A Fortran Package For Floating-Point Multiple-Precision Arithmetic

David M. Smith

Loyola Marymount University

FM is a collection of Fortran-77 routines which performs floating-point multiple-precision arithmetic and elementary functions. Results are almost always correctly rounded, and due to improved algorithms used for the elementary functions, reasonable efficiency is obtained.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General – *computer arithmetic;* G.1.2 [**Numerical Analysis**]: Approximation – *elementary function approximation;* G.4 [**Mathematics of Computing**]: Mathematical Software – *Algorithm analysis, efficiency, portability*

General Terms: Algorithms, arithmetic, multiple precision

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

1. INTRODUCTION

FM is a package of Fortran subroutines which performs floating-point multiple-precision arithmetic. Many such packages have been written, and FM provides several improvements to previous packages in the areas of speed, accuracy, exception handling, and maintainability of the code.

Brent's MP package [6] is probably the most widely used of these packages at present, due to its greater functionality and efficiency. The FM routines give comparable speed at low precision and greater speed at higher precision, and provide better rounding properties and exception handling. This increased speed comes mainly from the use of improved algorithms for computing the elementary functions in multiple-precision.

The FM package supports all the standard Fortran-77 intrinsic functions. Results are almost always correctly rounded, and exceptions are handled in a way which minimizes the chance of getting results which appear accurate but are not.

2. DESIGN OF THE PACKAGE

An FM number is represented in base b having t significant digits using $t+1$ words of an integer array. The first word contains the exponent, and words 2 through $t+1$ contain the t normalized base b digits. The sign of the number is carried on the second word. Zero is represented by words 1 and 2 being zero. The implied radix point is left of the first significant digit. The base may be any integer b greater than 1 such that b^2 is a representable integer. If M is the largest integer on a given machine, then the exponent range is defined by $m = M/\log M$. The overflow threshold is b^{m+1} and the underflow threshold is b^{-m-1}. This allows FM routines to temporarily extend the overflow/underflow threshold and simplifies detection of operations which will overflow or underflow. The only restriction on the size of t is the memory size of the machine.

For applications where space is critical, an option is provided to work with FM numbers which are packed two digits per word. This uses less than half the space for storing FM numbers, but the time spent packing and unpacking the numbers slows execution speed, especially in operations like addition, where it may take longer to unpack the arguments and pack the result than to perform the operation. This is less noticeable for functions like $\sin(x)$, because only one pack and unpack operation is done; when the sine routine calls other arithmetic routines, those arguments are kept unpacked.

Commonly used constants such as π, e, etc., are stored at the highest precision previously computed for a given base. These constants are then available without having to compute them again unless precision is later raised above the previous maximum.

The FM package is written in Fortran-77, and the use of features such as `IF-THEN-ELSE` blocks makes the routines much more readable than the code of previous packages written in Fortran-66.

Arithmetic is done on FM numbers using subroutine calls. The assignment `C = A + B` is done using `CALL FMADD(MA,MB,MC)`. A precompiler such as AUGMENT [7] can be used to convert a program to one using the corresponding FM calls. To keep the calling sequences simple, variables such as the base, precision, exponent range, etc., as well as scratch arrays used for holding temporary results in the package are stored in `COMMON`.

Two subroutines are provided which return characteristic values related to the current base and precision being used. The first returns the largest FM number less than the overflow threshold, and the second is a function which returns the number which represents 1 unit in the last place (ulp) of the input argument. Using these, it is easy to compute other base and precision dependent values such as the smallest positive FM number, "machine epsilon", or the next FM number bigger than a given one.

3. ACCURACY

Because the amount of precision being carried can be changed during the course of a sequence of operations, it is easy to carry guard digits so that the inevitable rounding errors are unlikely to propagate and cause the result to be wrong. Upon entry to an FM routine, the current precision is raised enough to make it likely that the final result will be correctly rounded when precision is restored at the end of the routine. It is not always easy to know in advance how many guard digits will be required, and the precision may be raised again in the routine if a partial result indicates that more digits are needed.

The goal of the FM routines is to compute results which are accurate to within 0.001 ulp before rounding to the current precision. This gives an error after rounding of no more than 0.501 ulp and means that for random input arguments no more than one in a thousand results will be incorrectly rounded. Here "correctly rounded" means that assuming the input arguments are exact, the result returned is the nearest FM number to the exact result, with rounding to an even last digit if the exact result lies halfway between two adjacent FM numbers. In the tradeoff between speed and accuracy the goal above can be achieved with only a slight time penalty. The extra time spent computing guard digits is most noticeable at low precision, and becomes almost negligible at high precision.

Many previous multiple-precision packages did not have good accuracy for the elementary functions. Since the user could easily raise the precision before calling the routine, responsibility for determining the number of guard digits was left to the user. Errors of tens of thousands of ulps were common when using a large base to evaluate elementary functions with these packages. Carrying two or three guard digits contained the error in most but not all of these cases. For a given function evaluation the number of guard digits needed depends on the base, precision, and input argument(s).

For example, Kahan [8] shows that computing tan(52174) is difficult because of cancellation. To illustrate the use of guard digits in the package, consider three computations using $b = 10,000$ and $t = 10$:

a. tan(0.3) This argument presents no problems due to cancellation. The "standard" number of guard digits used in FM is given by $5/\log_{10}(b) + 2$. This is three here, so the tangent calculation is done using $t = 13$ and then the result is rounded to ten digits.

b. tan(52173) Here some accuracy will be lost due to cancellation when the argument is reduced. The argument reduction is essentially done modulo $\pi/4$. Since the base b exponent of the argument is two, two additional guard digits are used. This means the argument reduction is done using $t = 15$, after which the main tangent calculation proceeds using $t = 13$.

c. tan(52174) As before the argument is reduced using $t = 15$. The result is 52174 mod $\pi/4 \approx$ 5.5E$-$6. The exponent base b of the reduced value is -1, indicating that more than the expected amount of cancellation has occurred. The argument reduction is done again with $t = 16$ and then the tangent calculation for the reduced argument uses $t = 13$.

Since the FM package is designed to give good performance when t is not extremely large, the formula for guard digits above does not depend on t. For more than about ten thousand significant digits a formula involving t would be needed for guard digits, and different algorithms would need to be used for multiplication, division, and most elementary functions.

Kahan [8, 9] has pointed out that even if rounding is only slightly sloppy, it can sometimes lead to highly inaccurate results. He also notes that it is a great boon to the user to know that the results are correctly rounded. The fact that identities are true and bounds on the errors are known simplifies any analysis of a computation enough to justify a small time penalty. Black, Burton, and Miller [3] reach similar conclusions in discussing the accuracy of elementary function routines.

4. EFFICIENCY

It takes extra time to compute guard digits and to check for different kinds of exceptions, but by using efficient algorithms for the high level functions and by doing the kind of code tuning described by Bentley [1, 2], the FM routines compare favorably with previous multi-precision packages in terms of speed.

Since the fundamental operations used to perform the FM operations are done using one-word integer arithmetic, the main way to gain speed is to use a large base. If $b = 10,000$ then four base 10 digits are stored in a word. Because of normalization, the first significant digit of the number may have only one base 10 digit, so $b = 10,000$ and $t = 26$ is roughly equivalent to $b = 10$ and $t = 101$.

Because of a code tuning trick used in multiplication and division it is faster to use a base which is only 1/4 to 1/2 as big as the largest possible base. Another consideration in choosing the base is that if b is a power of ten, then input and output conversion between base b and base ten is much faster. For these reasons, the best base is often the largest power of ten less than the square root of the largest one-word integer. This is 10,000 on most 32-bit machines.

In [5] Brent gives algorithms for the elementary functions which are asymptotically faster than those used in FM. Because these algorithms are more complicated they are slower than the FM functions unless t is very large. Some tests were done which indicated that only for precisions over ten thousand digits would the $O(\log t\, M(t))$ algorithm for $\exp(x)$ be faster than the $O(t^{1/3}\, M(t))$ version used in FM.

Table I gives timing results for a variety of FM routines and compares the times for the same calls using Brent's MP package. Times are given in seconds and are average times for a number of different arguments. Timing runs were made using $b = 10,000$ on a Macintosh IIcx microcomputer (a Motorola 68030-based machine running at 15.67 MHz). Times are rounded to about three significant digits, and will change slightly for different compilers or different sets of test numbers. The effect of internal guard digits is most evident at $t = 10$, since the FM routines are computing these functions using three more digits than MP.

Table 1. Timing comparisons

	$t = 10$		$t = 50$		$t = 250$	
	FM	MP	FM	MP	FM	MP
add	.0027	.0005	.0034	.0012	.0071	.0050
multiply	.0049	.0030	.0350	.0384	.7150	.8130
divide	.0140	.0259	.0705	.1650	.9030	2.6200
sqrt(x)	.0354	.0411	.1420	.2310	1.2200	3.3500
exp(x)	.0881	.1070	.8140	1.5100	24.0000	56.1000
ln(x)	.1860	.1610	1.4000	1.9300	32.2000	80.7000
sin(x)	.0914	.1230	.6980	1.8200	18.6000	83.4000
$\tan^{-1}(x)$	.2210	.2400	1.5400	4.5500	29.5000	307.0000

5. EXCEPTION HANDLING

Some previous multiple-precision packages print a message and halt execution when a condition like overflow, zero-divide, or negative argument to square root occurs. Others replace underflowed results with zero and set overflowed results to the largest possible value. FM tries to keep the program running when exceptions occur, but not to fall victim to inaccurate results like those which can come from setting underflows to zero.

Since the FM exponent range is large, overflow and underflow should not happen very often. For example, if $b = 10,000$ on a 32-bit computer, then the overflow threshold is about $10^{400,000,000}$. However, Kahan [10] has said that this can sometimes make overflow or underflow more likely, as people try more ambitious calculations for which there is less intuitive feel for the size of intermediate results.

FM defines special symbols to stand for signed overflow, signed underflow, and unknown results. Operations involving these symbols are defined so as to make results safe from the usual uncertainties present when exceptions have occurred. A result of "`UNKNOWN`" is returned whenever the result cannot be computed accurately or cannot definitely be placed in one of the four underflow or overflow categories. Here are a few examples:

```
3 + (+OVERFLOW) = +OVERFLOW
EXP(+OVERFLOW) = +OVERFLOW
1/(-OVERFLOW) = -UNDERFLOW
2/(+OVERFLOW) = UNKNOWN
(+OVERFLOW)/2 = UNKNOWN
0.4 + (-UNDERFLOW) = 0.4
SQRT(+UNDERFLOW) = UNKNOWN
COS(-UNDERFLOW) = 1
```

The user can control the action taken when an exception does occur. The options available include doing nothing and continuing execution, printing a warning message and continuing, or printing a message and stopping the program. A flag variable in common is set to a value indicating which exception has occurred. This can be tested by the user's program upon return from any FM routine. Any exceptional results which are converted by FM for output are printed as "`+OVERFLOW`", etc.

Since underflow is distinct from zero, programs which compare numbers by subtracting and comparing the result to zero can be converted to FM calls automatically without danger of underflow giving rise to incorrect comparisons.

A logical function is provided for making direct comparison of two FM numbers. It allows the same six types of comparisons as Fortran's logical IF statement, and is faster than subtracting and testing the sign of the result. Cases where the order of the two arguments cannot be determined, such as whether two overflowed results are equal, are treated similarly to other exceptions.

6. ARITHMETIC TRACING

It is often useful to be able to trace the execution of a calculation, and FM provides an option for automatic tracing of FM calls. The user can print input arguments and results for each operation, or just results. The trace gives the routine being called, call level within the FM package, and current values of the base and precision. The FM numbers may be printed either in formatted base ten form or in unformatted base b form where the integer array elements are printed. The trace can be set to print all FM calls to a specified call level within the package.

7. INPUT/OUTPUT

Two routines are provided for input and output. The input subroutine converts a character string to an FM number with the current base and precision. The conversion is free-format, and the number sent in the character string can be in any integer, fixed-point, single, double, quadruple, or FM format.

The output subroutine converts an FM number to a character string. The formats available correspond to Fortran's I, F, E, and 1PE formats. The number of digits displayed can be specified for each type.

Both subroutines take advantage of the fact that the conversion can be done much more quickly when b is a power of ten. The time is $O(t)$ for these values of b, and $O(t^2)$ for other values of b.

For printing one FM number per line a subroutine is available which automatically generates the output subroutine call and prints it under the currently specified format.

8. SOME ALGORITHMS USED IN FM

8.1 Addition and subtraction

The standard $O(t)$ algorithm is used essentially as in Knuth's description [11].

8.2 Multiplication and division

The $O(t^2)$ method is used because it faster for small and moderate t. Of the various methods [11] which are asymptotically faster, the two which seem most promising when t is of moderate size are the two-piece recursion method and the integer FFT method. The two-piece recursion method is $O(t^{\log_2 3}) \approx O(t^{1.58})$, and the FFT methods are $O(t \log t)$. Tests have shown that the $O(t^2)$ method is faster unless t is larger than a few thousand.

The speed of these operations is improved by minimizing the time spent normalizing partial results. When the base used is not too big the partial results need not be normalized after each step, since we can guarantee that integer overflow cannot happen on the next step. The smaller the base, the longer normalization may be postponed. In Brent's MP the base is restricted so that $8b^2 - 1$ is representable. This allows normalization to be done only once for each eight steps. In FM larger values of b are allowed, and the program computes during the operation whether the next step can be done before normalizing. This uses the digits actually being multiplied, instead of worst-case upper bounds. For example, on a 32-bit machine if $b = 10,000$ and t is large then normalization is done only about once each 40 steps.

In division it is more difficult to postpone normalization, since some normalization must be done in order to see if the trial quotient digit is correct. FM normalizes the first few digits of the partial result at each step and tries to confirm that the correct quotient digit has been chosen. Except in rare cases, examining these digits proves that the quotient digit is correct, and then for the rest of the partial result normalization can be postponed in a way similar to multiplication.

Instead of using integer arithmetic to select a trial quotient as in Knuth [11], FM uses single-precision arithmetic to divide the next few digits and select it. This gives a more accurate estimate, so it reduces the probability of having to make a correction step, and it eliminates the need to scale the numerator and denominator before starting the division. This long-division version is faster than the Newton iteration division algorithm used in MP [6], unless precision is high and a multiplication method is used which is faster than $O(t^2)$.

In subsequent algorithm descriptions, $M(t)$ will be used to denote the time required for t-digit multiplication.

8.3 Multiplication and division by small integers

Since these operations occur frequently and can be done in $O(t)$ time, FM has separate subroutines for them.

8.4 Square roots and other inverse functions

When doing multiple-precision Newton iteration, as in square root, logarithm, and arctangent, the precision is increased at each iteration so that only the final iteration is done using the full desired accuracy. The initial approximation can be generated quickly using double precision arithmetic and then the precision is almost doubled at each iteration. Thus all the iterations can be done in less than twice the time required for the last iteration. See Brent [4] for further details.

A subroutine is provided for calculating the sequence of precisions used for any Newton solution of a simple root using a double precision starting point.

8.5 Summing series

When summing a series with decreasing terms, as the terms get smaller, they need not be computed to the same precision being carried for the sum. The precision can be reduced to contain only those digits needed in the sum while computing the next term in the series. For example, in summing the Taylor series for $e^{1.2}$ with a precision of 50 digits (base 10), if the last term added was about 10^{-20} in magnitude and the next term is smaller, then it can be computed using only 30 digits. Any further digits would be shifted off the end and lost when the next addition is done.

In most series, computing the next term from the previous one may account for most of the time spent, since it involves multiplications, divisions, or other "slow" operations. For these, the time to sum the series may be reduced by a factor of two or three using this technique.

8.6 Argument reduction

Function computations can often be speeded up by using various identities to reduce the size of the argument prior to summing a series and then reversing the reduction at the end. For example, the exponential identity

$$\exp(x) = \exp(x/2^k)^{2^k}$$

can be used as follows. Compute $y = x/2^k$ using a few divide by integer operations, then sum the series for $\exp(y)$, then do k squarings to recover $\exp(x)$. In the tradeoff between the faster convergence of the series versus the overhead to reduce x and recover $\exp(x)$ the optimal value of k is $O(t^{1/2})$. This gives an algorithm with speed $O(t^{1/2}\,M(t))$. See Brent [4] for details.

8.7 Concurrent series

Many power series consist of terms which are closely related so that the next term can be obtained from the previous term by a few operations involving small integers and one $O(M(t))$ operation to get the next power of x. Since the operations with integers and the addition of the terms are all $O(t)$, reducing the number of multiplications is important. Computing the direct sum

$$\exp(x) \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

requires $(n-1)$ multiplications, $(n-1)$ divisions by an integer, and n additions. The sum can be rearranged as j concurrent sums

$$\begin{array}{llll} & 1 & +x^j/j! & +x^{2j}/(2j)! + \cdots \\ +x & [\,1 & +x^j/(j+1)! & + \cdots \\ +x^2 & [\,1/2! & +x^j/(j+2)! & + \cdots \\ +x^3 & [\,1/3! & +x^j/(j+3)! & + \cdots \\ \vdots & & & \vdots \\ +x^{j-1} & [\,1/(j-1)! & +x^j/(2j-1)! & + \cdots \end{array}$$

This now requires $n/j + j - 1$ multiplications (plus $O(\log j)$ multiplications to get x^j) and has the same number of divisions and additions as the direct sum.

The increase in efficiency is lessened somewhat due to the fact that in the direct series all the multiplications can be done at reduced precision (see section 8.5), while in the second form $j-1$ of the multiplications must be done at full precision. The optimal value of j is $O(t^{1/2})$ and using this method gives $\exp(x)$ in time $O(t^{1/2}\,M(t))$.

When argument reduction and concurrent series can both be used, the optimal values of both j and k are $O(t^{1/3})$, and the running time of the function is $O(t^{1/3}\,M(t))$. As it is used in the FM package the extra space required by this method amounts to $O(t^{1/3})$ FM numbers, which is $O(t^{4/3})$ integer words. This concurrent sum technique has apparently not been used in previous multiple-precision packages. See [14] for details.

8.8 Exponential function

For general arguments the exponential routine uses the techniques above and has running time $O(t^{1/3}\,M(t))$. The argument is split into integer and fraction parts: $x = n + f$. Starting with the stored value of e, $\exp(n)$ is done in $O(\log n)$ multiplications using the integer power routine, and $\exp(f)$ is computed as shown in the examples above.

When $\exp(1) = e$ is computed, the standard series is used. The time is $O(t^2)$, since computing the terms of the series uses only operations involving small integers.

8.9 Logarithms

The natural logarithm is done using Newton's method and $\exp(x)$, so the time required is $O(t^{1/3} M(t))$. For values of x close to 1, the Taylor series is used. This avoids cancellation and is faster unless t is large.

The routine for base 10 logarithms uses $\ln x / \ln 10$, so it has the same asymptotic running time and is slightly slower.

A routine is provided for computing $\ln(n)$. The integer n is approximated by the nearest integer m of the form $2^i\, 3^j\, 5^k\, 7^l$, and $\ln(m)$ is obtained as a linear combination of $\ln(125/126)$, $\ln(224/225)$, $\ln(2400/2401)$, and $\ln(4374/4375)$. These four values are computed in time $O(t^2)$ and then saved so they do not have to be recomputed on subsequent calls. Then $\ln(n/m)$ is computed, so that $\ln(n) = \ln(n/m) + \ln(m)$.

Thus if n itself has only 2,3,5,7 as prime factors, $\ln(n)$ is computed in $O(t)$ time, otherwise the time required is $O(t^2)$.

The general logarithm routine checks to see if the input argument can be scaled to a small integer by multiplying or dividing by a power of b. If so then the faster integer logarithm routine is used.

8.10 Power functions

For computing x^n where n is an integer the integer power routine uses the binary multiplication method and the time is $O(M(t) \log n)$.

The general x^y function is computed using $\exp(y \ln x)$ unless y can be expressed exactly as a one word integer. In that case, the integer power routine is used. This is usually much faster, and it allows x to be negative when y is an integer.

8.11 Trigonometric functions

For $\sin(x)$ the argument is first reduced to lie between 0 and $\pi/4$ using various identities. Then this value is further reduced by dividing by 3^k, and then the Taylor series is added as j concurrent sums in a manner similar to $\exp(x)$. After summing the series, $\sin(x)$ is recovered by k iterations of the formula $\sin(3a) = 3\sin(a) - 4\sin^3(a)$.

In [4] Brent suggests the argument reduction $y = x/2^k$ and the corresponding recovery formula $\sin(2a) = 2\sin(a)\,(1 - \sin^2(a))^{1/2}$. The triple angle formula used in FM is faster and can be used for smaller values of t.

For $\cos(x)$ and $\tan(x)$ the sine routine and identities are used. The time required for the trigonometric functions is $O(t^{1/3} M(t))$.

FM provides an option under which all angles in trigonometric functions and their inverses are given in degrees. This is sometimes more convenient for the user, and it allows the first part of the argument reduction to be done in degrees.

To compute π, FM uses Ramanujan's identity [12]

$$\frac{1}{\pi} = \frac{\sqrt{8}}{9801} \sum_{n=0}^{\infty} \frac{(4n)!\ (1103 + 26390n)}{(n!)^4\ 396^{4n}}$$

The series is summed using only multiplications and divisions by small integers, so the time is $O(t^2)$.

8.12 Inverse trigonometric functions

The arctangent is computed using Newton iteration and the sine function, unless the argument is very small, when the Taylor series is used. The other inverse functions use $\tan^{-1}(x)$ and identities. The time required is $O(t^{1/3}\, M(t))$.

8.13 Hyperbolic functions

These are computed using $\exp(x)$, except when x is small. Then the accurate value of $\exp(x)-1$ which is calculated in the exponential routine is used. This avoids cancellation in $\sinh(x)$ and $\tanh(x)$.

9. TESTING

Several different types of testing have been done. Many randomly generated arguments for each function have been used and function values (at low precision) compared to values obtained using Fortran's double precision intrinsic functions. To test accuracy, results from FM calls were compared to values obtained from the same computation done at higher precision. Tests were also made comparing FM's results to those of Brent's MP. Some special constants like π were compared to published values to high precision [13]. Tables of constants to 40 places [11] were also used.

Testing has been done using several different machines, with values for base and precision ranging from $b = 2$, $t = 2$ to $b = 3,037,000,499$, $t = 256$. The package is coded to provide portability and allow maintenance. All machine-dependent values are set in a single, well-documented subroutine.

Two small test programs are included to help install the FM package. Main program `TEST` uses FM to evaluate 11 of the constants given to 40 decimal places in Knuth [11]. FM numbers are kept in packed format. Program `ROOTS` solves for the roots of the 10th degree Legendre polynomial to 50 decimals. It uses unpacked FM numbers and performs variable-precision Newton iteration.

References

1. Bentley, J.L. *Programming Pearls.* Addison Wesley, Reading, Mass., 1986.

2. Bentley, J.L. *Writing Efficient Programs.* Prentice-Hall, Englewood Cliffs, N.J., 1982.

3. Black, C.M., Burton, R.P., and Miller, T.H. The Need for an Industry Standard of Accuracy for Elementary-Function Programs. *ACM Trans. Math. Software 10*, 4 (December 1984), 361-366.

4. Brent, R.P. The Complexity of Multiple-Precision Arithmetic. In *Complexity of Computational Problem Solving*, R.S. Anderssen and R.P. Brent, Eds., U. of Queensland Press, Brisbane, 1976, pp. 126-165.

5. Brent, R.P. Fast Multiple-Precision Evaluation of Elementary Functions. *J. ACM 23*, 2 (April 1976), 242-251.

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

7. Crary, F.D. A Versatile Precompiler for Nonstandard Arithmetics. *ACM Trans. Math. Software 5*, 2 (June 1979), 204-217.

8. Kahan, W.M. And Now For Something Completely Different: The TI SR-52. U.C. Berkeley Electronics Research Lab Report UCB/ERL M77/23.

9. Kahan, W.M. Can You Count on Your Calculator? U.C. Berkeley Electronics Research Lab Report UCB/ERL M77/21.

10. Kahan, W.M. Implementation of Algorithms. U.C. Berkeley Computer Science Technical Report 20, 1973. Also distributed by National Technical Information Service under DDC AD-769 124.

11. Knuth, D.E. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, second edition. Addison Wesley, Reading, Mass., 1981.

12. Ramanujan, S. Modular equations and approximations to π. *Quart. J. Math.* 45 (1914), 350-372. Also in *Collected papers of Srinivasa Ramanujan*, G.H. Hardy, P.V. Seshu Aiyar, and B.M. Wilson, Eds., Cambridge University Press, 1927, pp. 23-39.

13. Shanks, D., and Wrench, J.W. Calculation of π to 100,000 places. *Math. Comp.* 16 (1962) 76-99.

14. Smith, D.M. Efficient Multiple-Precision Evaluation of Elementary Functions. *Math. Comp.* 52 (1989) 131-134.