Multiple-Precision Exponential Integral and Related Functions

David M. Smith

Loyola Marymount University

This article describes a collection of Fortran-95 routines for evaluating the exponential integral function, error function, sine and cosine integrals, Fresnel integrals, Bessel functions and related mathematical special functions using the FM multiple-precision arithmetic package.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General – *computer arithmetic, multiple precision arithmetic;* G.1.2 [**Numerical Analysis**]: Approximation – *special function approximation;* G.4 [**Mathematical Software**]: – *Algorithm design and analysis, efficiency, portability*

General Terms: Algorithms, exponential integral function, multiple precision

Additional Key Words and Phrases: Accuracy, function evaluation, floating point, Fortran, mathematical library, portable software

---

1. INTRODUCTION

The functions described here are high-precision versions of those in the chapters on the exponential integral function, error function, sine and cosine integrals, Fresnel integrals, and Bessel functions in a reference such as Abramowitz and Stegun [1965]. They are Fortran subroutines that use the FM package [Smith 1991; 1998; 2001] for multiple-precision arithmetic, constants, and elementary functions.

The FM package supports high precision real, complex, and integer arithmetic and functions. All the standard elementary functions are included, as well as about 25 of the mathematical special functions. An interface module provides easy use of the package through three multiple-precision derived types, and also extends Fortran's array operations involving vectors and matrices to multiple-precision. The routines in this module can detect an attempt to use a multiple precision variable that has not been defined, which helps in debugging.

There is great flexibility in choosing the numerical environment of the package. In addition to setting the precision level, the user can specify the internal base for the arithmetic and pick one of four rounding modes. There are routines to provide easy input and output of the multiple-precision numbers, and some new routines to support very high precision.

2. ACCURACY

The FM package provides an option for selecting one of four rounding modes as with the IEEE floating-point standard: round to nearest, toward $+\infty$, toward $-\infty$, toward zero. The default rounding mode is to round to the nearest representable number.

Correctly rounded results are obtained in each rounding mode for the basic arithmetic operations and all the elementary and special functions in the package. Upon entry to a routine the precision is raised enough so that the final result can usually be correctly rounded to the user's precision. If an algorithm is used that will encounter too much cancellation error for a given x, precision is raised again before starting that method to compensate for the extra precision loss.

For the rare cases where even more precision is lost during the calculation, such as x very close to a root of the function, or where the pre-rounded result is too close to the rounding threshold (i.e., 1/2 ulp for default rounding), precision is raised further and the function is computed again.

Previous versions of FM had a "nearly perfect" rounding mode as the default and an option for perfect rounding. The nearly perfect rounding mode used enough guard digits to insure the error in the pre-rounded result was less than 0.501 ulps (units in the last place of the post-rounded result). Perfect rounding gave errors less than or equal to 0.500 ulps, but was slower. In this version of the package, the average time penalty for perfect rounding has been reduced to less than 1%, so perfect rounding is always done.

The directed rounding modes are useful for applications such as interval arithmetic. Kahan [2006] shows they can also be useful, although not infallible, in assessing the numerical stability of algorithms, by checking results with different rounding modes and the same precision, along with the usual checking using the same rounding mode and different precisions.

The intrinsic Fortran vector and matrix operations provided in FM, such as DOT_PRODUCT, MATMUL, PRODUCT, and SUM work as Kahan [2004] suggests, raising precision to compute the sums and/or products at the higher precision, then rounding the final result back to the user's precision to provide a more accurate result.

3. EFFICIENCY

Most high precision applications use no more than 30 to 50 significant digits, but some need more. Personal computers are now fast enough and have large enough memory that carrying millions of digits is feasible. To support such a wide range of applications, the basic data structure for storing the multiple-precision numbers has changed. In previous versions of the package, each FM number was stored in a separate fixed-size array. Now the numbers share one dynamic array, which can save space and improve speed for the derived-type operations.

To support higher precision a routine for FFT-based multiplication has been included, and

when precision gets high enough the algorithms for multiplication, division, squares, square roots, etc., automatically switch to using the FFT routine.

Binary splitting algorithms are used for several mathematical constants at high precision. At the time this version of FM was written, computing a million digits of e, pi, the logarithm of a small integer, or an exact integer factorial took a few seconds, while a million digits of Euler's constant took a few minutes. High precision change-of-base conversions also use binary splitting.

For programs that read or write lots of data, input/output conversion time can be significant. FM uses a power of ten by default for the arithmetic base in the internal representation of multiple precision numbers. There are applications such as emulating the base 2 arithmetic of a given machine's hardware, where other bases would be chosen, but numbers in a power of ten base can be converted to or from base ten much faster than with other bases.

Tables I and II give some timing data for these functions. Table I gives an idea of the speed of the routines for commonly used precisions under 100 significant digits. Together the two tables show the times with 10^2, 10^3, and 10^4 digits to give a rough indication of the behavior of the algorithms at higher precision.

Timing benchmarks are not always entirely reliable, but to give some indication of function times, the FM times are compared with Mathematica 7.0 in the tables, since Mathematica is a widely-used commercial program that supports all of these functions at high precision. Because the two programs are written in different languages and compiled with different compilers, in cases where one program's time in the table is within a factor of 1.5 or 2 times the other, the difference could be due to particular compiler strengths or weaknesses, optimization settings, etc. For the times in the table, FM was compiled using the gfortran compiler.

Here are some considerations that were used in generating these times.

(1) Both programs were run on the same machine, a 2.66 Ghz Intel i7 Mac laptop, and both used only a single cpu.

(2) The times in the tables are the average over many calls to each routine. Since some routines store their last input and result, different x-values were used in each call. Many routines need constants like π or γ that are only computed once, then saved values are used on subsequent calls. Averaging over many calls more nearly replicates the performance in a typical application.

(3) Both programs ran the same set of x-values for each case.

(4) To test the method selection logic of each routine, the x-values ranged from about 10^{-3} to 10^3.

(5) The x-values were chosen to have no obvious pattern in their digits, since special values like small integers or $\sqrt{2}$ have mostly zero digits in many or all of the x^n terms that get multiplied in the formulas. This makes those multiplications uncharacteristically fast at high precision.

If n is the number of different x-values to be tested, the values are:

$$x_1 = \left(1000\,\gamma\right)^{-1}, \quad r = \left(10^6\,\gamma^2\right)^{1/(n-1)}, \quad x_j = r\,x_{j-1}\,, \quad j = 2, \ldots, n,$$

where $\gamma = 0.5772156649\ldots$ is Euler's constant. The values of n used in the tables were (200000, 100000, 10000, 1000) for (50, 100, 1000, 10000) digits.

Table I. Function timing at lower precision (seconds per call)

	50 S.D.		100 S.D.	
	FM	Mathematica	FM	Mathematica
Ei(x)	.000 129	.000 182	.000 251	.000 314
$E_1(x)$	.000 109	.000 112	.000 211	.000 189
Li(x)	.000 158	.000 141	.000 276	.000 219
Erf(x)	.000 043	.000 108	.000 080	.000 212
Erfc(x)	.000 076	.000 107	.000 143	.000 209
Ci(x)	.000 135	.000 463	.000 245	.000 781
Si(x)	.000 078	.000 478	.000 149	.000 797
Chi(x)	.000 128	.000 445	.000 237	.000 676
Shi(x)	.000 070	.000 448	.000 138	.000 679
C(x)	.000 056	.000 470	.000 104	.000 876
S(x)	.000 057	.000 471	.000 105	.000 877
$J_1(x)$	.000 111	.000 121	.000 198	.000 300
$Y_1(x)$	.000 318	.000 311	.000 571	.000 784

Table II. Function timing at higher precision (seconds per call)

	1,000 S.D.		10,000 S.D.	
	FM	Mathematica	FM	Mathematica
Ei(x)	.004 380	.005 740	.262	.753
$E_1(x)$	.004 980	.006 080	.265	.773
Li(x)	.004 380	.005 120	.305	.707
Erf(x)	.002 410	.010 600	.265	2.730
Erfc(x)	.004 790	.010 300	.878	2.710
Ci(x)	.003 000	.015 500	.185	1.510
Si(x)	.001 980	.015 500	.115	1.510

Chi(x)	.003 020	.013 500	.182	1.550
Shi(x)	.001 980	.013 500	.115	1.550
C(x)	.002 970	.032 500	.497	7.260
S(x)	.002 960	.032 500	.501	7.300
$J_1(x)$	.002 980	.020 300	.200	1.180
$Y_1(x)$	.009 030	.052 300	.587	3.410

4. SOME ALGORITHMS USED IN THE PACKAGE

Most of these algorithms select from among several possible methods for computing a given function value. The decision about which of these methods to use for a given x depends on the base for the arithmetic, b, and the current precision, t.

The boundaries of the regions where each method is used are determined by the relative efficiencies of the methods, as well as any restrictions on where a given method can achieve the required accuracy.

4.1 Exponential Integral Ei

$$\mathrm{Ei}(x) = \int_{-\infty}^{x} \frac{e^t}{t}\, dt\ ,$$

for the Cauchy principal value of the integral.

The convergent series is used for small $|x|$, the asymptotic series is used for large $|x|$, and the continued fraction expansion is used for intermediate negative values of x.

$$\begin{aligned}
\mathrm{Ei}(x) &= \gamma + \ln(|x|) + \sum_{n=1}^{\infty} \frac{x^n}{n\, n!} \\
&\sim \frac{e^x}{x} \sum_{n=0}^{\infty} \frac{n!}{x^n} \\
&= \cfrac{-e^x}{-x + \cfrac{1}{1 + \cfrac{1}{-x + \cfrac{2}{1 + \cfrac{2}{-x + \cdots}}}}}
\end{aligned}$$

FM's exponential integral function is defined for $x \neq 0$.

(1) Use the convergent series: $-14.3 - 0.275\,t\,\ln(b) < x \le (t+3)\ln(b) + \ln(2\pi(t+3)\ln(b))/2$

(2) Use the asymptotic series: $|x| > (t+3)\ln(b) + \ln(2\pi(t+3)\ln(b))/2$

(3) Use the continued fraction: $-((t+3)\ln(b)+\ln(2\pi(t+3)\ln(b))/2) \le x < -14.3-0.275\,t\,\ln(b)$

4.2 Exponential Integral $\mathbf{E}_n$

$$\mathrm{E}_n(x) = \int_1^\infty \frac{e^{-x\,t}}{t^n}\,dt\ .$$

For positive values of x and n, there is a convergent series,

$$\mathrm{E}_n(x) = \frac{(-x)^{n-1}\,\big(H_{n-1} - \gamma - \ln(x)\big)}{(n-1)!} + \sum_{\substack{j=0\\ j\neq n-1}}^{\infty} \frac{(-x)^j}{(n-1-j)\;j!}\,,$$

where H_{n-1} is the harmonic number $1 + 1/2 + \cdots + 1/(n-1)$, with $H_0 = 0$.

For values of $x > 0$, there is a continued fraction,

$$\mathrm{E}_n(x) = \cfrac{e^{-x}}{x + \cfrac{n}{1 + \cfrac{1}{x + \cfrac{n+1}{1 + \cfrac{2}{x + \cfrac{n+2}{1 + \cfrac{3}{x + \cdots}}}}}}}$$

For $n = 1$, $\mathrm{E}_1(x) = -\mathrm{Ei}(-x)$, and for values of $n > 1$ there is a recurrence involving $\mathrm{Ei}(x)$,

$$\mathrm{E}_n(x) = \frac{1}{(n-1)!}\left(-(-x)^{n-1}\mathrm{Ei}(-x) + e^{-x}\sum_{j=0}^{n-2}(n-j-2)!(-x)^j\right).$$

When $n = 0$, $\mathrm{E}_0(x) = e^{-x}/x$, and for values of $n < 0$ there is a recurrence involving e^x,

$$\mathrm{E}_n(x) = \frac{(-n)!\;e^{-x}}{x^{1-n}}\sum_{j=0}^{-n}\frac{x^j}{j!}.$$

For large values of n, this routine uses an incomplete gamma function,

$$\mathrm{E}_n(x) = x^{n-1}\,\Gamma(1-n, x).$$

As x gets large, the incomplete gamma function underflows before $\mathrm{E}_n(x)$, so the routine uses an asymptotic expansion,

$$\mathrm{E}_n(x) \sim \frac{e^{-x}}{x}\left(1 - \frac{(n)_1}{x} + \frac{(n)_2}{x^2} + \frac{(n)_3}{x^3} + \cdots\right),$$

where $(n)_k = n(n+1)(n+2)\cdots(n+k-1)$ is Pochhammer's symbol.

The domain is $x \neq 0$ for integer $n \leq 0$, and $x > 0$ for integer $n > 0$

(1) Use the convergent series: $n > 0$ and $0 < x < \dfrac{t\ln(b)}{5.5 + 0.00095n} + \dfrac{\big(t\ln(b)\big)^2\big(4.22 + \ln(n)\big)}{10^4\big(24 + \ln(n)\big)}$

(2) Use the Ei recurrence: $1 \leq n < \max(2, (t+3)\ln(b)/32)$ and $x > (t+5)\ln(b) + \ln\big(2\pi(t+5)\ln(b)\big)/2$

(3) Use the continued fraction: $n > 0$ and x outside the range in (1) and (2)

(4) Use the e^x recurrence: $-0.032\,t\ln(b) - 71 < n < 0$

(5) Use incomplete gamma: n outside the range in (1) through (4)

(6) Use the asymptotic series: when the incomplete gamma underflows in (5).

4.3 Logarithmic Integral

$$\mathrm{Li}(x) = \int_0^x \frac{1}{\ln(t)}\,dt\,.$$

The logarithmic integral is computed using the exponential integral routine,

$$\mathrm{Li}(x) = \mathrm{Ei}(\ln(x)).$$

The domain is $x \geq 0$, with $x \neq 1$.

4.4 Error Function

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$$

The convergent series is used for small $|x|$, the second convergent series is used for intermediate $|x|$, and the continued fraction expansion is used for large values of $|x|$.

$$\begin{aligned}
\mathrm{erf}(x) &= \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(-1)^n\, x^{2n+1}}{n!\,(2n+1)} \\
&= \frac{e^{-x^2}}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{n!\,(2x)^{2n+1}}{(2n+1)!}, \\
&= 1 + \cfrac{-e^{-x^2}/\sqrt{\pi}}{x + \cfrac{1}{2x + \cfrac{2}{x + \cfrac{3}{2x + \cfrac{4}{x + \cdots}}}}}
\end{aligned}$$

The domain for $\mathrm{erf}(x)$ is all x.

(1) Use the first convergent series: $|x| \le \sqrt{(t+21)\ln(b)}/5$

(2) Use the second convergent series: $\sqrt{(t+21)\ln(b)}/5 < |x| \le ((t+6)\ln(b))^{0.6}/4.1$

(3) Use the continued fraction: x outside the range in (1) and (2).

The complimentary error function is defined by

$$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2}\, dt \;= 1 - \mathrm{erf}(x).$$

The erfc routine calls erf for all x except large positive values, when the continued fraction is used:

$$\cfrac{e^{-x^2}/\sqrt{\pi}}{x + \cfrac{1}{2x + \cfrac{2}{x + \cfrac{3}{2x + \cfrac{4}{x + \cdots}}}}}$$

The domain for $\mathrm{erfc}(x)$ is all x.

(1) Use $1 - \mathrm{erf}(x)$: $x \le ((t+6)\ln(b))^{0.7}/9$

(2) Use the continued fraction: $x > ((t+6)\ln(b))^{0.7}/9$

Because erfc underflows for values of x larger than a fairly small threshold, around $x = 30,000$ for most machines, a function that returns $\ln(\mathrm{erfc}(x))$ is provided so that larger x values can be

handled. For moderate values of x, this function calls erfc. For larger values it uses a continued fraction,

$$\ln(\mathrm{erfc}(x)) = \ln(u) - x^2 - \ln(x) - \ln(\sqrt{\pi}), \quad \text{where}$$

$$u = \cfrac{x}{x + \cfrac{1/2}{x + \cfrac{1}{x + \cfrac{3/2}{x + \cfrac{2}{x + \cdots}}}}}$$

This function is designed to be called only for $x > 1000$, where the continued fraction is used. For smaller values it calls erfc(x), except for x near zero, where there is cancellation in $\ln(\mathrm{erfc}(x))$, so with $u = \mathrm{erf}(x)$ a series is used,

$$\ln(\mathrm{erfc}(x)) = \ln(1-u) = -u - u^2/2 - u^3/3 - \cdots.$$

4.5 Sine Integral and Cosine Integral

$$\begin{aligned}
\mathrm{Si}(x) &= \int_0^x \frac{\sin(t)}{t}\, dt, \\
\mathrm{Ci}(x) &= \gamma + \ln(x) + \int_0^x \frac{\cos(t)-1}{t}\, dt.
\end{aligned}$$

These routines use a convergent series or an asymptotic series,

$$\begin{aligned}
\mathrm{Si}(x) &= \sum_{n=0}^{\infty} \frac{(-1)^n\, x^{2n+1}}{(2n+1)\,(2n+1)!} \\
&= \frac{\pi}{2} - f(x)\,\cos(x) - g(x)\,\sin(x),
\end{aligned}$$

$$\begin{aligned}
\mathrm{Ci}(x) &= \gamma + \ln(x) + \sum_{n=1}^{\infty} \frac{(-1)^n\, x^{2n}}{2n\,(2n)!} \\
&= f(x)\,\sin(x) - g(x)\,\cos(x).
\end{aligned}$$

The two auxiliary functions have these expansions:

$$\begin{aligned}
f(x) &\sim \frac{1}{x}\left(1 - \frac{2!}{x^2} + \frac{4!}{x^4} - \frac{6!}{x^6} + \cdots\right), \\
g(x) &\sim \frac{1}{x^2}\left(1 - \frac{3!}{x^2} + \frac{5!}{x^4} - \frac{7!}{x^6} + \cdots\right).
\end{aligned}$$

The domain for Si(x) is all x, and for Ci(x) is $x > 0$.

(1) Use the convergent series: $|x| \le (t+5)\ln(b) + (\ln(2\pi) + \ln((t+5)\ln(b))/2$

(2) Use the asymptotic series: $|x| > (t+5)\ln(b) + (\ln(2\pi) + \ln((t+5)\ln(b))/2$

4.6 Hyperbolic Sine Integral and Hyperbolic Cosine Integral

$$\begin{aligned} \mathrm{Shi}(x) &= \int_0^x \frac{\sinh(t)}{t}\, dt \\ \mathrm{Chi}(x) &= \gamma + \ln(x) + \int_0^x \frac{\cosh(t)-1}{t}\, dt \end{aligned}$$

These algorithms are similar to those for the sine integral and cosine integral.

$$\begin{aligned} \mathrm{Shi}(x) &= \sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)\,(2n+1)!} \\ &= f(x)\,\cosh(x) + g(x)\,\sinh(x) \\ \mathrm{Chi}(x) &= \gamma + \ln(x) + \sum_{n=1}^{\infty} \frac{x^{2n}}{2n\,(2n)!} \\ &= f(x)\,\sinh(x) + g(x)\,\cosh(x) \end{aligned}$$

with these auxiliary functions:

$$\begin{aligned} f(x) &\sim \frac{1}{x}\left(1 + \frac{2!}{x^2} + \frac{4!}{x^4} + \frac{6!}{x^6} + \cdots\right) \\ g(x) &\sim \frac{1}{x^2}\left(1 + \frac{3!}{x^2} + \frac{5!}{x^4} + \frac{7!}{x^6} + \cdots\right) \end{aligned}$$

The domain for Shi(x) is all x, and for Chi(x) is $x > 0$.

(1) Use the convergent series: $|x| \le (t+5)\ln(b) + (\ln(2\pi) + \ln((t+5)\ln(b))/2$

(2) Use the asymptotic series: $|x| > (t+5)\ln(b) + (\ln(2\pi) + \ln((t+5)\ln(b))/2$

4.7 Fresnel Integrals

The two Fresnel integrals are defined by

$$\mathrm{C}(x) = \int_0^x \cos\left(\frac{\pi\, t^2}{2}\right)\, dt,$$

$$\mathrm{S}(x) = \int_0^x \sin\left(\frac{\pi\, t^2}{2}\right)\, dt.$$

These routines use a convergent series or an asymptotic series.

$$\begin{aligned}\mathrm{C}(x) &= x \sum_{n=0}^{\infty} \frac{(-1)^n\, (\pi x^2/2)^{2n}}{(4n+1)\, (2n)!} \\ &= \frac{1}{2} + f(x)\, \sin(\pi x^2/2) - g(x)\, \cos(\pi x^2/2),\end{aligned}$$

$$\begin{aligned}\mathrm{S}(x) &= x \sum_{n=0}^{\infty} \frac{(-1)^n\, (\pi x^2/2)^{2n+1}}{(4n+3)\, (2n+1)!} \\ &= \frac{1}{2} - f(x)\, \cos(\pi x^2/2) - g(x)\, \sin(\pi x^2/2).\end{aligned}$$

The two auxiliary functions have these expansions:

$$f(x) \sim \frac{1}{\pi x}\left(1 - \frac{1\cdot 3}{(\pi x^2)^2} + \frac{1\cdot 3\cdot 5\cdot 7}{(\pi x^2)^4} - \frac{1\cdot 3\cdots 11}{(\pi x^2)^6} + \cdots\right),$$

$$g(x) \sim \frac{1}{\pi^2 x^3}\left(1 - \frac{1\cdot 3\cdot 5}{(\pi x^2)^2} + \frac{1\cdot 3\cdots 9}{(\pi x^2)^4} - \frac{1\cdot 3\cdots 13}{(\pi x^2)^6} + \cdots\right).$$

The domain for both C(x) and S(x) is all x. Let $v = (\pi x^2 - 1)/2$.

(1) Use the convergent series:
$(2v+1.5)\ln(2v+2) - (2v+1) - v\ln(2) - (v+0.5)\ln(v+1) + v - v\ln(2v+1) \le -(t+1)\ln(b)$

(2) Use the asymptotic series:
$(2v+1.5)\ln(2v+2) - (2v+1) - v\ln(2) - (v+0.5)\ln(v+1) + v - v\ln(2v+1) > -(t+1)\ln(b)$

4.8 Bessel Functions

The Bessel functions $J_n(x)$ and $Y_n(x)$ are solutions to the differential equation

$$x^2\, y'' + x\, y' + (x^2 - n^2)\, y = 0.$$

For small values of x, there is a convergent series,

$$J_n(x) = \sum_{k=0}^{\infty} \frac{(-1)^k\, x^{2k+n}}{k!\, (n+k)!\, 2^{2k+n}}.$$

For large values of x, J is computed from an asymptotic series,

$$J_n(x) \sim \sqrt{\frac{2}{\pi\, x}}\left(\cos(x - n\pi/2 - \pi/4) \sum_{k=0}^{\infty} a_k \;-\; \sin(x - n\pi/2 - \pi/4) \sum_{k=0}^{\infty} b_k\right),$$

where

$$a_k = \frac{(-1)^k\ \Gamma(2k+n+1/2)}{(2x)^{2k}\ (2k)!\ \Gamma(-2k+n+1/2)},$$

$$b_k = \frac{(-1)^k\ \Gamma(2k+n+3/2)}{(2x)^{2k+1}\ (2k+1)!\ \Gamma(-2k+n-1/2)}.$$

These terms are computed by simplifying the gamma ratios to get

$$a_{k+1} = \frac{-\left(1+4\,k-2\,n\right)\ \left(3+4\,k-2\,n\right)\ \left(1+4\,k+2\,n\right)\ \left(3+4\,k+2\,n\right)}{128\ \left(1+k\right)\ \left(1+2\,k\right)\ x^2}\ a_k,$$

$$b_{k+1} = \frac{-\left(3+4\,k-2\,n\right)\ \left(5+4\,k-2\,n\right)\ \left(3+4\,k+2\,n\right)\ \left(5+4\,k+2\,n\right)}{128\ \left(1+k\right)\ \left(3+2\,k\right)\ x^2}\ b_k,$$

with $a_0 = 1$ and $b_0 = (4\,n^2-1)/(8x)$.

For method selection in $J_n(x)$, the asymptotic series uses two series to one for the convergent series, and they are slower, since the terms are more complicated. But in the region where we switch from method 1 to 2, method 1 has to use higher precision to compensate for cancellation.

Define k_1 to be the smallest number of terms where the convergent series gets sufficient accuracy, and k_2 to be the smallest number of terms where the asymptotic series gets sufficient accuracy (with $k_2 = \infty$ if the asymptotic series never reaches current precision). Then k_1 is the minimum k such that

$2k\ \ln(|x|) - (2k+n)\ln(2) - \ln(\Gamma(k+1)) - \ln(\Gamma(n+k+1)) <$

$n(\ln(|x|) - \ln(2)) - \ln(\Gamma(n+1)) - t\ln(b) - 46.05,$

and k_2 is the minimum k such that

$\ln(\Gamma(n+k+1/2)) - (n-k)\ln(\Gamma(|n-k|+1/2))/|n-k| - k\ \ln(2) - k\ \ln(|x|) - \ln(\Gamma(k+1)) <$

$\min(0, \ln(|4n^2-1|) - \ln(|x|) - \ln(8)) - t\ln(b) - 96.05.$

The domain for $J_n(x)$ is all x.

(1) Use the convergent series: $k_1 \le k_2$

(2) Use the asymptotic series: $k_1 > k_2$

The convergent series used for Y is

$$Y_n(x) = \frac{-1}{\pi}\ \sum_{k=0}^{\infty}\ \frac{(-1)^k\ \left(\psi(k+1)+\psi(n+k+1)\right)\ x^{2k+n}}{k!\ (n+k)!\ 2^{2k+n}}$$

$$+\ \ \frac{2}{\pi}\ \ln(x/2)\ J_n(x) - \frac{1}{\pi}\ \sum_{k=0}^{n-1}\ \frac{(n-k-1)!\ x^{2k-n}}{k!\ 2^{2k-n}}\ .$$

The ψ terms are updated in this sum using $\psi(k+2) = \psi(k+1) + 1/(k+1)$, with $\psi(1) = -\gamma$.

For large values of x, Y is computed from an asymptotic series,

$$Y_n(x) \sim \sqrt{\frac{2}{\pi\, x}} \left(\sin(x - n\pi/2 - \pi/4) \sum_{k=0}^{\infty} a_k \;\; + \;\; \cos(x - n\pi/2 - \pi/4) \sum_{k=0}^{\infty} b_k \right),$$

where a_k and b_k are the same as in the asymptotic series for $J_n(x)$.

For method selection in $Y_n(x)$, define k_1 to be the smallest number of terms where the convergent series gets sufficient accuracy, and k_2 to be the smallest number of terms where the asymptotic series gets sufficient accuracy (with $k_2 = \infty$ if the asymptotic series never reaches current precision). Then k_1 is the minimum k such that

$(n + 2k)\,(\ln(|x|) - \ln(2)) - \ln(\Gamma(k+1)) - \ln(\Gamma(n+k+1)) <$

$n(\ln(|x|) - \ln(2)) - \ln(\Gamma(n+1)) - t\ln(b) - 46.05,$

and k_2 is the minimum k such that

$\ln(\Gamma(n+2k+1/2)) - (n-2k)\ln(\Gamma(|n-2k|+1/2))/|n-2k| - 2k\ \ln(2) - 2k\ \ln(|x|) - \ln(\Gamma(2k+1)) <$

$\min(0, \ln(|4n^2 - 1|) - \ln(|x|) - \ln(8)) - t\ln(b) - 96.05.$

The domain for $Y_n(x)$ is $x > 0$.

(1) Use the convergent series: $k_1 \le k_2$

(2) Use the asymptotic series: $k_1 > k_2$

5. SOME AUXILIARY ALGORITHMS USED IN THE PACKAGE

5.1 Continued Fractions

Many of the special functions use continued fraction expansions for part of their domains. The standard method for evaluating a continued fraction

$$b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cdots}}}$$

is to use the recurrence

$$P_0 = 1, \qquad Q_0 = 0$$

$$P_1 = b_0, \qquad Q_1 = 1$$

$$P_k = b_{k-1}\,P_{k-1} + a_{k-1}\,P_{k-2}, \qquad Q_k = b_{k-1}\,Q_{k-1} + a_{k-1}\,Q_{k-2} \quad \text{for } k > 1.$$

Then P_k/Q_k is the sequence of approximations to the continued fraction. This requires four multiplications and two additions for each term, plus one division (which need not be done every step), to check convergence.

For multiple precision calculations, one drawback to this method is that all the operations must be done at full precision. On the other hand, while summing a series where the terms get smaller, the next term in the series can often be computed at lower precision than the final result requires. This might speed up the entire calculation by a factor of two or three.

Since the continued fraction calculation can be expressed in terms of an equivalent series, we need to compare the work required for the series. Define

$$S_k = \frac{P_k}{Q_k} - \frac{P_{k-1}}{Q_{k-1}}.$$

Then we can write a given continued fraction quotient as

$$\frac{P_n}{Q_n} = b_0 + \sum_{k=2}^{n} S_k.$$

To simplify the expression for S_k, we have

$$\begin{aligned} S_k &= \frac{b_{k-1}\,P_{k-1} + a_{k-1}\,P_{k-2}}{b_{k-1}\,Q_{k-1} + a_{k-1}\,Q_{k-2}} - \frac{P_{k-1}}{Q_{k-1}} \\ &= \frac{a_{k-1}\,(P_{k-2}\,Q_{k-1} - P_{k-1}\,Q_{k-2})}{Q_k\;Q_{k-1}} \\ &= \frac{-a_{k-1}\;Q_{k-2}}{Q_k}\;S_{k-1} \end{aligned}$$

This gives a recurrence for S that involves only the Q terms. It uses three multiplications, one division, and two additions for each term in the sum. Of these operations, all but one addition can be done at reduced precision. This means the number of basic operations is the same as with the standard method, but in this form the decreasing precision summation technique is faster.

5.2 FFT Multiplication

When precision exceeds about 10,000 significant digits, the fast Fourier transform can be used to multiply two numbers more quickly than the classical $O(n^2)$ method. If x and y are numbers with n digits base b, then after scaling the exponents we can write

$$x = \sum_{k=1}^{n} x_k\,b^{-k} \qquad \text{and} \qquad y = \sum_{k=1}^{n} y_k\,b^{-k}.$$

The inputs to the Fourier transform are complex arrays, and storing the real input arrays directly as complex numbers with all imaginary parts zero would waste time and space. So the lists of digits

$\{x_k\}$ and $\{y_k\}$ are packed in complex form as $\{x_1{+}x_2\,i\,,\ x_3{+}x_4\,i\,,\ \ldots\}$ and $\{y_1{+}y_2\,i\,,\ y_3{+}y_4\,i\,,\ \ldots\}$, giving two complex arrays of length $n/2$. Storing the digits this way means that some extra steps must be done packing and unpacking intermediate results before and after the Fourier transform steps, to recover the convolution products of the original input lists, but that is still faster than using the unpacked form.

In practice, the length of these complex arrays, N, is chosen to be slightly larger than n, since the product of two n-digit numbers has $2n$ digits and the FFT is faster when the array length is a product of small primes. The elements past $n/2$ are initialized to zero.

Two FFT operations are done to get the transforms of these two arrays in $O(n\,\log(n)\,)$ time. The transformed arrays are unpacked and the element-by-element product is computed in $O(n)$ time.

Then this product is packed for input to the final FFT. The result of this final transform, z, has the digits of the original product xy in scrambled order. These digits can each be as big as nb^2 here, since the coefficients of each different power of b in the product are

$$x_1\,y_1, \quad (x_1\,y_2 + x_2\,y_1), \quad \cdots \quad (x_1\,y_n + x_2\,y_{n-1} + \cdots x_n\,y_1), \quad \cdots \quad x_n\,y_n.$$

The final step is to unscramble their order and to normalize the digits so that in base b form each digit is less than b.

When precision is high enough, the size of the digits being convolved must be reduced in order to keep the convolution products from being too big to exactly recover the integer results. The FFT operation has double precision rounding errors, but the result of the convolution of two lists of integers is really an integer.

For example, assume double precision carries 53 bits giving about 16 significant digit accuracy, all the (positive) numbers in the two lists $\{x_k\}$ and $\{y_k\}$ are less than b, and there are n numbers in each list. Then each element of the convolution is an integer less than nb^2. A typical case might have x and y in base 10^7 with 50,000 digits for about 350,000 significant digit precision. This means $b = 10^7$ and $n = 5 * 10^4$, so $nb^2 = 5 * 10^{18}$. That is too big to recover exact integers using 16-digit double precision.

Depending on the base being used, FM uses one of two methods to reduce the size of the elements in the input lists.

Method 1 is used if the base is a power of a small number ($b = s^j$ for $2 \le s \le 19$). Then we can change x and y to a base that is a smaller power of s to reduce the size of the individual digits. Changing to this smaller base is a fast $O(n)$ operation.

In the example above, x and y can be changed to numbers with about 117,000 digits in base 10^3. The original x with $b = 10^7$ and $n = 50,000$ might be

$$x = 1234567/b + 2345678/b^2 + \cdots.$$

The new x with $b = 10^3$ would be

$$x = 123/b + 456/b^2 + 723/b^3 + \cdots .$$

The new $nb^2 = 1.2 * 10^{11}$, so even after losing 2 or 3 digits to rounding in the FFT the results can be reliably rounded to the nearest integer. This is the method used for the default FM power-of-ten base.

Method 2 is used if a fast change to a smaller base is not available. Each of x and y is split into two pieces. Each piece is a number with the same base and precision that x and y have, but with artificially small digits.

Suppose the base is near 10^7 but not a power, say $b = 12345678$, and x is

$$x = 1234567/b + 2345678/b^2 + ... + 9876543/b^n.$$

Let $k = \text{int}(\sqrt{b}) = 3513$ be the upper bound on the size of the digits in the two pieces, $x1$ and $x2$. We write $x = x1 + k * x2$ by defining the digits of $x1$ to be the digits of x mod k, and the digits of $x2$ to be the digits of x divided by k. That gives

$$x1 = 1504/b + 2507/b^2 + ... + 1500/b^n,$$

$$x2 = 351/b + 667/b^2 + ... + 2811/b^n.$$

Now, $x * y = (x1 + k * x2)(y1 + k * y2) = x1 * y1 + k * (x1 * y2 + x2 * y1) + k^2 * x2 * y2$.

Since the digits of $x1$ and $x2$ are formed one at a time from the corresponding digits of x, generating $x1$ and $x2$ is a fast $O(n)$ operation. The terms in these products are still written in base b, but the digits are small, no more than k. These four multiplications are reduced to three, computing $x1*y1$, $x2*y2$, and $(x1+x2)*(y1+y2)$. Then $x1*y2+x2*y1 = (x1+x2)*(y1+y2)-x1*y1-x2*y2$. See Knuth [1998], section 4.3.3.

Method 2 is recursive, since if n is large enough, $n\,k^2$ may still be too large for the double precision rounding errors. In that case another splitting is done, giving digits less than $\sqrt{\text{int}(\sqrt{b})} = 59$ in this example.

For $b = 12345678$ and 53-bit double precision, the first splitting is done for all n, since $b^2 > 10^{14}$ is already too close to 16-digit integers. A second splitting is done for n larger than about $4 * 10^5$ (about 2.8 million decimal digits), and a third for $n > 1.4 * 10^9$ (about 10 billion decimals).

5.3 Binary Splitting

This is a recursive "divide and conquer" algorithm, going back to Brent [1976]. It has been used more often in the last decade as computers have become fast enough to make very high precision calculations more easily available. Haible and Papanikolaou [1998] give several examples.

The basic idea is to break a sum of rational numbers into pieces and use multiple-precision integer arithmetic to build up the numerator and denominator of the sum in such a way that most of the operations involve fairly small integers, so they are fast.

In general, suppose we want to compute the sum

$$S = \sum_{n=0}^{N} \frac{a(n)\ p(0) \cdots p(n)}{b(n)\ q(0) \cdots q(n)}$$

where $a(j)$, $b(j)$, $p(j)$, and $q(j)$ are all fairly small integers. Define

$$S(n_1, n_2) = \sum_{n=n_1}^{n_2} \frac{a(n)\ p(n_1) \cdots p(n)}{b(n)\ q(n_1) \cdots q(n)}$$

and let

$$\begin{aligned}
P(n_1, n_2) &= p(n_1) \cdots p(n_2) \\
Q(n_1, n_2) &= q(n_1) \cdots q(n_2) \\
B(n_1, n_2) &= b(n_1) \cdots b(n_2) \\
T(n_1, n_2) &= B(n_1, n_2)\ Q(n_1, n_2)\ S(n_1, n_2) \\
&= \sum_{n=n_1}^{n_2} a(n)\ \frac{B(n_1, n_2)}{b(n)}\ P(n_1, n)\ Q(n+1, n_2)
\end{aligned}$$

where $Q(j+1, j)$ is defined to be 1. These four values are all integers. If $n_2 - n_1$ is less than some small threshold, usually around 10 or 20, compute these values directly from the definitions above. For larger values of $n_2 - n_1$ let $m = \mathrm{int}((n_1 + n_2)/2)$ be the midpoint and use the recurrence relations below.

$$\begin{aligned}
P(n_1, n_2) &= P(n_1, m-1)\ P(m, n_2), \\
Q(n_1, n_2) &= Q(n_1, m-1)\ Q(m, n_2), \\
B(n_1, n_2) &= B(n_1, m-1)\ B(m, n_2), \\
T(n_1, n_2) &= B(m, n_2)\ Q(m, n_2)\ T(n_1, m-1) + B(n_1, m-1)\ P(n_1, m-1)\ T(m, n_2).
\end{aligned}$$

This algorithm is applied recursively starting with $n_1 = 0$ and $n_2 = N$. Then we do one final floating-point calculation to get S,

$$S = S(0, N) = \frac{T(0, N)}{B(0, N)\ Q(0, N)}.$$

For a specific example, consider computing the constant e to one million digits from the series

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}\ .$$

Stirling's formula gives a value of N to use for the sum, around $N = 206{,}000$. For this sum, the terms are quite simple, having $a(n) = 1$, $b(n) = 1$, $p(n) = 1$, and $q(n) = n$, with the special case $q(0) = 1$.

This means $P(n_1, n_2) = 1$ and $B(n_1, n_2) = 1$ for all values of n_1 and n_2, which leaves only Q and T to compute. The direct formulas for small $n_2 - n_1$ are

$$\begin{aligned} Q(0, n_2) &= 1 \cdots n_2, \\ Q(n_1, n_2) &= n_1 \cdots n_2, \quad \text{if } n_1 > 0, \\ T(n_1, n_2) &= ((n_1 + 1) \cdots n_2) + ((n_1 + 2) \cdots n_2) + \cdots + n_2 + 1. \end{aligned}$$

The recursive formulas are

$$\begin{aligned} Q(n_1, n_2) &= Q(n_1, m - 1)\ Q(m, n_2), \\ T(n_1, n_2) &= Q(m, n_2)\ T(n_1, m - 1) + T(m, n_2), \end{aligned}$$

and so the million-digit value of e is $T(0, 206000)/Q(0, 206000)$.

6. CONCLUSION

There are many multiple precision packages available. Commercial algebra systems like Mathematica and Maple provide high precision arithmetic and functions. Packages in C include GMP [2010] and MPFR [2007; 2010]. Other Fortran packages include Bailey's [1993] and Brent's [1978].

One typical use of FM is to begin with an existing Fortran program and check accuracy by converting that program to multiple precision using 30 to 100 digits of accuracy. FM makes it very easy to do that, and so does Bailey's MP. Bailey's package is excellent and can often be used as a double check with FM, but does not support most of the special functions discussed in this paper.

Using the reduced-precision summation technique for multiple precision evaluation of continued fractions is apparently new, as is the FFT splitting method used when the base is not a power of a small integer.

Many of these functions need constants like π or γ, and the binary splitting methods are used for these even at fairly low precision. At high precision, FM has been used to compute up to 200 million digits of π as part of the testing of the FFT algorithms and binary splitting.

FM is designed to provide an easy-to-use multiple precision environment for Fortran programs. It includes the numerical Fortran intrinsic functions, as well as intrinsic inquiry functions for the current base, precision, largest and smallest exponent, etc. FM supports the standard Fortran array syntax for vector and matrix operations, and has very similar output formatting options.

The special functions in FM extend Fortran's capabilities to some of the most commonly occurring advanced functions in math, science, and engineering applications.

References

Abramowitz, M., and Stegun, I.A. (Eds.) 1965. *Handbook of Mathematical Functions*, Dover, New York.

Bailey, D.H.. 1993. Multiprecision Translation and Execution of Fortran Programs *ACM Trans. Math. Softw. 19*, 3 (September), 288-319.

Brent, R.P. 1976. The Complexity of Multiple-Precision Arithmetic, in *The Complexity of Computational Problem Solving*, (edited by R.S. Anderson and R.P. Brent), University of Queensland Press, Brisbane.

Brent, R.P. 1978. A Fortran Multiple-Precision Arithmetic Package. *ACM Trans. Math. Softw. 4*, 1 (March), 57–70.

GNU Multiple Precision Arithmetic Library 2010. http://gmplib.org/

Haible, B. and Papanikolaou, T. 1998. Fast Multiprecision Evaluation of Series of Rational Numbers. *Lecture Notes in Computer Science 1423*, 338–352.

Kahan, W. 2004. Matlab's Loss is Nobody's Gain. http://www.cs.berkeley.edu/∼wkahan/MxMulEps.pdf

Kahan, W. 2006. How Futile are Mindless Assessments of Roundoff in Floating-Point Computation? http://www.cs.berkeley.edu/∼wkahan/Mindless.pdf

Knuth, D.E. 1998. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, third edition. Addison Wesley, Reading, Mass.

MPFR 2007. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Trans. Math. Softw. 33*, 2, Article 13 (June), 15 pages

MPFR Library 2010. http://www.mpfr.org/

Smith, D.M. 1991. A Fortran Package for Floating-Point Multiple-Precision Arithmetic. *ACM Trans. Math. Softw. 17*, 2 (June), 273–283.

Smith, D.M. 1998. Multiple Precision Complex Arithmetic and Functions. *ACM Trans. Math. Softw. 24*, 4 (December), 359–367.

Smith, D.M. 2001. Multiple Precision Gamma Function and Related Functions. *ACM Trans. Math. Softw. 27*, 4 (December), 377–387.