the independent variable or order in which to generate the mesh. More importantly, the mesh on which computation proceeds is not usually known before starting the solution process but somehow needs to be computed simultaneously with the solution; indeed, the determination of an optimal mesh can itself be a significant difficulty.

Example 14.1. Chapter 1 of Ascher et al. (1988) contains a long list of example applications of boundary value problems. We mention one of particular interest to

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_14,

our mathematical biology group, a classical and much-studied model of infection due to measles:

> Consider the following epidemiology model. Assume that a given population of size N can be divided into four categories: susceptibles, whose number at time t is $S(t)$; infectives $I(t)$; latents $L(t)$; and immunes $M(t)$. We have
>
> $$S(t) + I(t) + L(t) + M(t) = N$$
>
> for $t \in [0,1]$. Under certain assumptions on the disease, its dynamics can be expressed as
>
> $$\begin{aligned} \dot{y}_1 &= \mu - \beta(t) y_1 y_3 \\ \dot{y}_2 &= \beta(t) y_1 y_3 - \frac{y_2}{\lambda} \\ \dot{y}_3 &= \frac{y_2}{\lambda} - \frac{y_3}{\eta}, \end{aligned} \tag{14.1}$$
>
> where $y_1 = {}^{S}\!/\!_{N}$, $y_2 = {}^{L}\!/\!_{N}$, $y_3 = {}^{I}\!/\!_{N}$, $\beta = \beta_0(1 + \cos 2\pi t)$, and representative values of the constants are $\mu = 0.02$, $\lambda = 0.0279$, $\eta = 0.01$, and $\beta_0 = 1575$. The solution sought is periodic; that is, $y(0) = y(1)$. (Ascher et al. 1988 p. 13, Example 1.10)

The authors go on to describe a trick [introduce a new vector variable and equation $\dot{\mathbf{C}} = 0$, with $y_i(0) = C_i(0)$ and $y_i(1) = C_i(1)$] to put the boundary conditions in standard form. You will be asked to solve this as Exercise 14.2. ◁

The treatment of this present chapter differs from our IVP treatment in earlier chapters in that instead of focussing on how to use the MATLAB codes `bvp4c` and `bvp5c` to solve BVPs, and on how to know the MATLAB solution is reliable, we instead concentrate on constructing a method to solve a simple BVP, writing the code ourselves, in order to highlight the ideas used in solving BVPs. One simple reason for this change in focus is that the MATLAB codes for BVPs are so similar to the codes for IVPs that the reader will hardly need more than the MATLAB help files and demos to learn to use them, and to use `deval` to compute and examine the residual in an a posteriori error analysis exactly as we did for IVPs.

For completeness, though, we will do one extended example using `bvp4c` in this style (so the focus isn't *completely* changed), but we will then immediately go on to solve the problem ourselves using the method of *collocation*.

Example 14.2. The extended example we choose is a simple scalar second-order linear equation with conditions given at each end of the interval:

$$\ddot{y} - 9\dot{y} - 10y = 0 \tag{14.2}$$

subject to $y(0) = y(10) = 1$ on $0 \le x \le 10$. In spite of using t as the independent variable, this equation is not timelike, as you will investigate in the exercises. Solving it with `bvp4c` appears to be easy; you simply execute the following commands, compute the solution, and plot it:

```
f = @(x,y) [y(2,:); 9*y(2,:)+10*y(1,:)];
x = 0:10;
solinit = bvpinit( x, [1,0] );
sol = bvp4c( f, @(ya,yb) [ya(1)-1; yb(1)-1], solinit );
```

```
xi = RefineMesh( sol.x, 20 );
[y,dy] = deval( sol, xi );
res = dy - f(xi,y);
size( sol.x )
% Yields [1 49]: mesh with 48 subintervals; bvp5c is better
figure(1), plot( sol.x, sol.y(1,:), 'ko', xi, y(1,:), 'k-' )
set(gca,'fontsize',16)
xlabel( 't','fontsize',16 ), ylabel( 'y','fontsize',16 )
figure(2), semilogy( sol.x(2:end), diff(sol.x), 'ko')
set(gca,'fontsize',16)
xlabel( 't','fontsize',16 ), ylabel( 'stepsizes','fontsize',16 )
figure(3), semilogy( xi, abs(res), 'k.' )
set(gca,'fontsize',16)
xlabel( 't','fontsize',16 ), ylabel( 'residual','fontsize',16 )
axis( [ 0, 10, 1.0e-8, 1] )
```

Note the `bvpinit` command in line 3; we cover quasilinearization in Sect. 14.6.

We display the plots generated by those commands in Figs. 14.1–14.3. The last plot is a bit worrying: The residual seems larger than we would have expected. Redoing the computation using `bvp5c` instead of `bvp4c` results in a mesh with 33 points, and a maximum residual of about 10^{-2}—a better-quality solution, which apparently took less effort. Running the code with `bvp5c` yet again, using tighter tolerances, shows that the first solution was perfectly satisfactory after all. ◁

Now that the use of MATLAB code has been illustrated, we move to one of our recurring theme, that is, conditioning.

14.2 Conditioning

The theory of conditioning of BVPs is strongly related to that of IVPs. However, there are some differences that are important. In this section, we will only consider examples from the linear theory.

We will consider just two examples. First, we look at Eq. (14.2) with the given boundary conditions, $y(0) = y(10) = 1$. See Problem 14.1, which shows that the *initial*-value problem with $y(0) = 1$ and $y'(0) = \alpha$ is exponentially sensitive to changes in α and is thus an ill-conditioned IVP. However, we will now show that the BVP, with $y(0) = y(10) = 1$, is *well*-conditioned. We do this directly by solving

$$y'' - 9y' - 10y = 0$$

subject to $y(0) = a$, $y(L) = b$ on $0 \le x \le L$. That is, we will allow changes in both the left and right boundary conditions, and even allow the right boundary to move. The reference solution is

$$y_0(x) = \frac{(ae^{10L} - b)e^{-x}}{e^{10L} - e^{-L}} + \frac{(b - ae^{-L})e^{10x}}{e^{10L} - e^{-L}}, \tag{14.3}$$

as can be found in MAPLE by executing

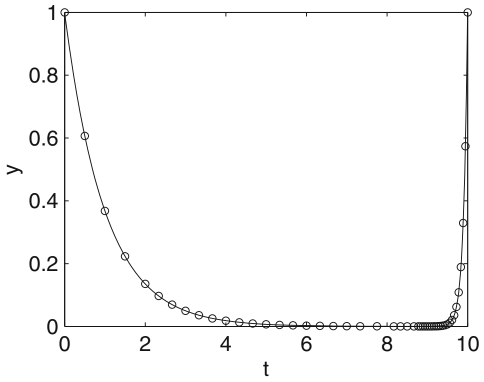


Fig. 14.1 Solution of (14.2) using `bvp4c` with default tolerances. Mesh points are plotted with circles, and the `deval` interpolant is used to fill in the gaps

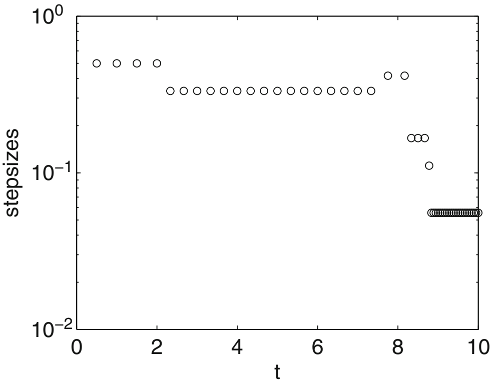


Fig. 14.2 Mesh widths used in the solution of (14.2) using `bvp4c` with default tolerances. Mesh widths are widest at the left end

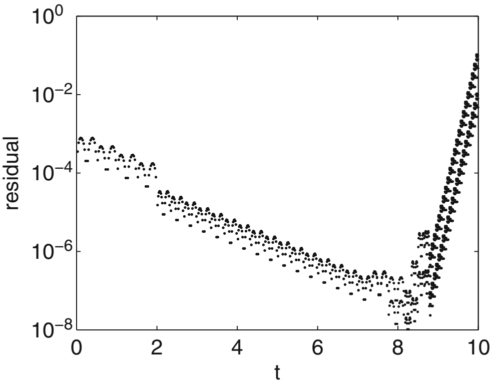


Fig. 14.3 Residuals in the solution of (14.2) using `bvp4c` with default tolerances. By plotting this, we see that we may have cause for worry in that the residual is surprisingly large at the right end of the interval

```
dsolve( {diff(y(x),x,x)-9*diff(y(x),x)-10*y(x),
          y(0)=a, y(L)=b}, y(x) ).
```

By inspection, we can see that at $a = b = 1$ and $L = 10$,

$$0 \le \frac{\partial y/\partial a}{y} \le 1$$
$$0 \le \frac{\partial y/\partial b}{y} \le 1,$$

and

$$-10 \le \frac{\partial y/\partial L}{y} \le 0\,.$$

It follows that small changes in a near $a = 1$, small changes in b near $b = 1$, and small changes in L near $L = 10$ have very little effect on the solution $y(x)$. That is, in contrast to the IVP with $y(0) = 1, y'(0) = \alpha$, the BVP is well-conditioned.

Notice that it is the boundary conditions alone that differ—the differential equation is the same in both cases. Therefore, a comprehensive theory of conditioning of BVP must take the boundary conditions into account.

The reader many recall, from his or her last course in linear ODE, the theory of Green's functions for solving *in*homogeneous BVPs (see Ascher et al. 1988 for a self-contained description of the theory, presented in a way that makes it easy to consider numerical issues). For this example, if we wished to solve

$$y'' - 9y' - 10y = \Delta(x)$$

subject to $y(0) = a$, $y(L) = b$ (by hand if we remember the theory, because while MAPLE will indeed solve the problem using

```
dsolve( {diff(y(x),x,x)-9*diff(y(x),x)-10*y(x)=Delta(t),
          y(0)=a, y(L)=b }, y(x) ),
```

its answer is complicated and hard to read), then we find

$$y(x) = y_0(x) + \int_0^L G(x,t)\Delta(t)dt\,, \tag{14.4}$$

where $G(x,t)$ is the Green's function for this problem. Note that $y_0(x)$ is the homogeneous solution from Eq. (14.3). The exact expression for $G(x,t)$ can be disentangled from the MAPLE output or, more easily, derived directly by hand using the standard theory. Either way, what concerns us about the conditioning of the BVP is the size of $G(x,t)$: By how much can it amplify $\Delta(x)$? The specific details of $G(x,t)$ are left to Problem 14.3.

Example 14.3. For further exposition here, consider a slightly simpler problem, Airy's differential equation

$$y'' - zy = 0 \tag{14.5}$$

but subject to the boundary conditions

$$y(0) = 1, \quad y(L) = 0\,. \tag{14.6}$$

Since $\mathrm{Ai}(z)$ decays rapidly, we expect $y(z) \doteq {}^{\mathrm{Ai}(z)}/_{\mathrm{Ai}(0)}$ and indeed we will see that

$$y(z) = \frac{\mathrm{Ai}(z)}{\mathrm{Ai}(0)} + O(\mathrm{Ai}(L))\,.$$

We could similarly consider the initial-value problem

$$v'' - zv = 0\,,$$

with the boundary conditions

$$v(0) = 1, \quad v'(0) = \frac{\mathrm{Ai}'(0)}{\mathrm{Ai}(0)} + \varepsilon$$

(which is approximately the associated IVP to the given BVP) and similarly expect

$$v(z) = \frac{\mathrm{Ai}(z)}{\mathrm{Ai}(0)} + O(\varepsilon)\,,$$

an expectation that would be quite true as we will see. Nonetheless, $y(z) - v(z)$ gets very large if L is large. This seems contradictory, but this apparent contradiction will be resolved algebraically below and the resolution highlights the difference between BVPs and IVPs.

Let us solve Eq. (14.5) subject to the boundary conditions (14.6). We have

$$y(z) = \alpha \mathrm{Ai}(z) + \beta \mathrm{Bi}(z)$$

for some α, β. Remember that

$$\mathrm{Ai}(z) \sim \frac{\exp(-2z^{3/2}/3)}{2\sqrt{\pi\sqrt{z}}}$$

decays very rapidly, while in contrast

$$\mathrm{Bi}(z) \sim \frac{\exp(+2z^{3/2}/3)}{2\sqrt{\pi\sqrt{z}}}$$

grows faster than exponentially. Our boundary conditions are

$$\begin{bmatrix} \mathrm{Ai}(0) & \mathrm{Bi}(0) \\ \mathrm{Ai}(L) & \mathrm{Bi}(L) \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}\,,$$

and since $|\mathrm{Ai}(L)| \ll |\mathrm{Bi}(L)|$, the last equation gives

$$\mathrm{Ai}(0)\alpha - \frac{\mathrm{Ai}(L)}{\mathrm{Bi}(L)}\mathrm{Bi}(0)\alpha = 1\,,$$

or, if we rearrange the terms,

$$\alpha = \frac{1}{\mathrm{Ai}(0) - \mathrm{Bi}(0)\dfrac{\mathrm{Ai}(L)}{\mathrm{Bi}(L)}} = \frac{1}{\mathrm{Ai}(0)}\left(\frac{1}{1-r}\right) = \frac{1}{\mathrm{Ai}(0)} + \frac{r}{\mathrm{Ai}(0)} + O(r^2)\,,$$

where

$$r = \frac{\mathrm{Bi}(0)\mathrm{Ai}(L)}{\mathrm{Ai}(0)\mathrm{Bi}(L)} \sim c\exp(-\frac{4}{3}L^{3/2})$$

is very small indeed. As a result, $y(z) = \alpha\mathrm{Ai}(z) + \beta\mathrm{Bi}(z)$ gives

$$y(z) = \left(\frac{1}{\mathrm{Ai}(0)} + \frac{r}{\mathrm{Ai}(0)}\right)\mathrm{Ai}(z) + \frac{\mathrm{Ai}(L)}{\mathrm{Bi}(L)}\frac{\mathrm{Bi}(z)}{\mathrm{Ai}(0)} + O(r^2)\,, \tag{14.7}$$

which is very close to $\mathrm{Ai}(z)/\mathrm{Ai}(0)$, differing only by terms of size r, or at worst

$$r\frac{\mathrm{Bi}(z)}{\mathrm{Bi}(0)} \sim O(\exp(-\frac{2}{3}L^{3/2}))\,.$$

Now, consider the (approximate) associated initial-value problem, with $v'(0) = \mathrm{Ai}'(0)/\mathrm{Ai}(0) + \varepsilon$. By straightforward MAPLE, we find that

$$v(z) = \frac{\mathrm{Ai}(z)}{\mathrm{Ai}(0)} - \frac{\pi}{3\Gamma(2/3)}\left(3^{1/3}\mathrm{Bi}(z) - 3^{5/6}\mathrm{Ai}(z)\right)\varepsilon = \frac{\mathrm{Ai}(z)}{\mathrm{Ai}(0)} + O(\varepsilon),$$

also as claimed. But there is a significant difference: For $y(z)$, the coefficient of the small term contains $1/\mathrm{Bi}(L)$, which is very small indeed; it is so small that when $x = L$, it balances out the growth of the terrible $\mathrm{Bi}(z)$ term, leaving only an "error" of size $\mathrm{Ai}(L)$ (that this is 100% is beside the point: $v(z)$ will be much worse).

One hundred percent error seems pretty bad, but it is exactly the error we make in saying $\mathrm{Ai}(L) \approx 0$ in the first place. For $v(z)$, on the other hand, the relative error is much worse:

$$\begin{aligned}\frac{v(z)}{\mathrm{Ai}(z)/\mathrm{Ai}(0)} - 1 &= \frac{\dfrac{\pi}{3\Gamma(2/3)}\left(3^{1/3}\mathrm{Bi}(z) - 3^{5/6}\mathrm{Ai}(z)\right)\varepsilon}{\mathrm{Ai}(z)/\mathrm{Ai}(0)} \\ &= \frac{\pi}{3^{2/3}\Gamma(2/3)}\mathrm{Ai}(0)\frac{\mathrm{Bi}(z)}{\mathrm{Ai}(z)}\varepsilon - \text{a smaller term}\,.\end{aligned}$$

This term grows as $O(\exp(\frac{4}{3}z^{3/2}))$. Already for $z = 4$, this is $35{,}000\varepsilon$; for $z = 5$, it is $2.4 \cdot 10^6\varepsilon$, while for $z = 10$, it is $1.6 \cdot 10^{18}\varepsilon$. What does this say about ε? It says that to get relative error merely 1 (i.e., 100%) at $z = 10$, we have to take $\varepsilon \doteq 5 \cdot 10^{-19}$.

In contrast, for y, when we insist $y(10) = 0$, we are making an error of size $\mathrm{Ai}(10)/\mathrm{Ai}(0) \doteq 3 \cdot 10^{-10}$. Thus, in some sense the BVP $y'' - zy = 0$ with the boundary conditions $y(0) = 1$ and $y(10) = 0$ is 10^8 times better conditioned in $0 \leq z \leq 10$ than the IVP $y'' - zy = 0$ with initial conditions $y(0) = 1$ and $y'(0) = \mathrm{Ai}'(0)/\mathrm{Ai}(0)$, which we naively thought at first was almost equivalent.

What is happening in this example is that the condition at the right endpoint is controlling the faster-than-exponential growth of the $\mathrm{Bi}(z)$ solution, while the condition at the left controls the decaying $\mathrm{Ai}(z)$ term. When classes of solutions of the underlying differential equation exhibit this split into growing and decaying terms, the DE is said to have a *dichotomy*. ◁

There is a great deal more to be said about the conditioning of BVPs. One didactic point that we have not seen stated elsewhere is that it is really with this numerical problem that the ideas of numerical stability of the method compared and contrasted with the well-conditioning of the problem start to sink in.

Finally, the full theory of conditioning, of course, needs to consider perturbations of the equation, such as (but not limited to) those introduced by the residual:

$$y'' - zy = \Delta(z)\,.$$

For linear problems, this brings us directly to computation of Green's function, where

$$y(z) = y_0(z) + \int_a^b G(z,\tau)\Delta(\tau)d\tau\,,$$

and a full development would take too much space. But given this, we emphasize two important points: BVPs have condition numbers that can be computed, and the boundary conditions really matter here. In particular, the naturally associated IVP can have a very different conditioning (as in both these examples).

14.3 A Method to Solve BVPODE

Let us leave the MATLAB codes behind for now, and consider constructing our own method to solve this problem and other ones like it (for didactic purposes only—of course, in practice, one would use the high-quality codes available, rather than rolling one's own). The didactic purpose is, of course, to learn how these methods work, so that we may use them better, and understand their limitations. The key ideas that we want to exhibit are *equidistribution* and *mesh adaptation*, *error estimates from the residual*, and *collocation*.

Before we begin, though, we talk briefly about an alternative approach, known as "shooting," suitable for some timelike problems. This example BVP *could* be rephrased as an initial-value problem, with a slope $y'(0) = \alpha$ to be determined in such a way that an IVP solver would march off from $t = 0$ and arrive at $t = 10$

with $y(10)$ just managing to hit the value 1: That is, we choose an angle α at which to "shoot" the IVP solver to try to hit the "target" $y(10) = 1$. This IVP is called the "associated IVP" to the BVP. One could imagine using some kind of rootfinder (perhaps Newton's method) to find a value of α for which this works. This approach is called "simple shooting"; while easy to describe and implement (and it has some theoretical advantages in discussing existence and uniqueness), it's not so easy to make it work, especially for problems like the one chosen here. In the exercises, you will be asked to show that this particular IVP is ill-conditioned, and as a general rule it is not recommended to replace a well-conditioned problem (as this BVP is) with an ill-conditioned problem, as a step toward the solution.

Instead we demonstrate a popular direct method (one among many popular methods, actually) that is well-suited to equations of this type, namely, *collocation*. The basic idea of collocation is that we approximate the solution in a finite-dimensional space and *require the differential equation to be satisfied exactly at a finite number of points*, and use those conditions to determine the coefficients of the approximate solution in that finite-dimensional space.

We change variables to x from now on, to emphasize the spacelike nature of the problem. We begin with a mesh

$$0 = x_0 < x_1 < x_2 < \ldots < x_{N-1} < x_N = 10 \tag{14.8}$$

that divides our interval into N subintervals, each of length $h_i = x_i - x_{i-1} > 0$, $i = 1, 2, \ldots, N$. Often (and for this problem) we begin with a uniform mesh, $h_i = {}^{(b-a)}/_{N} = {}^{10}/_{N}$. We choose to represent our solution by a piecewise polynomial of degree less than or equal to m; in the didactic procedure we discuss below, we use $m = 3$, generically giving cubic polynomials on each subinterval:

$$z_i(x) \qquad \text{on} \qquad x_{i-1} \le x \le x_i\,. \tag{14.9}$$

Thus, the finite-dimensional space in which we search for our approximate solution is the set of piecewise polynomials of degree at most 3 on the given mesh. We want $z(x)$ (the union of all these pieces) to be continuous, and even continuously differentiable, at $x = x_i$, $1 \le i \le N-1$. With only degree-3 polynomials, we cannot insist that $z(x)$ be twice continuously differentiable at the nodes. We also want $z(x_0) = b_0 = 1$ and $z(x_N) = b_1 = 1$, in order to match the given boundary conditions.

It is particularly convenient to use a Hermite interpolational representation for $z_i(x)$, with $z_i(x_{i-1}) = y_{i-1}$, $z_i'(x_{i-1}) = y_{i-1}'$, $z_i(x_i) = y_i$, and $z_i'(x_i) = y_i'$, because this automatically ensures that $z(x)$ is not only continuous, but also continuously differentiable. Thus, if $\theta = {}^{(x-x_{i-1})}/_{h_i}$, we may construct $z_i(x)$ from this data and from the partial fraction expansion of ${}^{1}/_{\theta^2(\theta-1)^2}$, as we did in Chap. 8. This gives

$$\begin{aligned} z(x) =& \frac{1}{h_i^3}(2x - 3x_{i-1} + x_i)(x - x_i)^2 y_{i-1} + \frac{1}{h_i^2}(x - x_{i-1})(x - x_i)^2 y_{i-1}' \\ &+ \frac{1}{h_i^2}(x - x_i)(x - x_{i-1})^2 y_i' + \frac{1}{h_i^3}(3x_i + x_{i-1} - 2x)(x - x_{i-1})^2 y_i\,. \end{aligned} \tag{14.10}$$

It is easy to find formulæ for $z_i'(x)$ and $z_i''(x)$ in this form:

$$\begin{aligned} z_i'(x) = {} & 6\,\frac{(x-x_i)\,(x-x_{i-1})\,y_{i-1}}{{h_i}^3} + \frac{(x-x_i)\,(3x-2x_{i-1}-x_i)\,{y'}_{i-1}}{{h_i}^2} \\ & -6\,\frac{(x-x_i)\,(x-x_{i-1})\,y_i}{{h_i}^3} + \frac{(x-x_{i-1})\,(3x-2x_i-x_{i-1})\,{y'}_i}{{h_i}^2} \end{aligned} \tag{14.11}$$

$$\begin{aligned} z_i''(x) = {} & 6\,\frac{(2x-x_{i-1}-x_i)\,y_{i-1}}{{h_i}^3} + 2\,\frac{(3x-2x_i-x_{i-1})\,{y'}_{i-1}}{{h_i}^2} \\ & -6\,\frac{(2x-x_{i-1}-x_i)\,y_i}{{h_i}^3} + 2\,\frac{(3x-2x_{i-1}-x_i)\,{y'}_i}{{h_i}^2}. \end{aligned} \tag{14.12}$$

Since the DE we are looking at is second order, we will need both those formulæ. If we had been trying to solve a third-order BVP, we would have needed to start with higher-degree polynomials and take a third derivative, or else rewrite the equation as a first-order system. We continue the present example of a scalar second-order DE and will take up that question later.

We have one unknown, y_0', at the left end, two unknowns y_i and y_i' at each interior node $1 \le i \le N-1$, and one unknown y_N' at the right end. This gives $2N$ unknowns in all. We are working directly in the Hermite interpolational basis: The piecewise polynomials are known completely once y_{i-1}, y_{i-1}', y_i, and y_{i-1}' are specified, and the total piecewise polynomial $z(x)$ is automatically continuously differentiable, because the derivatives at the left end of the interval are the same as the derivatives of the right end of the next-door interval if there is one. If we had been working instead in the monomial basis, the process would have been (slightly) less simple, because we would have had to explicitly add continuity and differentiability conditions to our equations: Each cubic polynomial in the monomial basis has four coefficients, leading to $4N$ unknowns, and we would have had to explicitly make things match up. This is what COLSYS and other collocation codes that use the monomial basis do, for example. Here, because we have chosen to use the Hermite interpolational basis, we automatically have the right level of continuity. The price we pay is the slightly more complicated formulae needed for the derivatives of the piecewise polynomials given in the Hermite interpolational basis.

Now we come to the idea of collocation. We require that the differential equations be satisfied exactly, at two discrete points in each subinterval. This means that we will have $2N$ equations for our $2N$ unknowns. Because the differential equation is *linear*, the equations will also be linear. Nonlinear differential equations will indeed generate nonlinear algebraic equations for the unknowns, and we will have to deal with this; for now, we continue with this linear example problem.

Consider the residual

$$r_i(x) = z_i''(x) - 9z_i'(x) - 10z_i(x) \tag{14.13}$$

(for a more complicated DE, this would be a lengthier formula, but otherwise the same idea). We will try to make the residual small *everywhere*, by making it exactly zero at two points, called collocation points, in the interval:

$$x_1 = x_{i-1} + \theta h_i \tag{14.14}$$
$$x_2 = x_{i-1} + \theta_2 h_i . \tag{14.15}$$

We choose $\theta_1 = 1/2 - 1/(2\sqrt{3})$ and $\theta_2 = 1/2 + 1/(2\sqrt{3})$, which are Gaussian quadrature nodes; we discuss other choices later. Thus,

$$r(x_1) = z_i''(x_1) - 9z_i'(x_1) - 10z_i(x_1) = 0 \tag{14.16}$$
$$r(x_2) = z_i''(x_2) - 9z_i'(x_2) - 10z_i(x_2) = 0 \tag{14.17}$$

for $1 \le i \le N$. Because we make the residual zero at two places in each subinterval, we expect (naively) that the method will be at most second-order accurate, that is, have residual $O(h^2)$ as $h \to 0$. We will look more closely at this naive expectation later.

Remark 14.1. If we had instead used higher-degree polynomial pieces in order to get higher-order convergence, say quintic Hermite interpolation, then we would have to specify y_i, y_i', and y_i'' at each mesh point (which would automatically guarantee a higher degree of continuity, too). Then we would have had $2+3(N-1)+2 = 3N+1$ unknowns and would have needed 3 collocation points per subinterval. This is enough, but it leaves us with one free parameter. With y_i, y_i', y_i'', and y_i''', we would get $2+4(N-1)+3 = 4N+2$ unknowns, leaving 2 free parameters. The algebra would get a little messier with such higher degrees but is not, in principle, more difficult.

However, we remind you that the purpose of this section of this chapter is not to encourage you to write your own code: Very few people have to do this. In MATLAB, there are `bvp4c` and `bvp5c`; in Fortran, there are COLNEW and MIRKDC, and a host of other codes. This example is not about solving BVPs, but rather about ideas. Here, 2 collocation points per subinterval give us enough equations to identify the unknowns in our piecewise cubic polynomial approximate solution. ◁

Suppose we take a very coarse mesh with $N = 5$ (that is, 5 subintervals). This gives us 10 equations, which we denote schematically below:

$$\begin{bmatrix}
x & x & x & x & & & & & & & & \\
x & x & x & x & & & & & & & & \\
 & & x & x & x & x & & & & & & \\
 & & x & x & x & x & & & & & & \\
 & & & & x & x & x & x & & & & \\
 & & & & x & x & x & x & & & & \\
 & & & & & & x & x & x & x & & \\
 & & & & & & x & x & x & x & & \\
 & & & & & & & & x & x & x & x \\
 & & & & & & & & x & x & x & x
\end{bmatrix}
\begin{bmatrix} 1 \\ y_0' \\ y_1 \\ y_1' \\ y_2 \\ y_2' \\ y_3 \\ y_3' \\ y_4 \\ y_4' \\ 1 \\ y_5' \end{bmatrix} = 0. \tag{14.18}$$

The two dashed columns multiply known solution values. This is an almost-block-diagonal (ABD) matrix. There are N blocks (here 5), each of two equations. These can be solved efficiently and stably, with $O(N)$ work; a block LU factoring is usual. Notice that careful bookkeeping is needed to use the boundary conditions in this formulation. Alternatively, linear boundary conditions can be applied as extra blocks:

$$\begin{bmatrix}
1 & 0 & & & & & & & & & & \\
x & x & x & x & & & & & & & & \\
x & x & x & x & & & & & & & & \\
 & & x & x & x & x & & & & & & \\
 & & x & x & x & x & & & & & & \\
 & & & & x & x & x & x & & & & \\
 & & & & x & x & x & x & & & & \\
 & & & & & & x & x & x & x & & \\
 & & & & & & x & x & x & x & & \\
 & & & & & & & & x & x & x & x \\
 & & & & & & & & x & x & x & x \\
 & & & & & & & & & & 1 & 0
\end{bmatrix}
\begin{bmatrix} y_0 \\ y_0' \\ \vdots \\ y_N \\ y_N' \end{bmatrix}
=
\begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} . \tag{14.19}$$

Periodic boundary conditions would complicate the block structure. In practice, we can use any solver we like; for large problems, there are specialized solvers that are accurate and efficient. Here, using the sparse matrix facilities in MATLAB is particularly convenient. In what follows, we do so. This is the key to the solution.

We have replaced a linear boundary value problem for ordinary differential equations with a system of $2N$ linear algebraic equations. The solution of this system gives us the values of y_i and y_i' at the mesh points. Using those values, we can then compute the values of a piecewise cubic Hermite interpolant at any point; differentiate it, and again, and evaluate the residual again at any point.

With $N = 5$, there are 42 nonzero entries in the matrix of the linear system, not so different than the $(2N)^2 = 100$ of a full matrix. However, the solution is quite poor, with only $N = 5$ subintervals; we need at least 12 to get visual accuracy. With $N = 12$, the number of nonzeros in the matrix is 98, compared to 576 in a 24×24 full matrix; using $N = 33$ (as used by `bvp5c`) gets us a very nice and accurate solution (not with a uniform mesh as here, but with a better mesh that we discuss later—the solution is still fairly poor with a uniform mesh, as we will see), but when $N = 33$ (uniform or nonuniform), the matrix has 266 nonzeros, versus 4356 in a full matrix. The sparsity and structure of this matrix are quite important.

Another important thing is the condition number of the matrix. For the uniform mesh we have presented so far, the condition number seems to scale as $\kappa(\mathbf{A}) = O(N^2)$; this may become an issue with this method, and COLSYS in particular pays close attention to the conditioning of the linear systems that arise, and in part the monomial basis was chosen to keep the condition number under control.

14.3.1 Solution on a Uniform Mesh

Let us look more closely at the solution on a uniform mesh. In Fig. 14.4, we see the residual samples at $x_1, x_{i+1/2}, x_{i+1}$ for $0 \le i \le N-1$, and $\sqrt{\Delta_i^2 + \Delta_{i+1/2}^2 + \Delta_{i+1}^2}$ is plotted; this is for $N = 50$ and $h = {}^{10}/_{N}$. The residual is less than 10^{-2} for $x < 9$, approximately, but becomes larger than 1 (even larger than 10) as $x \to 10$. This means that we have solved a grossly different DE than the one we had intended to solve: $z'' - 9z' - 10z = 10v(x)$, where now $v(x)$ is about 1. It would be expected that this solution would not be acceptable for most purposes. To contrast with the traditional view, we look at the forward error as well, which is possible in this case because the reference equation is linear and homogeneous and is possible to solve analytically:

$$y_{\text{reference}}(x) = \frac{\left(\mathrm{e}^{100} - 1\right)\mathrm{e}^{-x}}{\mathrm{e}^{100} - \mathrm{e}^{-10}} + \frac{\left(1 - \mathrm{e}^{-10}\right)\mathrm{e}^{10x}}{\mathrm{e}^{100} - \mathrm{e}.^{-10}} \tag{14.20}$$

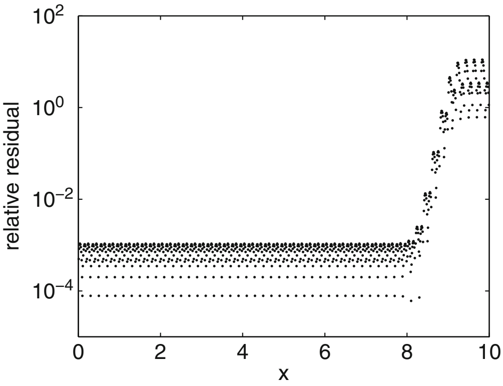


Fig. 14.4 Residual of the solution of $y'' - 9y' - 10y = 0$, $y(0) = y(10) = 1$, divided by y, with $N = 50$ uniform subintervals in the mesh, cubic Hermite interpolants, and collocation at Gaussian points in each subinterval. The residual is unacceptably large at the right end of the interval

In Fig. 14.5, the forward error $z(x) - y_{\text{reference}}(x)$ is plotted and is quite a bit smaller, but it is still huge at the right end of the interval.

In Fig. 14.6, we see the computed solution at the mesh points, plotted together with the true solution. You see that there are only two points in the layer at the right. In Fig. 14.7, we see a graph of $z'(x)$ at the mesh points; again, the right end layer is not well captured. The residual is estimated to be large there.

14.3.2 Solution on an Adapted Mesh

Now let us do the same thing again, but on an adapted (nonuniform) mesh. Somehow, we redistribute the x_k so that more of them are in the boundary layer near the right. We will shortly discuss how to find this adapted mesh. Figure 14.8 shows that,

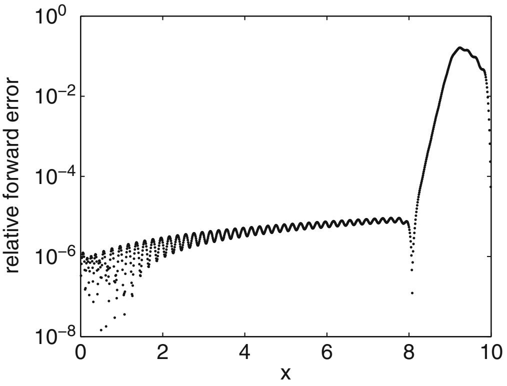


Fig. 14.5 Relative forward error $y/y_{\text{reference}} - 1$ in the solution of $y'' - 9y' - 10y = 0$, $y(0) = y(10) = 1$, with $N = 50$ uniform subintervals in the mesh, cubic Hermite interpolants, and collocation at Gaussian points in each subinterval. We see that the forward error is also unacceptably large at the right end of the interval, although much smaller than the residual because this problem is actually very well-conditioned

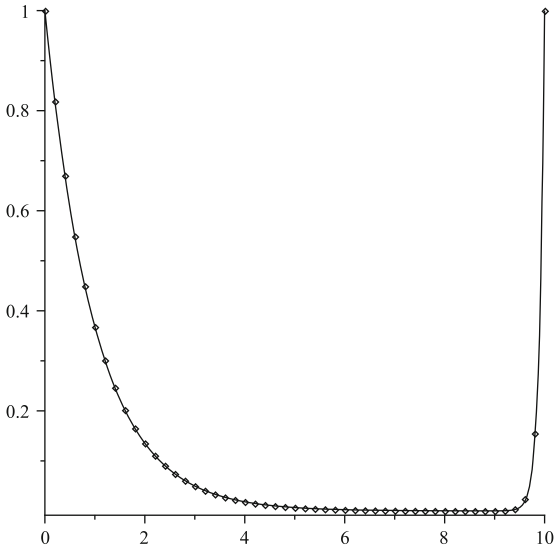


Fig. 14.6 Solution computed on a uniform mesh together with the exact solution. We see visual accuracy, but the number of points in the layer at the right seems low

after adapting the mesh to the problem, we find our residual to be nearly constant: $\|\Delta\| \doteq 0.03$, which suggests $O(\overline{h}^2)$ with a moderate constant.

In Fig. 14.9, we display a graph of the lengths of the adapted subintervals: Note that the widest subintervals have widths greater than 1, while the narrowest subintervals have width about 0.005. Subintervals of width greater than 10% of the overall

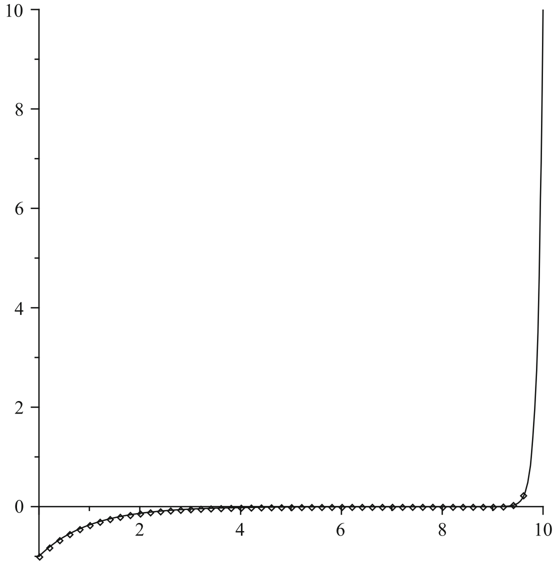


Fig. 14.7 $z'(x)$ at the mesh points plotted with the derivative of the exact solution; again, the right end layer is not well captured

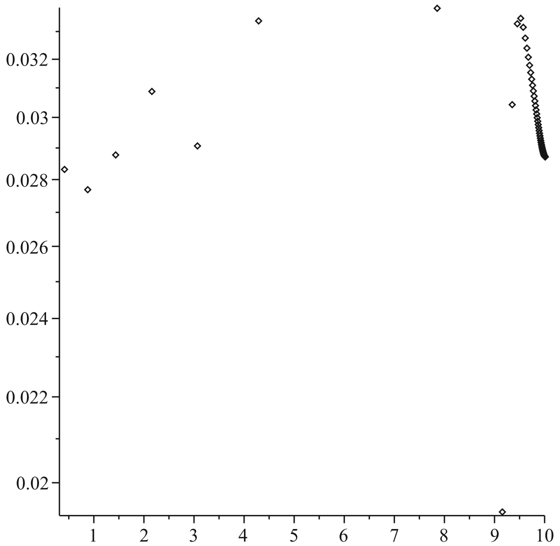


Fig. 14.8 Residual on an adapted nonuniform mesh; note the scale on the vertical axis, which shows that the variation in the residual is about 10%, except for one outlier where the residual is about 50% smaller

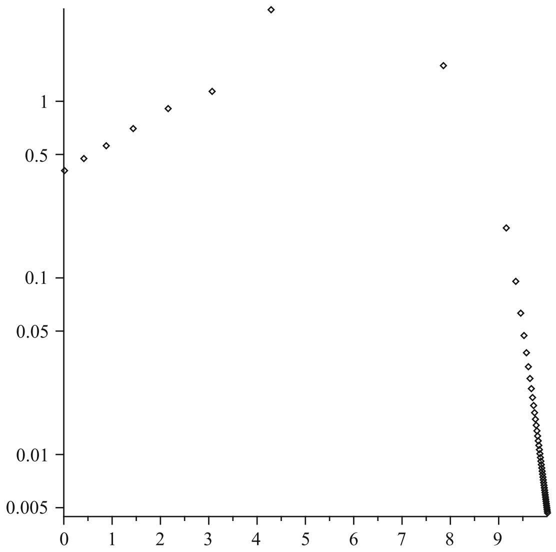


Fig. 14.9 Widths of an adapted mesh. Note that in the right-hand layer, which caused difficulty for a uniform mesh, we now have many small subintervals

interval are probably excessive in practice: This heuristically adapted mesh isn't really a very good one, although we see that it seems to have lowered the maximum residual and placed more mesh points in the layer at the right, where there is rapid change in the solution. The key figure is Fig. 14.8, which shows that $\|\Delta(x)\| \leq 0.033$ for $N = 50$ subintervals, whereas $\|\Delta(x)\|$ was at least 30 times larger for the uniform mesh. This is a substantial improvement. In the next section, we examine how this mesh redistribution is achieved, so that we obtain substantial improvement in the residual.

14.4 How Does It Work, and Why Does It Work?

The key idea is equidistribution of error. Here, by error we mean residual, but other authors use other measures of error, and indeed, it is an interesting part of a numerical analyst's purview to decide, in a specific problem context, what is the most meaningful choice of error. The technique of equidistribution will work for any choice, so long as the "error" can be measured reliably. Here, the residual on each subinterval is

$$\Delta_i(x) = z_i''(x) - 9z_i'(x) - 10z_i(x)\,, \qquad x_{i-1} \leq x \leq x_i\,, \tag{14.21}$$

which is (as always) computable and has a direct interpretation in that our computed $z(x)$ satisfies

$$z'' - 9z' - 10z = \Delta(x) = \begin{cases} \Delta_1(x) & x_0 \le x \le x_1 \\ \Delta_2(x) & x_1 \le x \le x_2 \\ \vdots & \vdots \end{cases}, \tag{14.22}$$

with some ambiguity at the nodes if $\Delta(x)$ is not continuous enough, which it won't be here (we would need to match second derivatives, in this formulation, which we haven't). As usual, if we use a residual as a measure of error, we must worry about the effects of such perturbations. We return to the conditioning of BVPs later.

We now consider how to measure $\Delta(x)$. One way is just to sample it with

$$\Delta_i(x_{i-1} + \theta_1 h_i) = 0 \tag{14.23}$$

$$\Delta_i(x_{i-1} + \theta_2 h_i) = 0\,. \tag{14.24}$$

It gives us two samples per subinterval. $\Delta_i(x_{i-1})$ and $\Delta_i(x_i)$ are two more, and $\Delta(x_{i-1/2})$ might also be interesting. For the purpose of this book, that is all we will do: Sample the residual at various places (arbitrarily we choose five: at $\theta = 0$, $\theta = 1/2$, $\theta = 1$, and the two collocation points, where, by definition, $\Delta_i(x) = 0$).

In Figs. 14.10–14.12, we see the residual, as sampled by the scheme given above, in each of three subintervals of our *adapted mesh*, each divided by $\overline{h}^2 = (1/50)^2$, that is, divided by the mean step size squared. All three are comparable in size. This is not an accident, as we will see in the next subsection.

14.4.1 Equidistribution, the Theory

The topic of equidistribution is treated fairly widely in the literature.[1] For theoretical reasons, we can expect the residual on a subinterval of small width h_i to be of size roughly

$$\|\Delta_i\| = \phi_i h_i^2\,, \tag{14.25}$$

where the ϕ_i are $O(1)$ and depend only weakly on the distribution of the mesh points. That is, we expect this collocation method to produce a *second-order* accurate solution: Each $\|\Delta_i\| = O(h_i^2)$, and thus $\|\Delta\| = O(h_{\max}^2)$, and thus, finally,

$$\|y - z\| = O(h_{\max}^2)\,. \tag{14.26}$$

The occurrence of $h_{\max}^2$, the square of the maximum h_i, is troubling, and indeed $h_{\max} > 1$ on our example nearly optimal mesh! Therefore, by this estimate, we shouldn't expect any accuracy at all—yet we do get it, so something is not quite right with our expectations. The plots earlier were scaled by $\overline{h}^2$, the square of the *mean* step size, not the maximum step size.

[1] We recommend Butcher (2008b) and Ascher et al. (1988) in particular. Much of the treatment here follows Corless (2000).

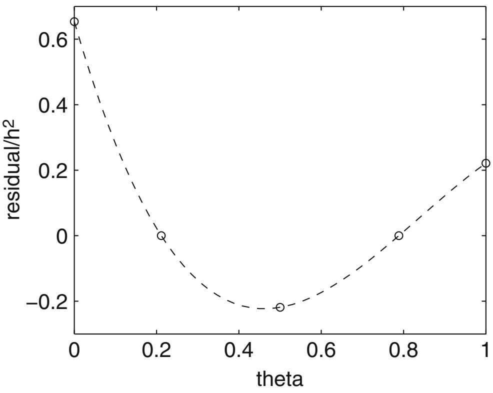


Fig. 14.10 Residual divided by $\overline{h}^2$, on $x_0 \le x \le x_1$ (the first subinterval)

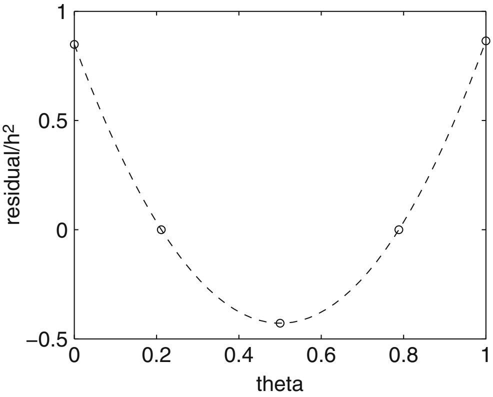


Fig. 14.11 Residual divided by $\overline{h}^2$, on $x_{24} \le x \le x_{25}$ (a subinterval in the middle)

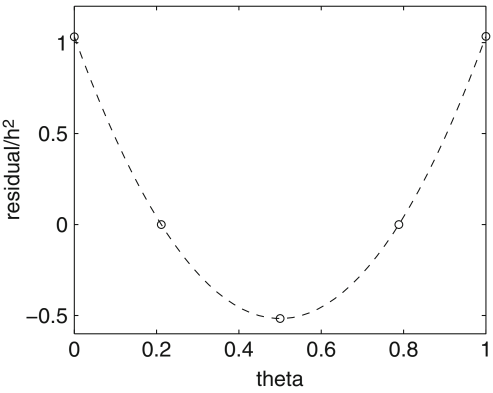


Fig. 14.12 Residual divided by $\overline{h}^2$, on $x_{49} \le x \le x_{50}$ (the last subinterval)

In order to explain why this will always be possible on an equidistributed mesh, we need to look a bit more carefully at the idea that each $\|\Delta_i\| \doteq \phi_i h_i^2$ and the meaning of the error coefficient ϕ_i. Recall that such power-law formulæ are common in interpolation problems:

$$f(t) - p(t) = w(t)\frac{f^{(n+1)}(\theta)}{(n+1)!} \doteq Kh^{n+1} \tag{14.27}$$

if the width of the interpolation interval is h. Similarly, finite differences (found by interpolation), quadrature, and Taylor series also have such power-law error formulae. Indeed, the only time this didn't help was with the spectrally accurate trapezoidal rule for quadrature of a periodic $\mathscr{C}^\infty$ function, where all the Taylor terms were zero. So it is to be expected that

$$\|\Delta_i\| = \phi_i h_i^p \tag{14.28}$$

will hold for some p, namely, the order of the method, and some vector of coefficients ϕ_i, $1 \le i \le N$, which we suppose nonnegative.

The mesh adaption problem is to find subinterval widths h_i such that $h_i > 0$ and $\sum_N h_i = b - a$, that is, they cover the width of the full interval, and with the property that

$$\max_{1\le i\le N} \|\Delta_i\| = \max_{1\le i\le N} \phi_i h^p \tag{14.29}$$

is as small as possible. That is, we wish to find the partition H of $[a,b]$ that has

$$\min \max_{1\le i\le N} \phi_i h_i^p \,. \tag{14.30}$$

Theorem 14.1 below allows us to do that.

First, remember that the r-norm of a vector $\phi = (\phi_1, \phi_2, \ldots, \phi_N)$ of positive reals is

$$\|\phi\|_r := \left(\sum_{i=1}^N \phi_i^r\right)^{1/r} \,.$$

Moreover, the Hölder r-mean of ϕ is

$$\mathscr{M}_r(\phi) := \left(\frac{1}{N}\sum_{i=1}^N \phi_i^r\right)^{1/r} \,.$$

These being given, we first state two lemmas that will lead us to the key theorem.

Lemma 14.1. *If* $\theta > 0$*, then* $\theta^p \|\phi\|_{-1/p} = \left(\dfrac{\theta}{N}\right)^p \mathscr{M}_{-1/p}(\phi)\,.$

Proof. Observe that

$$\begin{aligned}\theta^p\|\phi\|_{-1/p} &= \left(\frac{\theta}{N}\right)^p N^p\|\phi\|_{-1/p}\\ &= \left(\frac{\theta}{N}\right)^p N^p\left(\sum_{i=1}^{N}\phi_i^{-1/p}\right)^{-p} = \left(\frac{\theta}{N}\right)^p\left(\frac{1}{N}\sum_{i=1}^{N}\phi_i^{-1/p}\right)^{-p}\\ &= \left(\frac{\theta}{N}\right)^p \mathscr{M}_{-1/p}(\phi),\end{aligned}$$

as desired. $\sqcap\!\!\!\!\sqcup$

Lemma 14.2. $\big\|\,(\,\|(\phi_1,\phi_2,\dots,\phi_s)\|_r,\ \phi_{s+1})\,\big\|_r = \|(\phi_1,\phi_2,\dots,\phi_{s+1})\|_r$

Proof. Observe that

$$\begin{aligned}\left\|\left(\sum_{j=1}^{s}\phi_j^r\right)^{1/r},\phi_s\right\|_r &= \left(\left[\left(\sum_{j=1}^{s}\phi_j^r\right)^{1/r}\right]^r+\phi_s^r\right)^{1/r}\\ &= \left(\sum_{j=1}^{s+1}\phi_j^r\right)^{1/r} = \big\|(\phi_1,\dots,\phi_{s+1})\big\|_r\end{aligned}$$

as desired. $\sqcap\!\!\!\!\sqcup$

We can now state the theorem that will allow us to handle the minimax problem at the core of equidistribution of error for mesh adaptation.

Theorem 14.1. *If* $p,\ N\in\mathbb{N}$, $\phi_i>0$ *for* $1\le i\le N$, *and* $\theta>0$, *then*

$$\max\left\{\phi_i h_i^p : \sum h_i=\theta\right\} \ \ge\ \theta^p\|\phi\|_{-1/p} \ =\ \overline{h}^p\mathscr{M}_{-1/p}(\phi),$$

where $\overline{h}=\theta/N$, *with equality iff each* $\phi_i\ h_i^p=\mathscr{M}_{-1/p}(\phi)\overline{h}^p$.

Proof. Hölder's inequality[2] gives

$$[\mathscr{M}_1(h)]^p=\mathscr{M}_{1/p}(h^p)\le\mathscr{M}_{1/p\alpha}(1/\phi)\mathscr{M}_{1/p\beta}(\phi h^p)$$

if $\alpha+\beta=1$, with equality iff $(1/\phi_i)^{1/\alpha}$ is proportional to $(\phi_i h_i^p)^{1/\beta}$ for all i (i.e., as vectors). This equality condition can be written as $\phi_i^{1/\alpha}h_i^p=\lambda$ (the same constant, which depends on β and hence α). We rewrite the inequality using $\mathscr{M}_{-r}(a)=[\mathscr{M}_r(1/a)]^{-1}$ as

$$\begin{aligned}\mathscr{M}_{1/p\beta}(\phi h^p) &\ge [\mathscr{M}_1(h)]^p\left[\mathscr{M}_{1/p\alpha}(1/\phi)\right]^{-1}\\ &\ge \overline{h}^p\mathscr{M}_{-1/p\alpha}(\phi)\end{aligned}$$

[2] See Appendix C.

and if we now let $\beta \to 0^+$, so that $\alpha \to 1^-$ and $\mathscr{M}_{1/p\beta}(\phi h^p) \to \mathscr{M}_\infty(\phi h^p) = \max\limits_{1\le i\le N} \phi_i h_i^p$, we get

$$\max_{1\le i\le N} \phi_i h_i^p \ge \overline{h}^p \mathscr{M}_{-1/p}(\phi)$$

with equality iff $\phi_i h_i^p = \overline{h}^p \mathscr{M}_{-1/p}(\phi)$ for all i. ♮

Thanks to this result, we have a guide to equidistributing the error, as we explain in the next subsection.

14.4.2 Solution of the Minimax Problem

If we are asked to solve the minimax problem

$$\min\left\{ \max_{1\le i\le N} \phi_i h_i^p : \sum h_i = b-a \right\}$$

given the values $\phi_i > 0$ for $1 \le i \le N$, the solution is immediate from the inequality of Theorem 14.1. Every choice of h_i different from $\phi_i h_i^p = \mathscr{M}_{-1/p}(\phi)\overline{h}^p$ gives a larger value for the maximum; moreover, the solution given is unique. Note that this use of an inequality to solve a minimax problem is classical, and very much in the spirit of the beautiful book Niven (1981).

Remark 14.2. Notice that $\lambda = \mathscr{M}_{-1/p}(\phi)\overline{h}^p$ can be interpreted as an average ϕ times an arithmetic mean $\overline{h}$ to the p^{th} power, and this itself gives the minimax error $\mathscr{M}_{-1/p}(\phi)\overline{h}^p$. This interpretation of λ is not made in Ascher et al. (1988), but we believe it is a useful one. For example, when someone says a method is "fourth order," it is not always immediately clear what this means on a variable mesh. Some people mean this to say that the error behaves like $Kh_{\max}^4$ for some constant K and $h_{\max}$ being the maximum mesh width. What we have shown with this interpretation of λ is that on an optimal grid, the error behaves like $K\overline{h}^4$, where $\overline{h}$ is the *arithmetic mean* mesh width, and moreover we know that $K = \mathscr{M}_{-1/4}(\phi)$ is an average of the monitor function. Another interesting thing is that it is also equal to $K'h_{\max}^4$, where K' is the value of ϕ at the place where the mesh is widest, and by equidistribution, we have $K' < K$ and indeed the error is the same. One expects that for refinements of equidistributing meshes, the average h would decrease, and thus the error would decrease; it is not always clear that the maximum h decreases, and so to us, this interpretation of λ justifies the terminology "fourth order." We noticed this in the chapter on numerical quadrature already, in Eq. (10.46). ◁

This theorem tells us several useful things. First, the maximum error is minimized when the errors on each subinterval are the same. This "equitability" is very satisfying, and since many IVP codes try to do this—they choose the next step size as large as possible subject to the constraint $\phi_i h_i^p = \varepsilon$, the given tolerance—we see that insofar as step-size-selection algorithms equidistribute the error, they are optimal. Second, the theorem tells us that with a pth-order method, when the error is equidistributed, the error goes to zero with the *mean* stepsize $\overline{h}^p$. This explains some rather puzzling convergence phenomena, where we may have *some* $h_i > 1$ and still have very small error. For boundary value problems, especially for nonlinear boundary value problems, this theorem tells us what to do, ideally.

Unfortunately, it doesn't tell us *how* to do it. Finding an equidistributing mesh can be as hard as solving the problem itself. However, several useful heuristics allow us to approximately solve the problem, and even an "almost" optimal mesh can be a great help. Here, those words of Larry Shampine are particularly relevant:

> You've got to remember that you're trying to solve a differential equation, not find an optimal mesh.

The code that we have written for this chapter uses a simple iteration, with some heuristics limiting changes that are "too great," and is not intended as a production method. We do note that the introduction of equidistribution into a linear problem makes it *nonlinear*, and that some sort of iteration will be necessary.

When we look at the `shockbvp` example in `odeexamples('bvp')` and modify the demo to return the solution as well as plot it and to use a relative tolerance 10^{-9}, we get Fig. 14.13a. If we also execute

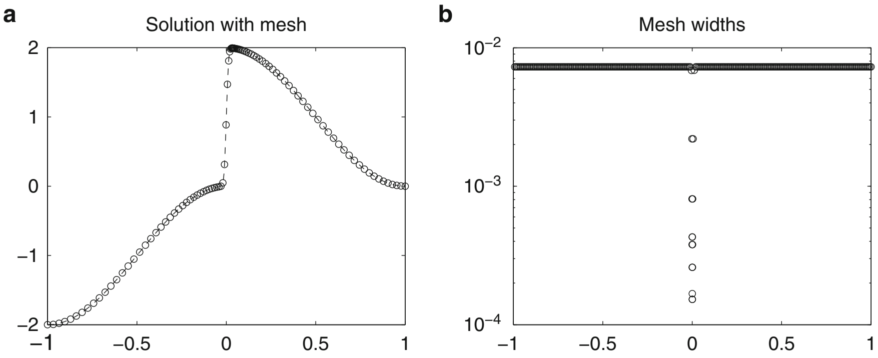


Fig. 14.13 Solution of the `odeexamples[bvp]` shock problem, with tolerance 10^{-9}. Mesh widths plotted on a log scale on the right. (**a**) shockbvpa. (**b**) shockbvpb

```
semilogy( sol.x(2:end), diff(sol.x), 'k.' )
```

we obtain Fig. 14.13b. We see that the mesh widths are four orders of magnitude smaller in the shock layer. The heuristics used to move the mesh have benefitted from continuation in the problem parameter ε; the problem was solved for $\varepsilon = 10^{-2}$ and the mesh and solution for that were used to solve the problem for $\varepsilon = 10^{-3}$, whose solution was then used for $\varepsilon = 10^{-4}$.

The code `bvp4c` uses the residual (it is one of the first widely distributed codes to do so), but it measures $\|\Delta_i(x)\|$ by

$$h_i \left(\int_{x_{i-1}}^{x_i} \Delta_i^2(\xi)\,d\xi \right)^{1/2} , \tag{14.31}$$

that is, in the 2-norm. Hence, the residual from `bvp4c` may contain small spikes, as we saw in Fig. 14.3. Normally, this is perfectly adequate. The code `bvp5c` attempts to estimate the condition number and equidistribute the global error, not the residual; it is sometimes closer to what people expect.

14.5 Superconvergence

With the choice $\theta_1 = 1/2 - \sqrt{3}/6, \theta_2 = 1/2 + \sqrt{3}/6$, and also with $\theta_1 = 1/4, \theta_2 = 3/4$, the collocation method used here is fourth order accurate at the nodes x_k, although the residual of the interpolant obtained by a cubic Hermite fit to the computed data y_k, y'_k is indeed only second order: $\|\Delta_k\| = \phi_k h_k^2$. But for this problem $y''_k = 9y'_k + 10y_k$ is trivial to obtain, so we may easily use piecewise quintic Hermite interpolation on the same data. When we do this, we find that our new Δ_ks have

$$\|\Delta_k\| = \phi_k h_k^4 = \mathscr{M}_{-1/4}(\phi)\overline{h}^4 \tag{14.32}$$

which seems very surprising, a lucky accident. For example, one could plot each panel's defect, divided by $\overline{h}^4$. It will be seen that this is $O(1)$, if all is as it should be, and that $\Delta(x_{i-1}) = \Delta(x_i) = 0$. This is because we now have not just a $\mathscr{C}'$ interpolant, but also $\mathscr{C}^2$, because y_k, y'_k, and y''_k are the same from left and right. This comes at a cost—the polynomial pieces are fifth order—but the extra smoothness may be of interest.

What about equidistribution? We found this mesh by approximately equidistributing $\phi_i h_i^2$ from the ϕ_is given by the second-order interpolant.

It turns out to be true that the relative sizes of the ϕ_i are the proportionate ones for the different powers, often, so

$$\mathscr{M}_{-1/4}(\phi_4) \doteq \mathscr{M}_{-1/2}(\phi_2)\,. \tag{14.33}$$

That is, equidistributing a lower-order estimate of the error will approximately equidistribute a more accurate error estimate, as well. There are many places where this can be taken advantage of.

14.6 Nonlinear Problems and Quasilinearization

Suppose now that the BVP we are trying to solve is not linear. For instance,

$$y'' - 9y' - 10y^2 = 0, \tag{14.34}$$

to make only a trivial change in the example. We also have to consider the case where the boundary conditions are nonlinear relations, which does happen; but for now, let's just change the DE. Consider what would happen if we "bulled ahead" with the collocation approach: That is, we set up our mesh $\{x_i\}_{i=0}^{N}$ as before, and chose our $2N$ collocation equations $r(x_i + \theta_1 h_i) = 0$ and $r(x_i + \theta_2 h_i) = 0$ for $0 \le i \le N-1$. These equations now are *nonlinear* equations for the unknowns y_i and y_i'. In this example, they are apparently nonlinear of a quite simple sort: Including the boundary conditions $y_0 = 1$ and $y_N = 1$, they are polynomials of degree at most 2, in the $2N+2$ unknowns. Computer algebra systems claim the ability to solve multivariate systems of polynomial equations, so perhaps we could try one of their algorithms.

Unfortunately, the *number of solutions* of such $2N$-variate systems is usually exponential in the number of variables, here $2N$ (after y_0 and y_N have been eliminated). That is, the number of solutions is on the order of c^{2N} for some constant $c > 1$. Every algorithm for exact solution must be capable of finding all solutions—which clearly requires at least an exponential amount of work (in the worst case, doubly exponential). Here, when $N = 3$, there turn out to be 64 solutions to the equations, as we find out by using MAPLE and its Gröbner basis package; when $N = 4$, we killed the job when the memory usage reached 2Gb and our patience ran out. Exponential growth means, sometimes, that even if you can solve six equations in six unknowns, you might not be able to solve eight. Exact methods are not appropriate here; we really only want one solution, the one that tends to $y(x)$ as the mesh is refined.

This suggests that we use Newton's method, if we can think of a good enough initial guess—and this turns out to be a pretty good idea. But rather than using Newton's method on the system of nonlinear algebraic equations that arises from collocation on the original nonlinear equation, it pays us to use a variation on Newton's method that is specific to differential equations: Instead of finding the Jacobian of the (big) system of discrete equations, we make an initial guess as to the solution of the BVP, call it $y^{(0)}(x)$, and find the closest *linear* differential equation to the BVP around this guess. That is, we differentiate the BVP with respect to the unknown solution, in an operator sense.

This is not hard, and in fact we have already been doing this: we called it the variational equation. We use this approach in order to estimate the condition number of a DE (and it's not a bad idea to do this for BVP either). For this example, put $y(x) = y^{(0)}(x) + \varepsilon(x)$, where we presume that $\varepsilon(x)$ is "small." Then, on neglecting terms of $O(\varepsilon^2)$, the differential equation becomes

$$\ddot{\varepsilon} - 9\dot{\varepsilon} - 20y^{(0)}(x)\varepsilon = -\left(\ddot{y}^{(0)} - 9\dot{y}^{(0)} - 10\left(y^{(0)}\right)^2\right), \tag{14.35}$$

which is a linear equation for $\varepsilon(x)$ that we can solve by the method outlined previously: Collocation just gives a system of linear equations. Once we know $\varepsilon(x)$, we then update the solution by saying $y^{(1)}(x) := y^{(0)}(x) + \varepsilon(x)$.

In general, if our differential equation is

$$\dot{\mathbf{y}}(x) = \mathbf{f}(\mathbf{y}(x)) \tag{14.36}$$

and our boundary conditions are

$$\mathbf{G}(\mathbf{y}(a), \mathbf{y}(b)) = \mathbf{0}\,, \tag{14.37}$$

then the linearized equation about the kth iterate is

$$\dot{\varepsilon}(x) = \mathbf{J}_f\left(\mathbf{y}^{(k)}(x)\right)\varepsilon(x) + \mathbf{f}(\mathbf{y}^{(k)}) - \dot{y}^{(k)}(x)\,, \tag{14.38}$$

with the linearized boundary conditions involving the Jacobian derivative of the boundary function $\mathbf{G}$. This is why you should supply Jacobians for nonlinear differential equations and boundary conditions, together with an initial guess as to the solution, to the routines `bvp4c` and similar. To solve nonlinear BVPs, these routines replace the original nonlinear equation with a sequence of linear equations, each (hopefully) a better approximation to the original than the last.

Notice that it matters how you do the linear algebra. For most nonlinear problems, the most costly part of the solution is the setting up and factoring of the Jacobian matrices. Notice that the collocation equations that arise from the linearized differential equation will be (impressionistically) of the form $\mathbf{I} - h\mathbf{J}$, where the $\mathbf{J}$ part will change with each update $y^{(k)}(x)$. That means that finding the solution at the mesh points (from which the interpolant is defined) requires factoring this matrix, every time. If $\mathbf{J}$ is sparse or structured, then if N is at all large, it is important that you tell the code that this is so, in order that it may take advantage of this fact. There are facilities in `bvp4c` for doing so: You may (and should, if it's true) tell `bvp4c` that your Jacobian matrix is sparse, or banded, or symmetric; this can make the difference between solving the system or not.

Also, as is usual with Newton's method, one can have convergence failures of various kinds; there may be multiple solutions, which you can reach by starting from different initial guesses; one can have slow convergence. As with the solution of nonlinear algebraic equations, the initial guess is the greatest single determinant of success or failure. It is so important that the command `bvpinit` basically requires the user to provide an initial guess (even if it is of the crudest kind), for all problems, not just nonlinear ones. How can we find good guesses, though?

The idea of "simple continuation," using the solution of a similar problem with only a slightly different parameter, as the initial guess for the solution of the current problem is a successful one. We have already seen it used in this chapter, not for a nonlinear problem but on a linear problem, the `shockbvp` example from the `odeexamples` demo. Continuation was needed not for the solution of a linear problem, but rather in an attempt to find an equidistributing mesh: The mesh used for a large value of ε was used as an initial mesh for a smaller one, and twice more until the final ε was reached. Of course, the problem of finding the optimal mesh is a *nonlinear* one, and so it is natural to iterate.

To end this chapter, we illustrate the concepts introduced with some examples.

Example 14.4. The first problem is an explicitly nonlinear one. We look at the boundary value problem generated by considering Jeffery–Hamel flow. In a form suitable for numerical solution in MAPLE but not yet in MATLAB, the equation is

$$\frac{d^4}{dt^4}F(t)+2\left(\frac{d}{dt}F(t)\right)\frac{d^2}{dt^2}F(t)+4\frac{d^2}{dt^2}F(t)=0 \tag{14.39}$$

and is subject to the boundary conditions

$$\left\{F''(0)=0, F'(\pi/2)=0, F(0)=0, F(\pi/2)=-2Re/3\right\}. \tag{14.40}$$

Solving this in MAPLE using Allan Wittkopf's sophisticated automatic continuation code [which, to our knowledge, has no published description apart from a remark in Corless and Assefa (2007)] to the effect that it apparently uses an interesting geometric method to construct an initial guess and a step-doubling continuation scheme) is very easy, even for large Reynolds numbers Re.

Let us try to solve this using `bvp5c`. First, we must convert it to standard first-order form: Let $y_1(t)=F(t)$, $y_2(t)=F'(t)$, $y_3(t)=F''(t)$, and $y_4(t)=F'''(t)$, as usual. Then the BVP becomes

$$\begin{aligned}
y_1'(t) &= y_2(t)\\
y_2'(t) &= y_3(t)\\
y_3'(t) &= y_4(t)\\
y_4'(t) &= -2y_2(t)y_3(t)-4y_2(t)
\end{aligned} \tag{14.41}$$

subject to $y_1(0)=y_3(0)=0$, $y_2(\pi/2)=0$, and $y_1(\pi/2)=-2Re/3$. The natural continuation parameter to use is the Reynolds number Re: We begin with small Re, and increase it as desired. The MATLAB code is as follows:

```
function jefferyhamel( Rey, nsteps )
%
% Set up and solve the Jeffery-Hamel BVP by continuation in the Reynolds number.
%
    function dy = JHFlow( t, y )
        dy = zeros(size(y));
        dy(1,:) = y(2,:);
        dy(2,:) = y(3,:);
        dy(3,:) = y(4,:);
        dy(4,:) = -2*y(2,:).*y(3,:) - 4*y(3,:);
    end

    function zer = JHBC( ya, yb )
        zer = [ ya(1)
                ya(3)
                yb(2)
                yb(1)+2*Reyl/3 ];
    end

    Reylocal = linspace( 1, Rey, nsteps );

    coarseoptions = bvpset('Vectorized','on', 'RelTol', 1.0e-3, '
        AbsTol', 1.0e-3 );

    options = bvpset('Vectorized','on', 'RelTol', 1.0e-7, 'AbsTol
        ', 1.0e-7, 'Nmax', 50000 );

    sol = bvpinit( linspace(0,pi/2,11), [1;0;0;0] );

    for i=1:nsteps-1,
        Reyl = Reylocal(i);
        sol = bvp5c( @JHFlow, @JHBC, sol, coarseoptions );
    end

    Reyl = Rey;
    sol = bvp5c( @JHFlow, @JHBC, sol, options ); % tight options
        on last one

    t = RefineMesh( sol.x, 20 );
    [y,dy] = deval( sol, t );
    res = dy - JHFlow( t, y );

    figure(1),semilogy( t, abs( res(4,:) ), 'k-' ),set(gca,'
        fontsize',16),xlabel('t','fontsize',16),ylabel('abs(
        residual)','fontsize',16)
    figure(2),plot( t, y(2,:), 'k-' ), set(gca,'fontsize',16),
        xlabel('t','fontsize',16), ylabel('f(t)␣=␣F''(t)','
        fontsize',16)

end
```

The output from asking for the solution with $Re = 10$ by taking 100 steps is given in Fig. 14.14. This simple example is worth playing with, taking different Re and different numbers of steps. If not enough steps are allowed, then the code might get into trouble and fail to converge on the intermediate solutions; and from those incomplete intermediate solutions, it is possible (and has happened for us, for negative Re) that the *final* iteration converges well enough—but to a different solution than if more continuation steps are used! This nonlinear problem has more than one solution, and which solution you converge to *depends on the initial guess*. This is something to be (very!) wary of when solving nonlinear problems, by whatever method.

This problem can also be solved using Chebfun. See Exercise 14.6. ◁

Example 14.5. We now look briefly at a harder problem, which models a steady-state shock wave in one-dimensional nozzle flow. This is Example 1.17 from Ascher et al. (1988). The equation depends on a parameter ε, which is "essentially the inverse of the Reynolds number." The value of the parameter mentioned in the example is $\varepsilon = 4.792 \cdot 10^{-8}$. The equation is

$$\varepsilon A(x)uu'' - \left(1 + \frac{\gamma}{2} - \varepsilon A'(x)\right) uu' + \frac{u'}{u} + \frac{A'(x)}{A(x)}\left(1 - \frac{\gamma - 1}{2}u^2\right) = 0 \qquad (14.42)$$

on $0 < x < 1$. The function $A(x)$ represents the area of the nozzle at position x and is taken to be $A(x) = 1 + x^2$ as an example. The value of γ is 1.4 for air. The boundary conditions are $u(0) = 0.9129$, apparently corresponding to supersonic flow in the throat, and $u(1) = 0.375$.

Ascher et al. (1988, p.22) comment that "Given its simple appearance, the BVP [Eq. (14.42)] turns out to be a surprisingly difficult nut to crack numerically." The difficulty is that a shock develops as $\varepsilon \to 0$, of width $O(\sqrt{\varepsilon})$ and whose position depends also on ε. The code below, which uses simple continuation, *fails* to find a solution using bvp4c—it fails on the last step, with the message "singular Jacobian encountered." If we replace bvp4c with bvp5c, it nearly succeeds, although it needs a very fine mesh in order to do so, so fine that the use of deval to compute the residual apparently fails owing to rounding errors in the computation of the (monomial basis) interpolant. Additionally, the code gives a warning:

```
Warning: Unable to meet the tolerance without using
more than 50000 mesh points. The last mesh of 14741
points and the solution are available in the output
argument. The maximum error is 2313.74, while
requested accuracy is 4.792e-009.
```

The code executed is this:

```
function sol = shocknozzle4c
%
% Shock wave in nozzle flow
% Example 1.17 in Ascher, Mattheij, and Russell
%
% Attempted solution by bvp4c---which fails.
```

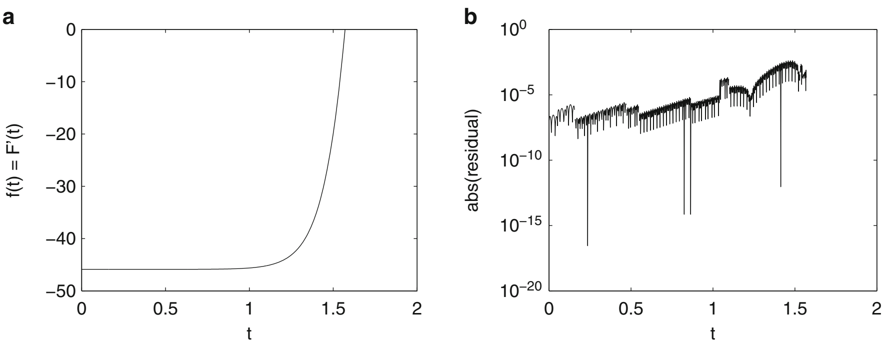


Fig. 14.14 The solution to Eq. (14.41) with *Re* = 100, obtained by asking for 10 continuation steps from *Re* = 0, and its residual. (**a**) Computed solution. (**b**) Residual

```
% Replace bvp4c with bvp5c in what follows, and it apparently succeeds.
% RMC July 2011
%
    function du = shocknozzle(x,u)
        A = 1 + x.*x;
        dA = 2*x;
        du = zeros(size(u));
        du(1,:) = u(2,:);
        du(2,:) = ((1+gamma/2-epsilon*dA).*u(1,:).*u(2,:) ...
            - u(2,:)./u(1,:) - (dA./A).*(1-(gamma-1)*u(1,:).^2/2)) ...
            /epsilon ./A ./u(1,:);
    end

    function y = shockbc( ua, ub )
        y = [ua(1)-0.9129
             ub(1)-0.375];
    end

solinit = bvpinit( linspace(0,1,10), [0.9;0] );

gamma = 1.4;
% simple continuation in epsilon
n = 40;
pow = linspace(1,8,n); % Chebfun with 512 died at 151, or epsilon = 4.224e-3
maxres = zeros(n,1);
xi = linspace(0,1,1001);
for i=1:n,
    epsilon = 4.792 * 10^(-pow(i));
    disp( [i,epsilon] );
    opts = bvpset('reltol',min(1.0e-6,epsilon/5),'NMax',10000);
    sol = bvp5c(@shocknozzle,@shockbc,solinit,opts);
    solinit = bvpinit( sol, [0, 1] );
    [u,du] = deval( sol, xi );
    res = du - shocknozzle(xi,u);
    maxres(i) = max(max(abs(res)));
end
% Polish the last solution a bit
opts = bvpset('reltol',min(1.0e-6,epsilon/10),'NMax',50000);
sol = bvp5c(@shocknozzle,@shockbc,solinit,opts);
[u,du] = deval( sol, xi );
res = du - shocknozzle(xi,u);
figure(1),plot(xi,u(1,:),'k');
xlabel('x','fontsize',16);
ylabel('solution','fontsize',16);
set(gca,'fontsize',16);
figure(2),semilogy( abs(res(2,:)), 'k' );
xlabel('x','fontsize',16);
ylabel('residual');
set(gca,'fontsize',16);
figure(3),semilogy( sol.x(2:end), diff(sol.x), 'k.' );
set(gca,'fontsize',16);
```

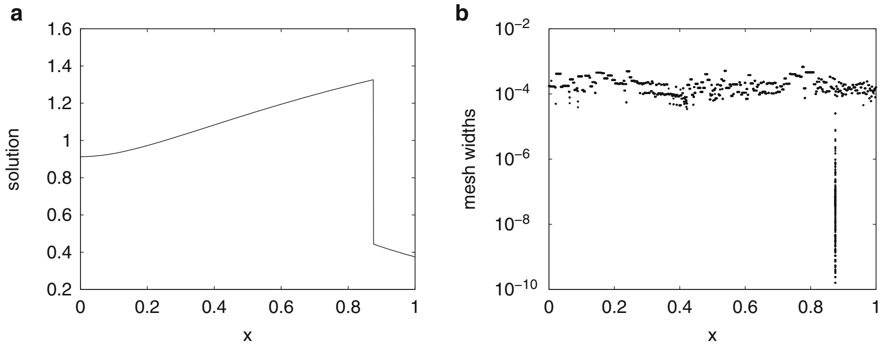


Fig. 14.15 The solution using `bvp5c` to Eq. (14.42) with $\varepsilon = 4.792 \cdot 10^{-8}$, obtained by asking for 40 continuation steps from $\varepsilon = 4.792$, and the final mesh needed—notice that there is a difference of six orders of magnitude in the smallest to the largest mesh width. (**a**) Computed solution. (**b**) Mesh widths

```
xlabel('x','fontsize',16);
ylabel('mesh␣widths','fontsize',16)
end
```

In spite of that, however, the solution looks correct (as seen in Fig. 14.15); we believe that it is the somewhat conservative error measures (which were intended, in MATLAB, for modest accuracy and not for problems like this one) that have failed.

When we tried Chebfun on this example, and MAPLE, we did not succeed in getting a solution for ε smaller than $4 \cdot 10^{-3}$. Of course, the scientific computing BVP code COLSYS succeeded on this already 30 years ago; the problem is not *that* hard. ◁

14.7 Notes and References

For a presentation of the general theory of the conditioning of linear BVPs and an explanation of how it also applies to nonlinear problems, see Ascher et al. (1988 chap. 3). See Ascher et al. (1983) for a discussion of the influence of the polynomial basis used on the conditioning of the ABD linear systems of equations that arise in the solution of BVPs. That paper, which apparently arose after lively discussion with M.R. Osborne and others at a meeting in Houston in 1978, suggests that the Hermite basis (as we use here) also shares the good conditioning properties needed for this application: If the mesh is not too nonuniform, then one expects only modest growth of the condition number with the number of mesh points. A further point in favor of the Hermite interpolational basis is that it results in matrices half the size because continuity is automatically enforced (although, to be fair, this can be dealt with in other bases also).

See Corless and Assefa (2007) for an extensive comparison of the numerical solution of the Jeffery–Hamel flow equations to an analytical solution using elliptic functions.

Residual control (also known as defect control) is used in MIRKDC and a parallel version called PMIRKDC, which is available from the University of Toronto. The basic theory is laid out in Enright and Muir (1996), the parallel version is discussed in Muir et al. (2003), and the use of efficient almost-block-diagonal solvers for the linear algebra is described in Pancer et al. (1997).

The problem of parameter estimation in nonlinear ODE models has been briefly mentioned in the Notes to Chap. 12, and we mention it again here because boundary value problem methods are also useful for this. In part, this is because trying to match up several data points in a measured solution is somehow more like a boundary value problem than it is like an initial-value problem. This is taken up in, for example, Li et al. (2005), and in work by M. R. Osborne (forthcoming).

Problems

Theory and Practice

14.1. Consider the ODE of Eq. (14.2) with *initial* condition $y(0) = y'(0) = \alpha$. Argue that there is an α (in fact, a unique α) such that $y(10) = 1$. Show that $y(x)$ on $0 \le x \le 10$ is exponentially sensitive to changes in α [more properly, if the problem is posed on $0 \le x \le L$, $y(0) = 1$, $y'(0) = \alpha$ with α chosen so that $y(L) = 1$, then $^{\partial y}/_{\partial \alpha}$ is proportional to e^{kL} for some $k > 0$].

14.2. Read the help files for `bvp4c` and `bvp5c`, and the tutorial `http://www.mathworks.com/bvp_tutorial`.

Solve the “measles problem,” taken from (Ascher et al. 1988 p. 13, Example 1.10), which is described in Example 14.1.

14.3. The standard Green’s function construction for $Ly = q$, where $Ly = y' - A(x)y$ subject to general linear boundary conditions at two separate points $a < b$, namely,

$$B_a y(a) = \beta_a, \qquad B_b y(b) = \beta_b,$$

needs a fundamental solution matrix Φ satisfying $L\Phi = 0$ and $B\Phi = I$. For our example problem,

$$B_a = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, B_b = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \beta = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix},$$

and

$$y(x) = \Phi(x)\beta + \int_0^L G(x,t)q(t)dt \tag{14.43}$$

with

$$G(x,t) = \begin{cases} \Phi(x)B_a\Phi^{-1}(t) & t \le x \\ \Phi(x)B_b\Phi^{-1}(t) & t > x \end{cases}$$

Find an explicit expression for $\Phi(x)$ and show directly that Eq. (14.43) solves $Ly = q(t)$ and satisfies the boundary conditions. Show also that G is "not too large."

14.4. Show that $\mathscr{M}_{-a}(\phi) = \mathscr{M}_a^{-1}\left(\phi^{-1}\right)$, where $\mathscr{M}$ is the Hölder mean of the vector ϕ defined on page 713. This shows that a Hölder mean with a negative power still makes sense, because it is a reciprocal mean of reciprocals (like a harmonic mean).

14.5. Solve the problem in Example 14.2 using Chebfun. How big is the residual?

14.6. Solve the Jeffery–Hamel flow equations (14.39) with the boundary conditions (14.40) for $Re = 100$ using Chebfun. Compare your solution with that of `bvp4c`. You may find the examples in the guide at `http://www2.maths.ox.ac.uk/chebfun/guide/html/guide10.shtml` helpful. From that guide:

> It is not clear when these approaches can or cannot compute in speed and robustness with BVP4C/BVP5C. But they offer something entirely new,[. . .]

See also Birkisson and Driscoll (2012).

14.7. Solve $y'' = y^2 - 1$ on $0 \le x \le 1$ subject to $y(0) = 0, y(1) = 1$.

Investigations and Projects

14.8. Write your own MATLAB collocation code to solve linear BVPs, and test your code on the BVP in `odeexamples('bvp')`. Linearize the BVP as appropriate.

14.9. Write your own equidistribution routine that attempts to find optimal mesh. Conjoin this code with your routine from Problem 14.8 and test it on the BVP in `odeexamples('bvp')`.

14.10. Add quasilinearization to your code from Problem 14.9.

14.11. Blasius' equation for the incompressible velocity profile for a flat plate is $2f''' + ff'' = 0$ on $0 \le \eta \le \infty$ subject to $f(0) = f'(0) = 0$ and $f'(\eta) \to 1$ as $\eta \to \infty$.

Solve this using `bvp4c` or `bvp5c`. Techniques for handling the infinite interval include chopping (i.e., choosing some large L and setting $f'(L) = 1$; better, a sequence of such Ls and examining convergence) or changing variables and thereby transforming the equation. The last technique usually introduces a singularity into the equation somehow.

14.12. In Sect. 12.10 and in Problem 13.36, we considered the problem of finding an interpolant for the skeleton of the numerical solution of the IVP $\dot{y} = f(y)$, $y(t_0) = y_0$ [call that skeleton (t_k, y_k) for $0 \le k \le N$] that minimized the 2-norm of the residual (deviation) on each step: That is, we sought $z(t)$ such that $z(t_k) = y_k$, $z(t_{k+1}) =$

y_{k+1} with minimal $\int_{t_k}^{t_{k+1}} \Delta^H(\tau)\Delta(\tau)d\tau$, where $\Delta(t) := \dot{z}(t) - f(z(t))$. This led to the nonlinear BVP on $t_k \le t \le t_{k+1}$

$$\begin{aligned}\dot{z}(t) &= f(z(t)) + \Delta(t)\\ \dot{\Delta}(t) &= -\mathbf{J}_\mathbf{f}^H(z(t))\Delta(t)\end{aligned}$$

subject to $z(t_k) = y_k$ and $z(t_{k+1}) = y_k$.

1. Write a MATLAB script to solve the BVP for each subinterval of the Euler method solution to $\dot{y} = y^2 - t$, $y(0) = {}^{-1}\!/\!_2$, on $0 \le t \le 5$ with $h = {}^5\!/\!_N$. How much smaller is the optimal $\|\Delta\|$? Is it still $O(h)$?
2. Write an MATLAB script to find the optimal interpolant for the leapfrog solution to the Hénon–Heiles problem and test it.
3. Write a general MATLAB m-file that finds the optimal interpolant for the skeleton of any numerical method. Test your code by some examples using `ode45` and `ode15s`. You should get better residuals although one expects problems if the step sizes are too small.
4. Discuss the benefits and drawbacks of this approach.

14.13. Use `bvp4c` or `bvp5c` to find periodic solutions to the following:

1. van der Pol's equation $y'' - \varepsilon(1-y^2)y' + y = 0$ with $\varepsilon = 0.3$;
2. the Rayleigh equation $y'' = \varepsilon(1 - \frac{4}{3}(y')^2)y' + y = 0$ with $\varepsilon = 0.3$.

Chapter 15
Numerical Solution of Delay DEs

Abstract Delay differential equations differ from ordinary differential equations in that they may need their initial conditions specified on an *interval*, not just at a finite set of points. The influence of *discontinuities* propagates forward in time as the solution progresses. As is the case with ODE, however, a good numerical solution gives the exact solution to a nearby problem. The same technique as used in previous chapters, namely, computing the *residual*, works here as well to verify that the computed solution is good in this sense. One then has to wonder, as usual, about the *conditioning* of the problem. ◁

Delay differential equations occur very frequently in applications, especially in mathematical biology. The numerical solution of delay differential equations (DDE) has a great similarity to the numerical solution of initial-value problems for ordinary differential equations, but has some important differences, as well. We demonstrate with the following simple example:

$$\dot{y}(t) = -\tfrac{1}{5}y(t-1)\,, \tag{15.1}$$

which is to hold for $t \geq 0$. Quite obviously, if we are to be able to find even $\dot{y}(0)$, we must therefore know $y(-1)$, and indeed to find $\dot{y}(t)$ for any t in $[0,1)$, we must know $y(t-1)$; that is, we must have knowledge of a *history* function $\phi(t)$ such that $y(t) = \phi(t)$ for $-1 \leq t \leq 0$. This history function plays the role of the initial condition, but it's more than an initial value: It is a quite general function defined on an interval. Solutions of DDE are therefore *infinite-dimensional* objects, and a good deal more complicated than IVP for ODE. Nonetheless, we may solve this equation, and others more difficult yet, with a simple call to either the MATLAB routine `dde23` or, as we will see later, `ddesd`.

For concreteness in this example, we first choose $\phi(t) = \exp(\lambda t)$, for the peculiar value of $\lambda = -0.259171101819074$. Then the commands

```
dde      = @(t,y,Z) -0.2*Z(:,1);
lambda   = -0.259171101819074;
```

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_15,

```
ddehist  = @(t) exp(lambda*t);
opts     = ddeset('AbsTol',1.0e-8) ;
tspan    = [0, 4];
delay    = 1;
sol      = dde23( dde, delay, ddehist, tspan, opts )
```

solve the problem with a claimed absolute tolerance of 10^{-8}. In this syntax, the variable y refers to an approximation of $y(t)$, while Z refers to an approximation of $y(t-1)$. The syntax `Z(:,1)` picks out the first column of Z, but for this problem, Z has, in fact, only one column. We will see more examples later. The solution, evaluated at many points with the help of the auxiliary routine `deval` (the same routine that evaluates the solution of an IVP or a BVP), is plotted in Fig. 15.1. How accurate is the solution really? Just as with IVP and BVP, we may compute a residual, and plot it. See Fig. 15.2. We take up how exactly to do this in MATLAB in a later section. That is, our computed solution is the *exact* solution to $\dot{y}(t) = -\frac{1}{5}y(t-1) + 2.5 \cdot 10^{-6}v(t)$, with $\|v(t)\| \le 1$, approximately. Naturally, this makes us wonder how significant such a perturbation is, but as usual in physical modeling, one has to think about such things anyway. Before we take that up in detail, however, let us consider another important difference between the numerical solution of IVP for ODE and the numerical solution of DDE.

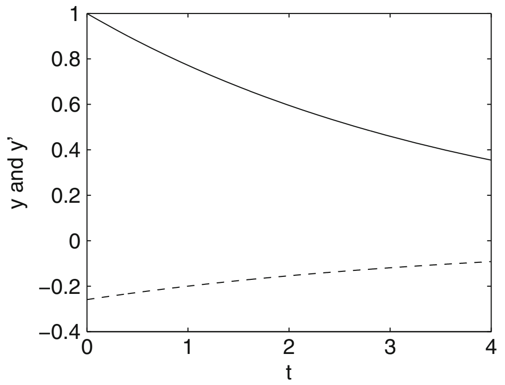


Fig. 15.1 The solution of $\dot{y}(t) = -\frac{1}{5}y(t-1)$, with $y(t) = \exp(\lambda t)$ on $-1 \le t \le 0$, where $\lambda = -0.259171101819074$. *Dotted line* is $\dot{y}(t)$

The history function $\exp(\lambda t)$, with that peculiar λ, was chosen artfully, so that $\exp(\lambda t)$ was, in fact, the exact solution of the DDE. This meant, in particular, that the transition between history and solution, at the initial point $t = 0$, was very smooth (if it weren't for rounding errors, it would be analytic there). This is highly unusual. It is almost always the case that there is a jump in the first derivative there, because usually $\dot{y}(0^+)$, defined by the DDE itself, is independent from $\dot{\phi}(0^-)$, the left-hand derivative of the history function. Indeed, $\dot{\phi}(0^-)$ may not even exist: The history may be discontinuous. However, even if it is continuous, it is usually the case that

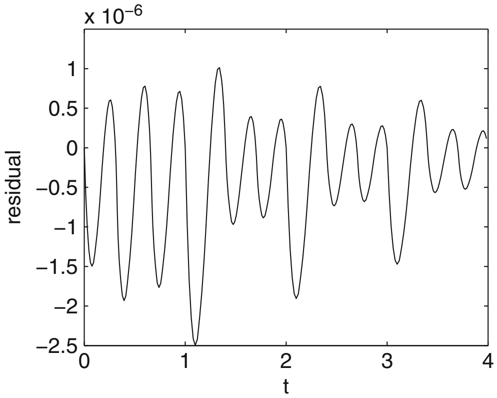


Fig. 15.2 The residual in the computed solution $r(t) = \dot{y}(t) + \frac{1}{5}y(t-1)$, with $\lambda = -0.259171101819074$. The residual is at most about 10^{-6}, whereas the tolerance asked for a solution accurate to about 10^{-8}

the derivative of ϕ is not the derivative of y, at $t = 0$. This initial discontinuity causes a problem for high-order numerical methods and must be treated with care. See Feldstein et al. (2006) for a discussion of this issue.

Worse, it propagates.[1] At $t = 1$, $\dot{y}(t) = -\frac{1}{5}y(t-1)$, and so we see that $\ddot{y}(t)$ has a discontinuity at $t = 1$. Similarly, the third derivative has a discontinuity at $t = 2$, and so on. For this simple DDE, with only one (constant) delay of unit length, it's easy to identify where the discontinuities are. Furthermore, they weaken as t increases, occurring in higher and higher derivatives, so the solution $y(t)$ is getting smoother and smoother. This is a useful feature of what's referred to as "retarded" delay differential equations, where the delays only occur in arguments to y, never in arguments to $\dot{y}$, when the DDE is called a "neutral" DDE (more about which in a moment).

In order to be robust, it turns out to be imperative that a code track discontinuities, or provide some foolproof method for detecting them. The code should never step over a strong discontinuity—that disrupts the error estimates extremely. Because `dde23` deals only with constant lags, it can work out ahead of time where all the discontinuities will occur (and how strong they will be). If there are multiple delays, then this is an onerous bookkeeping task, which we are glad that the code does for us.

To expand on that detail, consider what happens if our DDE is $\dot{y}(t) = f(t, y(t), y(t-\tau_1), y(t-\tau_2))$, with two delays. The primary discontinuity at $t = 0$ will be reflected at $t = \tau_1$, and again at $t = \tau_2$. Then again at $t = \tau_1 + \tau_2$. Also, of course, at each of $2\tau_1, 3\tau_1$, and so on, as well as $2\tau_2, 3\tau_2$, and so on. In fact, at each of $k\tau_1 + \ell\tau_2$. Some of these locations may be very close to one another. If the discontinuities weaken quickly, as they do with retarded DDE, then some of the locations can be

[1] In Brunner (2004), this propagating discontinuity is called a *primary* discontinuity (following notation of earlier authors).

safely ignored. If there are more than two delays, then the bookkeeping to keep track of them gets painful. Finally, there may be other, possibly stronger, discontinuities in the history than just the primary discontinuity at $t = 0$.

Let us modify our first example to see what happens if $\phi(t)$ is itself discontinuous. Choose a square-wave history, $\phi(t) = (-1)^{\lfloor -4t \rfloor}$, which has jump discontinuities at $t = -1$, $-{}^{3}\!/_{4}$, $-{}^{1}\!/_{2}$, and $-{}^{1}\!/_{4}$, in addition to the derivative discontinuity at $t = 0$. Now, we have this:

```
dde     = @(t,y,Z) -0.2*Z(:,1);
ddehist = @(t)  (-1).^floor(-4*t);
opts = ddeset('AbsTol',1.0e-8,'Jumps',[-1,-0.75,-0.5,-0.25,0]);
tspan    = [0, 4];
delay    = 1;
sol      = dde23( dde, delay, ddehist, tspan, opts )
```

It is instructive to look at the output from the call to `dde23` this time; typing `sol` to display the solution structure gives

```
     solver: 'dde23'
    history: @(t)(-1).^floor(-4*t)
    discont: [1x21 double]
          x: [1x29 double]
          y: [1x29 double]
      stats: [1x1 struct]
         yp: [1x29 double]
```

Notice that we helpfully informed `dde23` of the locations of the discontinuities in the history, using the `Jumps` option. Asking for the discontinuities, we receive this output:

```
sol.discont

ans =
   -1.0000    -0.7500    -0.5000    -0.2500
    0          0.2500     0.5000     0.7500
    1.0000     1.2500     1.5000     1.7500
    2.0000     2.2500     2.5000     2.7500
    3.0000     3.2500     3.5000     3.7500
    4.0000
```

(The output was edited for alignment and concision.) As we can see, thereafter the code has correctly predicted discontinuities at $t = {}^{k}\!/_{4}$. When we plot the solution and its derivative, we see some interesting features appear. Look at Fig. 15.3. The graph of $y(t)$ is smoother than the graph of $\dot{y}(t)$, as it should. If you look carefully, you see little sharp spikes in $\dot{y}(t)$, near the jump discontinuities in $0 \le t \le 1$. These spikes are numerical artifacts: they are errors (probably due to differentiating a polynomial interpolant to nondifferentiable data). They're also not surprising, because it's hard to solve discontinuous problems. We shall look again at this problem in Sect. 15.5.

So how accurate is the solution this time? See Fig. 15.4. We see that the residual is *much* larger than our tolerance, near the discontinuities. One saving grace is that the large residuals occur in very narrow subintervals, and so their influence on the

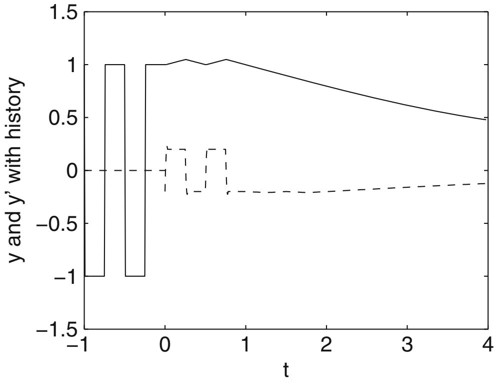


Fig. 15.3 The solution of $\dot{y}(t) = -\frac{1}{5}y(t-1)$, with square-wave history on $-1 \le t \le 0$. The *dotted line* is $\dot{y}(t)$

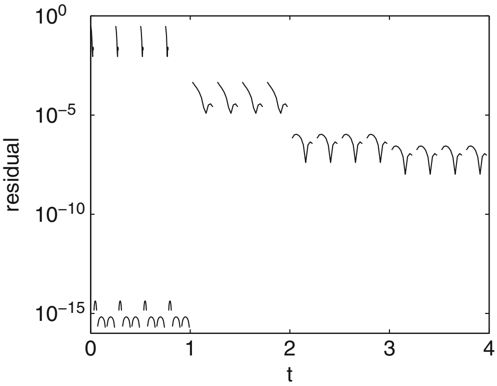


Fig. 15.4 The residual in the computed solution $r(t) = y_t(t) + \frac{1}{5}y(t-1)$, square-wave initial history, on a log scale. The residual is, near the discontinuities, quite large. Nonetheless, the effects of the discontinuities are confined to small intervals

solution is constrained. Nonetheless, we see that the discontinuities are giving the solver some trouble, as is only to be expected. What is welcome, however, is that the code does a creditable job in containing the influence of the discontinuities.

The routines dde23 and ddesd use explicit Runge–Kutta formulae together with natural interpolants. The low-order method dde23 uses the same $(2,3)$ pair that ode23 uses to integrate initial-value problems for ordinary differential equations. The higher-order code uses classic RK4 together with a natural interpolant to provide a continuous solution, and controls the size of the residual. There are some interesting differences to the use of these one-step formulæ for DDE as opposed to IVP for ODE. For example, it would be inefficient to restrict the time step of the underlying RK formula to be smaller than the smallest delay (which indeed may vanish, as we will see later). But the RK formulæ may need the value of the derivative at a point somewhere in the current step not at the natural places, which are artfully arranged so that the errors cancel out at the end of the step. Current high-quality codes use the interpolant from the previous step to predict these "unnatural" values, and then (even for explicit methods) iteration is used to correct the values to obtain a highly accurate estimate. This will be pursued further in the exercises.

It is true that *this* DDE is so simple that it can be integrated analytically, by what is known as the method of steps. On the interval $0 \le t \le 1$, the differential equation is just $\dot{y}(t) = -\frac{1}{5}\phi(t-1)$, and $\phi(t)$ is known; hence, we just integrate that piecewise constant function to get a piecewise linear function. Explicitly, on the interval $0 \le t \le 1$, the solution is

$$y(t) = \begin{cases} t/5 + 1 & t < 1/4 \\ -t/5 + 11/10 & t < 1/2 \\ \frac{t}{5} + 9/10 & t < 3/4 \\ \frac{-t}{5} + 6/5 & 3/4 \le t \end{cases}, \tag{15.2}$$

and this is continuous. In the interval $1 \le t \le 2$, now that we know the solution on $0 \le t \le 1$, we can find the solution again by just integrating. The solution is

$$y(t) = \begin{cases} -\frac{1}{50}\,(t-1)^2 - \frac{1}{5}t + \frac{6}{5} & t < 5/4 \\ \frac{1}{50}\,(t-1)^2 - \frac{11}{50}t + \frac{489}{400} & t < 3/2 \\ -\frac{1}{50}\,(t-1)^2 - \frac{9}{50}t + \frac{469}{400} & t < 7/4 \\ \frac{1}{50}\,(t-1)^2 - \frac{6}{25}t + \frac{251}{200} & 0 \le t - 7/4 \end{cases}, \tag{15.3}$$

which is continuously differentiable. In the next interval, the polynomials are degree 3, and so on; smoothness increases as t increases. This method is useful theoretically, to establish existence and uniqueness, in general; but in general, the symbolic integrations become extremely cumbersome. We shall return to this method in Sect. 15.5, however.

This behavior, the weakening of propagated discontinuities, is true only for DDE of the retarded type, that is,

$$\dot{y}(t) = f(t, y(t), y(d_1(t)), y(d_2(t)), \ldots, y(d_k(t)))\,, \tag{15.4}$$

where each delay function $d_j(t) < t$ (or possibly actually equals t, in which case we say that the delay *vanishes*, which itself is problematic, but less so than the next case). If the DDE also contains terms that depend on the *derivatives* of y at previous times, such as

$$\dot{y}(t) = f(t, y(t), y(t-1), \dot{y}(t-1))\,, \tag{15.5}$$

then we have what is called a *neutral* DDE. In this case, the discontinuities do not weaken as they propagate, and the solution is much harder. Indeed, even existence and uniqueness can be problematic. We shall return to this later.

15.1 The Residual, or Defect

DDE can be nonlinear, and often are in applications; see, for example, `ddex2` from the `odeexamples` demo in MATLAB. Nonlinearity per se doesn't present any new difficulties, but, of course, the old difficulties (nonuniqueness of solutions, requirement of solving nonlinear algebraic equations in order to advance a step), while familiar, are still potent. For example, a nonlinear equation due to Cooke et al. (1999) that models population $y(t)$ at time t by the equation

$$\dot{y}(t) = be^{-ay(t-T)-d_1T}y(t-T) - d\,y(t) \tag{15.6}$$

with, say, history $y(t) = 3.5$ for $t \leq 0$ and parameters $a = 1$, $d = 1$, $d_1 = 0$, and $b = 80$ can be solved on $0 \leq t \leq 25$ using `dde23` by the following commands:

```
%
% Cooke, van den Driessche and Zou example (1999)
%
clear all
close all
starttime=clock;
% Parameter values from Fig 3(d)
a = 1;
b = 80;
d1 = 0;
d = 1;
T = [0.2, 1.0, 2.4];
opts = ddeset( 'RelTol', 1.0e-7, 'AbsTol', 1.0e-8 );
% Use a structure for the history, to make computing
% the residual easier, later.  Take a wide enough interval
% that we cover all three delays used here.
hist = dde23( @(t,y,z) 0, [], 3.5, [-5,0], opts );
% Do three integrations for three different delays
for i=1:3,
   dde = @(t,y,z) b*exp(-a*z-d1*T(i))*z - d*y;
   sol(i) = ddesd(dde, T(i), hist, [0,25], opts );
end;
timetaken=etime(clock,starttime);
% Now plot the solutions
figure
plot( sol(1).x, sol(1).y, 'k-', sol(2).x,sol(2).y, 'k--', ...
      sol(3).x, sol(3).y, 'k-.' )
set(gca,'fontsize',16);
xlabel('t','fontsize',16);
ylabel('y','fontsize',16);
% Now compute and plot the residuals
npts = 10001;
t = linspace(0,25,npts);
r  = zeros(3,npts);
for i=1:3,
    dde = @(t,y,z) b*exp(-a*z-d1*T(i)).*z - d*y;
    [yt,dyt] = deval( sol(i), t );
    % Evaluate the solution at the delayed value of t, too
    zt = deval( sol(i), t-T(i) );
    r(i,:) = dyt - dde(t,yt,zt);
end
figure
r = abs(r);
semilogy( t, r(1,:), 'k-', t, r(2,:), 'k--', t, r(3,:), 'k-.' )
set(gca,'fontsize',16);
xlabel('t','fontsize',16);
ylabel('residual','fontsize',16);
axis([0,25,1E-20,1E5]);
set(gca,'YTick',[1E-20,1E-15,1E-10,1E-5,1,1E5]);
% The code produces the exact solution to a problem within
% 1.0e-5 of the stated prrroblem.
```

Notice that the history $y(t) = 3.5$ is given to dde23 not as a simple constant, but rather as the solution to the trivial DE $y'(t) = 0$, $y(-5) = 3.5$. This creates a solution structure, and when given to dde23 as the history for the subsequent nontrivial integration, the code automatically extends the solution to include the past history. This is convenient for plotting, but even more convenient for computing the residual, because then deval doesn't complain if you ask it to evaluate $y(t-T)$ when $t-T$ is somewhere in $-5 \le t-T \le 0$, that is, in the history. Other software packages may have similar tricks, but you could always do the bookkeeping yourself.

The solutions are plotted in Fig. 15.5, and the computed residual in Fig. 15.6. Notice that the residual is smaller than 10^{-5}, but larger than our requested tolerance. This is because the code uses an integral measure of size—it controls not the max norm of the residual, but rather a scaled integral norm. It controls not $\|r\|_\infty$, but $h_n\|r\|_2$. For well-conditioned problems, it is often true that the forward error, the difference between the computed solution and the reference solution, will be adequately controlled by this method. For our purposes, the factor of h_n is a distrac-

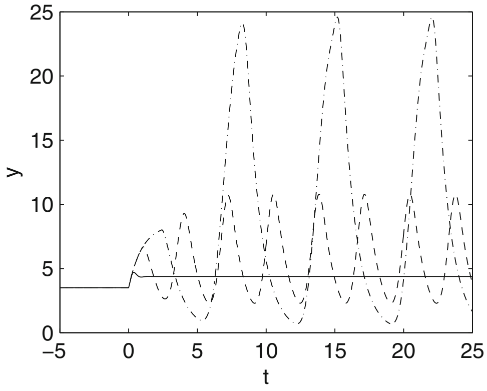


Fig. 15.5 The solutions to (15.6) for $T = 0.2$ (*solid line*), $T = 1$ (*dashed line*), and $T = 2.4$ (*dash-dot line*), with history $y = 3.5$ for $t < 0$. Parameter values are $a = d = 1$, $d_1 = 0$, and $b = 80$. Absolute tolerance was 10^{-8} and relative tolerance 10^{-7}

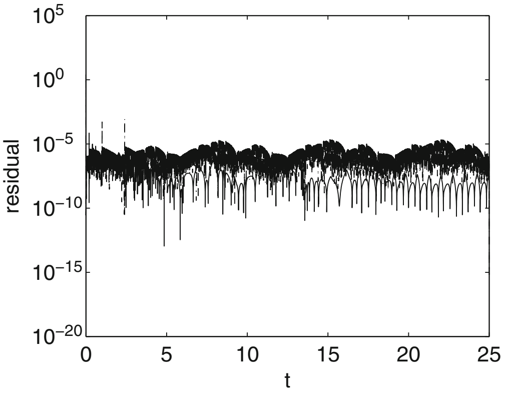


Fig. 15.6 The residual at 10,000 points in each of the computed solutions to (15.6) on a log scale. The residuals are uniformly small, albeit larger than our tolerances, but we have computed in each case the reference solution to a problem quite close to the original

tion, but not insuperable. One final note is that the interpolant that is used internally might not be the "best possible" interpolant for the problem—it is conceivable that the "best possible" residual might be smaller than this. The method of verification advocated in this book provides a sufficient condition for backward accuracy, in other words.

Still, by this analysis, what we have shown is that the solution from `dde23` is the *exact* solution to

$$\dot{y}(t) = be^{-ay(t-T)-d_1T}y(t-T) - d\,y(t) + 10^{-5}v(t)\,, \tag{15.7}$$

for some wiggly $v(t)$ with $\|v\|_\infty \leq 1$. Additionally, we can make the residual smaller by tightening the tolerances used. These are useful reassurances when solving a nonlinear equation such as this one.

But what does that residual *mean* for the DDE? The original DDE did not contain t explicitly—it was unforced by any time-dependent term. To see what this means, one could go back to the original equation and see that it is a single-species population model; and *already* it is an oversimplification, because populations are made up of individuals, and if the nominal population is in the millions, say, then we are talking about fluctuations on the order of about 10 organisms—extra deaths, extra births, coming into the population more or less randomly. This is a perfectly acceptable modification to the equation (in this case), because we have to think about such fluctuations anyway, in deciding whether or not the model is robust under real-life perturbations. The perturbation *does* make the theorems in Cooke et al. (1999) that talk about irrelevant equilibria as stated. With a time-dependent [called *persistent* in the literature; see, e.g., Definition 2.2 in Baker et al. (2006)] perturbation, those theorems no longer apply directly. One needs to extend the theorems to consider what happens to equilibria under small time-dependent perturbations, and of course there are results for this case as well. We do not take this topic up here in great detail, but instead direct the reader to the literature on perturbations of delay differential equations. In a later section, we will talk a little bit about stability in general. The end result for the use in Cooke et al. (1999), however, where they used numerical simulation to confirm their theorems, was that the numerics produce results that are consistent with their theorems (if the effects of perturbations are small; moreover, we seem to have some evidence that they are). In other applications, the residual can help us decide if the numerics did a good enough job that their results can be trusted, for other purposes.

15.2 State-Dependent Delays: `ddesd` Versus `dde23`

Nonlinearities can occur in many ways for ODE. What is new for DDE is that the delay functions $d_j(t)$ mentioned above can be time-dependent, or even *state-dependent*, and this can introduce nonlinearity in a whole new way. That is, instead of just $d_j(t)$, we can have $d_j(t, y(t))$; that is, the delay depends on the unknown value of the function itself. An example is `ddex3` in the `odeexamples` demo. Before we tackle that rather complicated example, which has a vanishing state-dependent delay, we consider the following simpler problem from Iserles (1994). Here the delay does not depend on the state, but it's not a simple lag either:

$$y'(t) = -\tfrac{1}{4}y(t) + y(t/2)\,(1 - y(t/2)) \qquad (15.8)$$

with the *initial condition* $y(0) = 1/2$. Here the delay function is $d_1(t) = t/2$, and for $t = 0$ this is no delay at all, but for all $t > 0$, we must have $t/2 < t$. The code `dde23` cannot handle such problems: It can only deal with constant delays. We turn to the more powerful `ddesd` (for DDE state-dependent):

```
% Iserles' 1994 example
% z'(t) = -0.25*z(t) + z(t/2)*(1-z(t/2)),
% z(0)=0.5, on 0 <= t <= 10^5
tend = 10^3;
sol  = ddesd( @(t,y,z) -0.25*y + z.*(1-z), @(t,y) t/2, 0.5, [0,
    tend] );
t    = linspace(0,tend,3*tend+1);
[y,yp] = deval( sol, t );
t2 = t/2;
y2 = deval( sol, t2 );
r2 = yp - (-0.25*y + y2.*(1-y2) );
figure
plot( t, y, 'k' )
set(gca,'fontsize',16);
xlabel('t','fontsize',16);
ylabel('y','fontsize',16);
figure
plot( t, r2, 'k' )
set(gca,'fontsize',16);
xlabel('t','fontsize',16);
ylabel('residual','fontsize',16);
```

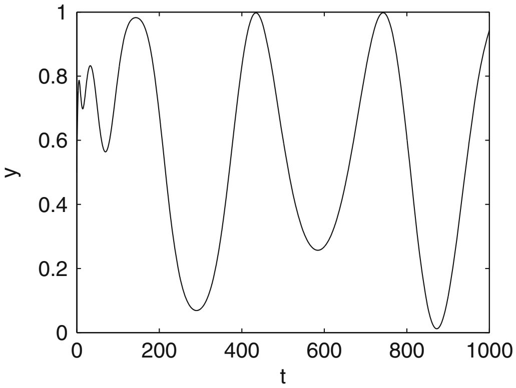


Fig. 15.7 The solutions to (15.8) with default tolerances

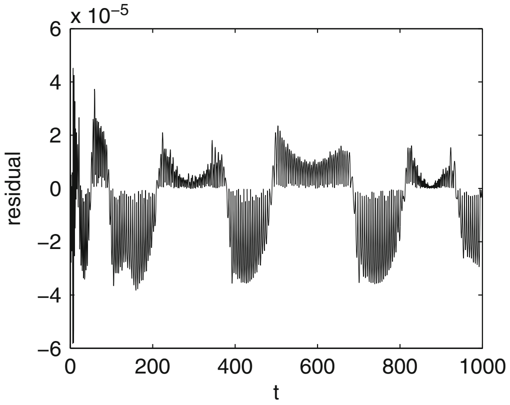


Fig. 15.8 The residual in the computed solution to (15.8) on a linear scale. The residuals are uniformly small, and we have computed the exact solution to a problem quite close to the original

The solution on $0 \le t \le 10^3$ is plotted in Fig. 15.7, and the residual (at default tolerances) is plotted in Fig. 15.8, where we see that it is uniformly less than 10^{-5} again. That is, we have not solved this pantograph-like equation exactly, but have rather solved $y'(t) = f(y(t), y(t/2)) + 10^{-5}v(t)$, where $v(t)$ is uniformly bounded. Again, to see if this is an acceptable result, one would have to consider natural perturbations in the original system being modeled. We defer this discussion to the section on stability, and note only that the solution by ddesd was straightforward and took only 261 steps. We had to specify the DDE as taking t, $y(t)$, and $y(t/2)$ (called "z"), and specify a lag function $t/2$, but otherwise the call was similar to dde23. To do the solution on $0 < t < 10^5$, as was claimed in Iserles (1994), takes a bit more effort (about 20,000 steps) but is still perfectly feasible, and the residual is again less than 10^{-5} in magnitude.

Now let us take a truly state-dependent example, from the `odeexamples` demo, taken there from Enright and Hayashi (1997b) (see also Enright and Hayashi (1997a)). The equation is

$$\begin{aligned}\dot{y}_1(t) &= y_2(t)\\ \dot{y}_2(t) &= -y_2\left(e^{1-y_2(t)}\right) y_2^2(t) e^{1-y_2(t)}\,. \end{aligned} \tag{15.9}$$

The delay function $d_1(t,y) = \exp(1-y_2(t))$ occurs in the second equation. It's not even clear a priori that $d_1 \le t$. If ever the "delay" functions ask for knowledge of the unknown function at *future* values of t, this is a real problem—solutions will often fail to exist, and if they do exist, they can be horribly ill-conditioned, even ill-posed. The routine `ddesd` does not allow $d_j > t$, by the simple device of using $\min(t, d_j(t))$ (this restricts problems to ones with $t_0 < t_f$, which is a good idea anyway). For this example problem, from the known exact solution $y_1(t) = \log t$ and $y_2(t) = {}^1\!/_t$ (remember, this is a test problem), we can plot $d_1(t)$ and learn thereby that the numerical method should be doing well enough, because $d_1(t) \le t$ on $0 < t < 5$, say. However, at $t = 1$, equality is achieved: The delay vanishes. This is a tough test problem.

Another difficulty is that to track propagating discontinuities, as we must, we have to solve nonlinear equations: For each previously known location τ^* of a discontinuity, we have to find all values of t in the current contemplated step that satisfy

$$d_j(t, y(t)) = \tau^*\,. \tag{15.10}$$

Such t will also have discontinuities in the solution (for retarded DDE, the discontinuities will be weaker). In general, this equation is nonlinear and depends on the unknown function $y(t)$. An initial guess might be available if the continuous extension to the numerical solution can be extrapolated into the current step, but in any case the location(s) of the discontinuities in the interval must be located by an iterative solution method. The code must step no farther than the first such discontinuity. The code `ddesd`, just as with other high-quality DDE solver codes, will handle this automatically. It is a difficult problem, however.

To get our example started, we use the known exact solution on $t < 0.1$ and integrate from $t = 0.1$ to $t = 5$. This is done in the `odedemo` file, but we include code here that has been modified to compute the residual:

```
function ddex3mod
%DDEX3  Example for DDESD, modified to compute the residual.
%
%   Modifications by Robert M. Corless, January 2011
%   See the DDEX3 example in odedemos['Delay Differential Equations']
%   by Jacek Kierzenka and Lawrence F. Shampine
close all
t0 = 0.1;
tfinal = 5;
tspan = [t0, tfinal];
```

```

teensy = 1.0e-5;

% Use a solution structure for the history, to make residuals easy
ddex3hist = ddesd( @(t,y,z) [1/t; -1.0./t.^2], [], ...
                  [log(teensy), 1.0/teensy], [teensy, 0.1] );
% The call is modified to use the structure history
sol = ddesd(@ddex3de,@ddex3delay,ddex3hist,tspan);

% Sample the residual
npts = 1001;
% Step *just* off the initial point (residual huge there)
t = linspace(t0+teensy,tfinal,npts);
[y,yp] = deval( sol, t );
% Compute the corresponding delays
tdelay = ddex3delay( t, y );
% Function at delayed arguments
z = deval( sol, tdelay );
resid = zeros(2,npts);
% ddex3de was not vectorized, step through to compute residuals
for i=1:npts,
    resid(:,i) = yp(:,i) - ddex3de(t(i), y(:,i), z(:,i) );
end
figure
semilogy(t, abs(resid(1,:)),'k')
xlabel('t')
ylabel('residual')
h = diff( sol.x );
figure
semilogy( sol.x(2:end), h, 'k.')
ylabel('step␣sizes')
xlabel('t')
% The following code is identical to that of DDEX3

function d = ddex3delay(t,y)
% State dependent delay function for DDEX3.
  d = exp(1 - y(2,:));

% ---------------------------------------------

function dydt = ddex3de(t,y,Z)
% Differential equations function for DDEX3.
  dydt = [ y(2); -Z(2)*y(2)^2*exp(1 - y(2))];
```

Instead of providing the exact solution history directly, we integrate $y_1'(t) = {}^1\!/\!_t$ and $y_2'(t) = -{}^1\!/\!_{t^2}$ on $\varepsilon < t \leq 0.1$ for some teensy epsilon. This gives us a solution structure to hand to ddesd, so that we may compute the residual in a convenient manner. The residual turns out to be less than 10^{-3}, except right at the start where it is rather large. To further examine the progress of the code, we look at the step sizes taken. When we plot the step sizes used by the code, we see that they vary smoothly from about 10^{-6} up to a little less than 1. The smoothness of the changes indicate that the code had little difficulty. Thus, the solution plotted in ddex3 in the demo is the exact solution to a perturbed problem, where the perturbation is about 10^{-3}. Of course, the demo compares the numerical solution to the known exact solution, which agrees at least to visual accuracy. With the residual approach, we note that we can assess the validity of a numerical solution *even when we do not know the reference solution*. This is an extremely important consideration. Of course, we then have to answer the question of whether or not the underlying problem is sensitive to changes. But applied mathematicians and scientists have to do that anyway.

15.3 Conditioning

The conditioning of delay differential equations is quite a difficult problem. An excellent compendium of results can be found in Baker et al. (2006) and the references therein, where several formulæ similar to the Gröbner–Alexeev nonlinear variation-of-constants formula are considered for various classes of DDE. Note that the problems are quite delicate: In some cases, the solutions are not differentiable with respect to some of the parameters (the delay itself can be such a parameter).

Let us return to our simple test problem $\dot{y}(t) = ay(t-1)$. The exact solution to this problem, as described by Wright (1947), can be found by the use of Laplace transforms and, if $a \neq -\exp(-1)$, expressed as a Dirichlet series (or nonharmonic Fourier series), namely,

$$y(t) = \sum_{k=-\infty}^{\infty} c_k e^{\lambda_k t}\,, \tag{15.11}$$

where the $\lambda_k = W_k(a)$ are the different branches of the Lambert W function evaluated at $z = a$. The values of c_k are determined by certain residues of the history function and the initial condition, as described by Wright (1947), or a truncated version can be found by least-squares fit to the history function, as in Heffernan and Corless (2006). For example, if $a = {}^{-1}\!/\!_2$, then the two terms in the series with the largest real part of λ_k are, with history function $\phi(t) = \cos \pi t$ on $0 \le t \le 1$,

$$1.784282170\, e^{-0.7940236323\,t} \cos(0.7701117505\,t) \\ - 4.960083346\, e^{-0.7940236323\,t} \sin(0.7701117505\,t)$$

because $W_k({}^{-1}\!/\!_2) \doteq -0.7940236323 \pm 0.7701117505i$ for $k = 0, -1$. The constants c_k in front are determined by the history function, and they don't really matter all that much. Since all the other terms have $\mathrm{Re}(\lambda_k) < -0.79$, we conclude that the identically zero solution of $\dot{y}(t) = -\frac{1}{2}y(t-1)$ is asymptotically stable (in the terminology of the literature). Existing perturbation results say, then, that for small enough time-dependent perturbations of this delay differential equation, the solution will come *close* to zero, and stay there. The details of just how close "close" is, and how small "small enough" must be, do not concern us at this time. At this point, it should only be plausible that the zero solution of this delay-differential equation is well-conditioned, in numerical analysis parlance.

Now, consider the following matrix version of that simple DDE:

$$\dot{\mathbf{y}}(t) = \mathbf{A}\mathbf{y}(t-1) \tag{15.12}$$

For example, we could choose $\mathbf{A} = \mathbf{D} - \mathbf{I}$, where $\mathbf{D}$ is the differentiation matrix on the Chebyshev nodes $\tau = \cos(j\pi/(n-1))$, for $0 \le j \le n-1$. You can create this matrix by the general-purpose `genbarywts` routine used in this book (see Sect. 8.2.3), or more accurately by the command `gallery('chebspec',n)`. The eigenvalues of $\mathbf{D}$ are all 0, and so the eigenvalues of $\mathbf{A}$ are all -1. All of the nonlinear eigenvalues $W_k(-1)$ therefore have negative real part: the largest real part is -0.3181, approximately. There is *no doubt* that the asymptotic value of the solution $y(t)$ will be zero, irrespective of the initial history. For convenience, we take the initial history to be a vector of constants, say $y_k = \sin \pi \tau_k$, on $-1 \le t \le 1$.[2]

When we try to solve this equation using either `dde23` or `ddesd`, however, we get a nasty surprise. Say we use this code:

```
%
% Delay DE well-conditioned according to eigenvalues,
% seen to be ill-conditioned by pseudospectra
%
close all
clear all
n = 10;
% Differentiation matrix on tau = cos(pi*(0:n-1)/(n-1))
D2=gallery('chebspec',n);
e = eig(D2)-1;
figure
% Eigenvalues exactly -1, multiple
plot(e,'ko')
hold on
nu=linspace(-pi/2,pi/2,407);
% Envelope where Re(W(k,lambda)) < 0
plot(-nu.*sin(nu),nu.*cos(nu),'k')
axis([-1.75,0.25,-1,1])
axis('square')
% sin(pi*tau) initial vector, constant history
hist = dde23( @(t,y,z) zeros(size(y)), [], ...
              sin(pi*cos(pi*(0:n-1)/(n-1))), [-1,0]);
opts=ddeset('abstol',1.0e-12,'reltol',1.0e-12);
% y'(t) = (D2-I)*y(t-1)
chebspecdde = @(t,y,z) D2*z-z;
% dde23 takes 58,664 steps to get to 4...
% ddesd takes 4,676
Tf = 8;
sol=ddesd(chebspecdde,1,hist,[0,Tf],opts);
figure
```

[2] After this example was constructed, we realized that it could be interpreted as a method-of-lines solution to the delayed PDE $u_t(t,x) = u_x(t-1,x) - u(t-1,x)$. Replacing $u(t,x)$ by a vector of values $y_k(t) = u(t,\tau_k)$ on the grid $x = \tau_k = \cos \pi(k-1)/(n-1)$ for $1 \le k \le n$, then the x-derivative could be replaced by using the Chebyshev spectral differentiation matrix. We do not pursue this here, but simply point out that this kind of DDE isn't all that unrealistic.

```
semilogy(sol.x,abs(sol.y),'k.')
axis([0,Tf,1.0e-3,1.0e9])
% Residual is less than 1.0e-4 (1.0e-7*||y||)
h=diff(sol.x);
samp = [sol.x(1:end-1)+h/3;sol.x(1:end-1)+2*h/3];
t=samp(:);
k=find(t>0);
tt=t(k);
[y,yp]=deval(sol,tt);
z=deval(sol,tt-1);
r=zeros(size(z));
[dummy,m]=size(z);
nrms=zeros(size(z));
for i=1:m,
    r(:,i)=yp(:,i)-chebspecdde(tt(i),y(:,i),z(:,i));
    nrms(:,i)=norm(y(:,i),2)*ones(n,1);
end
figure
semilogy(tt,abs(r)./nrms,'k.')
set(gca,'fontsize',16);
xlabel('t','fontsize',16);
ylabel('relative␣residual','fontsize',16)
```

The important thing to understand here is that asymptotically stable *is not the same* as well-conditioned. The codes have some trouble with this apparently simple problem (at tight tolerances—they do all right at tolerances about $1.0 \cdot 10^{-7}$). They do manage to keep the residual small, apart from some narrow intervals where the residual spikes several orders of magnitude (see Fig. 15.9). The residuals do spike (when the tolerances are 10^{-12}), but the width of the intervals containing the spikes are about 10^{-8}. The step sizes in those intervals go down to about 10^{-14}, about as low as they could possibly go. But the method works. The solution is acceptable

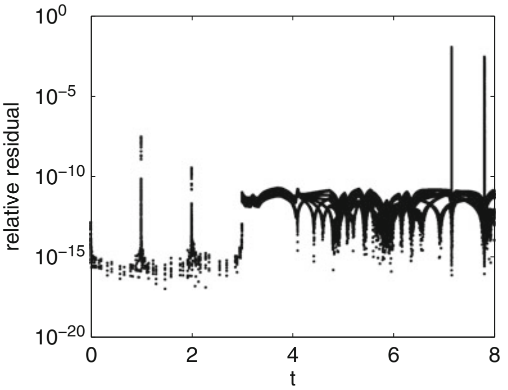


Fig. 15.9 The residual in the solution to (15.12) with tight relative tolerances, 10^{-12}, and then divided by the norm of the solution. Apart from narrow spikes (which we don't quite believe, as the step size is getting so small there—about 10^{-14}—that differentiation of the interpolant is dubious itself), the relative residual is everywhere small

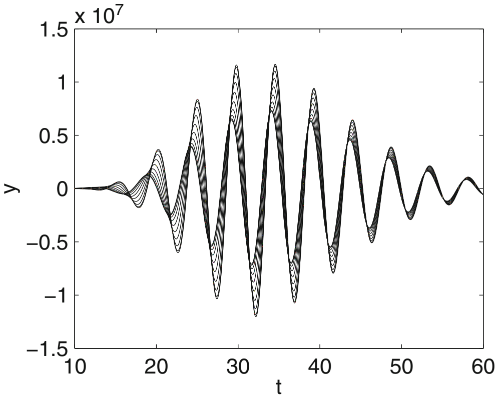


Fig. 15.10 The solution of (15.12) with relative tolerances 10^{-7}. In spite of having all nonlinear eigenvalues in the left half-plane, the solution grows very large first, before ultimately decaying as it must. The size of the "chirp" is exponential in the dimension n of the matrix (here $n = 10$, surely a modest-size problem)

(see Fig. 15.10). It's true that `dde23` has to work very hard in order to find the solution (the more sophisticated `ddesd` is more efficient at tighter tolerances). The ugly truth, though, is that the *reference* solution isn't very satisfactory! For $n = 10$, the reference solution grows as large as 1.2×10^7 before decaying. See also Fig. 15.14, where the DDE is solved another way. Yes, it is perfectly true that the reference solution $y(t) \to 0$ as $t \to \infty$; indeed, by $t = 1000$, the exact solution is very small. But for t between $t = 30$ and $t = 40$, the exact solution is oscillating between plus or minus 12 million. In the short term, this differential equation is very ill-conditioned.

Moreover, the solution to the reverse-engineered problem

$$\dot{\mathbf{y}}(t) = \mathbf{A}\mathbf{y}(t-1) + \varepsilon\mathbf{v}(t)\,, \tag{15.13}$$

unlike its unperturbed cousin, need *never* approach zero asymptotically—the persistent perturbation $v(t)$ from the numerical method might always excite the exponential growth modes. This is also a fact of life, not just of numerics. Physical perturbations of the original equation [or the delayed PDE we interpreted it as coming from, $u_t(t,x) = u_x(t-1,x) - u(t-1,x)$] would also excite these exponential growth modes.

This is worth considering in a little greater detail. By changing coordinates using the Schur factoring of $\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{U}^{\mathbf{H}}$, putting $\mathbf{q}(t) = \mathbf{U}\mathbf{y}(t)$, we can transform to a problem of the same type but with a triangular matrix. This reduces the problem to a sequence of scalar delay DE, and the standard conditioning theory says that if all the nonlinear eigenvalues $W_k(t_{jj})$ have negative real part, then the zero solution is asymptotically stable. But we have learned that even for initial-value problems for ordinary differential equations, this is not the full story if $\mathbf{A}$ is nonnormal (and this matrix $\mathbf{A} = \mathbf{D} - \mathbf{I}$ was chosen to exhibit nonnormality, of course). What matters in the nonnormal case is the *pseudospectrum* of $\mathbf{A}$, or more properly the *non-*

linear pseudospectrum of the nonlinear eigenvalues. The pseudospectral theorem (Theorem 5.3 from Chap. 5) applies here, with basis functions 1 and $\exp(-t)$. That is, the pseudospectrum of the nonlinear eigenvalues for (15.12) is the set

$$\Lambda_\varepsilon = \left\{ z \,\middle|\, \|(z\mathbf{I} - \exp(-z)\mathbf{A})^{-1}\| \geq (\varepsilon(\alpha_0|z| + \alpha_1|\exp(-z)|))^{-1} \right\} , \tag{15.14}$$

which contains the nonlinear eigenvalues $W_k(t_{jj})$, but may be substantially larger. Typically, the weights α_0 and α_1 would be taken to be 1 and $\|\mathbf{A}\|$. If any part of the pseudospectrum intrudes into $\mathrm{Re}(z) \geq 0$, for a physically realistic ε, then the problem must be considered ill-conditioned, because small perturbations of the right-hand side of the equation will *not* decay to zero but rather persist (and might even be amplified substantially). Rather than doing a full nonlinear pseudospectral analysis for this simple problem, we can look instead at the interaction between the ordinary pseudospectrum of $\mathbf{A}$ and the region of the z-plane for which $\mathrm{Re}(W_k(z)) < 0$, which is the teardrop inside the curve $(-v\sin v, v\cos v)$ for $-\pi \leq v \leq \pi$ (see Fig. 15.11).

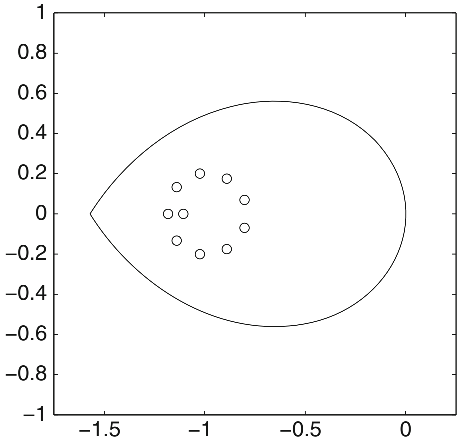


Fig. 15.11 The computed eigenvalues of $\mathbf{A}$ (when $n = 10$) all lie inside the teardrop of stability for $\dot{y}(t) = \lambda y(t-1)$. However, the reference eigenvalues of $\mathbf{A}$ are all identically -1—what we see here are the effects of perturbations about the size of machine epsilon, $2.2 \cdot 10^{-16}$. Perturbations about the size of 10^{-13} or so can bring λ right outside the teardrop, which makes the delay DE ill-conditioned

Keeping this in mind, the standard linear stability analysis, though limited, is still of some value. One often sees “sensitivity” studies of DDE models, for example. A lesson from numerical analysis is that, for nonnormal matrices, that’s not the whole story.

We conclude with one particular variation-of-constants formula from Baker et al. (2006). Consider the linear DDE system

$$\dot{\mathbf{y}}(t) = \mathbf{A}(t)\mathbf{y}(t) - \mathbf{B}(t)\mathbf{y}(t-\tau) + \varepsilon\mathbf{v}(t) , \tag{15.15}$$

for $t \geq 0$, subject to $\mathbf{y}(t) = \boldsymbol{\phi}(t)$ for $t \in [-\tau, 0]$. Then there exists a function $\mathbf{U}(t,s)$ (called a resolvent) such that

$$\mathbf{y}(t) = \mathbf{U}(t,0)\boldsymbol{\phi}(0) + \int_{-\tau}^{0} \mathbf{U}(t,s+\tau)\mathbf{B}(s+\tau)\boldsymbol{\phi}(s)\,ds + \varepsilon \int_{0}^{t} \mathbf{U}(t,s)\mathbf{v}(s)\,ds\,. \tag{15.16}$$

Inspection of this formula shows that it provides a way to bound or estimate the influence of perturbations $\varepsilon v(t)$ to the reference DDE. These perturbations can be numerical or physical. The size of $\mathbf{U}$ and its integral provide a kind of condition number for the delay DE. The pseudospectral results above tell us (a little more clearly than that theorem does) that in some cases $\mathbf{U}$ will grow very large, before decaying to zero eventually as $t - s \to \infty$; but also tell us that if $\mathbf{v}(s)$ never settles down, then the large size of $\mathbf{U}(t,s)$ will never lose its relevance.

15.4 Neutral Equations

There exist several numerical schemes to tackle so-called neutral DDE. These problems tend to be much less well-conditioned than even state-dependent RDDE, and so numerical methods need to be meticulously constructed. We here examine only one out of a wide variety: the artificial diffusion methods of Shampine (2008a). The key idea is to replace the neutral terms $\dot{y}(t-\tau)$ with backward difference approximations, transforming the neutral DDE into an RDDE, admittedly with at least two close time lags. Shampine reports some success with his code `ddeNsd`, and we give a simple example of his idea here.

Consider the neutral DDE

$$\dot{y}(t) = y(t) + \dot{y}(t-1)\,, \tag{15.17}$$

with initial history $y(t) = 1$ on $-1 \leq t \leq 0$. By the method of steps, this neutral DDE has a unique solution: Use the integrating factor $\exp(-t)$ to get $(\exp(-t)y(t))' = \exp(-t)\dot{y}(t-1)$, and on $0 \leq t \leq 1$ the right-hand side is known; integrate it to get $y(t)$ on that interval. Proceed from there to get $y(t)$ on $1 \leq t \leq 2$, and so on. This equation is used in Shampine (2009) as a test example for the method advocated there, namely, taking a finite-difference approximation to the neutral term $\dot{y}(t-1)$. That is, we solve the retarded two-delay equation

$$\dot{y}(t) = y(t) + \frac{y(t-1) - y(t-1-\delta)}{\delta} \tag{15.18}$$

and regard this as an approximate solution to (15.17). We take $\delta = 10^{-4}$, and call `dde23` as follows:

```
dde = @(t,y,z) y + (z(:,1)-z(:,2))/0.0001;
hist = dde23( @(t,y,z) 0, [], 1, [-1,0] );
sol2 = dde23( dde, [1,1.0001], hist, [0,4] );
```

Moreover, we compute the residual in the *original* equation as follows:

```
t = linspace(0,4,2001);
[y,yp] = deval( sol2, t );
[z,zp] = deval( sol2, t-1 );
res = yp - y - zp;
semilogy( t, abs(res./y), 'k' )
```

We have suppressed a warning message about computations at the primary discontinuity, $t = 0$. The scaled residual turns out to be uniformly less than about 10^{-3}, as can be seen in Fig. 15.12. This approach, while very simple, seems pragmatically effective, and verifiably so, for this kind of neutral DDE.

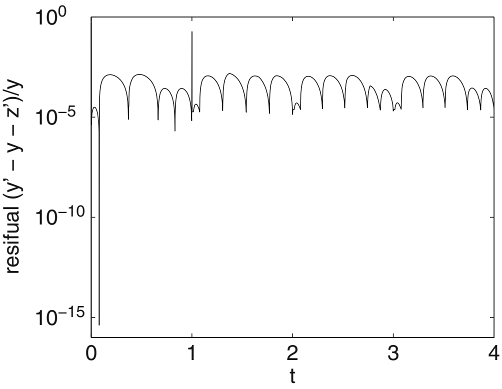


Fig. 15.12 The residual in the solution to the neutral DDE (15.17), divided by the norm of the solution. Apart from a single narrow spike at the primary discontinuity, the relative residual is everywhere small

15.5 Chebfun and Delay Differential Equations

The Chebfun package[3] can be used in a straightforward way to implement the method of steps to integrate simple delay differential equations (nearly) exactly. The method of steps, while simple, is indeed used in applications from time to time (see, e.g., Patwa and Wahl 2008). Consider the solution shown in Fig. 15.3, which has discontinuities in the history; `dde23` had a little difficulty resolving the discontinuities. We apply the method of steps using Chebfun to do the integrations for us, as follows (in a simple naive way):

```
%
% Solving y'(t) = -y(t-1)/5 with
% y(t) = (-1)^floor(-4t) on -1 <= t <= 0
% By the method of steps with chebfun
```

[3] See the original paper by Battles and Trefethen (2004) and the literature and code available at www.chebfun.org.

```
%
% RMC January 2011
clear all
close all
ex = chebfun( 'x', [-1,0] )
hist = (-1).^floor( -4*ex )
y0 = cumsum(-hist/5) + feval(hist,0)
y1 = cumsum( -y0/5 ) + feval(y0,0)
y2 = cumsum( -y1/5 ) + feval(y1,0)
y3 = cumsum( -y2/5 ) + feval(y2,0)
y4 = cumsum( -y3/5 ) + feval(y3,0)
z = [hist;y0;y1;y2;y3;y4]
figure
plot( z, 'k' )
hold on
xlabel('t','fontsize',16)
ylabel('y␣and␣y''','fontsize',16)
plot( diff(z), 'k' )
axis([-1,5,-1.5,1.5])
set(gca,'fontsize',16);
```

The results, plotted in Fig. 15.13, are very satisfactory: The discontinuities are well resolved, the plotting is simple, and the residual (not shown) is tiny. One disadvantage is that there is (as far as we know) as yet no general-purpose DDE solver using chebfuns—although Driscoll (2010) reports a method for solving integral and integro-differential equations that might do nicely instead. We do not take up that method here but content ourselves with exploring simple uses of Chebfun. The Chebfun package is under active development, and we expect to see interesting improvements in the near future.

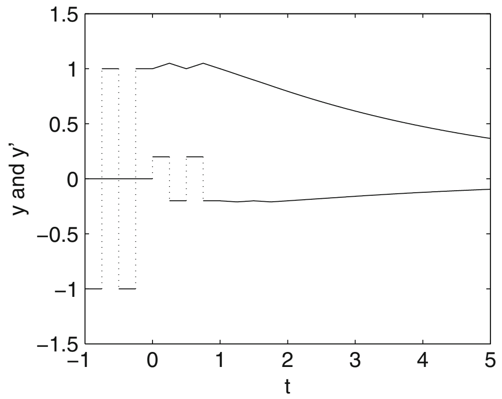


Fig. 15.13 The solution to $y_t(t) = -y(t-1)/5$, with a history that has jump discontinuities, using Chebfun. Compare Fig. 15.3

What if we have, not a scalar equation, but a matrix equation, such as (15.12)? Using a cell array of chebfuns, and (as before) the cumsum function which is overloaded in the Chebfun package to do quadrature exactly (apart from rounding errors) on chebfuns, the exact solution to (15.12) is simple to compute. Consider this code:

```
%
% Delay DE well-conditioned according to eigenvalues,
% seen to be ill-conditioned by pseudospectra,
% Solved (almost) exactly by using the method of steps in chebfun.
%
% RMC January 2011
%
close all
clear all
n = 10;
% Differentiation matrix on tau = cos(pi*(0:n-1)/(n-1))
D2=gallery('chebspec',n);
% Now solve the DDE y'(t) = (D2-I)y(t-1) by the method of steps
% almost by hand, using chebfun.
% sin(pi*tau) initial vector, constant history
hist = sin(pi*cos(pi*(0:n-1)/(n-1)));
one = chebfun( 1, [-1,0] );
hist0 = diag(hist)*repmat( one', n, 1 );
% Go out to t=m
m = 31 + 1;
% A cell array of chebfuns makes it simple
y = cell(m,1);
y{1} = hist0;
z = y{1};
for i=2:m,
    % Since the DDE contains no y, we need only integrate each step
    y{i} = cumsum( D2*y{i-1} - y{i-1} ) + diag(feval(y{i-1},0))*repmat(one',n,1);
    z = horzcat(z,y{i});
end
figure
plot( z, 'k' )
xlabel('t','fontsize',16)
ylabel('y','fontsize',16)
axis([10,26,-1.0e7,1.0e7])
set(gca,'fontsize',16);
% We see that whilst the zero solution is ultimately stable,
% we go there the lazy way, and the largest value of y
% is exponentially large in n.
% Asymptotically stable is NOT THE SAME as well-conditioned.
```

This code produces the solution in Fig. 15.14. As we understand it, there is current work under way to make available some automatic facilities for solving delay differential equations in the chebop subsystem of the Chebfun package; as stated before, in some sense, this has been done already for integral equations and integro-differential equations in Driscoll (2010). This example shows a semimanual attempt, which again has quite satisfactory results. Admittedly, this is a very simple type of equation, with only one constant lag, and moreover each step is just a quadrature.

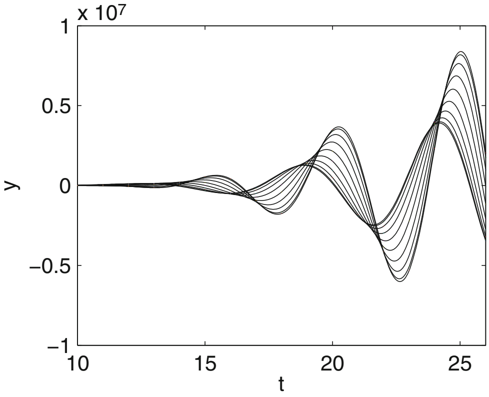


Fig. 15.14 A portion of the solution to (15.12) with $n = 10$ by the method of steps in Chebfun

15.6 Notes and References

Examples of DDEs in application and a discussion of some of the many kinds of such equations that are used can be found in, for instance, Bocharov and Rihan (2000).

The details of the MATLAB routine dde23 and ddesd are described in Shampine (2005) and Balachandran et al. (2009 Chapter 9); some of the examples studied here are taken from Shampine et al. (2003).

There are many books on delay differential equations, from the classic Bellman and Cooke (1963) through the numerical work Bellen and Zennaro (2003) to the latest (as of this writing) Balachandran et al. (2009). See, in particular, Chapter 9 of that last, by Shampine and Thompson, who there update the chapter in Shampine et al. (2003) on delay differential equations. We have used the MATLAB codes as exemplars of existing high-quality DDE solvers. There are many such solvers. For large-scale problems, you may wish to use a code designed for scientific computing, not a problem-solving environment. Several such codes are described in Shampine (2005) and in Baker et al. (2005). Most are available for free for academic use.

In Sect. 15.1, we discussed an example of Cooke et al. (1999). One of their published numerical solutions was in error; this was later noticed by Shampine et al. (2003). The technique advocated here, namely, plotting the residual, would have detected the error.

DDE are not just of the types presented here. One extremely useful generalization is to allow the "past history" of the solution to play a role not just at individual, discrete past times, but rather as a weighted average. This leads to integro-differential equations, and in particular to Volterra integro-differential equations. In some cases, when the kernels can be broken up into a discrete sum of products, in which case the kernel is termed *degenerate*, these can be reduced to simpler DDE such as the ones considered here. But this trick is not by any means general, and Volterra integral equations form a useful part of the modeler's arsenal. Consult, for example, Brunner (2004) for a thorough discussion of methods appropriate for such problems.

The literature on the sensitivity or “stability” of DDE is very large. See, for example, Baker et al. (2005), Wu et al. (2010), Rihan (2003), or Rihan (2006).

Considering another topic we emphasized, see Jarlebring and Damm (2007) and Michiels and Niculescu (2007) for a discussion of pseudospectra in the solution of delay differential equations. For more on our remark that what matters in the nonnormal case is the *nonlinear pseudospectrum* of the nonlinear eigenvalues, see Michiels and Niculescu (2007); it contains an extended discussion of this issue, and many applications.

Problems

Theory and Practice

15.1. Solve the simple constant-delay differential equation $\dot{x}(t) = -\frac{\pi}{2}x(t-1)$ with initial history $x(t) = \phi(t) = 1$ for $-1 \leq t \leq 0$. Solve it again with history $\phi(t) = \sin(\pi t)$. Comment.

15.2. Suppose you wish to write a Runge–Kutta based method for solving the DDE $\dot{y}(t) = f(t, y(t), y(t-\tau))$, and that τ is very small. Show that the nominally second-order Runge–Kutta method given by $k_1 = f(t_n, y_n, y(t_n - \tau))$ and $k_2 = f(t_n + h, y(t_n) + hk_1, y(t_n + h - \tau))$ will, in general, need to evaluate $y(t)$ somewhere in $(t_n, t_n + h)$, and that predicting this value just by using an Euler step from (t_n, y_n) gives an $O(h)$ method, not an $O(h^2)$ method as would be desired. This is why standard codes predict the value of $y(t_n + h - \tau)$ and then iterate to find a good value for this, prior to proceeding to the next step.

Investigations and Projects

15.3. The solutions to our favorite IVP for ODE, namely,

$$\dot{x}(t) = \cos(\pi t x(t)) \,, \tag{15.19}$$

give nice pictures on the square $0 \leq x \leq 5$, $0 \leq t \leq 5$, when the solutions from several initial conditions are plotted at once, as you saw in a previous problem. Now use `dde23` to solve

$$\dot{z}(t) = \cos\left(\pi t z(t-\tau)\right) \,, \tag{15.20}$$

with history provided by $x(t)$ (i.e., the solution to (15.19) integrated backward on the interval $[-\tau, 0]$). Use $x(0) = 0:0.2:5$ and a relative tolerance 10^{-9} throughout. Solve the problem three times, with $\tau = 0.01$, $\tau = 0.1$, and $\tau = 0.2$, and plot the results, including the history, in three separate graphs. Compare the three plots with the solution of (15.19) on $[0,5]$. Does introducing a delay cause any instability? Why can trajectories cross each other in the delayed case, but not in the undelayed case?

15.4. Adapt the Chebfun method of steps program given above to solve the equation

$$\dot{w}(t) = -\nu w(t) + \alpha f(w(t-\tau)) \tag{15.21}$$

where α, ν, and τ are constants, $f(x) = \exp(x) - 1$, and $w(t)$ has, say, constant history. You can do this entirely with `cumsum` if you supply an integrating factor.

Chapter 16
Numerical Solution of PDEs

Abstract We look at the *method of lines* using standard initial-value problem (IVP) software for *stiff problems*. Both *spectral methods* and *compact finite differences* are used for the spatial derivatives. We look briefly at the *transverse method of lines*, which instead uses standard boundary value problem (BVP) software that has *automatic mesh selection*. We also briefly consider *Fourier transform methods* for Poisson's equation. ◁

The solution of partial differential equations (PDEs) is perhaps the most heavily studied subject in numerical analysis. There are whole books devoted to individual aspects of just one class of problem. This present chapter is not going to do more than paint a picture of a scratch on the surface of the subject. Still, there are some simple things to say that are quite useful; some of the concepts and techniques we talk about here fit very well in a numerical analyst's toolbox. Given the amount of time we have invested in the solution of ODEs in this book, we believe that the most benefit from the least effort will occur if we mostly work with what are called *semidiscretizations* in the literature: This will lead naturally to solving systems of IVPs, or (in the transverse method of lines case) of BVPs, for ODEs.

Also, in keeping with the theme of this book so far, we will concentrate on methods that provide continuous solutions, for which we may compute a residual; thus, here, as elsewhere in the book, a computed solution of an equation will be the exact solution of a different, reverse-engineered, PDE. Boundary conditions will often be solved exactly (up to roundoff). If the reverse-engineered PDE is close to the reference PDE, then we say that the algorithm that computed our solution was numerically stable. As always, we will have to worry about whether or not the PDE is sensitive to changes or not, that is, ill-conditioned or not.

However, there is a catch, which happens to be the reason why residual-based methods are always the *second* approach to a problem: cost. For the second explicit time in the book, we have the possibility of encountering a "large" computational problem, because the cost of solving PDEs is exponential in the dimension

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_16,

(number of variables) of the problem.[1] For a truly large problem, you won't want to go to the expense of constructing a continuous solution, or to the expense of computing a residual to give an a posteriori analysis of the error. Or, even if you want to, you may not have the computing resources or the time to do so. This consideration will factor into some of our analysis, and although we are confident that computing resources will increase in power over the lifetime of this book, we suspect that the size of the problems being solved will keep pace, so that this cost will always be a concern. But the ideas of backward error and of conditioning will prove helpful nonetheless.

16.1 The Method of Lines

16.1.1 The Method of Lines Using Spectrally Accurate Spatial Mesh

In what follows, we show how to use the method of lines by example.

Example 16.1. Let us consider a motivating example problem, namely, the advection–diffusion equation for the unknown function $u(t,x)$ satisfying

$$u_t + au_x = \nu u_{xx}, \tag{16.1}$$

for all $t > 0$ and $-1 \le x \le 1$, subject to the initial condition $u(0,x) = \cos^2(\pi x/2)$ and $u(t,-1) = u(t,1) = 0$. The real parameters a and ν we take to be 1 and $1/100$, respectively. We solve this equation first by the venerable *method of lines*, although we use a spectrally accurate spatial mesh: Here we choose a grid of x-values, say $x_j = \cos(\pi j/(n+1))$ for $0 \le j \le n+1$, and note that $x_0 = 1$ and $x_{n+1} = -1$. We then look at $y_j(t) = u(t,x_j)$ and see that the boundary conditions imply that $y_0(t) = 0$ and $y_{n+1}(t) = 0$, for all time (this PDE is an easy problem). If we replace u_x with $\mathbf{D}y$, and u_{xx} with $\mathbf{D}^2 y$, where $\mathbf{D}$ is the differentiation matrix for polynomials on the Chebyshev points used here, then the PDE is replaced by the *system* of ODEs

$$\frac{d\mathbf{y}}{dt} = -\mathbf{D}\mathbf{y} + \frac{1}{100}\mathbf{D}^2\mathbf{y}\,, \tag{16.2}$$

subject to $y_j(0) = \cos^2 \pi x_j/2$. We solve this IVP using, say, `ode15s`. As is often the case for the method of lines, the system is stiff, making the use of a stiff solver appropriate. We integrate on $0 \le t \le 2.5$, with a code such as this:

```
function sol = advectionmol(a, nu, n)
%
% Advection-diffusion equation by the method of lines
% u_t = a u_x + nu u_xx
D = gallery('chebspec',n+2);
```

[1] We saw this possibility for the first time with large dense linear systems. The cost there wasn't exponential in the dimension, but it was very large.

```
x = cos( pi*(0:n+1)/(n+1) );
function dy = mol(t,y)
    my = [0; y; 0];
    tmp = -a*D*my + nu*D*D*my;
    dy = tmp(2:end-1);
end
inits = cos(pi*x/2).^2.*exp(-(0.0e-2).*x.^2./(1-x.^2));
% inits = exp(-100*x.^2);
tf = 2.5;
opts = odeset('RelTol',1.0e-3,'AbsTol',1.0e-6);
sol = ode15s( @mol, [0,tf], inits(2:end-1)', opts );
figure(1)
mesh( x, sol.x, [zeros(size(sol.x));sol.y;zeros(size(sol.x))]' ),
    ...
    view(160,50),colormap([0,0,0]), axis([-1,1,0,tf,0,1]);
end
```

The results are plotted in Fig. 16.1. ◁

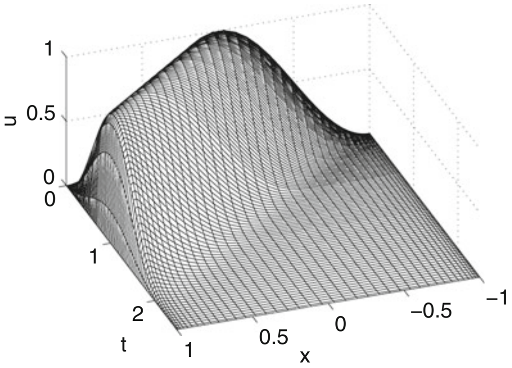


Fig. 16.1 An approximate solution to Eq. (16.1) with $a = 1$, $\nu = {}^1\!/_{100}$, $u(0,x) = \cos^2(\pi x/2)$ on $0 \le t \le 2.5$. The solution uses 40 fixed spatial mesh points (not counting the boundaries) and default time-integration tolerances. The view is such that time increases as your eye moves out of the page—the wave moves to your left. This view was chosen to show the steep drop as the wave encounters the boundary condition $u(t,1) = 0$. Notice that initially the time steps, chosen automatically by the solver, are very close together, but as the integration proceeds, the (stiff) integrator ode15s finds the problem progressively easier

How can we compute a residual? Better yet, can we be assured that the residual is small *without* computing it? In the PDE case, there is a lot more data to be concerned with, and so while, in principle, we may compute the residual anywhere we like, because we may interpolate the data accurately and differentiate the interpolants as before, there are a lot of such places to choose from (this is the cost issue that we brought up earlier). Let us just sample the residual at a few places, first, as follows:

```
function res = residadvectionmol(a,nu,sol,n,t0,tf,nr,scale)
x = cos( pi*(0:n+1)/(n+1) );
D = gallery('chebspec',n+2);
```

```
j0 = find(sol.x >= t0 );
jf = find(sol.x <= tf );
t = RefineMesh(sol.x(j0(1):jf(end)),nr);
em = length(t);
[u,ut] = deval( sol, t );
ux = D*[zeros(1,size(u,2));u;zeros(1,size(u,2))];
xi = RefineMesh(x,nr);
en = length(xi);
res = zeros(en,em);
w = genbarywts( x, 1 );
for i=2:em,
  [uxi, uxxi] = hermiteval( [0;u(:,i);0], xi, x, 1, w, D );
  utxi = hermiteval( [0;ut(:,i);0], xi, x, 1, w, D);
  [uxxi2, uxxxi] = hermiteval( ux(:,i), xi, x, 1, w, D );
  res(:,i) = utxi + a*uxxi - nu*uxxxi;
end
figure(2)
colormap([0,0,0])
mesh( t, xi(2:end-1), res(2:end-1,:) ) , axis([t0,tf, -1, 1, -
    scale,scale]), view(70,60)
```

The code produces the graphs in Figs. 16.2 and 16.3.

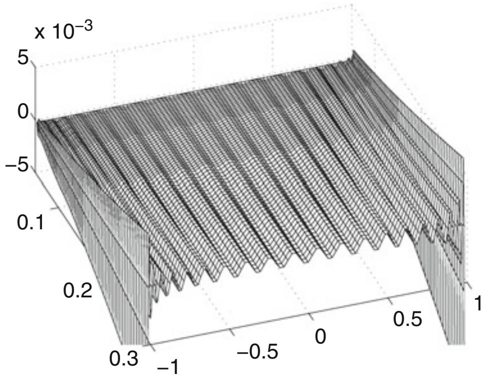


Fig. 16.2 A computed residual of an approximate solution to (16.1) with $a = 1$, $\nu = 1/100$, $u(0,x) = \cos^2(\pi x/2)$, sampled on $0.05 \le t \le 0.3$. The solution uses 40 mesh points in space and default time-integration tolerances. This residual is an overestimate of the true residual and is particularly unreliable near the edges

Remark 16.1. The interpolant in `ode15s` isn't quite up to the job for which we need it here: It is not continuously differentiable, and it gives a spuriously large residual in the very small steps that the code takes initially. In the graphs that follow, we trim that initial interval out. This is justifiable because that very small time interval allows only for a very small impulse in any event, even if the computed residual was a sharp bound. A proper cure involves the use of better interpolants, however. ◁

With this computation of the residual, we can say that we have the *exact* solution of the reverse-engineered equation

$$u_t = au_x + \nu u_{xx} + 5 \cdot 10^{-3} v(t,x)\,, \tag{16.3}$$

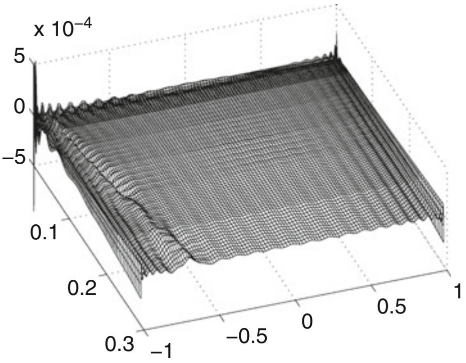


Fig. 16.3 A computed residual of an approximate solution to (16.1) with $a = 1$, $\nu = {}^1\!/_{100}$, $u(0,x) = \cos^2(\pi x/2)$, sampled on $0.005 \leq t \leq 0.5$. The solution this time uses 50 spatial mesh points and default time-integration tolerances. Notice that the residual is substantially smaller than the residual for $n = 40$ in Fig. 16.2. This residual is again an overestimate of the true residual, but since merely adding 10 spatial mesh points increased the accuracy by more than a factor of 10, we suspect that convergence here is spectrally fast. The command was `res = residadvectionmol(a,nu,sol,n,0.005,0.5,3,5.0e-4);`

where $v(t,x)$, while wavy, is less than 1 in magnitude. Notice that to compute the residual in Figs. 16.2 and 16.3, we used the differentiation matrix **D**, but this provided exact derivatives because $u(t,x)$ as computed is a piecewise polynomial in t (whose t-derivative was exactly computed by `deval`), but a global polynomial (of degree 41 and 51, respectively) in x, whose x-derivative is thus computed exactly by using **D**. Notice also the dramatic reduction in the size of the residual by moving from $n = 40$ to $n = 50$. This is 10 times the (residual) accuracy, for only 25% more work—this is the kind of thing that occurs with spectral methods. We thus have quite a bit of confidence that our solution is accurate, for this problem. In fact, the solution is even more accurate than it seems—if we used a better interpolant, we would be able to see that.[2] But this is enough to convince us that the process converges (even if it does have a bit of difficulty with the layer at the right endpoint and with the start, on coarser grids). As Brian Wetton remarked in a talk, “Everybody does the Burgers equation, mainly because it’s not as hard as it looks.”

We will now have a brief look at a classical nonlinear problem.

[2] One way to see that the $n = 40$ solution is more accurate than $5 \cdot 10^{-3}$ is to compare it with the $n = 80$ solution. They differ, at the nodes, by less than about $5 \cdot 10^{-5}$. Hence, we see that the $n = 40$ mesh produces better solutions, at the nodes, than we can measure in the fashion we have advocated here.

Example 16.2. Consider Burgers' equation,

$$u_t + \frac{1}{2}\left(u^2\right)_x = 0\,, \tag{16.4}$$

with *periodic* boundary conditions, $u(t,-1) = u(t,1)$, and initial condition $u(0,x) = \exp(i\pi x)$. This is a complex-valued function. Although it seems unlikely that `ode15s` would integrate complex-valued functions (even though t and x are both real), it can (see the exercises). But in order to exhibit the solution of a two-variable problem, we separate $u(t,x) = \sigma(t,x) + i\tau(t,x)$ and rewrite the equation as

$$\begin{bmatrix} \sigma_t \\ \tau_t \end{bmatrix} = -\begin{bmatrix} \sigma & -\tau \\ \tau & \sigma \end{bmatrix} \begin{bmatrix} \sigma_x \\ \tau_x \end{bmatrix}, \tag{16.5}$$

with initial conditions $\sigma(0,x) = \cos(x)$ and $\tau(0,x) = \sin(x)$. Some code to solve this by the method of lines is given next, showing how to handle a two-variable problem by the method of lines:

```
function [ x, sol, sig, ta ] = cburgermol( n )
%CBURGERMOL Solve complex Burger's eqn
%   u_t + u u_x = 0
%   -1 <= x <= 1, u(t,-1) = u(t,1)
%   u(0,x) = exp( i*pi*x )
%   JAC Weideman's solution u(t,x) = W( i*pi*t*exp(i*pi*x) )/(i*
    pi*t)
%   t > 0, W = LambertW.  Singular at x=1/2, t=1/pi/e.
%
%   u = sigma + i*tau gives
%
%  [ sigma_t ] = - [ sigma  -tau  ][ sigma_x ]
%  [   tau_t  ]     [  tau   sigma ][  tau_x  ]
x = cos( pi*(0:n)/n );
D = gallery('chebspec', n+1 );
inif = [ cos(pi*x), sin(pi*x) ];
% Spectral space derivatives
% Enforce periodic boundary conditions by using the mean
% of the left and right end-values to advance the left and
% right end-values.
    function yp = cb(t,y)
        yp = zeros(size(y));
        sigma = y(1:n+1,:);
        tau   = y(n+2:end,:);
        sp = (sigma(1,:)+sigma(n+1,:))/2;
        sigma(1,:)=sp;
        sigma(n+1,:)=sp;
        tp = (tau(1,:)+tau(n+1,:))/2;
        tau(1,:)=tp;
        tau(n+1,:)=tp;
        sx = D*sigma;
        tx = D*tau;
        yp(1:n+1,:) = -(sigma.*sx - tau.*tx);
        yp(n+2:end,:) = -(tau.*sx + sigma.*tx);
    end

```

```
opts = odeset('reltol',1.0e-6);
ts = 1/pi/exp(1);
sol = ode15s(@cb,[0,ts], inif, opts );

% Postprocess solution for visualization
sig = sol.y(1:n+1,:);
ta = sol.y(n+2:end,:);
figure
mesh(x,sol.x,sig');
xlabel('x','fontsize',16);
ylabel('t','fontsize',16);
zlabel('real','fontsize',16);
set(gca,'fontsize',16);
axis([-1,1,0,ts,-1,1]);
view(170,40);
figure
mesh(x,sol.x,ta');
xlabel('x','fontsize',16);
ylabel('t','fontsize',16);
zlabel('imag','fontsize',16);
set(gca,'fontsize',16);
axis([-1,1,0,ts,-1,3]);
end
```

Using $n = 80$, the results are plotted in Figs. 16.4, 16.5.

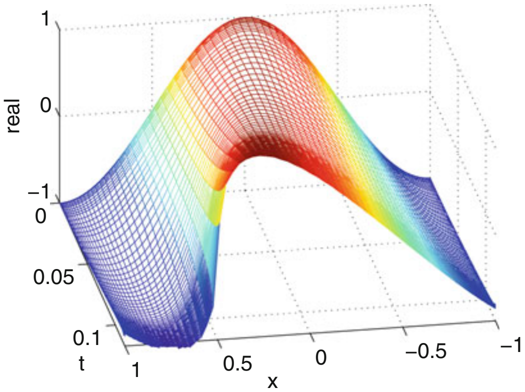


Fig. 16.4 Real part $\sigma(t,x)$ of $u(t,x)$ solving (16.4) with periodic boundary conditions and initial condition $u(0,x) = \exp(i\pi x)$. The wave breaks at $x = {}^1\!/_2$ and $t = {}^1\!/_{(e\pi)}$

Interestingly, Weideman (2003) points out that for this initial condition, the nonlinear equation can be solved exactly in terms of the Lambert W function, as follows. Any traveling wave solution must look like $u = f(x - ut)$, because the wave speed is u (the height of the wave), by analogy with $u_t + cu_x = 0$. But the initial shape of the wave is $u = f(x) = \exp(i\pi x)$, and so $u = \exp(i\pi(x - ut))$. This can be solved in terms of W: $u\exp(i\pi ut) = \exp(i\pi x)$, or $i\pi ut\exp(i\pi ut) = i\pi t\exp(i\pi x)$, whence $u = W_k(i\pi t\exp(i\pi x))/(i\pi t)$, for some branch k of W. But if $u(t,x)$ is to tend to $\exp(i\pi x)$ as $t \to 0$, then only the $k = 0$ branch will work, as all the others are singular at $t = 0$. Thus,

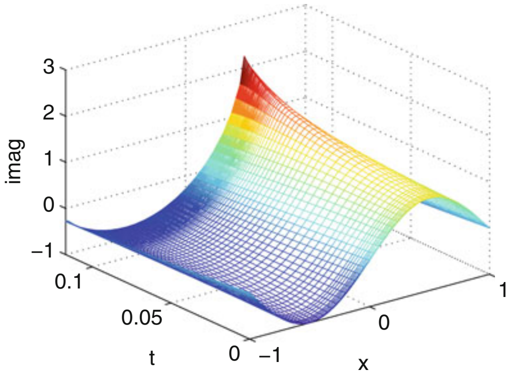


Fig. 16.5 Imaginary part $\tau(t,x)$ of $u(t,x)$ solving (16.4) with periodic boundary conditions and initial condition $u(0,x) = \exp(i\pi x)$. The wave peaks at $x = 1/2$ and $t = 1/(e\pi)$

$$u(t,x) = \frac{1}{i\pi t} W\left(i\pi t e^{i\pi x}\right) \tag{16.6}$$

for $t > 0$ solves the equation. This function is singular when the argument to W is equal to $-1/e$, the branch point. This happens if $-\exp(-1) = i\pi t \exp(i\pi x)$, or $-1 + i\pi = \ln(i\pi t) + i\pi x$, or $x = 1/2$ and $t = 1/(e\pi)$. When we solve the equation numerically, we find indeed that the waves break (in the real case) and peak (in the imaginary case) at this location and time. Trying to integrate past that just reveals instability, in the form of rapid oscillations. (Of course, that must be unstable: An infinite slope cannot be represented by a polynomial.) ◁

These results were obtained by finding values of $u(t,x_k)$ where the x_k were the Chebyshev points. What happens if instead we use equally spaced mesh points $x =$ `linspace(-1,1,n+2)`? Disaster! The residuals are nowhere near as small this time and show the typical equi-spaced interpolation error behavior: That is, the errors at the ends of the interval are much the largest. The problem is that interpolation by a high-degree polynomial at equally spaced points, as usual, introduces this huge error near the ends. This induces an instability into the solution, and we cannot proceed much past $t = 0.2$ with $n = 30$; and increasing n gets us nowhere (see the exercises). We now leave spectral methods for the moment, and investigate an alternative that is sometimes of interest: compact finite differences.

16.1.2 Compact Finite Differences and the Method of Lines

Let us give up on global high-degree polynomial interpolation in space, in order to work with the convenient equally spaced nodes $x_k = -1 + 2k/(n+1)$, for $0 \le k \le n+1$. Equally spaced meshes are so natural that we'd like to find a method that could work with them. There are many approaches that are satisfactory. We will use a variation of the Padé scheme discussed in Chap. 11, namely,

$$\frac{1}{6}f'(x_{k-1}) + \frac{2}{3}f'(x_k) + \frac{1}{6}f'(x_{k+1}) = \frac{1}{h}\left(-\frac{1}{2}f(x_{k-1}) + \frac{1}{2}f(x_{k+1})\right) \tag{16.7}$$

(here $h = 2/(n+1)$), which is to hold at each interior node x_k for $1 \leq k \leq n$. To deal with the x-derivatives at x_0 and x_{n+1}, we need to do something special. As discussed in Chap. 11, just after Eq. (11.88), we can create special formulæ of the same order and similar compactness. This leaves us with a tridiagonal system of equations to solve for the vector of derivatives, given the vector of values. This in effect computes a differentiation matrix $\mathbf{D} = \mathbf{M}_1^{-1}\mathbf{B}$, although of course we do not form $\mathbf{M}_1^{-1}$ explicitly. Since solving a tridiagonal system is $O(n)$ in cost, this is actually cheaper than multiplying by $\mathbf{D}$ above—not that cost is our primary concern here. Similarly, we can construct compact formulæ for *second* derivatives: Again, we can ensure fourth order of accuracy in h with a tridiagonal system. Being ambitious, we choose to use formulæ that allow us to factor the matrices $\mathbf{M}_1$ and $\mathbf{M}_2$ *analytically*, which speeds up the computation even more (see the exercises). The procedures are described in Algorithms 16.1 and 16.2.

Algorithm 16.1 A special fourth-order compact finite-difference scheme for the first derivative on an equally spaced grid. Implemented as `first.m`

```
Require: a vector $u$ of length $n$, and a mesh width $h$
  $c := 2+\sqrt{3}$
  $\alpha = 1/c$ {Finite differences on the function values for the right-hand side}
  $b(1) := ((1-3c)u(5)+(16c-6)u(4)+(18-36c)u(3)+(-10+48c)u(2)-(25c+3)u(1))/h/12$
  for $i$ from 2 by 1 to $n-1$ do
    $b(i) := 3(u(i+1)-u(i-1))/h$
  end for
  $b(n) := (103u(n)-182u(n-1)+126u(n-2)-58u(n-3)+11u(n-4))/12/h$
  {Forward elimination of the pre-factored tridiagonal system}
  $y(1) := b(1)$
  for $i$ from 2 by 1 to $n-1$ do
    $y(i) := b(i)-\alpha y(i-1)$
  end for
  {Back substitution}
  $u_x(n) := \alpha y(n)$
  for $i$ from $n-1$ by $-1$ to 1 do
    $u_x(i) := \alpha(y(i)-u_x(i+1))$
  end for
  return The vector $\mathbf{u_x}$ approximating $\partial u/\partial x$.
```

The code is as follows (not showing `first.m` or `second.m`, which you will write in the exercises):

```
function sol = compactadvectionmol(a,nu,n)
%
% Advection-diffusion equation by the method of lines
% u_t = a u_x + nu u_xx
x = linspace(-1,1,n+2);
h = x(2)-x(1);
function dy = mol(t,y)
```

Algorithm 16.2 A special fourth-order compact finite-difference scheme for the second derivative on an equally spaced grid. Implemented as `second.m`

```
Require: a vector $u$ of length $n$, and a mesh width $h$
  $c := 5+2\sqrt{6}$
  $\alpha = 1/c$
  {Finite differences on the function values for the right-hand side}
  $b(1) := ((45c+10)u(1) - (154c+15)u(2) + (214c-4)u(3) - (156c-14)u(4) + (61c-6)u(5) - (10c-1)u(6))/(12h^2)$
  for $i$ from 2 by 1 to $n-1$ do
    $b(i) := 12(u(i+1) - 2u(i) + u(i-1))/h^2$
  end for
  $b(n) := (-99u(n-5) + 604u(n-4) - 1546u(n-3) + 2136u(n-2) - 1555u(n-1) + 460u(n))/(12h^2)$
  {Forward elimination of the pre-factored tridiagonal system}
  $y(1) := b(1)$
  for $i$ from 2 by 1 to $n-1$ do
    $y(i) := b(i) - \alpha y(i-1)$
  end for
  {Back substitution}
  $u_x(n) := \alpha y(n)$
  for $i$ from $n-1$ by $-1$ to 1 do
    $u_x(i) := \alpha\,(y(i) - u_x(i+1))$
  end for
  return The vector $\mathbf{u_x x}$ approximating $\partial^2 u/\partial x^2$.
```

```
    ux = first( y, h );
    uxx = second( y, h );
    dy = [0; -a*ux(2:end-1) + nu*uxx(2:end-1); 0];
end
inits = cos(pi*x/2).^2;
inits(1) = 0;
inits(end) = 0;
tf = 2.5;
sol = ode15s( @mol, [0,tf], inits' );
figure(4),
mesh( x, sol.x, sol.y' ), view(160,50),colormap([0,0,0]), axis
    ([-1,1,0,tf,0,1]);
end
```

This turns out to work satisfactorily; it's not spectrally accurate, but not bad either. The difference between the results of the two approaches, sampled at $t = 1/2$, for $n = 80$, is plotted in Fig. 16.6.

Encouraged by this result, we may try one more example.

Example 16.3. Consider the one-way wave example treated in Trefethen (2000) and again in Shampine et al. (2003), namely,

$$u_t = -\left(\frac{1}{5} + \sin^2(x-1)\right) u_x\,, \tag{16.8}$$

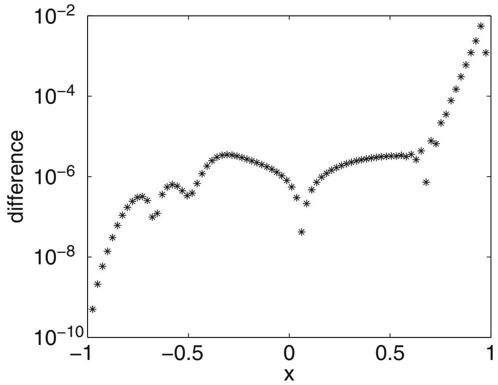


Fig. 16.6 The difference between two approximate solutions to (16.1) with $a = 1$, $\nu = {}^{1}/_{100}$, $u(0,x) = \cos^2(\pi x/2)$, sampled on $t = 0.5$. The spectral solution was interpolated with its global polynomial in x and the resulting polynomial evaluated at the equally spaced points used by the compact method, and the results subtracted. As you can see, the two solutions agree quite well outside the layer at the right edge

with initial condition $u(x,0) = \exp(-(x-1)^2/100)$, on $0 \leq x \leq 2\pi$. Again, we take $u(0,t) = u(2\pi,t) = 0$ although it isn't quite consistent at the corners. We use the MATLAB function

```
function yp = onewaywave( t, y )
%
% Method of lines (compact spatial derivative) for the
% linear wave equation
% Ut = -(1/5+sin^2(x-1))Ux
% u(0,t) = u(2pi,t) = 0
% y =u, dimension (n-1),
% x0 = 0, x1 = 2pi/n, ..., x[n-1] = 2pi(1 - 1/n), x[n] = 2pi.
%
m = length((y));
x = linspace(0,2*pi,m);
n = m-1;
h = x(2)-x(1);
ux = first( y, h );
up = -(1/5+sin(x-1).^2).*ux';
yp = [0, up(2:n), 0]';
```

and execute the following commands:

```
t=0:0.3:8;
n=256;
x=linspace(0,2*pi,n+1);
u0=exp(-100*(x-1).^2);
[t,slices]=ode23(@onewaywave,t,u0);
mesh(x,t,slices),view(10,70),axis([0,2*pi,0,8,0,2]),colormap
    ([0,0,0])
set(gca,'fontsize',16);
```

The results are plotted in Fig. 16.7. Curiously enough, `ode23` does a better job than `ode15s` does on this problem, at default tolerances (see the exercises). ◁

It should be quite clear that these methods are not restricted to linear problems. Any differential equation of the form $u_t = f(t,u,u_x,u_{xx})$ can be solved by the method of lines in this fashion, on a fixed spatial grid of sufficient fineness. With more effort, one can evaluate a residual and verify that the computed solution, say $U(t,x)$, satisfies $U_t = f(t,U,U_x,U_{xx}) + \varepsilon v(t,x)$ for some $v(t,x)$ that is uniformly bounded, and where ε can be made as small as one likes, with sufficient computing resources thrown at the problem.

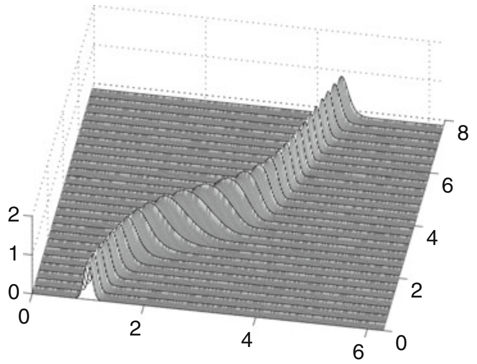


Fig. 16.7 The solution to the one-way wave Eq. (16.8) by compact finite differences in space, using `ode23` to do the time integration and $n = 256$ equally spaced points in the x-direction

16.2 Conditioning of PDEs

This raises, yet again in this book, the natural question of what happens to the solution of the equation we are trying to solve if the equation is modified a little bit. The problem of *conditioning* has been very well studied for PDEs, and again we will content ourselves with a linear perturbation analysis. We remark that, as usual, the terminology is not standardized: The theoretical developments typically use the terms “stability” or “sensitivity” to describe the relevant property of a PDE system that is not affected much by perturbation, where we want to reserve “stability” for the stability of an algorithm—an algorithm is stable if it gives the exact answer to a nearby problem, that is, generates a small residual. For consistency with the rest of this book, we use the term “well-conditioned” for a PDE system that is not sensitive to a small backward error.

Suppose that our PDE is, for example,

$$u_t = f(t,u,u_x,u_{xx}) \tag{16.9}$$

and that we are concerned with the difference between $u(t,x)$ and $U(t,x)$, where

$$U_t = f(t,U,U_x,U_{xx}) + \varepsilon v(t,x) \tag{16.10}$$

on some *compact* domain such as $a \le t \le b$, $c \le x \le d$. If we assume that $U(t,x) = u(t,x) + \varepsilon u_1(t,x) + O(\varepsilon^2)$ (of course, the hard part is justifying such an assumption, which we don't do here), then a standard perturbation argument suggests that $u_1(t,x)$ must satisfy the linear PDE

$$\frac{\partial u_1}{\partial t} = f_u(t,u,u_x,u_{xx})u_1 + f_{u_x}(t,u,u_x,u_{xx})\frac{\partial u_1}{\partial x} + f_{u_{xx}}(t,u,u_x,u_{xx})\frac{\partial^2 u_1}{\partial x^2} + v(t,x)$$

together with boundary conditions that ensure that $U(t,x)$ and $u(t,x)$ satisfy the original equations. For example, if Dirichlet conditions are given for $u(t,x)$, then we impose the Dirichlet conditions $u_1 = 0$ on the boundary. Thus, all we need to do to estimate the influence of $v(t,x)$ on the solution of the original PDE is solve this simpler linear PDE at the same time.

This is an example of the so-called forward method of sensitivity analysis, not to be confused with forward error analysis. When we try this on our advection–diffusion equation example above, we find that [pretty much no matter what smooth function we choose for $v(t,x)$] u_1 is not large; thus, we believe that this equation is well-conditioned. Of course, since this is a linear equation, it is its own linearization; and, moreover, there are a great many theoretical tools available to investigate the conditioning of linear equations.

16.3 The Transverse Method of Lines

The method of lines as described above discretizes in space and then solves an initial-value problem. It is very natural to wonder if this can be done in the other direction, discretizing in time and then using adaptive software to solve the resulting problems in space. The main goal is to use spatial adaptivity for efficiency. Moreover, this spatial adaptation can itself be informative. This approach, which Franzone et al. (2006) calls *Rothe's method*, and which is also called the *transverse* method of lines, replaces the PDE with a sequence of boundary-value problems. The pioneering work of Bornemann (1992) was perhaps the first to make the method practical. Such problems are now fully adaptively handled by the method outlined in Nowak (1996). See also Deuflhard and Bornemann (2002). Like the traditional method of lines discussed above, this is a semidiscretization.

The transverse method of lines idea is quite old.[3] However, the method was not popular until about the 1990s, because the adaptive mesh technology for BVP had needed to be developed first. Nowadays, we find that the method is very advanced indeed: See, for example, Franzone et al. (2006), where three-space dimension plus time dimension cardiac reaction–diffusion models are solved by this method, with adaptivity in all dimensions.

[3] See, for example, the footnote on p. 264 of Collatz (1966), which points out that the idea was already known to Hartree and Womersley in 1937.

Example 16.4. Let us attempt first the simple advection–diffusion example used above, yet one more time. This time we do not discretize in space first, but rather in time. That is, we replace $u(t,x)$ by a sequence of functions $u(t_k,x)$, for some temporal mesh $0 = t_0 < t_1 < \ldots t_n$. If we're smart (and ambitious), we will use the information obtained during the solution process to help us choose the temporal mesh, as we go.

For concreteness, let us choose to use the simple *backward Euler* scheme, based on

$$y'(t+h) = \frac{y(t+h)-y(t)}{h}\,. \tag{16.11}$$

Our equation is replaced by

$$\frac{u(t_{n+1},x)-u(t_n,x)}{t_{n+1}-t_n} = -au_x(t_{n+1},x) + \nu u_{xx}(t_{n+1},x)\,. \tag{16.12}$$

If we presume that $u(t_n,x)$ is known, then this gives us a second-order BVP as an *ordinary* differential equation to solve for the as-yet-unknown $u(t_{n+1},x)$. For brevity, write $u_n(x)$ or just u_n for $u(t_n,x)$, and h for $t_{n+1}-t_n$. The process is that at each stage we solve the equation

$$h\nu {u_{n+1}}'' - ha{u_{n+1}}' - u_{n+1} = -u_n(x)\,, \tag{16.13}$$

subject to the boundary conditions $u(t_n+h,-1)=0$, $u(t_n+h,1)=0$. To solve this BVP, we could use any good-quality boundary-value problem solver, and let it adapt the spatial mesh as it wishes. Here we will use `bvp4c` and `bvp5c`.

To assess the residual in our solution, we will use Hermite interpolation. If $t = t_n + \theta(t_{n+1}-t_n)$, then

$$\begin{aligned} u(t,x) = (2\theta+1)(\theta-1)^2 u(t_n,x) + (\theta-1)^2\,\theta u_t(t_n,x) \\ - \theta^2(2\theta-3)u(t_{n+1},x) + \theta^2(\theta-1)u_t(t_{n+1},x)\,, \end{aligned}$$

where the time derivatives $u_{n,t}(x)$ and $u_{n+1,t}(x)$ can be calculated for $n \geq 1$ by

$$u_{n,t}(x) = \frac{u_n(x)-u_{n-1}(x)}{t_n-t_{n-1}} \tag{16.14}$$

$$u_{n+1,t}(x) = \frac{u_{n+1}(x)-u_n(x)}{t_{n+1}-t_n}\,. \tag{16.15}$$

Those look like finite differences, but they are exact time derivatives essentially by definition: We use the PDE to compute $u_t = -au_x + \nu u_{xx}$, and then notice that we have defined $u(t_n+h,x)$ in (16.12) so as to use part of that with its relation to $u(t_n,x)$. Simplifying, we get the formula above, which happens to work for other equations as well.

The advection–diffusion equation solution by this means is left to the exercises; it works well, albeit somewhat slowly (actually quite a bit more slowly than the spectral method; but a more sophisticated time integration would even the contest a little). ◁

Example 16.5. Let us try the method on Burgers' equation with artificial viscosity:

$$u_t + uu_x = \nu u_{xx}\,, \tag{16.16}$$

subject to the initial conditions $u(0,x) = 1$ if $0 \le x \le 0.1$ and $u(0,x) = 0$ if $x > 0.1$. We solve the equation on $0 \le x \le 4$ for $0 < t \le 5$. The side conditions are $u(t,0) = 1$ and $u(t,4) = 0$. We choose the diffusion coefficient $\nu = 0.02$. This equation is nonlinear and in the absence of diffusion has a propagating shock wave. With this formulation, it has an exact solution in terms of complementary error functions, which we will not need. The solution with $\Delta t = 5/80$ is plotted in Fig. 16.8. The meshes generated with $\Delta t = 5/40$ are plotted in Fig. 16.9. Both were generated with this code:

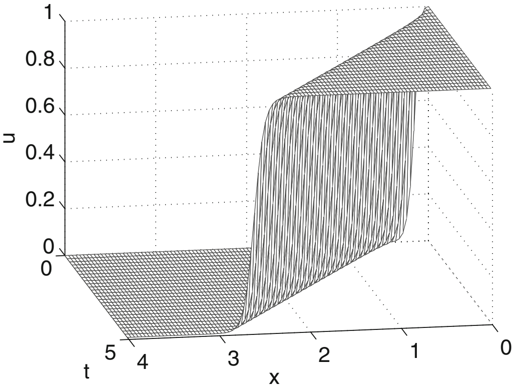


Fig. 16.8 The solution to Burgers' equation (16.16) with $\nu = 0.02$, $u(0,x) = 1$ if $x < 0.1$ and 0 otherwise, by the transverse method of lines using backward Euler for the time-stepping. A time step $\Delta t = 5/80$ was used. The built-in solver `bvp4c` was used to solve the BVP arising at each time step

```
function [ y ] = burgerstmol( tf )
%BURGERSTMOL Solve Burgers' equation by TMOL
%   Use bvp4c and Backward Euler
%   y = burgerstmol( tf );   v = u_x
%   u_t = -(u^2/2)_x + nu* v_x
```

The code demonstrates a simple-minded approach to solving Burgers' equation by the transverse method of lines. It is intended to demonstrate that the solution is possible, not to be a general-purpose solver. If you actually download the code from

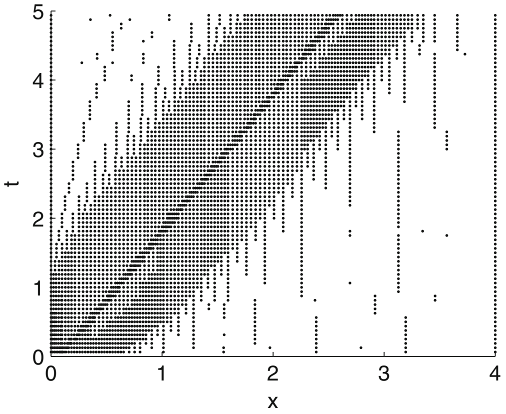


Fig. 16.9 The meshes used for the transverse method of lines solution to Burgers' equation (16.16) with $\nu = 0.02$, $u(0,x) = 1$ if $x < 0.1$ and 0 otherwise. A time step $\Delta t = 5/40$ was used. The built-in solver `bvp4c` was used to solve the BVP arising at each time step. The meshes plotted here were generated by `bvp4c` by continuation from the mesh generated on the previous time step. The initial mesh was $[0, 0.09, 0.099, 0.101, 0.11, 1., 2, 3, 4]$. As can be seen, the greatest density of mesh points follows the moving layer. This spatial adaptivity is the main advantage of using the transverse method of lines

the repository and read it, note the use of *cell arrays* to hold a sequence of solution structures for later use, for instance, by plotting or by computing the residual. ◁

It turns out to be slightly difficult to get this crude TMOL program started. In particular, if one takes too *small* a time step, then `bvp4c` fails to converge on the first step, complaining about a singular Jacobian matrix. The problem is indeed nonlinear, but `bvp4c` uses finite-difference Jacobians by default, and in this case that leads to singularity if one isn't careful. A convenient workaround, if one really does want to start with a very tiny time step, is to use a large one first and then use continuation, *reducing h* down to the desired level, and then starting the integration from there. This is done in the next example, really just for show. A better approach is to provide an analytical Jacobian, together with a better (smoother) initial guess, which is perfectly possible with this simple equation.

Example 16.6. Let us now have a look at the shock tube gas dynamics equations. This is another classic test problem. With artificial viscosity added, the equations are

$$\begin{aligned}\rho_t + m_x &= \lambda \rho_{xx} \\ m_t + \left(\frac{m^2}{\rho} + p\right)_x &= \lambda m_{xx} \\ E_t + \left(\frac{m}{\rho}(E+p)\right)_x &= \lambda E_{xx}\,. \end{aligned} \tag{16.17}$$

Here ρ is the density of the gas in the tube, which is taken to be uniform on the cross section at distance x, $m = \rho v$ is the momentum, p is the pressure, and E is the energy per unit volume. The equation of state relating the pressure p to the three unknown quantities ρ, m, and E is $p = (\gamma - 1)(E - m^2/2\rho)$. The gas constant γ is taken to be 1.4. The initial condition is taken to be $\rho(0,x) = 1$ if $x \le 0$, and $\rho(0,x) = 0.125$ if $x > 0$. That is, the tube initially has gases at two different densities, separated at time $t < 0$ by a membrane, which bursts instantaneously at $t = 0$. In addition, $m(0,x) = 0$, and $E(0,x) = 2.5$ if $x \le 0$ but $E(0,x) = 0.25$ if $x > 0$. We take the tube to occupy $-5 \le x \le 5$. The internal energy $e = p/((\gamma - 1)\rho)$ is also of interest.

Using the backward Euler method on the time derivatives, we arrive at the following boundary value problem for ordinary differential equations for the unknown functions $\rho(t_n + h, x)$, $m(t_n + h, x)$, and $E(t_n + h, x)$, assuming that we know the solutions $\rho(t_n, x)$, $m(t_n, x)$, and $E(t_n, x)$. Denote the unknown functions by $\rho^{(n+1)}(x)$, and so forth. To save space, write $\rho^{(n+1)}$ for $\rho(t_n + h, x)$, and so on, so that we have

$$\begin{aligned}
\rho_{xx}^{(n+1)} &= \frac{\left(m_x^{(n+1)} + (\rho^{(n+1)}(x) - \rho^{(n)}(x))/h\right)}{\lambda} \\
m_{xx}^{(n+1)} &= \frac{\left(A_x^{(n+1)} + (m^{(n+1)}(x) - m^{(n)}(x))/h\right)}{\lambda} \\
E_{xx}^{(n+1)} &= \frac{\left(B_x^{(n+1)} + (E^{(n+1)}(x) - E^{(n)}(x))/h\right)}{\lambda} ,
\end{aligned} \qquad (16.18)$$

where the flux variables $A = m^2/\rho + p$ and $B = m(E+p)/\rho$ are used here only for convenience. Using $\lambda = 10^{-3}$ and $h = 0.005/64$, we get the energy graph at $t = 0.6$ shown in Fig. 16.10. Here we are using a first-order method in time, which means that an enormous number of time steps are needed, and, moreover, the contact discontinuity is unacceptably diffused here (see the exercises). We show the final mesh used by `bvp5c` in Fig. 16.11, which shows a fine resolution used near the shocks and only there. For some reason, using `bvp4c` to solve these BVP produced an unacceptable mesh, with one interval of width about 10^{-8}; we don't know why. The code used begins with

```
function [ y ] = shocktube5tmol( n )
%SHOCKTUBETMOL Solve shock tube gas equation by TMOL
%   Use bvp4c and Backward Euler
%   u = [ rho; rho_x; m; m_x; E; E_x ]
%   u_t = -(u^2/2)_x + nu* v_x
```

See the code repository. The routine was called with the following, using `big = 7680` or $120 \cdot 4^3$ time steps, as follows:

```
%
% Long time run of shock tube
%
big = 120*4^3;
yshock5 = shocktube5tmol(big);
x = yshock5{big}.x;
```

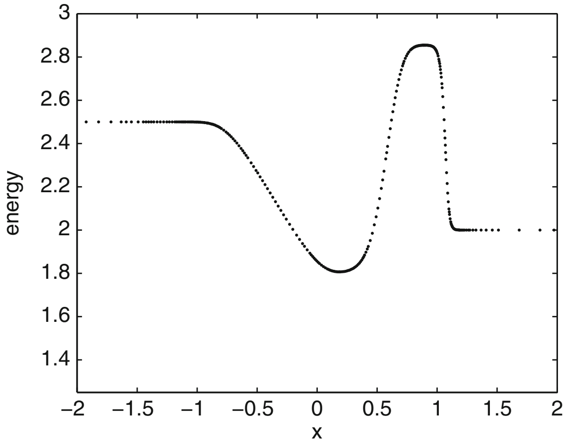


Fig. 16.10 The internal energy $e = p/((\gamma - 1)\rho)$ for the solution of the shock tube gas dynamics equations with artificial viscosity, at $t = 0.6$, by the transverse method of lines using the backward Euler method for temporal integration and `bvp5c` for the integration of the BVPODE at each time step. For this problem, the backward Euler method is too slow to use in practice

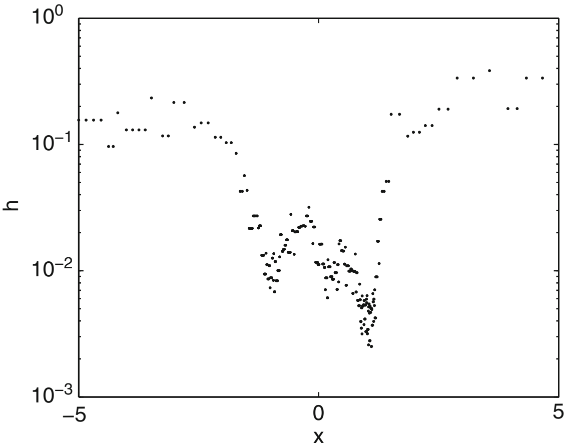


Fig. 16.11 The final mesh used for the transverse method of lines solution to the shock tube gas dynamics equations with artificial viscosity (16.18) with $\lambda = 10^{-3}$. A time step $\Delta t = {}^{0.005}/_{64}$ was used. The built-in solver `bvp5c` was used to solve the BVP arising at each time step. The mesh plotted here was generated by `bvp5c` by continuation from the meshes generated on the previous time steps. As can be seen, the greatest density of mesh points follows the shocks. Again, this spatial adaptivity is the main advantage of using the transverse method of lines

```
rho = yshock5{big}.y(1,:);
m = yshock5{big}.y(3,:);
E = yshock5{big}.y(5,:);
p = 0.4*(E-(m.^2./rho)/2);
e = p./(0.4*rho);
figure
plot( x, e, 'k.')
```

Integration took about a second per time step on a desktop machine (backward Euler is so inefficient). By using even a second-order method, we speed things up by a factor of 2000, and by using compiled code instead of MATLAB's interpreted code, we would gain another factor of 100 or so (see the exercises). ◁

16.4 Using PDEPE in MATLAB

Typing `odeexamples['pde']` at the MATLAB prompt gets you a menu of partial differential equation examples. The solver used, `PDEPE`, implements a sophisticated method of Skeel and Berzins (1990). The program was written by Larry Shampine and Jacek Kierzenka. The method handles PDEs of the parabolic-elliptic type, that can have mild singularities, in the following form:

$$\mathbf{C}(x,t,u,u_x)u_t = x^{-m}\left(x^m g(x,t,u,u_x)\right)_x + f(x,t,u,u_x)\,. \tag{16.19}$$

The entries of the matrix $\mathbf{C}$ may depend on x, t, u, u_x but not u_t or higher x-derivatives. The equation may be at most second order in space. The domain considered is finite in space, $a \le x \le b$, and the matrix $\mathbf{C}$ must be diagonal with nonnegative entries (Shampine and Kierzenka relax this so that some, but not all, of the entries may be zero in their implementation). The boundary conditions are

$$p_i(x,t,u) + q_i(x,t)g_i(x,t,u,u_x) = 0 \tag{16.20}$$

at $x = a$, b, for $i = 1, 2, \ldots, N$, where N is the number of PDEs. This class of problems covers a large number of physically interesting models. If $m \ge 0$, the method requires $a \ge 0$, but allows $a = 0$, which includes singular behavior at $x = 0$ in that case, though of a fairly restricted type. Other singularities in the PDE are also allowed, but mesh points x_k must be placed at the singularities so that the solution is smooth enough over the mesh intervals to be represented by piecewise polynomials. For an example, see `pdex2`, which is the second of the examples in `odexamples`.

Here we will look at `pdex3`, the third of the examples. In the comments for the source code for `pdex3`, the equation is presented as a model that arises in transistor theory:

$$c\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(d\frac{\partial u}{\partial x}\right) - \frac{\eta}{L}\frac{\partial u}{\partial x}\,, \tag{16.21}$$

where c, η, and d are physical constants (given nominal values in the example source: $d = 0.1$, $\eta = 10$). The equation is to hold on $0 \le x \le L$, and, moreover,

L is taken to be 1. The boundary conditions are $u(0,t) = 0$ and $u(L,t) = 0$. The initial condition, which contains another physical constant K, taken to be $K = d$, is

$$u(x,0) = \frac{KL}{d\eta}\left(1 - e^{-\eta(1-x/L)}\right) . \tag{16.22}$$

One interesting feature is that it isn't just the solution that is desired, but also the *emitter discharge current*

$$I(t) = \frac{I_p d}{K} u_x(0,t) , \tag{16.23}$$

which depends on the x-derivative of the solution at the left end. The constant I_p is taken to be 1. Using pdepe to compute the solution, and pdeval to evaluate that solution and its x-derivative, gives the graphs shown in the example (not shown here).

When we implement the example using our spectral method from the first section of this chapter (this example is really just the advection–diffusion equation in a different guise), we get the following code, except that the commented line 18 is the one that should define the PDE:

```
1  function [sol, I] = transistormol(n)
2  %
3  % transistor equation from pdex3 by the method of lines
4  %
5  close all
6  % Problem parameters, shared with nested functions.
7  L = 1;
8  d = 0.1;
9  eta = 10;
10 K = d;
11 I_p = 1;
12 % pdex3 has 0 <= x3 <= 1.  Here we use -1 <= x <= 1.
13 % Change of variable means x = 2x3 - 1, hence d/dx = (1/2) d/dx3.
14 D = gallery('chebspec',n+2);
15 x = cos( pi*(0:n+1)/(n+1) );
16 function dy = mol(t,y)
17     my = [0; y; 0];
18     % tmp = -eta/L*2*D*my + d*4*D*D*my  is the comment version
           pde
19     tmp = -d*eta*2*D*my + d*4*D*D*my; % is the coded version
           pdex3
20     dy = [tmp(2:end-1)];
21 end
22 x3 = (x+1)/2;
23 % (K*L/d)*(1 - exp(-eta*(1 - x)))/eta
24 inits = K*L/d/eta*(1-exp(-eta*(1-x3/L)));
25 tf = 1.0;
26 sol = ode15s( @mol, [0,tf], inits(2:end-1)' );
27 t = linspace(0,tf,101);
28 z = [zeros(size(sol.x)); sol.y ; zeros(size(sol.x)) ];
29 zx = 2*D*z;
```

```
30 figure
31 I = I_p*d*zx(end,:)/K;
32 semilogy( sol.x, I, 'k.')
33 xlabel('t')
34 ylabel('Emitter␣discharge␣current␣I(t)')
35 figure
36 %plot( x, inits, 'k+', x , sol.y(:,end), 'ko' )
37 mesh( x3(2:end-1), sol.x, sol.y' ), view(160,50),colormap
      ([0,0,0]), axis([0,1,0,tf,0,0.1]);
38 end
```

Executing this code with line 18 uncommented and line 19 commented out gives us a somewhat similar picture to the results of pdex3, but not on the right scale. There is an error somewhere, therefore. It turns out that the error is either in the comments to pdex3 or in the coding of the differential equation, which has something different than what is in the comments, namely,

$$u_t = du_{xx} - d\eta u_x . \tag{16.24}$$

There is a $d\eta u_x$ instead of η/Lu_x. Thus, we use line 19 instead, to get pictures on the same scale as the results of pdex3.

This is of little importance—it is only an example. Of more importance is that the emitter discharge current, graphed on a log scale in Fig. 16.12, does not look at all like the results of pdex3. This is puzzling. Is there another error in transcription? Or is the method used here, a spectral method, wrong somehow? Or is the result in pdex3, which they compare with a series solution, itself wrong somehow? (The answer is *no*, to all these questions.)

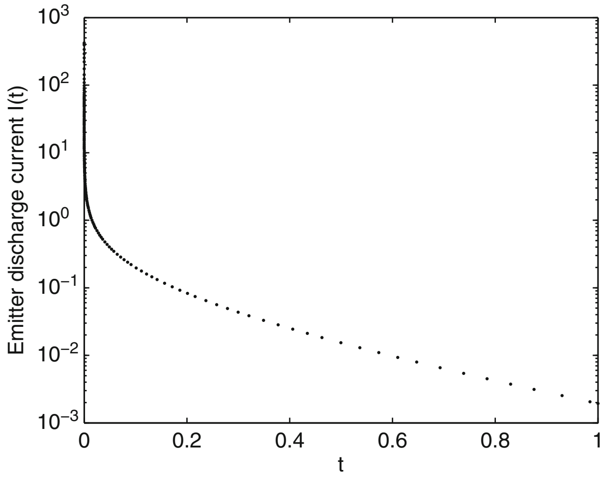


Fig. 16.12 The emitter discharge (16.23) plotted on a logarithmic scale. This should be compared to the graph produced by running pdex3 in the odeexamples['pde'] demo. The two figures are quite different; here we see a change of two orders of magnitude in the initial interval

To try to answer these questions, we turn to the computation of the residual. If the residual in the differential equation is small for the spectral method, and not small for the results of pdepe, then we will know which answer to trust. Likewise if it's the other way around. If *both* residuals are small, we suspect a blunder in the coding of the emitter discharge function, or a difficulty getting an accurate x-derivative, or perhaps a simple misunderstanding of a feature of the example (which turns out to be the case).

Because we have already computed a residual for a spectral method, we do that first:

```
function [res, t, xi, u, uxi ]  = residualtransistormol(sol,n,tf)
x = cos( pi*(0:n+1)/(n+1) );
D = gallery('chebspec',n+2);
% Problem parameters, shared with nested functions.
L = 1;
d = 0.1;
eta = 10;
K = d;
I_p = 1;
% pdex3 has 0 <= x3 <= 1.  Here we use -1 <= x <= 1.
% Change of variable means x = 2x3 - 1, hence d/dx = (1/2) d/dx3.
em = 2*50;
en = 200;
%t = linspace(0,tf,em);
t = RefineMesh( sol.x, 3 );
j0 = find(t>=2/em);
j0 = j0(1);
em = length(t);
[u,ut] = deval( sol, t );
ux = D*[zeros(1,size(u,2));u;zeros(1,size(u,2))];
%xi = linspace(-1,1,en);
xi = RefineMesh(x,3);
en = length(xi);
res = zeros(en,em);
w = genbarywts( x, 1 );
for i=2:em,
  [uxi, uxxi] = hermiteval( [0;u(:,i);0], xi, x, 1, w, D );
  utxi = hermiteval( [0;ut(:,i);0], xi, x, 1, w, D);
  [uxxi2, uxxxi] = hermiteval( ux(:,i), xi, x, 1, w, D );
  %res = ut + d*eta*ux -d * uxx;
  res(:,i) = utxi + 2*d*eta*uxxi - 4*d*uxxxi;
end
figure
colormap([0,0,0])
sc = 0.5e-3;
mesh( t(j0:end), (xi(2:end-1)+1)/2, res(2:end-1,j0:end) ) , axis
    ([0,tf, 0, 1, -5*sc, 5*sc]), view(70,60), set(gca,'fontsize'
    ,16),xlabel('t','fontsize',16), ylabel('x','fontsize',16),
    zlabel('residual','fontsize',16)
```

The results are plotted in Fig. 16.13. Computing a residual for the solution produced by pdex3 is possible but requires some tedious bookkeeping. Instead, we compare the solutions at $t = 1$ in Fig. 16.14. We see that the solutions agree very well. Since

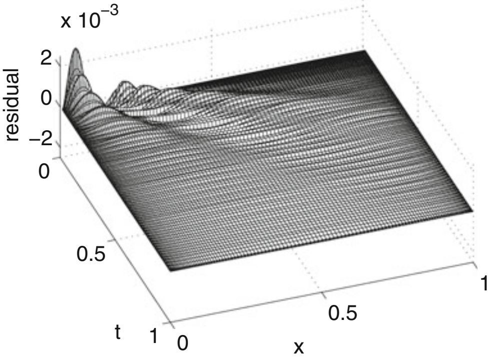


Fig. 16.13 The residual of the spectral solution (with $n = 40$ internal mesh points) to (16.24). That it is uniformly small, except possibly near the sharp initial layer (most of which we have trimmed off), shows that we have computed the exact solution to a nearby problem

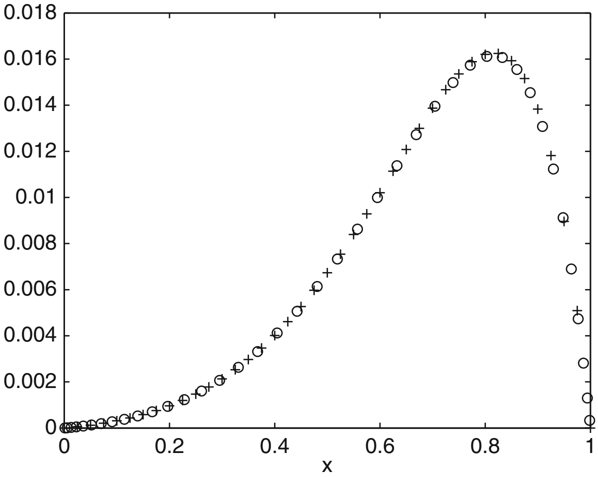


Fig. 16.14 The solution at $t = 1$ as computed by `pdepe` (+) and by the spectral method of the text (o). Agreement is good, to visual accuracy. Since only about 40 spatial mesh points were used in each computation, this agreement is about all that could be expected

the residual in the spectral solution is quite small, even very small away from the sharp corner at $x = 0$, $t = 0$, it seems that both solutions must be good. So why, then, the seemingly great difference in emitter discharge plots? It turns out to be because the spectral method took many small time steps initially in order to resolve the initial sharp layer, and we let the code choose where to plot the discharge! If we plot only the values of $I(t)$ for $t > 0.0171$ (ignoring the first 70 or so time steps), we get the same figure for $I(t)$ as `pdex3` does. Allowing the code to choose its own time steps shows the initial sharp change in the emitter discharge. We do not think this is a bad thing.

16.5 Poisson's Equation and the FFT

In the works Henrici (1979b), Chen (1987), and Iserles (2009), for example, we find discussions of how to solve Poisson's equation,

$$-\Delta u = f(x,y)\,, \tag{16.25}$$

in a square (or higher-dimensional domain), with either periodic boundary conditions or homogeneous Dirichlet conditions, using the FFT. Earlier finite-difference discussions are summarized in Collatz (1966) and are still worth reading. Recall that $\Delta u = u_{xx} + u_{yy}$ in two dimensions and Cartesian coordinates. This simple equation occurs in a vast array of applications. At least one MATLAB code for fast Poisson solving in 2D has been written and made publicly available.[4] Our code in `Poisson2D` described below is available in the code repository but is intended for didactic use only.

To implement this method, we will follow the discussion in Henrici (1979b), and use the FFT on a very simple finite-difference method (we will use a better method in the next section, and we could use it here but it's nice to begin with the simple methods) for approximating $-\Delta u = f$, namely,

$$-\Delta u \doteq 4u_{i,j} - u_{i,j+1} - u_{i,j-1} - u_{i-1,j} - u_{i+1,j} = h^2 f_{i,j} \tag{16.26}$$

on a uniform grid with spacing $h = 1/n$ in both the x- and y-directions. This approximation makes an error of $O(h^2)$. This gives rise to a linear system of equations for the n^2 unknowns $u_{i,j}$. As Henrici does, we solve a problem with $u = 0$ on the boundary, and we assume that $f = 0$ there likewise (this is without loss of generality). To this end we extend u and f as *odd* periodic sequences of period $2n$; this enforces $u = 0$ on the boundary.

Henrici points out that the combination of function values is a (2D) *convolution*, with the two-dimensional sequence $\mathbf{d}$ with $d_{0,0} = 4$, $d_{1,0} = d_{0,1} = d_{-1,0} = d_{0,-1} = -1$ and all other entries 0. Using the correspondence between convolutions in physical space and Hadamard products in Fourier space, we get the simple relation

$$4n^2 \hat{\mathbf{d}} \cdot \hat{\mathbf{u}} = h^2 \hat{\mathbf{f}}\,, \tag{16.27}$$

where the hat symbol signifies the two-dimensional Fourier transform, and the operation $\cdot$ is the Hadamard (elementwise) product. Henrici then gives one of those traditional mathematician phrases: "It is readily verified that $\hat{\mathbf{d}} = \{\hat{d}_{k,m}\}$, where

$$\hat{d}_{k,m} = \left(1 - 1/2 \cos\frac{k\pi}{n} - 1/2 \cos\frac{m\pi}{n}\right) h^2 \text{."} \tag{16.28}$$

[4] The code can be found at http://www.mathworks.com/matlabcentral/fileexchange/21472-2d-fast-poisson-solver. The author claims to have based the program on the discussion in Iserles (2009). We have not tested it.

If indeed this is true (see the exercises), then we have Algorithm 16.3 to solve the Poisson problem on a square with Dirichlet boundary conditions.

Algorithm 16.3 Solving Poisson's equation $-\Delta u = f$ with Dirichlet boundary conditions on a square, using the FFT, after Henrici (1979b)

```
Require: $n > 1$ parameterizing the $n \times n$ grid, $f_{ij}$ for $2 \le i, j \le n-1$ ($f = 0$ on the boundary)
  Define $f_{2n-i,j} = -f_{i,j}$, $f_{i,2n-j} = -f_{i,j}$, $f_{2n-i,2n-j} = f_{i,j}$ (extension to an odd function)
  Use fft2 or equivalent to compute $\hat{\mathbf{f}} = \mathscr{F}_{2n}^{(2)}\mathbf{f}$
  Define the elements of $\hat{\mathbf{d}}$ by (16.28)
  Compute $\mathbf{u} = h^2 \overline{\mathscr{F}_{2n}^{(2)}}\left(\hat{\mathbf{f}}./\hat{\mathbf{d}}\right)$
  Trim u to original size
  return An $O(h^2)$ approximation u to the solution of $-\Delta u = f$ at grid points $(i-1)/(n-1), (j-1)/(n-1)$.
```

However, some important difficulties remain. It turns out that implementing the algorithm in MATLAB is not completely straightforward. There are three issues that we noticed. The first is that Henrici indexes from 0, not 1, while MATLAB indexes from 1. Second, the discrete Fourier transform that Henrici used is

$$\mathbf{y} = \mathscr{F}_n(\mathbf{x})\,, \tag{16.29}$$

where

$$y_m = \frac{1}{n}\sum_{k=0}^{n-1} w_n^{-mk} x_k\,, \tag{16.30}$$

which is the *inverse* FFT in MATLAB. The third difficulty is the explicit use of the formula (16.28). When $k = m = 0$, we have $\hat{d}_{0,0} = 0$, and we are to divide by this—of course, this only works because $\hat{f}_{0,0}$ is also zero, and we are to take the ratio to be zero. If we do not explicitly replace $\hat{d}_{0,0}$ with something harmless, then NaNs invade our computation, making it useless. Still, all these difficulties can be overcome. A crude implementation is given below:

```
function [ xi, yi, u ] = Poisson2D( n, f )
%POISSON2D FFT method for solving 2D Poisson equation on a square
%   Using standard O(h^2) finite differences we get a linear
%   (block circulant with circulant blocks) system of equations
%   to solve for u: -(u_xx + u_yy) = f gives A u = f.
%   Using the 2d FFT diagonalizes the n^2 by n^2 system.

% Use left/upper part of grid
x = (0:n)/n;
h = 1/n;
[xi,yi] = meshgrid( x );
fv = feval(f, xi, yi );
% zero out the outer edges
fv(1,:) = zeros(1,n+1);
fv(n+1,:) = fv(1,:);
fv(:,1) = fv(1,:)';
fv(:,n+1) = fv(:,1);
% Odd extension
fv = [fv, -fv(:,end-1:-1:2); -fv(end-1:-1:2,:), fv(end-1:-1:2,end
    -1:-1:2) ];
% Henrici's Fn is Matlab's ifft
fhat = ifft2( fv );
% Explicit computation of dhat by formula
[k, m] = meshgrid( 2*(n):-1:1 );
dhat = (1-cos(k*pi/n)/2 - cos(m*pi/n)/2)*h^2;
dhat(1,1) = 1; % Don't divide by zero.
fz = fhat./dhat; % Nans will show up otherwise, and they hurt.
u = h^2*fft2( fz )/4/n^2;
u = real(u(1:n+1,1:n+1)); % Crude code, real u only
% Explicitly impose BC, avoid roundoff error
u(1,:) = zeros(1,n+1);
u(n+1,:) = u(1,:);
u(:,1) = u(1,:)';
u(:,n+1) = u(1,:)';

end
```

We test this with a simple problem with a known answer, $-\Delta u = 2\pi^2 \sin\pi x \sin\pi y$. This happens to be zero on the boundaries and has the known analytic solution (we built it this way) $u = \sin\pi x \sin\pi y$. Using $n = 32$ gives a forward error smaller than 8.04×10^{-4}, and with $n = 1024$, the error is smaller than 7.9×10^{-7}, indicating $O(h^2)$ behavior, as expected. We leave the residual-based error analysis to the exercises.

16.6 Reaction–Diffusion Equations and Turing Patterns

It will not have escaped the reader's notice that we have confined ourselves so far to PDE of only two dimensions: one space and one time or two space. It's now time to leave that restriction behind, but the book will end, before we get to very high dimensions. Part of our decision to stop there is computing cost (at the time of writing) in MATLAB; we *can* compute in three space dimensions and time (see the exercises), but the waiting time begins to be a drag, and the person doing the waiting will be strongly tempted at this point to move to a "proper" scientific computing language, and thereby shift into the fast-paced world of HPC.

Before that shift happens, there are things to learn. Let's look at a simple two-space and one-time dimensional PDE modeling reaction–diffusion, which we take from Ruuth (1995), namely, the Schnakenberg equations

$$\begin{aligned} u_t &= \lambda(0.126779 - u + u^2 v) + \Delta u \\ v_t &= \lambda(0.792366 - u^2 v) + 10\Delta v\,, \end{aligned} \tag{16.31}$$

where $\lambda = 1000$. We take periodic boundary conditions on the square $0 \le x, y \le 1$, and $t > 0$. The initial condition we use is slightly different from that used in

Ruuth (1995), but not qualitatively so—we just made it a little less symmetrical initially. Specifically, we take u and v to be small departures from constants, initially:

$$\begin{aligned} u(0,x,y) &= 0.919145 + 0.0016\cos(2\pi(x+y)) + 0.01\sum_{j=1}^{8}\cos(2\pi jx) \\ v(0,x,y) &= 0.937903 + 0.0316\cos(2\pi(2x-y)) + 0.01\sum_{j=1}^{8}\cos(2\pi jx)\,. \end{aligned} \tag{16.32}$$

We use the method of lines on an equally spaced grid in x and y, giving a grid of n^2 points $x_i = {}^{(i-1)}/_{n}$, $y_j = {}^{(j-1)}/_{n}$, for $1 \le i, j \le n$. By assuming that $x_0 = x_n$ and $x_{n+1} = x_1$, and likewise $y_0 = y_n$ and $y_{n+1} = y_1$, we impose the periodic boundary conditions.

Such a grid and periodicity suggest that we should use a spectral method again, and indeed that works well. Instead, however, we shall use compact finite differences, specifically the classical *Mehrstellenverfahren* of Collatz (1966). The reason for this is twofold: First, the method, though classical, is very powerful and flexible and can be treated with FFT methods (we do not do this here, but refer the reader to Henrici (1979b)). Second, it shows off an interesting feature of the MATLAB codes, namely, that they can be used with a *sparse mass matrix* in a quite efficient manner. In fact, because the system is block circulant with circulant blocks, this could be done even more efficiently than we do here, possibly by the (again FFT-based) methods of Chen (1987), but we simply want to show how straightforward use of MATLAB's sparse matrices can lead to significant utility.

The basic idea runs as follows. The *Mehrstellenverfahren* is a way of approximating Δu using values of u on a square grid by giving an $O(h^4)$-accurate relationship between Δu and u on a standard nine-point stencil (i.e., nine points arranged in three rows of three). Computation by Taylor series[5] shows that if $u_{i,j}$ is in the center of the stencil,

$$\begin{aligned} \Delta u_{i,j} + {}^{1}/_{8}\left(\Delta u_{i-1,j} + \Delta u_{i+1,j} + \Delta u_{i,j-1} + \Delta u_{i,j+1}\right) = \\ \Big({}^{1}/_{4}\left(u_{i-1,j-1} + u_{i+1,j-1} + u_{i-1,j+1} + u_{i+1,j+1}\right) \\ + \left(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}\right) - 5u_{i,j}\Big)/h^2 + O(h^4), \end{aligned} \tag{16.33}$$

and indeed the $O(h^4)$ term can be seen to be approximately $\left({}^{(u_{xxxxxx}+u_{yyyyyy})}/_{160} - {}^{(u_{xxxxyy}+u_{xxyyyy})}/_{96}\right)h^4 + O(h^5)$. This gives a linear relationship between Δu at the grid points and u at the grid points, which must be solved to give Δu.

Because the stencil is compact, the linear system is sparse. Because we are on a periodic square grid, the system is, in fact, block circulant with circulant blocks, each of which is sparse, and indeed that is how the matrices are constructed for our computations (see below). That is, we have a sparse linear system

$$\mathbf{A}\Delta u = \frac{1}{h^2}\mathbf{B}u \tag{16.34}$$

[5] Or, of course, simply reading Table VI of Collatz (1966).

to solve, in order to approximate the Laplacian. This approach is often used to solve Poisson's equation, by the way; see Collatz (1966) or Henrici (1979b).

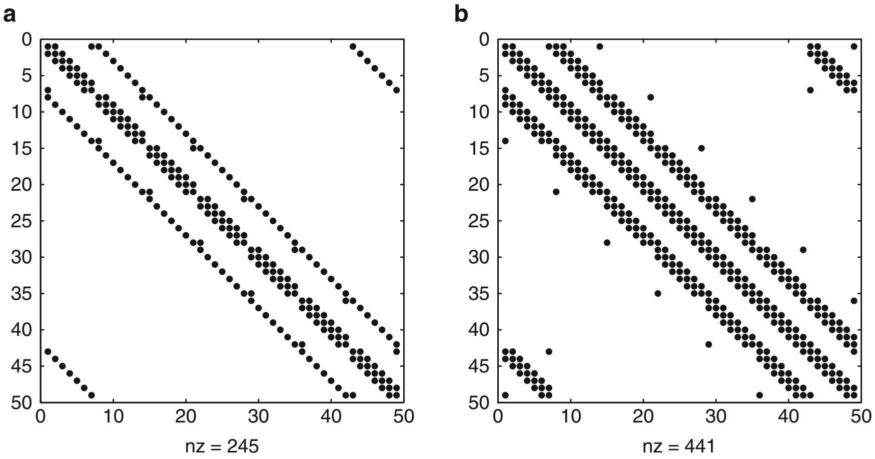


Fig. 16.15 The sparse block circulant structure with sparse circulant blocks of **A** and **B** in the *Mehrstellenverfahren* (16.33) on a 7 × 7 grid. (**a**) `spy(A)`. (**b**) `spy(B)`

We need also to specify an *ordering* on the grid. This choice can have significant impact on the structure of the resulting matrix. Here, we take the simplest possible approach, and number the points in row order: The point at (x_i, y_j) is numbered $j + (i-1)n$. By periodicity, then, for **A**, the compact finite-difference formula (16.33) will pick out the two nearest neighbors to the left and right (which might wrap around to the other edge of the matrix) and the nearest neighbors above and below, which puts them n places away to the left or the right (again, wrapping around if necessary). Each row of **A** has therefore four nondiagonal elements, and only four. As stated before, **A** is block circulant with circulant blocks, and at this point we note that it is also symmetric. Because we will be solving $\mathbf{A}\Delta = \mathbf{b}$, the condition number of **A** is of interest. In the exercises, you will be asked to show that the condition number of this matrix is bounded by 3, independently of the dimension.

The matrix **B** is similarly symmetric and block circulant with circulant blocks, but because it has eight nondiagonal entries per row, it is slightly less sparse. This is not significant, as we will be multiplying by **B** and not solving a system with **B** as the matrix (Fig. 16.15).

These things being considered, we can then use this code to attack the problem:

```
function [ A, B ] = compactlaplacian( n )
%COMPACTLAPLACIAN Fourth-order nine-point stencil Laplacian
%   Mehrstellen Verfahren of Collatz (1950)
%   A Del U = B U/h^2 to O(h^4)
%   Periodic boundary conditions on a square equally-spaced grid
%   Row-ordering of the grid
%   Both A and B are block circulant with circulant blocks
%   Both A and B are symmetric
%   Condition number of A is bounded by 3, independent of n

E = speye(n);
C  = sparse(2:n,1:n-1,1,n,n);
C(n,1)=1;
C = C + C'; % Sparse symmetric circulant
A = kron(C,E/8) + kron(E,C/8) + speye(n^2);

D = -5*E + C;
F = E + C/4;
B = kron( E, D ) + kron(C,F);

end
```

How does this help us with the method of lines? We embed u and v in a $2n^2$ vector as described below, then multiply the Schnakenberg equations (with now u and v being n^2-vectors) by the matrix $\mathbf{A}$ to get

$$\begin{aligned}\mathbf{A}u_t &= \mathbf{A}\lambda(0.126779 - u + u^2 v) + \frac{\mathbf{B}u}{h^2}\\ \mathbf{A}v_t &= \mathbf{A}\lambda(0.792366 - u^2 v) + \frac{10\mathbf{B}v}{h^2}\,,\end{aligned} \tag{16.35}$$

which now no longer has spatial derivatives in it, at a cost of making an $O(h^4)$ change in the right-hand side. Now, arranging a $2n^2$-vector $y(t)$ so that each column of u occupies a block of n entries in the first n^2 entries of y, and so that each column of v occupies a block of n entries in the second n^2 entries of y, we have a system of $2n^2$ ordinary differential equations to solve in a format suitable for MATLAB's ODE routines. Because this system is likely to be stiff, we consider only a stiff method here.

This works in MATLAB because `ode15s` (for example) allows ODE to be input with a so-called mass matrix, in this case the constant sparse matrix diag$[\mathbf{A},\mathbf{A}]$. The routine is careful not to invert this sparse matrix, which would produce a full Jacobian and thus contribute to an $O((n^2)^3) = O(n^6)$ linear algebra cost per step. We expect that the Jacobian now tends to be somewhat fuller because of the mixing $\mathbf{A}u^2v$, but this is of less consequence than the original diffusion terms. Experimentally, we see that the cost grows at least initially (for $n \le 64$) more slowly than $O(n^6)$; a reasonable fit of a line with slope $O(n^4)$ is obtained, for the range of n used here. Best of all, because the discretization is so high accuracy [compared to the simpler $O(h^2)$ explicit discretization of the Laplacian], we can get very accurate solutions with n as low as 24. Note that the solution in Ruuth (1995) used the standard second-order discretization and took $n = 128$; but a detailed efficiency comparison is not appropriate here as there are faster (FFT) methods than either of these. Perhaps even better than the "best of all" just mentioned, the coding is extraordinarily simple, as seen below:

```
function [ x, sol ] = Schnakenberg( n )
%SCHNAKENBERG Solve 2D Schnakenberg equation on a square
% Parameters and initial conditions taken from Ruuth (1995)
% (initial conditions slightly modified)
%
h=1/n;
lambda = 1000.;
du = 1.0;
dv = sqrt(10);
%
x = (0:n-1)/n; %linspace(0,1,n);
y = x;  % Tensor product grid
[A,B] = compactlaplacian(n);
%
ex = ones(n)*diag(x);
ey = diag(y)*ones(n);
cxy = cos(2*pi*(ex+ey));
cxy2= sin(2*pi*(2*ex-ey));
u0 = 0.919145*ones(n)+0.0016*cxy;
v0 = 0.937903*ones(n)+0.0316*cxy2;
for j=1:8,
    c = cos(2*pi*j*ex);
    u0 = u0 + 0.01*c;
    v0 = v0 + 0.01*c';
end
u0 = u0';
v0 = v0';
inif = [u0(:);v0(:)]; % Row order
% Compact fourth-order space derivatives
% Mehrstellen verfahren (Collatz, 1950)

% Mass matrix [[A,0],[0,A]]
 [i,j,s] = find(A);
 M = sparse([i,i+n^2],[j,j+n^2],[s,s],2*n^2,2*n^2);

    function yp = Sc(t,y)
        u = y(1:n^2,:);
        v = y(n^2+1:end,:);
        yp = zeros(size(y));
        usv = u.^2.*v;
        yp(1:n^2,:) = du^2*B*u/h^2+A*lambda*(0.126779 - u + usv);
        yp(n^2+1:end,:)= dv^2*B*v/h^2+A*lambda*(0.792366 - usv);
    end

opts = odeset('reltol',1.0e-7,'Mass',M,'Vectorized','on');
ts = 2.0;
sol = ode15s(@Sc,[0,ts], inif, opts );

end
```

So, how does it work? When we call this function with, say, the command `Schnakenberg(16)`, the answer comes back in a few seconds. Using $n = 64$ requires more time. See Fig. 16.16 for computation times. Well and good, but how good is the answer? Plotting the results of the 16×16 grid, which after all ought to

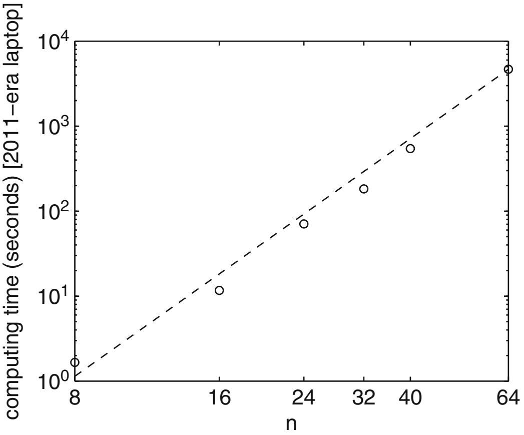


Fig. 16.16 Computing times to solve the Schnakenberg equations using the *Mehrstellenverfahren* for various n. The reference line has slope n^4

be comparably accurate to the 128 × 128 grid used in Ruuth (1995), gives an unacceptably rough and blocky plot—we simply don't have enough points for a smooth graph. It turns out that each point is pretty accurate, but in order to get a smooth plot, we must *interpolate*. Using the two-dimensional FFT interpolant you wrote in Chap. 9 (or found on the web: We used a 2D code by Bjorn Gustavsson based on a 1D code by Robert Piche), this is easily done. An alternative that is good enough for plotting is `interp2` with the "spline" option, and indeed that was what was used to produce the contour plots given in Fig. I.1 in the preface to this book.

How accurate is the solution? We used a temporal tolerance of 10^{-7}, and the mesh width is $^1/_{16}$, so we believe that the spatial error will be something like a multiple of $^1/_{16^4}$ because we are using a fourth-order method; that is, less than $^1/_{65,536}$. So can we expect perhaps four figures of accuracy at these 16 × 16 points? Of course, it's hard to tell by looking at the answers. But we can compute on a finer grid, say 32 × 32, and since this is just a refinement of the grid (every other point in the 32 × 32 grid is in the 16 × 16 grid), we can compare the answers directly. But could they both be wrong?

Of course, we wish to compute the residuals and see how well the solutions satisfy the equations. For this, the FFT interpolant is much the most convenient, because it is infinitely differentiable (differentiating the splines of `interp2` is possible but tedious, and in the end the derivatives aren't high-quality anyway). Here,

$$u(t,x,y) = \sum_{m=-n/2}^{n/2}{}' \sum_{\ell=-n/2}^{n/2}{}' \hat{u}_{m,\ell}\, e^{2\pi i(mx+\ell y)}\,, \tag{16.36}$$

where, as in Henrici (1979b), the *prime convention* means that when n is even (as it is here), the terms corresponding to $m = -^n/_2$ and $m = {}^n/_2$ are each multiplied by $^1/_2$. Similarly for $v(t,x,y)$. These interpolants are very easy to differentiate, and so we can compute the method in MATLAB as follows:

```
function [ resu, resv ] = Schnakres( n, t, xin, solin )
%Schnakres Residuals in the Schnakenberg equation
%
if nargin < 3,
   [x,sol] = Schnakenberg(n);
else
    x = xin;
    sol = solin;
end
[y,yt] = deval( sol, t );
u=zeros(n);
vt = zeros(n);
ut = zeros(n);
v=zeros(n);
u(:) = y(1:n^2);
ut(:) = yt(1:n^2);
v(:)=y(1+n^2:end);
vt(:)=yt(1+n^2:end);
u = u';
ut=ut';
v = v';
vt=vt';
m = 4*n;
[xi,yi] = meshgrid((0:m-1)/m);
um = interpftd2( u, m, m, 0, 0 );
utm = interpftd2( ut, m, m, 0, 0 );
%figure,surf(xi,yi,um)
vm = interpftd2( v, m, m, 0, 0 );
vtm = interpftd2( vt, m, m, 0, 0 );
uxx = interpftd2( u, m, m, 2, 0 );
uyy = interpftd2( u, m, m, 0, 2 );
delu = uxx + uyy;
%figure, surf(xi,yi,delu)
vxx = interpftd2( v, m, m, 2, 0 );
vyy = interpftd2( v, m, m, 0, 2 );
delv = vxx+vyy;
usv = um.^2.*vm;
resu = utm - (delu + 1000*(0.126779 - um + usv));
resv = vtm - (10*delv + 1000*(0.792366 - usv));
figure,surf(xi,yi,resv/1000),colormap([0,0,0])
figure,surf(xi,yi,resu/1000),colormap([0,0,0])

end
```

The residuals for $n = 8$, 16, 24, 32, 40, and 64 have been computed, and the maximum magnitudes of the residuals (scaled by $\lambda = 1000$) for u and for v are plotted on a log–log scale in Fig. 16.17. We see clear evidence of $O(h^4)$ behavior. At the end, we have evidence that the computed u and v are the exact solutions of the reverse-engineered equations

$$
\begin{aligned}
u_t &= \lambda(10^{-4}\delta_1(t,x,y) + 0.126779 - u + u^2 v) + \Delta u \\
v_t &= \lambda(10^{-4}\delta_2(t,x,y) + 0.792366 - u^2 v) + 10\Delta v\,,
\end{aligned}
\tag{16.37}
$$

where the functions $\delta_1(t,x,y)$ and $\delta_2(t,x,y)$ are $O(1)$ in magnitude.

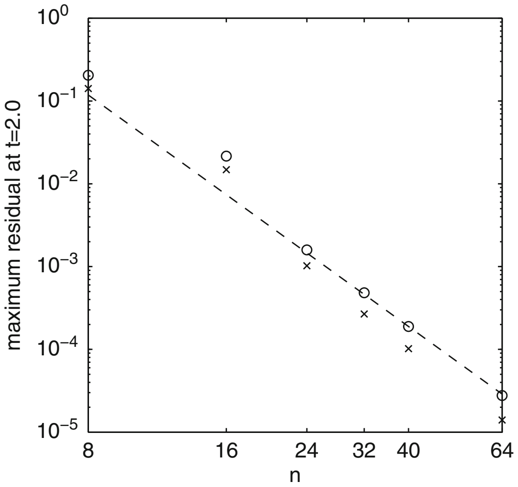


Fig. 16.17 Maximum residuals at time $t = 2$ in the solution the Schnakenberg equations using the *Mehrstellenverfahren* for various n. The reference line has slope $^1/_{n^4}$

16.7 Concluding Remarks

This presentation of an a posteriori analysis of the numerical solution of a nonlinear time-dependent PDE has had, we hope, a familiar feel: This style of analysis holds for every computational problem studied in this book. *As usual*, we are left with worrying about the effects of such perturbations, but again as usual this is something we have to worry about anyway, in view of physical perturbations. For this textbook, we have set up the need to study the conditioning (sensitivity) of the Schnakenberg PDE to changes. This is left as an exercise, but it is not unimportant. (You should find that this PDE is fairly well-conditioned near the solutions computed here. The Turing patterns generated by solving the Schnakenberg equations are correct.)

This chapter has used temporal adaptivity in the method of lines, by taking advantage of high-quality codes. The results of this can be seen, for example, in Figs. 16.5 and 16.4—near the singularity at the end, the code takes many time steps, and the fine temporal grid makes the surface look quite dark. Similarly, when using the transverse method of lines, the built-in spatial adaptivity of `bvp4c` was used, and the adaptive spatial mesh provided a useful advantage. In general, one would want to use both at once; in higher space dimensions, the generation of a good spatial mesh is itself a significant task. The principle of equidistribution discussed in Chap. 14 generalizes nicely; see, for example, Huang and Russell (2011) for its use in generating moving meshes. The work Franzone et al. (2006) gives an example of using adaptive meshes on a large-scale problem of medical interest.

We would really like to avoid all that work of interpolating, differentiating the interpolant, computing the residual, and looking how large or unusual it was. It would be preferable to guarantee ahead of time that the residual will be small.

Much classical numerical analysis of methods for linear PDE accomplishes just that. Nonlinear problems, solved by complicated methods, are more difficult, although progress has been made for some methods and some problems. We hope that this textbook presentation has encouraged the reader to investigate further on their own.

There is an impossibly large amount of material remaining to study in the field of numerical solution of PDE. Wherever we stop, it will seem abrupt, and we will have left out important things. We will mention in the references section that follows some places where you should look in order to pursue your investigation.

16.8 Notes and References

One of our favorite books has now been printed in a new edition: Iserles (2009). In that book, you will find an excellent overview of many points of both theoretical and practical interest, including a discussion of the fast Poisson solvers using the *Mehrstellenverfahren* (called more prosaically the modified nine-point method in that book) and the FFT. The "bluffer's guide" at the end is also useful!

The books Kreiss and Lorenz (1989) and Morton and Mayers (1994), and more recently Ascher (2008) and Bertoluzza et al. (2009), also provide good entry points to the study of the numerical solution of PDE. The last mentioned has a discussion of wavelets. The book Trefethen (2000), while short, is an excellent introduction to spectral methods and is focused on MATLAB. For state-of-the-art examples of using numerical methods to explore complicated physical models, see Provatas and Elder (2010). The book Deuflhard and Bornemann (2002) contains a more thorough introduction to the method of lines than given here. Multigrid and multilevel methods are among the fastest methods known (see Briggs and McCormick (2000) to get started). Probably the most important subject neglected in this chapter is the method of *finite elements*. There are a great many textbooks on finite elements, and the reader can start practically anywhere, but the books Zienkiewicz et al. (2000) are popular and thorough. Perhaps unusually, the ancient (in computer terms) book Strang and Fix (1973) is still worth recommending, as being readable and apt.

We direct the reader to more advanced treatments of the theory of conditioning of PDE, such as Li and Petzold (2004), for a more thorough picture. See Petzold et al. (2006) and the references therein for more details on the forward method of sensitivity analysis as discussed here, including some warnings of just when this method is impractical due to expense. For a recent application of these ideas, see Babuška and Gatica (2010). The history of the use of, first, a residual to estimate the backward error, and second, a conditioning estimate to improve the forward (global) error is very long. Early work by Babuška and others (e.g., Babuška and Rheinboldt 1978) in the theory and practice of finite elements for solving PDEs would give a good introduction to that history. Braack and Ern (2003) discuss the use of conditioning and residuals to estimate not only the effects of numerical errors, but also modeling errors—where the more complicated but more realistic physical equations are used only where necessary. This approach seems very interesting to us, especially because so much of modern applied mathematical modeling and computation involves the judicious choice of model and summary model in order to achieve sufficient computational speed to get useful answers in a reasonable time. In the past dozen years, there have been several excellent summaries of the issues; see, for example, Becker and Rannacher (2001) and Giles and Süli (2002).

For a discussion of providing a continuous solution even when the underlying method does not, see Enright (2000a). See Shampine et al. (2003) for a discussion on the benefits of using high-quality IVP software to do the time integration. We stop here: Some of those papers are about 100 pages each and have themselves been cited and followed up many times in the past 10 years. For this "paint a picture of a scratch on a surface" chapter, we will have to be content with having given just a few pointers to the literature. Happy reading!

Problems

Theory and Practice

16.1. Take the solution to (16.1) computed by the method of lines on the Chebyshev grid $x_j = \cos{}^{\pi(j-1)}/_{(n-1)}$ for $n = 32$ points; integrate for $a = 1$ and $\nu = {}^1/_{100}$ out to $t = {}^1/_2$. Interpolate your solution by the natural t-interpolant provided by `ode15s`, and use Lagrange interpolation in the x-direction. Compute the residual along the line $t = 0.3$ at, say, 1201 points on $-1 \leq x \leq 1$. You should find that the polynomials wiggle too much, that the resulting residual overestimates the backward error in this (quite smooth) solution substantially.

16.2. Without first separating $u(t,x)$ into real and imaginary parts, solve Burgers' equation (16.4) with the initial condition $u(0,x) = \exp(i\pi x)$ by the method of lines, by using the capacity of `ode15s` to integrate complex-valued differential equations. As in the text, use periodic boundary conditions. Graph the real and imaginary parts of the solution. Compare your graphs to the ones in the text. Explain any differences you see.

The equation has singularities if $t = -\exp(-1 - i\pi x)/(i\pi)$, as discussed in the text. For real x, these are located on a complex t-circle of radius ${}^1/_{(e\pi)}$. That is, the solution should be singular also if $t = -{}^1/_{(e\pi)}$. Switch the orientation of t, that is, integrate $u_t - uu_x = 0$ subject to $u(0,x) = \exp(i\pi x)$, and show that again waves break and peak.

16.3. Use equally spaced points on $-1 \leq x \leq 1$ to solve (16.1) by the method of lines, with a single degree-$(n+1)$ polynomial interpolating the $n+2$ points used. Use `genbarywts` to generate the differentiation matrix. Show that this approach fails to give a good solution, because of the interpolation error on equally spaced points. One way to do that is to plot the residual for a suitable time, and show that it is growing unacceptably large; show also that increasing n does not help.

16.4. Show that the finite-difference formulæ used in Algorithms 16.1–16.2 are fourth order. Implement them in MATLAB as `first.m` and `second.m`, respectively. Test them on (say) $\sin(x)$ for various equally spaced meshes, say with Fibonacci numbers of mesh points, and plot the resulting maximum error on a log–log scale, together with a line showing $O(1/n^4)$ slope, to verify that your implementation gives fourth-order accuracy at least for mesh widths large enough that rounding errors do not interfere.

16.5. Use `ode15s` and compact finite differences as in the text to solve the one-way wave equation (16.8). Can you explain the amplified numerical errors near $t = 8$, $x = 0$? If you have access to a copy of Trefethen (2000), compare with the spectral solution given there.

16.6. Verify Eq. (16.28).

16.7. Perhaps using MAPLE, use multivariate Taylor series to show that the *Mehrstellenverfahren* (16.33) are fourth-order accurate. Compare, if you have access to it, the formulæ in Collatz (1966 Table VI). Note that Henrici (1979b) suggests that if Fourier methods are to be used, there is an easier way in that the methods correspond in Fourier space to approximating the exponential.

Investigations and Projects

16.8. Use Algorithm 16.3 (either the crude implementation in the text or your own, better and faster one) to solve $-\Delta u = \sin^2 \pi x \sin^2 \pi y$ on the square $[0,1]^2$, with Dirichlet boundary conditions $u = 0$ on the boundary. This time no analytical solution is provided (you might be able to find one, though), and therefore to analyze the error in your solution, you are obliged to interpolate the computed solution, differentiate it, and check that the residual is small. Do you get $O(h^2)$ error behavior, as expected?

16.9. Use Chebfun to solve the one-space-dimension reaction–diffusion equation described in `http://www2.maths.ox.ac.uk/chebfun/examples/pde/pdf/ReactDiffSys.pdf`. Note that solution uses the overloaded (Chebfun) version of `pde15s`, which uses spectral spatial discretization on Chebyshev–Lobatto points as described in this chapter. Is the equation well-conditioned?

16.10. Table VI in Collatz (1966) contains a *sixth*-order-accurate compact finite-difference approximation for Δu. Replace the generation of **A** and **B** in the MATLAB code `compactlaplacian.m` by appropriate matrices that use this formula with periodic boundary conditions on the square, and solve the Schnakenberg equations using the method of this text. Is this cheaper for the same accuracy or more accurate for the same computing time than the $O(h^4)$ method?

16.11. Show that the Schnakenberg equations (16.31) are well-conditioned.

16.12. Use MATLAB to solve the three-space-dimension and time PDE

$$u_t = u_{xx} + u_{yy} + u_{zz} + u - u^3 \tag{16.38}$$

in the cube $[0,1]^3$, with periodic boundary conditions. Use $u(0,x,y,z) = \sin(2\pi(x+y+z)) + \cos(2\pi(x-y-2z))$ as your initial condition. How small is your residual?

Part V
Afterword

The objective of this book was to present a unified view of numerical computation, insofar as that is possible, for areas of numerical analysis as different as floating-point arithmetic, numerical linear algebra, function evaluation, rootfinding, interpolation, numerical differentiation and integration, and numerical solutions of differential equations. Given that our intended audience includes people studying to be applied mathematicians, scientists, and engineers, our guiding principle has been that numerical methods should be discussed as part of a more general practice of mathematical modelling. Thus, we have presented several topics in numerical analysis, all from the point of view of backward error analysis. The *motto* of backward error analysis—that is, that a numerical method's errors should be analyzable in the same terms as whatever physical (or chemical or biological or social or what-have-you) modeling errors—should now be second nature to you.

The main strengths of backward error analysis should also now be obvious. First and foremost, one sees that the computed solution is reliable and faithful to the mathematical model if the residual is small and can be interpreted in terms of the context of the model. We have summarized the methodology with a diagram:

$$\begin{tikzcd}
\text{input space} & \text{output space} \\
x \arrow[r, dash] \arrow[d, dash, "\text{backward error}"'] \arrow[dr, dash, dashed, "\hat{\varphi}"] & y = \varphi(x) \arrow[d, dash, "\text{forward error}"] \\
x + \Delta x \arrow[r, dash] & \hat{y} = \varphi(x + \Delta x) = \hat{\varphi}(x)
\end{tikzcd}$$

Here, $\hat{\varphi}$ is the problem exactly solved by the numerical method. Of course, one needs to know how sensitive the model is to physically meaningful modeling or measurement errors; this will allow judgement to be passed about the numerical, computational errors too. At its best, backward error analysis gives an entirely satisfactory validation of the computation, as good as any scientific endeavour may require. Second, our emphasis on the notion of residual also makes it possible to first compute a solution using some method, and only afterward use this computed solution to assess whether it is sufficiently accurate. This a posteriori mode of error analysis offers multiple advantages and is a natural complement to the backward error perspective.

Nonetheless, it would be disingenuous to pretend that backward error analysis is a panacea, and we don't want to give that impression, even if it is the approach we prefer overall. To begin with, we have mentioned examples such as the outer product, where a backward error analysis can't be performed at all: The computed outer product of two 3-vectors cannot be the exact outer product of two perturbed 3-vectors unless, miraculously, the nine rounding errors are correlated in such a way that they can be explained by merely six perturbations. Generically, this won't happen. Another objection is that the computation of the residual usually overestimates the backward error, sometimes significantly.

There are other limitations. One of which we haven't taken much notice in this book is cost. To do a backward error analysis, you have to compute a residual unless you are lucky enough to be using an algorithm and an arithmetic such that the computed answer comes with an a priori guarantee of small backward error. But computing the residual takes time and effort; it is often cheaper, but it sometimes requires as much as or more work than solving the problem in the first place. We usually discount this disadvantage because the alternative—computing a putative answer and hoping for the best—is not professionally acceptable. However, for very large problems, we admit that computing a residual everywhere might be prohibitively expensive; in such cases, sampling the residual might be a viable alternative. Clearly, one does the best one can.

Taken together with the asymptotic nature of the sensitivity analysis via the theory of conditioning, namely, the linear analysis that approximates

$$\varphi(x+\varepsilon) \sim \varphi(x) + \varphi'(x)\varepsilon + O(\varepsilon^2)$$

and ignores terms of order $O(\varepsilon^2)$, this means that the style of backward error analysis presented in this book gives estimates, not bounds, for errors[6]; there may be monsters hidden in the $O(\varepsilon^2)$ symbols. Still, the method is surprisingly effective.

Of course, this requires smoothness. Functions that are not differentiable at a point can be extraordinarily sensitive to changes in the problem: One may need to restrict changes ε to be such that structure (the so-called pejorative manifold) is taken into account.

Another difficulty with backward error results is that one is really interested in a backward error result for the whole problem. One is much less interested in backward error results for subproblems computed along the way to the solution of the whole problem. Here, indeed, we can run into a snag: If our outer problem is decomposed into a sequence of subproblems, say

$$P = P_1 \circ P_2 \circ \cdots \circ P_n\,,$$

then even if each of the subproblems can be solved with good backward error, it does not mean that the whole problem can be solved with good backward error. Indeed, backward error is not generally preserved under composition. But then, the same can sometimes be said for forward error analysis.

Finally, and perhaps most importantly, the context of the problem and the nature of allowable perturbations must be considered. It's all very well to say that one has the exact solution to

$$\dot{y} = f(y) + \varepsilon v(t)\,,$$

where $v(t)$ is a piecewise polynomial with $\|v\| \leq 1$ and $\varepsilon = 10^{-5}$, but what if time-dependent perturbations make no sense in the underlying application? A better model might be $\dot{y} = f(y) + h^4 F_y(y) + O(h^6)$, for instance. The context may truly matter. But that should not deter applied mathematicians, scientists, and engineers, for whom the context matters anyway. For writers of general-purpose software such

[6] Except for the theory of pseudospectra and pseudozeros, which gives a full nonlinear analysis.

as MATLAB's command \ or `ode15s`, that context is not knowable in advance. Human judgment seems to be necessary. In this case, aiming instead for small forward error (if at all possible) may lead to more satisfactory code.

In conclusion, we think that backward error analysis has proved to be a valuable tool for people doing scientific and engineering computing. Even with its limitations, we know that it will continue to be useful. We hope that it has helped you in developing scientifically and mathematically sound and general thinking habits. About such habits, Uri Ascher once said:

> "It's what a good numerical analyst does anyway."

Part VI
Appendices

We have tried to keep appendices to a minimum, especially since elementary material is now readily available online. However, a book on numerical analysis that doesn't define terms for floating-point arithmetic *at all* would not be as useful as it should be—hence, Appendix A. Gathering the material on floating-point arithmetic in an appendix allows us to present some key ideas systematically without major discontinuity in the text.

As for Appendix B, we expect that most graduate students will have had a course in complex variables, but a review is helpful. Moreover, there is a disconnect between a textbook presentation of branch cut issues and the de facto standardization of branch cuts and closures on branch cuts in computer languages, so this appendix discusses those standardizations. Some of the concepts reviewed here, especially the residue theorem, play an important role through most of the book.

Finally, Appendix C reviews a few elementary notions of linear algebra; one reason to include this material was to introduce our notation in a transparent way. The short list of properties of vector and matrix norms is intended to be convenient for the linear algebra chapters, and some of this might not have been previously encountered by all readers. We refer to the short section on the Schur complement on many occasions, and so the reader should make sure that this result is familiar.

Appendix A
Floating-Point Arithmetic

According to James Gosling (1998), creator of Java, “[Ninety-five percent] of the folks out there are completely clueless about floating-point.” To make sure you’re not one of them, this appendix gives a very brief overview of floating-point arithmetic (FPA) and its differences from arithmetic over the real and complex numbers. For more details and a more thorough treatment, see Goldberg (1991), Overton (2001), Ercegovac and Lang (2004), Muller et al. (2009), or Brent and Zimmermann (2011).

One distinctive feature of computer-assisted mathematics is that instead of computing with elements of continuous fields (e.g., the real or the complex numbers), one operates on a discrete, finite set of digital objects. The real numbers, for instance, can essentially only be represented by *infinite* strings of digits, and the operations on them can be seen as acting on those infinite strings. However, computation on a digital computer cannot carry out such operations, since only finite strings of digits can be manipulated. This does allow *some* real numbers to be worked with, the so-called computable reals, but not all. Therefore, instead of trying to work with the entire continuum, one works with only a discrete finite subset. This limitation has important consequences.

There are many possible approaches, some of which are discussed in Knuth (1981), including exotic but interesting arithmetics that include exact rational arithmetic with limited size denominators and numerators. But by far, the most widespread system is floating-point arithmetic, and specifically the floating-point arithmetic covered by the IEEE-754 standard.

Many nice number-theoretical properties—such as associativity of operations—are typically not satisfied in floating-point arithmetic. One finds, therefore, that FPA is a quite different type of arithmetic; key concepts, such as roundoff error, underflow, and overflow, emerge when we switch to floating-point operations. As a result, for FPA we need an independent mathematical theory that explains how we can accurately represent and operate on real numbers with finite strings of digits.

One of the main challenges is to guarantee that floating-point operations are correctly rendering results, where “correctly” is measured on the basis of standard arithmetic, insofar as that is possible.

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0,

A.1 Number Representation

A digital number system associates digital representations with numbers. To take a simple example, we can associate the number ”four” with the decimal representation “4,” with the binary representation “100,” or with the roman numeral “IV.” Such digital representations are also called *numerals*. Thus, a digital number system is composed of a set of numbers N, a set of numerals W, and a representation mapping $\phi : N \to W$. If the association is one–one, we can similarly write $\phi^{-1} : W \to N$. In this case, we could write, for example, $\phi(\text{four}) = \text{IV}$ and $\phi^{-1}(\text{IV}) = \text{four}$. As we see, ϕ associates the number ‘four’—a uniquely identifiable element of a number structure (e.g., a ring or a field)—with its digital representation in Roman numerals.

Integer and Rational Representation

A nonnegative integer $x \in \mathbb{N}_0$ is represented by a *digit-vector*

$$[d_{n-1}, d_{n-2}, \dots, d_1, d_0] ,$$

from which we obtain the more standard $\phi(x) = d_{n-1}d_{n-2}\dots d_1 d_0$ by concatenating the elements of the digit-vector. The concatenated digit-vector is what we called a *numeral*. The number of digits n is called the *precision*. Each d_i belongs to a set D, the set of digits available for the representation (e.g., $\{0,1\}$ in the binary case). If D contains m elements, it will then only be possible to form $|W| = m^n$ distinct numerals. Since m and n are finite, $|W|$ is also finite. This is much less than the $\aleph_0$ integers or rationals that we want to represent! In fact, each numeral will be used to represent many numbers.

The cases we are interested with here are the so-called weighted or positional representations of *fixed radix*. If we let r be the radix, we associate a number $x \in N$ with a numeral $w \in W$ by a mapping ϕ such that

$$\phi(x) = \phi\left(\sum_{i=0}^{n-1} d_i r^i\right) = d_{n-1}d_{n-2}\dots d_1 d_0 = w\,. \tag{A.1}$$

The most familiar system—the decimal system—has $r = 10$ and $D = \{0,1,2,\dots,9\}$. For instance, in the decimal representation, we have

$$\phi(\text{nine hundred eighty-four}) = 9\cdot 10^2 + 8\cdot 10^1 + 4\cdot 10^0 = 984.$$

Note that, for computer implementation, $r = 2$ and $D = \{0,1\}$ are usually favored.

From the definitions above, it follows that the range of nonnegative integers that can be exactly represented with a precision-n and radix-r number system is $[0, r^n - 1]$:

$$0 = \sum_{i=0}^{n-1} 0\cdot r^i \le x \le \sum_{i=0}^{n-1} (r-1)\cdot r^i = r^n - 1$$

To represent both positive and negative integers—namely, to represent *signed* integers—we need a way to determine the sign of the integer represented by a given numeral. There are two main types of representations:

1. use a bit for the sign and the rest for the magnitude;
2. use all the bits for the magnitude and add a bias.

In the former case, we reserve a digit in the word to determine the sign. 0 usually represents '$+$,' while 1 represents '$-$.' Then, for an n-bit word, the range is

$$[-r^{n-1}+1,\, r^{n-1}-1]\,. \tag{A.2}$$

In the latter case, all the bits are determining the magnitude; the value represented is then the value it would represent under the standard positional representation, minus a certain bias B. A standard bias for an n-bit radix-r representation is $B = r^{n-1}-1$.[1] Then, for an n-bit word, the range is $[-r^{n-1}+1,\, r^n-1-(r^{n-1}-1)]$; that is,

$$[-r^{n-1}+1,\, r^{n-1}(r-1)]\,. \tag{A.3}$$

In the binary case $r=2$, this just results in $[-r^{n-1}+1,\, r^{n-1}]$. In comparison to the sign-and-magnitude representation, we see that it provides us with one additional value.

Note that rational numbers can be written in the form

$$d_{n-1}d_{n-2}\ldots d_2 d_1 d_0\,.\,d_{-1}d_{-2}\ldots d_{-f-2}d_{-f-1}d_{-f}\,.$$

Hence, we see that it is simply a pair of words with respective precisions n and f. The first word is the *integer part*, and the second is the *fractional part*. Provided that the radix is the same for both words,

$$x = \sum_{i=-f}^{n-1} d_i \cdot r^i\,. \tag{A.4}$$

It is usually assumed that the integer part is a signed integer, whereas the fractional part is a nonnegative integer.

Floating-Point Representation

We now introduce the notion of a *floating-point number*. A floating-point number is any real number that has an *exact* floating-point representation. Formally, if we let $\mathbb{F}$ be the set of floating-point numbers, W the set of floating-point words (to be defined), and $\phi : \mathbb{F} \to W$ some floating-point representation mapping (to be defined), we have

$$\mathbb{F} = \left\{ x \in \mathbb{R} \;\middle|\; \phi(x) = w \text{ for some } w \in W \right\}\,. \tag{A.5}$$

[1] We will always use this bias, unless otherwise specified.

Since there will be, once again, only a finite number of exactly representable numbers, $\mathbb{F}$ is finite. As a result, $\mathbb{F}$ has unique minimal and maximal elements, for both the positive and negative numbers. Consequently, we find that the floating-point number "line" can be represented as in Fig. A.1.

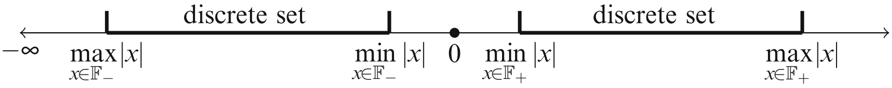

discrete set
discrete set
$-\infty$
$\max_{x\in\mathbb{F}_-}|x|$
$\min_{x\in\mathbb{F}_-}|x|$
0
$\min_{x\in\mathbb{F}_+}|x|$
$\max_{x\in\mathbb{F}_+}|x|$

Fig. A.1 Floating-point number line. The intervals $(-\infty, \max_{x\in\mathbb{F}_-}|x|)$ and $(\max_{x\in\mathbb{F}_+}|x|), \infty+)$ are called, respectively, negative and positive overflow. Similarly, the intervals $(\min_{x\in\mathbb{F}_-}|x|, 0)$ and $(0, \min_{x\in\mathbb{F}_+}|x|)$ are called negative and positive underflow

An n-bit floating-point representation has two components:

1. an m-bit word ($0 < m < n$) called the *mantissa* or *significand*,[2] representing a signed rational number M with sign S_M (using a sign-and-magnitude representation);
2. an $n-m$-bit word called the *exponent*, representing a signed integer E (using a biased representation).

The choice of type of representations for the signed integers M and E follows the IEEE standard. We will assume that the mantissa and the exponent have the same radix r. The corresponding floating-point number in a base-b system is then

$$M \times b^E \,. \tag{A.6}$$

The IEEE standard requires that M be normalized, that is, of the form $\pm 1.F$, where F is the fractional part. It is then not required to use a bit for the integer part (the "1" is said to be a *hidden bit*), and the m bits for the mantissa are used for the sign and the fractional part.[3] In the rest of this appendix, we will deal with such normalized numbers. In Fig. A.2, one can see the bits partition for a representation of a floating-point number with 32 bits.

Bit type	S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
Bit number	1	2							9	10																						32

Fig. A.2 Thirty-two-bit word for a floating-point number. The biased exponent occupies 8 bits and the mantissa occupies 24 bits (one for the sign, 23 for the fractional part or the normalized mantissa)

[2] m is often used for the length of the unsigned mantissa. It is, of course, just a notational convention; one must simply keep track of all the $+1$ and -1 in the exponents to have agreeing results.

[3] We should note that restricting representation to normalized M makes it impossible to represent some very small numbers. The IEEE standard also defines *unnormalized* representations (also called denormals or subnormals) to deal with those numbers. However, we will ignore this refinement in this appendix.

Values Represented

As we have seen, the set of (normalized) floating-point representations W is determined by three numbers n, m, r. The set of floating point numbers $\mathbb{F}$ is, in turn, determined by the set of representations W and by a base b. We can then write $\mathbb{F}(n,m,r,b)$. For a given number system $\mathbb{F}(n,m,r,b)$, one can list all the values that can be represented in a finite table, as in Table A.1. The values computed in Table A.1 are represented on the "$\mathbb{F}(7,4,2,2)$ number line" in Fig. A.3. One sees at a glance that the numbers are not uniformly distributed. In fact, they are much more densely distributed close to 0. As is easy to see, the distribution depends on n, m, b and r.

Table A.1 $\mathbb{F}(7,4,2,2)$: represented values when $n = 7$, $m = 4$, $r = 2$, $b = 2$. The words for the mantissa and the exponent are written in binary. For convenience, the values are in decimal. Only the positive values of the mantissa have been listed

Exponent Mantissa	−11	−10	−01	±00	01	10	11
1.000	0.125	0.25	0.5	1	2	4	8
1.001	0.140625	0.28125	0.5625	1.125	2.25	4.5	9
1.010	0.15625	0.3125	0.625	1.25	2.5	5	10
1.011	0.171875	0.34375	0.6875	1.375	2.75	5.5	11
1.100	0.1875	0.375	0.75	1.5	3	6	12
1.101	0.203125	0.40625	0.8125	1.625	3.25	6.5	13
1.110	0.21875	0.4375	0.875	1.75	3.5	7	14
1.111	0.234375	0.46875	0.9325	1.875	3.75	7.5	15

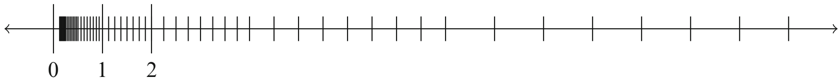


Fig. A.3 The $\mathbb{F}(7,4,2,2)$ positive number "line"

Range and Machine Epsilon

Perhaps the main advantage of floating-point numbers is their near-automatic scale invariance. Working with a collection of numbers S is very much the same as working with $10^3 S$ or with $S/10^3$, so long as overflow does not occur. Indeed, if the numbers are scaled by the base b, there will be no rounding effects as long as overflow and underflow do not occur. Prior to the adoption of floating-point, fixed-point approximations were used together with manual scaling as needed when the computations demanded.

Another, related, advantage of floating-point numbers is their *range*, which is much larger than the range of fixed-point numbers. As we have seen, the largest number that can be represented by an n-bit radix-r word is $r^n - 1$, resulting in the range $[0, r^n - 1]$. However, the largest floating-point number in the set $\mathbb{F}(n,m,r,b)$ is $M_{\max} \cdot b^{E_{\max}}$, where $M_{\max}$ and $E_{\max}$ are, respectively, the largest mantissa and the largest exponent. In the normalized case, since $M_{\max} = 1.F_{\max}$, we obtain

$$\begin{aligned} M_{\max} &= 1.F_{\max} = 1 + \frac{r^{m-1}-1}{r^{m-1}} \\ E_{\max} &= r^{n-m-1}(r-1) \\ \max\mathbb{F} &= \left(1 + \frac{r^{m-1}-1}{r^{m-1}}\right) \cdot b^{r^{n-m-1}(r-1)} \, . \end{aligned} \tag{A.7}$$

As an example, the largest 32-bit radix-2 fixed-point word is $2^{32} - 1 \approx 4 \cdot 10^9$. If we make it signed, so that it includes negative numbers, the largest one will be $2^{31} - 1 \approx 2 \cdot 10^9$. However, the largest base-2 floating-point number represented by 32-bit words partitioned as in Fig. A.2 will have mantissa $+1$ followed by 23 ‘1’s for the fractional part and biased exponent with 8 ‘1’s, that is,

$$\left(1 + \frac{2^{23}-1}{2^{23}}\right) \cdot 2^{2^7} \approx 7 \cdot 10^{38} \, .$$

We see that the range is much larger. In double precision, which is the default in MATLAB, the range is larger still: `realmax` is $2^{1024} \approx 1.7977 \times 10^{308}$ and `realmin` $= 2^{-1022} \approx 2.2251 \times 10^{-308}$. This range suffices for a great many applications, although both overflow and underflow do occur.

Software floats in MAPLE (the so-called arbitrary precision floating-point numbers) have an even larger range because the base b in MAPLE is 10 and the radix is the size of a word; the actual maximum and minimum in the range depend on the machine type used, and on the machine this is being typed on, `MAX_REAL` is $1 \times 10^{2,147,483,646}$ and `MIN_REAL` is the reciprocal of that. The maximum number of decimal digits is “only” $268,435,448$, but, really, computation with a quarter of a billion digit numbers takes too long on a standard machine to be useful. In practice, one sets `Digits` to be 15 (which makes all the computations happen in IEEE double precision similar to MATLAB’s, except conversion to decimal happens more often), or 30 or 100 or so when extra precision is required. One notices a great slowdown in computation moving from `Digits:=15` to `Digits:=16`, which marks the boundary between hardware floats (IEEE doubles) and software floats.

An important notion for the analysis of floating-point error is the *unit in the last place*, or *ulp*, which is the difference of two consecutive values of the mantissa. Since the values of the mantissa are uniformly distributed, *ulp* is a constant. For given values of m and r, we find that

$$ulp = \left(1 + \frac{r^{m-1}-x}{r^{m-1}}\right) - \left(1 + \frac{r^{m-1}-(x+1)}{r^{m-1}}\right) = r^{-m+1} \, . \tag{A.8}$$

Assuming $b = r$, it follows immediately that the difference between two floating-point numbers x_1 and x_2 with the same exponent E will be given by

$$\Delta x = x_1 - x_2 = (M_1 - M_2)r^E = r^{-m+1} r^E = r^{E-m+1} . \tag{A.9}$$

For normalized floating-point numbers between 1 and 2, where $E = 0$, we simply obtain *ulp*.

The spacing of floating-point numbers when $E = 0$ is called the *machine epsilon*, which is denoted 'ε_M.' Each number system has its value of ε_M, and they generally differ.

Example A.1. In the case of $\mathbb{F}(7,4,2,2)$ discussed above, $\varepsilon_M = 2^{-4+1} = 0.125$. ◁

Example A.2. To demonstrate the machine epsilon in MATLAB, we look for the smallest power of two that, when added to 1, gives a number larger than 1. This construction makes no sense in the real numbers and represents an important difference of floating-point numbers. The statements (which return the logical values 1 for "true" and 0 for "false")

```
>> 1.0 + 2^(-52) > 1

ans =
     1
```

and

```
>> 1.0 + 2^(-53) > 1

ans =
     0
```

show that the machine epsilon in MATLAB is $\varepsilon_M = 2^{-52} \approx 2.2 \times 10^{-16}$.

In MAPLE, for example, at its default setting of `Digits:=10`, we find that any ε that rounds to $10^{1-\text{Digits}}$ will satisfy $1 + \varepsilon > 1$, so we can take $\varepsilon = 5 \times 10^{-\text{Digits}}$, although the rounding on input runs into the table maker's dilemma. This rule works for higher settings of `Digits`, as well. ◁

An important related value is the maximum relative error due to the floating-point representation, called the *roundoff level* or *unit roundoff*, which is just $\varepsilon_M/2$. We will denote this quantity by '$\mu_M = \varepsilon_M/2$'. Since roundoff is the main source of arithmetic error, this quantity will be used throughout this book as a unit of error.

Complex Floats

A complex floating-point number is simply a pair of floating-point numbers, usually $z = (x,y)$, where x and y are real floats. In MATLAB, a complex number can be formed by the constructor `complex(a,b)` or by the operation `a + 1i*b`. Note the use of the construct `1i`, which ensures that $\sqrt{-1}$ is meant by the symbol `i` even if the variable `i` has been used previously for another purpose (such as a loop index, as is common: `for i=1:10,` etc.).

Complex arithmetic is defined so that the usual rules hold, insofar as possible: $(0,y)\cdot(0,y) = (-y^2,0)$, for example, provided that overflow or underflow does not happen. Since $(x,y) \otimes (u,v) = (x \otimes u \ominus y \otimes v, x \otimes v \oplus y \otimes u)$ involves several real floating-point operations, there are more opportunities for rounding errors in complex floating-point arithmetic. The bounds for rounding error discussed here and in Chap. 1 need to take this into account, therefore. See also Higham (2002).

The IEEE Standard 754

Some additional constraints are given by different floating-point representations. The current standard for floating-point arithmetic has been developed by the IEEE (Institute of Electrical and Electronics Engineers). The *single*-precision format represents numbers with 32 bits, while the *double*-precision format represents them with 64 bits. MATLAB uses the double-precision format by default, and so we will present this format in this section.

The number system associate with the IEEE standard 754 double-precision format is basically just $\mathbb{F}(64,53,2,2)$ with a few tweaks. The radix and the basis are both 2, which is the standard practice for floating-point arithmetic (this base minimizes so-called wobble). The 64 bits are partitioned as follows:

- mantissa: $m = 53$;
- Eeponent: $n - m = 11$.

As mentioned before, the mantissa is a normalized signed integer with a sign-and-magnitude representation and the exponent is a signed integer with a biased representation. Thanks to the hidden bit, this format has precision $p = 53$. The range of the values represented by the mantissa is

$$\left[1,\, 1+\frac{r^{m-1}-1}{r^{m-1}}\right] = \left[1,\, 1+\frac{2^{52}-1}{2^{52}}\right] \approx [1,2)\,. \tag{A.10}$$

The bias of the exponent is the standard bias:

$$B = r^{n-m-1} - 1 = 2^{10} - 1 = 1023 \tag{A.11}$$

Consequently, the range of the values represented by the exponent is

$$[-r^{n-m-1}+1,\, r^n - 1] = [-2^{10}+1,\, 2^{10}] = [-1023,\, 1024]\,. \tag{A.12}$$

However, -1023 and 1024 are reserved to denote negative and positive infinity, that is, `-Inf` and `Inf` in MATLAB. As a result, the range of the exponent is $[-1022,\, 1023]$.

Consequently, the range of the positive double-precision floating-point numbers is

$$\begin{aligned}\left[M_{\min}\cdot 2^{E_{\min}},\, M_{\max}\cdot 2^{E_{\max}}\right] &= \left[2^{-1022},\, \left(1+\frac{2^{52}-1}{2^{52}}\right)\cdot 2^{1023}\right] \\ &\approx \left[2.2\cdot 10^{-308},\, 1.8\cdot 10^{308}\right]\,. \end{aligned} \tag{A.13}$$

MATLAB calls the limit points of this range `realmin` and `realmax`. The machine epsilon—MATLAB calls this value `eps`—is

$$\varepsilon_M = r^{-m+1} = 2^{-52} \approx 2.2 \cdot 10^{-16}. \tag{A.14}$$

Finally, since zero has no direct representation in this system (due to normalization), the word with $M = 1.0$ and $E = 0$ is used to represent it.

Signed Zero

Perhaps surprisingly, zero can have a sign in the IEEE standard. See Kahan (1986), or perhaps the Wikipedia entry on ”signed zero” for a description of the need for this. In MATLAB, which does *not quite* comply with the IEEE standard—there is a document at the Mathworks website that points out that they use several different packages that have conflicting implementations—and although the signed zero is present, it is not as much used as it could be. With hexadecimal format turned on, so we can see the results, we find that

```
>> format hex
>> 0

ans =
   0000000000000000
```

and

```
>> -0

ans =
   8000000000000000
```

The leading four bits are $8_{16} = 1000_2$ (the subscript denotes the basis). However, signed zeros cannot be used for either the real part or the imaginary part of a complex number in MATLAB (except the number $0+0i$, which can be either ± 0), which is a pity. This can be checked by executing

```
>> complex( -3, 0 )

ans =
   c008000000000000
```

and

```
>> complex( -3, 1 )

ans =
   c008000000000000   3ff0000000000000i
```

and finally,

```
>> complex( -3, -0 )

ans =
   c008000000000000
```

It is not possible to have $-3-0i$ for example, which would be useful in taking logarithms or dealing with other branch cuts: $\ln(-3-0i)=\ln(3)-i\pi$, whereas $\ln(-3)=\ln(3)+i\pi$.

In MAPLE on the other hand, there is indeed a signed complex zero, and this is occasionally useful.

NaN and Infinity Arithmetic

The exponent pattern corresponding to 1024 in decimal after subtracting the bias is `7FF`, that is, using 3 bits for $7=111_2$ and the remaining eight bits in the exponent being 11111111_2 or `FF` in hexadecimal. Negating this turns the leading bit to a 1, making `FFF`, not `7FF`. Thus, infinities print as follows in format `hex`:

```
>> -inf

ans =
   fff0000000000000
```

and

```
>> inf

ans =
   7ff0000000000000
```

A little bookkeeping shows that there are as-yet-unused bit patterns left over in the scheme. Some of these are taken up as NaNs, or "Not A Number"s, and as *denormalized numbers*. NaNs arise in various arithmetic exceptions: $\frac{0}{0}$, for example. Denormalized numbers are numbers smaller than `realmin` and thus cannot be represented with a hidden bit as the other floats are. Implementing denormalized numbers allows gradual underflow, which improves some algorithms. Executing the following in MATLAB is informative:

```
>> 0/0

ans =
   fff8000000000000
```

Arithmetic with infinities and with NaNs tries to be logical and consistent, but most of all useful. One over (positive) 0 is `Inf`, and one over (negative) 0 is `-Inf`. Zero times `Inf` is a NaN, and so on. The first use for NaNs that you will likely encounter is in allowing functions executed on vector arguments to have an occasional undefined value without causing an error break in the computation; one can then plot the results without overly fussing.

Example A.3. The MATLAB commands

```
x = linspace( 0, 4, 101 );
y = sin(pi*x)./(pi*x);
plot( x, y, 'k' ), set(gca,'fontsize',16), xlabel('x'), ylabel( '
    sin(\pi␣x)/(\pi␣x)' )
```

produces a nice plot, in spite of the fact that `y(1)` is a NaN produced by dividing by 0. ◁

Summary of Floating-Point Surprises

1. Addition is not associative or associative: $1+\varepsilon^2-1$ may give 0 or ε^2 depending on the ordering, while $B-B+1$ for large B may give 0 or 1 depending on the association $(B-B)+1$, or $B+(-B+1)$ when the intermediate result rounds to $-B$. Moreover, adding positive numbers in one order may give a different answer to the result of adding in the opposite order even though no cancellation occurs.
2. There are actual infinities, or rather symbols for numbers too large to represent, that nonetheless can be further manipulated and occasionally can give useful answers; having $1/\infty$ simplify to 0 makes some continued fraction formulae simpler to program, for example.
3. The set is not closed under addition, for example: Adding two numbers may overflow, giving `inf`.
4. There can be gradual underflow to 0, using denormalized numbers:

   ```
   >> format hex
   >> realmin

   ans =
      0010000000000000

   >> realmin/100

   ans =
      000028f5c28f5c29
   ```

   These numbers are less precise than other floats because some of their leading bits are zero. This allows more graceful behavior for some algorithms.
5. Underflow to zero permits *zero divisors* (for example, `realmin`·`realmin` returns the answer 0 even though neither factor is zero). Thus, floating-point numbers are not a field. Because of the lack of associativity, they're not a group or a ring, either, of course.
6. Zero has a sign. Various computer languages have various ways of detecting or displaying it. In MATLAB, -0 by default displays in exactly the same way as $+0$; you can only see the sign in hex format.

A.2 Operations and Roundoff

The IEEE standard also defines the result of floating-point arithmetic operations (called *flops*, although in counting them to estimate costs, it is usual to say that one flop is one multiplication or division and one addition or subtraction and (possibly) one comparison). It is easy to understand the importance of having such standards! In the last section, the reader might have noticed that floating-point representation works just like the scientific notation of real numbers—in which numbers are written in the form $a \cdot 10^b$—to the exception that we mostly use base 2 and that both a and b have length restrictions. The same analogy will hold true for the four basic operations.

The four basic operations on the real numbers are functions $* : \mathbb{R}^2 \to \mathbb{R}$, with

$$* \in \{+, -, \times, /\}$$

In floating-point arithmetic, we use similar operations, but using floating-point numbers. The four basic operations on floating-point numbers are thus functions $\circledast : \mathbb{F}^2 \to \mathbb{F}$, with

$$\circledast \in \{\oplus, \ominus, \otimes, \oslash\}.$$

These operations have many different implementations. Given our expectation that floating-point operations return results that are very close to what the real operations would return, the important question is: How do $\oplus, \ominus, \otimes, \oslash$ relate to their counterparts $+, -, \times, /$ in the real numbers? The best-case scenario would be the following: Given a rounding procedure converting a real number into a floating-point number, the floating-point operations always return the floating-point number that is closest to the real value of the real operation. The good news is, if we are only interested with the impact of floating-point arithmetic in applications, there is no need to examine the detailed implementations of the floating-point operations. The IEEE standard guarantees that, for the four basic operations $\oplus, \ominus, \otimes, \oslash$, the best-case scenario is obtained.

Let us formulate this more rigorously. A *rounding* operation $\Box : \mathbb{R} \to \mathbb{F}$ is a procedure converting real numbers into floating-point numbers satisfying the following properties:

1. $\Box x = x$ for all $x \in \mathbb{F}$;
2. $x \leq y \Rightarrow \Box x \leq \Box y$ for all $x, y \in \mathbb{R}$;
3. $\Box(-x) = -\Box x$ for all $x \in \mathbb{R}$.

There are many rounding operations satisfying this definition. In what follows, we will use the rounding to the nearest floating-point number (with ties toward $+\infty$), denoted '$\bigcirc$.' If we let f_1, f_2 be two consecutive floating-point numbers and $x \in \mathbb{R}$ such that $f_1 \leq x \leq f_2$, then $\bigcirc$ is defined by[4]

$$\bigcirc x = \begin{cases} f_1 & \text{if } |x - f_1| < |x - f_2| \\ f_2 & \text{if } |x - f_1| \geq |x - f_2| \end{cases} . \tag{A.15}$$

Then the IEEE standard guarantees that the following equations hold:

$$x \oplus y = \bigcirc(x + y) \tag{A.16}$$

$$x \ominus y = \bigcirc(x - y) \tag{A.17}$$

[4] To consider the cases where x does not lie within the range of the floating-point number system, we need to specify that if $|\bigcirc x| > \max\{|y| : y \in \mathbb{F}\}$ or $0 < |\bigcirc x| < \min\{|y| : 0 \neq y \in \mathbb{F}\}$, the rounding procedure returns, respectively, overflow and underflow.

$$x \otimes y = \bigcirc(x \times y) \tag{A.18}$$

$$x \oslash y = \bigcirc(x/y); \tag{A.19}$$

that is, for $* : \mathbb{R}^2 \to \mathbb{R}$ and $\bigcirc : \mathbb{R} \to \mathbb{F}$, we obtain $\circledast : \mathbb{R}^2 \to \mathbb{F}$ such that

$$\begin{tikzcd}
\mathbb{R}^2 \arrow[r, "*"] \arrow[dr, "\circledast"'] & \mathbb{R} \arrow[d, "\bigcirc"] \\
& \mathbb{F}
\end{tikzcd}$$

These equations jointly mean that the result of a floating-point operation is the correctly rounded result of a real operation.

However, if things are so nice, why do we need error analysis? We need error analysis precisely because it is not always so nice for sequences of operations. Problem 1.13 shows, for example, that

$$((((x_1 \oplus x_2) \ominus x_3) \oplus x_4) \ominus x_5) = \bigcirc(x_1 + x_2 - x_3 + x_4 - x_5) \tag{A.20}$$

does not hold generally. So the big question is: When are compound operations reliable? When no result of guaranteed validity exists, the error analysis must be left to the hands of the user. This leads us to Chap. 1.

Problems

A.1. Write an algorithm to convert integers from the decimal basis to the binary basis, and vice versa. (Hint: You can use Horner's recursive scheme described in Sect. 2.2.1.)

A.2. MATLAB and MAPLE have built-in routines to convert integers from one basis to another. However, they don't have such routines for rational numbers. Write one (it may rely on the built-in routines for integers). Use it to give the first 15 binary digits of π.

A.3. Consider the floating-point number system $\mathbb{F}(128, 113, 2, 2)$, a *quadruple-precision* arithmetic. Find the range of positive quadruple-precision floating-point numbers (i.e., `realmin` and `realmax`), the machine epsilon ε_M, and the roundoff level μ_M for this system. Can you think of any real-world applications that would generate overflow?

A.4. Prove that $\bigcirc$ is a rounding procedure.

A.5. Conversion from binary or hex floats to decimal floats is a headache. Some terminating fractions in decimal do not terminate in binary (e.g., $1/5$). Conversion from binary to decimal and back again therefore may induce rounding errors. How big an error might this be?

A.6. Without looking up the details, describe the representational format for IEEE 754 double-precision numbers.

Appendix B
Complex Numbers

There are many excellent textbooks on complex arithmetic, complex variables, and complex analysis. We recommend, in particular, Henrici (1974). Alternatively, a beautiful geometric treatment is given in Needham (1999). A more standard treatment is given in Levinson and Redheffer (1970) and in Saff and Snider (1993). In this appendix, we cover some important differences between theoretical complex variables and complex variables on computers.

B.1 Elementary Complex Arithmetic

Complex numbers are ordered pairs of real numbers; in particular, the pair $(0,1)$ is called i. They come with the following rules for operations on them, both theoretically and on a computer:

$$(x_1,y_1)+(x_2,y_2) = (x_1+x_2, y_1+y_2) \tag{B.1}$$

$$(x_1,y_1)\cdot(x_2,y_2) = (x_1x_2-y_1y_2, x_1y_2+y_1x_2)\,. \tag{B.2}$$

With those rules, $i^2=(-1,0)$, and in this sense $i=\sqrt{-1}$. On a computer, complex numbers are typically represented as pairs of machine numbers. In MATLAB, they can be entered as `complex(x,y)` or as `x+1i*y` if x and y are floats. Matrices can have complex entries.

In MAPLE, and in other CAS, complex numbers can have a variety of machine-representable reals as parts, but the idea is the same: A complex number is an ordered pair of real numbers with special rules for operations. The syntax for complex numbers in MAPLE (which we have never liked) uses the capital letter `I` for the square root of -1, that is $(0,1)$, instead of a small letter. This can be changed by use of the `interface(imaginaryunit=...)` command.

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0,

B.2 Polar Coordinates and the Two-Argument arctan Function

The rule (B.2) for multiplication of complex numbers is more easily understood in polar coordinates. Consider the complex number $z = (x,y)$ to define the Cartesian coordinates of a point in the plane. The polar coordinates $[\rho, \theta]$ of that point are given by

$$\rho = \sqrt{x^2 + y^2} \tag{B.3}$$

and

$$\theta = \arctan(y,x)\,. \tag{B.4}$$

Note the ordering (y,x), reversing the order in the pair $z = (x,y) = x + iy$. You may have seen the two-argument arctan before, perhaps expressed as $\tan^{-1}({}^{y}\!/\!{}_{x})$, and have been expected to work out for yourself which quadrant the angle was in and therefore get the angle correct in $-\pi < \theta \le \pi$. This is because, of course, ${}^{y}\!/\!{}_{x} = {}^{(-y)}\!/\!{}_{(-x)}$; hence, points in quadrants I and III, II and IV are indistinguishable once this ratio has been taken. The purpose of the two-argument arctan function (just `arctan(y,x)` in MAPLE and `atan2(y,x)` in MATLAB) is to save you the trouble. The two-argument arctan takes the Cartesian coordinates of a point in the plane and returns the polar angle of the point, in the interval $(-\pi, \pi]$.

Remark B.1. By convention, the angle is taken in $(-\pi, \pi]$. This defines the *branch cut* and its *closure* (by which we mean that the angle is continuous or “closed” as you approach the negative real axis from above, or in a counterclockwise direction around the origin) for the angle or “argument” function. This is only a convention and could have been chosen anywhere else (say $[0, 2\pi)$)—but it was not. This convention, called “counterclockwise closure” or “CCC” by Kahan (1986), is by now nearly universal in computer languages. This one little convention has a great many logical implications and standardizes what appears very differently in different textbooks. It’s also the principal reason for writing this appendix. Your favorite textbook may well do things differently than the way in which computers do it! In particular, the wonderful theoretical idea of Riemann surfaces gains almost no traction in a computer world, where a function of z is expected to return one answer and not require a construction of a new algebraic object with many possible values. ◁

In polar coordinates, the multiplication rule (B.2) becomes

$$[\rho_1, \theta_1] \cdot [\rho_2, \theta_2] = [\rho_1 \rho_2, \theta_1 + \theta_2]\,, \tag{B.5}$$

as is easily seen by converting (x_1, y_1) and (x_2, y_2) to polar coordinates and using the trigonometric identities

$$\begin{aligned}\cos(\theta_1 + \theta_2) &= \cos\theta_1 \cos\theta_2 - \sin\theta_1 \sin\theta_2 \\ \sin(\theta_1 + \theta_2) &= \cos\theta_1 \sin\theta_2 + \cos\theta_2 \sin\theta_1\,.\end{aligned}$$

Here we do not care if the angle sum $\theta_1 + \theta_2$ is outside the range $(-\pi, \pi]$, because in translating back to Cartesian coordinates via

$$x = \rho\cos\theta \tag{B.6}$$
$$y = \rho\sin\theta\,, \tag{B.7}$$

all that matters is θ modulo 2π. Sadly, in polar coordinates, addition is now more complicated, so we need both systems of coordinates.

B.3 The Exponential Function

We are now ready to define the exponential function of a complex variable. We write

$$e^z = \exp(z) := \sum_{k=0}^{\infty} \frac{z^k}{k!}\,. \tag{B.8}$$

In standard textbooks, for example, the beautiful Henrici (1977), we find proofs that this series converges for all finite z and that this is therefore well defined. From this definition, with some work, we may deduce Euler's formula:

$$e^{x+iy} = e^x(\cos y + i\sin y)\,, \tag{B.9}$$

from which many properties of the exponential function can be shown, including the relation between the Cartesian and polar coordinates:

$$z = x + iy = \rho\cos\theta + i\rho\sin\theta = \rho e^{i\theta}\,. \tag{B.10}$$

The property that concerns us most here is that

$$e^{2\pi ik} = \cos(2\pi k) + i\sin(2\pi k) = 1 \tag{B.11}$$

for any integer k: That is, the exponential function is many-to-one. Therefore, its *functional inverse* is multivalued. We look at it in the next section.

B.4 The Natural Logarithm

We define the *principal branch* of the natural logarithm of $z = x + iy = \rho e^{i\theta}$ to be (from the polar coordinate form)

$$\ln z = \ln \rho e^{i\theta} := \ln\rho + i\theta = \ln\rho + i\arctan(y,x)\,. \tag{B.12}$$

This therefore inherits the same branch cut as the two-argument arctan function, namely, along the negative real axis, with closure from above. See Fig. B.1a,b. We have indicated closure on the top of the negative real axis in the domain of the logarithm by marking it as a solid line and putting a gap underneath to indicate

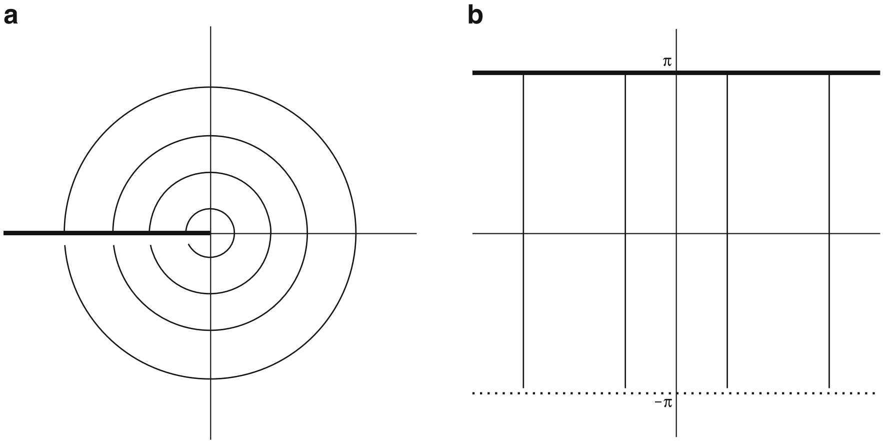


Fig. B.1 Complex logarithm. The branch cut is identified by a *solid line* and closed at π. (**a**) *Circular arcs* in the complex domain of logarithm. (**b**) The principal complex range of logarithm

that the domain is open there. The corresponding lines in the principal range of the logarithm are not circular arcs but rather straight-line segments. Closure is again indicated by a mixture of solid lines and dashed lines. For other visual descriptions of branch cut closures, see Steele (1990). For more discussion of these closures, see Corless et al. (2000).

Once we have defined a logarithm, we may define general powers as follows:

$$z^a := e^{a \ln z}. \tag{B.13}$$

It turns out to be hard to write a more efficient general complex powering operation than just using this definition.

Because of Eq. (B.11), we could equally well have chosen

$$\ln_k z = \ln z + 2\pi i k \tag{B.14}$$

for any other integer k to be our canonical logarithm. We do not and no one else has either. Following this, the de facto standard, we choose $k = 0$, and thus $-\pi < \theta \leq \pi$, to be the one. Every computer algebra language and numerical language follows this standard and takes the complex logarithm to have its imaginary part (also called the "argument" of z) in this range. With this definition, $(-8)^{1/3} = 1 + i\sqrt{3}$, and not -2. If you wish *real-valued* cube roots (or other rational roots) in MAPLE, use `surd`.

Remark B.2. Several rules that we all learned in high school that are valid for positive reals are not necessarily valid for complex numbers. In particular, it is not true that $\ln(ab) = \ln a + \ln b$, or that $\ln z^2 = 2\ln z$, or that $(z^a)^b = z^{ab}$. Corrections to these identities are presented in Table B.1. The corrections here use the *unwinding number*, which is defined by the first entry. ◁

Table B.1 Some identities for the complex logarithm

$$\begin{aligned}
\mathcal{K}(z) &= \left\lceil \frac{\operatorname{Im}(z)-\pi}{2\pi} \right\rceil \\
\ln e^{z} &= z - 2\pi i \mathcal{K}(z) \\
\ln(z_1 z_2) &= \ln z_1 + \ln z_2 - 2\pi i \mathcal{K}(\ln z_1 + \ln z_2) \\
\ln(z^{a}) &= a \ln z + 2\pi i \mathcal{K}(a \ln z) \\
\mathcal{K}(n \ln z) &= 0 \ \forall z \text{ iff} -1 < n \leq 1 \\
(z^{a})^{b} &= z^{ab} e^{2\pi i b \mathcal{K}(a \ln z)} \\
\sqrt{z^2} &= z \operatorname{csgn}(z) := z e^{\pi i \mathcal{K}(2 \ln z)} \\
(z^{n})^{1/n} &= z C_n(z) := z e^{2\pi i \mathcal{K}(n \ln z)/n}
\end{aligned}$$

The final identity defines what is now known as the matrix sector function when the argument z is not a complex number but rather a matrix (Laszkiewicz and Ziętak 2009).

B.5 The Complex Sign Functions

The signum function for real numbers has the definition $\operatorname{signum}(x) = 1$ if $x > 0$, $\operatorname{signum}(x) = -1$ if $x < 0$, and is ambiguous if $x = 0$; in some circumstances $+1$ is wanted, and in others it doesn't matter. Many programs including MATLAB take the signum of 0 to be 0. In MATLAB the signum function is called `sign`. In MAPLE it is called `signum`, and the word `sign` is used for a different function, which can and does cause confusion.

For complex numbers, signum can be generalized as follows:

$$\operatorname{signum}(z) = e^{i\theta}, \tag{B.15}$$

where $z = r\exp(i\theta)$, or $\log z = \log r + i\theta$, so $\theta = \arctan(y, x)$. Again, this is ambiguous if $x = y = 0$, but now the ambiguity is greater than in the real case, and so it seems more reasonable to return a NaN. However, since MATLAB replaces $0 + 0i$ with the real 0, this doesn't happen.

There is another complex signum-like function, called csgn in MAPLE. We can define it with the unwinding number from Table B.1:

$$\operatorname{csgn}(z) := e^{\pi i \mathcal{K}(2 \ln z)}. \tag{B.16}$$

This function is $+1$ in the entire right half-plane $\operatorname{Re}(z) > 0$, is -1 in the entire left half-plane $\operatorname{Re}(z) < 0$, and on the imaginary axis is positive on the upper half

$\mathrm{Im}(z) > 0$ and negative on the lower half $\mathrm{Im}(z) < 0$. At 0, like the signum function, it is ambiguous, although now the choices are only ±1. This function is useful numerically as well, though it is not implemented in MATLAB.

B.6 Trigonometric Functions and Hyperbolic Functions

Once we have the exponential function, it is possible to define the complex trigonometric functions sin and cos by

$$\begin{aligned}\sin z &= \frac{e^{iz} - e^{-iz}}{2i} \\ \cos z &= \frac{e^{iz} + e^{-iz}}{2},\end{aligned}$$

and from thence all the other trigonometric functions: $\tan z := {}^{\sin z}/_{\cos z}$, $\csc z := {}^{1}/_{\sin z}$, $\sec z := {}^{1}/_{\cos z}$, and $\cot z := {}^{1}/_{\tan z}$.

By solving $y = \sin z$ for z, we arrive at an expression for $\arcsin y$ that depends on logarithms. We do not include many here; you can get them with MAPLE, or other CAS. In MAPLE, the equation for $\arctan z$, for example, in terms of logarithms is found by issuing the command

```
evalc(arctan(x +I*y)),
```

which produces

$$\frac{1}{2}\arctan(x, 1-y) - \frac{1}{2}\arctan(-x, y+1) + \frac{1}{4} i \ln\left(\frac{x^2 + (y+1)^2}{x^2 + (y-1)^2}\right). \tag{B.17}$$

Numerical evaluation of complex elementary functions usually uses these separations into real functions of x and y for the real and imaginary parts separately.

B.7 The Residue Theorem

This book makes heavy use of the residue theorem, proofs of which can be found in any of the complex analysis texts cited above. In particular, the residue theorem is used to prove that certain formulæ are true for polynomials of degrees at most n.

A *residue* at $z = a$ of a function $f(z)$ analytic in an annulus surrounding the point $z = a$ is simply the coefficient of ${}^{1}/_{(z-a)}$ in the Laurent expansion of $f(z)$ about $z = a$. The residue theorem states that the integral around a contour surrounding poles of an otherwise analytic function $f(z)$ is simply $2\pi i$ times the sum of the residues.

The Cauchy integral formula, Eq. (B.18) below, represents derivatives as contour integrals (actually, Taylor coefficients as contour integrals). If $f(z)$ is analytic inside a contour C, then

$$\frac{f^{(j)}(z)}{j!} = \frac{1}{2\pi i}\oint_C \frac{f(\zeta)}{(\zeta - z)^{j+1}}\,d\zeta\,. \tag{B.18}$$

Our main application is the use of these theorems on rational functions $f(z) = {}^{p(z)}/_{q(z)}$, where the degree of the denominator $q(z)$ is at least two more than the degree of the numerator $p(z)$, so that $f(z) \sim {}^{A}/_{z^2}$ for large enough z. Then since

$$\oint_C \frac{1}{z^2}\,dz = \int_{\theta=-\pi}^{\pi} \frac{iRe^{i\theta}}{R^2e^{2i\theta}}\,d\theta = O\left(\frac{1}{R}\right)$$

as $R \to \infty$, the integral must, in fact, be 0. Thus, the sum of all the residues of ${}^{p(z)}/_{q(z)}$ must be zero. Once this fact is established, this gives us the partial fraction decomposition of ${}^{p(z)}/_{q(z)}$, which can then be used to develop interpolation formulæ for arbitrary smooth functions $f(z)$. Indeed, these theorems are used repeatedly in the text for formulæ involving barycentric Lagrange and Hermite interpolation, and to derive polynomial formulæ generally.

Problems

B.1. Given the definitions above for i, complex number identity, sum, and product, show
(a) that $\mathbb{C}$ is a field,
(b) that the complex conjugate satisfies

$$\overline{z_1 + z_2 + \cdots + z_n} = \overline{z_1} + \overline{z_2} + \cdots + \overline{z_n} \tag{B.19}$$
$$\overline{z_1 z_2 \cdots z_n} = \overline{z_1}\cdot\overline{z_2}\cdots\overline{z_n}\,, \tag{B.20}$$

(c) that the absolute value satisfies

$$|z_1 z_2 \cdots z_n| = |z_1|\cdot|z_2|\cdots|z_n| \tag{B.21}$$
$$\left|\frac{z_1}{z_2}\right| = \frac{|z_1|}{|z_2|} \tag{B.22}$$
$$|z_1 + z_2 + \cdots + z_n| \le |z_1| + |z_2| + \cdots + |z_n|\,, \tag{B.23}$$

(d) that if $z_1 z_2 = 0$, then either $z_1 = 0$ or $z_2 = 0$.

B.2. Show that $\exp(\ln z) = z$ for all complex z (so *some* high school identities are true).

B.3. Show that there exists complex (in fact, real) z such that $\sqrt{{}^1/_z} \ne {}^1/_{\sqrt{z}}$.

B.4. Show that there exists complex (in fact, real) z, w such that $\sqrt{zw} \neq \sqrt{z} \cdot \sqrt{w}$.

B.5. Show that, with the Maple definitions,

$$\arcsin z = \arctan \frac{z}{\sqrt{1-z^2}} + \pi \mathscr{K}(-\ln(1+z)) - \pi \mathscr{K}(-\ln(1-z)) \,. \tag{B.24}$$

B.6. Fill in the details of the proof that if $p(z)$ is of degree at most n, while the degree of $q(z)$ is at least $n+2$, then the integral of ${}^{p(z)}/_{q(z)}$ around a sufficiently large contour C must be zero.

B.7. Show by direct computation that

$$\frac{1}{2\pi i} \oint_C \frac{f(\zeta)}{(\zeta - z)^{j+1}} \, d\zeta = \frac{f^{(j)}(z)}{j!} \,, \tag{B.25}$$

for $f(z) = 1 + z + z^2$ and $j = 0$, 1, 2, and 3. Use the contours $\zeta = z + R\exp(i\theta)$.

Appendix C
Vectors, Matrices, and Norms

The objective of Part II is to introduce you to the main problems of numerical linear algebra. The study of numerical linear algebra, however, relies on some central concepts of theoretical linear algebra. This appendix will briefly review some of the concepts that together form the ground on which we will build. In particular, you should make sure that the material on vector and matrix norms is familiar.

C.1 Notation and Structure of Matrices

The system of equations

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \tag{C.1}$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \tag{C.2}$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \tag{C.3}$$

can be, by convention, converted into the matrix equation

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} . \tag{C.4}$$

The convention is that

$$\begin{bmatrix} a_{11} & a_{12} & a_{12} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \end{bmatrix} , \tag{C.5}$$

which defines the standard inner product $\langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T \mathbf{b}$, and that $[x] = x$; that is, the $[x]$ matrices are scalars. In this book, we will almost always use bold letters for vectors and matrices: vectors in lowercase, matrices in uppercase. So, we would write

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0,

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix},$$

so that Eq. (C.4) would be simply written $\mathbf{Ax} = \mathbf{b}$. Components of a vector or matrix will usually simply be written with lowercase letters with subscripts as above. On occasion we will use the notation $(\mathbf{A})_{ij}$ for the entry (i, j) of a matrix, or even simply $A(i, j)$, which parallels MATLAB's syntax. In Chap. 13, we will also use superscripts instead of subscripts for vector components; for instance, we will write x^j instead of x_j, in order to take advantage of tensor notation. The context should make it clear.

From these conventions, together with the usual laws of arithmetic over fields, flows all of matrix algebra:

- matrix-vector product;
- matrix-matrix product.

Write $\mathbf{B} = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \mathbf{b}_3 \end{bmatrix}$, where $\mathbf{b}_i$ is the ith column vector of $\mathbf{B}$. Then

$$\mathbf{AB} = \mathbf{A} \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \mathbf{b}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{Ab}_1 & \mathbf{Ab}_2 & \mathbf{Ab}_3 \end{bmatrix}. \tag{C.6}$$

In cases like that, we will often use the notation of Golub and van Loan (1996); it closely follows the MATLAB syntax. Consider a vector $\mathbf{v}$ with 100 components. To denote the vector formed with $v_1, v_2 \ldots, v_{20}$, we simply write $\mathbf{v}(1 : 20)$. So, in general, if $\mathbf{v}$ is an n-vector and $m < n$, we denote the vector formed with $v_1, v_2, \ldots, v_m$ as $\mathbf{v}(1 : m)$. The notation is similar for matrices, except that it has two sets of indices. If $\mathbf{M}$ is 100×100 and we want the 10×10 submatrix formed from the lower-right corner entries of $\mathbf{A}$, we simply write $\mathbf{M}(91 : 100, 91 : 100)$. If we want the first three rows of the matrix, we can write $\mathbf{M}(1 : 3, :)$; the last colon, without numbers, indicates that it ranges over all columns. Similarly, we obtain the first three columns from $\mathbf{M}(:, 1 : 3)$. This notation will be especially useful to characterize operations on partitioned matrices. Note that $\mathbf{M}^H(1 : 3, 1 : 5)$ is *not* the same as $\mathbf{M}(1 : 3, 1 : 5)^H$! The first is a 3×5 matrix, the second a 5×3 matrix; their entries will also not generally be the same.

C.2 Norms

Definition C.1. A *norm* $\|\cdot\|$ is a scalar-valued univariate function satisfying the following three properties:

1. $\|x\| \geq 0$, with $\|x\| = 0$ if and only if $x = 0$;
2. $\|x + y\| \leq \|x\| + \|y\|$;
3. $\|\alpha x\| = |\alpha| \cdot \|x\|$ for scalars α.

We will see many instances of this concept in this book. ◁

In particular, the absolute-value functions $|x|$ for $x \in \mathbb{R}$ and $|z| = \sqrt{a^2 + b^2}$ for $z = a + ib \in \mathbb{C}$ can be shown to be norms. Moreover, except for absolute values, x is typically a vector or a matrix, in which case we respectively call the norms a *vector norm* and a *matrix norm*.

Vector p-norms

Let us begin with vector norms. The most important vector norms are the so-called p-norms:

$$\|\mathbf{x}\|_p = \left(\sum_{k=1}^{n} |x_k|^p\right)^{1/p} = \left(|x_1|^p + |x_2|^p + \cdots + |x_n|^p\right)^{1/p} \tag{C.7}$$

for $1 \le p < \infty$. Three p-norms include almost all applications. In particular, they include the most usual vector norm, the *Euclidean length of a vector*, since if we let $p = 2$, we obtain the familiar

$$\|\mathbf{x}\|_2 = \left(\sum_{k=1}^{n} |x_k|^2\right)^{1/2} = \sqrt{|x_1|^2 + |x_2|^2 + \cdots + |x_n|^2}\,. \tag{C.8}$$

Note that it's important to have the sum of $|x_k|^p$, and not just x_k^p, to cover the case of complex vectors. From the convention above, we also have the following useful relation between the squared 2-norm and the inner products:

$$\|\mathbf{x}\|_2^2 = \langle \mathbf{x}, \mathbf{x} \rangle = \mathbf{x}^H \mathbf{x}. \tag{C.9}$$

Among the other p-norms, the two most common are the 1-norm, defined by

$$\|\mathbf{x}\|_1 = \sum_{k=1}^{n} |x_k| = |x_1| + |x_2| + \ldots + |x_n|\,, \tag{C.10}$$

which is also known as the Manhattan or taxicab norm, and the ∞-norm with the limiting case $p = \infty$, defined by

$$\|\mathbf{x}\|_\infty = \max_{1 \le i \le n} |x_i|\,. \tag{C.11}$$

It is instructive to graph the "unit circles" $\|\mathbf{x}\|_p = 1$ in two dimensions for $p = 1, 2, \infty$, as in Fig. C.1. It is also instructive to check for oneself that the p-norms indeed satisfy Definition C.1. Finally, the following is a very important fact about norms:

Theorem C.1 (Cauchy–Schwarz inequality). *The* 2*-norm satisfies*

$$|\mathbf{x}^H \mathbf{y}| \le \|\mathbf{x}\|_2 \|\mathbf{y}\|_2 \tag{C.12}$$

for any two vectors **x** *and* **y**.

Pairs of p-norms that satisfy $1/p + 1/q = 1$ are said to be *dual* to each other, including the limiting case $(p,q) = (1,\infty)$ or $(p,q) = (\infty,1)$. Dual norms are used in this book when Hölder's inequality is called for.

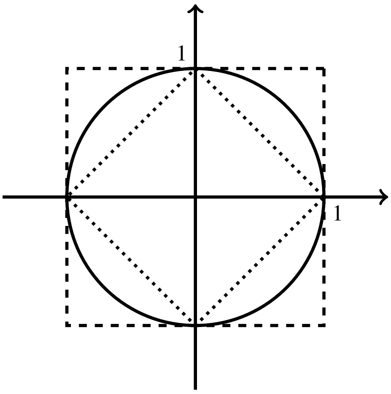


Fig. C.1 Unit circles with the 1-norm (*dotted line*), the 2-norm (*solid line*), and the ∞-norm (*dashed line*)

Theorem C.2 (Hölder's inequality). *Suppose* $\mathbf{x},\mathbf{y} \in \mathbb{C}^n$ *and* $p,q \geq 1$*, so that* $^1/_p + {}^1/_q = 1$ *(including the limiting case where one of* p,q *is 1 and the other is* ∞*, by convention). Then*

$$|\mathbf{x}\cdot\mathbf{y}| \leq \|\mathbf{x}\|_p\|\mathbf{y}\|_q, \tag{C.13}$$

and equality holds only if one of $\mathbf{x}$ *or* $\mathbf{y}$ *is the zero vector or there exist constants* $\lambda \neq 0$ *and* θ *so that* $|x_j|^{1/q} = \lambda|y_j|^{1/p}$ *and* x_j *and* y_j *are zero or* $\arg x_j + \arg y_j = \theta$ *for* $a \leq j \leq n$*.*

In most of the references we have consulted, the proof was left as an exercise or only the real case was proved. Since we use this theorem repeatedly, we provide the proof here.

Proof. The case where $\mathbf{x}$ or $\mathbf{y}$ or both are the zero vector is trivial. Henceforth we assume some components are nonzero. Put $x_j = \rho_j e^{i\varphi_j}$ and $y_j = \sigma_j e^{i\psi_j}$ with $\rho_j, \sigma_j \geq 0$. Then

$$|\mathbf{x}\cdot\mathbf{y}| = \left|\sum_{j=1}^{n} \rho_j\sigma_j e^{i(\varphi_j+\psi_j)}\right| \leq \sum_{j=1}^{n} \rho_j\sigma_j \tag{C.14}$$

by the triangle inequality and using $|e^{i(\varphi_j+\psi_j)}| = 1$. Again by the triangle inequality, equality in the statement is attained only when the $x_j y_j$ are co-directed, that is, $\exists\theta$ such that $\varphi_j + \psi_j = \theta$ for $1 \leq j \leq n$ (Fig. C.2). We have thus reduced consideration to

$$\boldsymbol{\rho}\cdot\boldsymbol{\sigma} = \sum_{j=1}^{n} \rho_j\sigma_j \tag{C.15}$$

with $\rho_j, \sigma_j \geq 0$. The theorem will be established if we prove

$$\boldsymbol{\rho}\cdot\boldsymbol{\sigma} \leq \|\boldsymbol{\rho}\|_p\|\boldsymbol{\sigma}\|_q \tag{C.16}$$

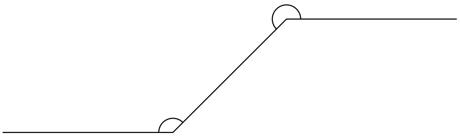


Fig. C.2 The length of the sum of complex numbers is less than the sum of the lengths unless the complex numbers are co-directed

with equality only if $\exists\lambda$ such that $\rho_j^{1/q} = \lambda\sigma_j^{1/p}$. Assume for the moment that each $\rho_j, \sigma_j > 0$ (we will relax this after proving the hard case). We use induction. Also, assume for the moment that $1 < p \leq q < \infty$.

The case $n = 1$ is trivial and of no help. Therefore, we consider the case $n = 2$. We wish to show

$$\rho_1\sigma_1 + \rho_2\sigma_2 \leq (\rho_1^p + \rho_2^p)^{1/p}(\sigma_1^q + \sigma_2^q)^{1/q}. \tag{C.17}$$

We consider two cases, $0 < \rho_2 \leq \rho_1$ with $0 < \sigma_2 \leq \sigma_1$ and $0 < \rho_1 \leq \rho_2$ with $0 < \sigma_2 \leq \sigma_1$, which by symmetry cover all cases. Suppose first $0 < \rho_2 \leq \rho_1$ with $0 < \sigma_2 \leq \sigma_1$. Put $\xi = \rho_2/\rho_1$ and $\eta = \sigma_2/\sigma_1$, so $0 < \xi,\ \eta \leq 1$. Then we wish to prove

$$\rho_1\sigma_1(1 + \xi\eta) \leq \rho_1\sigma_1(1 + \xi^p)^{1/p}(1 + \eta^q)^{1/q} \tag{C.18}$$

or

$$F(\xi, \eta) := (1 + \xi^p)^{1/p}(1 + \eta^q)^{1/q} - (1 + \xi\eta) \geq 0. \tag{C.19}$$

Fix ξ in $0 < \xi \leq 1$, and consider $F(\xi, \eta)$ as a function of one variable, η. Then at a local minimum we must have $F_\eta = 0$, where

$$F_\eta = (1 + \xi^p)^{1/p}\eta^{q-1}(1 + \eta^q)^{1/q-1} - \xi. \tag{C.20}$$

It is helpful at this point to recognize that $1/p + 1/q = 1$ can be rewritten as $1/p = 1 - 1/q$ or $(q-1)/q$ so $p = q/(q-1)$. We also find

$$F_{\eta\eta} = q(q-1)\left(\frac{\eta^q}{1+\eta^q}\right)^{q-2} \frac{\eta^{q-1}}{(1+\eta^q)^2}(1 + \xi^p)^{1/p} > 0 \tag{C.21}$$

in $0 < \eta < 1$, so in the interior any place with $F_\eta = 0$ is indeed a minimum. Solving $F_\eta = 0$, we have

$$\frac{\eta^{q-1}}{(1+\eta^q)^{(q-1)}q} = \frac{\xi}{(1+\xi^p)^{1/p}} = \left(\frac{\xi^p}{1+\xi^p}\right)^{1/p} = \left(\frac{\xi^p}{1+\xi^p}\right)^{(q-1)/q}. \tag{C.22}$$

Since all quantities are nonnegative,

$$\frac{\eta^q}{1+\eta^q} = \frac{\xi^p}{1+\xi^p} \qquad \text{or} \qquad \eta^q = \xi^p.$$

So

$$\left(\frac{\rho_2}{\rho_1}\right)^p = \left(\frac{\sigma_2}{\sigma_1}\right)^q \quad \text{or} \quad \left(\frac{\rho_2}{\rho_1}\right)^{1/q} = \left(\frac{\sigma_2}{\sigma_1}\right)^{1/p} \quad \text{or} \quad \frac{\rho_2^{1/q}}{\sigma_2^{1/p}} = \frac{\rho_1^{1/q}}{\sigma_1^{1/p}} =: \lambda\,.$$

Equivalently, $\exists\lambda$ such that

$$\rho_2^{1/q} = \lambda\sigma_2^{1/p} \qquad \text{and} \qquad \rho_1^{1/q} = \lambda\sigma_1^{1/p}\,.$$

At all other η the function $F(\xi,\eta)$ is strictly larger than this. By inspection, $F(\xi,\eta) = 0$ at this minimum [one easy way to see this is to put $\rho_1^p = \lambda^{pq}\sigma_1^q$ and $\rho_2^p = \lambda^{pq}\sigma_2^q$ in $(\rho_1^p + \rho_2^p)^{1/p}$ to get $(\lambda^{pq})^{1/p}(\sigma_1^q + \sigma_2^q)^{1/p}$. So

$$(\rho_1^p + \rho_2^p)^{1/p}(\sigma_1^q + \sigma_2^q)^{1/q} = \lambda^q(\sigma_1^q + \sigma_2^q)^{1/p+1/q} \tag{C.23}$$

$$= \lambda^q(\sigma_1^q + \sigma_2^q) \tag{C.24}$$

and

$$\rho_1\sigma_1 + \rho_2\sigma_2 = \lambda^q\sigma_1^{q/p}\sigma_1 + \lambda^q\sigma_2^{q/p}\sigma_2 \tag{C.25}$$

$$= \lambda^q(\sigma_1^q + \sigma_2^q), \tag{C.26}$$

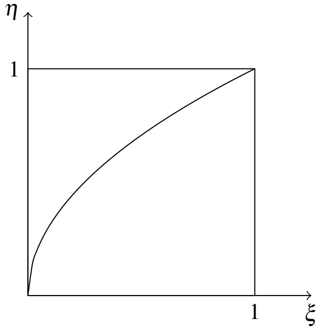


Fig. C.3 The curve $\eta^q = \xi^p$

showing equality is attained in this case, directly]. Examining the edge cases $\xi = 0$, $\xi = 1$, $\eta = 0$, and $\eta = 1$ lead to the same conclusion.

The other case where $0 < \rho_2 \leq \rho_1$ but $0 < \sigma_1 \leq \sigma_2$ gives rise to a different function, as follows, but a similar conclusion. Put $\xi = \rho_2/\rho_1 \leq 1$ as before but now $\eta = \sigma_1/\sigma_2 \leq 1$. Then

$$\rho_1\sigma_1 + \rho_2\sigma_2 = \rho_1\sigma_2(\eta + \xi) \tag{C.27}$$

and

$$(\rho_1^p + \rho_2^p)^{1/p}(\sigma_1^q + \sigma_2^q)^{1/q} = \rho_1\sigma_2(1+\xi^p)^{1/p}(\eta^q+1)^{1/q}\,, \tag{C.28}$$

so the function to minimize is

$$G(\xi,\eta) = (1+\xi^p)^{1/p}(1+\eta^q)^{1/q} - (\xi+\eta)\,. \tag{C.29}$$

As before, $G_{\eta\eta} > 0$ on the interior $0 < \xi,\eta < 1$, but now

$$G_\eta = (1+\xi^p)^{1/p}\eta^{q-1}(1+\eta^q)^{1/q-1} - 1, \tag{C.30}$$

which is zero iff

$$\left(\frac{\eta^q}{1+\eta^q}\right)^{(q-1)/q} = \frac{1}{(1+\xi^p)^{1/p}} \tag{C.31}$$

or

$$\frac{\eta^q}{1+\eta^q} = \frac{1}{1+\xi^p} \qquad \text{so} \qquad \xi^p = \eta^{-q}\,. \tag{C.32}$$

However, unlike before, $\eta = \sigma_1/\sigma_2$, so this again gives

$$\frac{\rho_2^{1/q}}{\rho_1^{1/q}} = \frac{\sigma_2^{1/p}}{\sigma_1^{1/p}} \qquad \text{or} \qquad \rho_j^{1/q} = \lambda\sigma_j^{1/p} \tag{C.33}$$

for $j = 1,2$, for some constant λ. Again, we check the edge cases $\xi = 0$, $\xi = 1$, $\eta = 0$, and $\eta = 1$ with similar conclusions. This concludes the proof for $n = 2$ and $1 < p \le q$. By symmetry, it is also true for $1 < q \le p$.

We now make the inductive assumption that the theorem is true for $n = N$ and consider the vectors $\hat{\boldsymbol{\rho}} = [\rho_1,\rho_2,\dots,\rho_N,\rho_{N+1}]$ and $\hat{\boldsymbol{\sigma}} = [\sigma_1,\sigma_2,\dots,\sigma_N,\sigma_{N+1}]$. Notice that

$$\hat{\boldsymbol{\rho}}\cdot\hat{\boldsymbol{\sigma}} = \sum_{j=1}^{N}\rho_j\sigma_j + \rho_{N+1}\sigma_{N+1} \le \|\boldsymbol{\rho}\|_p\|\boldsymbol{\sigma}\|_q + \rho_{N+1}\sigma_{N+1}\,, \tag{C.34}$$

where the N-vectors $\boldsymbol{\rho}$ and $\boldsymbol{\sigma}$ have the obvious meaning. Now equality is obtained here iff $\rho_j^{1/q} = \lambda\sigma_j^{1/p}$ for some constant λ, for $1 \le j \le N$, but not $N+1$.

Consider now the 2-vectors $[\|\boldsymbol{\rho}\|_p, \rho_{N+1}]$ and $[\|\boldsymbol{\sigma}\|_q, \sigma_{N+1}]$. Their dot product appears on the right-hand side of Eq. (C.34). By the $n = 2$ case of Hölder's inequality, this satisfies

$$\|\boldsymbol{\rho}\|_p\|\boldsymbol{\sigma}\|_q + \rho_{N+1}\sigma_{N+1} \le \left(\|\boldsymbol{\rho}\|_p^p + \rho_{N+1}^p\right)^{1/p}\left(\|\boldsymbol{\sigma}\|_q^q + \sigma_{N+1}^q\right)^{1/q}\,. \tag{C.35}$$

Notice that $\|\boldsymbol{\rho}\|_p^p = \sum_{j=1}^N \rho_j^p$ and $\|\boldsymbol{\sigma}\|_q^q = \sum_{j=1}^N \sigma_j^q$. This proves $\hat{\boldsymbol{\rho}}\cdot\hat{\boldsymbol{\sigma}} \le \|\hat{\boldsymbol{\rho}}\|_p\|\hat{\boldsymbol{\sigma}}\|_q$, which is the first half of what is to be proved.

Considering the case of equality, this is attained in Eq. (C.35) iff $\exists\hat{\lambda}$ with $\|\boldsymbol{\rho}\|_p^{1/q} = \hat{\lambda}\|\boldsymbol{\sigma}\|_q^{1/p}$ and $\rho_{N+1}^{1/q} = \hat{\lambda}\sigma_{N+1}^{1/p}$, or

$$\|\boldsymbol{\rho}\|_p^p = \hat{\lambda}^{pq}\|\boldsymbol{\sigma}\|_q^q \quad \text{and} \quad \rho_{N+1}^p = \hat{\lambda}^{pq}\sigma_{N+1}^q\,. \tag{C.36}$$

Equality in (C.34) happens simultaneously iff $\exists\lambda$ such that $\rho_j^p = \lambda^{pq}\sigma_j^q$ for $1 \le j \le N$, in which case

$$\|\boldsymbol{\rho}\|_p^p = \sum_{j=1}^{N}\rho_j^p = \sum_{j=1}^{N}\lambda^{pq}\sigma_j^q = \lambda^{pq}\|\boldsymbol{\sigma}\|_q^q, \tag{C.37}$$

which implies $\lambda = \hat{\lambda}$ and the induction is complete for the case $1 < p \le q$, and by symmetry for $1 < q \le p$. All that remains are the limiting cases when $p \to 1$ and $q \to \infty$ and vice versa, and the case when some ρ_j or σ_j are zero. We consider $p \to 1$ first.

Notice that the conditions for equality make sense in the limit $p \to 1$: $\rho_j^{1/q} = \lambda\sigma_j^{1/p} \to 1 = \lambda\sigma_j$ or all σ_j tend to the same constant. Direct use of the inequality $\sigma_j \le \|\boldsymbol{\sigma}\|_\infty$ gives

$$\sum_{j=1}^{n}\rho_j\sigma_j \le \sum_{j=1}^{n}\rho_j\|\boldsymbol{\sigma}\|_\infty = \left(\sum_{j=1}^{n}\rho_j\right)\|\boldsymbol{\sigma}\|_\infty = \|\boldsymbol{\rho}\|_1\|\boldsymbol{\sigma}\|_\infty, \tag{C.38}$$

and equality cannot occur if any $\sigma_j < \|\boldsymbol{\sigma}\|_\infty$.

All that remains is the case when some ρ_j or σ_j is zero. But this is simple: A zero component removes the corresponding σ_j or ρ_j from the dot product, allowing the use of the Hölder inequality on a shorter (but nonempty by hypothesis) vector; this gives $\boldsymbol{\rho}\cdot\boldsymbol{\sigma} \le \|\boldsymbol{\rho}\|_p\|\boldsymbol{\sigma}\|_q$ with the short vectors. Including the σ_j (or ρ_j) on the right-hand side strictly increases the right-hand side unless the σ_j (or ρ_j) is also zero, which, if zero, trivially satisfies $\rho^{1/q} = \lambda\sigma_j^{1/p}$. ♮

All p-norms are equivalent in the following sense: There exist constants $c(p,q,n)$ and $C(p,q,n)$ such that

$$c\|\mathbf{x}\|_p \le \|\mathbf{x}\|_q \le C\|\mathbf{x}\|_p \tag{C.39}$$

holds for any vector $\mathbf{x}$. Thus, a vector that is "large" in one norm cannot be "too small" in another, for instance.

Weighted Norms

Weighted norms are of great interest in practice. For a set of weights $w_k \ge 0$ that are not all zeros, a weighted vector p-norm is defined by

$$\|\mathbf{x}\|_{w,p} = \left(\sum_{k=1}^{n}|w_k x_k|^p\right)^{1/p}, \tag{C.40}$$

and the ∞ case is $\|\mathbf{x}\|_{w,\infty} = \max_k w_k |x_k|$. In this definition, it is important that the weights w_k appear *inside* the pth power, for otherwise the limit as $p \to \infty$ does not give the right result. The dual norm is then

$$\|\mathbf{x}\|_{w^*,q} = \left(\sum_{k=1}^{n} |w_k^* x_k|^q \right)^{1/q}, \tag{C.41}$$

where ${}^1\!/_p + {}^1\!/_q = 1$ and $w^* = 0$ if $w_k = 0$ and $x_k = 0$, and $w_k^* = {}^1\!/_{w_k}$ otherwise. However, if $x_k \neq 0$, where $w_k = 0$, then $\|\mathbf{x}\|_{w^*,q} = \infty$. Then

$$|\mathbf{x} \cdot \mathbf{y}| \leq \|\mathbf{x}\|_{w,p} \|\mathbf{y}\|_{w^*,q} \tag{C.42}$$

is the generalized Hölder inequality. Proofs of the extended results can be found, for example, in Rezvani Dehaghani (2005).

Matrix Norms

Two types of matrix norms will be used in the book. From the vector p-norms, we can show that the following function is a norm:

$$\|\mathbf{A}\|_p = \max_{\mathbf{x} \neq 0} \frac{\|\mathbf{A}\mathbf{x}\|_p}{\|\mathbf{x}\|_p}. \tag{C.43}$$

These norms are known as the *induced p-norms*. One can show that this implies the following convenient equation:

$$\|\mathbf{A}\|_p = \max_{\|\mathbf{x}\|_p = 1} \|\mathbf{A}\mathbf{x}\|_p. \tag{C.44}$$

So, the norm of $\mathbf{A}$ is the length of the longest vector obtained by transforming a unit vector with a multiplication by $\mathbf{A}$. The cases $p = 1$ and $p = \infty$ have the virtue of being easy to compute:

$$\|\mathbf{A}\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{n} |a_{ij}| \tag{C.45}$$

$$\|\mathbf{A}\|_\infty = \max_{1 \leq j \leq n} \sum_{j=1}^{n} |a_{ij}|. \tag{C.46}$$

The former is the maximum column sum, and the latter is the maximum row sum. The computation of $\|\mathbf{A}\|_2$ is the computation of the largest singular value σ_1 of $\mathbf{A}$ (see Chap. 4).

Two other matrix norms will be of interest to us. First, the Frobenius norm will be used in componentwise analysis. It is defined as follows, for an $m \times n$ matrix:

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=i}^{n} |a_{ij}|^2}. \tag{C.47}$$

Second, we will occasionally use the spectral ”norm,” defined by

$$\rho(\mathbf{A}) = \max_{\lambda \in \boldsymbol{\Lambda}(\mathbf{A})} |\lambda| \,, \tag{C.48}$$

where $\boldsymbol{\Lambda}(\mathbf{A})$ is the spectrum of $\mathbf{A}$. This last function is not truly a norm because, for example, it is possible to have $\rho(\mathbf{A}) = 0$ when $\mathbf{A} \neq \mathbf{0}$; consider, for instance, the matrix

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} .$$

Two good references to learn more about norms are Golub and van Loan (1996) and Steele (2004).

Function Norms

Function norms serve a similar purpose. The tradition is to define the p-norm of a function f by

$$\|f(x)\|_p = \left(\int_A |f(x)|^p dx \right)^{1/p} \,, \tag{C.49}$$

where $f(x)$ is such that this makes sense, and similarly to take

$$\|f(x)\|_\infty = \sup_{x \in A} |f(A)| \,. \tag{C.50}$$

For complex functions, the definition is analogous. Note that we can only claim that

$$\|f(z)\|_p \quad \leftrightarrow \quad f(z) = 0$$

almost everywhere, that is, except on a set of measure zero. But for numerical purposes, we consider only smooth functions anyway—usually analytic in this book, in fact—and so this restriction does not bother us much. Again, this satisfies Definition C.1. These norms can be extended in a straightforward fashion to vector- or matrix-valued functions.

Weighted function norms occur especially often in the context of orthogonal polynomials, although the main use is of the inner product

$$\langle f, g \rangle = \int_a^b w(x) f^*(x) g(x) dx \tag{C.51}$$

itself, not the norm $f = \langle f, f \rangle$. Still, approximation by minimizing the weighted norm of the error is a fruitful source of numerical methods.

The principal use of these concepts in numerical analysis is in proofs of convergence. This book relies more on a posteriori analysis than on predictive proofs of convergence; instead, one does the computation and measures the norm of the residual afterward. Computation of the condition number (sensitivity) follows, and thus in the particular case being considered analysis is complete, even if no general proof of convergence has been presented (or is even known).

C.3 Derivation of the Normal Equations

If $\mathbf{A}$ is $m \times n$ with $m \geq n$, we may wish to find the vector $\mathbf{x}$ minimizing the residual $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$. To this effect, consider

$$\mathbf{r} + \Delta\mathbf{r} = \mathbf{b} - \mathbf{A}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b} - \mathbf{A}\mathbf{x} - \mathbf{A}\Delta\mathbf{x}\,. \tag{C.52}$$

Then we find that

$$\begin{aligned} \|\mathbf{r} + \Delta\mathbf{r}\|_2^2 &= (\mathbf{r} + \Delta\mathbf{r})^H(\mathbf{r} + \Delta\mathbf{r}) & \text{(C.53)} \\ &= \mathbf{r}^H\mathbf{r} + \mathbf{r}^H\Delta\mathbf{r} + \Delta\mathbf{r}^H\mathbf{r} + (\Delta\mathbf{r})^H\Delta\mathbf{r} & \text{(C.54)} \\ &= \mathbf{r}^H\mathbf{r} + 2\Delta\mathbf{r}^H\mathbf{r} + (\Delta\mathbf{r})^H\Delta\mathbf{r} & \text{(C.55)} \end{aligned}$$

because each of $\mathbf{r}^H\Delta\mathbf{r}$ and $\Delta\mathbf{r}^H\mathbf{r}$ is a scalar and real (real^H=real). If we can find $\mathbf{x}$ such that $\Delta\mathbf{r}^H\mathbf{r} = 0$ for all $\Delta\mathbf{x}_i$, then we will have found the minimum, because in that case

$$\begin{aligned} \|\mathbf{r} + \Delta\mathbf{r}\|_2^2 &= \|\mathbf{r}\|^2 + \|\Delta\mathbf{r}\|^2 & \text{(C.56)} \\ &\geq \|\mathbf{r}\|^2\,, & \text{(C.57)} \end{aligned}$$

with equality only if $\Delta\mathbf{r} = \mathbf{A}\Delta\mathbf{x} = \mathbf{0}$. The equations $\Delta\mathbf{r}^H\mathbf{r} = \mathbf{0}$ or $\Delta\mathbf{x}^H\mathbf{A}^H(\mathbf{b} - \mathbf{A}\mathbf{x})$ must hold for all $\mathbf{A}^H$. If we can find a vector $\mathbf{x}$ such that

$$0 = \mathbf{A}^H(\mathbf{b} - \mathbf{A}\mathbf{x})\,, \tag{C.58}$$

then we will have attained our purpose. These are the so-called normal equations for the least-squares solutions:

$$\mathbf{A}^H\mathbf{A}\mathbf{x} = \mathbf{A}^H\mathbf{b}. \tag{C.59}$$

Note that this is $n \times n$, and $\mathbf{A}^H\mathbf{A}$ might be singular if $\mathbf{A}$ does not have full column rank.

C.4 The Schur Complement

We will use the Schur complement on multiple occasions in this book. Zhang (2005) edited a beautiful book providing an introduction and multiple applications of the Schur complement. Here, we only mention the very basic idea of it. If $\mathbf{D}$ is invertible, then the *Schur determinantal formula* for a block matrix is

$$\det \left[\begin{array}{c|c} \mathbf{A} & \mathbf{B} \\ \hline \mathbf{C} & \mathbf{D} \end{array}\right] = \det(\mathbf{D})\det(\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})\,, \tag{C.60}$$

where $\mathbf{A}-\mathbf{BD}^{-1}\mathbf{C}$ is the Schur complement. One can see that this relation holds by observing that

$$\begin{bmatrix}\mathbf{A} & \mathbf{B}\\ \mathbf{C} & \mathbf{D}\end{bmatrix} = \begin{bmatrix}\mathbf{A}-\mathbf{BD}^{-1}\mathbf{C} & \mathbf{BD}^{-1}\\ \mathbf{0} & \mathbf{I}\end{bmatrix}\begin{bmatrix}\mathbf{I} & \mathbf{0}\\ \mathbf{C} & \mathbf{D}\end{bmatrix}, \tag{C.61}$$

which is a block factoring, and use the formula for the determinant of a product of matrices.

C.5 Eigenvalues

Here are some important facts about eigenvalues and eigenvectors:

1. If $\mathbf{A}$ is diagonal, its eigenvalues are the diagonal entries; that is, $\lambda_i = a_{ii}$, with eigenvectors $\mathbf{e}_i$ and (left) $\mathbf{e}_i^T = \mathbf{e}_i^H$.
2. If $\mathbf{A} = \mathbf{T}$ is triangular (say upper-triangular), then again its eigenvalues appear on the diagonal: This time, though, the eigenvectors are not always so easy. Take a 3 × 3 example:

$$\begin{bmatrix}t_{11} & t_{12} & t_{13}\\ & t_{22} & t_{23}\\ & & t_{33}\end{bmatrix}\begin{bmatrix}1\\0\\0\end{bmatrix} = t_{11}\begin{bmatrix}1\\0\\0\end{bmatrix}. \tag{C.62}$$

Clearly, t_{11} is an eigenvalue and $\mathbf{e}_1$ is its eigenvector. However, for t_{22}, we obtain

$$\begin{bmatrix}t_{11} & t_{12} & t_{13}\\ & t_{22} & t_{23}\\ & & t_{33}\end{bmatrix}\begin{bmatrix}a\\1\\0\end{bmatrix} = \begin{bmatrix}at_{11}+t_{12}\\ t_{22}\\ 0\end{bmatrix}, \tag{C.63}$$

which is $t_{22}[a,1,0]^T$ only if $at_{11}+t_{12} = at_{22}$ or, if $t_{11} \neq t_{22}$, $a = {}^{t_{12}}\!/\!_{(t_{22}-t_{11})}$. If $t_{11} = t_{22}$ but $t_{12} \neq 0$, there is no other eigenvector different to $\mathbf{e}_1$ anyway. Similarly, you can identify the third eigenvector:

$$\begin{bmatrix}t_{11} & t_{12} & t_{13}\\ & t_{22} & t_{23}\\ & & t_{33}\end{bmatrix}\begin{bmatrix}a\\b\\1\end{bmatrix} = t_{23}\begin{bmatrix}a\\b\\1\end{bmatrix} \tag{C.64}$$

only if $bt_{22}+t_{23} = b_2t_{33}$ or $b = {}^{t_{23}}\!/\!_{(t_{33}-t_{22})}$ and $at_{11}+bt_{12}+t_{13} = at_{33}$ or $a = ((t_{12}t_{23})/(t_{33}-t_{22}))/(t_{33}-t_{11})$. So, we see that the diagonal elements t_{11}, t_{22} and t_{33} are indeed eigenvalues, with computable eigenvectors if the eigenvalues are distinct.

Lemma C.1. *If* $\mathbf{T}$ *is upper-triangular and normal, then* $\mathbf{T}$ *is diagonal.*

Proof. By induction. If $n = 1$, there is nothing to prove. Suppose all upper-triangular normal matrices of dimension $(n-1)\times(n-1)$ are diagonal. Consider

$$\mathbf{T} = \begin{bmatrix} t_{11} \; t_{12} \cdots \; t_{1n} \\ \mathbf{T}_1 \end{bmatrix} = \left[\begin{array}{c|c} t_{11} & \mathbf{p} \\ \hline 0 & \mathbf{T}_1 \end{array}\right], \tag{C.65}$$

partitioning **T**. Then $\mathbf{T}\mathbf{T}^H = \mathbf{T}^H\mathbf{T}$ means

$$\begin{bmatrix} t_{11} & \mathbf{p} \\ & \mathbf{T}_1 \end{bmatrix} \begin{bmatrix} \bar{t}_{11} & \\ \mathbf{p}^H & \mathbf{T}_1^H \end{bmatrix} = \begin{bmatrix} |t_{11}|^2 + \sum_{k=2}^n |t_{1k}|^2 & p\mathbf{T}_1^H \\ \mathbf{T}_1 p^H & \mathbf{T}_1\mathbf{T}_1^H \end{bmatrix}, \tag{C.66}$$

where $p = [t_{12}, t_{13}, \ldots, t_{1n}]$, and this must equal

$$\begin{bmatrix} \mathbf{t}_{11} & \\ \mathbf{p}^H & \mathbf{T}_1^H \end{bmatrix} \begin{bmatrix} t_{11} & \mathbf{p} \\ & \mathbf{T}_1 \end{bmatrix} = \begin{bmatrix} |t_{11}|^2 & \bar{t}_{11}\mathbf{p} \\ t_{11}\mathbf{p}^H & \mathbf{T}_1^H\mathbf{T}_1 + \mathbf{p}^H\mathbf{p} \end{bmatrix} \tag{C.67}$$

$$|t_{11}|^2 = |t_{11}|^2 + \sum_{k=2}^n |t_{1k}|^2 . \tag{C.68}$$

Hence, $\sum_{k=2}^n |t_{1k}|^2 = 0$, and thus all $t_{ik} = 0$ for $k \geq 2$ and $\mathbf{T}_1^H\mathbf{T}_1 = \mathbf{T}_1\mathbf{T}_1^H$, so $\mathbf{T}_1$ is itself upper-triangular, normal, and dimension $(n-1) \times (n-1)$. Hence, **T** is diagonal. □

Theorem C.3. *If* $\mathbf{A}\mathbf{A}^H = \mathbf{A}^H\mathbf{A}$, *that is, if* **A** *is normal, then it is unitarily similar to a diagonal matrix,* $\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{U}^H$ *with* **T** *diagonal.*

Proof. We have $\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{U}^H$ and $\mathbf{A}^H = \mathbf{U}\mathbf{T}^H\mathbf{U}^H$. So $\mathbf{A}\mathbf{A}^H = \mathbf{U}\mathbf{T}\mathbf{U}^H\mathbf{U}\mathbf{T}^H\mathbf{U}^H = \mathbf{U}\mathbf{T}\mathbf{T}^H\mathbf{U}^H$ and $\mathbf{A}^H\mathbf{A} = \mathbf{U}\mathbf{T}^H\mathbf{U}^H\mathbf{U}\mathbf{T}\mathbf{U}^H = \mathbf{U}\mathbf{T}^H\mathbf{T}\mathbf{U}^H$. Hence, $\mathbf{A}\mathbf{A}^H = \mathbf{A}^H\mathbf{A}$ implies $\mathbf{T}\mathbf{T}^H = \mathbf{T}^H\mathbf{T}$. So **T** is normal. By Lemma C.1, **T** is diagonal. □

References

Ablowitz, M. J., & Clarkson, P. A. (1991). *Solitons, nonlinear evolution equations and inverse scattering*. Cambridge: Cambridge University Press.

Abramowitz, M., & Stegun, I., Eds. (1972). *Hanbbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. 10th printing. National Bureau of Standards. Applied Mathematics Series, vol 55.

Alefeld, G. (1981). On the convergence of Halley's method. *American Mathematical Monthly, 88*(7), 530–536.

Amiraslani, A. (2004). Dividing polynomials when you only know their values. In: *Proceedings EACA, Santander*, pp. 5–10.

Amiraslani, A. (2006). *Algorithms for Matrices, Polynomials, and Matrix Polynomials*. PhD thesis, University of Western Ontario, London.

Amiraslani, A., Corless, R. M., & Lancaster, P. (2009). Linearization of matrix polynomials expressed in polynomial bases. *IMA Journal of Numerical Analysis, 29*(1), 141–157.

Anderson, E., Bai, Z., & Bischof, C. (1999). *LAPACK users' guide* (3rd ed.). Philadelphia: SIAM.

Anderssen, R. S., & Hegland, M. (1999). For numerical differentiation, dimensionality can be a blessing! *Mathematics of Computation, 68*, 1121–1141.

Anderssen, R. S., & Hegland, M. (2010). Derivative spectroscopy—an enhanced role for numerical differentiation. *Journal of Integral Equations and Applications, 22*(3), 355–367.

Andrews, G. E., Askey, R., & Roy, R. (1999). *Special functions*. Cambridge: Cambridge University Press.

Arena, P., & Fortuna, L. (2002). Analog cellular locomotion control of hexapod robots. *Control Systems Magazine, IEEE, 22*(6), 21–36.

Arenstorf, R. F. (1963). Periodic solutions of the restricted three body problem representing analytic continuations of Keplerian elliptic motions. *American Journal of Mathematics, 85*(1), 27–35.

Arnol'd, V. I. (1971). On matrices depending on parameters. *Russian Mathematical Surveys, 26*(2), 29–43.

Aruliah, D. A., & Corless, R. M. (2004). Numerical parameterization of affine varieties using ODEs. In: *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pp. 12–18. New York: ACM.

Aruliah, D. A., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., Haddock, S. H. D., Huff, K., Mitchell, I., Plumbley, M., Waugh, B., White, E. P., Wilson, G., & Wilson, P. (2012). Best practices for scientific computing. *CoRR*, abs/1210.0530.

Ascher, U. (2008). *Numerical methods for evolutionary equations*. Computational Science and Engineering. Philadelphia: SIAM.

Ascher, U. M., Mattheij, R. M. M., & Russell, R. D. (1988). *Numerical solution of boundary problems for ordinary differential equations*. Upper Saddle River, NJ: Prentice-Hall.

Ascher, U., Pruess, S., & Russell, R. D. (1983). On spline basis selection for solving differential equations. *SIAM Journal on Numerical Analysis, 20*, 121–142.

Babuška, I., & Gatica, G. (2010). A residual-based a posteriori error estimator for the Stokes–Darcy coupled problem. *SIAM Journal on Numerical Analysis, 48*, 498–523.

Babuška, I., & Rheinboldt, W. (1978). A-posteriori error estimates for the finite element method. *International Journal for Numerical Methods in Engineering, 12*(10), 1597–1615.

Bailey, D. H., Jeyabalan, K., & Li, X. S. (2005). A comparison of three high-precision quadrature schemes. *Experimental Mathematics, 14*(3), 317–329.

Baker, C. T. H., Agyingi, E. O., Parmuzin, E. I., Rihan, F. A., & Song, Y. (2006). Sense from sensitivity and variation of parameters. *Applied Numerical Mathematics, 56*, 397–412.

Baker, C., Bocharov, G., Paul, C., & Rihan, F. (2005). Computational modelling with functional differential equations: Identification, selection, and sensitivity. *Applied Numerical Mathematics, 53*(2-4), 107–129.

Balachandran, B., Kalmár-Nagy, T., & Gilsinn, D. (2009). *Delay differential equations: recent advances and new directions*. New York: Springer.

Barbeau, E. J. (2003). *Polynomials*. New York: Springer.

Barton, D. (1980). On Taylor series and stiff equations. *ACM Transactions on Mathematical Software (TOMS), 6*(3), 280–294.

Barton, D., Willers, I., & Zahar, R. (1971). The automatic solution of systems of ordinary differential equations by the method of Taylor series. *The Computer Journal, 14*(3), 243.

Batterman, R. W. (1993). Defining chaos. *Philosophy of Science, 60*(1), 43–66.

Batterman, R. W. (2002). *The devil in the details: asymptotic reasoning in explanation, reduction, and emergence*. Oxford: Oxford University Press.

Battles, Z., & Trefethen, L. (2004). An extension of Matlab to continuous functions and operators. *SIAM Journal on Scientific Computing, 25*(5), 1743–1770.

Bearman, P. W., & Harvey, J. K. (1976). Golf ball aerodynamics. *Aeronautics Quarterly, 27*, 112–122.

Becker, R., & Rannacher, R. (2001). An optimal control approach to a posteriori error estimation in finite element methods. *Acta Numerica, 10*, 1–102.

Beckermann, B. (2000). The condition number of real Vandermonde, Krylov and positive definite Hankel matrices. *Numerische Mathematik, 85*(4), 553–577.

Bellen, A., & Zennaro, M. (2003). *Numerical methods for delay differential equations*. Oxford: Oxford University Press.

Bellman, R., & Cooke, K. (1963). *Differential-difference equations*. Boston: Academic Press.

Bellman, R., Kashef, B., & Casti, J. (1972). Differential quadrature: a technique for the rapid solution of nonlinear partial differential equations. *Journal of Computational Physics, 10*(1), 40–52.

Bender, C. M., & Orszag, S. A. (1978). *Advanced mathematical methods for scientists and engineers*. New York: McGraw-Hill.

Berntsen, J., Espelid, T. O., & Genz, A. (1991). An adaptive algorithm for the approximate calculation of multiple integrals. *ACM Transactions on Mathematical Software, 17*, 437–451.

Berrut, J., & Mittelmann, H. (2000). Rational interpolation through the optimal attachment of poles to the interpolating polynomial. *Numerical Algorithms, 23*(4), 315–328.

Berrut, J., & Trefethen, L. N. (2004). Barycentric Lagrange interpolation. *SIAM Review, 46*(3), 501–517.

Berrut, J., Baltensperger, R., & Mittelmann, H. (2005). In: D.H. Mache, J. Szabados, M.G. de Bruin (Eds.), Recent developments in barycentric rational interpolation. *Trends and applications in constructive approximation*, ISNM International Series of Numerical Mathematics, Springer, *151*, 27–51.

Bertoluzza, S., Falletta, S., Russo, G., & Shu, C.-W. (2009). *Numerical solutions of partial differential equations*. Basel: Birkhäuser.

Bini, D., & Boito, P. (2010). In: D.A. Bini, V. Mehrmann, V. Olshevsky, E.E. Tyrtyshnikov, M. van Barel (Eds.), A fast algorithm for approximate polynomial GCD based on structured matrix computations. *Numerical methods for structured matrices and applications*, Operator Theory: Advances and Applications, Springer, *199*, 155–173.

Bini, D., & Mourrain, B. (1996). Polynomial test suite, 1996. Technical report, INRIA. See www-sop.inria.fr/saga/POL.

Bini, D., & Pan, V. Y. (1994). *Polynomial and matrix computations: fundamental algorithms*, vol. 1. Basel: Birkhäuser Verlag.

Birkhoff, G., & Rota, G.-C. (1989). *Ordinary differential equations* (4th ed.). New York: Wiley.

Birkisson, A., & Driscoll, T. A. (2012). Automatic Fréchet differentiation for the numerical solution of boundary-value problems. *ACM Transactions on Mathematical Software (TOMS), 38*(4), 26.

Blum, L., Cucker, F., Shub, M., & Smale, S. (1998). *Complexity and real computation*. New York: Springer.

Bocharov, G. A., & Rihan, F. A. (2000). Numerical modelling in biosciences using delay differential equations. *Journal of Computational and Applied Mathematics, 125*(1-2), 183–199.

Boole, G. (1880). *A treatise on the calculus of finite differences* (1951 ed. edited by J. F. Moulton). Macmillan and Company, 3rd Edition, London.

Bornemann, F. A. (1992). An adaptive multilevel approach to parabolic equations III. 2D error estimation and multilevel preconditioning. *IMPACT of Computing in Science and Engineering, 4*(1), 1–45.

Borwein, J., & Borwein, P. (1984). The arithmetic-geometric mean and fast computation of elementary funtions. *SIAM Review, 26*(3), 351–366.

Bostan, A., Lecerf, G., & Schost, É. (2003). Tellegen's principle into practice. In: *Proceedings ISSAC*, pp. 37–44. New York: ACM.

Böttcher, A., & Grudsky, S. M. (1987). *Spectral properties of banded Toeplitz matrices*. Philadelphia: SIAM.

Boyd, J. P. (2002). Computing zeros on a real interval through Chebyshev expansion and polynomial rootfinding. *SIAM Journal on Numerical Analysis, 40*(5), 1666–1682.

Boyd, J. P. (2013). Finding the zeros of a univariate equation: proxy rootfinders, Chebyshev interpolation, and the companion matrix. *SIAM Review, 55*(2), 375–396.

Braack, M., & Ern, A. (2003). A posteriori control of modeling errors and discretization errors. *Multiscale Modeling & Simulation, 1*(2), 221–238.

Brankin, R., & Gladwell, I. (1989). Shape-preserving local interpolation for plotting solutions of ODEs. *IMA Journal of Numerical Analysis, 9*(4), 555–566.

Brassard, G., & Bratley, P. (1996). *Fundamentals of algorithmics*. Upper Saddle River, NJ: Prentice-Hall.

Braun, M. (1992). *Differential equations and their applications: an introduction to applied mathematics*, vol. 11. New York: Springer.

Brent, R. P. (1973). *Algorithms for minimization without derivatives*. Upper Saddle River, NJ: Prentice-Hall.

Brent, R. (1976). Fast multiple-precision evaluation of elementary functions. *Journal of the ACM, 23*(2), 242–251.

Brent, R. P., & Zimmermann, P. (2011). *Modern computer arithmetic*, vol. 18. Cambridge: Cambridge University Press.

Brezinski, C. (1980). The Mühlbach–Neville–Aitken algorithm and some extensions. *BIT Numerical Mathematics, 20*(4), 443–451.

Brezinski, C., & Redivo-Zaglia, M. (2012). Padé-type rational and barycentric interpolation. Technical report. arXiv:1107.4854.

Briggs, W. L., & Henson, V. E. (1995). *The DFT: an owner's manual for the discrete Fourier transform*. Philadelphia: SIAM.

Briggs, W., & McCormick, S. (2000). *A multigrid tutorial*. Philadelphia: SIAM.

Bronstein, M. (2005). *Symbolic integration I: transcendental functions*. New York: Springer.

Bronstein, M., & Lafaille, S. (2002). Solutions of linear ordinary differential equations in terms of special functions. In: *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, pp. 23–28. New York: ACM.

Brunner, H. (2004). *Collocation methods for Volterra integral and related functional differential equations*. Cambridge: Cambridge University Press.

Brutman, L., & Pinkus, A. (1980). On the Erdos conjecture concerning minimal norm interpolation on the unit circle. *SIAM Journal of Numerical Analysis, 17*(3), 373–375.

Bunse-Gerstner, A., Byers, R., Mehrmann, V., & Nichols, N. (1991). Numerical computation of an analytic singular value decomposition of a matrix valued function. *Numerische Mathematik, 60*(1), 1–39.

Butcher, J. C. (1963). Coefficients for the study of Runge–Kutta integration processes. *Journal of the Australian Mathematical Society, 3*, 185–201.

Butcher, J. C. (1964). Implicit Runge–Kutta processes. *Mathematics of Computation, 18*(85), 50–64.

Butcher, J. C. (1967). A multistep generalization of Runge–Kutta methods with four or five stages. *Journal of ACM, 14*(1), 84–99.

Butcher, J. C. (2001). Numerical methods for ordinary differential equations in the 20th century. *Ordinary Differential Equations and Integral Equations, 6*, 1.

Butcher, J. C. (2008a). Numerical analysis. *Journal of Quality Measurement and Analysis, 4*(1), 1–9.

Butcher, J. C. (2008b). *Numerical methods for ordinary differential equations*. Wiley Online Library. New York: Wiley.

Butcher, J. C., Corless, R. M., Gonzalez-Vega, L., & Shakoori, A. (2011). Polynomial algebra for Birkhoff interpolants. *Numerical Algorithms, 56*(3), 319–347.

Calvetti, D., Reichel, L., & Sgallari, F. (2001). A modified companion matrix method based on Newton polynomials. In: V. Olshevsky (Ed.), *Structured matrices in mathematics, computer science, and engineering I*, vol. 280 of *Contemporary mathematics*. Philadelphia: American Mathematical Society.

Calvo, M., Murua, A., & Sanz-Serna, J. (1994). Modified equations for ODEs. In: *Chaotic numerics: an International Workshop on the Approximation and Computation of Complicated Dynamical Behavior, Deakin University, Geelong, Australia, July 12–16, 1993*, vol. 172 of *Contemporary Mathematics*, pp. 63. Philadelphia: American Mathematical Society.

Campbell, D., & Rose, H. (1983). Preface. In: D. Campbell, & H. Rose (Eds.), *Order in Chaos: Proceedings of the International Conference on Order in Chaos held at the Center for Nonlinear Studies*, pp. vii–viii, Los Alamos.

Cao, Y., Li, S., Petzold, L., & Serban, R. (2003). Adjoint sensitivity analysis for differential-algebraic equations: the adjoint DAE system and its numerical solution. *SIAM Journal on Scientific Computing, 24*(3), 1076.

Chang, Y., & Corliss, G. (1994). ATOMFT: solving ODEs and DAEs using Taylor series. *Computers & Mathematics with Applications, 28*(10-12), 209–233.

Channell, P., & Scovel, C. (1990). Symplectic integration of Hamiltonian systems. *Nonlinearity, 3*, 231.

Chartier, P., Hairer, E., & Vilmart, G. (2010). Algebraic structures of B–series. *Foundations of Computational Mathematics, 10*(4), 407–427.

Chartier, P., Hairer, E., Vilmart, G., & et al. (2007). Numerical integrators based on modified differential equations. *Mathematics of Computation, 76*(260), 1941–1954.

Chen, L., Eberly, W., Kaltofen, E., Saunders, D. B., Turner, W., & Villard, G. (2002). Efficient matrix preconditioners for black box linear algebra. *Linear Algebra and Its Applications, 343*, 119–146.

Chen, M. (1987). On the solution of circulant linear systems. *SIAM Journal on Numerical Analysis, 24*(3), 668–683.

Chin, F. Y. (1977). The partial fraction expansion problem and its inverse. *SIAM Journal on Computing, 6*(3), 554–562.

Christlieb, A., Macdonald, C. B., & Ong, B. (2010). Parallel high-order integrators. *SIAM Journal of Scientific Computing, 32*(2), 818–835.

Chu, M. T. (1984). The generalized Toda flow, the QR algorithm and the center manifold theory. *Journal on Algebraic and Discrete Methods, 5*, 187.

Clark, C. W. (1972). *The theoretical side of calculus*. Belmont, CA: Wadsworth.

Clarkson, P., & Olver, P. (1996). Symmetry and the Chazy equation. *Journal of Differential Equations, 124*(1), 225–246.

Clement, P. A. (1959). A class of triple-diagonal matrices for test purposes. *SIAM Review, 11*(1), 50–52.

Collatz, L. (1966). *The numerical treatment of differential equations* (3rd ed.). New York: Springer.

Cooke, K., den Driessche, P. V., & Zou, X. (1999). Interaction of maturation delay and nonlinear birth in population and epidemic models. *Journal of Mathematical Biology, 39*(4), 332–352.

Cools, R., Kuo, F. Y., & Nuyens, D. (2006). Constructing embedded lattice rules for multivariate integration. *SIAM Journal on Scientific Computing, 28*(6), 2162–2188.

Corless, R. M. (1992). Continued fractions and chaos. *The American Mathematical Monthly, 99*(3), 203–215.

Corless, R. M. (1993). Six, lies, and calculators. *The American Mathematical Monthly, 100*(4), 344–350.

Corless, R. M. (1994a). Error backward. In: P. Kloeden, & K. Palmer, (Eds.), *Proceedings of Chaotic Numerics, Geelong, 1993*, vol. 172 of *AMS Contemporary Mathematics*, pp. 31–62.

Corless, R. M. (1994b). What good are numerical simulations of chaotic dynamical systems? *Computers & Mathematics with Applications, 28*(10–12), 107–121.

Corless, R. M. (2000). An elementary solution of a minimax problem arising in algorithms for automatic mesh selection. *ACM SIGSAM Bulletin, 34*(4), 7–15.

Corless, R. M. (2002). *Essential Maple 7: an introduction for scientific programmers*. New York: Springer.

Corless, R. M. (2004). Computer-mediated thinking. In: *Proceedings of Technology in Mathematics Education*.

Corless, R. M., & Assefa, D. (2007). Jeffery-Hamel flow with Maple: a case study of integration of elliptic functions in a CAS. In: *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, ISSAC '07, pp. 108–115. New York: ACM.

Corless, R., & Ilie, S. (2008). Polynomial cost for solving IVP for high-index DAE. *BIT Numerical Mathematics, 48*(1), 29–49.

Corless, R. M., & Jeffrey, D. J. (1997). The Turing factorization of a rectangular matrix. SIGSAM *Communications in Computer Algebra, 31*(3), 20–30.

Corless, R. M., & Jeffrey, D. J. (2002). The Wright ω function. In: J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, & V. Sorge (Eds.), *Artificial intelligence, automated reasoning, and symbolic computation*, vol. 2385 of *LNAI*, pp. 76–89. New York: Springer.

Corless, R. M., & Watt, S. M. (2004). Bernstein bases are optimal, but, sometimes, Lagrange bases are better. In: *Proceedings of SYNASC, Timisoara*, pp. 141–153. Timisoara: MIRTON Press.

Corless, R. M., Gianni, P. M., & Trager, B. M. (1997). A reordered Schur factorization method for zero-dimensional polynomial systems with multiple roots. In: *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*, ISSAC '97, pp. 133–140. New York: ACM.

Corless, R. M., Gonnet, G., Hare, D., Jeffrey, D., & Knuth, D. E. (1996). On the Lambert *W* function. *Advances in Computational Mathematics, 5*(1), 329–359.

Corless, R. M., Ilie, S., & Reid, G. (2006). Computational complexity of numerical solution of polynomial systems. In: *Proceedings of the Transgressive Computing*, pp. 405–408.

Corless, R. M., Jeffrey, D. J., Watt, S. M., & Davenport, J. H. (2000). According to Abramowitz and Stegun. *SIGSAM Bulletin, 34*(2), 58–65.

Corless, R. M., Shakoori, A., Aruliah, D., & Gonzalez-Vega, L. (2008). Barycentric Hermite interpolants for event location in initial-value problems. *JNAIAM, 3*(1-2), 1–18.

Corliss, G. F. (1980). Integrating ODE's in the complex plane—pole vaulting. *Mathematics of Computation, 35*(152), 1187–1189.

Corliss, G. F., Faure, C., Griewank, A., Hascoët, L., & Naumann, U. (2002). *Automatic differentiation of algorithms: from simulation to optimazation*. New York: Springer.

Cullum, J. (1971). Numerical differentiation and regularization. *SIAM Journal on Numerical Analysis, 8*(2), 254–265.

Curry, H., & Feys, R. (1958). *Combinatory logic*, vol. 1 of *Studies in logic and the foundations of mathematics*. Amsterdam: North-Holland.

Dahlquist, G. (1963). A special stability problem for linear multistep methods. *BIT Numerical Mathematics, 3*(1), 27–43.

Dahlquist, G. (1976). Error analysis for a class of methods for stiff non-linear initial value problems. *Numerical Analysis*, Lecture Notes in Mathematics. Springer Berlin Heidelberg, *506*, 60–72.

Dahlquist, G., & Björck, Å. (2008). *Numerical methods in scientific computing*, vol. 1. Philadelphia: SIAM.

Datta, B. N. (2010). *Numerical linear algebra and applications* (2nd ed.). Philadelphia: SIAM.

Davis, M. (1982). *Computability & unsolvability*. Dover, Mineola, New York.

Davis, P. (1994). *Circulant matrices*. Chelsea Publications, New York, 2nd edition.

Davis, T. (2006). *Direct methods for sparse linear systems*. Philadelphia: SIAM.

Davis, T. A., & Hu, Y. (2011). The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software, 38*(1), 1:1–1:25.

de Boor, C. (1978). A practical guide to splines. Applied Mathematical Sciences (Book 27) Springer NY.

de Boor, C. (2005). Divided differences. *Surveys in approximation theory*, *1*, 46–69.

De Boor, C., & Pinkus, A. (1978). Proof of the conjectures of Bernstein and Erdös concerning the optimal nodes for polynomial interpolation. *Journal of Approximation Theory, 24*, 289–303.

de Camp, L. S. (1960). *The ancient engineers*. Dorset Press, New York, NY.

de Jong, L. (1977). Towards a formal definition of numerical stability. *Numerische Mathematik, 28*(2), 211–219.

Dekker, T. (1969). Finding a zero by means of successive linear interpolation. In: B. Dejon and P. Henrici (editors), *Constructive Aspects of the Fundamental Theorem of Algebra*, Wiley-Interscience, New York, 37–48.

Demmel, J. W. (1997). *Applied numerical linear algebra*. Philadelphia: SIAM.

Demmel, J., & Kågström, B. (1993a). The generalized Schur decomposition of an arbitrary pencil $A - \lambda B$ robust software with error bounds and applications. Part I: theory and algorithms. *ACM Transactions on Mathematical Software, 19*(2), 160–174.

Demmel, J., & Kågström, B. (1993b). The generalized Schur decomposition of an arbitrary pencil $A - \lambda B$ robust software with error bounds and applications. Part II: software and applications. *ACM Transactions on Mathematical Software, 19*(2), 175–201.

Demmel, J., & Koev, P. (2005). The accurate and efficient solution of a totally positive generalized Vandermonde linear system. *SIAM Journal on Matrix Analysis and Applications, 27*(1), 142–152.

Deuflhard, P. (2011). *Newton methods for nonlinear problems: affine invariance and adaptive algorithms*, vol. 35. New York: Springer.

Deuflhard, P., & Bornemann, F. (2002). *Scientific computing with ordinary differential equations*. New York: Springer.

Deuflhard, P., & Hohmann, A. (2003). *Numerical analysis in modern scientific computing: an introduction*, vol. 43. New York: Springer.

Dixon, J. (1982). Exact solution of linear equations using p-adic expansions. *Numerische Mathematik, 40*(1), 137–141.

Don, W. S., & Solomonoff, A. (1995). Accuracy and speed in computing the Chebyshev collocation derivative. *SIAM Journal of Scientific Computing, 16*, 1253–1268.

Driscoll, T. A. (2010). Automatic spectral collocation for integral, integro-differential, and integrally reformulated differential equations. *J. Comput. Phys., 229*, 5980–5998.

Edelman, A., & Murakami, H. (1995). Polynomial roots from companion matrix eigenvalues. *Mathematics of Computation, 64*(210), 763–776.

Elliot, Ralph WV. Runes: An Introduction Praeger (1981)

Enright, W. H. (1989a). Analysis of error control strategies for continuous Runge-Kutta methods. *SIAM Journal on Numerical Analysis, 26*, 588–599.

Enright, W. H. (1989b). A new error-control for initial value solvers. In: *Proceedings of the Conference on Numerical Ordinary Differential Equations*, pp. 288–301. New York: ACM.

Enright, W. H. (2000a). Accurate approximate solution of partial differential equations at off-mesh points. *ACM Transactions on Mathematical Software, 26*, 274–292.

Enright, W. H. (2000b). Continuous numerical methods for ODEs with defect control. *Journal of Computational and Applied Mathematics, 125*, 159–170.

Enright, W. (2006a). Software for ordinary and delay differential equations: accurate discrete approximate solutions are not enough. *Applied Numerical Mathematics, 56*(3-4), 459–471.

Enright, W. H. (2006b). Verifying approximate solutions to differential equations. *Journal of Computational and Applied Mathematics, 185*, 203–211.

Enright, W. H., & Hayashi, H. (1997a). A delay differential equation solver based on a continuous Runge–Kutta method with defect control. *Numerical Algorithms, 16*(3), 349–364.

Enright, W. H., & Hayashi, H. (1997b). The evaluation of numerical software for delay differential equations. In: *Proceedings of the IFIP TC2/WG2.5 Working Conference on Quality of Numerical Software: Assessment and Enhancement*, pp. 179–193. London: Chapman & Hall.

Enright, W. H., & Hayes, W. B. (2007). Robust and reliable defect control for Runge–Kutta methods. *ACM Transactions on Mathematical Software, 33*(1), 1–19.

Enright, W., & Higham, D. (1991). Parallel defect control. *BIT Numerical Mathematics, 31*(4), 647–663.

Enright, W. H., & Muir, P. H. (1996). Runge-Kutta software with defect control for boundary value ODEs. *SIAM Journal of Scientific Computing, 17*, 479–497.

Enright, W. H., & Pryce, J. D. (1987). Two FORTRAN packages for assessing initial value methods. *ACM Transactions on Mathematical Software, 13*(1), 1–27.

Ercegovac, M. D., & Lang, T. (2004). *Digital arithmetic*. Elsevier Science, San Francisco, CA.

Essex, G. C., Davison, M., & Schulzky, C. (2000). Numerical monsters. *SIGSAM Bulletin: Communications in Computer Algebra, 34*, 16–32.

Farin, G. (1996). *Curves and surfaces for computer-aided geometric design: a practical code*. Boston: Academic Press.

Farouki, R. T., & Goodman, T. N. T. (1996). On the optimal stability of the Bernstein basis. *Mathematics of Computation, 65*(216), 1553–1566.

Farouki, R., & Rajan, V. (1987). On the numerical condition of polynomials in Bernstein form. *Computer Aided Geometric Design, 4*(3), 191–216.

Feldstein, A., Neves, K. W., & Thompson, S. (2006). Sharpness results for state dependent delay differential equations: An overview. *Applied Numerical Mathematics, 56*, 472–487.

Fornberg, B., & Weideman, J. (2011). A numerical methodology for the Painlevé equations. *Journal of Computational Physics, 230*(2011), 5957–5973.

Forsythe, G., & Moler, C. (1967). *Computer solution of linear algebraic systems*. Englewood Cliffs, NJ: Prentice-Hall.

Forsythe, G. E. (1966). How do you solve a quadratic equation. Technical Report CS40, Stanford.

Forsythe, G. E. (1970). Pitfalls in computation, or why a math book isn't enough. *The American Mathematical Monthly, 77*(9), 931–956.

Forth, S. A. (2006). An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software, 32*(2), 195–222.

Franzone, P. C., Deuflhard, P., Erdmann, B., Lang, J., & Pavarino, L. F. (2006). Adaptivity in space and time for reaction-diffusion systems in electrocardiology. *SIAM Journal of Scientific Computing, 28*, 942–962.

Freeth, T., Jones, A., Steele, J., & Bitsakis, Y. (2008). Calendars with Olympiad display and eclipse prediction on the Antikythera Mechanism. *Nature, 454*(7204), 614–617.

Gautschi, W. (1975). Optimally conditioned Vandermonde matrices. *Numerische Mathematik, 24*(1), 1–12.

Gautschi, W. (1983). The condition of Vandermonde-like matrices involving orthogonal polynomials. *Linear Algebra and Its Applications, 52*, 293–300.

Geddes, K. O., & Fee, G. J. (1992). Hybrid symbolic-numeric integration in MAPLE. In: *Papers from the international symposium on Symbolic and algebraic computation*, ISSAC '92, pp. 36–41. New York, NY: ACM.

Geddes, K. O., & Mason, J. C. (1975). Polynomial approximation by projections on the unit circle. *SIAM Journal on Numerical Analysis, 12*(1), 111–120.

Geddes, K. O., Czapor, S. R., & Labahn, G. (1992). *Algorithms for computer algebra*. Boston: Kluwer Academic.

Geist, K., Parlitz, U., & Lauterborn, W. (1990). Comparison of different methods for computing Lyapunov exponents. *Progress of Theoretical Physics, 83*(5), 875–893.

Gemignani, L. (2005). A unitary Hessenberg QR-based algorithm via semiseparable matrices. *Journal of Computational and Applied Mathematics, 184*(2), 505–517.

Gibbons, A. (1960). A program for the automatic integration of differential equations using the method of Taylor series. *The Computer Journal, 3*(2), 108–111.

Giesbrecht, M., Labahn, G., & Lee, W.-s. (2009). Symbolic-numeric sparse interpolation of multivariate polynomials. *Journal of Symbolic Computation, 44*(8), 943–959.

Gil, A., Segura, J., & Temme, N. M. (2002). Computing complex Airy functions by numerical quadrature. *Numerical Algorithms, 30*, 11–23. 10.1023/A:1015636825525.

Gil, A., Segura, J., & Temme, N. M. (2007). *Numerical methods for special functions.* Philadelphia: SIAM.

Giles, M., & Süli, E. (2002). Adjoint methods for PDEs: a posteriori error analysis and postprocessing by duality. *Acta Numerica, 11*, 145–236.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *Computing Surveys, 23*(1), 5–48.

Golub, G., & Uhlig, F. (2009). The QR algorithm: 50 years later its genesis by John Francis and Vera Kublanovskaya and subsequent developments. *IMA Journal of Numerical Analysis, 29*, 467–485.

Golub, G., & Welsch, J. (1969). Calculation of Gauss quadrature rules. *Mathematics of Computation, 23*, 221–230.

Golub, G. H., & van Loan, C. F. (1996). *Matrix computations* (3rd ed.). Baltimore: The Johns Hopkins University Press.

Golubitsky, O., & Watt, S. (2009). Online recognition of multi-stroke symbols with orthogonal series. In: *2009 10th International Conference on Document Analysis and Recognition*, pp. 1265–1269. IEEE, Washington, DC.

Golubitsky, O., & Watt, S. (2010). Distance-based classification of handwritten symbols. *International Journal on Document Analysis and Recognition, 13*(2), 133–146.

Gonnet, P. (2010). Increasing the reliability of adaptive quadrature using explicit interpolants. *ACM Transactions on Mathematical Software, 37*, 26:1–26:32.

Good, I. (1961). The colleague matrix, a Chebyshev analogue of the companion matrix. *The Quarterly Journal of Mathematics, 12*(1), 61.

Gosling, J. (1998). Extensions to Java for numerical computing. Keynote address at the ACM 1998 Workshop on Java for High-Performance Network Computing held at Stanford University.

Graham, R., Knuth, D., & Patashnik, O. (1994). *Concrete mathematics: a foundation for computer science*. Reading, MA: Addison-Wesley.

Graillat, S., & Trébuchet, P. (2009). A new algorithm for computing certified numerical approximations of the roots of a zero-dimensional system. In: *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*, ISSAC '09, pp. 167–174. New York: ACM.

Grcar, J. (2011). John von Neumann's analysis of Gaussian elimination and the origins of modern numerical analysis. *SIAM Review*, 53(4):607–682.

Green, K., & Wagenknecht, T. (2006). Pseudospectra and delay differential equations. *Journal of Computational and Applied Mathematics, 196*(2), 567–578.

Griffiths, D., & Sanz-Serna, J. (1986). On the scope of the method of modified equations. *SIAM Journal on Scientific and Statistical Computing, 7*, 994.

Guggenheimer, H., Edelman, A., & Johnson, C. (1995). A simple estimate of the condition number of a linear system. *College Mathematics Journal, 26*(1), 2–5.

Habgood, K., & Arel, I. (2011). A condensation-based application of Cramer's rule for solving large-scale linear systems. *Journal of Discrete Algorithms, 10*, 98–109.

Hairer, E., & Wanner, G. (1996). *Analysis by its history*. New York: Springer.

Hairer, E., & Wanner, G. (2000). Order stars and stiff integrators. *Journal of Computational and Applied Mathematics, 125*(1-2), 93–105.

Hairer, E., & Wanner, G. (2002). *Solving ordinary differential equations II: stiff and differential-algebraic problems*, vol. 14. New York: Springer.

Hairer, E., Lubich, C., & Wanner, G. (2006). *Geometric numerical integration: structure-preserving algorithms for ordinary differential equations*. New York: Springer.

Hairer, E., Nørsett, S. P., & Wanner, G. (1993). *Solving ordinary differential equations: nonstiff problems*. New York: Springer.

Hamming, R. W. (1973). *Numerical methods for scientists and engineers* (2nd ed.). New York: McGraw-Hill.

Hanson, P. M., & Enright, W. H. (1983). Controlling the defect in existing variable-order Adams codes for initial-value problems. *ACM Transactions on Mathematical Software, 9*, 71–97.

Hardy, G. (1949). *Divergent series*. Clarendon Press, Oxford.

Heffernan, J. M., & Corless, R. M. (2006). Solving some delay differential equations with computer algebra. *Mathematical Scientist, 31*(1), 21–34.

Heinig, G. (2001). Stability of toeplitz matrix inversion formulas. In: V. Olshevsky, (Ed.), *Structured matrices in mathematics, computer science, and engineering I*, vol. 280 of *Contemporary mathematics*. Philadelphia: American Mathematical Society.

Henrici, P. (1964). *Elements of numerical analysis*. New York: Wiley.

Henrici, P. (1974). *Applied and computational complex analysis*, vol. 1. New York: Wiley.

Henrici, P. (1977). *Applied and computational complex analysis*, vol. 2. New York: Wiley.

Henrici, P. (1979a). Barycentric formulas for interpolating trigonometric polynomials and their conjugates. *Numerische Mathematik, 33*, 225–234.

Henrici, P. (1979b). Fast Fourier methods in computational complex analysis. *SIAM Review, 21*(4), 481–527.

Henrici, P. (1982). *Essentials of numerical analysis*. New York: Wiley.

Hickernell, F., & Wozniakowski, H. (2001). The price of pessimism for multidimensional quadrature. *Journal of Complexity, 17*(4), 625–659.

Higham, D., & Stuart, A. (1998). Analysis of the dynamics of local error control via a piecewise continuous residual. *BIT Numerical Mathematics, 38*, 44–57.

Higham, D., & Trefethen, L. (1993). Stiffness of ODEs. *BIT Numerical Mathematics, 33*(2), 285–303.

Higham, D. J. (1989a). Robust defect control with Runge–Kutta schemes. *SIAM Journal on Numerical Analysis, 26*(5), 1175–1183.

Higham, D. J., & Higham, N. J. (1992). Backward error and condition of structured linear systems. *SIAM Journal on Matrix Analysis and Applications, 13*(1), 162–175.

Higham, D. J., & Higham, N. J. (2005). *MATLAB guide*. Philadelphia: SIAM.

Higham, N. (1989b). The accuracy of solutions to triangular systems. *SIAM Journal on Numerical Analysis, 26*(5), 1252–1265.

Higham, N. (2008). *Functions of matrices: theory and computation*. Philadelphia: SIAM.

Higham, N. J. (2002). *Accuracy and stability of numerical algorithms* (2nd ed.). Philadelphia: SIAM.

Higham, N. J. (2004). The numercal stability of barycentric Lagrange interpolation. *IMA Journal of Numerical Analysis, 24*, 547–556.

Hoffman, P. (1998). *The man who loved only numbers: the story of Paul Erdös and the search for mathematical truth*. New York: Hyperion.

Hogben, L. (Ed.), (2006). *Handbook of linear algebra*. Chapman and Hall/CRC, Boca Raton, FL.

Horn, R. A., & Johnson, C. R. (1990). *Matrix analysis*. Cambridge: Cambridge University Press.

Huang, W., & Russell, R. D. (2011). *Adaptive moving mesh methods*, vol. 174 of *Applied mathematical sciences*. New York: Springer.

Hubbard, J. H., & West, B. H. (1991). *Differential equations: a dynamical systems approach*, vol. 5, 18. New York: Springer.

Hull, T., Fairgrieve, T., & Tang, P. (1994). Implementing complex elementary functions using exception handling. *ACM Transactions on Mathematical Software (TOMS), 20*(2), 215–244.

Hyman, J. (1983). Accurate monotonicity preserving cubic interpolation. *SIAM Journal on Scientific and Statistical Computing, 4*(4), 645–654.

Ilie, S., Söderlind, G., & Corless, R. M. (2008). Adaptivity and computational complexity in the numerical solution of ODEs. *Journal of Complexity, 24*(3), 341–361.

Immerman, N. (1999). *Descriptive complexity*. New York: Springer.

Iserles, A. (1994). On nonlinear delay differential equations. *Transactions of the American Mathematical Society, 344*(1), 441–477.

Iserles, A. (2009). *A first course in the numerical analysis of differential equations*. Cambridge: Cambridge University Press.

Iserles, A., Nørsett, S., & Olver, S. (2006). In: A. Bermúdez de Castro, D. Gómez, P. Quintela, P. Salgado (Eds.), Highly oscillatory quadrature: the story so far. *Numerical Mathematics and Advanced Applications*, Springer Berlin Heidelberg, 97–118.

Jarlebring, E., & Damm, T. (2007). Technical communique: the Lambert *W* function and the spectrum of some multidimensional time-delay systems. *Automatica, 43*, 2124–2128.

Jenkins, M., & Traub, J. (1970). A three-stage variable-shift iteration for polynomial zeros and its relation to generalized Rayleigh iteration. *Numerische Mathematik, 14*(3), 252–263.

Johnson, R. (2010). *The elements of* MATLAB *style*. Cambridge: Cambridge University Press.

Jones, W. B., & Thron, W. J. (1974). Numerical stability in evaluating continued fractions. *Mathematics of Computation, 28*(127), 795–810.

Kahan, W. (1980). Handheld calculator evaluates integrals. *Hewlett-Packard Journal, 31*(8), 23–32.

Kahan, W. (1986). Branch cuts for complex elementary functions. In: *Proceedings of the joint IMA/SIAM Conference on the State of the Art in Numerical Analysis*. Oxford University Press.

Kahan, W. (2009). *Needed remedies for the undebuggability of large-scale floating-point computations in science and engineering*. Presented at the Computer Science Dept. Colloquia at the University of California, Berkely.

Kahan, W., & Darcy, J. D. (1998). How Java's floating-point hurts everyone everywhere. ACM 1998 Workshop on Java for High-Performance Network Computing held at Stanford University.

Kahaner, D., Moler, C., & Nash, S. (1989). Numerical methods and software. *Englewood Cliffs*, Prentice–Hall, Englewood Cliffs, NJ.

Kailath, T., & Sayed, A. (1995). Displacement structure: theory and applications. *SIAM Review, 37*, 297.

Kansy, K. (1973). Elementare fehlerdarstellung für ableitungen bei der Hermite-interpolation. *Numerische Mathematik, 21*(4), 350–354.

Khashin, S. (2009). A symbolic-numeric approach to the solution of the Butcher equations. *Canadian Applied Mathematics Quarterly, 17*(3), 555–569.

Khashin, S. (2012). Butcher algebras for Butcher systems. *Numerical Algorithms, 61*(2), 1–11.

Kierzenka, J., & Shampine, L. F. (2001). A BVP solver based on residual control and the Matlab PSE. *ACM Transactions on Mathematical Software, 27*, 299–316.

Knuth, D. (1981). *The art of computer programming: seminumerical algorithms*, vol. 2. Reading, Massachusetts. Addison-Wesley.

Körner, T. (1989). *Fourier analysis*. Cambridge: Cambridge University Press.

Kreiss, H.-O., & Lorenz, J. (1989). *Initial-boundary value problems and the Navier–Stokes equations*. Boston: Academic Press.

Kulisch, U. W. (2002). *Advanced Arithmetic for the digital computer: design of arithmetic units*. New York: Springer.

Kung, H., & Tong, D. (1977). Fast algorithms for partial fraction decomposition. *SIAM Journal on Computing, 6*, 582.

Kuo, F., & Sloan, I. (2005). Lifting the curse of dimensionality. *Notices of the AMS, 52*(11), 1320–1328.

Lakshmanan, M., & Rajasekar, S. (2003). *Nonlinear dynamics: integrability, chaos, and patterns*. New York: Springer.

Lanczos, C. (1988). *Applied analysis*. Dover, New York, NY.

Laszkiewicz, B., & Ziętak, K. (2009). A Padé family of iterations for the matrix sector function and the matrix *p*th root. *Numerical Linear Algebra with Applications, 16*(11-12), 951–970.

Lawrence, P. W., & Corless, R. M. (2011). Numerical stability of barycentric Hermite root-finding. In: *Proceedings of the 2011 International Workshop on Symbolic-Numeric Computation*, SNC '11, pp. 147–148. New York: ACM.

Lawrence, P. W., Corless, R. M., & Jeffrey, D. J. (2012). Algorithm 917: complex double-precision evaluation of the Wright ω function. *ACM Transactions on Mathematical Software, 38*(3), 20:1–20:17.

Lawrence, Piers W., & Robert M. Corless. (2013). Stability of rootfinding for barycentric Lagrange interpolants. *Numerical Algorithms*, 1–18.

Lele, S. K. (1992). Compact finite difference schemes with spectral-like resolution. *Journal of Computational Physics, 103*(1), 16–42.

Leon, S. J., Björck, Å., & Gander, W. (2013). Gram–Schmidt orthogonalization: 100 years and more. *Numerical Linear Algebra with Applications, 20*(3), 492–532.

Leppänen, T., Karttunen, M., Kaski, K., Barrio, R. A., & Zhang, L. (2002). A new dimension to Turing patterns. *Physica D: Nonlinear Phenomena, 168-169*, 35–44.

Levinson, N., & Redheffer, R. (1970). *Complex variables*. Holden-Day, San Francisco, CA.

Li, S., & Petzold, L. (2004). Adjoint sensitivity analysis for time-dependent partial differential equations with adaptive mesh refinement. *Journal of Computational Physics, 198*(1), 310–325.

Li, Z., Osborne, M., & Prvan, T. (2005). Parameter estimation of ordinary differential equations. *IMA Journal of Numerical Analysis, 25*(2), 264–285.

Lu, S., & Pereverzev, S. V. (2006). Numerical differentiation from a viewpoint of regularization theory. *Mathematics of Computation, 75*(256), 1853.

Luk, F. T., & Qiao, S. (2003). A fast singular value algorithm for hankel matrices. In: V. Olshevsky, (Ed.), *Fast algorithms for structured matrices: theory and applications*, vol. 323, pp. 169–178. Philadelphia: American Mathematical Society.

Lyche, T., & Peña, J. M. (2004). Optimally stable multivariate bases. *Advances in Computational Mathematics, 20*, 149–159.

Lyness, J. N., & Moler, C. B. (1967). Numerical differentiation of analytic functions. *SIAM Journal on Numerical Analysis, 4*(2), 202–210.

Manocha, D. (1994). Solving systems of polynomial equations. *IEEE Computer Graphics and Applications, 14*, 46–55.

Manocha, D., & Demmel, J. (1994). Algorithms for intersecting parametric and algebraic curves I: simple intersections. *ACM Transactions on Graphics, 13*(1), 73–100.

Martelli, M., Dang, M., & Seph, T. (1998). Defining chaos. *Mathematics Magazine, 71*(2), 112–122.

Mattheij, R., & Molenaar, J. (1996). *Ordinary differential equations in theory and practice*. New York: Wiley.

McNamee, J. (1993). A bibliography on roots of polynomials. *Journal of Computational and Applied Mathematics, 47*(3), 391–394.

McNamee, J. (1997). A supplementary bibliography on roots of polynomials. *Journal of Computational and Applied Mathematics, 78*(1), 1–2.

McNamee, J. (2007). *Numerical methods for roots of polynomials*. Elsevier Science, Amsterdam.

Meyer, C. D. (2001). *Matrix analysis and applied linear algebra*. Philadelphia: SIAM.

Michiels, W., & Niculescu, S.-I. (2007). *Stability and stabilization of time-delay systems: an eigenvalue-based approach*, vol. 12. Philadelphia: SIAM.

Mill, J. S. (1873). *Autobiography*. Penguin Books. London, 1989.

Milne-Thomson, L. M. (1951). *The calculus of finite differences* (2nd ed., 1st ed. 1933). MacMillan and Company, London.

Moir, R. H. C. (2010). Reconsidering backward error analysis for ordinary differential equations. Master's thesis, The University of Western Ontario.

Moler, C. (2004). *Numerical computing with* MATLAB (Electronic ed.). Philadelphia: SIAM.

Moler, C. B., & Morrison, D. (1983). Replacing square roots by Pythagorean sums. *IBM Journal of Research and Development, 27*(6), 577–581.

Morgan, A. P. (1987). *Solving polynomial systems using continuation for engineering and scientific problems*. Englewood Cliffs, NJ: Prentice-Hall.

Mori, M., & Sugihara, M. (2001). The double-exponential transformation in numerical analysis. *Journal of Computational and Applied Mathematics, 127*(1-2), 287–296.

Morton, K. W., & Mayers, D. F. (1994). *Numerical solution of partial differential equations: an introduction*. Cambridge: Cambridge University Press.

Muir, P. H., Pancer, R. N., & Jackson, K. R. (2003). PMIRKDC: a parallel mono-implicit Runge–Kutta code with defect control for boundary value ODEs. *Parallel Computing, 29*(6), 711–741.

Muller, J., Brisebarre, N., de Dinechin, F., Jeannerod, C., Melquiond, G., Revol, N., Stehlé, D., & Torres, S. (2009). *Handbook of floating-point arithmetic*. Basel: Birkhäuser.

Muller, N., Magaia, L., & Herbst, B. (2004). Singular value decomposition, eigenfaces, and 3D reconstructions. *SIAM Review, 46*(3), 518–545.

Nagle, R., Saff, E., & Snider, A. (2000). *Fundamentals of differential equations and boundary value problems* (3rd ed.). Addison-Wesley.

Nedialkov, N., & Jackson, K. (1999). An interval Hermite–Obreschkoff method for computing rigorous bounds on the solution of an initial value problem for an ordinary differential equation. *Reliable Computing, 5*(3), 289–310.

Nedialkov, N., & Pryce, J. (2005). Solving differential-algebraic equations by Taylor series (I): computing Taylor coefficients. *BIT Numerical Mathematics, 45*(3), 561–591.

Nedialkov, N., & Pryce, J. (2007). Solving differential-algebraic equations by Taylor series (II): computing the system Jacobian. *BIT Numerical Mathematics, 47*(1), 121–135.

Needham, T. (1999). *Visual complex analysis*. Oxford: Oxford University Press.

Neumaier, A. (2001). *Introduction to numerical analysis*. Cambridge: Cambridge University Press.

Nievergelt, Y. (1991). Numerical linear algebra on the HP-28 or how to lie with supercalculators. *The American Mathematical Monthly, 98*(6), 539–544.

Niven, I. (1981). *Maxima and minima without calculus*. Number 6 in Dolciani Mathematical Expositions. The Mathematical Association of America, Washington, D.C.

Nowak, U. (1996). A fully adaptive MOL-treatment of parabolic 1-D problems with extrapolation techniques. *Applied Numerical Mathematics, 20*(1-2), 129–141.

Oettli, W., Prager, W., & Wilkinson, J. (1965). Admissible solutions of linear systems with not sharply defined coefficients. *Journal of the Society for Industrial and Applied Mathematics: Series B, Numerical Analysis, 2*(2), 291–299.

Olds, C. D. (1963). *Continued fractions*. New York: Random House.

Olshevsky, V., (Ed.), (2001a). *Structured matrices in mathematics, computer science, and engineering I*, vol. 280 of *Contemporary mathematics*. Philadelphia: American Mathematical Society.

Olshevsky, V., (Ed.), (2001b). *Structured matrices in mathematics, computer science, and engineering II*, vol. 281 of *Contemporary mathematics*. Philadelphia: American Mathematical Society.

Olshevsky, V. (2003a). *Fast algorithms for structured matrices: theory and applications,* vol. 323 of *Contemporary mathematics*. Philadelphia: American Mathematical Society.

Olshevsky, V. (2003b). *Fast algorithms for structured matrices: theory and applications*, vol. 323. Philadelphia: American Mathematical Society.

Olver, S., & Townsend, A. (2013). A fast and well-conditioned spectral method. *SIAM Review, 55*(3), 462–489.

O'Meara, K., Clark, J., & Vinsonhaler, C. (2011). *Advanced topics in linear algebra: weaving matrix problems through the Weyr form*. Oxford: Oxford University Press.

Orendt, T. (2011). *Resolution of Geometric Singularities by Complex Detours–Modeling, Complexity and Application*. PhD thesis, Universitätsbibliothek der TU München.

Ostrowski, A. (1940). Recherches sur la méthode de Graeffe et les zéros des polynomes et des séries de Laurent. *Acta Mathematica, 72*, 99–257.

Ostrowski, A. (1973). *Solution of equations in Euclidean and Banach spaces* (3rd ed.). New York: Academic Press.

Overton, M. L. (2001). *Numerical computing with IEEE floating point arithmetic*. Philadelphia: SIAM.

Owren, B., & Zennaro, M. (1991). Order barriers for continuous explicit Runge–Kutta methods. *Mathematics of Computation, 56*(56), 645–661.

Pachón, R., & Trefethen, L. (2009). Barycentric–Remez algorithms for best polynomial approximation in the Chebfun system. *BIT Numerical Mathematics, 49*(4), 721–741.

Pachón, R., Gonnet, P., & Van Deun, J. (2012). Fast and stable rational interpolation in roots of unity and Chebyshev points. *SIAM Journal on Numerical Analysis, 50*(3), 1713–1734.

Pan, V. Y., & Zheng, A.-L. (2011). New progress in real and complex polynomial root-finding. *Computers & Mathematics with Applications, 61*(5), 1305–1334.

Pancer, R., Jackson, K., & et al. (1997). The parallel solution of almost block diagonal systems arising in numerical methods for BVPs for ODEs. In: *the Proceedings of the Fifteenth IMACS World Congress, Berlin, Germany*, vol. 2, pp.57–62. Citeseer.

Parhami, B. (2000). *Computer arithmetic: algorithms and hardware designs*. Oxford: Oxford University Press.

Patwa, Z., & Wahl, L. (2008). Fixation probability for lytic viruses: the attachment-lysis model. *Genetics, 180*(1), 459.

Petković, M., Petković, L., & Herceg, D. (2010). On Schröder's families of root-finding methods. *Journal of Computational and Applied Mathematics, 233*(8), 1755–1762.

Pettigrew, M. F., & Rasmussen, H. (1996). A compact method for second-order boundary value problems on nonuniform grids. *Computers & Mathematics with Applications, 31*(9), 1–16.

Petzold, L., Li, S., Cao, Y., & Serban, R. (2006). Sensitivity analysis of differential-algebraic equations and partial differential equations. *Computers & Chemical Engineering, 30*(10–12), 1553–1559. Papers from Chemical Process Control VII - CPC VII.

Pour-El, M., & Richards, J. (1989). *Computability in analysis and physics*. Berlin: Springer.

Press, W., Flannery, B., Teukolsky, S., Vetterling, W., & et al. (1986). *Numerical recipes*. Cambridge: Cambridge University Press.

Preto, M., & Tremaine, S. (1999). A class of symplectic integrators with adaptive time step for separable Hamiltonian systems. *The Astronomical Journal, 118*, 2532.

Priest, D. M. (2004). Efficient scaling for complex division. *ACM Transactions on Mathematical Software, 30*(4), 389–401.

Provatas, N., & Elder, K. (2010). *Phase-field methods in materials science and engineering*. New York: Wiley-VCH.

Püschel, M., Franchetti, F., & Voronenko, Y. (2011). Spiral. *Encyclopedia of parallel computing*. New York: Springer.

Qin, X., Wu, W., Feng, Y., & Reid, G. (2012). Structural analysis of high-index DAE for process simulation. *arXiv preprint arXiv:1206.6177*.

Quarteroni, A., Sacco, R., & Saleri, F. (2007). *Numerical mathematics*. New York: Springer.

Ramsay, J., Hooker, G., Campbell, D., & Cao, J. (2007). Parameter estimation for differential equations: a generalized smoothing approach. *Journal of the Royal Statistical Society: Series B (Statistical Methodology), 69*(5), 741–796.

Rebillard, L. (1997). Rational approximation in the complex plane using a τ-method and computer algebra. *Numerical Algorithms, 16*(2), 187–208.

Rezvani, N., & Corless, R. M. (2005). The nearest polynomial with a given zero, revisited. *SIGSAM Bulletin, 39*, 73–79.

Rezvani Dehaghani, N. (2005). Approximate polynomials in different bases. Master's thesis, University of Western Ontario.

Rihan, F. A. (2003). Sensitivity analysis for dynamic systems with time-lags. *Journal of Computational and Applied Mathematics, 151*(2), 445–462.

Rihan, F. A. (2006). Sensitivity analysis of cell growth dynamics with time lags. *Journal of the Egyptian Mathematical Society, 14*, 91–107.

Rivlin, T. (1990). *Chebyshev polynomials: from approximation theory to number theory*. New York: Wiley.

Robidoux, N. (2002). *Numerical solution of the steady diffusion equation with discontinuous coefficients*. PhD thesis, The University of New Mexico.

Rokicki, J., & Floryan, J. (1995). Domain decomposition and the compact fourth-order algorithm for the Navier–Stokes equations. *Journal of Computational Physics, 116*(1), 79–96.

Ronveaux, A., & Rebillard, L. (2002). Expansion of multivariable polynomials in products of orthogonal polynomials in one variable. *Applied Mathematics and Computation, 128*(2-3), 387–414.

Ruuth, S. J. (1995). Implicit-explicit methods for reaction-diffusion problems in pattern formation. *Journal of Mathematical Biology, 34*(2), 148–176.

Saff, E. B., & Snider, A. D. (1993). *Fundamentals of complex analysis for mathematics, science, and engineering* (2nd ed.). Englewood Cliffs, NJ: Prentice-Hall.

Sakurai, T., & Sugiura, H. (2003). A projection method for generalized eigenvalue problems using numerical integration. *Journal of Computational and Applied Mathematics, 159*(1), 119–128.

Salzer, H. (1972). Lagrangian interpolation at the Chebyshev points $x_{n,\nu} = \cos(\nu\pi/n)$, $\nu = 0(1)n$; some unnoted advantages. *The Computer Journal, 15*(2), 156.

Schmidt, E. (1907). Zur Theorie der linearen und nichtlinearen integralgleichungen. *Mathematische Annalen, 63*(4), 433–476.

Schneider, C., & Werner, W. (1991). Hermite interpolation: the barycentric approach. *Computing, 46*(1), 35–51.

Schönfinkel, M. (1924). On the building blocks of mathematical logic. In: J. Van Heijenoort, (Ed.), *From Frege to Gödel: a source book in mathematical logic, 1879–1931*, pp. 355. Harvard University Press, Harvard, MA.

Scott, D., & Peeples, D. R. (1988). A constructive proof of the partial fraction decomposition. *American Mathematical Monthly, 95*, 651–653.

Shakoori, A. (2008). *Bivariate polynomial solver by values*. PhD thesis, The University of Western Ontario.

Shampine, L. (1985). Interpolation for Runge–Kutta methods. *SIAM Journal on Numerical Analysis, 22*(5), 1014–1027.

Shampine, L. (2002). Variable order Adams codes. *Computers & Mathematics with Applications, 44*(5-6), 749–761.

Shampine, L. F. (2005). Solving ODEs and DDEs with residual control. *Applied Numerical Mathematics, 52*, 113–127.

Shampine, L. (2008a). Dissipative approximations to neutral DDEs. *Applied Mathematics and Computation, 203*(2), 641–648.

Shampine, L. F. (2008b). Vectorized adaptive quadrature in MATLAB. *Journal of Computational and Applied Mathematics, 211*(2), 131–140.

Shampine, L. F. (2010). Weighted quadrature by change of variable. *Neural, Parallel, and Scientific Computations, 18*, 195–206.

Shampine, L. F., & Corless, R. M. (2000). Initial value problems for ODEs in problem solving environments. *Journal of Computational and Applied Mathematics, 125*(1), 31–40.

Shampine, L. F., & Gordon, M. (1975). *Computer solution of ordinary differential equations: the initial value problem*. Freeman, San Francisco, CA.

Shampine, L., & Gear, C. (1979). A user's view of solving stiff ordinary differential equations. *SIAM Review, 21*, 1–17.

Shampine, L. F., & Gordon, M. (1975). *Computer solution of ordinary differential equations: the initial value problem*. Freeman.

Shampine, L., & Reichelt, M. (1997). The Matlab ODE suite. *SIAM Journal on Scientific Computing, 18*(1), 1–22.

Shampine, L., Allen, R., & Pruess, S. (1997). *Fundamentals of numerical computing*. New York: Wiley.

Shampine, L., Gladwell, I., & Thompson, S. (2003). *Solving ODEs with MATLAB*. Cambridge: Cambridge University Press.

Shonkwiler, R. W., & Herod, J. (2009). *Mathematical biology*. New York: Springer.

Skeel, R., & Berzins, M. (1990). A method for the spatial discretization of parabolic equations in one space variable. *SIAM Journal on Scientific and Statistical Computing, 11*(1), 1–32.

Skeel, R. D. (1980). Iterative refinement implies numerical stability for Gaussian elimination. *Mathematics of Computation, 35*(151), 817–832.

Skokos, C. (2010). The Lyapunov characteristic exponents and their computation. In: J.J. Souchay, R. Dvorak (Eds.), *Dynamics of Small Solar System Bodies and Exoplanets*, Springer Berlin Heidelberg, pp. 63–135.

Sloan, I., & Joe, S. (1994). *Lattice methods for multiple integration*. Oxford: Oxford University Press.

Sloan, I., & Wozniakowski, H. (2001). Tractability of multivariate integration for weighted Korobov classes. *Journal of Complexity, 17*(4), 697–721.

Smith, R. L. (1962). Algorithm 116: Complex division. *Communications of the ACM, 5*(8), 435.

Smoktunowicz, A. (2002). Backward stability of Clenshaw's algorithm. *BIT Numerical Mathematics, 42*(3), 600–610.

Smoktunowicz, A., Barlow, J., & Langou, J. (2006). A note on the error analysis of classical Gram-Schmidt. *Numerische Mathematik, 105*(2), 299–313.

Söderlind, G. (1984). On nonlinear difference and differential equations. *BIT Numerical Mathematics, 24*(4), 667–680.

Söderlind, G. (2003). Digital filters in adaptive time-stepping. *ACM Transactions on Mathematical Software, 29*, 1–26.

Sommese, A. J., & Wampler, C. W. (2005). *The numerical solution of systems of polynomials arising in engineering and science*. Hackensack, NJ: World Scientific.

Specht, W. (1960). Die lage der nullstellen eines polynoms. iv. *Mathematische Nachrichten, 21*, 201–222.

Squire, W., & Trapp, G. (1998). Using complex variables to estimate derivatives of real functions. *SIAM Review, 40*(1), 110–112.

Steele, G. (1990). *Common LISP: the language*. Digital Press, Woburn, MA.

Steele, J. M. (2004). *The Cauchy–Schwarz master class: an introduction to the art of mathematical inequalities*. Cambridge: Cambridge University Press.

Stetter, H. (1973). *Analysis of discretization methods for ordinary differential equations*. New York: Springer.

Stetter, H. J. (1999). The nearest polynomial with a given zero, and similar problems. *SIGSAM Bulletin, 33*, 2–4.

Stetter, H. J. (2004). *Numerical polynomial algebra*. Philadelphia: SIAM.

Stewart, G. (1998). *Afternotes goes to graduate school*. Philadelphia: SIAM.

Stewart, G., & Sun, J.-g. (1990). *Matrix perturbation theory*, vol. 175. Boston: Academic Press.

Stewart, G. W. (1985). A note on complex division. *ACM Transactions on Mathematical Software, 11*(3), 238–241.

Storjohann, A. (2005). The shifted number system for fast linear algebra on integer matrices. *Journal of Complexity, 21*(4), 609–650.

Strang, G. (1986). *Introduction to applied mathematics*. Wellesley, MA: Wellesley-Cambridge Press.

Strang, G. (1991). A chaotic search for *i*. *The College Mathematics Journal, 22*(1), 3–12.

Strang, G. (2002). Too much calculus. *SIAM Linear Algebra Activity Group, Newsletter*.

Strang, G. (2006). *Linear algebra and its applications* (4th ed.). Cengage, Stamford, CT.

Strang, G., & Fix, G. (1973). *Analysis of the finite element method*. Upper Saddle River, NJ: Prentice-Hall.

Stuart, A. M., & Humphries, A. R. (1995). The essential stability of local error control for dynamical systems. *SIAM Journal on Numerical Analysis, 32*(6), 1940–1971.

Suhartanto, H., & Enright, W. (1992). Detecting and locating a singular point in the numerical solution of IVPs for ODEs. *Computing, 48*(2), 161–175.

Takahasi, H., & Mori, M. (1974). Double exponential formulas for numerical integration. *RIMS, Kyoto University*, pp. 721–741.

Tisseur, F., & Higham, N. J. (2001). Structured pseudospectra for polynomial eigenvalue problems, with applications. *SIAM Journal on Matrix Analysis and Applications, 23*(1), 187–208.

Toh, K.-C., & Trefethen, L. N. (1994). Pseudozeros of polynomials and pseudospectra of companion matrices. *Numerische Mathematik, 68*, 403–425.

Tourigny, Y. (1996). The numerical analysis of spontaneous singularities in nonlinear evolution equations. *Numerical analysis: AR Mitchell 75th birthday volume*, pp. 355.

Trefethen, L. N. (1992). The definition of numerical analysis. *SIAM News, 25*, 6 and 22.

Trefethen, L. N. (2000). *Spectral methods in MATLAB*. Philadelphia: SIAM.

Trefethen, L. N. (2008a). Is Gauss quadrature better than Clenshaw–Curtis? *SIAM Review, 50*, 67–87.

Trefethen, L. N. (2008b). Numerical analysis. *The Princeton companion to mathematics*. Princeton, NJ: Princeton University Press.

Trefethen, L. (2010). Householder triangularization of a quasimatrix. *IMA Journal of Numerical Analysis, 30*(4), 887.

Trefethen, L. N. (2013). *Approximation theory and approximation practice*. Philadelphia: SIAM.

Trefethen, L. N., & Bau, D. (1997). *Numerical linear algebra*. Philadelphia: SIAM.

Trefethen, L., & Embree, M. (2005). *Spectra and pseudospectra: the behavior of nonnormal matrices and operators*. Princeton, NJ: Princeton University Press.

Trefethen, L. N., & Schreiber, R. S. (1990). Average-case stability of Gaussian elimination. *SIMAX, 11*(3), 335–360.

Tsai, Y.-F., & Farouki, R. T. (2001). Algorithm 812: BPOLY: an object-oriented library of numerical algorithms for polynomials in Bernstein form. *ACM Transactions on Mathematical Software, 27*, 267–296.

Tucker, W. (2002). A rigorous ODE solver and Smale's 14th problem. *Foundations of Computational Mathematics, 2*(1), 53–117.

Turing, A. M. (1952). The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences, 237*(641), 37–72.

van der Sluis, A. (1969). Condition numbers and equilibration of matrices. *Numerische Mathematik, 14*(1), 14–23.

van Deun, J., Deckers, K., Bultheel, A., & Weideman, J. (2008). Algorithm 882: Near-best fixed pole rational interpolation with applications in spectral methods. *ACM Transactions on Mathematical Software, 35*(2), 1–21.

Van Loan, C. (1992). *Computational frameworks for the fast Fourier transform.* Philadelphia: SIAM.

Varah, J. M. (1994). Backward error estimates for Toeplitz systems. *SIAM Journal on Matrix Analysis and Applications, 15*(2), 408–417.

Von Neumann, J., & Goldstine, H. (1947). Numerical inverting of matrices of high order. *Bulletin of the American Mathematical Society, 53*(11), 1021–1099.

von zur Gathen, J., & Gerhard, J. (2003). *Modern computer algebra.* Cambridge: Cambridge University Press.

Waldvogel, J. (2011). Towards a general error theory of the trapezoidal rule. *Approximation and Computation, 42*, 267–282.

Wan, F. (1989). *Mathematical models and their analysis.* Harper & Row.

Warming, R., & Hyett, B. (1974). The modified equation approach to the stability and accuracy analysis of finite-difference methods. *Journal of Computational Physics, 14*(2), 159–179.

Watkins, D. S. (1982). Understanding the QR algorithm. *SIAM Review, 24*(4), 427–440.

Watkins, D. S. (1984). Isospectral flows. *SIAM Review, 26*(3), 379–391.

Watson, G. (1991). An algorithm for optimal ℓ_2 scaling of matrices. *IMA Journal of Numerical Analysis, 11*(4), 481–492.

Weideman, J. A. C. (2002). Numerical integration of periodic functions: a few examples. *The American Mathematical Monthly, 109*(1), 21–36.

Weideman, J. A. C. (2003). Computing the dynamics of complex singularities of nonlinear PDEs. *SIAM Journal on Applied Dynamical Systems, 2*(2), 171–186.

Weideman, J. A., & Reddy, S. C. (2000). A MATLAB differentiation matrix suite. *ACM Transactions on Mathematical Software, 26*, 465–519.

Wilf, H. S. (1962). *Mathematics for the physical sciences.* Dover, New York, NT.

Wilkinson, J. (1959a). The evaluation of the zeros of ill-conditioned polynomials. Part I. *Numerische Mathematik, 1*(1), 150–166.

Wilkinson, J. (1959b). The evaluation of the zeros of ill-conditioned polynomials. Part II. *Numerische Mathematik, 1*(1), 167–180.

Wilkinson, J. H. (1963). *Rounding errors in algebraic processes.* Prentice-Hall Series in Automatic Computation. Englewood Cliffs, NJ: Prentice-Hall.

Wilkinson, J. H. (1971). Modern error analysis. *SIAM Review, 13*(4), 548–568.

Wilkinson, J. H. (1984). The perfidious polynomial. In: G. H. Golub, (Ed.), *Studies in numerical analysis*, vol. 24, pp. 1–28. Mathematical Assosication of America, Washington, DC.

Wright, E. M. (1947). The linear difference-differential equation with constant coefficients. *Proceedings of the Royal Society of Edinburgh A, LXII*, 387–393.

Wu, W., Wang, F., & Chang, M. (2010). Sensitivity analysis of dynamic biological systems with time-delays. *BMC Bioinformatics, 11*(Suppl 7), S12.

Zeng, Z. (2004). Algorithm 835: Multroot—a MATLAB package for computing polynomial roots and multiplicities. *ACM Transactions on Mathematical Software, 30*(2), 218–236.

Zhang, F., (Ed.) (2005). *The Schur complement and its applications*, vol. 4 of *Numerical methods and algorithms.* New York: Springer.

Zhao, J., Davison, M., & Corless, R. M. (2007). Compact finite difference method for American option pricing. *Journal of Computational and Applied Mathematics, 206*, 306–321.

Zhi, L. (2003). Displacement structure in computing approximate GCD of univariate polynomials. In: *Proc. Sixth Asian Symposium on Computer Mathematics (ASCM 2003)*, vol. 10, pp. 288–298.

Zienkiewicz, O., Taylor, R., & Zhu, J. (2000). *The finite element method, vol. 1, 2, 3*. Oxford: Butterworth-Heinemann.

Index

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0,