21 Bessel functions

EACH CYCLE OF A RECURSIVE PROCESS NOT ONLY GENERATES ITS OWN ROUNDING ERRORS, BUT ALSO INHERITS THE ROUNDING ERRORS COMMITTED IN ALL PREVIOUS CYCLES. IF CONDITIONS ARE UNFAVORABLE, THE RESULTING PROPAGATION OF ERRORS MAY WELL BE DISASTROUS.

— WALTER GAUTSCHI
Computational Aspects of Three-Term Recurrence Relations (1967).

A large family of functions known as Bessel[1] functions is treated in four chapters of the *Handbook of Mathematical Functions* [AS64, Chapters 9–12], with more coverage than any other named functions in that famous compendium. Although those functions were first discovered by Daniel Bernoulli (1700–1782), who in 1732 worked with the order-zero function that is now known as $J_0(x)$, it was Friedrich Wilhelm Bessel who generalized them about 1824 and brought them to mathematical prominence, and they bear his name, instead of that of Bernoulli. Leonhard Euler (1707–1783) discussed their generalizations to arbitrary integer orders, $J_n(x)$, in 1764. The definitive textbook treatment in English, first published in 1922, is *A Treatise on the Theory of Bessel Functions* [Wat95], a revised reprint of the 1944 second edition.

The Bessel functions arise in several areas of astronomy, engineering, mathematics, and physics as solutions of certain equations containing unknown functions and their derivatives. Such equations are known as *differential equations*, a topic that is outside the scope of this book. The Bessel functions appear as analytic solutions of those equations, particularly in problems with cylindrical and spherical geometries, and older literature commonly refers to them as cylinder and sphere functions. Many textbooks on mathematical physics describe applications of Bessel functions to topics such as diffraction of light, electromagnetic potentials, planetary motion, radiation from moving charges, scattering of electromagnetic waves, and solutions of the Helmholtz equation, the Laplace equation and the Poisson equation [AW05, AWH13, Jac75, MH80, MF53, PP05].

Computation of Bessel functions is treated in several books on special functions, including [GDT$^+$05, Chapter 7], [GST07, Chapters 7 and 12], [HCL$^+$68, Section 6.8], [Luk69a], [Luk77, Chapters 17–19], [Mos89, Chapter 6], [Olv74, Chapters 2 and 7], [Tho97, Chapters 14–15], and [ZJ96, Chapters 5–11]. Software algorithms for the Bessel function family are also described in numerous research articles [ADW77a, ADW77b, Amo86, BS92, Cai11, Cam80, CMF77, Cod80, Cod83, CS89, CS91, Cod93b, Gau64, GST02, GST04, HF09, Har09b, Hil77, Hil81, Jab94, JL12, Kod07, Kod08, Kra14, Sko75, VC06, Wie99]. The journals *Mathematics of Computation* and *SIAM Journal on Mathematical Analysis* contain dozens of articles on the computation of Bessel functions, and their derivatives, integrals, products, sums, and zeros, although without software.[2] A search of the journals *Computer Physics Communications* and *Journal of Computational Physics* finds almost 50 articles on the computation of Bessel functions, and many more on their applications. The *MathSciNet* database has entries for about 3000 articles with *Bessel* in their titles, and the *zbMATH* database lists more than 8000 such articles.

Some of the Bessel functions that we treat in this chapter are special cases of more general functions, the *Coulomb wave functions* (see [AS64, Chapter 14] and [OLBC10, Chapter 33]), and some publications in the cited physics journals take that approach. However, it seems unlikely to this author that a three-parameter function that is even more difficult computationally can provide a satisfactory route to the two-parameter Bessel functions for the argument ranges and accuracy required for the mathcw library.

Bessel functions are normally defined in terms of real orders, ν (sometimes restricted to nonnegative values), and complex arguments, z, although complex orders are possible [Kod07]. However, this book deals mostly with the computation of functions of real arguments. Among all the functions treated in this book, the Bessel functions

[1]Friedrich Wilhelm Bessel (1784–1846) was a German astronomer and mathematician. Bessel achieved fame in astronomy, cataloging the positions of more than 50,000 stars, and was the first to use parallax to calculate the distance of stars from the Earth. He was appointed director of the Königsberg Observatory at the age of 26 by the King of Prussia. A large crater on the Moon, and Asteroid 1552 Bessel, are named after him.

[2]Extensive bibliographies of the contents of those journals are available at links from `http://www.math.utah.edu/pub/tex/bib/index-table.html`.

© Springer International Publishing AG 2017
N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_21

are the most difficult to determine accurately, and their accurate computation for complex arguments is *much harder* than for real arguments. In this chapter, we adopt the convention that mathematical formulas are usually given for real ν and complex z, but when we discuss computation, we consider only integer orders n and real arguments x. Nevertheless, keep in mind that many applications of Bessel functions, particularly in physics problems, require real orders and complex arguments, and sometimes, the orders are large, and the magnitudes of the real and imaginary parts differ dramatically. Our implementations of several of the Bessel functions do not address those needs. Indeed, a completely satisfactory software treatment of those functions in freely available and highly portable software remains elusive. The *GNU Scientific Library* [GDT+05] handles the case of real orders, but at the time of writing this, is restricted to real arguments.

21.1 Cylindrical Bessel functions

There are about two dozen members of the Bessel function family, as shown in **Table 21.1** on the next page. None of them is mentioned in the ISO C Standards, but implementations of the C library on various flavors of UNIX since the mid-1970s have included functions for computing $J_0(x)$, $J_1(x)$, and $J_n(x)$, as well as $Y_0(x)$, $Y_1(x)$, and $Y_n(x)$, but only for integer orders and real arguments. The Fortran 2008 Standard [FTN10] introduces them to that language with names like `bessel_j0(x)`. The POSIX Standards require those six functions, but confusingly, their software names are spelled in lowercase, despite the fact that they compute the *cylindrical*, rather than the *spherical*, Bessel functions. Their prototypes look like this:

```
double j0 (double x);                 double y0 (double x);
double j1 (double x);                 double y1 (double x);
double jn (int n, double x);          double yn (int n, double x);
```

There are companions for other floating-point types with the usual type suffixes. Particularly in the physics literature, the function of the second kind is commonly referred to as the Neumann function, $N_\nu(z)$, but it is identical to $Y_\nu(z)$. For real arguments, the functions of the first and second kinds have real values, but the functions of the third kind have complex values.

For the `mathcw` library, and this book, we implement the six Bessel functions required by POSIX, in each supported floating-point precision. We also supply the modified cylindrical Bessel companions, and the spherical functions that correspond to each of the supported cylindrical functions. To augment those scalar functions, we provide a family that returns in an array argument the values of sequences of Bessel functions with a single argument x and orders $k = 0, 1, 2, \ldots, n$. Such sequences are needed in series expansions with terms involving Bessel functions. The sequence values may be computable more economically than by separate invocations of the functions for specific orders and arguments.

The mathematical functions and software routines for the ordinary Bessel functions are related as follows:

$$J_0(x) \equiv \texttt{j0(x)} = \texttt{jn(0,x)}, \qquad J_1(x) \equiv \texttt{j1(x)} = \texttt{jn(1,x)},$$
$$Y_0(x) \equiv \texttt{y0(x)} = \texttt{yn(0,x)}, \qquad Y_1(x) \equiv \texttt{y1(x)} = \texttt{yn(1,x)}.$$

For mathematical reasons that are discussed later when we develop computer algorithms, implementations of order-n Bessel functions are almost certain to invoke the functions of a single argument directly for $n = 0$ and $n = 1$, rather than providing independent computational routes to those two particular functions.

Symbolic-algebra systems include many of the Bessel functions listed in **Table 21.1** on the facing page. The functions of interest in the first part of this chapter, $J_\nu(z)$ and $Y_\nu(z)$, are called

- `BesselJ(nu,z)` and `BesselY(nu,z)` in Maple and REDUCE,
- `BesselJ[nu,z]` and `BesselY[nu,z]` in Mathematica,
- `bessel_j(nu,z)` and `bessel_y(nu,z)` in Maxima,
- `besselJ(nu,z)` and `besselY(nu,z)` in Axiom and MuPAD, and
- `besselj(nu,z)` and `besseln(nu,z)` in PARI/GP.

Those systems permit `nu` to be real, and `z` to be complex, rather than restricting them to integer and real values, respectively.

Table 21.1: The Bessel function family. The subscripts are called the *order* of the function, and except as noted, may be positive or negative. The order ν is any real number, the order n is any integer, and the order k is any nonnegative integer. The argument z is any complex number, and for the Kelvin functions, the argument x is any nonnegative real number.
Some authors call the functions of the first kind the *regular* functions, and those of the second kind, *irregular* functions.

Function	Description
$J_\nu(z)$	ordinary Bessel function of the first kind
$Y_\nu(z)$	ordinary Bessel function of the second kind, sometimes called the Neumann function or Weber's function
$H_\nu(z)$	ordinary Bessel function of the third kind, also known as the Hankel function
$I_\nu(z)$	modified (or hyperbolic) Bessel function of the first kind
$K_\nu(z)$	modified (or hyperbolic) Bessel function of the second kind, also called the Basset function and the Macdonald function
$N_\nu(z)$	Neumann function, identical to $Y_\nu(z)$
$Z_\nu(z)$	arbitrary ordinary Bessel function of first, second, or third kinds,
$j_n(z)$	spherical Bessel function of the first kind, equal to $\sqrt{\pi/(2z)}J_{n+1/2}(z)$
$y_n(z)$	spherical Bessel function of the second kind, equal to $\sqrt{\pi/(2z)}Y_{n+1/2}(z)$
$h_n(z)$	spherical Bessel function of the third kind
$i_n(z)$	modified spherical Bessel function of the first kind, equal to $\sqrt{\pi/(2z)}I_{n+1/2}(z)$
$k_n(z)$	modified spherical Bessel function of the second kind, equal to $\sqrt{\pi/(2z)}K_{n+1/2}(z)$
$n_n(z)$	spherical Neumann function, identical to $y_n(z)$
$S_n(z)$	Riccati–Bessel function of the first kind, equal to $zj_n(z)$
$C_n(z)$	Riccati–Bessel function of the second kind, equal to $-zy_n(z)$
$\zeta_n(z)$	Riccati–Bessel function of the third kind, equal to $zh_n(z)$
$\mathrm{ber}_k(x)$	Kelvin (or Thomson) function of the first kind
$\mathrm{bei}_k(x)$	Kelvin (or Thomson) function of the first kind
$\mathrm{ker}_k(x)$	Kelvin (or Thomson) function of the second kind
$\mathrm{kei}_k(x)$	Kelvin (or Thomson) function of the second kind
$\mathrm{Ai}(z)$	Airy function, equal to $(\sqrt{z}/3)(I_{-1/3}(\xi) - I_{+1/3}(\xi))$, where ξ is the Greek letter *xi*, and also equal to $(1/\pi)\sqrt{z/3}K_{+1/3}(\xi)$, where $\xi = (2/3)z^{3/2}$
$\mathrm{Bi}(z)$	Airy function, equal to $\sqrt{z/3}(I_{-1/3}(\xi) + I_{+1/3}(\xi))$
$\mathbf{H}_\nu(z)$	ordinary Struve function
$\mathbf{L}_\nu(z)$	modified Struve function
$\mathbf{J}_\nu(z)$	Anger's function
$\mathbf{E}_\nu(z)$	Weber's function

21.2 Behavior of $J_n(x)$ and $Y_n(x)$

A few of the ordinary Bessel functions of the first and second kinds are graphed in Figure 21.1 on the next page. From the plots, and selected numerical evaluations, we can make some important observations that are relevant to their computation:

- The functions of the first kind, $J_n(x)$, look like damped cosine ($n = 0$) and sine ($n > 0$) waves, but their zeros are not equally spaced. Instead, the zeros appear to get closer together as x increases, and differ for each value of n. That means that argument reductions like those that we found for the trigonometric functions are not applicable.
- Values of $J_n(x)$ lie in $[-1, +1]$, so intermediate overflow is unlikely to be a problem in their computation.
- The x position of the first positive maximum of $J_n(x)$ and $Y_n(x)$, and their first positive nonzero root, increase as n gets larger.
- For $n > 0$, $J_n(10^{-p}) \approx \mathcal{O}(10^{-np})$, so $J_1(x)$ is representable for tiny x, although the higher-order functions underflow to zero.
- The functions of the second kind, $Y_n(x)$, have single poles at $x = 0$ for all n, and their zeros appear to get closer together as x increases.
- The approach to the single pole in $Y_n(x)$ slows as n increases.

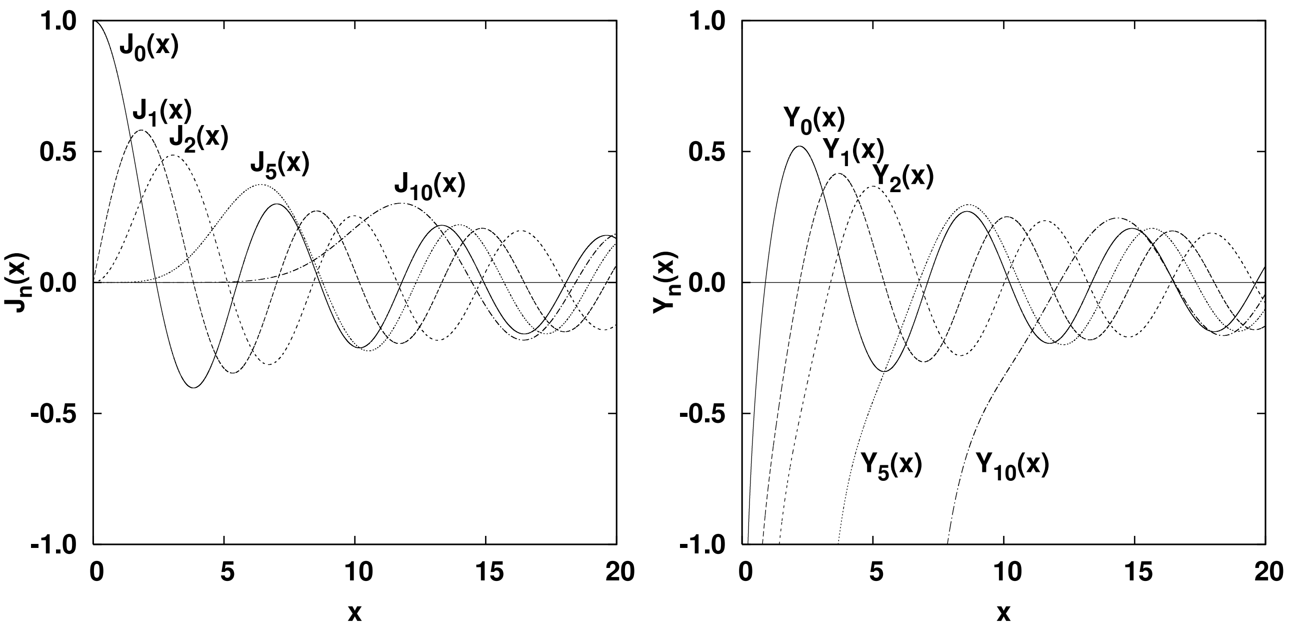


Figure 21.1: Ordinary Bessel functions of the first and second kinds, $J_n(x)$ and $Y_n(x)$. The solid lines, and highest positive maxima, correspond to $n = 0$.
The functions of the first kind can be extended to the negative axis through the relation $J_n(-x) = (-1)^n J_n(x)$, so even orders are symmetric about the origin, and odd orders are antisymmetric.
The functions of the second kind have real values only for $x \geq 0$; they have complex values when x is negative. For integer orders, they all go to $-\infty$ as x approaches zero.

- Values of $Y_n(x)$ lie in $(-\infty, 0.53]$, so overflow must be anticipated and handled for $x \approx 0$. Numerical evaluation with x set to the smallest representable floating-point number shows that $Y_0(x)$ is of modest size, but overflow occurs in $Y_n(x)$ for all $n > 0$.

- Although both $J_n(x)$ and $Y_n(x)$ decay as x increases, the attrition is not rapid. For example, we have $J_0(10^6) \approx 0.000\,331$, $Y_0(10^6) \approx -0.000\,726$, $J_0(10^{600}) \approx 0.449 \times 10^{-300}$, and $Y_0(10^{600}) \approx 0.659 \times 10^{-300}$. Thus, in floating-point arithmetic, the maxima of those functions *cannot underflow* for any representable x. Some deficient software implementations of those functions, such as those in the GNU/LINUX run-time libraries, suffer premature underflow over most of the range of x, or produce spurious NaN results. Here is an example from that operating system on an AMD64 platform when hoc is linked with the native math library:

```
hoc64> for (x = 1.0e7; x <= 1.0e18; x *= 10) \
hoc64>     printf("%.2g  % g\n", x, J0(x))
1e+07  -8.68373e-05
1e+08   3.20603e-05
1e+09  -qnan(0x31)
1e+10  -qnan(0x33)
1e+11  -qnan(0x35)
1e+12  -qnan(0x37)
1e+13  -qnan(0x39)
1e+14  -qnan(0x3b)
1e+15  -qnan(0x3d)
1e+16  -qnan(0x3f)
1e+17   0
1e+18   0
```

Native library improvements on that system removed some of those irregularities as this book was nearing completion. However, tests on more than a dozen flavors of UNIX found that only MAC OS X, OSF/1, and SOLARIS IA-32 (but not SPARC) produced expected results over most of the entire floating-point range, and even those lost all significant digits for large arguments. The conclusion is that implementations of the Bessel functions in UNIX libraries may be neither reliable nor robust.

- The oscillatory nature of the Bessel functions suggests that recurrence formulas are likely to suffer subtraction loss for certain ranges of x.
- When x is large, there is insufficient precision in a floating-point representation to resolve the waves of the functions, because consecutive floating-point numbers eventually bracket multiple wave cycles. The best that we can hope for then is to get the correct order of magnitude of the waves.

In the next few sections, we investigate those graphical observations further, and make them more precise.

21.3 Properties of $J_n(z)$ and $Y_n(z)$

The ordinary Bessel functions of the first and second kinds have these limiting values, where ν is any real number, z is a complex number, and $e = \exp(1)$:

$$\begin{aligned}
J_\nu(z) &\approx (z/2)^\nu/\Gamma(\nu+1), && z \to 0, \nu \neq -1, -2, -3, \ldots, \\
Y_0(z) &\approx (2/\pi)\log(z), && z \to 0, \\
Y_\nu(z) &\approx -(1/\pi)\Gamma(\nu)(2/z)^\nu && z \to 0, \nu > 0, \\
J_\nu(x) &\to \sqrt{2/(\pi x)}\cos(x - \nu\pi/2 - \pi/4), && x \to +\infty, x \gg \nu, \nu \geq 0, \\
Y_\nu(x) &\to \sqrt{2/(\pi x)}\sin(x - \nu\pi/2 - \pi/4), && x \to +\infty, x \gg \nu, \nu \geq 0, \\
J_\nu(x) &\to (1/\sqrt{2\pi\nu})(ex/(2\nu))^\nu, && \nu \to +\infty, \nu \gg x, \\
Y_\nu(x) &\to -(1/\sqrt{2\pi\nu})(ex/(2\nu))^{-\nu}, && \nu \to +\infty, \nu \gg x, \\
J_0(0) &= 1, \qquad Y_0(0) = -\infty, && \\
J_n(0) &= 0, \qquad Y_n(0) = -\infty, && n > 0, \\
J_n(\infty) &= 0, \qquad Y_n(\infty) = 0. &&
\end{aligned}$$

The large-argument limits of $J_\nu(x)$ and $Y_\nu(x)$ answer the question about the spacing of the roots: they are ultimately separated by π, rather than squeezing ever closer together, as might be suggested by the function graphs for small x. Table 21.2 on the following page shows a few of the roots, easily found in Maple with calls to `BesselJZeros(nu,k)` and `BesselYZeros(nu,k)`. For $k \gg \nu$, higher roots of $J_\nu(r_{\nu,k}) = 0$ and $Y_\nu(s_{\nu,k}) = 0$ can be estimated to about three correct figures by the formulas

$$r_{\nu,k} \approx (k + \nu/2 - 1/4)\pi, \qquad s_{\nu,k} \approx (k + \nu/2 - 3/4)\pi.$$

In particular, that relation shows that the roots of the Bessel functions of orders $\nu - 1, \nu, \nu + 1, \ldots$ are well separated, a fact that has significance later in their evaluation by recurrence relations.

The functions have these symmetry relations:

$$J_n(-x) = (-1)^n J_n(x), \qquad J_{-n}(x) = (-1)^n J_n(x), \qquad Y_{-n}(x) = (-1)^n Y_n(x).$$

They allow the computation for real arguments to be done for $n \geq 0$ and $x \geq 0$, followed by a negation of the computed result when n is negative and odd.

For $|x| \gg |\nu|$, the two Bessel functions have asymptotic expansions as linear combinations of cosines and sines, like this:

$$\begin{aligned}
\theta &= x - (\nu/2 + 1/4)\pi, \\
J_\nu(x) &\asymp \sqrt{2/(\pi x)}\,(\mathrm{P}(\nu, x)\cos(\theta) - \mathrm{Q}(\nu, x)\sin(\theta)), \\
Y_\nu(x) &\asymp \sqrt{2/(\pi x)}\,(\mathrm{Q}(\nu, x)\cos(\theta) + \mathrm{P}(\nu, x)\sin(\theta)).
\end{aligned}$$

Table 21.2: Approximate roots of ordinary Bessel functions, $J_\nu(r_{\nu,k}) = 0$ and $Y_\nu(s_{\nu,k}) = 0$.

	k								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	...
$r_{0,k}$	2.405	5.520	8.654	11.792	14.931	18.071	21.212	24.352	...
$r_{1/2,k}$	3.142	6.283	9.425	12.566	15.708	18.850	21.991	25.133	...
$r_{1,k}$	3.832	7.016	10.173	13.324	16.471	19.616	22.760	25.904	...
$r_{3/2,k}$	4.493	7.725	10.904	14.066	17.221	20.371	23.519	26.666	...
$r_{2,k}$	5.136	8.417	11.620	14.796	17.960	21.117	24.270	27.421	...
$r_{10,k}$	14.476	18.433	22.047	25.509	28.887	32.212	35.500	38.762	...
$r_{100,k}$	108.836	115.739	121.575	126.871	131.824	136.536	141.066	145.453	...
$s_{0,k}$	0.894	3.958	7.086	10.222	13.361	16.501	19.641	22.782	...
$s_{1/2,k}$	1.571	4.712	7.854	10.996	14.137	17.279	20.420	23.562	...
$s_{1,k}$	2.197	5.430	8.596	11.749	14.897	18.043	21.188	24.332	...
$s_{3/2,k}$	2.798	6.121	9.318	12.486	15.644	18.796	21.946	25.093	...
$s_{2,k}$	3.384	6.794	10.023	13.210	16.379	19.539	22.694	25.846	...
$s_{10,k}$	12.129	16.522	20.266	23.792	27.207	30.555	33.860	37.134	...
$s_{100,k}$	104.380	112.486	118.745	124.275	129.382	134.206	138.821	143.275	...

The right-hand sides of $J_\nu(x)$ and $Y_\nu(x)$ are product-sum expressions of the form $ab + cd$ that may be subject to subtraction loss. Our `PPROSUM()` function family (see Section 13.24 on page 386) can evaluate them accurately.

$\mathrm{P}(\nu, x)$ and $\mathrm{Q}(\nu, x)$ are auxiliary functions defined by

$$\begin{aligned}
\mu &= 4\nu^2, \\
\mathrm{P}(\nu, x) &\asymp 1 - \frac{(\mu - 1^2)(\mu - 3^2)}{2!\,(8x)^2} + \frac{(\mu - 1^2)(\mu - 3^2)(\mu - 5^2)(\mu - 7^2)}{4!\,(8x)^4} - \cdots, \\
\mathrm{Q}(\nu, x) &\asymp \frac{\mu - 1^2}{8x} - \frac{(\mu - 1^2)(\mu - 3^2)(\mu - 5^2)}{3!\,(8x)^3} + \\
&\quad \frac{(\mu - 1^2)(\mu - 3^2)(\mu - 5^2)(\mu - 7^2)(\mu - 9^2)}{5!\,(8x)^5} - \cdots.
\end{aligned}$$

Those series look complicated, but their terms can easily be generated by recurrence relations:

$$\begin{aligned}
\mathrm{P}(\nu, x) &\asymp p_0 + p_1 + p_2 + \cdots, \\
p_0 &= 1, \\
p_k &= -\frac{(\mu - (4k-3)^2)(\mu - (4k-1)^2)}{2k(2k-1)} \times \frac{1}{(8x)^2} \times p_{k-1}, && k = 1, 2, 3, \ldots, \\
\mathrm{Q}(\nu, x) &\asymp q_0 + q_1 + q_2 + \cdots, \\
q_0 &= \frac{\mu - 1}{8x}, \\
q_k &= -\frac{(\mu - (4k-1)^2)(\mu - (4k+1)^2)}{2k(2k+1)} \times \frac{1}{(8x)^2} \times q_{k-1}, && k = 1, 2, 3, \ldots.
\end{aligned}$$

As we describe in Section 2.9 on page 19, asymptotic expansions are not convergent, but they can be summed as long as term magnitudes decrease. The error in the computed function is then roughly the size of the omitted next term in the sum, and that size then determines the available accuracy. For large order ν, the factor in the formulas for the terms p_k and q_k is roughly $\nu^4/(16k^2x^2)$, so as long as $\nu^2 < x$, convergence to arbitrary machine precision is rapid, with each term contributing at least four additional bits to the precision of the sum.

Several published algorithms for $J_\nu(x)$ and $Y_\nu(x)$ exploit the forms of the asymptotic expansions for $\mathrm{P}(\nu, x)$ and $\mathrm{Q}(\nu, x)$ by using instead polynomial approximations in the variable $t = 1/x^2$. To compute such approximations, we first need closed forms for $\mathrm{P}(\nu, x)$ and $\mathrm{Q}(\nu, x)$. We can find $\mathrm{P}(\nu, x)$ by multiplying the two equations involving the Bessel functions by $\cos(\theta)$ and $\sin(\theta)$, respectively, adding the equations, and simplifying. The other function is

Table 21.3: Trigonometric formulas needed for the asymptotic formulas for the ordinary Bessel functions $J_n(x)$ and $Y_n(x)$, with $\theta = x - (n/2 + 1/4)\pi$, and $n \geq 0$. See the text for accurate computation of the sums and differences in these formulas.

n mod 4	**cos(θ)**	**sin(θ)**
0	$+\sqrt{\frac{1}{2}}(\cos(x) + \sin(x))$	$-\sqrt{\frac{1}{2}}(\cos(x) - \sin(x))$
1	$-\sqrt{\frac{1}{2}}(\cos(x) - \sin(x))$	$-\sqrt{\frac{1}{2}}(\cos(x) + \sin(x))$
2	$-\sqrt{\frac{1}{2}}(\cos(x) + \sin(x))$	$+\sqrt{\frac{1}{2}}(\cos(x) - \sin(x))$
3	$+\sqrt{\frac{1}{2}}(\cos(x) - \sin(x))$	$+\sqrt{\frac{1}{2}}(\cos(x) + \sin(x))$

found by swapping the trigonometric multipliers, and subtracting. We then have these results:

$$\begin{aligned}
P(\nu, x) &= \sqrt{\pi x/2}\,(J_\nu(x)\cos(\theta) + Y_\nu(x)\sin(\theta)),\\
Q(\nu, x) &= \sqrt{\pi x/2}\,(Y_\nu(x)\cos(\theta) - J_\nu(x)\sin(\theta)).
\end{aligned}$$

Once the replacements for the asymptotic series have been evaluated, along with the two trigonometric functions, both Bessel functions can be produced with little additional cost, and some software packages do just that. Subtraction loss in the cosines and sines of shifted arguments can be prevented by using the double-angle formula for the cosine, and then solving for the problematic sums and differences:

$$\begin{aligned}
\cos(2\theta) &= (\cos(\theta))^2 - (\sin(\theta))^2\\
&= (\cos(\theta) - \sin(\theta)) \times (\cos(\theta) + \sin(\theta)),\\
\cos(\theta) - \sin(\theta) &= \cos(2\theta)/(\cos(\theta) + \sin(\theta)),\\
\cos(\theta) + \sin(\theta) &= \cos(2\theta)/(\cos(\theta) - \sin(\theta)).
\end{aligned}$$

When $\nu = 0$, we require these functions:

$$\begin{aligned}
\cos(\theta) &= \cos(x - (1/4)\pi)\\
&= \sqrt{\tfrac{1}{2}}\,(\cos(x) + \sin(x))\\
&= \sqrt{\tfrac{1}{2}}\,\frac{\cos(2x)}{\cos(x) - \sin(x)},\\
\sin(\theta) &= \sin(x - (1/4)\pi)\\
&= -\sqrt{\tfrac{1}{2}}\,(\cos(x) - \sin(x))\\
&= -\sqrt{\tfrac{1}{2}}\,\frac{\cos(2x)}{\cos(x) + \sin(x)}.
\end{aligned}$$

For each of them, we use whichever of the second or third formulas that does not involve a subtraction.

When $\nu = 1$, we get a reduction to the two cases just treated:

$$\begin{aligned}
\cos(\theta) &= \cos(x - (3/4)\pi)\\
&= \sin(x - (1/4)\pi),\\
\sin(\theta) &= \sin(x - (3/4)\pi)\\
&= -\cos(x - (1/4)\pi).
\end{aligned}$$

For the general case of nonnegative integer orders, there are just four possibilities, summarized in Table 21.3.

Although our formulas for trigonometric sums and differences may require $\cos(2x)$, we do not normally compute it by invoking the cosine function directly, for these reasons:

- The argument $2x$ overflows when x is near the overflow limit.

- The cosine computation is comparatively expensive when x is large.
- The argument $2x$ may be inexact for floating-point bases other than two, and if x is large, the computed $\cos(2x)$ would then be wildly incorrect.

Instead, we use well-known trigonometric relations to compute it accurately from quantities that we already have:

$$\cos(2x) = \begin{cases} 2(\cos(x))^2 - 1, & \textit{if } |\cos(x)| \le 1/2, \\ 1 - 2(\sin(x))^2, & \textit{if } |\sin(x)| \le 1/2, \\ (1-\tan(x)^2)/(1+\tan(x)^2), & \textit{if } \tan(x)^2 \textit{ is outside } (\tfrac{1}{2}, \tfrac{3}{2}), \\ -2\,\mathrm{fma}(\sin(x), \sin(x), -\tfrac{1}{2}), & \textit{if } |\sin(x)| \le |\cos(x)|, \\ 2\,\mathrm{fma}(\cos(x), \cos(x), -\tfrac{1}{2}), & \textit{otherwise.} \end{cases}$$

The first two cases each cover about a third of the period of the cosine. The third case computes $\tan(x) = \sin(x)/\cos(x)$ instead of invoking the tangent function, and covers about a sixth of the period. Accuracy loss happens in the remaining sixth of the period, but it can be reduced by use of the fused multiply-add function call. However, if $2x$ is finite and exact, as it is when $\beta = 2$, and sometimes is when $\beta \neq 2$, then we call the cosine function in preference to the fused multiply-add function. There is a tradeoff between the last two alternatives, and in practice, we also call the cosine function when $2x$ is not exact, but x is less than some cutoff, set at 200 in our code.

When the host arithmetic has wobbling precision, more care is needed in the application of the tangent formula. We use it only when $\tan(x)^2 < 1/2$, and compute it as $\frac{1}{2}(1 - \tan(x)^2)/(\frac{1}{2} + \frac{1}{2}\tan(x)^2)$, to avoid leading zero bits in the numerator and denominator. The value of $\tan(x)$ lies in $[\sqrt{\frac{1}{3}}, \sqrt{\frac{1}{2}}] \approx [0.577, 0.707]$, so neither it, nor the sine and cosine from which it is computed, has leading zero bits. Were we to use the tangent formula when $\tan(x)^2 > 3/2$, numerical experiments show that leading zero bits occur about 10% of the time.

The code for the computation of $\cos(2x)$ is needed in a half-dozen Bessel-function files, so it is defined as a private function in the header file `cosdbl.h`.

The functions of the second kind have complex values for negative arguments, and the POSIX specification permits implementations to return either $-\infty$, or the negative of the largest normal number if Infinity is not supported, or else a NaN. The global value `errno` is then set to `EDOM`.

The derivatives of the Bessel functions are:

$$\begin{aligned} dJ_n(x)/dx &= nJ_n(x)/x - J_{n+1}(x), \\ dY_n(x)/dx &= nY_n(x)/x - Y_{n+1}(x). \end{aligned}$$

From the derivatives, we find these error-magnification factors (see **Section 4.1** on page 61):

$$\begin{aligned} \mathrm{errmag}(J_n(x)) &= xJ'_n(x)/J_n(x) \\ &= n - xJ_{n+1}(x)/J_n(x), \\ \mathrm{errmag}(Y_n(x)) &= n - xY_{n+1}(x)/Y_n(x). \end{aligned}$$

The error magnifications are therefore large near the zeros of the Bessel functions, and also when n or x is large.

Three-term recurrence relations relate functions of consecutive orders with fixed argument z:

$$\begin{aligned} J_{\nu+1}(z) &= (2\nu/z)J_\nu(z) - J_{\nu-1}(z), \\ Y_{\nu+1}(z) &= (2\nu/z)Y_\nu(z) - Y_{\nu-1}(z). \end{aligned}$$

When $z \gg \nu$, the first term on the right is negligible, and we have $J_{\nu+1}(z) \approx -J_{\nu-1}(z)$. A sequence of those functions for $\nu = 0, 1, 2, \ldots$ then looks like $J_0(x)$, $J_1(x)$, $-J_0(x)$, $-J_1(x)$, $J_0(x)$, $J_1(x)$, ... The same observation applies to $Y_{\nu+1}(z)$ and sequences of the functions of the second kind.

Unfortunately, the recurrence relations are frequently unstable because of subtraction loss, especially for $|x| < \nu$, as illustrated in the graphs of the ratios of the terms on the right shown in **Figure 21.2** on the next page. Whenever those ratios are in $[\frac{1}{2}, 2]$, the terms have the same sign, and similar magnitudes, and the subtraction loses one or more leading bits in binary arithmetic. That happens near the zeros of the functions on the left-hand side. Whether the accuracy loss affects values of higher-order functions depends on the relative magnitudes of the terms on the right in the next iteration. In addition, when the first terms on the right dominate, errors in the computed $J_\nu(z)$ and $Y_\nu(z)$

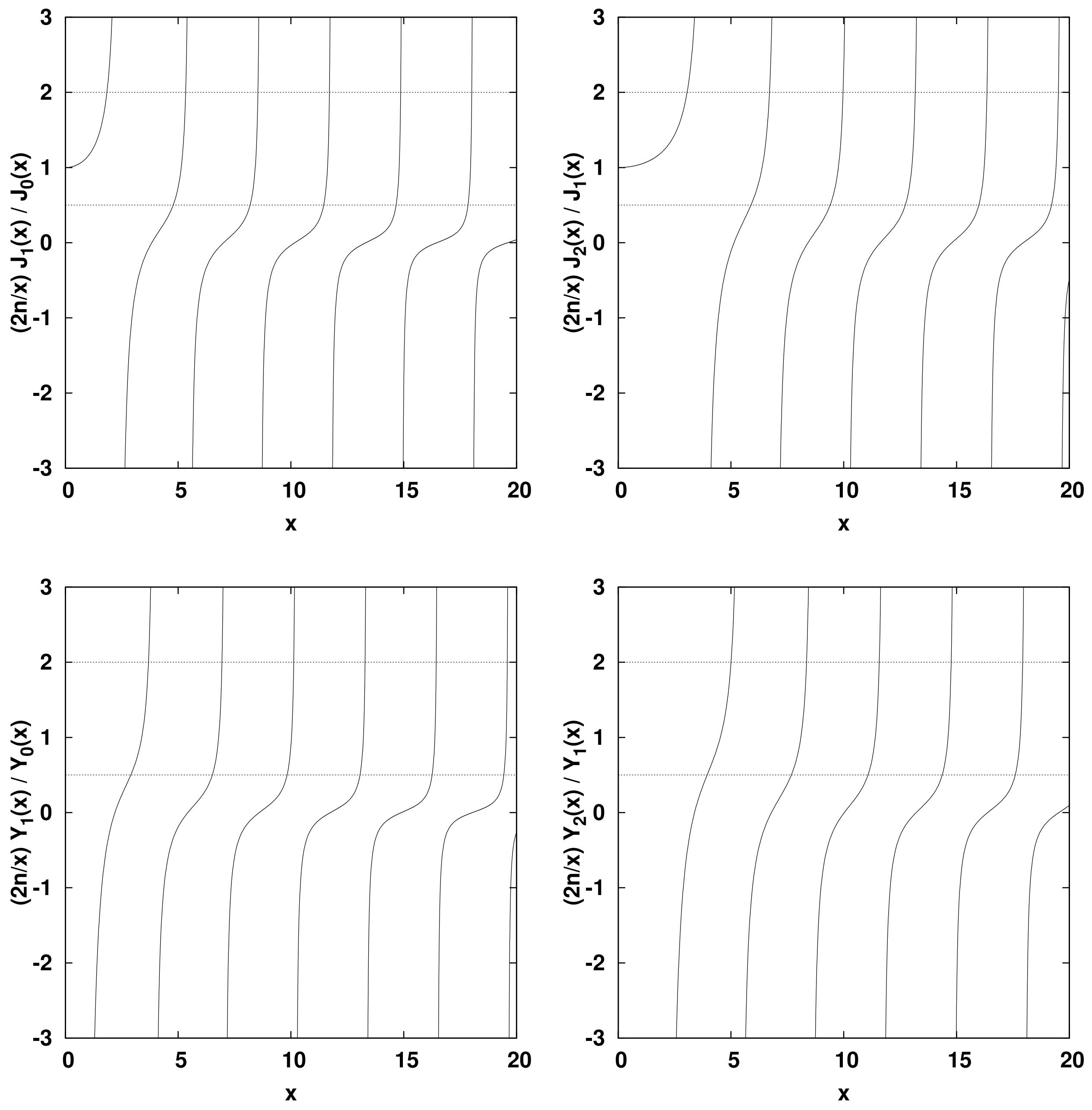


Figure 21.2: Ratios of ordinary Bessel functions of the first and second kinds, $(2n/x)J_n(x)/J_{n-1}(x)$ and $(2n/x)Y_n(x)/Y_{n-1}(x)$, for $n = 1$ and $n = 2$. As n increases, the graphs are qualitatively similar, but shifted to the right.
The recurrence formulas suffer subtraction loss in the region between the horizontal dotted lines.

are magnified by $2\nu/z$, which is large for $\nu \gg |z|$. Under those conditions, just a few iterations of the recurrence relations can easily produce results with *no significant digits* whatever. We show numerical examples of that problem in **Section 21.4** on page 705.

There is a continued fraction (see **Section 2.7** on page 12) for the ratio of two successive orders of the ordinary

Bessel function of the first kind:

$$\begin{aligned}\frac{J_\nu(z)}{J_{\nu-1}(z)} &= 0 + \frac{1}{2\nu/z -}\;\frac{1}{2(\nu+1)/z -}\;\frac{1}{2(\nu+2)/z -}\;\frac{1}{2(\nu+3)/z -}\;\cdots \\ &= 0 + \frac{(1/2)z/\nu}{1 -}\;\frac{(1/4)z^2/\nu(\nu+1)}{1 -}\;\frac{(1/4)z^2/((\nu+1)(\nu+2))}{1 -} \\ &\quad \frac{(1/4)z^2/((\nu+2)(\nu+3))}{1 -}\;\frac{(1/4)z^2/((\nu+3)(\nu+4))}{1 -}\;\cdots.\end{aligned}$$

We show later in **Section 21.6** on page 710 how that continued fraction can be used to find $J_\nu(z)$.

The ordinary Bessel functions of the first and second kinds have these relations:

$$\begin{aligned}\sin(\nu\pi)Y_\nu(z) &= \cos(\nu\pi)J_\nu(z) - J_{-\nu}(z), \\ -2\sin(\nu\pi) &= (\pi z)\big(J_{\nu+1}(z)J_{-\nu}(z) + J_\nu(z)J_{-(\nu+1)}(z)\big), \\ 2 &= (\pi z)\big(J_{\nu+1}(z)Y_\nu(z) - J_\nu(z)Y_{\nu+1}(z)\big).\end{aligned}$$

At least for small ν, those equations can be useful for testing software implementations of the functions, as long as ν and z are chosen so that there is no subtraction loss on the right-hand sides.

The Bessel functions can also be expressed as integrals, although the oscillatory nature of the integrand for large n and/or large z makes it difficult to evaluate them accurately by numerical quadrature:

$$\begin{aligned}J_0(z) &= (1/\pi)\int_0^\pi \cos(z\sin(t))\,dt, \\ &= (1/\pi)\int_0^\pi \cos(z\cos(t))\,dt, \\ J_n(z) &= (1/\pi)\int_0^\pi \cos(z\sin(t) - nt)\,dt, \\ Y_0(z) &= (4/\pi^2)\int_0^{\pi/2} \cos(z\cos(t))\big(\gamma + \log(2z(\sin(t))^2)\big)\,dt.\end{aligned}$$

Here, $\gamma \approx 0.577\cdots$ is the Euler–Mascheroni constant.

For x in $[0, \pi/2]$, the integrand for $J_0(x)$ is smooth and positive, so numerical quadrature could be attractive, and accurate. For example, a 28-point Gauss–Chebyshev quadrature recovers $J_0(\pi/4)$ to within 2 ulps of the correct value in 64-bit IEEE 754 arithmetic, a 32-point Gauss–Legendre quadrature is correct to 30 decimal digits, and a 24-point Simpson's rule quadrature produces 34 correct decimal digits. That too could be useful for software checks.

The summation formula

$$J_\nu(z) = (z/2)^\nu \sum_{k=0}^{\infty} \frac{(-(z^2/4))^k}{k!\,\Gamma(\nu+k+1)}$$

converges rapidly for $|z| < 1$, and the terms fall off sufficiently fast that there is no loss of leading digits in the subtractions of successive terms. Even with larger values of $|z|$, convergence is still reasonable, as shown in **Table 21.4** on the next page. However, for $|z| > 2$ and small n, the first few terms grow, and are larger than $J_n(z)$, so there is necessarily loss of leading digits during the summation. Higher intermediate precision, when it is available, can sometimes hide that loss.

When ν is an integer, the gamma function in the denominator reduces to a factorial, and we then have simpler summations suitable for the Bessel functions of the first kind required by POSIX:

$$\begin{aligned}J_0(z) &= 1 - (z^2/4)/(1!)^2 + (z^2/4)^2/(2!)^2 - (z^2/4)^3/(3!)^2 + \cdots, \\ J_1(z) &= (z/2)(1 - (z^2/4)/(1!\,2!) + (z^2/4)^2/(2!\,3!) - (z^2/4)^3/(3!\,4!) + \cdots), \\ J_n(z) &= (z/2)^n \sum_{k=0}^{\infty} \frac{(-(z^2/4))^k}{k!\,(k+n)!}.\end{aligned}$$

It is convenient to introduce two intermediate variables to simplify the sum for $J_n(z)$:

$$v = z/2, \qquad w = v^2, \qquad J_n(z) = v^n \sum_{k=0}^{\infty} \frac{(-w)^k}{k!\,(k+n)!}.$$

Table 21.4: Series term counts needed to reach a given accuracy for $J_n(x)$. The digit counts correspond to those of extended IEEE 754 decimal floating-point arithmetic, and are close to those of extended IEEE 754 binary arithmetic.

$x = 0.1$	Decimal digits			
n	7	16	34	70
0	3	5	9	17
1	2	5	9	16
5	2	4	8	16
10	2	4	8	15
100	2	4	7	13
1000	2	3	6	11

$x = 1$	Decimal digits			
n	7	16	34	70
0	5	9	15	27
1	5	9	15	26
5	4	8	14	25
10	4	7	13	24
100	3	5	11	20
1000	2	4	8	16

$x = 5$	Decimal digits			
n	7	16	34	70
0	11	17	27	43
1	10	16	26	42
5	9	15	24	40
10	7	13	23	38
100	4	9	16	30
1000	3	6	12	22

For example, for $n = 0$ or $n = 1$, and x in $[0, 2]$, the number of terms required in the four extended IEEE 754 decimal formats is 8, 14, 25, and 45. For fixed n and a chosen interval of x, the sum could also be replaced by a Chebyshev polynomial economization.

We can factor the sum so that the terms t_k can be computed with a simple recurrence relation:

$$J_n(z) = \frac{z^n}{2^n n!}(t_0 + t_1 + t_2 + \cdots) = \frac{v^n}{n!}(t_0 + t_1 + t_2 + \cdots),$$
$$t_0 = 1, \qquad t_k = \frac{-w}{k(k+n)} t_{k-1}, \qquad \text{for } k = 1, 2, 3, \ldots.$$

In the scale factor $v^n/n!$, the numerator can overflow or underflow, and the denominator can overflow, even though the scale factor may be representable if it were computed in exact arithmetic. In those cases, it must be computed with a logarithm and an exponential.

For integer orders only, there is a corresponding, but complicated, sum for $Y_n(z)$:

$$Y_n(z) = -\frac{1}{\pi}\Bigg[v^{-n}\sum_{k=0}^{n-1}\frac{(n-k-1)!}{k!}w^k - 2\log(v)J_n(z) + v^n\sum_{k=0}^{\infty}\big(\psi(k+1) + \psi(k+n+1)\big)\frac{(-w)^k}{k!\,(k+n)!}\Bigg].$$

That sum looks unsuitable for fast computation because of the presence of the psi functions (see Section 18.2 on page 536). Fortunately, their arguments are integers, for which they have simple forms in terms of the partial sums, h_k, of the *harmonic series* of reciprocal integers, and γ, the Euler–Mascheroni constant:

$$h_0 = 0, \qquad h_k = 1 + 1/2 + 1/3 + \cdots + 1/k, \qquad k > 0,$$
$$\psi(1) = -\gamma, \qquad \psi(k) = -\gamma + \sum_{m=1}^{k-1}\frac{1}{m} = h_{k-1} - \gamma, \qquad k > 1.$$

Substitute those values into the formula for $Y_n(z)$ to get

$$Y_n(z) = -\frac{1}{\pi}\Bigg[v^{-n}\sum_{k=0}^{n-1}\frac{(n-k-1)!}{k!}w^k - 2\log(v)J_n(z) + v^n\sum_{k=0}^{\infty}(h_k + h_{k+n} - 2\gamma)\frac{(-w)^k}{k!\,(k+n)!}\Bigg].$$

A computationally satisfactory formula for the ordinary Bessel function of the second kind results from substitution of the expansion of $J_n(z)$ in that result:

$$Y_n(z) = -\frac{1}{\pi}\left[v^{-n}\sum_{k=0}^{n-1}\frac{(n-k-1)!}{k!}w^k - 2(\log(v)+\gamma)J_n(z) + v^n\sum_{k=0}^{\infty}(h_k+h_{k+n})\frac{(-w)^k}{k!\,(k+n)!}\right].$$

From the general case, we can now easily find these formulas for the functions of orders zero and one:

$$Y_0(z) = -\frac{2}{\pi}\left[-(\log(v)+\gamma)J_0(z) + \sum_{k=1}^{\infty}h_k\frac{(-w)^k}{(k!)^2}\right],$$
$$Y_1(z) = -\frac{1}{\pi}\left[\frac{1}{v} - 2(\log(v)+\gamma)J_1(z) + v\sum_{k=0}^{\infty}(h_k+h_{k+1})\frac{(-w)^k}{k!\,(k+1)!}\right].$$

The series for $Y_0(z)$ requires about the same number of terms as that for $J_0(z)$. Convergence of the series for $Y_1(z)$ is faster than that for $Y_0(z)$, because corresponding terms are smaller by a factor of $1/(k+1)$.

The presence of the harmonic partial sums, h_k, suggests three possible computational approaches:

- Use small precomputed tables of h_k and $k!$ to compute the series sums for small z.
- Precompute the complete coefficients of w^k. Numerical experiments for x in $[0, 2]$ show that 8, 12, 20, and 32 terms suffice for computing the infinite sum in $Y_1(z)$ in the four extended IEEE 754 decimal precisions.
- Replace the infinite sums by Chebyshev economizations; they need 6, 10, 16, and 27 terms for x in $[0, 2]$ for those four decimal precisions.

With another intermediate variable, and expansions of $J_0(z)$ and $J_1(z)$, we can further simplify the formulas for $Y_0(z)$ and $Y_1(z)$ to obtain fast formulas suitable for tiny arguments:

$$\begin{aligned}
s &= \log(v)+\gamma,\\
Y_0(z) &= -\frac{2}{\pi}(-s-(-s+1)w-(1/8)(2s-3)w^2-(1/216)(-6s+11)w^3-\\
&\qquad\cdots),\\
Y_1(z) &= -\frac{1}{\pi v}(1+(-2s+1)w+(s-5/4)w^2+(1/18)(-3s+5)w^3+\cdots).
\end{aligned}$$

In the term $\log(v)+\gamma$, notice that when $z < 2$, the logarithm is negative, so there is a subtraction from γ that can lose leading bits. There is complete loss near $z = 2\exp(-\gamma) \approx 1.123$. The solution is to rewrite the problem expression like this:

$$\begin{aligned}
\log(v)+\gamma &= \log(v)+\log(\exp(\gamma))\\
&= \log(v\exp(\gamma))\\
&= \log(v \times 1.781\,072\,417\,990\,197\,985\,236\,504\,103\,107\,179\cdots)\\
&= \log(z \times 0.890\,536\,208\,995\,098\,992\,618\,252\,051\,553\,589\cdots).
\end{aligned}$$

The last form is preferred, because in hexadecimal arithmetic, it preserves maximal accuracy in the constant.

Unfortunately, the series for $Y_0(z)$ has two other computational problems:

- When the argument of the logarithm is near one, which happens for $z \approx 1.123$, the logarithm is small, and loses accuracy. One solution is to replace it with the logarithm-plus-one function, log1p(), and compute the argument accurately with the help of a fused multiply-add operation:

$$\begin{aligned}
v\exp(\gamma) &= 1+d,\\
d &= v\exp(\gamma)-1\\
&= \mathrm{fma}(v, \exp(\gamma), -1),\\
\log(v\exp(\gamma)) &= \mathrm{log1p}(d).
\end{aligned}$$

We can improve the accuracy of the argument d by splitting the exponential constant into exact high and approximate low parts:

$$\begin{aligned}\exp(\gamma) &= c_{\mathrm{hi}} + c_{\mathrm{lo}},\\ d &= \mathrm{fma}(v, c_{\mathrm{hi}}, -1) + v c_{\mathrm{lo}}\\ &= \mathrm{fma}(v, c_{\mathrm{lo}}, \mathrm{fma}(v, c_{\mathrm{hi}}, -1)).\end{aligned}$$

- The summation contains terms of alternating signs, and numerical experiments show that the ratio of the sums of positive and negative terms exceeds $\frac{1}{2}$ when $2.406 \leq x$, implying loss of leading digits. For $x = 5$, two decimal digits are lost, for $x = 10$, four are lost, and for $x = 20$, eight are lost.

Higher precision, when available, can hide the summation loss if x is not too big, but a better approach is to replace the sum, or the entire expansion of $Y_0(x)$, with a polynomial fit. Several published algorithms for that function do just that, and to limit the polynomial degrees, the range of x where the sum is used is split into several regions, each with its own polynomial fit.

21.4 Experiments with recurrences for $J_0(x)$

The epigraph that begins this chapter comes from an often-cited paper on three-term recurrence relations [Gau67], and its author shows a numeric example of upward recurrence from accurate starting values of $J_0(1)$ and $J_1(1)$. Here is that experiment done in 32-bit decimal arithmetic in hoc, using initial values correctly rounded to seven digits from a high-precision computation in Maple, and augmented with comments showing correct results:

```
hocd32> x = 1
hocd32> J_km1 = 0.7651977
hocd32> J_k   = 0.4400506
hocd32> printf("%2d %.6e\n", 0, J_km1)
hocd32> for (k = 1; k <= 20; ++k) \
hocd32> {
hocd32>     J_kp1 = (2 * k / x) * J_k - J_km1
hocd32>     printf("%2d %.6e\n", k, J_k)
hocd32>     J_km1 = J_k
hocd32>     J_k = J_kp1
hocd32> }
 0 7.651977e-01                # expect  0   7.651977e-01
 1 4.400506e-01                # expect  1   4.400506e-01
 2 1.149035e-01                # expect  2   1.149035e-01
 3 1.956340e-02                # expect  3   1.956335e-02
 4 2.476900e-03                # expect  4   2.476639e-03
 5 2.518000e-04                # expect  5   2.497577e-04
 6 4.110000e-05                # expect  6   2.093834e-05
 7 2.414000e-04                # expect  7   1.502326e-06
 8 3.338500e-03                # expect  8   9.422344e-08
 9 5.317460e-02                # expect  9   5.249250e-09
...
15 7.259898e+06                # expect 15   2.297532e-17
16 2.175373e+08                # expect 16   7.186397e-19
17 6.953934e+09                # expect 17   2.115376e-20
18 2.362163e+11                # expect 18   5.880345e-22
19 8.496833e+12                # expect 19   1.548478e-23
20 3.226435e+14                # expect 20   3.873503e-25
```

Almost half the digits are incorrect at $k = 4$, all are incorrect at $k = 6$, and instead of decreasing towards zero, the values computed with seven-digit arithmetic begin to grow at $k = 7$.

Increasing precision helps only marginally. In 64-bit decimal arithmetic, with 16 digits of precision, half the digits are wrong at $k = 6$, all are wrong at $k = 9$, and values increase at $k = 11$.

In 128-bit decimal arithmetic, with 34 digits of precision, half the digits are in error at $k = 11$, all are erroneous at $k = 16$, and the increase begins at $k = 17$.

Similar experiments with $x = 1/10$ show that the generated values are completely wrong at $k = 4$, and with $x = 1/1000$, at $k = 2$. Clearly, forward recurrence for $J_n(x)$ is extremely unstable when $x \ll n$.

We now repeat the experiment in seven-digit arithmetic using *backward* recurrence:

```
hocd32> x = 1
hocd32> J_kp1 = 9.227622e-27       # J(21,x)
hocd32> J_k   = 3.873503e-25       # J(20,x)
hocd32> printf("%2d %.6e\n", 20, J_k); \
hocd32> for (k = 20; k > 0; --k)       \
hocd32> {
hocd32>     J_km1 = (2*k/x)*J_k - J_kp1
hocd32>     printf("%2d %.6e\n", k - 1, J_km1)
hocd32>     J_kp1 = J_k
hocd32>     J_k = J_km1
hocd32> }
20 3.873503e-25                # expect 20   3.873503e-25
19 1.548478e-23                # expect 19   1.548478e-23
18 5.880342e-22                # expect 18   5.880345e-22
17 2.115375e-20                # expect 17   2.115376e-20
16 7.186395e-19                # expect 16   7.186397e-19
15 2.297531e-17                # expect 15   2.297532e-17
...
 9 5.249246e-09                # expect  9   5.249250e-09
 8 9.422337e-08                # expect  8   9.422344e-08
 7 1.502325e-06                # expect  7   1.502326e-06
 6 2.093833e-05                # expect  6   2.093834e-05
 5 2.497577e-04                # expect  5   2.497577e-04
 4 2.476639e-03                # expect  4   2.476639e-03
 3 1.956335e-02                # expect  3   1.956335e-02
 2 1.149035e-01                # expect  2   1.149035e-01
 1 4.400506e-01                # expect  1   4.400506e-01
 0 7.651977e-01                # expect  0   7.651977e-01
```

Although there are differences in final digits in two-thirds of the results, the largest relative error is just 0.83 ulps, at $k = 11$. The final six results are correctly rounded representations of the exact values. Evidently, backward recurrence is stable for that computation.

For $|x| \gg n$, the stability problem in the upward recurrence largely disappears. Even for $x = 20$ in our seven-digit-arithmetic example, the computed values, not shown here, have a maximum absolute error of 4×10^{-7}. Because the starting values for upward recurrence are simpler than for downward recurrence, we exploit that fact later in code for sequences of Bessel functions of a fixed argument and increasing orders.

Interval arithmetic provides another way to illustrate the instability of upward recurrence for $J_n(x)$ when $x \ll n$, and the return of stability when $x \gg n$. Here is the output of an interval version of the recurrence implemented in the 32-bit binary version of hoc, where the arguments of the test function are the maximum order and the interval bounds for x. The starting values of $J_0(x)$ and $J_1(x)$ are determined from higher-precision computation, then converted to interval form with a half-ulp halfwidth. Each line shows the order n, the lower and upper bounds on the function value, the interval midpoint and its halfwidth, and the expected value determined by computing it in higher precision, and then casting it to working precision:

```
hoc32> load("ijn")    # code for interval version of UPWARD recurrence for Jn(n,x)
hoc32> test_Jn_up(20, 1, 1)
 0  [ 7.651_976e-01,  7.651_978e-01] =  7.651_977e-01 +/- 5.960_464e-08  # expect  7.651_977e-01
 1  [ 4.400_505e-01,  4.400_506e-01] =  4.400_506e-01 +/- 2.980_232e-08  # expect  4.400_506e-01
 2  [ 1.149_033e-01,  1.149_036e-01] =  1.149_035e-01 +/- 1.192_093e-07  # expect  1.149_035e-01
 3  [ 1.956_272e-02,  1.956_373e-02] =  1.956_323e-02 +/- 5.066_395e-07  # expect  1.956_335e-02
 4  [ 2.472_758e-03,  2.479_076e-03] =  2.475_917e-03 +/- 3.159_046e-06  # expect  2.476_639e-03
 5  [ 2.183_318e-04,  2.698_898e-04] =  2.441_108e-04 +/- 2.577_901e-05  # expect  2.497_577e-04
 6  [-2.957_582e-04,  2.261_400e-04] = -3.480_911e-05 +/- 2.609_491e-04  # expect  2.093_834e-05
...
15  [-1.202_778e+08,  8.013_022e+07] = -2.007_379e+07 +/- 1.002_040e+08  # expect  2.297_532e-17
16  [-3.611_190e+09,  2.408_198e+09] = -6.014_958e+08 +/- 3.009_694e+09  # expect  7.186_397e-19
17  [-1.156_382e+11,  7.718_263e+10] = -1.922_779e+10 +/- 9.641_042e+10  # expect  2.115_376e-20
18  [-3.934_108e+12,  2.627_821e+12] = -6.531_434e+11 +/- 3.280_964e+12  # expect  5.880_344e-22
19  [-1.417_051e+14,  9.471_720e+13] = -2.349_394e+13 +/- 1.182_111e+14  # expect  1.548_478e-23
20  [-5.387_421e+15,  3.603_188e+15] = -8.921_166e+14 +/- 4.495_304e+15  # expect  3.873_503e-25
```

Notice that already at $n = 6$, the interval includes zero, and the interval halfwidth is larger than its midpoint. The rapidly growing interval width gives one little confidence in the function values estimated from the interval midpoints.

We then repeat the experiment with $x = 20$, and see that, although the interval widths grow, they remain small compared to the midpoints, and the midpoints are close to the expected values:

```
hoc32> test_Jn_up(20, 20, 20)
 0  [ 1.670_246e-01,  1.670_247e-01] =  1.670_247e-01 +/- 1.490_116e-08  # expect  1.670_247e-01
 1  [ 6.683_312e-02,  6.683_313e-02] =  6.683_312e-02 +/- 7.450_581e-09  # expect  6.683_312e-02
 2  [-1.603_414e-01, -1.603_413e-01] = -1.603_414e-01 +/- 2.235_174e-08  # expect -1.603_414e-01
 3  [-9.890_141e-02, -9.890_138e-02] = -9.890_139e-02 +/- 1.490_116e-08  # expect -9.890_139e-02
 4  [ 1.306_709e-01,  1.306_710e-01] =  1.306_709e-01 +/- 3.725_290e-08  # expect  1.306_709e-01
 5  [ 1.511_697e-01,  1.511_698e-01] =  1.511_697e-01 +/- 3.725_290e-08  # expect  1.511_698e-01
 6  [-5.508_611e-02, -5.508_599e-02] = -5.508_605e-02 +/- 5.960_464e-08  # expect -5.508_605e-02
...
15  [-8.174_032e-04, -8.067_638e-04] = -8.120_835e-04 +/- 5.319_715e-06  # expect -8.120_690e-04
16  [ 1.451_691e-01,  1.451_905e-01] =  1.451_798e-01 +/- 1.072_884e-05  # expect  1.451_798e-01
17  [ 2.330_773e-01,  2.331_223e-01] =  2.330_998e-01 +/- 2.249_330e-05  # expect  2.330_998e-01
18  [ 2.510_408e-01,  2.511_387e-01] =  2.510_898e-01 +/- 4.899_502e-05  # expect  2.510_898e-01
19  [ 2.187_510e-01,  2.189_724e-01] =  2.188_617e-01 +/- 1.106_933e-04  # expect  2.188_619e-01
20  [ 1.644_882e-01,  1.650_068e-01] =  1.647_475e-01 +/- 2.593_249e-04  # expect  1.647_478e-01
```

21.5 Computing $J_0(x)$ and $J_1(x)$

For tiny x, summing the first few terms of the Taylor series of $J_n(x)$ provides a correctly rounded result. The mathcw library code uses four-term series for the Bessel functions $J_0(x)$ and $J_1(x)$.

For x in $[\text{tiny}, 2]$, the series can be summed to machine precision, or $J_0(x)$ and $J_1(x)$ can be represented by polynomial approximations. The mathcw library code for those two functions implements both methods, and for each function, uses a single Chebyshev polynomial table (see **Section 3.9** on page 43) that is truncated at compile time to the accuracy needed for the current working precision, avoiding the need for separate minimax polynomials for each precision. For those functions, and the intervals treated in this section, minimax fits are only slightly more accurate than Chebyshev fits of the same total degree.

The polynomial for $J_0(x)$ is chosen to fit the remainder of the Taylor series after the first two terms, and because of the symmetry relation, we assume that x is nonnegative:

$$\begin{aligned}
t &= x^2, && \textit{t in } [0,4], \textit{ x in } [0,2], \\
u &= t/2 - 1, && \textit{Chebyshev variable u in } [-1,+1], \\
f(t) &= (J_0(\sqrt{t}) - (1 - t/4))/t^2, \\
&= \sum_{k=0}^{N} c_k T_k(u), && \textit{Chebyshev polynomial fit} \\
J_0(x) &= 1 - t/4 + t^2 f(t) \\
&= 1 - x^2/4 + t^2 f(t) \\
&= \operatorname{fma}(-x/2, x/2, 1) + t^2 f(t) \\
&= (1 - x/2)(1 + x/2) + t^2 f(t), && \textit{use when } \beta \neq 16,
\end{aligned}$$

$$= 2((1 - x/2)(1/2 + x/4)) + t^2 f(t), \qquad \textit{use when } \beta = 16.$$

When $t \approx 4$, the leading sum $1 - t/4$ suffers serious loss of leading digits, unless it is computed with a fused multiply-add operation. However, rewriting it in factored form moves the subtraction loss to the term $1 - x/2$, and for $\beta = 2$, that is computed almost exactly, with only a single rounding error. Using higher precision for intermediate computations, but not for the Chebyshev sum, reduces the worst-case errors by about one ulp.

Two preprocessor symbols, `USE_CHEBYSHEV` and `USE_SERIES`, select the algorithm used in `j0x.h` when x is in $[\text{tiny}, 2]$. Error plots show that both methods are equally accurate for small x, but errors are smaller for the Chebyshev fit and higher intermediate precision, so that is the default if neither symbol is defined.

For $J_1(x)$ with x in $[0, 2]$, t and u are as before, and we develop a polynomial fit for nonnegative x like this:

$$\begin{aligned} g(t) &= (2J_1(\sqrt{t})/\sqrt{t} - (1 - t/8))/t^2 \\ &= \sum_{k=0}^{N} d_k T_k(u), && \textit{Chebyshev polynomial fit,} \\ J_1(x) &= (x/2)(1 - t/8 + t^2 g(t)). \end{aligned}$$

Here, $t/8 \le 1/2$, so there is no subtraction loss in the first two terms, and the terms can be summed safely from right to left. The major sources of error are the two rounding errors from the final subtraction and the final product with $x/2$.

There are single roots $J_0(2.404\cdots) = 0$ and $J_1(3.831\cdots) = 0$ in the interval $[2, 4]$. Straightforward polynomial approximations on that interval are possible, but near the roots, they must sum to a small number, and as a result, have a large relative error. The Cephes library [Mos89, Chapter 6] removes that error by instead using a polynomial approximation in which the roots in the interval are factored out. For our case, we call the respective roots r and s, and we have:

$$\begin{aligned} J_0(x) &= (x^2 - r^2)p(x), & p(x) &= \sum_{k=0}^{N} p_k T_k(u), && \textit{Chebyshev polynomial fit,} \\ J_1(x) &= x(x^2 - s^2)q(x), & q(x) &= \sum_{k=0}^{N} q_k T_k(u), && \textit{Chebyshev polynomial fit.} \end{aligned}$$

The critical computational issues are that we must avoid massive subtraction loss in the factors with the roots, and we must account for the fact that the exact roots are not machine numbers. That is best done by representing each root as a sum of exact high and approximate low parts, and then computing the factor like this:

$$\begin{aligned} x^2 - r^2 &= (x - r)(x + r), \\ &= ((x - r_{\text{hi}}) - r_{\text{lo}})((x + r_{\text{hi}}) + r_{\text{lo}}). \end{aligned}$$

When x is near a root, the difference $x - r_{\text{hi}}$ is computed exactly, because both terms have the same floating-point exponent. Subtraction of r_{lo} then provides an important correction to the difference, with only a single rounding error. The second factor is computationally stable because it requires only additions.

We need a different split of each root for each machine precision, and a Maple function in the file `split-base.map` generates the needed C code, with embedded preprocessor conditional statements to select an appropriate set for the current precision. For example, one such pair with its compile-time selector in `j0.h` looks like this for the IEEE 754 and VAX 32-bit formats:

```
#elif (T >= 24) && (B != 16)

static const fp_t J0_R1_HI = FP(10086569.0) / FP(4194304.0);
static const fp_t J0_R1_LO = FP(1.08705905e-07);
```

The one drawback to our factored representation is that the final function value is no longer computed as the sum of an exact value and a small correction. Instead, it has cumulative rounding errors from each of the factors and their products. We therefore expect larger errors on the interval $[2, 4]$ than on $[0, 2]$, but we can nevertheless guarantee a small relative error, instead of a small absolute error.

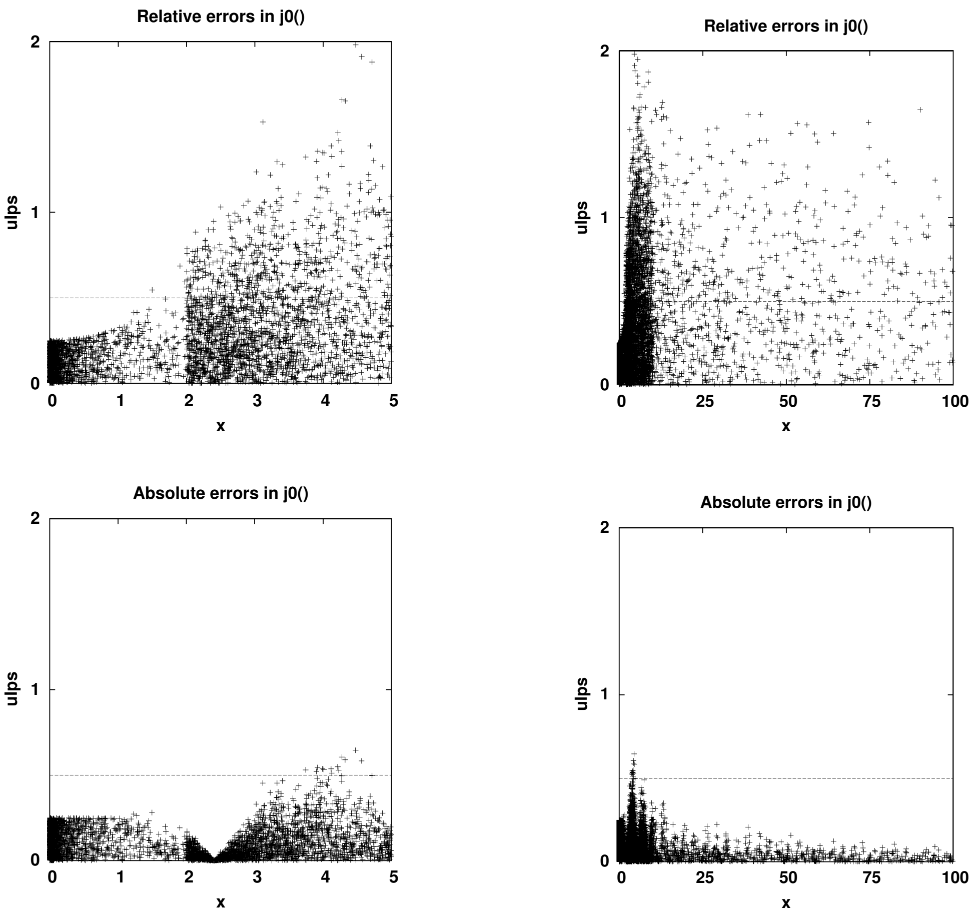


Figure 21.3: Relative (top) and absolute (bottom) errors in `j0(x)` for two argument ranges. The largest relative errors occur near the zeros of the function that lie in $[4, \infty)$, and can grow arbitrarily large. However, the absolute errors remain small.

For the interval $[4, \infty)$, we use Chebyshev polynomial fits to the functions $\mathrm{P}(0, x)$, $\mathrm{P}(1, x)$, $\mathrm{Q}(0, x)$, and $\mathrm{Q}(1, x)$ (see Section 21.3 on page 699), and then recover the Bessel functions from the formulas involving those functions and the cosine and sine. There are now cumulative errors from the polynomial evaluations, the trigonometric functions, and their final product sums. We therefore expect higher errors than on the interval $[0, 4]$, and sadly, we can only provide small absolute error near the roots after the first one.

For large x, $\mathrm{P}(\nu, x)$ is $\mathcal{O}(1)$, and dominates $\mathrm{Q}(\nu, x)$, which is $\mathcal{O}(1/x)$. Nevertheless, because the trigonometric multipliers can be small, and of either sign, the two products may be of comparable size, and their sum subject to subtraction loss. Also, for large x, the accuracy of the computed Bessel functions depends critically on that of the trigonometric argument reduction, and it is precisely here that most historical implementations of $J_n(x)$ and $Y_n(x)$ may deliver results where every digit is in error, even if the magnitudes are correct. Thanks to the exact argument reduction used in the `mathcw` library, that is not a problem for us, and as long as x is exactly representable, and not near a Bessel function root, our `j0(x)` and `j1(x)` functions produce results that agree to within a few ulps of high-precision values computed by symbolic-algebra systems, even when x is the largest number representable in the floating-point system.

Figure 21.3 and Figure 21.4 on the following page show the relative and absolute errors for our implementations in data type `double` of the $J_0(x)$ and $J_1(x)$ functions. Error plots for other binary and decimal precisions are similar, and thus, not shown. The observed errors quantify the rough estimates that we made based on the numerical steps of the computations.

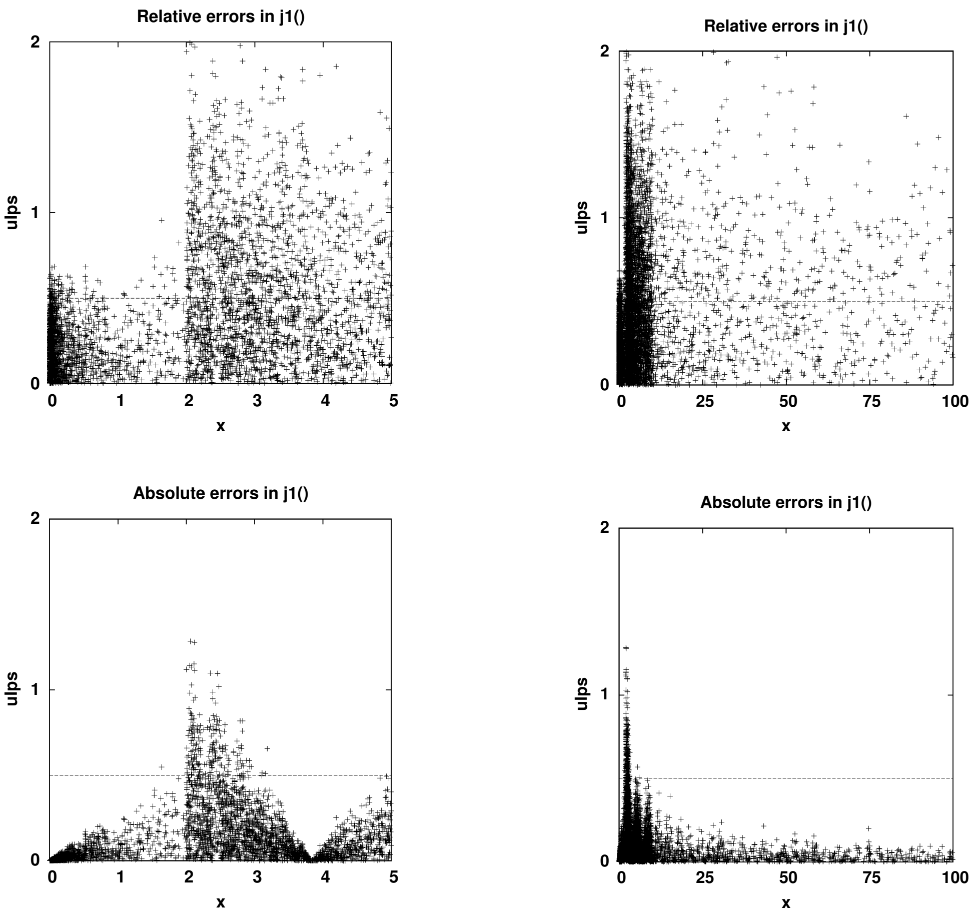


Figure 21.4: Relative (top) and absolute (bottom) errors in `j1(x)` for two argument ranges. The largest relative errors occur near the zeros of the function that lie in $[4, \infty)$, and can grow arbitrarily large. However, the absolute errors remain small.

21.6 Computing $J_n(x)$

We now turn to the problem of computing $J_n(x)$ for $n > 1$. As we observed earlier, when x is small, the series formula converges rapidly, and is the best way to compute the function for arbitrary n.

For larger x values, although upward recurrence for computing $J_n(x)$ is unsuitable, downward recurrence has been found to be stable, but the problem is that we do not have a good way to compute the two starting Bessel functions directly when both n and x are large. The continued fraction presented earlier on page 702 leads to an algorithm for finding those two functions, but the procedure is not immediately obvious, and requires some explanation.

In the notation of Section 2.7 on page 12, the elements of the continued fraction for $J_\nu(z)/J_{\nu-1}(z)$ are given by

$$\begin{aligned} a_1 &= +1, & a_k &= -1, & k &> 1, \\ b_0 &= 0, & b_k &= 2(\nu + k - 1)/z, & k &> 0. \end{aligned}$$

The numerically preferred backward evaluation of the continued fraction is complicated because we have two parameters that affect the starting value of the iteration. However, either of the Lentz or Steed algorithms allows evaluation in the forward direction with early loop exit as soon as the value of the continued fraction has converged. That gives us the ratio $J_\nu(z)/J_{\nu-1}(z)$, but we still do not know $J_\nu(z)$ itself. To find that value, rewrite the recurrence relation in the downward direction for decreasing *integer* orders, and divide each equation by $J_n(z)$:

$$\begin{aligned}
J_{n-2}(z)/J_n(z) &= (2(n-1)/z)J_{n-1}(z)/J_n(z) - 1,\\
J_{n-3}(z)/J_n(z) &= (2(n-2)/z)J_{n-2}(z)/J_n(z) - J_{n-1}(z)/J_n(z),\\
J_{n-4}(z)/J_n(z) &= (2(n-3)/z)J_{n-3}(z)/J_n(z) - J_{n-2}(z)/J_n(z),\\
\ldots &\quad \ldots\\
J_2(z)/J_n(z) &= (6/z)J_3(z)/J_n(z) - J_4(z)/J_n(z),\\
J_1(z)/J_n(z) &= (4/z)J_2(z)/J_n(z) - J_3(z)/J_n(z),\\
J_0(z)/J_n(z) &= (2/z)J_1(z)/J_n(z) - J_2(z)/J_n(z).
\end{aligned}$$

The right-hand side of the first equation contains known values, so we can easily compute its left-hand side. The right-hand side of the second equation requires two ratios that we now have, so we can find its left-hand side. We need to remember at most two consecutive ratios to repeat the process, and we finally obtain a value for $J_0(z)/J_n(z)$. Calling that result $f_n(z)$, we now compute $J_0(z)$ independently by the methods of **Section 21.5** on page 707, and then recover the desired function value $J_n(z)$ as $J_0(z)/f_n(z)$.

There is an important refinement to be made in that last step. If we are near a zero of $J_0(z)$, then the function value is likely to have a high relative error that propagates into the computed $J_n(z)$. It is then better to use the second-last ratio, and compute $J_n(z)$ from $J_1(z)/(J_1(z)/J_n(z))$. We use that alternative when the magnitude of the ratio $J_1(z)/J_n(z)$ exceeds that of $J_0(z)/J_n(z)$. The root separation of the estimates in **Section 21.3** on page 697 guarantees that successive ratios cannot both be tiny.

Although that procedure may look complicated, the code that implements it is short. Here is a hoc function that computes $J_n(x)$, with checks for special cases of n and x omitted:

```
func Jncf(n,x)                              \
{   # return J(n,x) via the continued-fraction algorithm
    rinv = cf(n,x)          # J(n,x) / J(n-1,x)
    rk = 1 / rinv           # J(k,x) / J(n,x), for k == n - 1
    rkp1 = 1
    s = 1

    for (k = n - 1; k > 0; --k)             \
    {
        rkm1 = (2 * k / x) * rk - rkp1
        rkp1 = rk
        rk = rkm1

        if (isinf(rkm1))                    \
        {
            rk *= MINNORMAL
            rkp1 *= MINNORMAL
            s *= MINNORMAL
            rkm1 = (2 * k / x) * rk - rkp1
        }
    }

    if (abs(rkp1) > abs(rk))                \
        return (s * J1(x) / rkp1)           \
    else                                    \
        return (s * J0(x) / rk)
}
```

The check for infinity in the downward loop is essential, because when n/x is large, the successive ratios grow quickly toward the overflow limit. The computation cannot be allowed to proceed normally, because the next one or two iterations require subtraction of infinities, producing NaN results. To prevent that, the code instead performs an exact downward scaling that is undone in the `return` statement. Our choice of scale factor forces `s`, and thus, the final result, to underflow if scaling is required more than once. For older floating-point designs where overflow is fatal,

Table 21.5: Iteration counts for evaluating the continued fraction of $J_n(x)/J_{n-1}(x)$ in the IEEE 754 128-bit format (approximately 34 decimal digits) using the forward Lentz algorithm. Counts for the Steed algorithm are almost identical.

				n			
x	2	10	100	1000	10 000	100 000	1 000 000
1	16	15	10	7	5	7	4
10	37	30	14	9	7	5	5
100	156	148	58	15	9	7	5
1 000	1120	1112	1021	119	16	9	9
10 000	10 259	10 251	10 165	9255	251	15	9
100 000	>25 000	>25 000	>25 000	>25 000	>25 000	533	15
1 000 000	>25 000	>25 000	>25 000	>25 000	>25 000	>25 000	1122

a safe-arithmetic function like the `is_fmul_safe()` procedure that we introduced in our treatment of the incomplete gamma function (see Section 18.4 on page 560) can check whether the expression assigned to `rkm1` would overflow without causing overflow, and exit the loop with `rk` set to the largest floating-point number.

The private function `cf(n,x)` evaluates the continued fraction, and an instrumented version of its code stores the iteration count in a global variable that allows recovery of the data shown in Table 21.5. Evidently, the continued fraction converges quickly when x/n is small, but it is impractical for $x > 1000$ if $x > n$.

Our simple prototype for computing $J_n(x)$ can be improved in at least these ways, most of which are implemented in the C code in the file `jnx.h`:

- For small x, use the general Taylor series (see Section 21.3 on page 702). It is particularly effective for large n and tiny x.
- If `s` is zero, or if `1/rk` or `1/rkp1` underflows, the calculation of $J_0(x)$ or $J_1(x)$ is unnecessary.
- Instead of allowing the ratios to grow toward the overflow limit, it is better to rescale earlier. A suitable cutoff is roughly the square root of the largest representable number. With that choice, overflow elsewhere in the loop is prevented, and there is no need to have separate code for older systems where overflow is fatal.
- If possible, use higher working precision in the continued fraction and downward recurrences, to improve accuracy near zeros of $J_n(x)$.
- For large $|x|$, and also whenever the continued fraction is found not to converge, switch to the asymptotic expansion given in Section 21.3 on page 697. Terminate the summations for $P(n,x)$ and $Q(n,x)$ as soon as either the partial sums have converged to machine precision, or else the terms are found to increase. Then take particular care with the trigonometric argument reductions to avoid needless loss of accuracy.
- In the region where the asymptotic series is used, the quality of the underlying cosine and sine function implementations, coupled with exact argument reduction, are of prime importance for the accuracy of $J_n(x)$, as well as for some of the spherical Bessel functions that we treat later in this chapter. The trigonometric code in many existing libraries performs poorly for large arguments, and may cause complete loss of significance in the Bessel functions.

The algorithm does not require storage of all of the right-hand side ratios, $J_k(z)/J_n(z)$, but if we preserve them, we can quickly recover a vector of values $J_0(z)$, $J_1(z)$, …, $J_{n-1}(z)$ simply by multiplying each of the saved ratios by $J_n(z)$.

Alternatively, if we determine $J_{n-1}(z)$ and $J_n(z)$ with that algorithm, we can use downward recurrence to find earlier members of the sequence. That algorithm requires $\mathcal{O}(4n)$ floating-point operations, in addition to the work required for the continued fraction, so if n is large, computation of $J_n(x)$ for large x is expensive.

Figure 21.5 on the next page shows the measured errors in our implementation of $J_n(x)$ for a modest value of n.

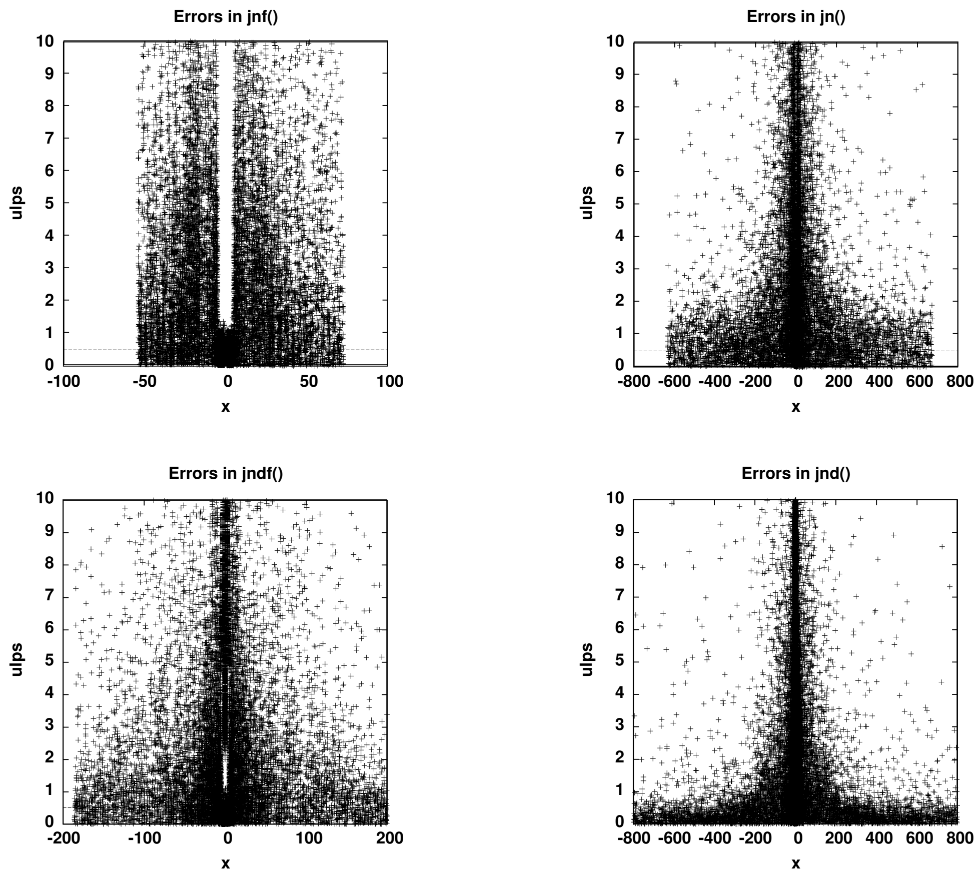


Figure 21.5: Errors in the binary (top) and decimal (bottom) `jn(n,x)` family for $n = 25$.

21.7 Computing $Y_0(x)$ and $Y_1(x)$

As with our implementations of the ordinary Bessel functions of the first kind, for x in $(4, \infty)$, we compute the functions of the second kind, $Y_0(x)$ and $Y_1(x)$, from the trigonometric formulas with the factors $\mathrm{P}(\nu, x)$ and $\mathrm{Q}(\nu, x)$ defined on Section 21.3 on page 698.

The major difficulties for arguments in $[0, 4]$ are the approach to $-\infty$ as $x \to 0$, and the presence of two zeros of $Y_0(x)$, and one of $Y_1(x)$. Unless the zeros are specifically accounted for, polynomial fits or series sums for the functions have large relative errors near those zeros. Consequently, we prefer to use several different approximations in that region. In the following, we assume that the special arguments of NaN, Infinity, and zero are already handled.

For $Y_0(x)$, we use these regions and Chebyshev polynomial approximations, $f_r(u)$, with u on $[-1, +1]$:

x in $(0, \text{tiny}]$: Sum the four-term Taylor series in order of increasing term magnitudes, with the cutoff chosen so that the magnitude of the last term is below $\frac{1}{2}\epsilon/\beta$, but may affect the rounding of the final result.

x in $[3/4, 1]$: Factor out the root in this region, with $Y_0(x) = (x^2 - s_{0,1}^2) f_2(8x - 7)$. The difference of squares must be factored and computed accurately from a two-part split of the root, as described in Section 21.5 on page 707.

x in $(\text{tiny}, 2]$: Sum the series for $Y_0(x)$ starting with the second term, exiting the loop as soon as the last-computed term no longer affects the sum. Then add the first term. If x is in $[-\exp(-\gamma), -3\exp(-\gamma)]$ (roughly $[0.56, 1.69]$), the argument of the logarithm lies in $[\frac{1}{2}, \frac{3}{2}]$, and is subject to accuracy loss, so add the term $\operatorname{log1p}(d) \times J_0(x)$, where d is computed accurately with a fused multiply-add operation and a two-part split of the constant $\exp(\gamma)$. Otherwise, add $\log(\operatorname{fma}(v, c_{\text{hi}}, vc_{\text{lo}})) \times J_0(x)$. $Y_0(x)$ is then that sum times $2/\pi$.

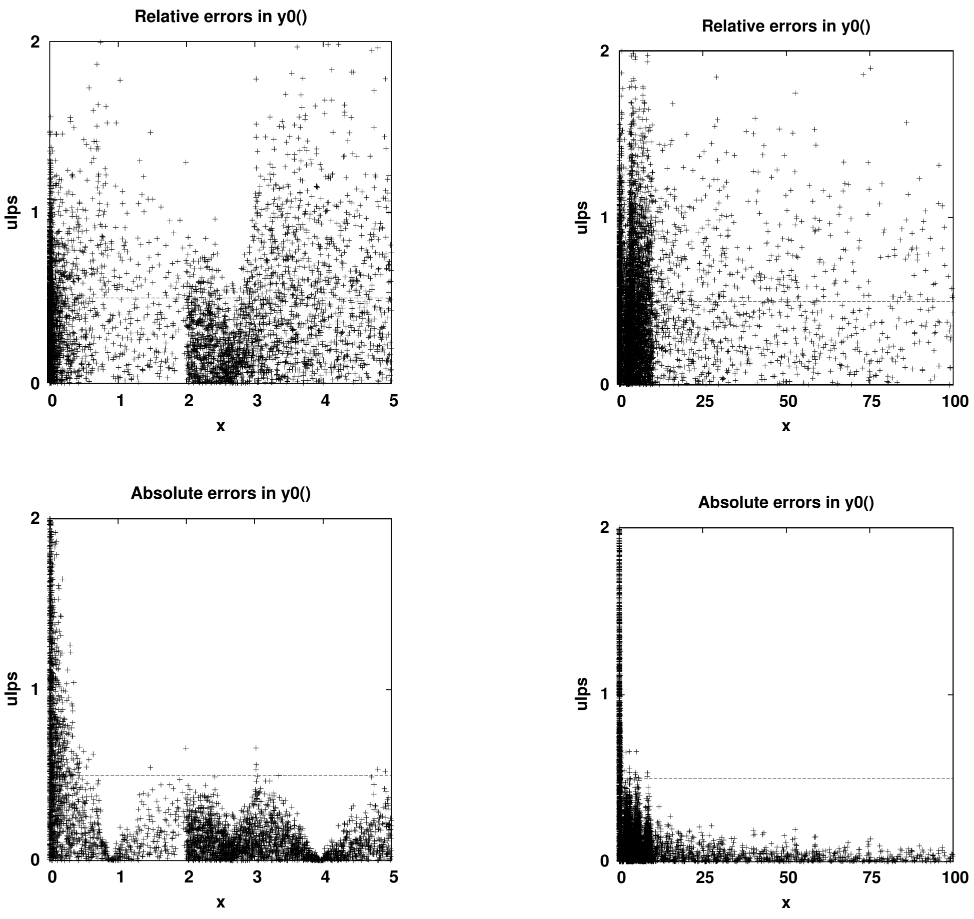


Figure 21.6: Relative (top) and absolute (bottom) errors in `y0(x)` for two argument ranges. The largest relative errors occur near the zeros of the function that lie in $[4, \infty)$, and can grow arbitrarily large. However, the absolute errors remain small.

x in $(2, 3]$: The function is positive and decreasing on this interval, so to get an approximation that is the sum of a positive exact value and a positive correction, use $Y_0(x) = Y_0(3) + x^2 f_4((2x^2 - 13)/5)$. The constant $Y_0(3)$ is represented as a two-part split, with the low part added first.

x in $(3, 4]$: Factor out the root in this region from the approximation, and compute $Y_0(x) = x^2(x^2 - s_{0,2}^2) f_5(2x - 7)$, with the usual careful handling of the difference of squares.

x in $(4, \infty)$: Use the P–Q fit.

The final approximation form in each region is the result of experiments with several alternatives suggested by prior work. The goal is to find auxiliary functions that are almost linear on the region, in order to minimize the length of the Chebyshev expansions. Most published work on computing Bessel functions ignores the issue of the function zeros, and thus, can only achieve small absolute, rather than relative, error. For our approximations, the worst case for 16-digit accuracy is region $(2, 3]$, where Chebyshev terms up to $T_{25}(u)$ are needed.

Cody and Waite generally recommend making the implementation of each elementary or special function independent of related ones. However, for small arguments, eliminating the dependence of $Y_0(x)$ on $J_0(x)$ leads to excessively long polynomial expansions. Further searches for alternate approximation forms are needed to see whether that blemish can be removed without losing accuracy.

Figure 21.6 shows the measured errors in our implementation of $Y_0(x)$. Because the function magnitude grows as its argument approaches zero, the absolute errors must also increase in that region.

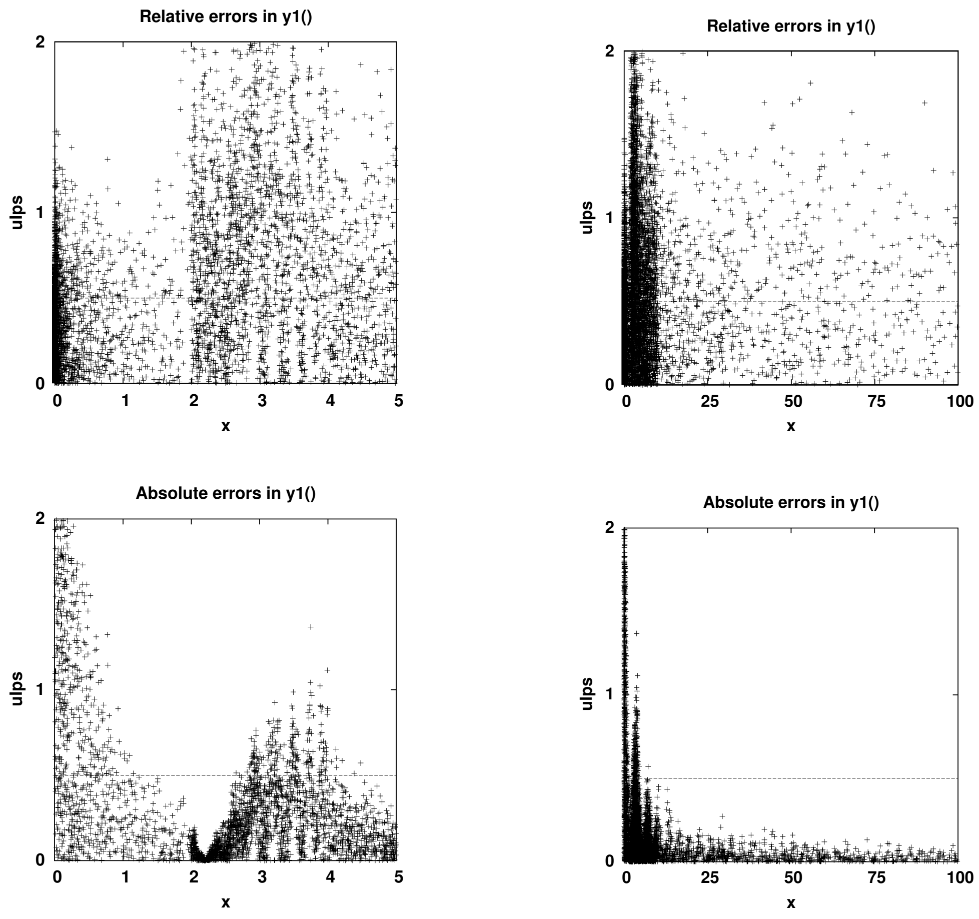


Figure 21.7: Relative (top) and absolute (bottom) errors in `y1(x)` for two argument ranges. The largest relative errors occur near the zeros of the function that lie in $[4, \infty)$, and can grow arbitrarily large. However, the absolute errors remain small.

For $Y_1(x)$, we use these regions and different Chebyshev polynomial fits, $f_r(u)$:

x in $(0, \text{tiny}]$: Sum the four-term Taylor series, similar to our approach for tiny arguments in $Y_0(x)$.

x in $(\text{tiny}, 2]$: Use $Y_1(x) = (2/\pi)(\log(x) \times J_1(x) - 1/x) + xf_2(x^2/2 - 1)$.

x in $(2, 4]$: Use $Y_1(x) = (x^2 - s_{1,1}^2)f_3(x - 3)$.

x in $(4, \infty)$: Use the P–Q fit.

Here too, we have been unable to remove the dependence of $Y_1(x)$ on $J_1(x)$ for small arguments without unacceptably long polynomial fits.

Figure 21.7 shows the measured errors in our implementation of $Y_1(x)$.

21.8 Computing $Y_n(x)$

The Bessel function $Y_n(x)$ is complex-valued for negative x, so the code should return a quiet NaN for that case. If NaN is not available in the host floating-point system, then a suitable replacement might be the negative of the largest representable number.

For negative n, use the symmetry relation $Y_{-n}(x) = (-1)^n Y_n(x)$ so that only nonnegative n values need to be considered further.

For $n = 0$ and $n = 1$, use the routines for $Y_0(x)$ and $Y_1(x)$.

For small arguments, $Y_n(x)$ is best computed by summing the general Taylor series (see **Section 21.3** on page 703).

Overflow is possible for tiny x, but the complexity of the series is such that it is not practical to precompute cutoffs for general n below which the code could return $-\infty$, or if infinities are not available, the negative of the largest representable number. Numerical experiments show that for a tiny fixed x and increasing n, the magnitude of $Y_n(x)$ is larger than x^{-n}, so if overflow is a fatal error, then it could be necessary to return an indication of a likely overflow when $x < \exp(-\log(\textit{largest representable number})/n)$, even though that is an expensive test to make.

The recurrence relation for $Y_n(x)$ is stable in the upward direction, so once $Y_0(x)$ and $Y_1(x)$ have been computed by the methods of the preceding section, $Y_{|n|}(x)$ is produced in a loop of short code in $|n| - 2$ iterations. The final result is then negated if n is negative and odd, to account for the symmetry relation.

The explicit formula for $Y_n(x)$ contains an n-term sum, a product of a logarithm and $J_n(x)$, and sum of an infinite number of terms. When $x < 2$, the first sum is larger than the product and the second sum, and the terms in both sums fall off quickly. Upward recurrence requires $Y_0(x)$ and $Y_1(x)$, but they are easier to compute than $J_n(x)$. It is therefore unclear which algorithm should be chosen. Timing tests show that the series algorithm is about two to four times faster than upward recurrence on some platforms, whereas on others, it is about twice as slow. On the common IA-32 architecture, the two algorithms have nearly equal performance.

Although the code for the asymptotic series of $J_n(x)$ can be adapted to compute $Y_n(x)$ with only a one-line change, it is not essential, because upward recurrence from $Y_0(x)$ and $Y_1(x)$ works for arguments of any size. However, we provide code for the asymptotic series for $Y_n(x)$, because there is a tradeoff in efficiency between a long recurrence, and a short sum that also requires a complicated argument reduction inside the trigonometric functions. Timing tests on two common platforms for $Y_{25}(x)$ and $Y_{1000}(x)$ with random arguments in the region where the asymptotic code is used shows that code to be two to ten times faster than the code that uses downward recurrence. The asymptotic series is therefore the default algorithm for large arguments.

We do not have a satisfactory algorithm to handle the case of $n \gg x \gg 1$ for either $J_n(x)$ or $Y_n(x)$, because the continued fraction then converges too slowly to be practical, three-term recurrences take too many steps, and the asymptotic series cannot produce sufficient accuracy. Fortunately, for most applications where Bessel functions are needed, such extreme arguments are rare.

Figure 21.8 on the next page shows the measured errors in our implementation of $Y_n(x)$ for modest n.

21.9 Improving Bessel code near zeros

After this chapter, and its software, were completed, John Harrison described a significant improvement in the computation of the Bessel functions $J_0(x)$, $J_1(x)$, $Y_0(x)$, and $Y_1(x)$ that he implemented in the math library for the Intel compiler family [Har09b]. His code uses separate polynomial fits around the zeros and extrema of those functions for arguments x in $[0, 45]$. Above that region, the functions are represented with single trigonometric functions as

$$J_n(x) \approx \mathcal{P}(1/x)\cos(x - (\tfrac{1}{2}n + \tfrac{1}{4})\pi - \mathcal{Q}(1/x)),$$
$$Y_n(x) \approx \mathcal{P}(1/x)\sin(x - (\tfrac{1}{2}n + \tfrac{1}{4})\pi - \mathcal{Q}(1/x)),$$

where $\mathcal{P}(1/x)$ and $\mathcal{Q}(1/x)$ are polynomials in inverse powers of x. His algorithm improves the relative accuracy of the Bessel functions near their zeros, and ensures monotonocity near their extrema.

Harrison also carried out an exhaustive search to find arguments $x < 2^{90} (\approx 10^{27})$ that are the worst cases for determining correct rounding. For the IEEE 754 64-bit binary format, he found that at most *six* additional bits in the Bessel-function values are required for correct rounding decisions. In particular, accurate computations in the 80-bit format provide more than the required additional bits to guarantee correct rounding in the 64-bit format.

Because the mathcw library supports more floating-point formats, and decimal arithmetic, using Harrison's approach would require a large number of polynomial tables. It therefore is reasonable to ask whether there is a simpler way to improve relative accuracy near the zeros of the Bessel functions. We start by generating the Taylor series of $J_0(x)$ near a point $x = z + d$, such that $J_0(z) = 0$:

```
% maple
> alias(J = BesselJ):
```

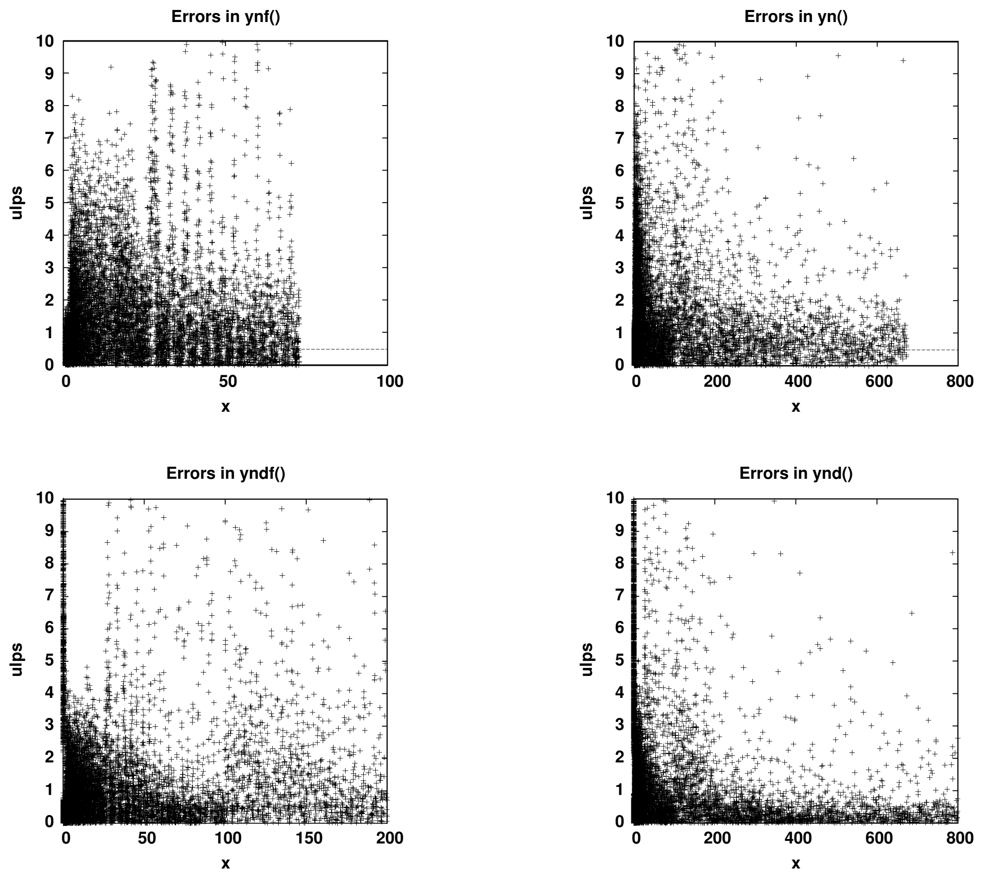


Figure 21.8: Errors in the binary (top) and decimal (bottom) yn(n,x) family for $n = 25$.

```
> taylor(J(0, z + d), d = 0, 5);
                                                 2
J(0, z) - J(1, z) d + (1/4 J(2, z) - 1/4 J(0, z)) d  +
                                 3
    (-1/24 J(3, z) + 1/8 J(1, z)) d  +
                                                 4       5
    (1/192 J(4, z) - 1/48 J(2, z) + 1/64 J(0, z)) d  + O(d )
```

The expansion coefficients require values of higher-order Bessel functions at z, so let us apply the three-term recurrence relation, and then display the numerator and denominator:

```
> t := convert(%, polynom):
> u := subs(J(4,z) = (6/z)*J(3,z) - J(2,z),
            J(3,z) = (4/z)*J(2,z) - J(1,z),
            J(2,z) = (2/z)*J(1,z) - J(0,z),
            J(0,z) = 0, t):
> numer(u);
                  3         2       2         2  3       3    3  2
-J(1, z) d (12 z  - 6 d z  + 4 d  z - 2 d  z  - 3 d  + d  z )
> denom(u);
    3
12 z
```

Notice that the Taylor series reduces to a rational polynomial scaled by the constant $J_1(z)$. Next, introduce two intermediate variables and simplify:

```
> simplify(subs(v^2 = w, subs(z = 1/v, u)));
                                       2        2      3        3
1/12 d J(1, 1/v) (-12 + 6 v d - 4 d  w + 2 d  + 3 d  v w - d  v)
```

We now have a simple polynomial in powers of d, scaled by $J_1(z)$.

A more complex Maple program in the file `J0taylor.map` finds the general form of the polynomial coefficients to any desired order. In Horner form, they involve only whole numbers, and look like this:

```
c[0] = 0;
c[1] = -1;
c[2] = (v) / 2;
c[3] = (1 - 2 * w) / 6;
c[4] = ((-1 + 3 * w) * v) / 12;
c[5] = (-1 + (7 - 24 * w) * w) / 120;
c[6] = ((1 + (-11 + 40 * w) * w) * v) / 240;
c[7] = (1 + (-15 + (192 - 720 * w) * w) * w) / 5040;
c[8] = ((-1 + (24 + (-330 + 1260 * w) * w) * w) * v) / 10080;
c[9] = (-1 + (26 + (-729 + (10440 - 40320 * w) * w) * w) * w) / 362880;
...
```

The important point here is that the coefficients for the expansions near *all* of the zeros of $J_0(x)$ are represented with symbolic expressions that can be evaluated at run time for any particular zero, z.

Because v and w are reciprocals, their higher powers decrease and prevent overflow in the coefficients c_k. In addition, the values of the numerators of the leading coefficients are often in $[\frac{1}{2}, 1]$, reducing accuracy loss when the host arithmetic has wobbling precision.

Subtraction loss in the Taylor series is most severe in the sum $c_2d^2 + c_3d^3$, so we can use the series only when $|d| < \frac{1}{2}c_2/c_3 \asymp 3/(2z)$. For the zeros of $J_0(x)$ that we treat, the upper limit on $|d|$ is roughly 0.27.

If we tabulate just the zeros z_k and the corresponding values $J_1(z_k)$, then for any particular zero, z, we can compute numerical values of the coefficients and obtain the Bessel function from its Taylor series as

$$J_0(z + d) = J_1(z)(c_0 + c_1d + c_2d^2 + c_3d^3 + \cdots),$$

where that sum is best evaluated in Horner form.

Given a value x, the Bessel-function zero estimates allow us to find z quickly without having to look at more than three of the tabulated roots. To further enhance accuracy, we store the z_k and $J_1(z_k)$ values as two-part sums of high and low parts, where the high part is exactly representable, correctly rounded, and accurate to working precision.

Similar investigations show that the Taylor series of $J_1(x)$ requires a scale factor of $J_0(z)$, the series for $Y_1(x)$ needs $Y_0(z)$, and that for $Y_1(x)$ needs $Y_0(z)$. The coefficients c_k are identical for $J_0(x)$ and $Y_0(x)$, and another set of c_k values handles both $J_1(x)$ and $Y_1(x)$.

By computing coefficients up to c_{17}, we can use the series for $J_0(x)$ for $|d| < 0.805$ in the IEEE 754 64-bit formats, and $|d| < 0.060$ in the 128-bit formats, subject to the additional limit required to avoid subtraction loss. Otherwise, we fall back to the algorithms described in earlier sections of this chapter.

As long as x lies within the table, we can achieve high accuracy near the zeros. The table requirements are modest: 100 entries handle $x < 311$, and 320 entries suffice for $x < 1000$.

Revised versions of the files `j0x.h`, `j1x.h`, `y0x.h`, and `y1x.h` incorporate optional code that implements the general Taylor-series expansions, and their corresponding header files contain the required data tables of zeros and Bessel-function values at those zeros. The new code is selected by default, but can be disabled with a compile-time preprocessor macro definition. The error reduction in $J_0(x)$ is evident in Figure 21.9 on the facing page. Plots for other precisions, and the three other Bessel functions, show similar improvements, so they are omitted.

21.10 Properties of $I_n(z)$ and $K_n(z)$

The modified Bessel functions of the first kind, $I_n(z)$, and the second kind, $K_n(z)$, have quite different behavior from the ordinary Bessel functions, $J_n(z)$ and $Y_n(z)$. Instead of taking the form of decaying waves, the modified functions

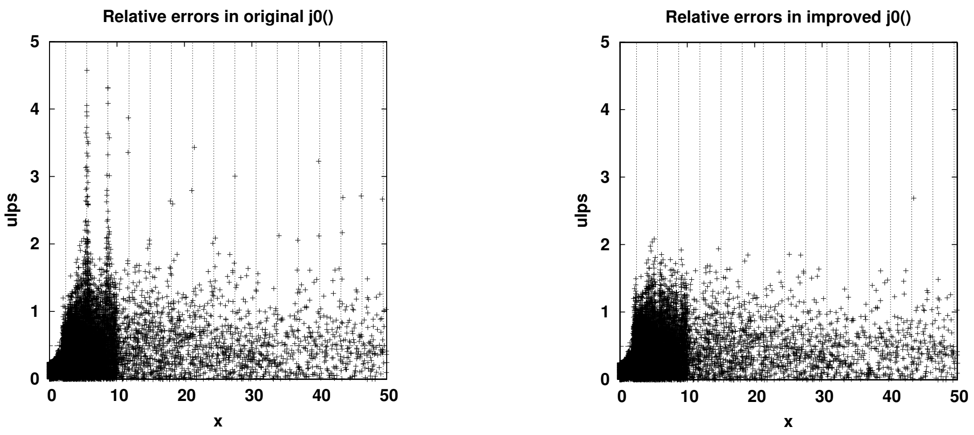


Figure 21.9: Errors in the $J_0(x)$ Bessel function before and after adding code for Taylor-series expansions around the roots indicated by the vertical dotted lines.

resemble rising and falling exponentials, as shown in **Figure 21.10** on the next page. That property makes them computationally easier than the ordinary Bessel functions, because there are no roots where high relative accuracy is difficult to achieve.

The scaled companion functions defined by

$$\mathrm{Is}_\nu(z) = \exp(-z)I_\nu(z), \qquad \mathrm{Ks}_\nu(z) = \exp(+z)K_\nu(z)$$

are finite and representable for arguments over much of the floating-point range, whereas the unscaled ones soon overflow or underflow.

Because the modified Bessel functions are not specified by POSIX or any ISO programming-language standards, we get to choose names for their unscaled and scaled software implementations in the mathcw library. Only one function in Standard C89, and four functions in C99, begin with the letter b, so we choose that letter to prefix our function names, and identify them as members of the Bessel family:

```
double bi0 (double x);              double bk0 (double x);
double bi1 (double x);              double bk1 (double x);
double bin (int n, double x);       double bkn (int n, double x);

double bis0 (double x);             double bks0 (double x);
double bis1 (double x);             double bks1 (double x);
double bisn (int n, double x);      double bksn (int n, double x);
```

They have the usual suffixed companions for other floating-point types.

The modified functions satisfy these symmetry relations:

$$\begin{aligned} I_n(-z) &= (-1)^n I_n(z), && \text{for integer } n, \\ I_{-n}(z) &= I_n(z), && \text{for integer } n, \\ K_{-\nu}(z) &= K_\nu(z), && \text{for real } \nu. \end{aligned}$$

Like $Y_\nu(z)$, the $K_\nu(z)$ functions are complex-valued for negative z. For such arguments, our software implementations therefore call QNAN("") to produce a quiet NaN as the return value, and set errno to EDOM.

Their limiting forms for small arguments, $z \to 0$, are

$$\begin{aligned} I_\nu(z) &\to (z/2)^\nu/\Gamma(\nu+1), && \text{if } \nu \neq -1, -2, -3, \ldots, \\ I_0(0) &= 1, \\ I_n(0) &= 0, && \text{for } n = \pm1, \pm2, \pm3, \ldots, \end{aligned}$$

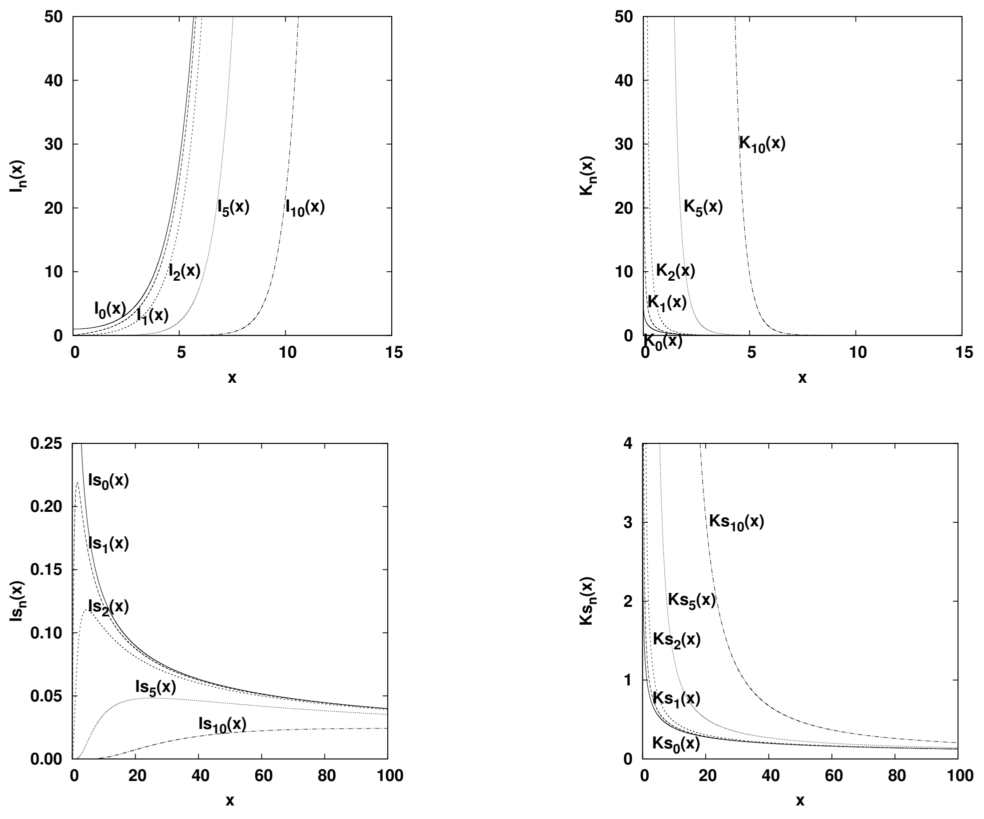


Figure 21.10: Modified Bessel functions, $I_n(x)$ and $K_n(x)$, and scaled modified Bessel functions, $e^{-x}I_n(x)$ and $e^{x}K_n(x)$.

$$K_0(z) \to -\log(z),$$
$$K_\nu(z) \to \tfrac{1}{2}\Gamma(\nu)/(z/2)^\nu, \qquad \textit{if } \nu > 0.$$

For large arguments, $z \to \infty$, the functions behave like this:

$$I_\nu(z) \to \frac{\exp(z)}{\sqrt{2\pi z}} \to +\infty,$$
$$I_n(-z) \to (-1)^n \frac{\exp(z)}{\sqrt{2\pi z}} \to (-1)^n\infty, \qquad \textit{for integer } n,$$
$$K_\nu(z) \to \exp(-z)\sqrt{\frac{\pi}{2z}} \to 0.$$

The functions satisfy these three-term recurrence relations:

$$I_{\nu+1}(z) = -(2\nu/z)I_\nu(z) + I_{\nu-1}(z),$$
$$K_{\nu+1}(z) = +(2\nu/z)K_\nu(z) + K_{\nu-1}(z).$$

They are stable in the downward direction for $I_\nu(z)$, and in the upward direction for $K_\nu(z)$. With those direction choices, there is never subtraction loss, because for real arguments and integer orders, the recurrences require only addition of positive terms.

Similar to what we observed on page 700 for $J_\nu(x)$ and $Y_\nu(x)$, when $|x| \gg \nu$, a sequence of $I_\nu(x)$ values for $\nu = 0, 1, 2, \ldots$ looks like $I_0(x)$, $I_1(x)$, $I_0(x)$, $I_1(x)$, $I_0(x)$, $I_1(x)$, $\ldots$, and similarly for sequences of $K_\nu(x)$.

Ratios of modified functions of the first kind satisfy a continued fraction similar to that for $J_\nu(z)/J_{\nu-1}(z)$ (see **Section 21.3** on page 702), except that minus signs become plus signs:

$$\begin{aligned}\frac{I_\nu(z)}{I_{\nu-1}(z)} &= 0 + \frac{1}{2\nu/z+}\ \frac{1}{2(\nu+1)/z+}\ \frac{1}{2(\nu+2)/z+}\ \frac{1}{2(\nu+3)/z+}\ \cdots\\ &= 0 + \frac{(1/2)z/\nu}{1+}\ \frac{(1/4)z^2/\nu(\nu+1)}{1+}\ \frac{(1/4)z^2/((\nu+1)(\nu+2))}{1+}\\ &\qquad \frac{(1/4)z^2/((\nu+2)(\nu+3))}{1+}\ \frac{(1/4)z^2/((\nu+3)(\nu+4))}{1+}\ \cdots.\end{aligned}$$

The same computational technique that we described for the ordinary Bessel functions of integer orders allows us to compute ratios down to $I_n(z)/I_0(z)$, and then recover $I_n(z)$ by multiplying that ratio by a separately computed $I_0(z)$. That solves the problem of the unstable upward recurrence for $I_n(z)$. Furthermore, because the ratios are invariant under uniform scaling, the continued fraction is also valid for ratios of scaled functions, $\mathrm{Is}_\nu(z)/\mathrm{Is}_{\nu-1}(z)$.

The derivatives of the modified Bessel functions

$$\begin{aligned}dI_n(x)/dx &= +I_{n+1}(x) + nI_n(x)/x,\\ dK_n(x)/dx &= -K_{n+1}(x) + nK_n(x)/x,\end{aligned}$$

lead to these error-magnification factors (see **Section 4.1** on page 61):

$$\begin{aligned}\mathrm{errmag}(I_n(x)) &= xI'_n(x)/I_n(x)\\ &= n + xI_{n+1}(x)/I_n(x),\\ \mathrm{errmag}(K_n(x)) &= n - xK_{n+1}(x)/K_n(x).\end{aligned}$$

The ratios of the Bessel functions are modest, so the general behavior of those factors is $n \pm rx$, where r is the ratio. Thus, errors in the computed functions are expected to grow almost linearly with x, and are large when either n or x is large.

The error-magnification factors of the scaled functions look like this:

$$\begin{aligned}\mathrm{errmag}(\mathrm{Is}_n(x)) &= n - x + xI_{n+1}(x)/I_n(x),\\ \mathrm{errmag}(\mathrm{Ks}_n(x)) &= n + x + xK_{n+1}(x)/K_n(x).\end{aligned}$$

Plots of those functions for fixed n and increasing x show that the first has decreasing magnitude, whereas the second grows. However, for arguments of modest size, the scaled functions should be accurately computable.

These relations between the two functions

$$\begin{aligned}I_\nu(z)K_{\nu+1}(z) + I_{\nu+1}(z)K_\nu(z) &= 1/z,\\ \mathrm{Is}_\nu(z)\,\mathrm{Ks}_{\nu+1}(z) + \mathrm{Is}_{\nu+1}(z)\,\mathrm{Ks}_\nu(z) &= 1/z,\end{aligned}$$

where the terms on the left are positive for positive orders and positive real arguments, are useful for checking software implementations.

Checks can also be made with relations to integrals that can be evaluated accurately with numerical quadrature, as long as their integrands are not too oscillatory:

$$\begin{aligned}I_0(z) &= (1/\pi)\int_0^\pi \cosh(z\cos t)\,dt,\\ I_n(z) &= (1/\pi)\int_0^\pi \exp(z\cos t)\cos(nt)\,dt,\\ K_0(z) &= -(1/\pi)\int_0^\pi \exp(\pm z\cos t)\big(\gamma + \log(2z(\sin(t))^2)\big)\,dt,\\ &= \int_0^\infty \exp(-z\cosh(t))\,dt, && \Re(z) > 0,\\ K_\nu(z) &= \int_0^\infty \exp(-z\cosh(t))\cosh(\nu t)\,dt, && \Re(z) > 0.\end{aligned}$$

There does not appear to be a similar integral for $K_n(z)$ on $[0, \pi]$.

The functions inside the infinite integrals fall off extremely rapidly. For $n = 0$, $x = 1$, and $t = 10$, the integrand is $\mathcal{O}(10^{-4782})$, so quadrature over a small finite range can be used. For example, a 40-point Simpson's rule quadrature for t on $[0, 5]$ produces 34 correct decimal digits for $K_0(1)$.

The modified Bessel function of the first kind has a series that looks like that for $J_\nu(z)$ (see **Section 21.3** on page 702), but without sign changes:

$$I_\nu(z) = (z/2)^\nu \sum_{k=0}^{\infty} \frac{(z^2/4)^k}{k!\,\Gamma(\nu+k+1)}.$$

We can again simplify the formula with the help of two intermediate variables, and also exhibit special cases for positive integer orders, and for $\nu = 0$ and $\nu = 1$:

$$\begin{aligned}
v &= z/2, \\
w &= v^2, \\
I_\nu(z) &= v^\nu \sum_{k=0}^{\infty} \frac{w^k}{k!\,\Gamma(\nu+k+1)}, \\
I_n(z) &= v^n \sum_{k=0}^{\infty} \frac{w^k}{k!\,(k+n)!}, && \text{for integer } n = 0, 1, 2, 3, \ldots, \\
I_0(z) &= \sum_{k=0}^{\infty} \frac{w^k}{(k!)^2}, \\
I_1(z) &= v \sum_{k=0}^{\infty} \frac{w^k}{k!\,(k+1)!}.
\end{aligned}$$

The modified Bessel function of the second kind has a series for integer orders that looks like that for $Y_n(z)$ (see **Section 21.3** on page 703):

$$\begin{aligned}
K_n(z) = {}& \tfrac{1}{2} v^{-n} \sum_{k=0}^{n-1} \frac{(n-k-1)!}{k!} (-w)^k - (-1)^n \log(v) I_n(z) \\
& + (-1)^n \tfrac{1}{2} v^n \sum_{k=0}^{\infty} \big(\psi(k+1) + \psi(k+n+1)\big) \frac{w^k}{k!\,(k+n)!}.
\end{aligned}$$

As with the ordinary Bessel functions, we can replace the psi functions of integer arguments by differences of the partial sums of the harmonic series, h_k (see **Section 21.3** on page 703), and the Euler–Mascheroni constant, γ:

$$\begin{aligned}
K_n(z) = {}& \tfrac{1}{2} v^{-n} \sum_{k=0}^{n-1} \frac{(n-k-1)!}{k!} (-w)^k - (-1)^n \log(v) I_n(z) \\
& + (-1)^n \tfrac{1}{2} v^n \sum_{k=0}^{\infty} (h_k + h_{k+n} - 2\gamma) \frac{w^k}{k!\,(k+n)!}.
\end{aligned}$$

We can then recognize part of the infinite sum to be $I_n(z)$, giving a further simplification:

$$\begin{aligned}
K_n(z) = {}& \tfrac{1}{2} v^{-n} \sum_{k=0}^{n-1} \frac{(n-k-1)!}{k!} (-w)^k - (-1)^n (\log(v) + \gamma) I_n(z) \\
& + (-1)^n \tfrac{1}{2} v^n \sum_{k=0}^{\infty} (h_k + h_{k+n}) \frac{w^k}{k!\,(k+n)!}.
\end{aligned}$$

The special cases for the first two orders are:

$$\begin{aligned}
K_0(z) &= -(\log(v) + \gamma) I_0(z) + \sum_{k=0}^{\infty} h_k \frac{w^k}{(k!)^2}, \\
K_1(z) &= \frac{1}{2v} + (\log(v) + \gamma) I_1(z) - \frac{v}{2} \sum_{k=0}^{\infty} (h_k + h_{k+1}) \frac{w^k}{k!\,(k+1)!}.
\end{aligned}$$

The terms in the infinite sums are all positive, providing better numerical stability than we have for the sums in $Y_n(z)$, where the terms have alternating signs. Unfortunately, there is subtraction loss when the infinite sums are added to the remaining terms. The formula for $K_0(z)$ is only stable when $z < 0.825$, and that for $K_1(z)$, when $z < 1.191$, provided that the logarithmic factor is handled properly.

The series for the modified Bessel functions of the first kind have these leading terms, suitable for use with small arguments:

$$\begin{aligned}
I_0(z) &= 1 + w + w^2/4 + w^3/36 + w^4/576 + w^5/14\,400 + \cdots, \\
I_1(z) &= v(1 + w/2 + w^2/12 + w^3/144 + w^4/2880 + w^5/86\,400 + \cdots).
\end{aligned}$$

All terms are positive, and convergence is rapid for small arguments. With the six terms shown, values of $z = 1$, $z = 1/10$, and $z = 1/1000$ produce results correct to 7, 17, and 37 decimal digits, respectively.

On systems with hexadecimal floating-point arithmetic, the series coefficients should be halved, and the final result doubled, so as to reduce accuracy loss from wobbling precision.

To simplify the series for the modified Bessel functions of the second kind, we again introduce the intermediate variable

$$s = \log(v) + \gamma.$$

To avoid loss of leading digits, s must be computed carefully as described in **Section 21.3** on page 704. We then have these formulas for fast computation for small arguments, without the need for values of other Bessel functions:

$$\begin{aligned}
K_0(z) &= -s + (1-s)w + (1/8)(-2s+3)w^2 + (1/216)(-6s+11)w^3 + \\
&\qquad (1/6912)(-12s+25)w^4 + (1/864\,000)(-60s+137)w^5 + \cdots, \\
K_1(z) &= \frac{1}{2v}(1 + (2s-1)w + (1/4)(4s-5)w^2 + (1/18)(3s-5)w^3 + \\
&\qquad (1/1728)(24s-47)w^4 + (1/86\,400)(60s-131)w^5 + \cdots).
\end{aligned}$$

The terms in those series diminish rapidly for $z < 2$. For $z = 1$, the six terms shown produce results correct to nearly six decimal digits. For $z = 1/10$, they give function values good to 16 decimal digits. For $z = 1/1000$, the results are accurate to 36 digits.

There are asymptotic expansions of the modified Bessel functions for large arguments:

$$\begin{aligned}
\mu &= 4\nu^2, \\
I_\nu(z) &\asymp \frac{\exp(z)}{\sqrt{2\pi z}}\Big(1 - \frac{\mu - 1^2}{8z} + \frac{(\mu-1^2)(\mu-3^2)}{2!\,(8z)^2} \\
&\qquad - \frac{(\mu-1^2)(\mu-3^2)(\mu-5^2)}{3!\,(8z)^3} + \cdots\Big), \\
K_\nu(z) &\asymp \sqrt{\frac{\pi}{2z}}\exp(-z)\Big(1 + \frac{\mu-1^2}{8z} + \frac{(\mu-1^2)(\mu-3^2)}{2!\,(8z)^2} \\
&\qquad + \frac{(\mu-1^2)(\mu-3^2)(\mu-5^2)}{3!\,(8z)^3} + \cdots\Big).
\end{aligned}$$

Asymptotic expansions usually limit the attainable accuracy (see **Section 2.9** on page 19), but provided that $|z| > \nu^2$, the sums can be computed to machine precision.

Although the asymptotic formulas look complex, the parenthesized sums are straightforward to compute, using the term recurrences

$$t_0 = 1, \qquad t_k = (-1)^p \frac{\mu - (2k-1)^2}{8kz} t_{k-1}, \qquad k = 1, 2, 3, \ldots,$$

where $p = 1$ for $I_\nu(z)$, and $p = 0$ for $K_\nu(z)$. For fixed ν and large z, convergence is rapid. As usual, accuracy is improved by omitting the first one or two terms, summing the remaining terms until the result has converged to machine precision, and finally, adding the omitted terms. For a hexadecimal base, the series for $K_\nu(z)$ has the undesirable form $r \times (1 + \delta)$: compute it as $2 \times (r \times (\frac{1}{2} + \frac{1}{2}\delta))$ to avoid unnecessary loss of leading bits from wobbling precision.

We use the asymptotic formulas directly for large arguments with $\nu = 0$ and $\nu = 1$. For smaller z values, we use their forms as a guide, and for the parenthesized sums, compute polynomial expansions in the variable $t = 1/z$. That way, the exponential behavior is handled entirely by the exponential functions, and the parenthesized sums are $\mathcal{O}(1)$, far from the overflow and underflow limits.

21.11 Computing $I_0(x)$ and $I_1(x)$

After much experimentation based on published algorithms for computing the modified Bessel functions of the first kind, and examination of their error plots, this author concluded that many of those recipes are inadequate, and more care is needed to achieve the accuracy expected of functions in the mathcw library.

The lack of an argument-reduction formula for the Bessel functions means that we need to handle arguments over the entire floating-point range, and that requires more intervals, each with separate polynomial approximations.

Although Maple is able to compute Chebyshev fits to formulas involving the Bessel functions over a reasonable range of arguments, high-precision minimax fits in some argument regions are infeasible. For example, Maple reports failure like this:

```
% maple
> with(numapprox):
> alias (BI = BesselI):
> BIS := proc(n, x) return exp(-x) * BI(n, x) end proc:
> Digits := 800:
> minimax((BIS(1,x) - 13/128), x = 10 .. 25, [11, 11], 1, 'maxerror'):
  Error, (in numapprox:-remez) error curve fails to oscillate
  sufficiently; try different degrees
```

That error can often be made to disappear by increasing the value of `Digits`, but its value here of 800 is the result of several earlier unsuccessful experiments with lower precisions, and the computation time at the point of failure is excessive.

The two scaled functions decay smoothly, and slowly, for $x > 2$. That suggests using fits on intervals $[a, b]$ to functions such that

$$\mathrm{Is}_n(x) = d + \begin{cases} f_1(x), \\ f_2(x)/x, \\ f_3(x)/x^2, \\ f_4(\sqrt{x})/x. \end{cases}$$

Here, the constant d is chosen to be roughly the function average on the interval, $\frac{1}{2}(\mathrm{Is}_n(a) + \mathrm{Is}_n(b))$, and the interval $[a, b]$ is selected to make the fitting function a *small* correction to d. That way, we can get results that are often correctly rounded, remedying a deficiency of most published algorithms for those functions. Because the functions decay, the correction is positive for $x \approx a$, and negative for $x \approx b$. We adjust d to a nearby value that is exactly representable in both binary and decimal `float` formats, such as $d = 25/128 = 0.195\,312\,5\dot{0}$, and so that the magnitude of a negative correction is smaller than $d/2$, preventing loss of leading bits in the subtraction.

In practice, there is little difference in the lengths of the Chebyshev expansions for $f_1(x)$, $f_2(x)$, $f_3(x)$, and $f_4(\sqrt{x})$, so we pick the simplest form, $f(x) = f_1(x) = \mathrm{Is}_n(x) - d$. A short private function then makes it easy to compute the scaled Bessel function on any of the required intervals:

```
static fp_t
evch (fp_t x, fp_t a, fp_t b, fp_t d, const int nc, const fp_t c[])
{   /* compute Is(n,x) = d + f(x), where f(x) is a Chebyshev fit */
    fp_t sum, u;

    u = (x + x - (b + a)) / (b - a);
    sum = ECHEB(u, nc, c);
    return (d + sum);
}
```

That function is used in a series of range tests, one of which looks like this:

```
    else if (x < FP(10.0))
        result = evch(x, FP(1.0), FP(10.0), FP(25.0) / FP(128.0), NC1_10, C1_10);
                     /* I0(x) = 25/128 + f(x), for x in [1,10] */
```

The symmetry relations allow us to compute only for positive arguments. For $\mathrm{Is}_1(x)$, if the input argument is negative, we must reverse the sign of the result computed with $|x|$ before returning.

After the usual checks for special arguments, the final algorithm adopted for the scaled functions `bis0(x)` and `bis1(x)` looks like this:

x **in** $(\mathbf{0}, \mathbf{small}]$: Use a *six*-term loop-free Taylor series. That is a larger term count than we usually handle for small arguments, but doing so allows a larger cutoff, and faster computation.

x **in** $(\mathbf{small}, \mathbf{1}]$: Evaluate a Chebyshev fit to the series in powers of w (see Section 21.10 on page 722).

x **in** $(\mathbf{1}, \mathbf{10}]$: Use $\mathrm{Is}_n(x) = d + f(x)$, where d is chosen as described earlier, and $f(x)$ is a Chebyshev fit.

x **in** $(\mathbf{10}, \mathbf{25}]$: Similar to previous interval, with different constant and fitting function.

x **in** $(\mathbf{25}, \mathbf{100}]$: Ditto.

x **in** $(\mathbf{100}, \mathbf{1000}]$: Ditto.

x **in** $(\mathbf{1000}, \mathbf{10^8}]$: Use $\mathrm{Is}_n(x) = \sqrt{x}\, p(1/x)$, where $p(1/x)$ is a Chebyshev fit.

x **in** $(\mathbf{10^8}, \boldsymbol{\infty})$: Sum the parenthesized asymptotic series to machine precision, where at most nine terms suffice for 70 decimal digits of accuracy. The final result is the product of that sum and `RSQRT((x + x) * PI)`, unless the base is 16, in which case, we must avoid leading zero bits in the stored constant π, so we compute the multiplier as $\sqrt{1/\pi} \times (\sqrt{\frac{1}{2}} \times \texttt{RSQRT(x)})$, where the two leading square-root constants are precomputed.

The modified Bessel functions of the first kind, `bi0(x)` and `bi1(x)`, without scaling, are computed like this:

x **in** $(\mathbf{0}, \mathbf{tiny}]$: Use a fast three-term Taylor series.

x **in** $(\mathbf{tiny}, \mathbf{5}]$: Sum the series in powers of w (see Section 21.10 on page 722). At most 43 terms are needed for 70 decimal digits of accuracy.

x **in** $(\mathbf{5}, \boldsymbol{\infty})$: The relations to the unscaled functions are

$$\texttt{bi0(x)} = \exp(x) \times \texttt{bis0(x)}, \qquad \texttt{bi1(x)} = \exp(x) \times \texttt{bis1(x)}.$$

Compute the exponential function and the scaled Bessel function, but if the exponential function overflows, compute their product indirectly from a logarithm and another exponential. For example, the code for $I_0(x)$ looks like this:

```
s = EXP(x);
t = BIS0(x);
result = (s < FP_T_MAX) ? (s * t) : EXP(x + LOG(t));
```

For large x, $\mathrm{Is}_0(x)$ is smaller than one, so there is a small region near the overflow limit of $\exp(x)$ where exact computation of $s \times t$ might produce a finite representable result, but s overflows. In such a case, we switch to the indirect form to avoid that premature overflow.

A sensibly implemented exponential function returns the largest floating-point number to indicate overflow when Infinity is not available in the floating-point design, so we compare s with that largest value, instead of calling the usual `ISINF()` wrapper.

When $x \leq 5$, no elementary or special functions are required, so the computational speed is largely determined by the number of series terms summed. That argument-dependent count is reasonably close to the values in Table 21.4 on page 703, and is less than 20 for data type `double` on most systems.

Figure 21.11 on the following page through Figure 21.15 on page 730 show the measured errors in our implementations of the modified Bessel functions of the first kind.

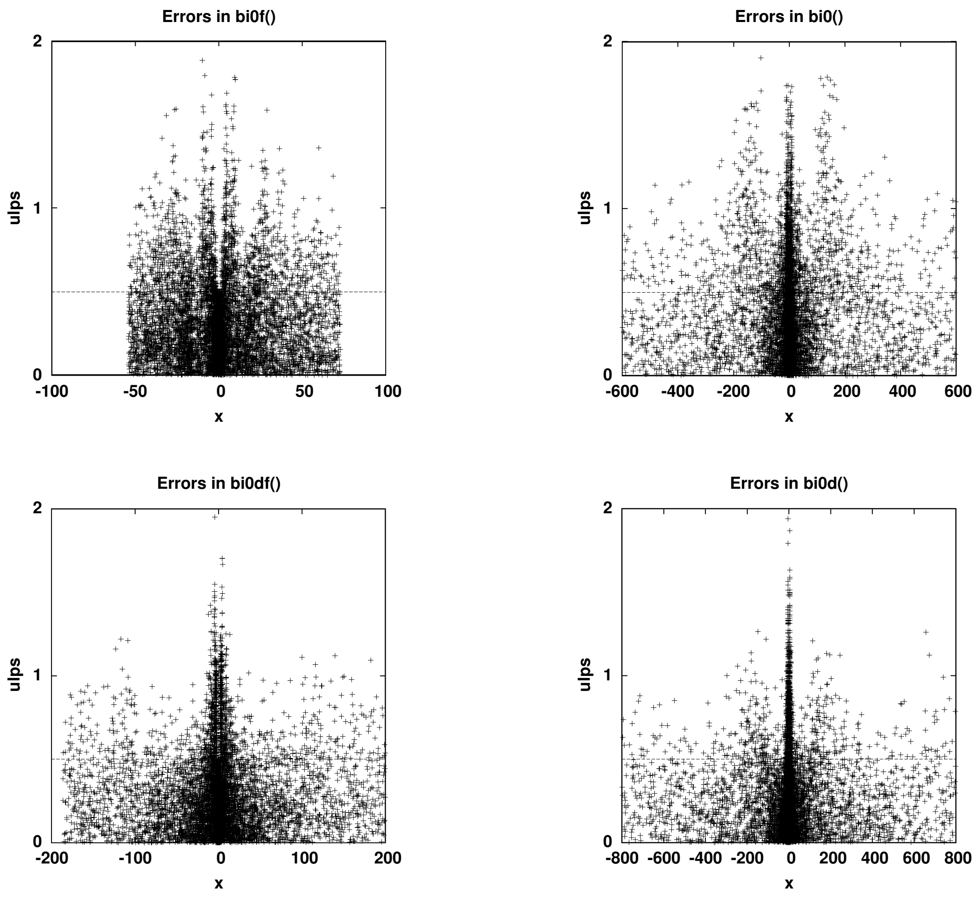


Figure 21.11: Errors in the binary (top) and decimal (bottom) `bi0(x)` family.

21.12 Computing $K_0(x)$ and $K_1(x)$

For technical reasons of exponent-size limitations, and how the `chebyshev()` function accesses the user-provided function to be fit to a Chebyshev expansion, Maple is unable to compute fits to the Bessel functions $K_0(x)$ and $K_1(x)$ for expansions in $1/x$ when $x > 10^8$. Although it seems reasonable to ask for a fit to the function $f(t) = \exp(-1/t)K_0(1/t)\sqrt{t}$, which is smooth and nearly linear for tiny t, Maple reports failure:

```
% maple
> with(numapprox):
> Digits := 30:
> chebyshev(exp(1/t) * BesselK(0, 1/t) * sqrt(t),
            t = 0 .. 1/100,
            1.0e-16);
Error, (in numapprox:-chebyshev) function does not evaluate to numeric
```

Attempts to incorporate a conditional test in the argument function to check for tiny t, and switch to the asymptotic formula, fail with the same error report.

After dealing with the usual special arguments, the final algorithm adopted for the scaled functions `bks0(x)` and `bks1(x)` follows these steps:

x in $(0, a]$: Here $a = \frac{3}{4}$ for $\mathrm{Ks}_0(x)$ and $a = 1$ for $\mathrm{Ks}_1(x)$. Evaluate the infinite sums involving harmonic-number coefficients using a Chebyshev expansion in the variable $t = x^2$, which produces fewer terms than an expan-

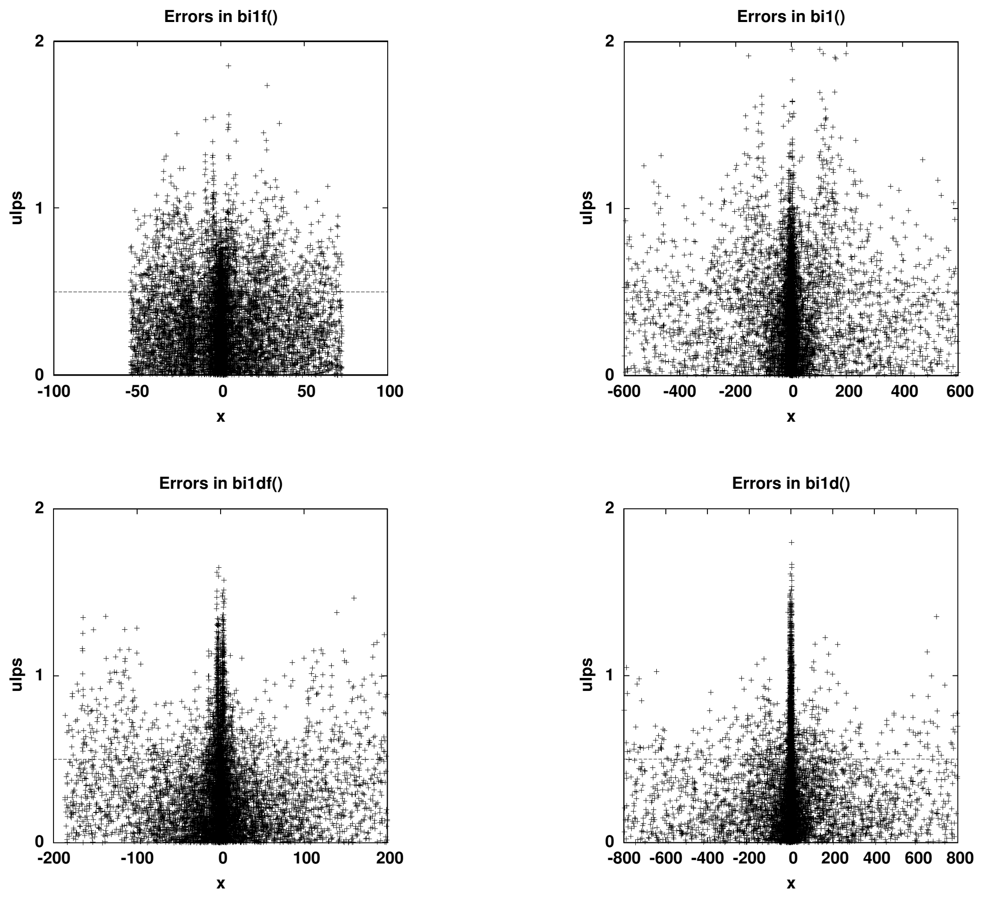


Figure 21.12: Errors in the binary (top) and decimal (bottom) `bi1(x)` family.

sion in the variable x. Add the remaining terms, computing the factor $\log(v) + \gamma$ carefully as described in Section 21.3 on page 704. Multiply the final sum by $\exp(x)$.

***x* in $(a, 5]$** : For $\mathrm{Ks}_0(x)$, use a Chebyshev fit to $(\mathrm{K}_0(t^2) - \frac{3}{4}) \times t^2$ for $t = \sqrt{x}$ in $[\frac{27}{32}, \frac{5}{4}]$. The result is the value of that expansion divided by x, plus $\frac{3}{4}$.

The fit interval is slightly larger than $(a, 5]$ to get a simple form with exact coefficients of the mapping to the Chebyshev variable interval $[-1, +1]$.

For $\mathrm{Ks}_1(x)$, use a Chebyshev fit to $(\mathrm{K}_1(x) - \frac{9}{8}) \times x$. The result is the value of that expansion divided by x, plus $\frac{9}{8}$.

***x* in $(5, 10]$** : Evaluate a Chebyshev fit to $\mathrm{Ks}_0(x) - \frac{5}{32}$ or $\mathrm{Ks}_1(x) - \frac{1}{2}$ on that interval.

***x* in $(10, 25]$** : Similar to previous interval, with different shift constants and fitting function.

***x* in $(25, 100]$** : Ditto.

***x* in $(100, 1000]$** : Ditto.

***x* in $(1000, 10^8]$** : Evaluate a Chebyshev fit with the variable $t = 1/x$.

***x* in $(10^8, \infty)$** : Sum all but the first two terms of the asymptotic expansion to machine precision, then add those two terms.

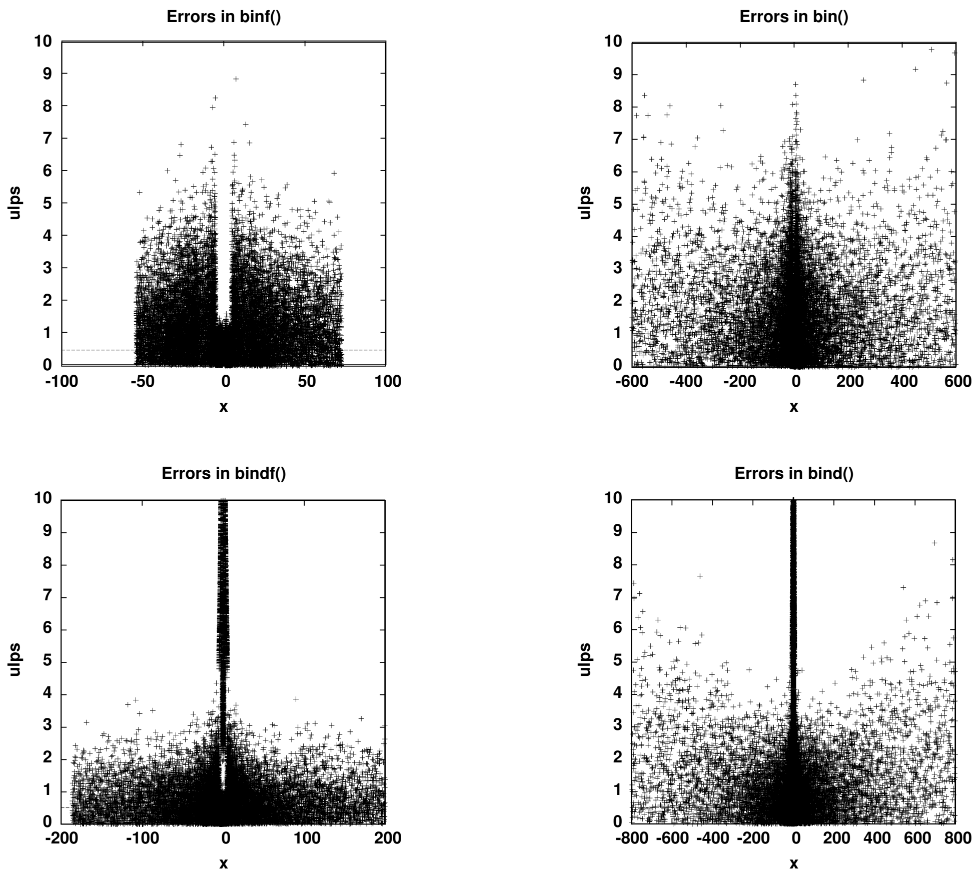


Figure 21.13: Errors in the binary (top) and decimal (bottom) `bin(n,x)` family for $n = 25$.

Although we dislike the dependence in the first region on the exponential function, and in the first two regions on the logarithm and on $I_0(x)$ or $I_1(x)$, no obvious simple functions accurately handle the approach to the singularity at $x = 0$.

We compute the unscaled functions `bk0(x)` and `bk1(x)` in two regions:

x in $(0, a]$: Same as for the scaled functions, but omit the final multiplication by $\exp(x)$.

x in $(a, \infty]$: In this region, the danger is from underflow instead of overflow. Because the scaled functions are smaller than one for large x, if $\exp(-x)$ underflows, the modified Bessel function of the second kind does as well, and the computation of the scaled function can then be avoided entirely.

Here too, when $x \le a$, there is direct and hidden dependence on the logarithm, and on $I_0(x)$ or $I_1(x)$. In both regions, there is also dependence on the scaled functions.

Figure 21.17 on page 732 through Figure 21.22 on page 737 show the measured errors in our implementations of the modified Bessel functions of the second kind. An extended vertical scale is used for the tests with $n > 1$.

21.13 Computing $I_n(x)$ and $K_n(x)$

For the modified Bessel function of the first kind for general integer order, $I_n(x)$, and its scaled companion, $\mathrm{Is}_n(x)$, after the usual handling of special arguments, and accounting for any sign change mandated by symmetry relations, we use two computational procedures in `binx.h` and `bisnx.h`:

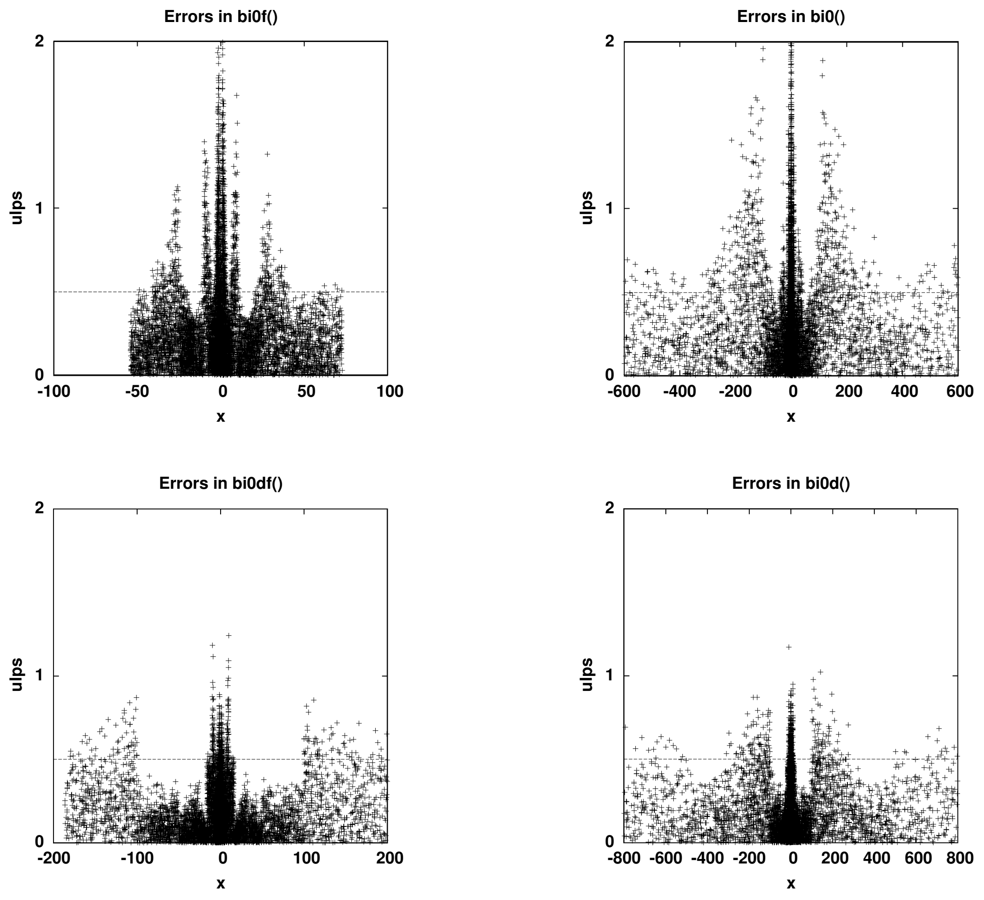


Figure 21.14: Errors in the binary (top) and decimal (bottom) `bis0(x)` family.

x **in** $[0, 5]$: Sum the series for $I_n(x)$ (see **Section 21.10** on page 722) to machine precision, adding the first two terms last to improve accuracy for small arguments. For the scaled function, multiply the result by $\exp(-x)$.

x **in** $(5, \infty)$: Evaluate the continued fraction for the ratios $I_n(x)/I_{n-1}(x)$ or $\mathrm{Is}_n(x)/\mathrm{Is}_{n-1}(x)$ (see **Section 21.10** on page 721) using the forward Lentz algorithm. Then use the downward recurrence for the ratios, as in **Section 21.6** on page 711, to find the ratio $I_n(x)/I_0(x)$ or $\mathrm{Is}_n(x)/\mathrm{Is}_0(x)$. The final result is the product of that ratio and the separately computed $I_0(x)$ from `bi0(x)`, or $\mathrm{Is}_0(x)$ from `bis0(x)`.

In both cases, computation time is proportional to the order n, so large orders are costly to compute.

Vector versions of the Bessel functions that we describe later in **Section 21.18** on page 755 compute all orders from 0 to n for fixed x using only the continued fraction algorithm, preserving the ratios in the argument vector for the final scaling by the zeroth-order function.

For the modified Bessel function of the first kind for general integer order, we consider the scaled function $\mathrm{Ks}_n(x)$ as the computational kernel. Because upward recurrence is stable, we can use starting values of `bks0(x)` and `bks1(x)` to obtain the final result needed in `bksn(n,x)`.

For the unscaled function, `bkn(x)`, there are argument regions where the factor $\exp(-x)$ underflows, and $\mathrm{Ks}_n(x)$ overflows, yet their product in exact arithmetic is finite and representable. For example, if $n = 200$ and $x = 110$, then in single-precision IEEE 754 arithmetic, $\exp(-x) \approx 10^{-48}$ underflows, and $\mathrm{Ks}_{200}(110) \approx 10^{66}$ overflows, but the exact product is $\mathcal{O}(10^{18})$, so $K_n(x)$ is representable. Similarly, both $\mathrm{Is}_{200}(110) \approx 10^{-68}$ and $\exp(+x) \approx 10^{48}$ are out of range, yet $I_{200}(110) \approx 10^{-20}$ is representable.

We cannot handle the extremes satisfactorily without intermediate scale factors, or having a separate function to

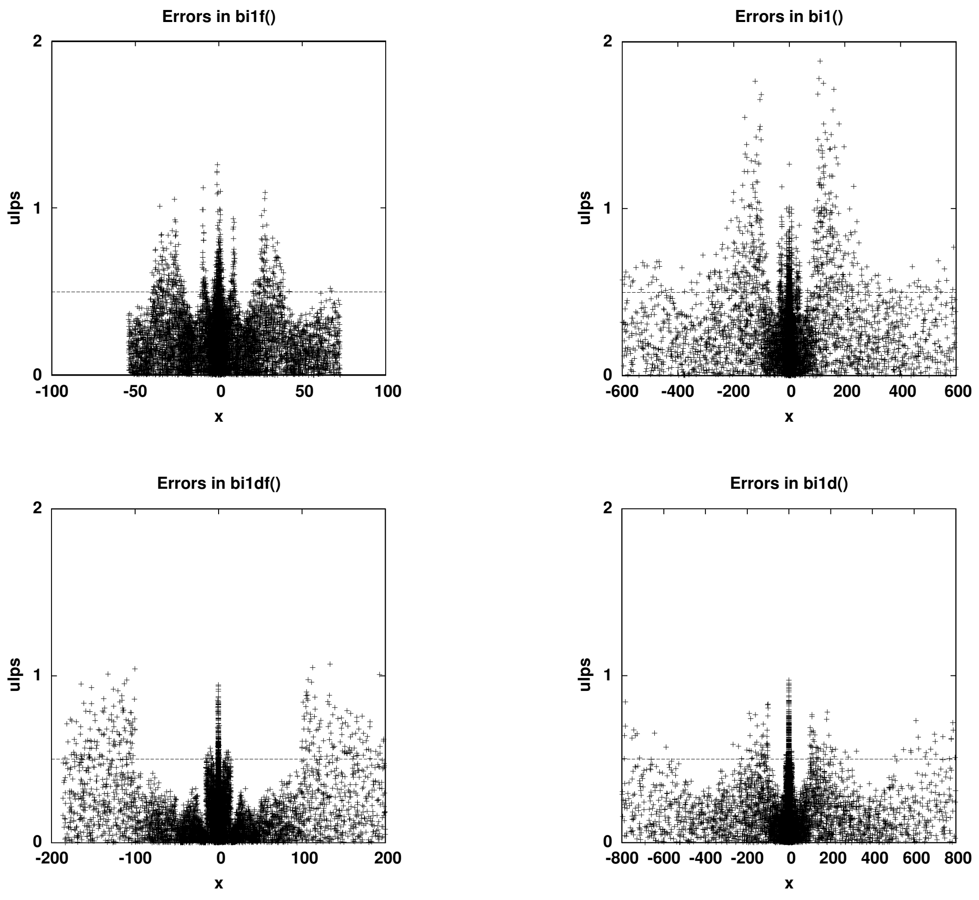


Figure 21.15: Errors in the binary (top) and decimal (bottom) `bis1(x)` family.

compute the logarithms of the scaled functions. The best that we can do is detect and report the problem through the global variable `errno`, with code like this in `bknx.h`:

```
s = EXP(-x);
t = BKSN(n, x);

if (t >= FP_T_MAX)              /* overflow in scaled function */
    result = (s == ZERO) ? SET_ERANGE(QNAN("")) : SET_ERANGE(INFTY());
else
    result = s * t;
```

The Bessel functions are good examples of the need for wider exponent ranges in floating-point designs. User code that requires Bessel functions for large orders or arguments may find it necessary to invoke versions of the functions in the highest available precision, if that format provides more exponent bits that might eliminate out-of-range intermediate results.

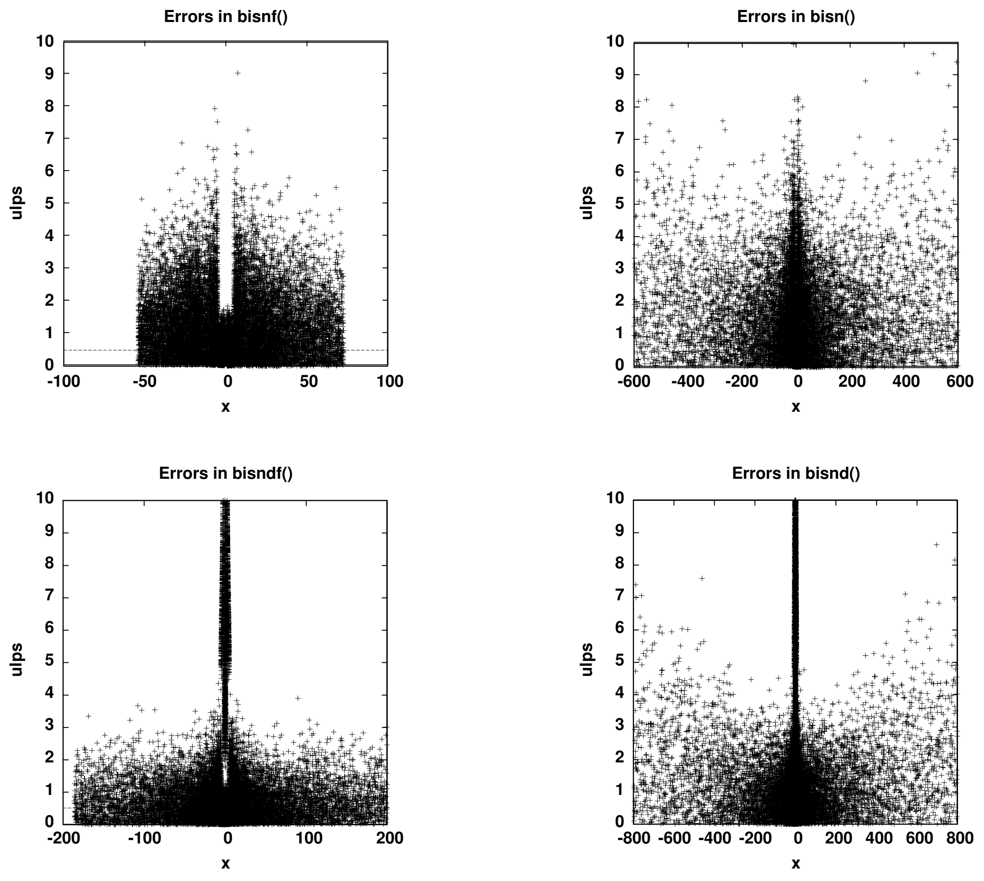


Figure 21.16: Errors in the binary (top) and decimal (bottom) `bisn(n,x)` family for $n = 25$.

21.14 Properties of spherical Bessel functions

The spherical Bessel functions are conventionally denoted by lowercase letters, and are related to the ordinary and modified Bessel functions of *half-integral order*, like this:

$$j_n(z) = \sqrt{\pi/(2z)}J_{n+\frac{1}{2}}(z), \qquad \textit{ordinary spherical Bessel function (first kind),}$$

$$y_n(z) = \sqrt{\pi/(2z)}Y_{n+\frac{1}{2}}(z), \qquad \textit{ordinary spherical Bessel function (second kind),}$$

$$i_n(z) = \sqrt{\pi/(2z)}I_{n+\frac{1}{2}}(z), \qquad \textit{modified spherical Bessel function (first kind),}$$

$$k_n(z) = \sqrt{\pi/(2z)}K_{n+\frac{1}{2}}(z), \qquad \textit{modified spherical Bessel function (second kind).}$$

Some books, including the *Handbook of Mathematical Functions* [AS64, §10.2], call $k_n(z)$ a function of the *third* kind, even though they refer to its cylindrical companion $K_n(z)$ as a function of the *second* kind. That is another instance of the lack of standardization of Bessel-function terminology.

Unlike the cylindrical Bessel functions, the spherical Bessel functions have closed forms in terms of trigonometric, hyperbolic, and exponential functions, and Table 21.6 on page 738 shows a few of them. However, the common factor $\sqrt{\pi/(2z)}$ used in most textbook presentations of those functions needs to be rewritten as $\sqrt{\frac{1}{2}\pi}/\sqrt{z}$ to agree with the closed forms. The two factors are identical for both real and complex z, *except* for negative real z, where they differ in

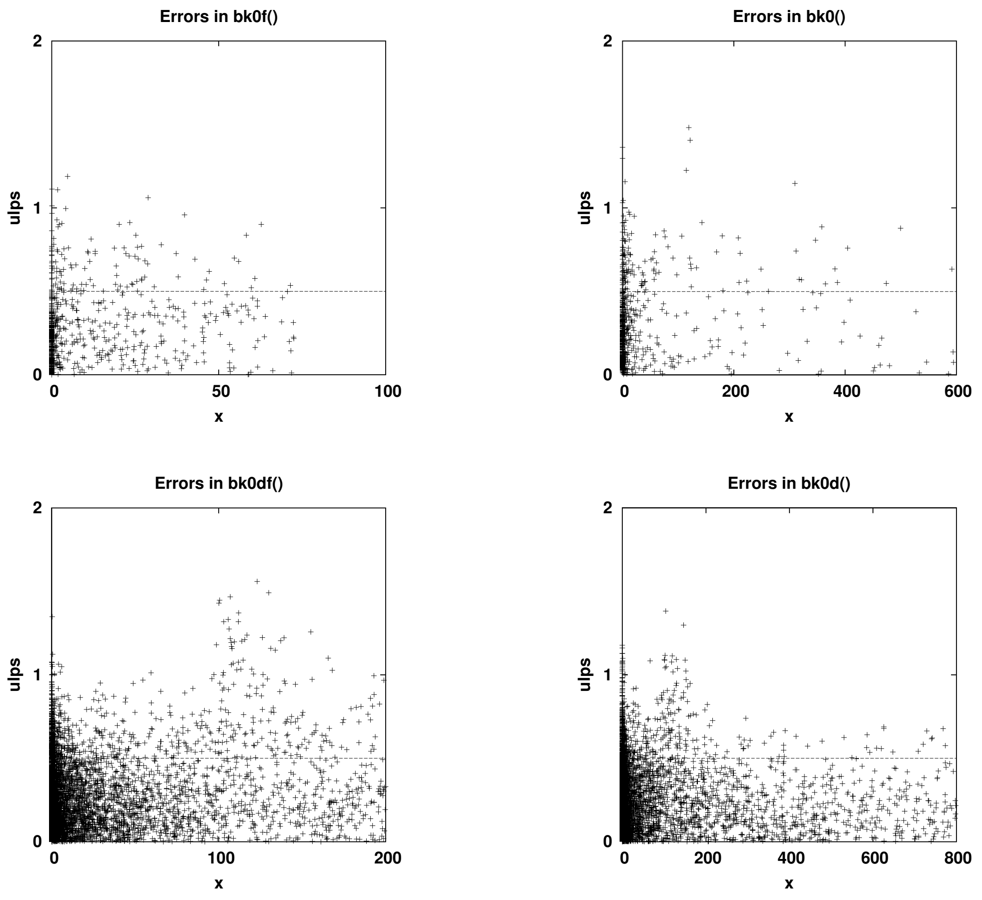


Figure 21.17: Errors in the binary (top) and decimal (bottom) `bk0(x)` family.

sign. The negative real axis here is the *branch cut* of the complex square-root function (see **Section 17.3** on page 476). Users of software for computation of spherical Bessel functions with negative arguments should be careful to check the implementation's sign conventions.

The spherical Bessel functions have these symmetry relations:

$$i_n(-z) = (-1)^n i_n(z), \qquad j_n(-z) = (-1)^n j_n(z), \qquad y_n(-z) = (-1)^{n+1} y_n(z).$$

The values $k_n(\pm z)$ do not have a simple relation, although from the tabulated closed forms, we can see that for small $|z|$, where the exponential term is nearly one and the function values are large, we have $k_n(-z) \approx (-1)^{n+1} k_n(z)$.

The modified spherical Bessel functions $i_n(z)$ grow exponentially, and the $k_n(z)$ functions fall exponentially, so scaled companions are commonly used. For large arguments, the scaled functions $\mathrm{is}_n(z)$ and $\mathrm{ks}_n(z)$ are proportional to $z^{-(n+1)}$, and $\mathrm{is}_{-n}(z)$ to z^{-n}, so they are representable in floating-point arithmetic over much of the argument range when n is small. For large n and z, they soon underflow to zero.

Although the square root in their definitions in terms of the cylindrical functions might suggest a restriction to $z \geq 0$, the spherical Bessel functions have real, rather than complex, values for both positive and negative real arguments. **Figure 21.23** on page 739 shows plots of the low-order spherical functions.

The error-magnification formulas for the spherical Bessel functions of integer order look like this:

$$\begin{aligned}
\mathrm{errmag}(i_n(x)) &= -\tfrac{1}{2} + \tfrac{1}{2}x(i_{n-1}(x) + i_{n+1}(x))/i_n(x), \\
\mathrm{errmag}(j_n(x)) &= -\tfrac{1}{2} + \tfrac{1}{2}x(j_{n-1}(x) - j_{n+1}(x))/j_n(x), \\
\mathrm{errmag}(k_n(x)) &= -\tfrac{1}{2} - \tfrac{1}{2}x(k_{n-1}(x) + k_{n+1}(x))/k_n(x),
\end{aligned}$$

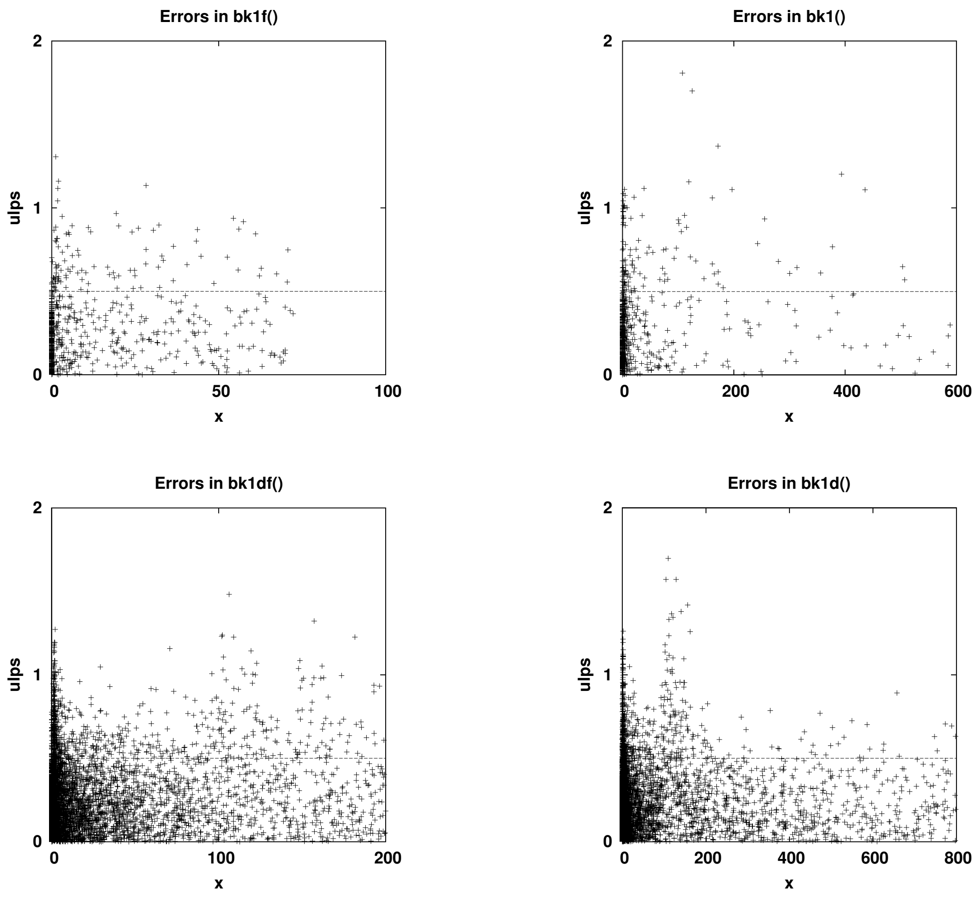


Figure 21.18: Errors in the binary (top) and decimal (bottom) `bk1(x)` family.

$$\text{errmag}(y_n(x)) = -\tfrac{1}{2} + \tfrac{1}{2}x(y_{n-1}(x) - y_{n+1}(x))/y_n(x).$$

Plots of those formulas for various n and modest ranges of x show that errors grow roughly linearly in x for $i_n(x)$ and $k_n(x)$. For the other two, the error factor lies in $[-n, n]$, and away from the zeros of the function, but, of course, grows without bound near those zeros.

The first spherical Bessel function, $j_0(z) = \sin(z)/z$, is identical to the sinc function, `sinc(z)`, which has important applications in approximation theory [LB92, Ste93, KSS95]. Considering its simple form, further mathematical development of the theory of the sinc function is surprisingly complex, but its rewards are rich, leading to rapidly convergent approximations that can produce results of arbitrarily high precision. However, we do not consider the sinc-function approach further in this book.

Because the recurrence relations for the ordinary and modified Bessel functions are valid for arbitrary order ν, the spherical Bessel functions of integer order have similar relations:

$$\begin{aligned}
j_{n+1}(z) &= ((2n+1)/z)j_n(z) - j_{n-1}(z),\\
y_{n+1}(z) &= ((2n+1)/z)y_n(z) - y_{n-1}(z),\\
i_{n+1}(z) &= (-(2n+1)/z)i_n(z) + i_{n-1}(z),\\
k_{n+1}(z) &= ((2n+1)/z)k_n(z) + k_{n-1}(z).
\end{aligned}$$

Those relations are numerically stable in the upward direction for $y_{n+1}(z)$ and $k_{n+1}(z)$, and in the downward direction for the other two.

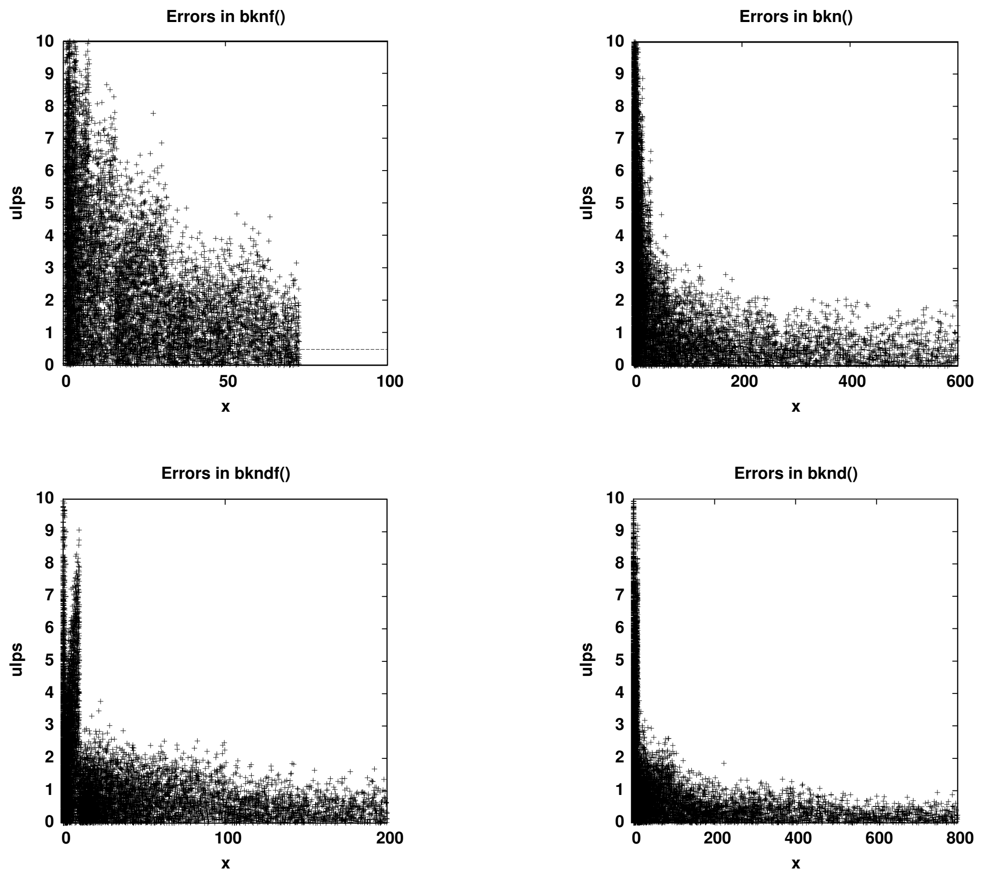


Figure 21.19: Errors in the binary (top) and decimal (bottom) `bkn(n,x)` family for $n = 25$.

As long as the implementations of the trigonometric functions employ *exact argument reduction*, as those in the mathcw library do, $j_0(z)$ and $y_0(z)$ can be computed accurately from the formulas of **Table 21.6** on page 738 for *any* representable value z. However, the higher-order functions $j_n(z)$ and $y_n(z)$ suffer serious subtraction loss when $n > 0$. For real arguments, the $k_n(x)$ functions are well-behaved for all n and $x \geq 0$ because all terms are positive, and the $i_n(x)$ functions have a dominant term for large x, so they too are computationally reasonable. We investigate the stability of the computation of $k_n(x)$ for negative arguments later when we develop computer algorithms for those functions in **Section 21.17.7** on page 754.

Three of the spherical Bessel functions of negative order are simply related to those of positive order:

$$j_{-n}(z) = (-1)^n y_{n-1}(z), \qquad y_{-n}(z) = (-1)^{n+1} j_{n-1}(z), \qquad k_{-n}(z) = k_{n-1}(z).$$

The function $i_{-n}(z)$ does not have a simple relation to $i_{n-1}(z)$, because the hyperbolic cosine and sine are exchanged in the closed forms shown in **Table 21.6** on page 738. However, is z is large, $\cosh(z) \approx \sinh(z)$, so we can conclude that when $z \gg 1$, $i_{-n}(z) \approx i_{n-1}(z)$. For example, $i_{-4}(10)$ and $i_3(10)$ agree to eight decimal digits.

The case of $n < 0$ for $i_n(z)$ is most easily handled by the stable downward recurrence starting from $i_1(z)$ and $i_0(z)$.

The spherical Bessel functions have the limiting behaviors summarized in **Table 21.7** on page 740. The argument of the sine function in the large-argument limit for $j_n(z)$ cannot be determined accurately when n is large unless high precision is available, but the angle sum formula allows it to be replaced by $\sin(z)\cos(n\pi/2) - \cos(z)\sin(n\pi/2)$, and because n is an integer, that collapses to one of $\pm\sin(z)$ or $\pm\cos(z)$. Similar considerations allow accurate reduction of the cosine in the large-argument limit for $y_n(z)$.

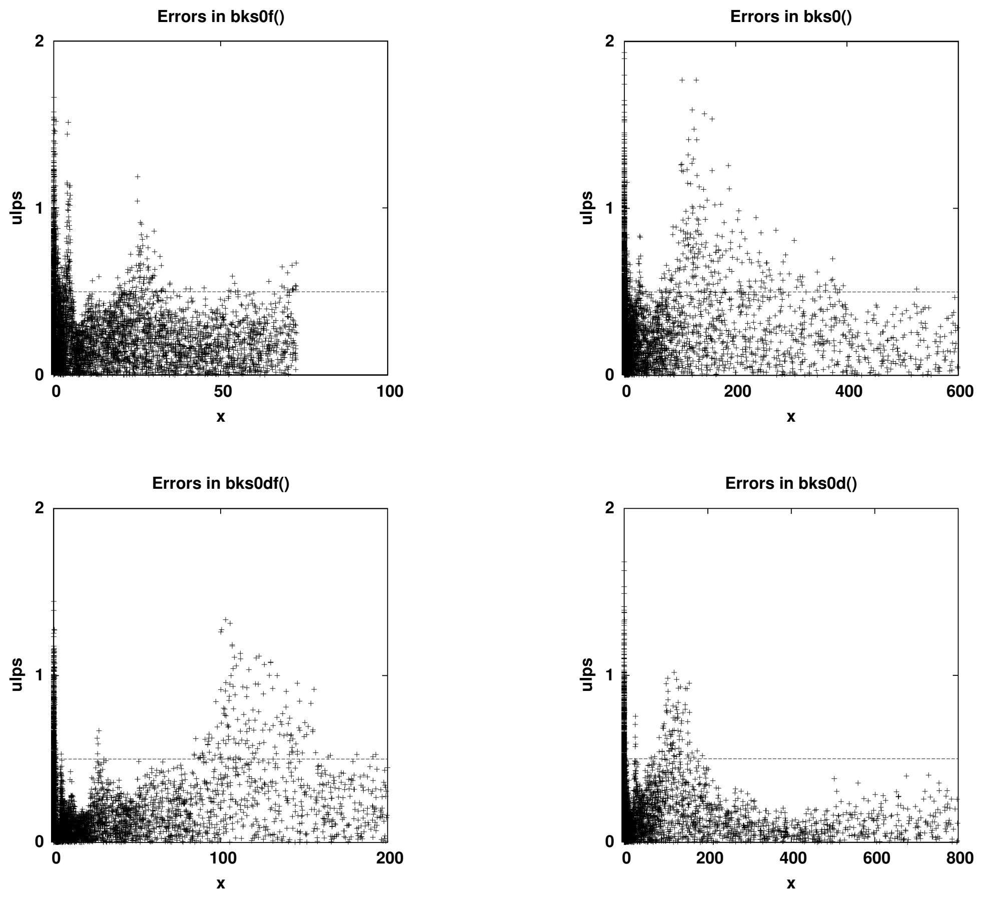


Figure 21.20: Errors in the binary (top) and decimal (bottom) `bks0(x)` family.

21.15 Computing $j_n(x)$ and $y_n(x)$

Because the spherical Bessel functions of the first and second kinds have simple relations to their cylindrical companions, the same computational techniques can be used for both types. However, the closed forms $j_0(x) = \sin(x)/x$ and $y_0(x) = -\cos(x)/x$, together with our exact argument reduction inside the trigonometric functions, suggest the use of those formulas, except for small arguments, where truncated Taylor series provide a faster route. The needed series are trivially obtained from those of the trigonometric functions, so that we have

$$j_0(x) \approx 1 - x^2/6 + x^4/120 - x^6/5040 + x^8/362\,880 - \cdots,$$
$$y_0(x) \approx -\frac{1}{x}(1 - x^2/2 + x^4/24 - x^6/720 + x^8/40\,320 - \cdots).$$

The closed forms $j_1(x) = (-x\cos(x) + \sin(x))/x^2$ and $y_1(x) = -(\cos(x) + x\sin(x))/x^2$ look simple, but suffer massive subtraction loss near the zeros of those functions. For small arguments, series expansions solve the accuracy-loss problem:

$$j_1(x) \approx \frac{x}{3}(1 - x^2/10 + x^4/280 - x^6/15\,120 + x^8/1\,330\,560 - \cdots),$$
$$y_1(x) \approx -\frac{1}{x^2}(1 + x^2/2 - x^4/8 + x^6/144 - x^8/5760 + \cdots).$$

In binary floating-point arithmetic, it is advisable to compute the series for $j_1(x)$ as $x/4 + (x/12 - x^3/30 + \cdots)$, so that the first term is exact, and the second term is a small correction. For other bases, the error from the division by

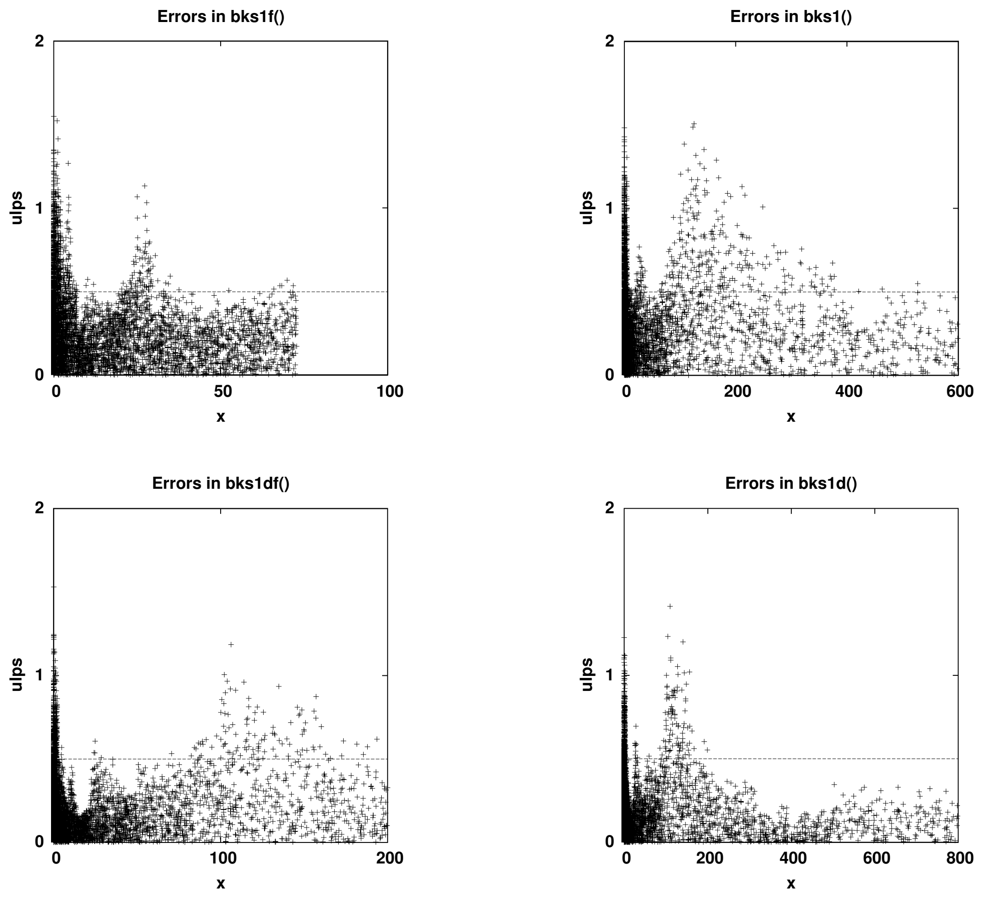


Figure 21.21: Errors in the binary (top) and decimal (bottom) `bks1(x)` family.

three can be reduced if higher precision is used.

By techniques discussed later in Section 21.17.5 on page 750, we can find the general form of the Taylor series for arbitrary integer $n \geq 0$:

$$\begin{aligned}
j_n(x) = {}& \frac{x^n}{(2n+1)!!}(1 - \frac{1}{2(2n+3)}x^2 + \frac{1}{8(2n+3)(2n+5)}x^4 - \\
& \frac{1}{48(2n+3)(2n+5)(2n+7)}x^6 + \\
& \frac{1}{384(2n+3)(2n+5)(2n+7)(2n+9)}x^8 - \cdots), \\
y_n(x) = {}& -\frac{(2n-1)!!}{x^{n+1}}(1 + \frac{1}{2(2n-1)}x^2 + \frac{1}{8(2n-1)(2n-3)}x^4 + \\
& \frac{1}{48(2n-1)(2n-3)(2n-5)}x^6 + \\
& \frac{1}{384(2n-1)(2n-3)(2n-5)(2n-7)}x^8 + \cdots).
\end{aligned}$$

The series for $j_n(x)$ is free of leading bit loss only for $x < \sqrt{2n+3}$, so the minimal cutoff for using the series with $n \geq 2$ in a computer program is $x_{\text{TS}} = \sqrt{7}$. Although it might not be immediately evident, the terms in the series for

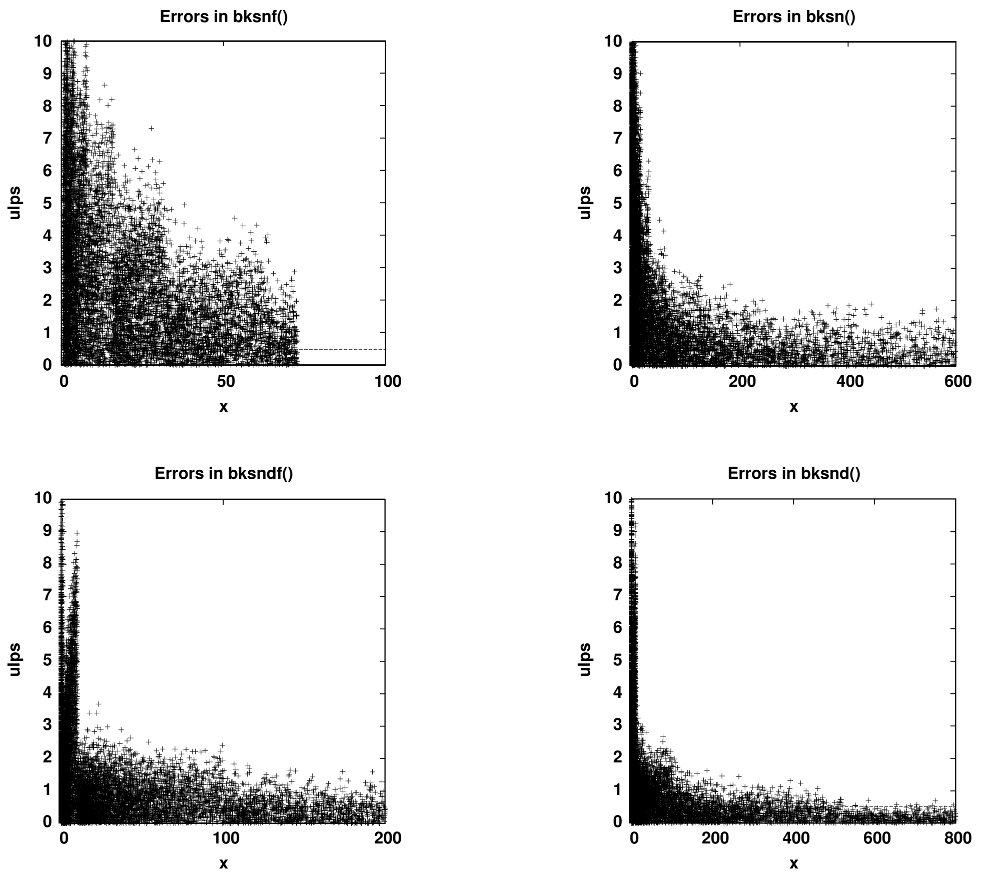


Figure 21.22: Errors in the binary (top) and decimal (bottom) `bksn(n,x)` family for $n = 25$.

$y_n(x)$ can also alternate in sign. For $n = 2$, the term containing x^6 is negative, and subtraction loss is prevented by using the series only for $x < \sqrt{3}$.

The series expansions of $j_n(x)$ and $y_n(x)$ lead to simple recurrence formulas for successive terms:

$$t_0 = 1, \qquad t_k = -\left(\frac{1}{2k}\right)\left(\frac{1}{2n+2k+1}\right) x^2\, t_{k-1}, \qquad \text{for } k = 1, 2, 3, \ldots.,$$

$$u_0 = 1, \qquad u_k = \left(\frac{1}{2k}\right)\left(\frac{1}{2n+1-2k}\right) x^2\, u_{k-1}, \qquad \text{for } k = 1, 2, 3, \ldots,$$

$$j_n(x) = \frac{x^n}{(2n+1)!!}(t_0 + t_1 + t_2 + \cdots),$$

$$y_n(x) = -\frac{(2n-1)!!}{x^{n+1}}(u_0 + u_1 + u_2 + \cdots).$$

Certainly for $|x| < 1$, and also for $n \gg x^2$, the terms fall off so rapidly that there is no subtraction loss when they alternate in sign. Provided that the terms are accumulated in order of increasing magnitudes, the major source of inaccuracy in all but $j_0(x)$ is from the outer multiplication and division. We can further reduce rounding error by factoring out x^2 in the sums after the first term so that, for example, we compute

$$y_1(x) \approx -\left(\frac{1}{x^2} + (1/2 - x^2/8 + x^4/144 - x^6/5760 + \cdots)\right).$$

The widest interval where the closed form for $j_1(x)$ loses leading bits is $[0, 1.166]$, and that for $y_1(x)$ is $[2.458,$

Table 21.6: Explicit forms of low-order spherical Bessel functions, with their order-symmetry relations in **bold**.

$j_0(z) = \sin(z)/z$	$i_0(z) = \sinh(z)/z$
$j_1(z) = (-z\cos(z) + \sin(z))/z^2$	$i_1(z) = (z\cosh(z) - \sinh(z))/z^2$
$j_2(z) = (-3z\cos(z) - (z^2 - 3)\sin(z))/z^3$	$i_2(z) = (-3z\cosh(z) + (z^2 + 3)\sinh(z))/z^3$
$j_3(z) = ((z^3 - 15z)\cos(z) - (6z^2 - 15)\sin(z))/z^4$	$i_3(z) = ((z^3 + 15z)\cosh(z) - (6z^2 + 15)\sinh(z))/z^4$
$y_0(z) = -\cos(z)/z$	$k_0(z) = (\pi/(2z))\exp(-z)$
$y_1(z) = (-\cos(z) - z\sin(z))/z^2$	$k_1(z) = (\pi/(2z^2))(z + 1)\exp(-z)$
$y_2(z) = ((z^2 - 3)\cos(z) - 3z\sin(z))/z^3$	$k_2(z) = (\pi/(2z^3))(z^2 + 3z + 3)\exp(-z)$
$y_3(z) = ((6z^2 - 15)\cos(z) + (z^3 - 15z)\sin(z))/z^4$	$k_3(z) = (\pi/(2z^4))(z^3 + 6z^2 + 15z + 15)\exp(-z)$
$j_{-1}(z) = \cos(z)/z$	$i_{-1}(z) = \cosh(z)/z$
$j_{-2}(z) = (-\cos(z) - z\sin(z))/z^2$	$i_{-2}(z) = (-\cosh(z) + z\sinh(z))/z^2$
$j_{-3}(z) = -((z^2 - 3)\cos(z) - 3z\sin(z))/z^3$	$i_{-3}(z) = ((z^2 + 3)\cosh(z) - 3z\sinh(z))/z^3$
$j_{-4}(z) = ((6z^2 - 15)\cos(z) + (z^3 - 15z)\sin(z))/z^4$	$i_{-4}(z) = ((-6z^2 - 15)\cosh(z) + (z^3 + 15z)\sinh(z))/z^4$
$\boldsymbol{j_{-n}(z) = (-1)^n y_{n-1}(z)}$	
$y_{-1}(z) = \sin(z)/z$	$k_{-1}(z) = (\pi/(2z))\exp(-z)$
$y_{-2}(z) = -(-z\cos(z) + \sin(z))/z^2$	$k_{-2}(z) = (\pi/(2z^2))(z + 1)\exp(-z)$
$y_{-3}(z) = (-3z\cos(z) - (z^2 - 3)\sin(z))/z^3$	$k_{-3}(z) = (\pi/(2z^3))(z^2 + 3z + 3)\exp(-z)$
$y_{-4}(z) = -((z^3 - 15z)\cos(z) - (6z^2 - 15)\sin(z))/z^4$	$k_{-4}(z) = (\pi/(2z^4))(z^3 + 6z^2 + 15z + 15)\exp(-z)$
$\boldsymbol{y_{-n}(z) = (-1)^{n+1} j_{n-1}(z)}$	$\boldsymbol{k_{-n}(z) = k_{n-1}(z)}$

2.975]. We could use polynomial approximations in those regions, but that provides only a partial solution to a problem that occurs uncountably often for larger x values. The code in `sbj1x.h` uses a rational polynomial fit in the first loss region.

For larger arguments, however, the approach in most existing implementations of $j_1(x)$ and $y_1(x)$ is to suffer the subtraction loss from straightforward application of the closed formulas, which means accepting small absolute error, rather than small relative error. That does not meet the accuracy goals of the `mathcw` library, and because no obvious rearrangement of the closed forms for arguments $|x| > 1$ prevents the subtraction loss analytically, the only recourse is then to use higher precision, when available. In practice, that means that the `float` functions can achieve high accuracy, and on some systems, the `double` functions as well. However, the relative accuracy for longer data types can be expected to be poor near the function zeros.

Because two trigonometric functions are usually needed for each of $j_1(x)$ and $y_1(x)$, some libraries compute both spherical Bessel functions simultaneously. Ours does not, but we use the `SINCOS()` function family to get the sine and the cosine with a single argument reduction, and little more than the cost of just one of the trigonometric functions.

For sufficiently large x, the terms containing the factor x in $j_1(x)$ and $y_1(x)$ dominate, and we can avoid one of the trigonometric functions, computing $j_1(x) \approx -\cos(x)/x$, and $y_1(x) \approx -\sin(x)/x$. A suitable cutoff for $j_1(x)$ is found by setting $\sin(x) \approx 1$ and $\cos(x) \approx \epsilon$ (the machine epsilon), for which we have $j_1(x) \approx (-x\epsilon + 1)/x^2$, and 1 is negligible in that sum if $1/(x\epsilon) < \frac{1}{2}\epsilon/\beta$. We solve that to find the cutoff $x_c = 2\beta/\epsilon^2 = 2\beta^{2t-1}$, where t is the number of base-β digits in the significand. The same cutoff works for $y_1(x)$ too.

An alternate approach is to evaluate the continued fraction for the ratio $j_1(x)/j_0(x)$ (see **Section 21.6** on page 710), and then determine $j_1(x)$ from the product of that ratio with an accurate value of $j_0(x)$. Numerical experiments with that technique show that the errors are higher than with direct use of the trigonometric closed forms, even when those forms cannot be computed in higher precision.

For orders $n \geq 2$, $j_n(x)$ is computed using the same algorithm as for $J_n(x)$: series summation for small arguments, and downward recurrence with the continued fraction to find the ratio $j_n(x)/j_0(x)$, from which $j_n(x)$ can be easily found.

Unfortunately, the continued fraction for the Bessel function ratios converges poorly for large arguments: the

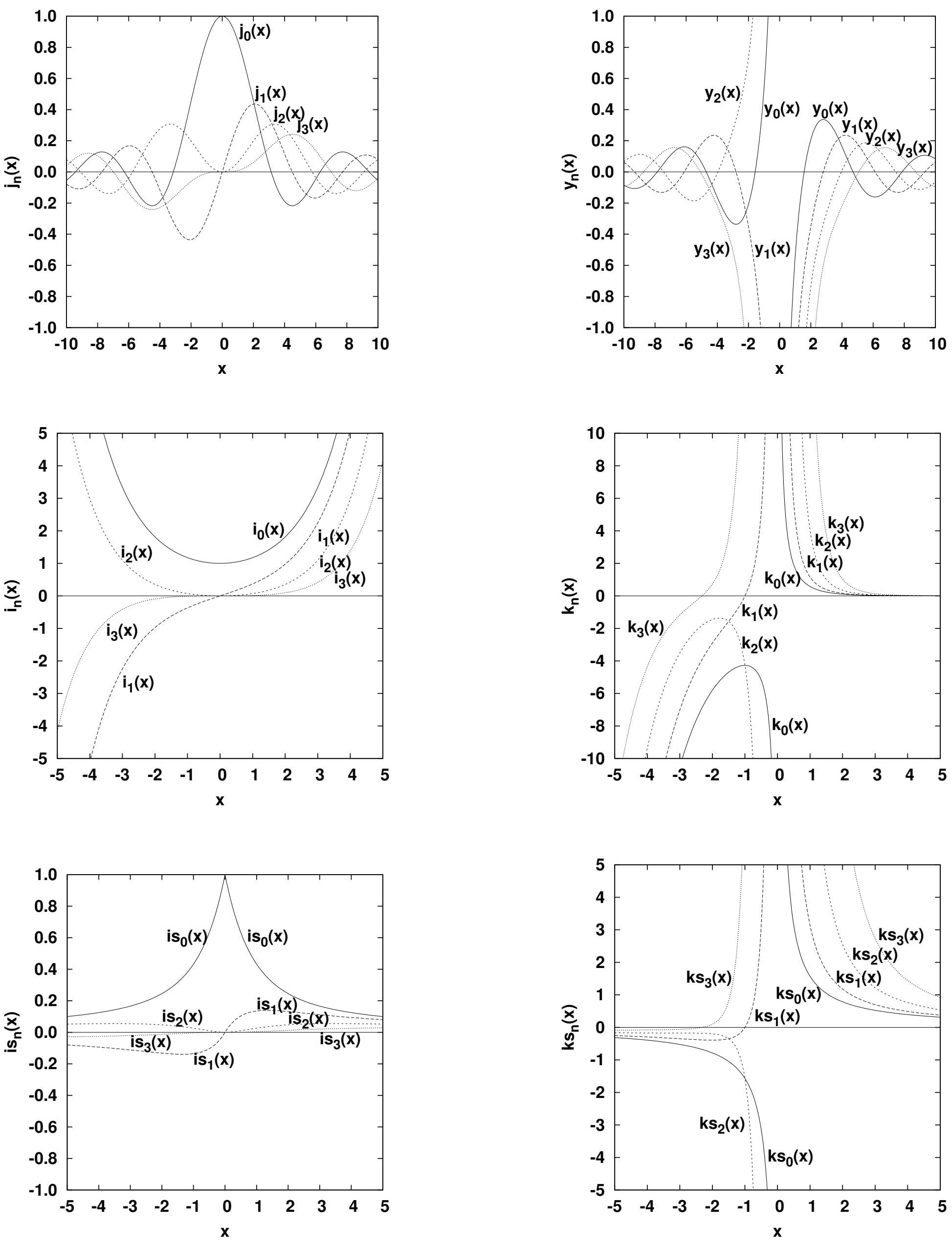


Figure 21.23: Spherical Bessel functions, $j_n(x)$ and $y_n(x)$, modified spherical Bessel functions, $i_n(x)$ and $k_n(x)$, and scaled modified spherical Bessel functions, $e^{-|x|}i_n(x)$ and $e^x k_n(x)$.

number of iterations required is roughly $\lfloor x \rfloor$ (see **Section 6.7** on page 136). It is then better to switch to the formulas suggested by the asymptotic relations involving the functions $\mathrm{P}(\nu, x)$ and $\mathrm{Q}(\nu, x)$ that we introduced in **Section 21.3** on page 698.

For $n \geq 2$, $y_n(x)$ can be computed stably by upward recurrence, or when $(n + \frac{1}{2})^2 < |x|$, by the asymptotic expansion. However, if x lies near a zero of $y_1(x)$, namely, the values for which $x \sin(x) = \cos(x)$, or $x = \cot(x)$, then as we noted, $y_1(x)$ may be inaccurate. Its error may then contaminate all higher $y_n(x)$ values computed with the upward recurrence. The only simple solution to that problem is to use higher-precision arithmetic for the recurrence,

Table 21.7: Limiting values of spherical Bessel functions.

$j_0(0) = 1,$	
$j_n(0) = 0,$	$n > 0,$
$j_n(z) \to (1/z)\sin(z - n\pi/2),$	$\lvert z\rvert \to \infty,$
$j_n(z) \to z^n/(2n+1)!!,$	$\lvert z\rvert \to 0,$
$y_n(0) = -\infty,$	*for all integer* $n \geq 0,$
$y_n(z) \to -(2n-1)!!/z^{n+1},$	$\lvert z\rvert \to 0,$
$y_n(z) \to -(1/z)\cos(z - n\pi/2),$	$\lvert z\rvert \to \infty,$
$i_0(0) = 1,$	
$i_n(0) = 0,$	*for all integer* $n > 0,$
$i_n(z) \to z^n/(2n+1)!!,$	$\lvert z\rvert \to 0,$
$i_{-n}(z) \to (-1)^{n+1}(2n-3)!!/z^n,$	$\lvert z\rvert \to 0,$
$i_n(z) \to \exp(z)/(2z),$	$\lvert z\rvert \to \infty,$
$k_n(0) = +\infty,$	
$k_n(z) \to (\frac{1}{2}\pi)(2n-1)!!/z^{n+1},$	$\lvert z\rvert \to 0,$
$k_n(z) \to (\frac{1}{2}\pi/z)\exp(-z),$	$\lvert z\rvert \to \infty.$

but that is not possible at the highest-available precision.

Figure 21.24 through Figure 21.29 on page 746 show the measured errors in our implementations of the ordinary spherical Bessel functions.

21.16 Improving $j_1(x)$ and $y_1(x)$

In Section 21.9 on page 716, we showed how to use a symbolic-algebra system to find the general form of the Taylor-series expansions near the zeros of low-order ordinary cylindrical Bessel functions. We can do the same for the spherical Bessel functions, but we can omit the order-zero functions because their closed forms can be evaluated accurately near all of their zeros, as long as exact trigonometric argument reduction is available, as it is in the mathcw library.

The Maple files `sbj1taylor.map` and `sby1taylor.map` generate the expansions around a particular zero, z:

$$j_1(z+d) = a_0 + a_1 d + a_2 d^2 + a_3 d^3 + a_4 d^4 + \cdots,$$
$$y_1(z+d) = b_0 + b_1 d + b_2 d^2 + b_3 d^3 + b_4 d^4 + \cdots.$$

Four intermediate variables

$$v = 1/z, \qquad w = v^2, \qquad c = \cos(z), \qquad s = \sin(z),$$

simplify the coefficients, the first few of which look like this:

$$\begin{aligned}
a_0 &= 0,\\
a_1 &= (((1-2w)s + 2cv)v)/1,\\
a_2 &= (-((3-6w)vs + (-1+6w)c)v)/2,\\
a_3 &= (((-1+(12-24w)w)s + (-4+24w)vc)v)/6,\\
a_4 &= (-((-5+(60-120w)w)vs + (1+(-20+120w)w)c)v)/24,\\
b_0 &= 0,
\end{aligned}$$

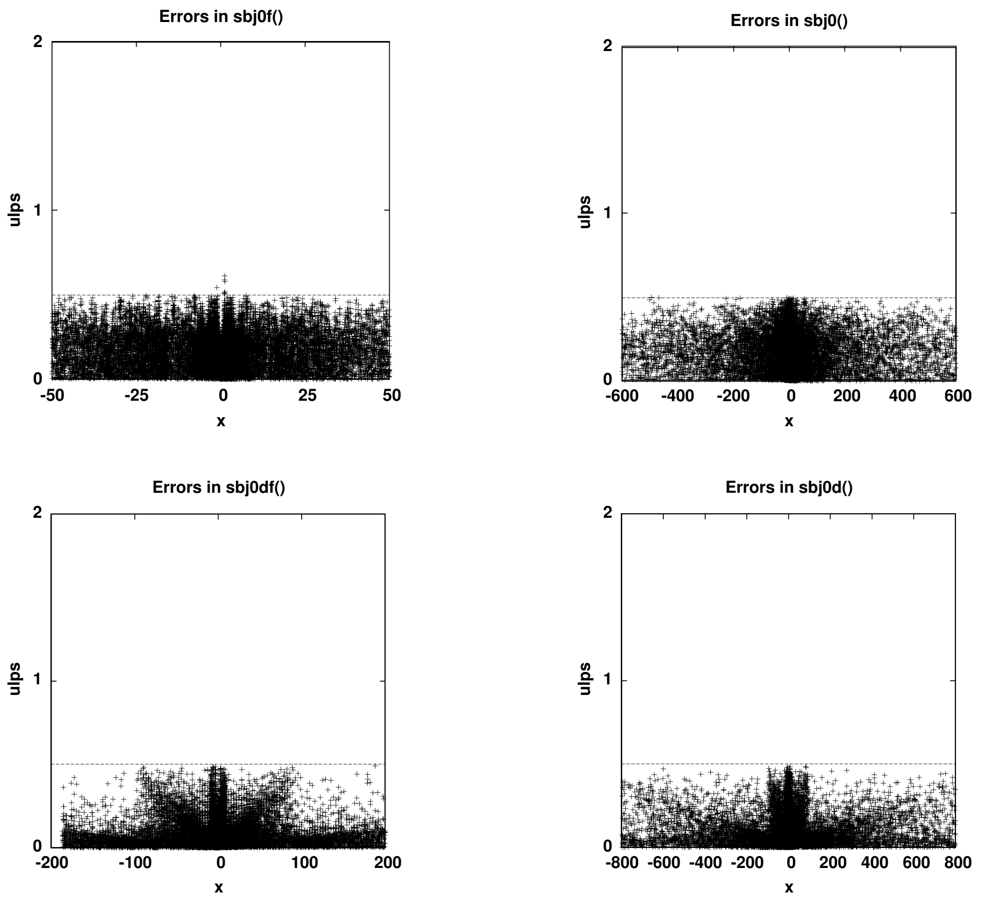


Figure 21.24: Errors in the binary (top) and decimal (bottom) `sbj0(x)` family.

$$\begin{aligned}
b_1 &= ((2sv + (-1 + 2w)c)v)/1,\\
b_2 &= (-((-1 + 6w)s + (-3 + 6w)vc)v)/2,\\
b_3 &= (((-4 + 24w)vs + (1 + (-12 + 24w)w)c)v)/6,\\
b_4 &= (-((1 + (-20 + 120w)w)s + (5 + (-60 + 120w)w)vc)v)/24.
\end{aligned}$$

Our improved code for `sbj1(x)` and `sby1(x)` computes the coefficients up to $k = 17$, and then evaluates the polynomials in Horner form using the `QFMA()` wrapper.

Sign alternations in the Taylor-series expansions may lead to loss of leading digits when the series are summed, and digit loss is also possible in the computation of the individual coefficients a_k and b_k. The maximum size of $|d|$ for which the expansions can be used without digit loss depends on z, and is best determined by high-precision numerical experiment. The two Maple programs do just that, and their output shows that loss is almost always most severe in the sums $a_2d^2 + a_3d^3$ and $b_2d^2 + b_3d^3$. Tables of the first 318 zeros handle $x < 1000$, for which a cutoff of $|d| < 0.003$ suffices for both functions. In practice, a somewhat larger cutoff, such as $|d| < 0.05$, could probably be used, because the terms for $k = 1$ dominate the sums.

As in the improved code for the cylindrical functions, we tabulate the zeros as pair sums of exact high and accurate low parts so that d can be determined to machine precision from $(x - z_{\text{hi}}) - z_{\text{lo}}$. We could also tabulate the values of the sine and cosine at the zeros, but instead conserve storage at the expense of somewhat longer compute times when $|d|$ is small enough for the series to be used. In addition, we suppress the use of the Taylor expansions entirely when the number of digits in the `hp_t` data type is at least twice that in the `fp_t` data type, because our normal algorithm then has sufficient precision to provide low relative error near the zeros.

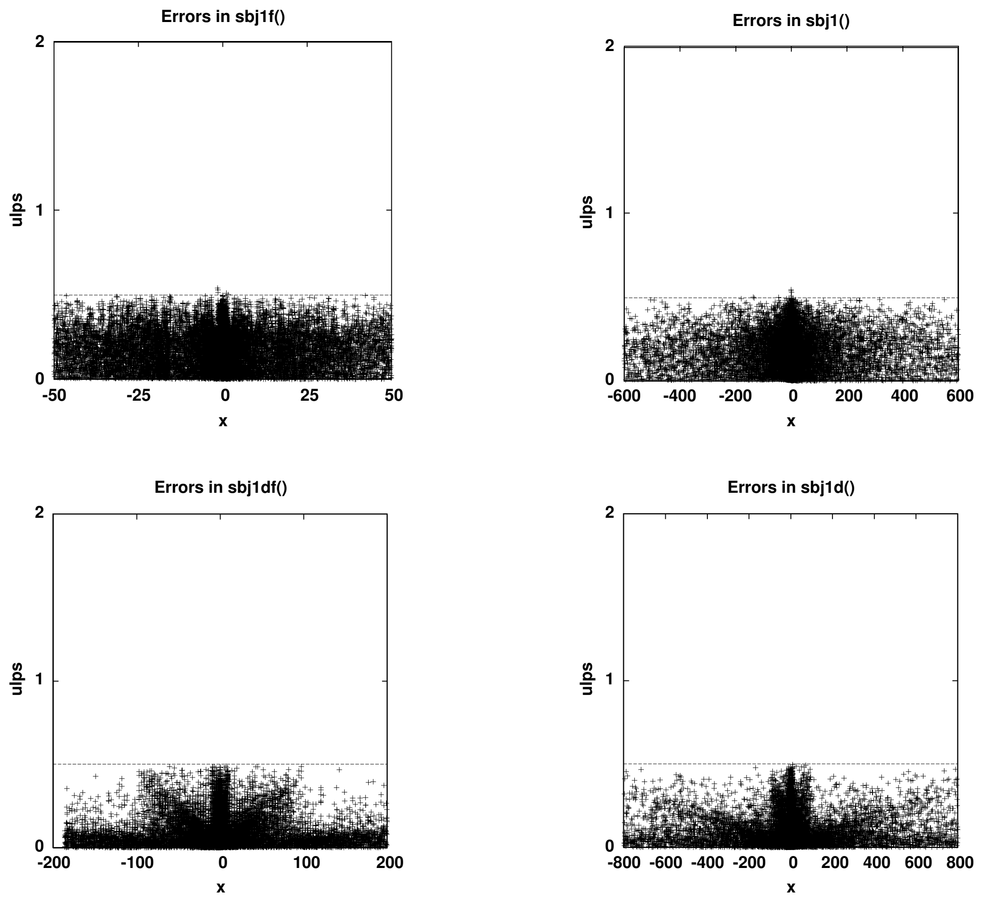


Figure 21.25: Errors in the binary (top) and decimal (bottom) `sbj1(x)` family.

The error plots in Figure 21.25 for `sbj1(x)`, and in Figure 21.28 on page 745 for `sby1(x)`, exhibit low relative error because of our use of higher intermediate precision, so we do not show plots from the improvements of this section. Instead, we do a short numerical experiment at the adjacent zeros bracketing the table end, using high-precision values from a symbolic-algebra system for comparison:

```
% hocd128 -lmcw
hocd128> x318 = 1_000.596_260_764_587_333_582_227_925_178_11

hocd128> x319 = 1_003.737_856_546_205_566_637_222_858_984_913

hocd128> sbj1(x318);  0.670873169872646828693521368291249352922e-34
         6.708_731_698_726_468_286_935_213_682_912_493e-35
         6.708_731_698_726_468_286_935_213_682_912_494e-35

hocd128> sbj1(x319); -0.371136263838085458455232524434396532682e-33
         -3.711_362_459_535_284_444_620_381_903_540_337e-34
         -3.711_362_638_380_854_584_552_325_244_343_965e-34

hocd128> x319a = 1_003.737_856_546_206

hocd128> sbj1(x319a); -0.431748747189598548200146130945033677271e-15
         -4.317_487_471_895_985_482_001_461_299_690_095e-16
```

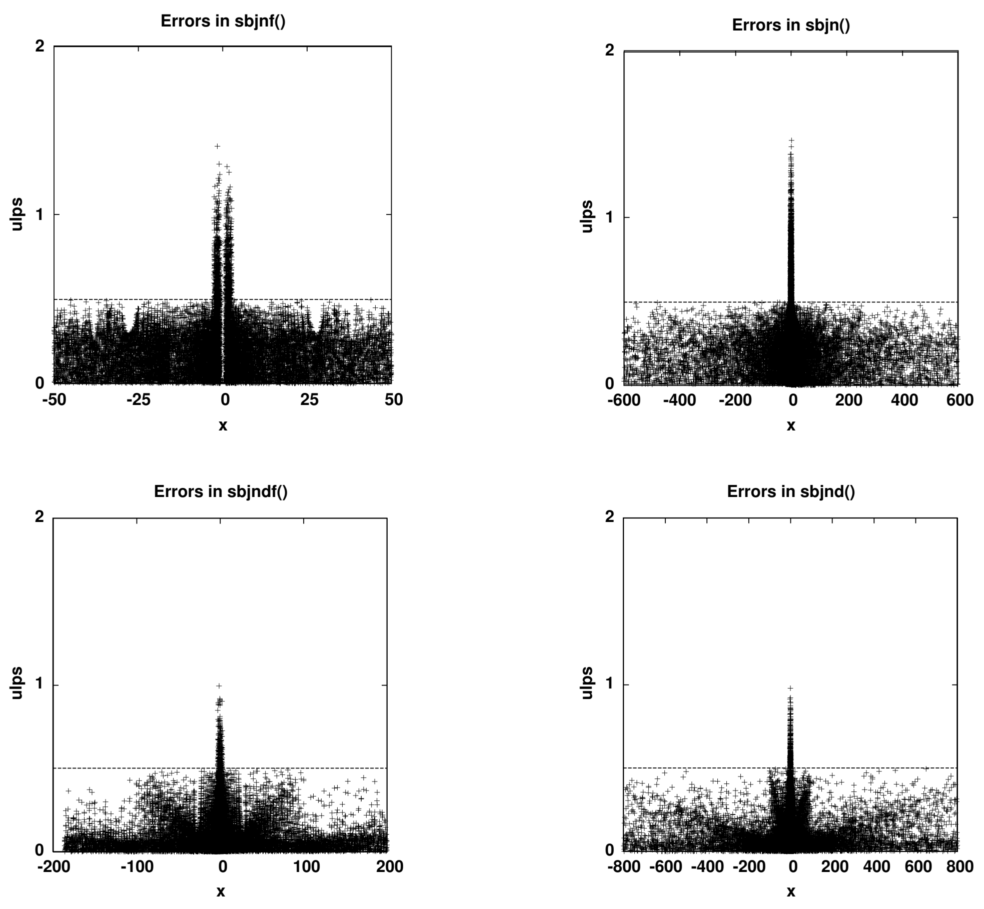


Figure 21.26: Errors in the binary (top) and decimal (bottom) `sbjn(n,x)` family for $n = 25$.

```
        -4.317_487_471_895_985_482_001_461_309_450_337e-16
```

In the region covered by the tabulated zeros, we find relative errors of about one ulp or less, but closest to the first zero outside that region, all but seven of the computed digits are wrong. The last experiment shows the improvement when we have a 16-digit approximation to the zero.

21.17 Modified spherical Bessel functions

The unscaled and scaled modified spherical Bessel functions of the first and second kinds are provided by functions with these prototypes:

```
double sbi0  (double);          double sbk0  (double);
double sbi1  (double);          double sbk1  (double);
double sbin  (int, double);     double sbkn  (int, double);
double sbis0 (double);          double sbks0 (double);
double sbis1 (double);          double sbks1 (double);
double sbisn (int, double);     double sbksn (int, double);
```

They have companions with the usual type suffixes for other precisions and bases.

As the function names suggest, the cases $n = 0$ and $n = 1$ receive special treatment. Code for arbitrary n can then use those functions internally.

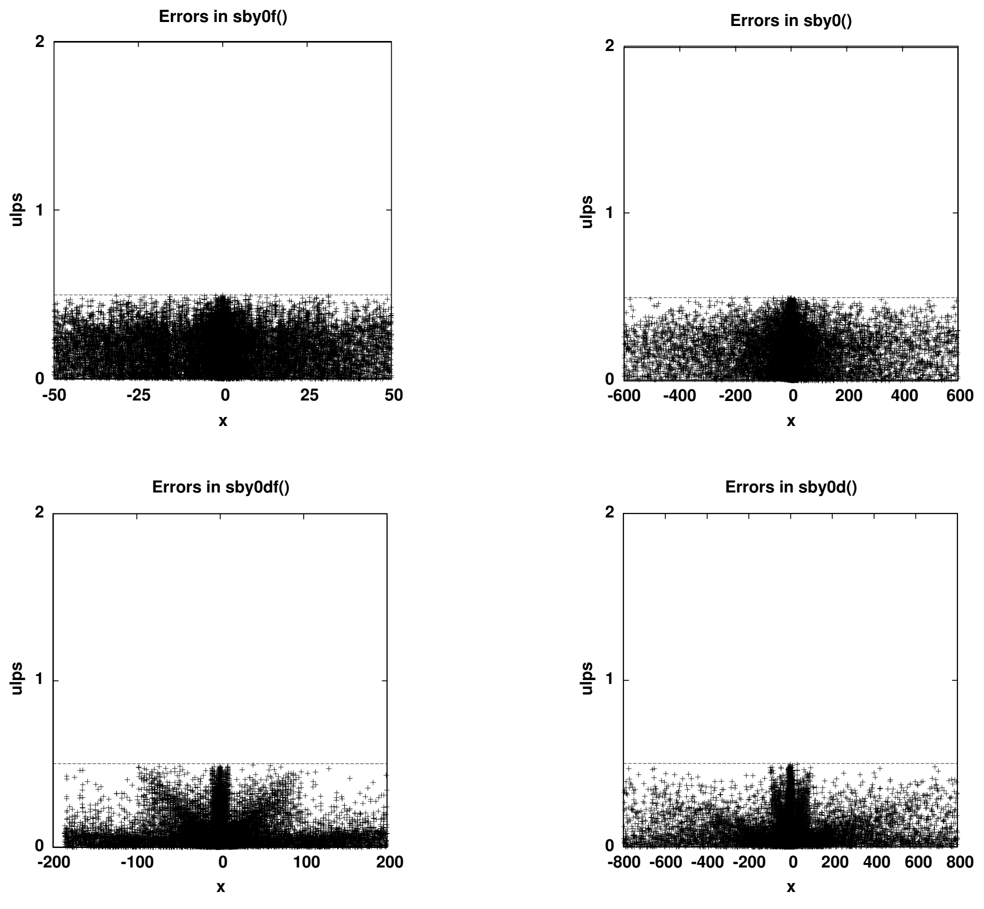


Figure 21.27: Errors in the binary (top) and decimal (bottom) `sby0(x)` family.

21.17.1 Computing $i_0(x)$

To compute the modified function of the first kind of order zero, `sbi0(x)`, a one-time initialization block first determines two cutoffs for Taylor series expansions, and the arguments above which $\sinh(x)$ and $i_0(x)$ overflow. The symmetry relation $i_0(-x) = i_0(x)$ allows us to work only with nonnegative x.

For small arguments, the Taylor-series expansion and its term recurrence looks like this:

$$\begin{aligned} i_0(x) &= 1 + (1/6)x^2 + (1/120)x^4 + (1/5040)x^6 + (1/362\,880)x^8 + \\ &\qquad (1/39\,916\,800)x^{10} + \cdots, \\ &= t_0 + t_1 + t_2 + \cdots, \\ t_0 &= 1, \qquad t_k = \left(\frac{1}{2k(2k+1)}\right) x^2 t_{k-1}, \qquad \text{for } k = 1, 2, 3, \ldots. \end{aligned}$$

That series enjoys rapid convergence, and all terms are positive, so there is never subtraction loss. The series is usable for x values larger than one. If $x = 4$, then 24 terms recover 34 decimal digits, and 39 terms produce 70 decimal digits.

The Taylor-series cutoffs are straightforward expressions that determine the value of x for which the k-th series term is smaller than half the rounding error: $c_k x_{\text{TS}}^k < \frac{1}{2}\epsilon/\beta$. Thus, we have $x_{\text{TS}} < \sqrt[k]{\frac{1}{2}\epsilon/(\beta c_k)}$, and we can choose k so that the k-th root needs only square roots or cube roots.

Unlike the implementations of functions described in most other chapters of this book, for the spherical Bessel functions, we use Taylor series over a wider range. For $i_0(x)$, in the four extended IEEE 754 decimal formats, the

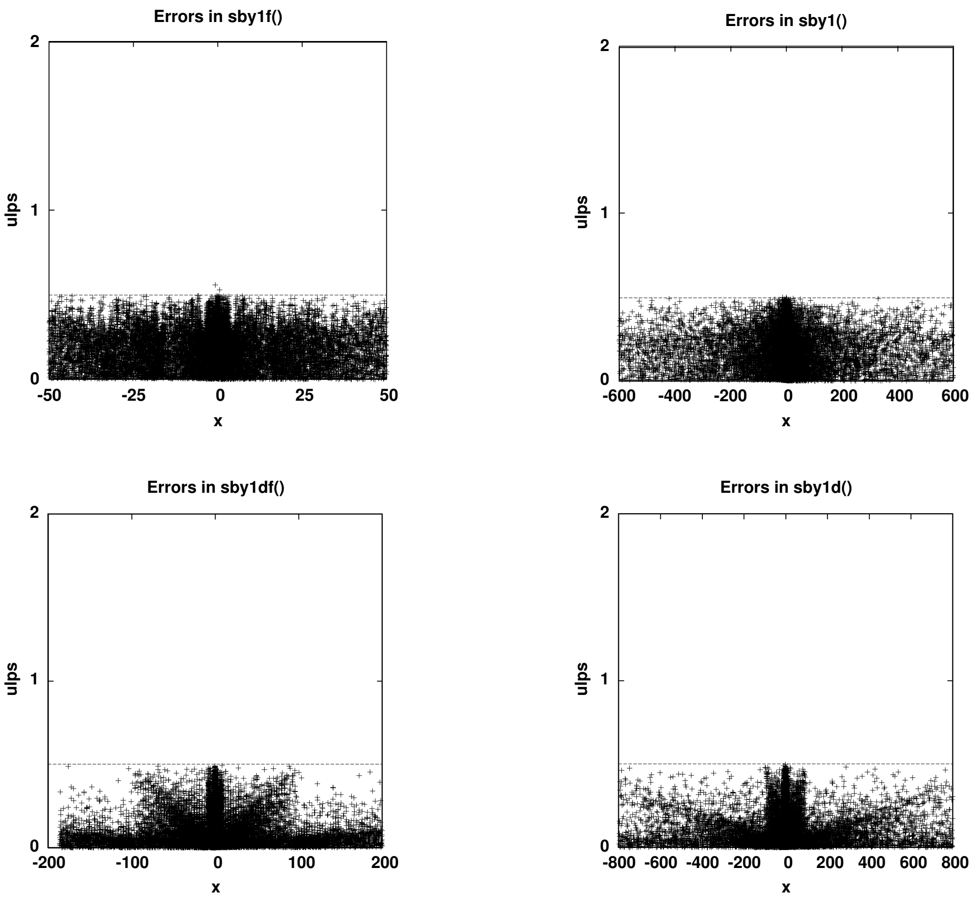


Figure 21.28: Errors in the binary (top) and decimal (bottom) `sby1(x)` family.

cutoffs for a nine-term series are about 2.84, 0.777, 0.0582, and 0.000 328. For $i_1(x)$, the cutoffs are larger, and importantly, for the lowest precision, cover the range where subtraction loss in the closed form of the function is a problem. Having both a short and a long series allows tiny arguments to be handled at lower cost.

For large x, $\sinh(x) \approx \exp(x)/2$, so overflow happens for $x > \log(2) + \log(\texttt{FP\_T\_MAX})$, where the argument of the last logarithm is the largest representable value. However, experiments on various systems show that implementations of the hyperbolic sine in some vendor libraries suffer from premature overflow, so we reduce the cutoff by $1/16$.

For large x, $i_0(x) = \sinh(x)/x \approx \exp(x)/(2x)$, but that does not give a simple way to determine x when the left-hand side is set to `FP_T_MAX`. We can find x by using Newton–Raphson iteration to solve for a root of $f(x) = \log(i_0(x)) - \log(\texttt{FP\_T\_MAX}) \approx x - \log(2x) - \log(\texttt{FP\_T\_MAX})$. Starting with x set to $\log(\texttt{FP\_T\_MAX})$, convergence is rapid, and five iterations produce a solution correct to more than 80 digits.

With the initialization complete, we first handle the special cases of NaN, zero, and $|x|$ above the second overflow cutoff. For small x, we sum three-term or nine-term Taylor series expansions in nested Horner form in order of increasing term magnitudes. For x below the first overflow cutoff, we use the hyperbolic sine formula. For x between the two overflow cutoffs, we can replace $\sinh(x)$ by $\exp(x)/2$, but we need to proceed carefully to avoid premature overflow from the exponential. Preprocessor conditionals select base-specific code for $\beta = 2, 8, 10$, and 16. For example, for decimal arithmetic, we compute $\exp(x)/(2x) = \exp(x/10)^{10}/(2x)$ like this:

```
volatile fp_t t;
fp_t u, u2, u4;
```

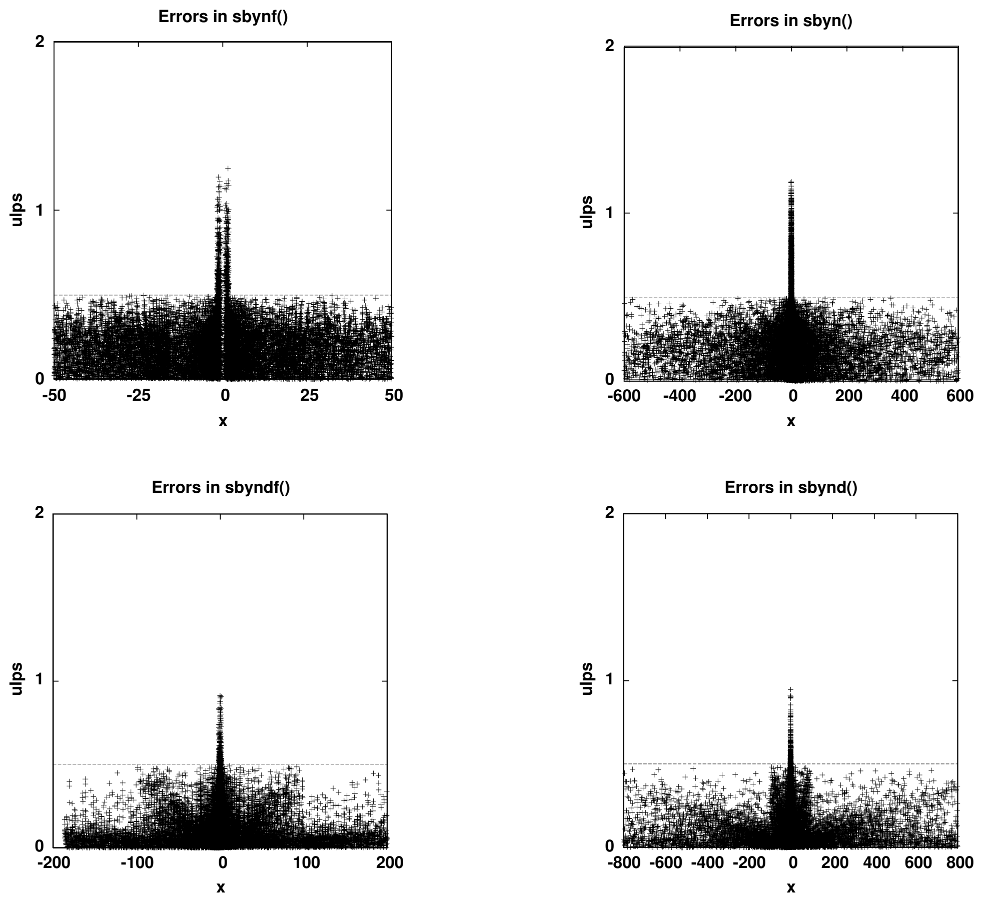


Figure 21.29: Errors in the binary (top) and decimal (bottom) `sbyn(n,x)` family for $n = 25$.

```
    u = EXP(x * FP(0.1));   /* exact argument scaling */
    u2 = u * u;
    u4 = u2 * u2;
    t = HALF * u2 / x;
    STORE(&t);
    result = t * u4 * u4;
```

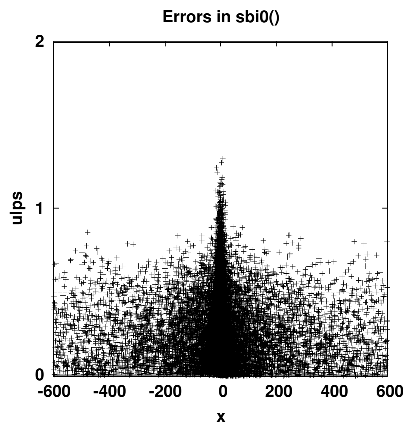
Because of the error magnification of the exponential, it is imperative to scale its argument exactly. The `volatile` keyword and the `STORE()` macro force intermediate expression evaluation to prevent an optimizing compiler from delaying the division by x until the products have been computed, and overflowed. Avoidance of premature overflow therefore costs at least *seven* rounding errors in a decimal base (or *four* when $\beta = 2$), but that happens only near the overflow limit.

Figure 21.30 on the facing page shows the measured accuracy in two of the functions for computing $i_0(x)$. Plots for their companions are similar, and thus, not shown.

21.17.2 Computing $\mathrm{is}_0(x)$

The scaled modified spherical Bessel function of order zero is defined by

$$\begin{aligned}\mathrm{is}_0(x) &= \exp(-|x|)i_0(x)\\ &= \exp(-|x|)\sinh(x)/x\end{aligned}$$



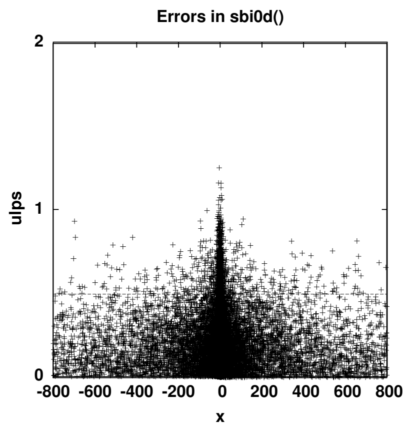


Figure 21.30: Errors in the binary (left) and decimal (right) `sbi0(x)` family.

$$= (1 - \exp(-2|x|))/(2|x|).$$

For $|x|$ in $[0, -\frac{1}{2}\log(\frac{1}{2})] \approx [0, 0.347]$, there is subtraction loss in the numerator. We could handle that by using a polynomial approximation in that region, but we have already done so elsewhere in the library, because we can rewrite the function as $\mathrm{is}_0(|x|) = -\operatorname{expm1}(-2|x|)/(2|x|)$.

For small arguments, the Taylor series is useful:

$$\mathrm{is}_0(x) = 1 - x + (2/3)x^2 - (1/3)x^3 + (2/15)x^4 - (2/45)x^5 + (4/315)x^6 - \\ (1/315)x^7 + (2/2835)x^8 - (2/14\,175)x^9 + \cdots .$$

There is no loss of leading bits if we use it only for $|x| < 1/2$.

The series terms can be computed with this recurrence:

$$t_0 = 1, \qquad t_k = -\left(\frac{2}{k+1}\right) x t_{k-1}, \qquad \text{for } k = 1, 2, 3, \ldots,$$
$$\mathrm{is}_0(x) = t_0 + t_1 + t_2 + \cdots .$$

For sufficiently large $|x|$, the exponential function is negligible, and the function reduces to $1/(2|x|)$. That happens for $|x|$ in $[-\frac{1}{2}\log(\frac{1}{2}\epsilon/\beta), \infty)$. In IEEE 754 arithmetic, the lower limit is about 8.664 in the 32-bit format, and 39.510 in the 128-bit format. Thus, over most of the floating-point range, we do not even need to call an exponential function.

The code in `sbis0x.h` has a one-time initialization block that computes two Taylor-series cutoffs, the subtraction-loss cutoff, and the upper cutoff where the exponential function can be omitted. It then handles the case of NaN and zero arguments. Otherwise, it records whether x is negative, and forces it positive, and then splits the computation into five regions: a four-term Taylor series, a nine-term Taylor series, the subtraction loss region, the nonneglible exponential region, and the final region for large x where the result is just $1/(2x)$. The final result is then negated if x was negative on entry to the function.

There are subtle dependencies on the base in the handling of the denominator $2x$. In the loss region, for $\beta \neq 2$, compute the result as $-\frac{1}{2}\operatorname{expm1}(-(x+x))/x$ to avoid introducing an unnecessary rounding error in the denominator. In the exponential region, the result is $\frac{1}{2}((1 - \exp(-(x+x)))/x)$ when $\beta \neq 2$. In the overflow region, the result is $\frac{1}{2}x$.

Figure 21.31 on the next page shows the measured accuracy in two of the functions for computing $\mathrm{is}_0(x)$.

21.17.3 Computing $i_1(x)$

The code for the spherical Bessel function $i_1(x) = (\cosh(x) - \sinh(x)/x)/x$ begins with a one-time initialization block that computes two Taylor series cutoffs, and two overflow cutoffs where the hyperbolic functions overflow,

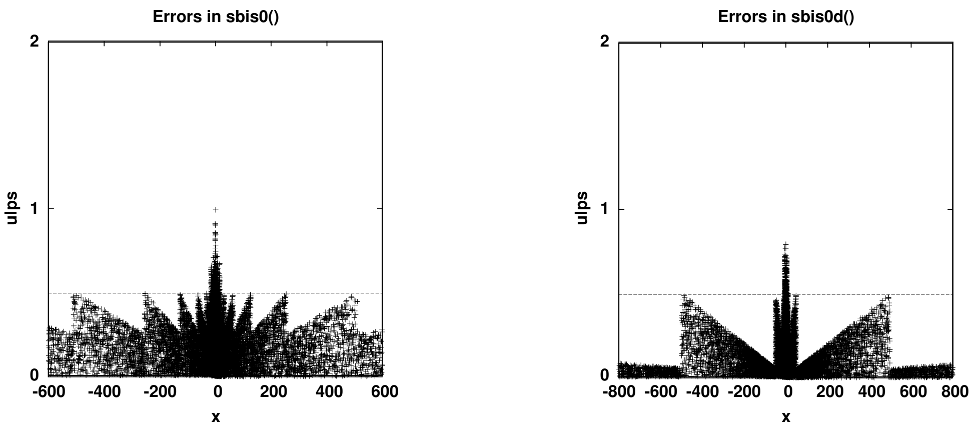


Figure 21.31: Errors in the binary (left) and decimal (right) `sbis0(x)` family.

and where $i_1(x)$ itself overflows. As in the code for $i_0(x)$, Newton–Raphson iteration is needed to find the second cutoff.

The code is easy for arguments that are NaN, zero, or have magnitudes beyond the second overflow cutoff. Otherwise, if x is negative, the symmetry relation $i_1(-x) = -i_1(x)$ allows x to be forced positive, as long as we remember to negate the final result. For small x, we sum three-term or nine-term Taylor series in Horner form.

For x between the second Taylor series cutoff and the first overflow limit, we could evaluate the hyperbolic functions that define $i_1(x)$. However, there is subtraction loss in the numerator for x in $[0, 1.915]$, and all digits can be lost when x is small. There does not appear to be a simple alternative expression for $i_1(x)$ that avoids the loss, and requires only functions that we already have. We therefore have two choices: sum the general Taylor series until it converges, or use a polynomial approximation.

The Taylor-series expansion and its term recurrence looks like this:

$$\begin{aligned}
i_1(x) &= (1/3)x + (1/30)x^3 + (1/840)x^5 + (1/45\,360)x^7 + \cdots \\
&= \sum_{k=1}^{\infty} \frac{2k}{(2k+1)!} x^{2k-1}, \\
&= (x/3)(t_1 + t_2 + t_3 + + \cdots) \\
t_1 = 1, \qquad & t_{k+1} = \left(\frac{1}{2k(2k+3)}\right) x^2 t_k, \qquad k = 1, 2, 3, \ldots.
\end{aligned}$$

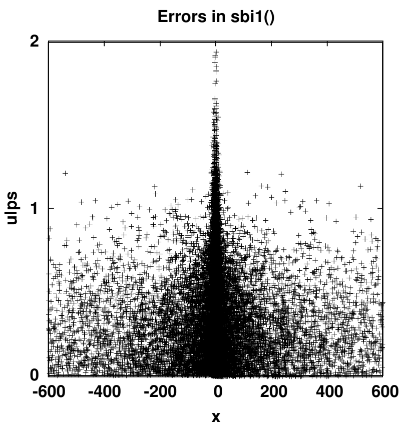
All terms have the same sign, so no subtraction loss is possible. It is best to start with $k = 2$ and sum the series until it has converged, and then add the first term. Convergence is slowest for the largest x, and numerical experiments show that we need at most 7, 12, 13, 19, and 32 terms for the five binary extended IEEE 754 formats. Accumulation of each term costs two adds, two multiplies, and one divide. However, if we store a precomputed table of values of $1/(2k(2k+3))$, the sum costs only one add and two multiplies per term.

In most cases, only the last two rounding errors affect the computed function value. In binary arithmetic, one of those errors can be removed by rewriting $x/3$ as $x/4 + x/12$, and then moving the term $x/12$ into the sum of the remaining terms. That sum is then added to the *exact* value $x/4$.

For the polynomial-fit alternative, we can compute the Bessel function as

$$\begin{aligned}
i_1(x) &\approx x/3 + x^3\mathcal{R}(x^2) \\
&\approx x/4 + (x/12 + x^3\mathcal{R}(x^2)), \\
\mathcal{R}(x) &= (i_1(\sqrt{x}) - \sqrt{x}/3)/\sqrt{x^3}, \qquad \textit{fit to rational polynomial.}
\end{aligned}$$

For x on $[0, 1]$, the term ratio $x^3\mathcal{R}(x^2)/(x/3)$ lies on $[0, 0.103]$, so the polynomial adds at least one decimal digit of precision. For x on $[1, 1.915]$, that ratio reaches 0.418.



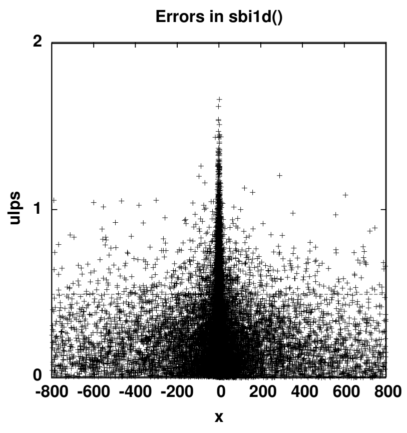


Figure 21.32: Errors in the binary (left) and decimal (right) `sbi1(x)` family.

Numerical experiments in Maple show that rational polynomial fits of orders ⟨2/1⟩, ⟨4/3⟩, ⟨4/4⟩, ⟨7/6⟩, and ⟨12/12⟩ are needed for the five binary extended IEEE 754 formats. Those fits require less than a quarter of the operation counts of Taylor-series sums for the same accuracy. Consequently, rational polynomials are the default evaluation method in `sbi1(x)`, although the others can be selected at compile time by defining preprocessor symbols.

Above the interval $[0, 1.915]$, we can safely use the hyperbolic functions, until we reach the region between the two overflow cutoffs, where we have $i_1(x) \approx \exp(x)(1 - 1/x)/(2x) = (\exp(x)/(2x^2))(x - 1)$. Having a single function that computes both `sinh(x)` and `cosh(x)` simultaneously is clearly useful here, and our library supplies `sinhcosh(x,psh,pch)` for that purpose. As we did for $i_0(x)$, to avoid premature overflow in the exponential, expand it as $\exp(x/\beta)^\beta$, and include the factor $1/(2x)$ early in the product of the factors. For example, for a hexadecimal base, the code looks like this:

```
volatile fp_t t;
fp_t u, u2, u4, v;

u = EXP(x * FP(0.0625));    /* exact argument scaling */
u2 = u * u;
u4 = u2 * u2;
t = HALF * u4 / (x * x);
STORE(&t);
v = t * u4 * u4 * u4;
result = v * (x - ONE);
```

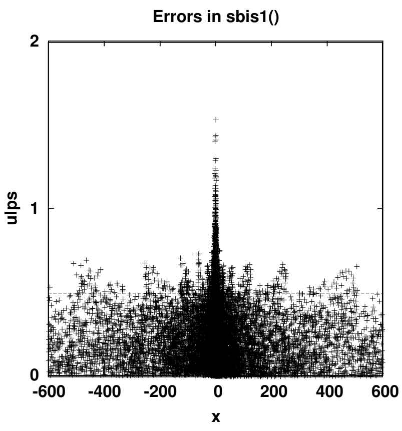
Avoidance of premature overflow costs *ten* rounding errors (or *six* when $\beta = 2$), but only near the overflow limit.

Figure 21.32 shows the measured accuracy in two of the functions for computing $i_1(x)$.

21.17.4 Computing $\mathrm{is}_1(x)$

The scaled modified spherical Bessel function of order one has the symmetry relation $\mathrm{is}_1(-x) = -\mathrm{is}_1(x)$, so we henceforth assume that x is nonnegative, and we set a flag to negate the final result if the argument is negative. The function is then defined by

$$\begin{aligned}
\mathrm{is}_1(x) &= \exp(-x)i_1(x), \qquad \text{for } x \geq 0, \\
&= \exp(-x)(\cosh(x) - \sinh(x)/x)/x \\
&= \exp(-x)((\exp(x) + \exp(-x)) - (\exp(x) - \exp(-x))/x)/(2x) \\
&= (1 + \exp(-2x) - (1 - \exp(-2x))/x)/(2x)
\end{aligned}$$



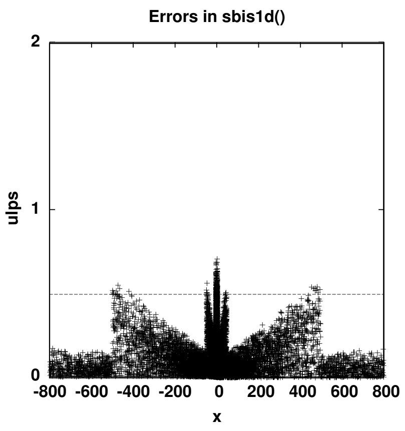


Figure 21.33: Errors in the binary (left) and decimal (right) `sbis1(x)` family.

$$= ((1 - 1/x) + (1 + 1/x)\exp(-2x))/(2x).$$

As we observed in Section 21.17.3 on page 747, there is subtraction loss in the hyperbolic form of the numerator for x in $[0, 1.915]$, and that region is best handled by summing a Taylor series, or with a polynomial approximation.

The Taylor series for $\mathrm{is}_1(x)$, and its term recurrence relation, look like this:

$$\begin{aligned}\mathrm{is}_1(x) &= (x/3)(1 - x + (3/5)x^2 - (4/15)x^3 + (2/21)x^4 - (1/35)x^5 + \\ &\quad (1/135)x^6 - (8/4725)x^7 + (2/5775)x^8 - (2/31\,185)x^9 + \cdots), \\ &= (x/3)(t_0 + t_1 + t_2 + \cdots), \\ &= (x/4)(t_0 + t_1 + t_2 + \cdots) + (x/12)(t_0 + t_1 + t_2 + \cdots), \qquad \text{when } \beta = 2, \\ t_0 = 1, \qquad t_k &= -\left(\frac{2(k+1)}{k(k+3)}\right) x t_{k-1}, \qquad k = 1, 2, 3, \ldots.\end{aligned}$$

Unfortunately, that series converges slowly, so its use is limited to $|x| \ll 1$. In the 32-bit IEEE 754 formats, the nine-term series can be used only for $|x| < 0.31$. Higher precisions reduce that cutoff. Just as we did for $i_1(x)$ in a binary base, replacing the inexact $x/3$ factor by $x/4 + x/12$ reduces the outer rounding error.

In the exponential form, the second term is almost negligible for x above the value for which $\exp(-2x)$ is smaller than the rounding error $\frac{1}{2}\epsilon/\beta$. That cutoff is then $-\frac{1}{2}\log(\frac{1}{2}\epsilon/\beta)$. However, that is a slight underestimate, and a better choice that works in all IEEE 754 formats is larger by $1/8$. We could, of course, use Newton–Raphson iteration to find the precise cutoff value, but a simple increment by $1/8$ is easier, and of little significance for later computation.

The code in `sbis1x.h` first handles NaN and zero arguments. Otherwise, it enforces the symmetry relation by forcing x to be positive with a negation flag set for later use, and then splits the computation into five regions: three-term or nine-term Taylor series for small x, a polynomial approximation in the loss region, the exponential form below the upper cutoff, and above that cutoff, simply $(\frac{1}{2} - (\frac{1}{2})/x)/x$. In the last region, to reduce rounding error, multiplication by the reciprocal of x should be avoided.

Figure 21.33 shows the measured accuracy in two of the functions for computing $\mathrm{is}_1(x)$.

21.17.5 Computing $i_n(x)$

To find the unscaled modified spherical Bessel functions of arbitrary order, `sbin(n,x)`, we often need to use the recurrence relation when $n > 1$. As we observed in Section 21.13 on page 728, stable computation of the $i_n(x)$ Bessel functions requires using the continued-fraction form to find the ratio $i_n(x)/i_{n-1}(x)$, and then downward recurrence to find $i_n(x)/i_0(x)$, from which the function value can be found by multiplying the ratio by the result returned by `sbi0(x)`.

For small arguments, it is desirable for speed and accuracy to sum a series, but Maple is unable to find one for arbitrary n. However, Mathematica is successful:

```
% math
In[1]:= sbin = Function[{n, x}, BesselI[n + 1/2,x] * Sqrt[Pi/(2*x)]];

In[2]:= Simplify[Series[sbin[n,x], {x, 0, 6}]]

              -1 - n              -2 - n           2
         n   2       Sqrt[Pi]    2       Sqrt[Pi] x
Out[2]= x  (---------------- + --------------------- +
                    3                          3
              Gamma[- + n]     (3 + 2 n) Gamma[- + n]
                    2                          2

               -4 - n          4
              2       Sqrt[Pi] x
>       ------------------------------ +
                         2        3
        (15 + 16 n + 4 n ) Gamma[- + n]
                                  2

                     -5 - n           6
                    2       Sqrt[Pi] x                    7
>       ------------------------------------------- + O[x] )
                              2      3        3
        3 (105 + 142 n + 60 n  + 8 n ) Gamma[- + n]
                                              2
```

That expansion looks horrid, but we recognize some common factors on the right-hand side, and try again with those factors moved to the left-hand side:

```
In[3]:= Simplify[Series[sbin[n,x] * 2^(n+1) * Gamma[n + 3/2] /
                        Sqrt[Pi], {x, 0, 6}]]

                    2               4
         n         x               x
Out[3]= x  (1 + ------- + ------------------- +
                6 + 4 n                     2
                           120 + 128 n + 32 n

                      6
                     x                      7
>       ------------------------------- + O[x] )
                               2      3
        48 (105 + 142 n + 60 n  + 8 n )
```

The series coefficients are reciprocals of polynomials in n with integer coefficients. For $n > x^2$, the leading term is the largest, and the left-hand side grows like x^n.

The factor in the left-hand side looks ominous, until we remember that half-integral values of the gamma function have simple values that are integer multiples of $\sqrt{\pi}$. We recall some results from **Section 18.1** on page 522 to find the form of the factor:

$$\begin{aligned}
\Gamma(\tfrac{1}{2}) &= \sqrt{\pi}, \\
\Gamma(n+\tfrac{1}{2}) &= (2n-1)!!\,\Gamma(\tfrac{1}{2})/2^n, && \textit{if } n \geq 0, \\
&= (2n-1)!!\sqrt{\pi}/2^n, \\
\Gamma(n+\tfrac{3}{2}) &= (2(n+1)-1)!!\sqrt{\pi}/2^{n+1} \\
&= (2n+1)!!\sqrt{\pi}/2^{n+1}, \\
\mathbf{(2n+1)!!} &= \mathbf{2^{n+1}\Gamma(n+\tfrac{3}{2})/\sqrt{\pi}}, && \textit{left-hand side factor.}
\end{aligned}$$

With a larger limit in the series expansion in Mathematica, and application of its polynomial-factorization function, `Factor[]`, we therefore find the first few terms of the Taylor series as

$$\begin{aligned} i_n(x) = \frac{x^n}{(2n+1)!!}(1 &+ \frac{1}{2(3+2n)}x^2 + \frac{1}{8(3+2n)(5+2n)}x^4 + \\ &\frac{1}{48(3+2n)(5+2n)(7+2n)}x^6 + \\ &\frac{1}{384(3+2n)(5+2n)(7+2n)(9+2n)}x^8 + \\ &\frac{1}{3840(3+2n)(5+2n)(7+2n)(9+2n)(11+2n)}x^{10} + \\ &\frac{1}{46\,080(3+2n)(5+2n)(7+2n)(9+2n)(11+2n)(13+2n)}x^{12} + \\ &\cdots). \end{aligned}$$

Notice the additional structure that coefficient factorization exposes, and that all terms are positive. Taking ratios of adjacent coefficients shows that the terms of the Taylor series have a delightfully simple recurrence:

$$t_0 = 1, \qquad t_k = \left(\frac{1}{2k(2k+2n+1)}\right) x^2 t_{k-1}, \qquad \textit{for } k = 1, 2, 3, \ldots,$$
$$i_n(x) = \frac{x^n}{(2n+1)!!}(t_0 + t_1 + t_2 + t_3 + \cdots).$$

The general Taylor-series term and its coefficient can be written explicitly like this:

$$\begin{aligned} t_k &= c_k x^{2k}, && \textit{for } k = 0, 1, 2, \ldots, \\ c_0 &= 1, \\ c_k &= 1/\prod_{j=1}^{k}(2j(2n+1+2j)), && \textit{for } k = 1, 2, 3, \ldots, \\ &= 1/\big(2^k\, k!\prod_{j=1}^{k}(2n+1+2j)\big) \\ &= 1/\big(2^{2k}\, k!\prod_{j=1}^{k}(n+\tfrac{1}{2}+j)\big) \\ &= 1/\big(2^{2k}\, k!\,(n+\tfrac{3}{2})_k\big), && \textit{see text for this notation.} \end{aligned}$$

In the last equation, the notation $(a)_k$ stands for the product $a \times (a+1) \times (a+2) \times \cdots \times (a+k-1) = \Gamma(a+k)/\Gamma(a)$. It is called the *rising factorial function*, or sometimes, the *Pochhammer symbol*. Maple provides it as the function `pochhammer(a,k)`, and Mathematica as the function `Pochhammer[a,k]`.

Normally, we apply the Taylor series only for small x, so that only a few terms are needed to reach a given accuracy. However, here we can see that the series also needs only a few terms if $n > x^2$, because the denominator of the k-th term is larger than $2^{2k}k!$, effectively providing more than $2k$ additional bits to the sum. That is an important observation, because our other computational route to $i_n(x)$ involves a recurrence whose execution time is proportional to n, plus the separate computation of $i_0(x)$. To illustrate how well the series converges, **Table 21.8** on the next page shows the accuracy obtainable with modest numbers of terms for various choices of n and x. The limited exponent range of most floating-point systems means that we usually cannot compute $i_n(x)$ for n values as large as those shown in that table before the function overflows.

With care, we can use the Taylor series for larger values of x than permitted by the condition $n > x^2$. If we are prepared to sum up to k terms, and if $x^2 > n$, then the first few terms grow, but eventually they get smaller because of the rapid growth of the denominator. If term k is smaller than the rounding error compared to the first term, then the sum has surely converged to machine precision, so we have the requirements that

$$t_k = c_k x^{2k} < \epsilon/(2\beta), \qquad x^2 < \sqrt[k]{\epsilon/(2\beta c_k)}, \qquad x^2/n < \sqrt[k]{2^{2k}k!\epsilon/(2\beta)}.$$

Table 21.8: Convergence of the Taylor series of $i_n(x)$, showing the relative error in a sum of N terms.

n	relerr	n	relerr	n	relerr
		$x = 1$			
$N = 10$		$N = 20$		$N = 30$	
10	`2.82e-25`	10	`3.02e-57`	10	`7.47e-93`
100	`1.47e-33`	100	`4.76e-72`	100	`4.14e-113`
1000	`2.48e-43`	1000	`3.00e-91`	1000	`2.03e-141`
10 000	`2.61e-53`	10 000	`3.66e-111`	10 000	`3.12e-171`
		$x = 10$			
$N = 10$		$N = 20$		$N = 30$	
10	`2.82e-05`	10	`3.02e-17`	10	`7.47e-33`
100	`1.47e-13`	100	`4.76e-32`	100	`4.14e-53`
1000	`2.48e-23`	1000	`3.00e-51`	1000	`2.03e-81`
10 000	`2.61e-33`	10 000	`3.66e-71`	10 000	`3.12e-111`
		$x = 100$			
$N = 10$		$N = 20$		$N = 30$	
1000	`0.00248`	1000	`3.00e-11`	1000	`2.03e-21`
10 000	`2.61e-13`	10 000	`3.66e-31`	10 000	`3.12e-51`
100 000	`2.63e-23`	100 000	`3.73e-51`	100 000	`3.25e-81`
1 000 000	`2.63e-33`	1 000 000	`3.74e-71`	1 000 000	`3.27e-111`

The last inequality follows from the replacement in c_k of $(n + \frac{3}{2})_k$ by the *smaller* value n^k. For a large fixed k, computation of the right-hand requires rewriting it in terms of logarithms and an exponential to avoid premature overflow. However, it gives us a scale factor, s, that needs to be computed only once for the chosen limit on k, and that can then be used to determine how large n can be compared to x^2 to use the series.

When $x^2 > n$, some term after the first is the largest, and because each term suffers four rounding errors, those errors can have a large affect on the computed sum. One solution would be to accumulate the sum in higher precision. Alternatively, we can just reduce the scale factor to force a switch to an alternative algorithm for smaller values of x, and that is what we do in the code in `sbinx.h`.

After a one-time initialization block to compute a Taylor-series cutoff and the limit on x^2/n, the code in `sbinx.h` for computing $i_n(x)$ has several blocks. First, there are checks for x is a NaN, Infinity, or zero, then checks for $n = 0$ or $n = 1$. Otherwise, we record a negation flag that tells us whether x is negative and n is even, and then we force x to be positive. The computation is then split into four regions, the first where $n < 0$ and the downward recurrence is stable, the second where the four-term Taylor series can be summed, the third where the general Taylor series is effective because $x^2/n < s$, and the last where the Lentz algorithm evaluates the continued-fraction ratio $i_n(x)/i_{n-1}(x)$, then downward recurrence is used to find $i_n(x)/i_0(x)$, and the magnitude of the final result is obtained by multiplying that ratio by `sbi0(x)`. If the negation flag is set, the last result must be negated.

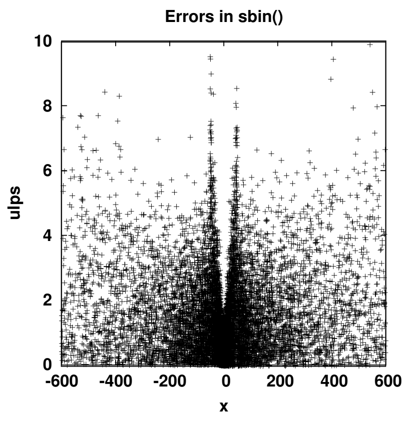
Figure 21.34 on the following page shows the measured accuracy in two of the functions for computing $i_{25}(x)$. The extended vertical range is needed to show how numerical errors increase in the recurrence.

21.17.6 Computing $\mathrm{is}_n(x)$

The scaled modified spherical Bessel functions of arbitrary order, implemented in the function `sbisn(n,x)`, are defined as $\mathrm{is}_n(x) = \exp(-|x|)i_n(x)$, and satisfy the argument symmetry relation $\mathrm{is}_n(-x) = (-1)^n \mathrm{is}_n(x)$. The exponential scaling damps the growth of $i_n(x)$, making $\mathrm{is}_n(x)$ representable over more of the floating-point range. For example, $i_{10}(1000) \approx 10^{431}$, but $\mathrm{is}_{10}(1000) \approx 10^{-4}$.

As happens with the other spherical Bessel functions of order n, Maple is unable to find a Taylor series expansion of the scaled functions for arbitrary n, but Mathematica can do so, and we can display its results like this:

$$\mathrm{is}_n(x) = \frac{x^n}{(2n+1)!!}(1 - x +$$
$$\frac{2+n}{3+2n}x^2 - \frac{1}{3}\frac{3+n}{3+2n}x^3 +$$



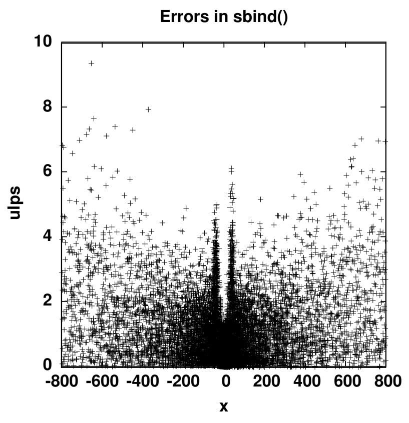


Figure 21.34: Errors in the binary (left) and decimal (right) `sbin(n,x)` family for $n = 25$.

$$\begin{aligned}
&\frac{1}{6}\frac{(3+n)(4+n)}{(3+2n)(5+2n)}x^4 - \frac{1}{30}\frac{(4+n)(5+n)}{(3+2n)(5+2n)}x^5 + \\
&\frac{1}{90}\frac{(4+n)(5+n)(6+n)}{(3+2n)(5+2n)(7+2n)}x^6 - \\
&\frac{1}{630}\frac{(5+n)(6+n)(7+n)}{(3+2n)(5+2n)(7+2n)}x^7 + \\
&\cdots) \\
= {}&\frac{x^n}{(2n+1)!!}(t_0 + t_1 + t_2 + \cdots).
\end{aligned}$$

Successive terms can be produced with this recurrence:

$$t_0 = 1, \qquad t_k = -\frac{2}{k}\left(\frac{k+n}{k+1+2n}\right)xt_{k-1}, \qquad k = 1, 2, 3, \ldots.$$

Convergence is slower than that of the series for $i_n(x)$, and because the signs alternate, subtraction loss is a problem unless $|x| < \frac{1}{2}$. Nevertheless, for $x = \frac{1}{2}$ and $n = 1$, only 10, 17, 20, 31, and 53 terms are required for the five binary extended IEEE 754 formats, and convergence is faster for smaller $|x|$ or larger n.

The computational approach in `sbisnx.h` is similar to that in `sbinx.h`: a one-time initialization block to compute a Taylor series cutoff, special handling when x is NaN, zero, or Infinity, or $n = 0$ or $n = 1$, downward recurrence if $n < 0$, a four-term Taylor series for small x, a general Taylor series for $|x| < \frac{1}{2}$, and otherwise, the Lentz algorithm for the continued fraction, downward recurrence, and then normalization by `sbis0(x)`.

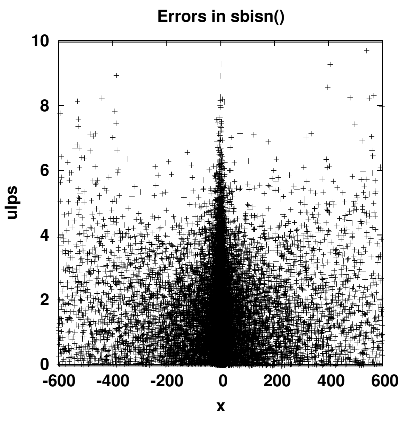
Figure 21.35 on the next page shows the measured accuracy in two of the functions for computing $\mathrm{is}_{25}(x)$.

21.17.7 Computing $k_n(x)$ and $\mathrm{ks}_n(x)$

Because the modified spherical Bessel function of the second kind for order n, $k_n(x)$, is the product of $(\pi/(2x^{n+1}))\exp(-x)$ and a polynomial of order n in x with positive integer coefficients, its upward recurrence relation is certainly stable for positive arguments. There *is* subtraction loss for negative arguments, but the error is always small compared to the dominant constant term in the polynomial factor, so in practice, the computation is also stable for negative arguments.

The two lowest-order unscaled functions have these Taylor-series expansions:

$$\begin{aligned}
k_0(x) = (\pi/(2x))(1 - x + (1/2)x^2 - (1/6)x^3 + (1/24)x^4 - \\
(1/120)x^5 + (1/720)x^6 - (1/5040)x^7 + \cdots),
\end{aligned}$$



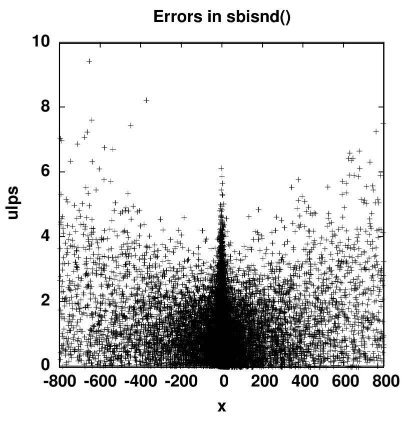


Figure 21.35: Errors in the binary (left) and decimal (right) `sbisn(n,x)` family for $n = 25$.

$$k_1(x) = (\pi/(2x^2))(1 - (1/2)x + (1/3)x^2 - (1/8)x^3 + (1/30)x^4 - (1/144)x^5 + (1/840)x^6 - (1/5760)x^7 + \cdots).$$

Leading bit loss in their sums is avoided if we choose the cutoffs $x_{\mathrm{TS}} = \frac{1}{2}$ for $k_0(x)$ and $x_{\mathrm{TS}} = \frac{3}{4}$ for $k_1(x)$.

The series for the scaled functions are just the polynomials given in **Table 21.6** on page 738.

For large x, the value of $k_n(x)$ approaches $(\pi/(2x))\exp(-x)$, so the only significant concern in its computation is the optimization of avoiding a call to the exponential when x is large enough that $k_n(x)$ underflows to zero. Because that value of x is independent of n, it is a cutoff that we can compute in a one-time initialization block. The scaled companion, $\mathrm{ks}_n(x) = \exp(x)k_n(x)$, needs no exponential, and thus, no cutoff test.

Code for the cases $n = 0$ and $n = 1$ is implemented in the files `sbk0x.h`, `sbks0x.h`, `sbk1x.h`, and `sbks1x.h`. The functions contain a one-time initialization block to compute overflow and underflow cutoffs. They then check for the special cases of x is negative, a NaN, or small enough to cause overflow. The unscaled functions also check whether x is above the underflow cutoff. Otherwise, the functions are simply computed from their definitions, but taking care to represent $\pi/2$ as $2(\pi/4)$ in any base with wobbling precision.

The files `sbknx.h` and `sbksnx.h` contain the code for arbitrary n, and differ only in their references to the unscaled or scaled functions of orders 0 and 1. Once n and x have been determined to be other than one of the special cases, upward recurrence starting from function values for $n = 0$ and $n = 1$ finds the result in at most n steps. The loop test at the start of iteration k includes the expected test $k < n$, and a second test that the function value is still nonzero, so that early loop exit is possible once the underflow region has been reached.

Figure 21.36 on the next page through **Figure 21.41** on page 757 show the measured accuracy in two of the functions for computing the modified spherical Bessel functions of the second kind.

21.18 Software for Bessel-function sequences

Some applications of Bessel functions require their values for a fixed argument x, and a consecutive sequence of integer orders. The existence of three-term recurrence relations suggests that, at least for some argument ranges, it should be possible to generate members of a Bessel function sequence for little more than the cost of computing two adjacent elements. Many of the software implementations published in the physics literature cited in the introduction to this chapter produce such sequences, but the POSIX specification of the Bessel functions provides only for single function values of $J_n(x)$ and $Y_n(x)$.

For the `mathcw` library, we implement several families of functions that return a vector of $n + 1$ Bessel function values for a fixed argument x, starting from order zero:

```
void vbi  (int n, double result[], double x); /* I(0..n,x) */
void vbis (int n, double result[], double x); /* I(0..n,x)*exp(-|x|) */
```

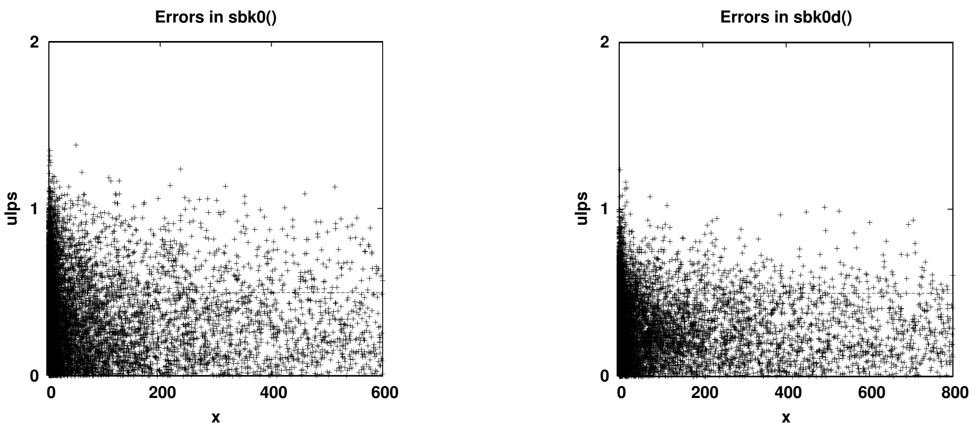


Figure 21.36: Errors in the binary (left) and decimal (right) `sbk0(x)` family.

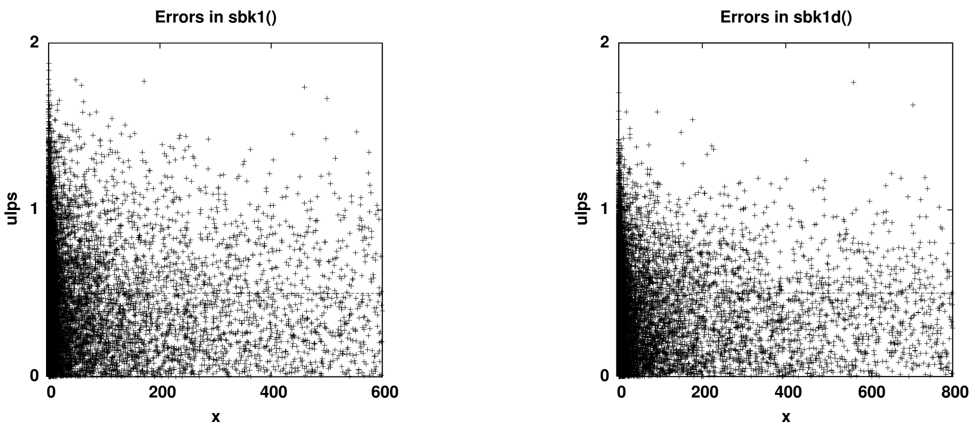


Figure 21.37: Errors in the binary (left) and decimal (right) `sbk1(x)` family.

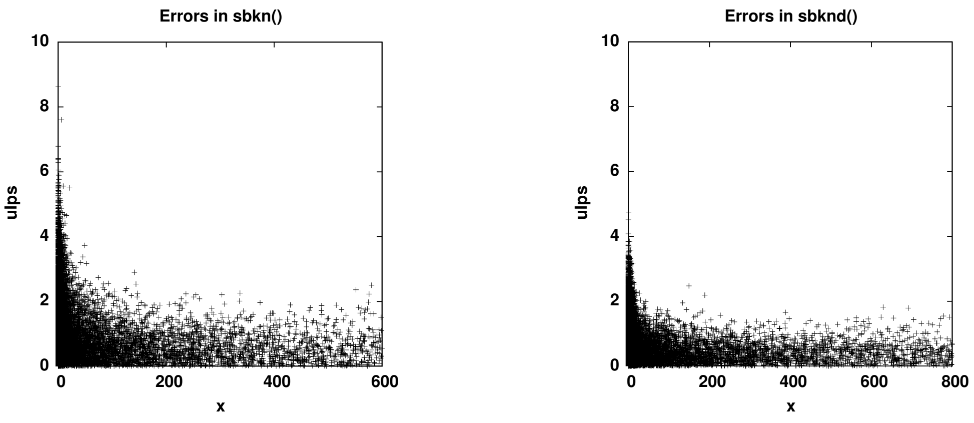


Figure 21.38: Errors in the binary (left) and decimal (right) `sbkn(n,x)` family for $n = 25$.

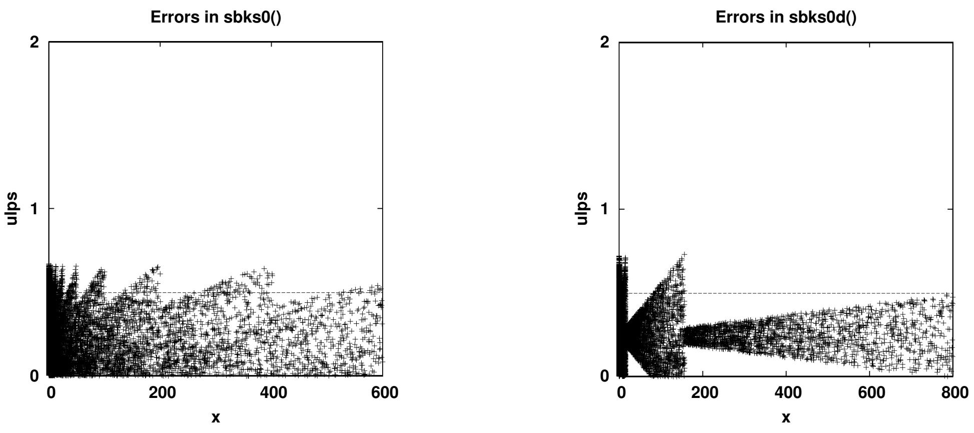


Figure 21.39: Errors in the binary (left) and decimal (right) `sbks0(x)` family.

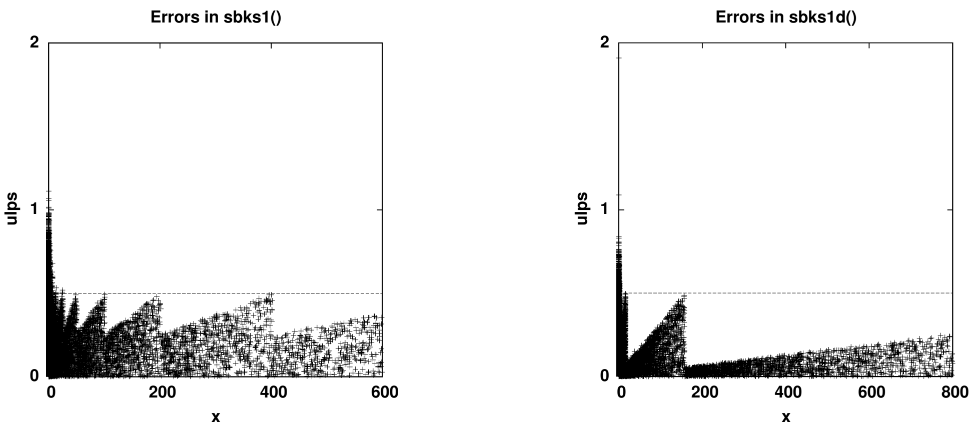


Figure 21.40: Errors in the binary (left) and decimal (right) `sbks1(x)` family.

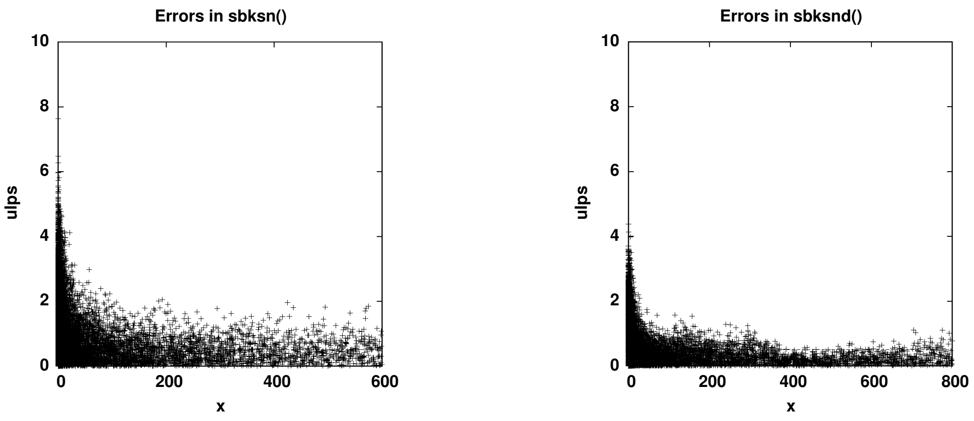


Figure 21.41: Errors in the binary (left) and decimal (right) `sbksn(n,x)` family for $n = 25$.

```
void vbj  (int n, double result[], double x); /* J(0..n,x) */
void vbk  (int n, double result[], double x); /* K(0..n,x) */
void vbks (int n, double result[], double x); /* K(0..n,x) * exp(x) */
void vby  (int n, double result[], double x); /* Y(0..n,x) */

void vsbi (int n, double result[], double x); /* i(0..n,x) */
void vsbis(int n, double result[], double x); /* i(0..n,x)*exp(-|x|) */
void vsbj (int n, double result[], double x); /* j(0..n,x) */
void vsbk (int n, double result[], double x); /* k(0..n,x) */
void vsbks(int n, double result[], double x); /* k(0..n,x) * exp(x) */
void vsby (int n, double result[], double x); /* y(0..n,x) */
```

Each of those functions has companions with the usual precision suffices. For example, after a call to the decimal_double function vsbyd(n,result,x), the array result[] contains the ordinary spherical Bessel function values $y_0(x), y_1(x), \ldots, y_n(x)$.

The vector functions follow a common design:

- Check for the condition $n < 0$, indicating an empty output array, in which case, set errno to ERANGE, and return without referencing the array. There is no provision for arrays of negative orders, even though such orders are well defined for all of the functions.
- Check for the special cases of x is a NaN, Infinity, and zero and handle them quickly, taking care to preserve symmetry properties for negative arguments, and handle signed zeros properly. Within a single case, the elements of the returned arrays have identical magnitudes, but may differ in sign.
- For functions that have complex values on the negative axis, if $x < 0$, set the array elements to the quiet NaN returned by QNAN(""), and set the global variable errno to EDOM.
- Handle the special cases of $n = 0$ and $n = 1$ by calling the corresponding scalar Bessel functions. We can then later assume that $n > 1$, and avoid bounds checks on storage into the output array.
- For small arguments, when the general Taylor series is known and has simple term recurrences, use it to compute all array entries. That is easily done by an outer loop over orders $m = 0, 1, 2, \ldots, n$, and an inner loop that sums series terms after the first to machine precision, after which the first term, and any outer scale factor, are incorporated. The last step may require special handling for systems that have wobbling precision.

  If there is an outer scale factor, such as $x^m/(2n+1)!!$, update it. If that factor overflows or underflows, the remaining array elements can be quickly supplied, and the outer loop exited early.

  To illustrate that description, here is the code block for the Taylor region for $I_n(x)$ in the file vbix.h:

```
else if (QABS(x) < FIVE)    /* x in (0,5): use Taylor series */
{
    fp_t f_m, scale, v, w;
    int m;

    v = x * HALF;
    w = v * v;
    scale = ONE;            /* k = 0: scale = v**m / m! */

    for (m = 0, f_m = ZERO; m <= n; ++m, ++f_m)
    {
        fp_t f_k, sum, t_k, u;
        int k;
        static const int MAX_TERMS = 43; /* enough for 70D */

        sum = ZERO;
        t_k = ONE;

        for (k = 1, f_k = ONE; k <= MAX_TERMS; ++k, ++f_k)
```

```
            {                   /* form sum = t_1 + t_2 + ... */
                fp_t new_sum;

                t_k *= w / (f_k * (f_k + f_m));
                new_sum = sum + t_k;

                if (new_sum == sum)
                    break;      /* converged: early loop exit */

                sum = new_sum;
            }

#if defined(HAVE_WOBBLING_PRECISION)
            u = HALF * scale;
            u = u + u * sum;    /* u = (1/2)*scale*(t_0 + ...) */
            result[m] = u + u;  /* scale * (t_0 + t_1 + ...) */
#else
            u = scale * sum;    /* scale * (t_1 + t_2 + ...)  */
            result[m] = scale + u; /* scale * (t_0 + t_1 + ...) */
#endif /* defined(HAVE_WOBBLING_PRECISION) */

            scale *= v / (f_m + ONE);

            if (scale == ZERO)  /* remaining elements underflow */
            {
                if (n > m)
                    VSET(n - m, &result[m + 1], ZERO);

                break;          /* early loop exit on underflow */
            }
        }
    }
```

Our algorithm leads to fast code, but for large n values, the use of repeated multiplication for the computation of an outer scale factor that involves a power and a factorial is *less accurate* than we could produce by calling our `IPOW()` function and using a stored table of correctly rounded factorials.

Separate handling of small arguments with Taylor series is *essential* for those Bessel functions, such as $J_n(x)$, that decay with increasing n and require downward recurrence. Otherwise, we could start the recurrence with zero values for $J_n(x)$ and $J_{n-1}(x)$, and all lower elements would then also be set to zero, producing unnecessary premature underflow in the algorithm.

- If upward recurrence is always stable, or x is sufficiently large compared to n that upward recurrence is safe, call the scalar functions for $n = 0$ and $n = 1$ and save their values in the output array, and then quickly fill the remainder of the array using the recurrence relation. The code block for $I_n(x)$ in the file `vbix.h` looks like this:

```
    else if (HALF * QABS(x) > (fp_t)n)
    {                /* n > 1: use stable UPWARD recurrence */
        fp_t f_k, two_over_x;

        two_over_x = TWO / x;
        result[0] = BI0(x);
        result[1] = BI1(x);

        if (ISINF(result[0]) || ISINF(result[1]))
        {
            fp_t inf_val;
            inf_val = INFTY();

            for (k = 2; k <= n; ++k)
                result[k] = ( (x < ZERO) && IS_ODD(k) ) ? -inf_val
                                                       : inf_val;
        }
        else
        {
            for (k = 1, f_k = ONE; k < n; ++k, ++f_k)
                result[k + 1] = QFMA(-f_k * two_over_x, result[k],
                                     result[k - 1]);
        }
    }
```

If x is large, then the reciprocal could underflow to subnormal or zero in IEEE 754 arithmetic. We could prevent that by replacing $(2k)/x$ by $(2ks)/(xs)$, with the scale factor s set to a suitable power of the base, such as ϵ^2. However, $I_n(x)$ grows quickly to the overflow limit, so we are unlikely to use x values large enough to require scaling, but infinite function values need special handling to avoid generating NaNs in later subtractions. The major source of accuracy loss in the recurrence is from subtractions near Bessel-function roots, and fused multiply-add operations can reduce that loss.

- Otherwise, use downward recurrence. Generate the first two array values by calls to the scalar functions of orders zero and one. Even though they could be also found from the recurrence, the scalar functions are likely to be more accurate, and because the initial terms are larger for some of the Bessel functions, that may be beneficial in the later use of the results. Call the general scalar functions for orders n and $n-1$ to get the last two array elements, and finally, use the downward recurrence relation to compute the intervening elements.

  Care is needed to prevent unnecessary generation of Infinity, and subsequent NaN values from subtractions of Infinity, or multiplications of Infinity by zero. To see how that is done, here is the code block for $I_n(x)$ in the file `vbix.h`:

```
    else                /* n > 1: use stable DOWNWARD recurrence */
    {
        fp_t two_k, one_over_x;

        one_over_x = ONE / x;

        if (ISINF(one_over_x)) /* prevent NaNs from Infinity * 0 */
            one_over_x = COPYSIGN(FP_T_MAX, x);

        result[0] = BI0(x);
        result[1] = BI1(x);
        result[n] = BIN(n, x);

        if (n > 2)
            result[n - 1] = BIN(n - 1, x);

        for (k = n - 1, two_k = (fp_t)(k + k); k > 2;
             --k, two_k -= TWO)
        {
            volatile fp_t b_k_over_x;

            b_k_over_x = one_over_x * result[k];
            STORE(&b_k_over_x);
            result[k - 1] = QFMA(two_k, b_k_over_x, result[k + 1]);
        }
    }
```

For speed, division by x is replaced by multiplication by its reciprocal, introducing another rounding error.

If $1/x$ overflows, as it can when x is near the underflow limit on important historical architectures such as the DEC PDP-10, PDP-11, and VAX, and the IBM System/360, and in IEEE 754 arithmetic if x is subnormal, replace it by the largest representable number of the correct sign to prevent later subtractions of Infinity that produce NaN values. Overflow cannot happen if we have already handled small arguments with Taylor series.

We avoid calling `BIN(n - 1, x)` if we already have its value.

Because the calls `BIN(n, x)` and `BIN(n - 1, x)` each generate sequences of Bessel function ratios that could be used to recover the complete function sequence, the work is repeated three times, and an improved implementation of the vector functions would copy that code from the scalar functions, or make it available from them via a separate private function.

We terminate the downward loop once `result[2]` has been generated.

In the loop, we use the `volatile` qualifier and the `STORE()` macro to prevent compilers from reordering the computation $(2k)((1/x)I_k(x))$ to $(1/x)((2k)I_k(x))$, possibly producing premature overflow because $I_k(x)$ can be large.

If either, or both, of $I_{n-1}(x)$ and $I_n(x)$ overflow to Infinity, then that overflow propagates down to $I_2(x)$, even though some of those elements could be finite. A more careful implementation would check for infinite elements generated by the downward recurrence, and then recompute them individually. In practice, applications that need that Bessel function for arguments large enough to cause overflow are better rewritten to use the scaled functions `BISN(n,x)` or `VBIS(n,result,x)`.

- Although the recurrences involve expressions like $2k/x$, throughout the code, we are careful to avoid, or minimize, integer-to-floating-point conversions. Although machines with hardware binary arithmetic can do such conversions quickly, conversions for decimal arithmetic are slower, especially if the arithmetic is done in software.

Comparison of results from the vector Bessel functions with those from the scalar Bessel functions shows two main problems: accuracy loss near Bessel-function zeros, and accuracy loss for large n from the use of repeated multiplication for powers and factorials. When those issues matter, it is better to stick with the slower scalar functions, possibly from the next higher precision, or else to call a vector function of higher precision, and then cast the results back to working precision.

21.19 Retrospective on Bessel functions

The large number of books and research articles on the computation of Bessel functions that we reported at the beginning of this chapter reflects the interest in, and importance of, those functions, as well as the difficulty in computing them accurately.

Many of the recurrence relations and summation formulas presented in this chapter are subject to loss of leading digits in subtractions, particularly for the ordinary Bessel functions, $J_\nu(z)$ and $Y_\nu(z)$, and their spherical counterparts, $j_\nu(z)$ and $y_\nu(z)$, when the argument is near one of their uncountably many roots.

Fortunately, for the Bessel functions that we treat, when upward recurrence is unstable, downward recurrence is stable, and vice versa. The continued fraction for the ratio of Bessel functions is an essential starting point for downward recurrence, but we saw in **Table 21.5** on page 712 that convergence of the continued fraction may be unacceptably slow for large values of the argument z.

The modified functions $I_\nu(z)$ and $K_\nu(z)$ have no finite nonzero roots, but they quickly reach the overflow and underflow limits of conventional floating-point number representations. Subtraction loss often lurks elsewhere, as we saw when the finite and infinite sums for $K_\nu(z)$ are added, and when the term $\log(v) + \gamma$ is computed for $Y_\nu(z)$ and $K_\nu(z)$.

The exponentially scaled functions, $\mathrm{Is}_\nu(z)$ and $\mathrm{Ks}_\nu(z)$, delay the onset of overflow and underflow, but do not prevent it entirely. In addition, the unscaled modified Bessel functions have argument ranges where the function values are representable, yet either, or both, of the exponential factor or the scaled function are out of range, making the unscaled function uncomputable without access to arithmetic of wider range, or messy intermediate scaling, or independent implementations of logarithms of the scaled modified Bessel functions.

The presence of two parameters, the order ν and the argument z, often requires separate consideration of the cases $|\nu| \gg z$ and $|\nu| \ll z$, and the computation time may be proportional to either ν or z.

The spherical Bessel functions of integer order have closed forms that require only trigonometric or hyperbolic functions, and as long as the underlying trigonometric functions provide exact argument reduction, as ours do, can be computed accurately for some argument intervals over the entire floating-point range. Unfortunately, for $n > 0$, subtraction loss is a common problem, and no obvious rewriting, as we did in **Section 21.3** on page 699 for the sines and cosines of shifted arguments, seems to provide a simple and stable computational route that guarantees low *relative* error, instead of low absolute error, for arguments near the function zeros. Higher working precision is then essential.

With complex arguments, and real or complex orders, all of those difficulties are compounded, and we have therefore excluded those cases from the mathcw library and this book.