# A Portable High Performance Multiprecision Package

David H. Bailey

RNR Technical Report RNR-90-022
May 18, 1993

**Abstract**

The author has written a package of Fortran routines that perform a variety of arithmetic operations and transcendental functions on floating point numbers of arbitrarily high precision, including large integers. This package features (1) virtually universal portability, (2) high performance, especially on vector supercomputers, (3) advanced algorithms, including FFT-based multiplication and quadratically convergent algorithms for $\pi$ and transcendental functions, and (4) extensive self-checking and debug facilities that permit the package to be used as a rigorous system integrity test. Converting application programs to run with these routines is facilitated by an automatic translator program.

This paper describes the routines in the package and includes discussion of the algorithms employed, the implementation techniques, performance results and some applications. Notable among the performance results is that this package runs up to 40 times faster than another widely used package on a RISC workstation, and it runs up to 400 times faster than the other package on a Cray supercomputer.

The author is with the NAS Applied Research Branch, NASA Ames Research Center, Moffett Field, CA 94035. E-mail: `dbailey@nas.nasa.gov`.

## 1. Applications of Multiprecision Computation

One question that is always raised in discussions of multiprecision computation is what applications justify such a facility. In fact, a number of applications, some of them entirely practical, have surfaced in recent years.

One important area of applications is in pure mathematics. While some still dispute whether a computer calculation can be the basis of a formal mathematical proof, certainly computations can be used to explore conjectures and reject those that are not sound. Such computations can thus save pure mathematicians a great deal of time by allowing them not to waste time searching for proofs of false notions. On the other hand, it must be acknowledged that if a conjecture is confirmed by computation to very high precision, then its validity is extremely likely. One can even ask which is more firmly established, a theorem whose lengthy proof has been read only by two or three people in the world, or a conjecture that has been independently confirmed by a number of high precision computations?

Some particularly nice applications of high precision computation to pure mathematics include the disproof of the Mertens conjecture by A. M. Odlyzko and H. J. J. te Riele [26], the disproof of the Bernstein conjecture in approximation theory by Varga and Carpenter and the resolution of the "one-ninth" conjecture [31]. A number of other examples of multiprecision applications in analysis, approximation theory and numerical analysis are also described in [31].

One area in which multiprecision computations are especially useful is the study of mathematical constants. For example, although Euler's constant $\gamma$ is believed to be transcendental, it has not been proven that $\gamma$ is even irrational. There is similar ignorance regarding other classical constants, such as $\log \pi$ and $e + \pi$, and also regarding some constants that have arisen in twentieth century mathematics, such as the Feigenbaum $\delta$ constant $(4.669201609 \cdots)$ [13, 18] and the Bernstein $\beta$ constant $(0.2801694990 \cdots)$ [31].

However, in most of these cases algorithms are known that permit these numbers to be computed to high precision. When this is done, the hypothesis of whether a constant $\alpha$ satisfies some reasonably simple, low-degree polynomial can be tested by computing the vector $(1, \alpha, \alpha^2, \cdots, \alpha^{n-1})$ and then applying one of the recently discovered integer relation finding algorithms [5, 19, 20]. Such algorithms, when applied to an $n$-long vector $x$, determine whether there exist integers $a_i$ such that $\sum a_i x_i = 0$. Thus if a computation finds such a set of integers $a_i$, these integers are the coefficients of a polynomial satisfied by $\alpha$. Even if no such relation is found, these algorithms also produce bounds within which no relation can exist, which results are of interest by themselves. Clearly such analysis can be applied to any constant that can be computed to sufficiently high precision.

The author has performed some computations of this type [4, 5], and others are in progress. Some recent results include the following: if $\gamma$ satisfies an integer polynomial of degree 50 or less, then the Euclidean norm of the coefficients must exceed $7 \times 10^{17}$; if Feigenbaum's $\delta$ constant satisfies an integer polynomial of degree 20 or less, then the Euclidean norm of the coefficients must exceed $2 \times 10^{15}$. This last result is based on joint work with K. Briggs of the University of Melbourne in Australia.

Computations of this sort have also been applied to study a certain conjecture regarding

the $\zeta$ function. It is well known [8] that

$$\begin{aligned}
\zeta(2) &= 3\sum_{k=1}^{\infty}\frac{1}{k^2\binom{2k}{k}} \\
\zeta(3) &= \frac{5}{2}\sum_{k=1}^{\infty}\frac{(-1)^{k-1}}{k^3\binom{2k}{k}} \\
\zeta(4) &= \frac{36}{17}\sum_{k=1}^{\infty}\frac{1}{k^4\binom{2k}{k}}
\end{aligned}$$

These results have led some to suggest that

$$Z_5 = \zeta(5)/\sum_{k=1}^{\infty}\frac{(-1)^{k-1}}{k^5\binom{2k}{k}}$$

might also be a simple rational or algebraic number. Unfortunately, the author and K. Briggs have established that if $Z_5$ satisfies a polynomial of degree 25 or less, then the Euclidean norm of the coefficients must exceed $2 \times 10^{37}$.

In one case the author, working in conjunction with H. R. P. Ferguson, obtained the following positive result: the third bifurcation point of the chaotic iteration $x_{k+1} = rx_k(1-x_k)$, namely the constant $3.54409035955\cdots$, satisfies the polynomial $4913+2108t^2-604t^3-977t^4+8t^5+44t^6+392t^7-193t^8-40t^9+48t^{10}-12t^{11}+t^{12}$. In this case, it can be proven fairly easily that this constant must be algebraic. The fact that it satisfies a polynomial of only degree 12 is something of a surprise.

One of the oldest applications of multiprecision computation is to explore the perplexing question of whether the decimal expansions (or the expansions in any other radix) of classical constants such as $\pi, e, \sqrt{2}$, etc. are random in some sense. Almost any reasonable notion of randomness could be used here, including the property that every digit occurs with limiting frequency $1/10$, or the stronger property that every $n$-long string of digits occurs with limiting frequency $10^{-n}$. This conjecture is believed to hold for a very wide range of mathematical constants, including all irrational algebraic numbers and the transcendentals $\pi$ and $e$, among others. Its verification for a certain class of constants would potentially have the practical application of providing researchers with provably reliable pseudorandom number generators. Unfortunately, however, this conjecture has not been proven in even a single instance among the classical constants of mathematics. Thus there is continuing interest in computations of certain constants to very high precision, in order to see if there are any statistical anomalies in the results. The computation of $\pi$ has been of particular interest in this regard, and recently the one billion digit mark was passed by both Kanada [22] and the Chudnovskys [15], and the Chudnovskys have more recently extended their calculation to beyond two billion digits [16]. Statistical analyses of these results have so far not yielded any statistical anomalies (see for example [2]).

An eminently practical application of multiprecision computation is the emerging field of public-key cryptography, in particular research on the Rivest-Shamir-Adleman (RSA)

cryptosystem [27, 17]. This cryptosystem relies on the exponentiation of a large integer to a large integer power modulo a third large integer. The RSA cryptosystem has also spawned a great deal of research into advanced algorithms for factoring large integers, since the RSA system can be "broken" if one can factor the modulus. The most impressive result in this area so far is the recent factorization of the ninth Fermat number $2^{512}+1$, an integer with 155 digits, which was accomplished by means of numerous computer systems communicating by electronic mail. This computation employed a new factoring algorithm, known as the "number field sieve" [25].

An indirect application of multiprecision computation is the integrity testing of computer systems. A unique feature of multiprecision computations is that they are exceedingly unforgiving of hardware or compiler error. This is because if even a single computational error occurs, the result will almost certainly be completely incorrect after a possibly correct initial section. In many other scientific computations, a hardware error in particular might simply retard the convergence to the correct solution.

Not only do the final results constitute an integrity test, but there are several consistency checks that can be performed in the course of a multiprecision computation. One of these derives from the fact that when performing multiprecision multiplication using a complex fast Fourier transform (FFT), the final inverse FFT results should be quite close to integer values. If any result exceeds a nominal distance from an integer value, a hardware error has likely occurred. Details are given in section 5.

The author's experience using current and prior versions of this package has confirmed this principle. Systems in which hardware or software errors have been disclosed by the author's routines include the following: the Cray-2, the Cray X-MP, the Cray Y-MP, the CDC 205, the ETA-10, the DEC VAX, the Silicon Graphics Personal IRIS, the Sun-4 and the Intel iPSC-860. This list includes virtually all of the computer systems that the author has worked with in the last few years! Fortunately most of these problems have subsequently been rectified.

## 2. A Comparison of MPFUN with Other Multiprecision Systems

Several software packages are available for multiprecision computation. One that has been around for a while is the Brent MP multiprecision package, authored by R. P. Brent [11]. This package has the advantage of being freely available either from the author or from various other sources. It is very complete, including detailed numerical controls and many special functions. Recently Smith [30] described a similar package that features improved performance for certain transcendental functions.

Another package available at some sites is MACSYMA, which was originally developed at MIT but is now distributed by Symbolics, Inc. MACSYMA is actually a complete symbolic mathematics package, and its multiprecision arithmetic capability is only one part. A newer package of this sort is Mathematica [32], distributed by Wolfram Research, Inc. It features support of impressive full-color graphics for owners of advanced workstations.

There exist a number of other multiprecision systems in use that are specifically targeted for a particular computer system or for special applications. A package for large integer

computation, with a focus on the Cray-2, is described in [14]. With the multiprecision computation tools currently available, some may question the need for yet another. In this regard, the author has attempted to combine some of the more valuable features of existing packages with a high performance design. The resulting package of Fortran routines and functions is known as "MPFUN".

First of all, like Brent's MP package, the author's MPFUN package is freely available, whereas the commercial products typically have hefty price tags and annual maintenance fees. Secondly, MPFUN runs virtually without change on any scientific computer, whereas most of the others require significant customization from system to system. As a result, an application written to call the author's routines on a workstation or even on a personal computer can be effortlessly ported to a more powerful system, such as a supercomputer, for extended computations.

One key feature of the MPFUN package is that it was written with a vector supercomputer or RISC floating point computer in mind from the beginning. Virtually all inner loops are vectorizable and employ floating point operations, which have the highest performance on supercomputers. As a result, MPFUN exhibits excellent performance on these systems. None of the other widely available packages, to the author's knowledge, exhibits respectable performance on supercomputers such as Crays. Also, the package avoids constructs that inhibit multiple processor computation. As a result, it can easily be modified to employ multitasking software.

Another distinguishing feature of the MPFUN package is its usage of advanced algorithms. For many functions, both a "basic" and an "advanced" routine are provided. The advanced routines employ advanced algorithms and exhibit superior performance for extra-high precision (i.e. above about 1000 digit) calculations. For example, an advanced multiplication routine is available that employs a fast Fourier transform (FFT), and routines implementing quadratically convergent algorithms for exp, log, cos and sin are also provided.

Finally, usage of this package is greatly facilitated by the existence of an automatic translation program, which accepts as input a standard Fortran-77 program to which has been added certain special comments that declare the desired level of precision and declare certain variables in each program unit to be treated as multiprecision. This translator then parses the input code and generates a program that has all of the calls to the appropriate MPFUN routines. This translator program is described in a separate paper available from the author [1].

## 3. Overview of the Package

The MPFUN package consists of approximately 10,000 lines of Fortran code organized into 87 subprograms. These routines operate on three custom data types: multiprecision (MP) numbers, multiprecision complex (MPC) numbers and double precision plus exponent (DPE) numbers.

A MP number is represented by a single precision floating point array. The sign of the first word is the sign of the MP number. The magnitude of the first word is the

number of mantissa words. The second word of the MP array contains the exponent, which represents the power of the radix, which is either $2^{22} = 4194304$ for Cray systems or $2^{24} = 16777216$ for most other systems, including systems based on the IEEE 754 standard. Words beginning with the third word in the array contain the mantissa. Mantissa words are floating point whole numbers between 0 and one less than the radix. For MP numbers with zero exponent, the "decimal" point is assumed after the first mantissa word. For example, the MP number 3 is represented by the three-long array `(1., 0., 3.)`. A MP zero is represented by the two-long array `(0., 0.)`.

If sufficient memory is available, the maximum precision level for MP numbers is approximately 50 million digits. The limiting factor for this precision level is the accuracy of calculations in the FFT-based multiplication routine. Beyond a certain level, rounding the double precision results of the final FFT to nearest integer is no longer reliable (see section 5 below). The maximum dynamic range of MP numbers is about $10^{\pm 14,000,000}$.

A MPC number is represented as two consecutive MP numbers, which are the real and imaginary parts of the complex number. A DPE number is a pair `(A, N)`, where `A` is a double precision scalar and `N` is an integer. It represents `A * 2**N`. These DPE numbers are useful in multiple precision applications to represent numbers that do not require high precision but may have large exponent ranges. A DPE zero is denoted by the pair `(0.D0, 0)`.

This decision to base the package on floating point data derives from the fact that floating point performance is becoming the principal emphasis on almost all advanced scientific computers, from workstations to high-end supercomputers. This is particularly true on Cray systems, where the hardware instruction sets do not even include 64 bit integer multiplication or division instructions — such operations must be performed by first converting the arguments to floating and then by using the floating point functional units. Basing MPFUN on floating point operations has the additional benefit that it permits virtually universal portability in the resulting program code.

There is one additional reason that the implementation is based on floating point arithmetic, and that the package may appear to be optimized for systems based on vector or RISC processors. This is because except for extremely high levels of precision (i.e. tens of thousands or millions of digits), there is not a great deal of low-level parallelism in multiprecision calculations. Thus except for modest-length vectors at the base level, multiprecision applications need to be parallelized at a higher level. For example, if one is performing computations with a matrix of multiprecision numbers, it is likely that parallelism can be exploited at the level of rows or columns of the matrix. This suggests that the preferred architecture for the parallel processing of multiprecision applications is a MIMD array of vector or RISC processors. Thus this code was thus written with such a computer model in mind.

MPFUN routines are available to perform the four basic arithmetic operations between MP numbers, to compare MP numbers, to produce the integer and fractional parts, to produce a random MP number and to perform binary to decimal and decimal to binary conversion. Other routines perform operations between DPE numbers or between MP

and DPE numbers, which saves time compared with performing these operations with the full MP routines. Some higher level routines sort MP numbers; perform complex arithmetic; compute square roots, cube roots, $n$-th powers, $n$-th roots, and $\pi$; evaluate the functions exp, log, cos, sin, cosh, sinh, inverse cos and sin; find the real or complex roots of polynomials; and find integer relations in real vectors. For many of these functions, both basic and advanced versions are available. The advanced routines employ advanced algorithms suitable for extra high precision computation.

Computations on large integers can be efficiently performed using this package by setting the working precision level two or three words higher than the largest integer that will be encountered (including products). These extra words of precision permit accurate integer division to be performed, using a multiprecision floating point division routine followed by a call to the routine that produces the integer and fractional parts of an MP number. There is no wasted computation when the actual size of an integer argument is much less than working precision level, since the MPFUN routines only perform arithmetic on the actual sizes of input data.

## 4. Portability and Testing

As mentioned earlier, one distinguishing feature of the MPFUN package is its portability. The standard version of MPFUN should run correctly, without alteration, on any computer with a Fortran-77 compiler that satisfies the following requirements:

1. The floating point arithmetic has a binary (or hexadecimal) radix.
2. The truncation of a double precision value not exceeding $2^{50}$ is correct.
3. The decimal-to-binary conversion of a double precision constant, which is either a power of two not exceeding $2^{30}$ or the fraction $1/2$, is exact.
4. The addition, subtraction, multiplication, truncated division, and exponentiation of integer variables or constants, where the arguments and results do not exceed $2^{30}$ in absolute value, produce exact results.
5. Integers up to $2^{24}$ are represented exactly as single precision floating point numbers.
6. The addition, subtraction and multiplication of double precision variables or constants, where the arguments and results are whole numbers not exceeding $2^{53}$ in absolute value, produce exact results.
7. The results of `**`, `EXP`, `LOG`, `LOG10`, `COS`, `SIN`, and `ATAN2` with double precision arguments are correct to within one part in $2^{-45}$, provided the arguments are of reasonable size and not close to singularities.

The author is not aware of any serious scientific computer system in use today that fails to meet these requirements. Any system based on the IEEE 754 floating point standard, with a 24 bit mantissa in single precision and a 52 bit mantissa in double precision (24 and

53 bits, including the hidden bit), easily meets these requirements. All DEC VAX systems meet these requirements. All IBM mainframes and workstations meet these requirements. Cray systems meet these requirements with double precision disabled (i.e. by using only single precision), provided that the number $2^{50}$ in item 2 is reduced to $2^{45}$ and the number $2^{53}$ in item 6 is reduced to $2^{48}$.

To insure that these routines are working correctly, a test suite is available. It exercises virtually all of the routines in the package and checks the results. This test program is useful in its own right as a computer system integrity test. As mentioned in the introduction, versions of this program have on numerous occasions disclosed hardware and software bugs in scientific computer systems.

## 5. The Four Basic Arithmetic Operations

Multiprecision addition and subtraction are not computationally expensive compared to multiplication, division, and square root extraction. Thus simple algorithms suffice to perform addition and subtraction. The only part of these operations that is not immediately conducive to vector processing is releasing the carries for the final result. This is because the normal "schoolboy" approach of beginning at the last cell and working forward is a recursive (i.e. non-vectorizable) operation. On a vector computer this is better done by starting at the beginning and releasing the carry only one cell back for each cell processed. Unfortunately, it cannot be guaranteed that one application of this process will release all carries. Thus it is necessary to repeat this operation until all carries have been released, usually only one or two additional times. In the rare cases where three applications of this operation are not successful in releasing all carries, the author's program resorts to the scalar "schoolboy" method. On scalar or RISC computers, only the "schoolboy" scheme is used.

A key component of a high-performance multiprecision arithmetic system is the multiply operation, since in real applications typically a significant fraction of the total time is spent here. The author's basic multiply routine, which is used for modest levels of precision, employs a conventional "schoolboy" scheme, although care has been taken to insure that the operations are vectorizable. A significant saving is achieved by not releasing the carries after each vector multiply operation, but instead waiting until 32 such vector multiply operations have been completed (16 on Crays). An additional saving is achieved by computing only the first half of the multiplication "pyramid".

The schoolboy scheme for multiprecision multiplication has computational complexity proportional to $n^2$, where $n$ is the number of words or digits. For higher precision levels, other more sophisticated techniques have a significant advantage, with complexity as low as $n \log n \log \log n$. The history of the development of advanced multiprecision multiplication algorithms will not be reviewed here. The interested reader is referred to Knuth [24]. Because of the difficulty of implementing these advanced schemes and the widespread misconception that these algorithms are not suitable for "practical" application, they are rarely employed. For example, none of the widely used multiprecision packages employs an "advanced" multiplication algorithm, to the author's knowledge. One instance where an

advanced multiplication technique was employed is [17]. Another is Slowinski's searches for large Mersenne prime numbers [29].

The author has implemented a number of these schemes, including variations of the Karatsuba-Winograd algorithm and schemes based on the discrete Fourier transform (DFT) in various number fields [24]. Based on performance studies of these implementations, the author has found that a scheme based on complex DFTs appears to be the most effective and efficient for modern scientific computer systems. The complex DFT and the inverse complex DFT of the sequence $x = (x_0, x_1, x_2, \cdots, x_{N-1})$ are given by

$$\begin{aligned} F_k(x) &= \sum_{j=0}^{N-1} x_j e^{-2\pi ijk/N} \\ F_k^{-1}(x) &= \frac{1}{N} \sum_{j=0}^{N-1} x_j e^{2\pi ijk/N} \end{aligned}$$

Let $C(x, y)$ denote the circular convolution of the sequences $x$ and $y$:

$$C_k(x, y) = \sum_{j=0}^{N-1} x_j y_{k-j}$$

where the subscript $k - j$ is to be interpreted as $k - j + N$ if $k - j$ is negative. Then the convolution theorem for discrete sequences states that

$$F[C(x, y)] = F(x)F(y)$$

or expressed another way

$$C(x, y) = F^{-1}[F(x)F(y)]$$

This result is applicable to multiprecision multiplication in the following way. Let $x$ and $y$ be $n$-long representations of two multiprecision numbers (without the sign or exponent words). Extend $x$ and $y$ to length $2n$ by appending $n$ zeroes at the end of each. Then the multiprecision product $z$ of $x$ and $y$, except for releasing the carries, can be written as follows:

$$\begin{aligned} z_0 &= x_0 y_0 \\ z_1 &= x_0 y_1 + x_1 y_0 \\ z_2 &= x_0 y_2 + x_1 y_1 + x_2 y_0 \\ &\cdot \\ &\cdot \\ z_{n-1} &= x_0 y_{n-1} + x_1 y_{n-2} + \cdots + x_{n-1} y_0 \\ &\cdot \\ &\cdot \\ z_{2n-3} &= x_{n-1} y_{n-2} + x_{n-2} y_{n-1} \\ z_{2n-2} &= x_{n-1} y_{n-1} \\ z_{2n-1} &= 0 \end{aligned}$$

It can now be seen that this multiplication pyramid is precisely the convolution of the two sequences $x$ and $y$, where $N = 2n$. In other words, the multiplication pyramid can be obtained by performing two forward DFTs, one vector complex multiplication, and one inverse DFT, each of length $N = 2n$. Once the inverse DFT results have been adjusted to the nearest integer to compensate for any numerical error, the final multiprecision product may be obtained by merely releasing the carries as described above.

The computational savings arises from the fact that complex DFTs may of course be economically computed using some variation of the fast Fourier transform (FFT) algorithm. The particular FFT algorithm utilized for the MPFUN advanced multiplication routine is described in [3]. It was first proposed by Swarztrauber and is sometimes called the "Stockham-Transpose-Stockham" FFT. This algorithm features reasonably high performance on most computers, including vector and cache memory systems, and it can easily be modified for multiple processor computation if desired. For the implementation in this package, different techniques are employed for the matrix transposition step depending on the computer system and the size of the array. Since in this application the two inputs and the final output of the convolution are purely real, an algorithm is employed that converts the problem of computing the FFT on real data to that of computing the FFT on complex data of half the size. This results in a computational savings of approximately 50 percent.

One important detail has been omitted from the above discussion. Since the radix of MP numbers is either $2^{22}$ or $2^{24}$, the products $x_j y_{k-j}$ may be as large as $2^{48} - 1$, and the sum of a large number of these products cannot be represented exactly as a 64 bit floating point value, no matter how it is computed. In particular, the nearest integer operation at the completion of the final inverse FFT cannot reliably recover the exact multiplication pyramid result. For this reason, each input data word is split into two words upon entry to the FFT-based multiply routine. This permits computations of up to approximately 50 million digits to be performed correctly.

Included in the advanced multiply routine (although normally commented out) is some code that determines the maximum FFT roundoff error, i.e. the maximum difference between the final FFT results and the nearest integer values, and tests if it is greater than a certain reliable level. This code can be used as a system integrity test, since for modest levels of precision, the maximum FFT roundoff error should be a rather small number, and an excessive value indicates that a hardware or compiler error has occurred.

The division of two MP numbers of modest precision is performed using a fairly straightforward scheme. Trial quotients are computed in double precision. This guarantees that the trial quotient is virtually always correct. In those rare cases where one or more words of the quotient are incorrect, the result is automatically fixed in a cleanup routine at no extra computational cost.

In the advanced division routine, the quotient of $a$ and $b$ is computed as follows. First the following Newton-Raphson iteration is employed, which converges to $1/b$:

$$x_{k+1} = x_k + (1 - bx_k)x_k$$

where the muliplication $(\cdot) * x_k$ is performed with only half of the normal level of precision.

The final iteration is performed as follows [23]:

$$a/b \approx (ax_n) + [a - (ax_n)b]x_n$$

where the multiplications $ax_n$ and $[\cdot]x_n$ are performed with only half of the final level of precision. Note that each iteration of this algorithm involves only an addition and a subtraction, plus two multiplications, which can be performed using the FFT-based technique mentioned above.

Algorithms based on Newton iterations have the desirable property that they are inherently self-correcting. Thus these Newton iterations can be performed with a precision level that doubles with each iteration. One difficulty with this procedure is that errors can accumulate in the trailing mantissa words. This error can be economically controlled by repeating the next-to-last iteration. This increases the run time by only about 25 percent, and yet the result is accurate to all except possibly the last two words.

It can easily be seen that the total cost of computing a reciprocal by this means is about twice the cost of the final iteration. The total cost of a multiprecision division is only about four times the cost of a multiprecision multiplication operation of equivalent size.

## 6. Other Algebraic Operations

Complex multiprecision multiplication is performed using the identity

$$(a_1 + a_2 i)(b_1 + b_2 i) \quad = \quad [a_1 b_1 - a_2 b_2] + [(a_1 + a_2)(b_1 + b_2) - a_1 b_1 - a_2 b_2]i$$

Note that this formula can be implemented using only three multiprecision multiplications, whereas the straightforward formula requires four. Complex division is performed using the identity

$$\frac{a_1 + a_2 i}{b_1 + b_2 i} \quad = \quad \frac{(a_1 + a_2 i)(b_1 - b_2 i)}{b_1^2 + b_2^2}$$

where the complex product in the numerator is evaluated as above. Since division is significantly more expensive than multiplication, the two real divisions ordinarily required in this formula are replaced with a reciprocal computation of $b_1^2 + b_2^2$ followed by two multiplications. The advanced routines for complex multiplication and division utilize these same formulas, but they call the advanced routines for real multiplication and division.

The general scheme described in the last section to perform division by Newton iterations is also employed to evaluate a number of other algebraic operations. For example, square roots are computed by employing the following Newton iteration, which converges to $1/\sqrt{a}$:

$$x_{k+1} \quad = \quad x_k + (1 - a x_k^2) x_k / 2$$

where the muliplication $(\cdot)x_k$ is performed with only half of the normal level of precision. The final iteration is performed as follows [23]:

$$\sqrt{a} \approx (ax_n) + [a - (ax_n)^2]x_n/2$$

where the multiplications $ax_n$ and $[\cdot]x_n$ are performed with only half of the final level of precision.

As with division, these iterations are performed with a precision level that approximately doubles with each iteration. The basic square root routine computes each iteration to one word more than a power of two. As a result, errors do not accumulate very much, and it suffices to repeat the third-from-the-last iteration to insure full accuracy in the final result. The added cost of repeating this iteration is negligible.

The advanced square root routine cannot compute each iteration to one greater than a power of two words, since the levels of precision are restricted to exact powers of two by the FFT-based multiply procedure. Thus the advanced routine repeats the next-to-last iteration. As in the advanced divide routine, repeating the next-to-last iteration adds about 25 percent to the run time.

The complex square root of $z = x + iy$ can be computed by applying the formulas

$$\begin{aligned} s &= \sqrt{\frac{|x| + \sqrt{x^2 + y^2}}{2}} \\ \sqrt{z} &= s + i\frac{y}{2s} \qquad \text{if } x \geq 0 \\ &= \frac{|y|}{2s} \pm is \qquad \text{if } x < 0 \end{aligned}$$

where the $\pm$ sign is taken to be the same as the sign of $y$.

Cube roots are computed by the following Newton iteration, which converges to $a^{-2/3}$:

$$x_{k+1} = x_k + \frac{x_k}{3}(1 - a^2 x_k^3)$$

The final iteration for the cube root, corresponding to the final iteration for the square root, is the following [23]:

$$a^{1/3} \approx (ax_n) + [a - (ax_n)^3]x_n/3$$

Included in the MPFUN package are basic and advanced routines to compute the $n$-th power of multiprecision real and complex numbers. These operations are performed using the binary rule for exponentiation [24]. When $n$ is negative, the reciprocal is taken of the final result.

Along with the $n$-th power routines are two $n$-th root routines. When the argument $a$ is very close to one and $n$ is large, the $n$-th root is computed using a binomial expansion. Otherwise, it is computed using the following Newton iteration, which converges to $a^{-1/n}$:

$$x_{k+1} = x_k + \frac{x_k}{n}(1 - ax_k^n)$$

The reciprocal of the final approximation to $a^{-1/n}$ is the $n$-th root. These iterations are performed with a dynamic precision level as before.

The MPFUN package includes four routines for computing roots of polynomials. There is a basic and an advanced routine for computing real roots of real polynomials and complex

roots of complex polynomials. Let $P(x)$ be a polynomial and let $P'(x)$ be the derivative of $P(x)$. Let $x_0$ be a starting value that is close to the desired root. These routines then employ the following Newton iteration, which converges directly to the root:

$$x_{k+1} = x_k - P(x_k)/P'(x_k)$$

These iterations are computed with a dynamic precision level scheme similar to the routines described above.

One requirement for this method to work is that the desired root is not a repeated root. If one wishes to apply these routines to find a repeated root, it is first necessary to reduce the polynomial to one that has only simple roots. This can be done by performing the Euclidean algorithm in the ring of polynomials to determine the greatest common divisor $Q(x)$ of $P(x)$ and $P'(x)$. Then $R(x) = P(x)/Q(x)$ is a polynomial that has only simple roots.

In the introduction, the usage of integer relation finding algorithms was mentioned in exploring the transcendence of certain mathematical constants. The author has tested two recently discovered algorithms for this purpose, the "small integer relation algorithm" in [20], which will be termed the HJLS routine from the initials of the authors, and the "partial sum of squares" (PSOS) algorithm of H. R. P. Ferguson [5]. While each has its merits, the author has found that the HJLS routine is generally faster. Thus it has been implemented in MPFUN. For those readers interested in the PSOS algorithm, a routine implementing it is also available from the author.

Since both the HJLS and PSOS algorithms are quite complicated, neither will be presented here. Interested readers are referred to the respective papers.

## 7. Computing $\pi$

The computation of $\pi$ to high precision has a long and colorful history. Interested readers are referred to [6] for discussion of the classical history of computing $\pi$. Recently a number of advanced algorithms have been discovered for the computation of $\pi$ that feature very high rates of convergence [8, 9]. The first of these was discovered independently by Salamin [28] and Brent [10] and is referred to as either the Salamin-Brent algorithm or the Gauss-Legendre algorithm, since the mathematical basis of this algorithm has its roots in the nineteenth century. This algorithm exhibits quadratic convergence, i.e. each iteration approximately *doubles* the number of correct digits. Subsequently the Borweins have discovered a class of algorithms that exhibit $m$-th order convergence for any $m$ [8, 9].

The author has tested a number of these algorithms. Surprisingly, although the Borwein algorithms exhibit higher rates of convergence, the overall run time is generally comparable to that of the Salamin-Brent algorithm. Since the Salamin-Brent algorithm is simpler, it was chosen for implementation in MPFUN. It may be stated as follows. Set $a_0 = 1$, $b_0 = 1/\sqrt{2}$, and $d_0 = \sqrt{2} - 1/2$. Then iterate the following operations beginning with $k = 1$:

$$\begin{aligned} a_k &= (a_{k-1} + b_{k-1})/2 \\ b_k &= \sqrt{a_{k-1} b_{k-1}} \\ d_k &= d_{k-1} - 2^k (a_k - b_k)^2 \end{aligned}$$

Then $p_k = (a_k + b_k)^2/d_k$ converges quadratically to $\pi$. Unfortunately this algorithm is not self-correcting like algorithms based on Newton iterations. Thus all iterations must be done with at least the precision level desired for the final result.

## 8. Transcendental Functions

The basic routine for exp employs a modification of the Taylor's series for $e^t$:

$$e^t = (1 + r + \frac{r^2}{2!} + \frac{r^3}{3!} + \frac{r^4}{4!} \cdots)^{256}\, 2^n$$

where $r = t'/256$, $t' = t - n \log 2$ and where $n$ is chosen to minimize the absolute value of $t'$. The exponentiation in this formula is performed by repeated squaring. Reducing $t$ modulo $\log 2$ and dividing by 256 insures that $-0.001 < r \leq 0.001$, which significantly accelerates convergence in the above series.

The basic routine for log employs the following Newton iteration, which converges to $\log t$:

$$x_{k+1} = x_k + \frac{t - \exp x_k}{\exp x_k}$$

The run time of the basic log routine is only about 2.5 times that of the exp routine.

The advanced routine for log employs a quadratically convergent algorithm due to Salamin, as described in [12]. Inputs $t$ that are extremely close to 1 are handled using a Taylor series. Otherwise let $n$ be the number of bits of precision required in the result. If $t$ is exactly two, select $m > n/2$. Then the following formula gives $\log 2$ to the required precision:

$$\log 2 = \frac{\pi}{2mA(1, 4/2^m)}$$

Here $A(a, b)$ is the limit of the arithmetic-geometric mean: let $a_0 = a$ and $b_0 = b$. Then iterate

$$a_{k+1} = \frac{a_k + b_k}{2}$$
$$b_{k+1} = \sqrt{a_k b_k}$$

For other $t$ select $m$ such that $s = t2^m > 2^{n/2}$. Then the following formula gives $\log t$ to the required precision:

$$\log t = \frac{\pi}{2A(1, 4/s)} - m \log 2$$

The advanced routine for exp employs the following Newton iteration, which converges to $e^t$:

$$x_{k+1} = x_k(t + 1 - \log x_k)$$

It might be mentioned that quadratically convergent algorithms for exp and log were first presented by Brent in [10], and others were presented in by the Borweins in [7]. Based on the author's comparisons, however, the Salamin algorithm is significantly faster than either the Brent or the Borwein algorithm. For this reason the Salamin algorithm was selected for inclusion in this package.

The basic routine for sin and cos utilizes the Taylor's series for $\sin s$:

$$\sin s = s - \frac{s^3}{3!} + \frac{s^5}{5!} - \frac{s^7}{7!} \cdots$$

where $s = t - a\pi/2 - b\pi/16$ and the integers $a$ and $b$ are chosen to minimize the absolute value of $s$. We can then compute

$$\begin{aligned} \sin t &= \sin(s + a\pi/2 + b\pi/16) \\ \cos t &= \cos(s + a\pi/2 + b\pi/16) \end{aligned}$$

by applying elementary trigonometric identities for sums. The sin and cos of $b\pi/16$ are of the form $0.5\sqrt{2 \pm \sqrt{2 \pm \sqrt{2}}}$. Reducing $t$ in this manner insures that $-\pi/32 < s \leq \pi/32$, which significantly accelerates convergence in the above series.

The advanced routines for cos and sin, and for inverse cos and inverse sin, employ complex arithmetic versions of the advanced algorithms described above for exp and log (recall that $e^{ix} = \cos x + i \sin x$).

## 9. Accuracy of Results

Most of the basic routines, and the advanced multiplication routine, are designed to produce results correct to the last word of working precision. In the case of the transcendental functions, the last word should be accurate provided the input values $\pi$ and $\log 2$ have been computed to at least one word of precision greater than the working precision. Even so, an entire word can easily be lost in many calculations due to normalization, such as when the reciprocal of a number slightly less than one is computed. Thus computations should always be performed with at least one extra word of precision than required for the final results.

For the advanced routines other than multiplication, the last two to four words are not reliable, as explained in the previous sections. For example, the ratio of two integers computed using the advanced division routine, the first of which is an exact multiple of the second, may not give the correct integer result. This situation should be familiar to users of Cray computers, which also uses Newton iterations to calculate reciprocals. Most anomalies of this sort can be remedied by adding a "fuzz" to results.

The accuracy of results from the MPFUN routines can also be controlled by setting a rounding mode parameter. Depending on the value of this parameter, results are either truncated at the last mantissa word of working precision, or else the last word is rounded up depending on contents of the first omitted word.

Whichever routines and rounding mode are used, it is not easy to determine ahead of time what level of precision is necessary to produce results accurate to a desired tolerance.

Also, despite safeguards and testing, a package of this sort cannot be warranted to be free from bugs. Additionally, compiler and hardware errors do occur, and it is not certain that they will be detected by the package. Thus the following procedure is recommended to increase one's confidence in computed results:

1. Start with a working double precision program, and then check that the ported multiprecision code duplicates intermediate and final results to a reasonable accuracy.

2. Where possible, use the ported multiprecision code to compute special values that can be compared with other published high precision values.

3. Repeat the calculation with the rounding mode parameter changed, in order to test the sensitivity of the calculation to numerical error. Alternatively, repeat the calculation with the precision level set to a higher level.

4. Repeat the calculation on another computer system, in order to certify that no hardware or compiler error has occurred.

## 10. Using the MPFUN Package

Detailed instructions and other information regarding the usage of the MPFUN package may be found in the appendix to this paper. The appendix also contains a complete list of the MPFUN routines, together with calling sequences and other information.

However, as mentioned above, the author has recently developed a translator program that converts ordinary Fortran-77 programs to code that calls the MPFUN routines. By using the translator, the user does not need to know details of the operation of or calling sequences for the MPFUN routines. For many applications, this converted program is entirely satisfactory, and the performance is not significantly different from "hand-coded" efforts (i.e. manually written programs that call the MPFUN routines according to the instructions in the appendix).

This translation program allows one to extend the Fortran-77 language with the data types `MULTIP INTEGER`, `MULTIP REAL` and `MULTIP COMPLEX`. These data types can be used for integer, floating point or complex numbers of an arbitrarily high but pre-specified level of precision. Variables in the input program may be declared to have one of these multiprecision types in the output program by placing directives (special comments) in the input file. In this way, the input file remains an ANSI Fortran-77 compatible program and can be run at any time using ordinary arithmetic on any Fortran system for comparison with the multiprecision equivalent.

This translator supports a large number of Fortran-77 constructs involving multiprecision variables, including all the standard arithmetic operators, mixed mode expressions, automatic type conversions, comparisons, logical IF statements (including IF-THEN-ELSE constructs), function calls, read/write statements and most of the Fortran intrinsics (i.e. ABS, MOD, COS, EXP, etc.). Storage is automatically allocated for multiprecision variables, including temporaries, and the required initialization for the MPFUN package is automatically performed.

Complete details of this translator program may be found in [1].

## 11. Performance

One application of a package such as MPFUN is to remedy difficult numerical problems that sometimes arise in conventional scientific programs. In these cases, a precision level perhaps double or triple that of ordinary machine precision is all that is required. One might wonder how much longer such a program is likely to run using calls to MPFUN.

Using the translator program mentioned in the previous section, the author has converted to multiprecision a program that, among other things, computes fast Fourier transforms (FFTs). The precision level was 40 digits. On a Silicon Graphics RISC workstation, the multiprecision code ran 135 times slower than the same program with ordinary double precision (64-bit) arithmetic. Thus while such runs are indeed possible, they are not to be considered lightly. The main reason this ratio is so high is that much of the computational effort at this precision level is merely the overhead of making numerous subroutine calls. The author has not attempted to convert this program to multiprecision using another package, but timing ratios even higher than 135 have been reported by other researchers who have attempted such conversions [21].

Another application of a package such as MPFUN is for problems where the precision level required is much higher than that which can be obtained through ordinary machine arithmetic, perhaps hundreds or even thousands of digits. Such applications arise most often in numerical studies of mathematical questions. In such cases the dominant computational cost is not merely subroutine calling overhead, and algorithmic factors become more significant.

One way to compare the performance of the author's package with other multiprecision packages is to compare timings for the computation of a mathematical constant such as $\pi$ to high precision, since this is easily programmed and yet exercises all of the basic arithmetic routines. Tables 1 and 2 give some performance results on this problem for the MPFUN package, the Mathematica 2.0 package and Brent's package. The algorithm used by Mathematica is not mentioned in the Mathematica reference book [32], but it is almost certainly either the Salamin-Brent algorithm or one of the Borwein algorithms. The algorithm used by Brent's package for computing $\pi$ is the Salamin-Brent algorithm, basically the same as described in section 7.

The timings in Table 1 are for a Silicon Graphics model 4D-380 RISC workstation (one processor), which has a theoretical peak performance of 16 Mflop/s and a Linpack performance of 4.9 Mflop/s (double precision figures). The timings in Table 2 are for a Cray Y-MP supercomputer (one processor), which has a theoretical peak performance of 330 Mflop/s and a Linpack performance of 90 Mflop/s. When these runs were made, the SGI system was running IRIX 3.3 system software, and the Cray was running UNICOS 6.0. A blank in the table indicates that the run would have taken an unreasonable amount of time and was not performed. The numbers of digits in the second column of the two tables correspond to $7.225 \cdot 2^m$ and $6.623 \cdot 2^m$, respectively, which are the sizes convenient for the FFT-based multiplication scheme described above.

On the SGI system, MPFUN is slightly faster than Mathematica and roughly four times as fast as Brent's package for modest precision levels. Once the level of precision rises above 1000 digits, MPFUN has a considerable advantage, due mainly to its FFT-based multiply routine. At 29,590 digit precision, the highest level at which all three could be compared, the MPFUN package is five times faster than Mathematica 2.0 and 40 times faster than Brent's package.

On the Cray Y-MP, the MPFUN package is three times faster than Brent's package at the lowest precision level and 415 times faster at 54,250 digits precision, the highest level at which both could be compared. Two reasons this ratio is so high on the Cray Y-MP are (1) the MPFUN routines employ floating point arithmetic, whereas Brent's package uses integer operations, and (2) a high percentage of operations in the MPFUN routines are performed in vector mode, whereas much of the computation in Brent's package is done in scalar mode. At the highest precision level listed, the Y-MP is running the author's code at 195 Mflop/s, or 59% of the one processor peak rate.

Since Brent's package and Mathematica are perhaps the most widely used packages of this sort, other authors typically compare their performance figures with one of these. For example, Smith [30] compares his package with Brent's. Since Smith's timings for fundamental add and multiply operations are roughly comparable to Brent's, it would be expected that MPFUN would exhibit similar performance ratios with Smith's package.

**Acknowledgments**

The author wishes to acknowledge helpful comments and suggestions by W. Kahan of the University of California, Berkeley, by K. Briggs of the University of Melbourne, Australia, and by R. Brent of the Australian National University.

| m | Digits | MPFUN | Mathematica 2.0 | Brent |
|---|---|---|---|---|
| 3 | 60 | 0.008 | 0.020 | 0.033 |
| 4 | 115 | 0.018 | 0.030 | 0.062 |
| 5 | 230 | 0.038 | 0.050 | 0.174 |
| 6 | 460 | 0.096 | 0.130 | 0.543 |
| 7 | 925 | 0.297 | 0.360 | 2.150 |
| 8 | 1,850 | 0.710 | 1.260 | 8.610 |
| 9 | 3,700 | 2.410 | 4.490 | 34.550 |
| 10 | 7,400 | 6.260 | 18.610 | 145.300 |
| 11 | 14,795 | 18.920 | 78.890 | 619.100 |
| 12 | 29,590 | 64.980 | 302.700 | 2661.000 |
| 13 | 59,185 | 246.880 | | |
| 14 | 118,370 | 804.200 | | |

Table 1: SGI Workstation Performance Results (seconds)

| m | Digits | MPFUN | Brent |
|---|---|---|---|
| 4 | 105 | 0.006 | 0.019 |
| 5 | 210 | 0.009 | 0.037 |
| 6 | 425 | 0.016 | 0.088 |
| 7 | 845 | 0.032 | 0.229 |
| 8 | 1,695 | 0.074 | 0.815 |
| 9 | 3,390 | 0.120 | 3.176 |
| 10 | 6,780 | 0.270 | 13.040 |
| 11 | 13,565 | 0.535 | 54.620 |
| 12 | 27,125 | 1.115 | 230.800 |
| 13 | 54,250 | 2.345 | 975.600 |
| 14 | 108,505 | 5.070 | |
| 15 | 217,010 | 11.070 | |
| 16 | 434,020 | 24.300 | |
| 17 | 868,045 | 53.560 | |
| 18 | 1,736,090 | 114.700 | |

Table 2: Cray Y-MP Performance Results (seconds)

# References


[1] Bailey, D. H., "Automatic Translation of Fortran Programs to Multiprecision", RNR Technical Report RNR-91-025, NAS Applied Research Branch, NASA Ames Research Center.

[2] Bailey, D. H., "The Computation of $\pi$ to 29,360,000 Decimal Digits Using Borweins' Quartically Convergent Algorithm", *Mathematics of Computation*, vol. 50 (Jan. 1988), p. 283 – 296.

[3] Bailey, D. H., "A High Performance FFT Algorithm for Vector Supercomputers", *International Journal of Supercomputer Applications*, vol. 2 (Spring 1988), p. 82 – 87.

[4] Bailey, D. H., "Numerical Results on the Transcendence of Constants Involving $\pi$, $e$, and Euler's Constant", *Mathematics of Computation*, vol. 50 (Jan. 1988), p. 275 – 281.

[5] Bailey, D. H., and Ferguson, H. R. P., "Numerical Results on Relations Between Numerical Constants Using a New Algorithm", *Mathematics of Computation*, vol. 53 (October 1989), p. 649 – 656.

[6] Beckmann, P., *A History of Pi*, Golem Press, Boulder CO, 1977.

[7] Borwein, J. M., and Borwein, P. B., "The Arithmetic-Geometric Mean and Fast Computation of Elementary Functions", *SIAM Review* vol. 26 (1984), p. 351 – 365.

[8] Borwein, J. M., and Borwein, P. B., *Pi and the AGM*, John Wiley, New York, 1987.

[9] Borwein, J. M., Borwein, P. B., and Bailey, D. H., "Ramanujan, Modular Equations, and Approximations to Pi", *The American Mathematical Monthly*, vol. 96 (1989), p. 201 – 219.

[10] Brent, R. P., "Fast Multiple-Precision Evaluation of Elementary Functions", *Journal of the ACM*, vol. 23 (1976), p. 242 – 251.

[11] Brent, R. P., "A Fortran Multiple Precision Arithmetic Package", *ACM Transactions on Mathematical Software*, vol. 4 (1978), p. 57 – 70.

[12] Brent, R. P., "Multiple-Precision Zero-Finding Methods and the Complexity of Elementary Function Evaluation", *Analytic Computational Complexity*, Academic Press, New York, 1976, p. 151 – 176.

[13] Briggs, Keith, "A Precise Calculation of the Feigenbaum Constants", *Mathematics of Computation*, vol. 57 (1991), p. 435 – 439.

[14] Buell, D., and Ward, R., "A Multiprecise Integer Arithmetic Package", *Journal of Supercomputing*, vol. 3 (1989), p. 89 – 107.

[15] Chudnovsky, D. V. and Chudnovsky, G. V., "Computation and Arithmetic Nature of Classical Constants", *IBM Research Report*, IBM T. J. Watson Research Center, RC14950 (#66818), 1989.

[16] Chudnovsky, D. V. and Chudnovsky, G. V., personal communication, 1991.

[17] Comba, P. G., "Exponentiation Cryptosystems on the IBM PC", *IBM Systems Journal*, vol. 29 (1990), p. 526 – 538.

[18] Feigenbaum, M. J., "Quantitative Universality for a Class of Nonlinear Transformations", *Journal of Statistical Physics*, vol. 19 (1978), p. 25 – 52.

[19] Ferguson, H. R. P., and Forcade, R. W., "Generalization of the Euclidean Algorithm for Real Numbers to All Dimensions Higher Than Two", *Bulletin of the American Mathematical Society*, vol. 1 (1979), p. 912 – 914.

[20] Hastad, J., Just, B., Lagarias, J. C., and Schnorr, C. P., "Polynomial Time Algorithms for Finding Integer Relations Among Real Numbers", *SIAM Journal on Computing*, vol. 18 (1988), p. 859 – 881.

[21] Kahan, W., personal communication, 1991.

[22] Kanada, Y., personal communication, 1989.

[23] Karp, A., and Markstein, P., "High Precision Division and Square Root", HP Labs Report, Hewlett-Packard Laboratories, 1530 Page Mill Road, Palo Alto, CA 94304, 1993.

[24] Knuth, D. E., *The Art of Computer Programming*, Addison Wesley, Menlo Park, 1981.

[25] Lenstra, A. K., Lenstra, H. W., Manasse, M. S., Pollard. J. M., "The Number Field Sieve", *1990 ACM Symposium on the Theory of Computing*, p. 564 – 572.

[26] Odlyzko, A. M. and te Riele, H. J. J., "Disproof of the Mertens Conjecture", *J. Reine Angew. Mathematik*, vol. 357 (1985), p. 138 – 160.

[27] Rivest, R. L., Shamir, A., and Adleman, L., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Communications of the ACM*, vol. 21 (1978), p. 120 – 126.

[28] Salamin, E., "Computation of $\pi$ Using Arithmetic-Geometric Mean", *Mathematics of Computation*, vol. 30 (1976), p. 565 – 570.

[29] Slowinski, D., personal communication, 1991.

[30] Smith, David M., "A FORTRAN Package for Floating-Point Multiple-Precision Arithmetic", *ACM Transactions on Mathematical Software*, vol. 17, no. 2 (June 1991), p. 273 – 283.

[31] Varga, R. S., *Scientific Computation on Mathematical Problems and Conjectures*, SIAM, Philadelphia, 1990.

[32] Wolfram, Stephen. *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, New York, 1988.

**Appendix: Usage Instructions**

The MPFUN routines are listed, together with a brief functional description in Tables 3 and 4. Note that no routines are provided for absolute value or negation, since these operations may be performed by merely taking absolute value of or negating the first word of the single precision vector representing an MP number. Before calling any of these routines, some integer parameters in common block `MPCOM1` should be set. In order, they are `NW, IDB, LDB, IER, MCR, IRD, ICS, IHS`, and `IMS`. They are defined as follows:

1. `NW` is the maximum number of mantissa words. This should be set by the user in the main calling program to `1+ND/7.225`, where `ND` is the desired maximum precision level in digits (`1+ND/6.623` on Crays). Some routines alter this parameter but restore it prior to exiting. Default: 16 (i.e. about 110 digits).

2. `IDB` is a debug flag and ordinarily should be set to zero. Setting `IDB` to an integer between 4 and 10 produces debug printouts in varying degrees of detail from the subroutines of this package. Values of `IDB` between 1 and 3 are available for use as debug flags in the user's calling program if desired. Default: 0.

3. `LDB` is the logical unit number for output of debug and error messages. Default: 6.

4. `IER` is an error flag and should initially be set to zero. It is set to nonzero values by the routines when an error condition is detected.

5. `MCR` is the "crossover" point for the advanced routines — if an advanced routine is called with a precision level `NW` that is `2**MCR` or less, the advanced routine merely calls the basic MP routine. Default: 8.

6. `IRD` controls the rounding mode: when `IRD = 0`, the last word is truncated, i.e. the same compared with higher precision; when `IRD = 1`, the last mantissa word is rounded up if the first omitted word is half the radix or more; when `IRD = 2`, the last mantissa word is rounded up if the first omitted word is nonzero. Default: 1.

7. `ICS` is the current single precision scratch space stack pointer and should initially be set to one.

8. `IHS` is the current "high water mark" of `ICS` and should be initially set to one.

9. `IMS` is the maximum single precision scratch space available. Default: 1024.

Common block `MPCOM0` contains some parameters related to the radix. These include `BDX`, the radix itself, and `NBT`, the number of bits in the radix. These parameters are set in the block data subprogram and should not be changed except as required for a different computer systems.

Common block `MPCOM2` contains an integer array `KER` of length 72. This array controls the action taken when one of the MP routines detects an error condition, such as an

attempted divide by zero. If the entry corresponding to a particular error number is zero, the error is ignored and execution resumes, although the routine that detects the condition is often exited. If the entry is one, then a message is output on unit `LDB`, `IER` is set to the error number (an integer between 1 and 72), and execution resumes, although subsequent calls to MPFUN routines are immediately exited (i.e. control is quickly returned to the user program). If the entry is two, a message is output and execution is immediately terminated. All entries of the `KER` array are initialized to two in the block data subprogram below. The user may change some of these entries by including the common `MPCOM2` in the user's program.

An MP argument may not be used as both as an input and an output variable in a call to one of the MP routines. Output arrays for holding MP results generally require `NW+4` single precision cells. An exception is `MPDMC`, where the output MP variable only requires eight cells. Output arrays for holding MPC results require twice the offset between real and imaginary parts. In most applications this offset is simply `NW+4` cells, so that the memory requirement for MPC variables is usually `2*NW+8` cells. Character variables in the argument lists of `MPINPC, MPOUTC, MPINP` and `MPOUT` require up to `7.225*NW+100` cells of type `CHARACTER*1`. Many of these routines require either single precision or double precision scratch space or both. Single precision scratch space is contained in common block `MPCOM3`, while double precision scratch space is contained in common block `MPCOM4`. As a default, the package allocates 1024 cells in each of these two blocks.

The amount of single precision scratch space needed for an application varies widely depending on the routines used and the precision level `NW`. The maximum amount for each routine is given in Tables 3 and 4. The simplest way to determine the proper amount for a program is to run the code with an ample amount and then output the value of the parameter `IHS` in common `MPCOM1` upon completion. If insufficient space has been allocated, an error message will be output (error code 5) and execution will be terminated. If this occurs, the user must allocate a larger single precision array (with at least the number of words indicated in the error message) and place it in common `MPCOM3` in the user's main program. In addition, the parameter `IMS` in `MPCOM1` must be set to this larger number.

There is no nesting of double precision scratch space, and so the amount required is simple to determine. Let `NX` be the largest precision level `NW` used in an application. Then at most `NX+7` double precision cells are required for the basic MP routines, and at most `12*NX+6` cells are required for the advanced routines. If more than the default 1024 cells are required, the user must allocate a larger array and place it in common `MPCOM4` in the user's main program. Since it is straightforward to determine ahead of time the required level of double precision scratch space, the package does not perform automatic checking as it does for single precision scratch space.

If any of the advanced routines (i.e. those whose name ends in `X`) is called, the user must also allocate and initialize an array in common `MPCOM5`. The advanced routines should be called with a level of precision `NW` that is a power of two, so let `NX = 2**MX` be the largest level that will be used in a program. Then the user must allocate at least `8*NX` double

precision cells in common block MPCOM5 and must call MPINIX with argument MX to initialize the array in MPCOM5. Again, since the amount of space is straightforward to determine, no automatic checking is performed.

If the user does allocate arrays in MPCOM3, MPCOM4 or MPCOM5 with different sizes than the default 1024 cells, some systems may flag a warning message when the user's program is compiled and linked with the precompiled MPFUN package, since this usage is technically not allowed under a strict reading of the Fortran-77 standard. The author is not aware of any system that flags a fatal error for such usage, provided that the common blocks in the user's main program are at least as large as in this package.

The following sample program computes the value of the constant $\pi$ to approximately 7390 digits on 32 bit systems. For Cray systems, replace the constant 7.225 by 6.623, and the program will compute approximately 6770 digits.

```
      PROGRAM PIX
      DOUBLE PRECISION D, U
      CHARACTER*1 C
      PARAMETER (MX = 10, NX = 2 ** MX, NS = 9.5 * NX + 47,
     $ ND = 7.225 * NX)
      DIMENSION C(ND+50), PI(NX+4)
      COMMON /MPCOM1/ NW, IDB, LDB, IER, MCR, IRD, ICS, IHS, IMS
      COMMON /MPCOM3/ S(NS)
      COMMON /MPCOM4/ D(12*NX+6)
      COMMON /MPCOM5/ U(8*NX)
      NW = NX
      IMS = NS
      CALL MPINIX (MX)
      CALL MPPIX (PI)
      CALL MPOUT (6, PI, ND, C)
      STOP
      END
```

**Notes for Tables 3 and 4.**

In Tables 3 and 4, the argument PI denotes an MP value of $\pi$, which must have been previously computed by calling either MPPI or MPPIX. The argument AL2 denotes an MP value of $\log 2$, which must have been previously computed by calling MPLOG. Notation such as (A, N) in the third column denotes a DPE number, which has value A * 2**N. The variable CS denotes a CHARACTER string. In the scratch space column, NW is the precision level parameter in common MPCOM1. All other variables, unless indicated otherwise, denote MP numbers.

1. MPANG and MPANGX compute the MP angle A subtended by the MP pair [X, Y] considered as a point in the $x, y$ plane. This is more useful than an arctan or arcsin routine, since it places the result correctly in the full circle, i.e. in the interval $(-\pi, \pi]$.

| Routine Name | Calling Sequence | Functional Description | Single Prec. Scratch Space |
|---|---|---|---|
| DPADD | (A, NA, B, NB, C, NC) | (C, NC) = (A, NA) + (B, NB) | |
| DPDEC | (A, NA, B, NB) | (B, NB) = (A, NB) in decimal form. | |
| DPDIV | (A, NA, B, NB, C, NC) | (C, NC) = (A, NA) / (B, NB) | |
| DPMUL | (A, NA, B, NB, C, NC) | (C, NC) = (A, NA) * (B, NB) | |
| DPPWR | (A, NA, B, NB, C, NC) | (C, NC) = (A, NA) ** (B, NB) | |
| DPSQRT | (A, NA, B, NB) | (B, NB) = Sqrt (A, NA) | |
| DPSUB | (A, NA, B, NB, C, NC) | (C, NC) = (A, NA) - (B, NB) | |
| MPADD | (A, B, C) | C = A + B | |
| MPANG | (X, Y, PI, A) | A = Ang [X, Y]. See 1. | 15*NW+88 |
| MPANGX | (X, Y, PI, A) | A = Ang [X, Y]. See 1. | 19.5*NW+87 |
| MPCADD | (L, A, B, C) | C = A + B complex. See 2. | |
| MPCBRT | (A, B) | B = A ** (1/3) | 3*NW+15 |
| MPCBRX | (A, B) | B = A ** (1/3) | 4.5*NW+27 |
| MPCDIV | (L, A, B, C) | C = A / B complex. See 2. | 5*NW+20 |
| MPCDVX | (L, A, B, C) | C = A / B complex. See 2. | 8*NW+32 |
| MPCEQ | (L, A, B) | B = A complex. See 2. | |
| MPCMLX | (L, A, B, C) | C = A * B complex. See 2. | 4*NW+16 |
| MPCMUL | (L, A, B, C) | C = A * B complex. See 2. | 4*NW+16 |
| MPCPLX | (N, LA, A, X1, NX, LX, X) | Finds complex roots. See 3. | 18*NW+72 |
| MPCPOL | (N, LA, A, X1, NX, LX, X) | Finds complex roots. See 3. | 15*NW+75 |
| MPCPR | (A, B, IC) | Compares A and B. See 4. | |
| MPCPWR | (L, A, N, B) | B = A ** N complex. See 2. | 11*NW+55 |
| MPCPWX | (L, A, N, B) | B = A ** N complex. See 2. | 14*NW+56 |
| MPCSHX | (A, AL2, X, Y) | X = Cosh [A], Y = Sinh [A] | 16.5*NW+75 |
| MPCSQR | (A, B) | B = Sqrt [A] complex. See 2. | 6*NW+27 |
| MPCSQX | (A, B) | B = Sqrt [A] complex. See 2. | 7.5*NW+39 |
| MPCSSH | (A, AL2, X, Y) | X = Cosh [A], Y = Sinh [A] | 9*NW+50 |
| MPCSSN | (A, PI, X, Y) | X = Cos [A], Y = Sin [A] | 10*NW+53 |
| MPCSSX | (A, PI, X, Y) | X = Cos [A], Y = Sin [A] | 27.5*NW+119 |
| MPCSUB | (L, A, B, C) | C = A - B complex. See 2. | |
| MPDEB | (CS, A) | Outputs A preceded by the character string CS. | |
| MPDIV | (A, B, C) | C = A / B | |
| MPDIVD | (A, B, N, C) | C = A / (B, N). See 5. | |
| MPDIVX | (A, B, C) | C = A / B | 3*NW+12 |
| MPDMC | (A, N, B) | B = (A, N). See 5. | |
| MPEQ | (A, B) | B = A | |
| MPEXP | (A, AL2, B) | B = Exp [A] | 5*NW+25 |
| MPEXPX | (T, PI, AL2, Z) | Z = Exp [T] | 13.5*NW+63 |
| MPINFR | (A, B, C) | B = Int [A], C = Frac [A] | |
| MPINIX | (M) | Initializes for extra high precision. See 6. | |
| MPINP | (IU, A, CS) | Inputs A from unit IU. See 7. | 5*NW+27 |
| MPINPC | (A, N, B) | Converts for input. See 8. | 5*NW+27 |

Table 3: List of Routines

| Routine Name | Calling Sequence | Functional Description | Single Prec. Scratch Space |
|---|---|---|---|
| MPINQP | (CS, IB) | IB = value of MPCOM1 parameter with name CS. | |
| MPINRL | (N, LX, X, MN, MT, LR, R, IQ) | Finds integer relations. See 9. | N1*(NW+4) |
| MPINRX | (N, LX, X, MN, MT, LR, R, IQ) | Finds integer relations. See 9. | N2*(NW+4) |
| MPLOG | (A, AL2, B) | B = Log [A] | 8*NW+45 |
| MPLOGX | (Z, PI, AL2, T) | T = Log [Z] | 10.5*NW+51 |
| MPMDC | (A, B, N) | (B, N) = A | |
| MPMMPC | (A, B, L, C) | C = A + B i. See 2. | |
| MPMPCM | (L, A, B, C) | B = Real (A), C = Imag (A). See 2. | |
| MPMUL | (A, B, C) | C = A * B | |
| MPMULD | (A, B, N, C) | C = A * (B, N). See 5. | |
| MPMULX | (A, B, C) | C = A * B | |
| MPNINT | (A, B) | B = Nint [A] | NW+4 |
| MPNPWR | (A, N, B) | B = A ** N | 2*NW+10 |
| MPNPWX | (A, N, B) | B = A ** N | 5*NW+20 |
| MPNRT | (A, N, B) | B = A ** (1/N) | 6*NW+32 |
| MPNRTX | (A, N, B) | B = A ** (1/N) | 9*NW+36 |
| MPOUT | (IU, A, LA, CS) | Outputs A on unit IU. See 7. | 4*NW+22 |
| MPOUTC | (A, B, N) | Converts for output. See 8. | 4*NW+22 |
| MPPI | (PI) | PI = Pi | 8*NW+43 |
| MPPIX | (PI) | PI = Pi | 9.5*NW+47 |
| MPPOL | (N, L, A, X1, NX, X) | Finds real roots of polys. See 10. | 5*NW+25 |
| MPPOLX | (N, L, A, X1, NX, X) | Finds real roots of polys. See 10. | 8*NW+32 |
| MPRAND | (A) | A = Random number in [0, 1]. | |
| MPSETP | (CS, IB) | Sets MPCOM1 parameter with name CS to IB. | |
| MPSORT | (N, LA, A, IP) | Sorts entries of A. See 11. | 2*NW+8 |
| MPSQRT | (A, B) | B = Sqrt [A] | 3*NW+15 |
| MPSQRX | (A, B) | B = Sqrt [A] | 4.5*NW+27 |
| MPSUB | (A, B, C) | C = A - B | |

Table 4: List of Routines, Cont.

2. MPCADD, MPCDIV, MPCDVX, MPCEQ, MPCMPM, MPCMUL, MPCMLX, MPCPWR, MPCPWX, MPCSQR, MPCSQX, MPCSUB and MPMCMP perform operations on complex MP (MPC) numbers. In each of these routines L is the offset between the real and imaginary parts of the MPC arguments. L should be at least NW+4.

3. MPCPOL and MPCPLX find a complex root of the N-th degree polynomial whose MPC coefficients are in A by Newton-Raphson iterations, beginning at the complex DPE value (X1(1), NX(1)) + i (X1(2), NX(2)), and return the complex MP root in X. The N+1 coefficients $a_0, a_1, ..., a_N$ are assumed to start in locations A(1), A(2*LA+1), A(4*LA+1), etc. LA is the offset between the real and the imaginary parts of each input coefficient. Typically LA = NW+4. LX, also an input parameter, is the offset between the real and the imaginary parts of the MPC result to be stored in X. LX must be at least NW+4.

4. MPCPR compares the MP numbers A and B and returns in IC the value -1, 0, or 1 depending on whether A < B, A = B, or A > B. It is faster than merely subtracting A and B and looking at the sign of the result.

5. Note that if A is not an exact binary fraction, the MP converted value of the DPE number (A, N), while retaining all significant bits of precision in A, may not produce the intended result. An example of a value that is an exact binary fraction is 3.125. An example of a value that is not an exact binary fraction is 1.1. MPINPC may be used to perform fully accurate multiprecision decimal-to-binary conversion of constants that are not exact binary fractions.

6. MPINIX must be called prior to using any advanced routine (i.e. any routine whose name ends in X). Calling MPINIX with argument M initializes for precision level NW = 2**M. The user must also allocate 2**(M+3) double precision cells in an array in common MPCOM5. Only one call to MPINIX is required in a program.

7. MPINP inputs the MP number A on Fortran unit IU. CS is a scratch array with at least 7.225*NW+100 cells of type CHARACTER*1. The digits of A may span more than one line. A comma at the end of the last line denotes the end of the MP number. The input lines may not exceed 120 characters in length. Embedded blanks are allowed anywhere. See item 8 for the allowable format of an input MP number. MPOUT outputs the exponent plus LA mantissa digits of the MP number A on Fortran unit IU, with a comma at the end. Here the scratch array CS must have LA+25 cells of type CHARACTER*1.

8. MPINPC converts the CHARACTER*1 string A of length N to MP form in B. Strings input to MPINPC must be in the format 10^sa x tb.c where a, b and c are digit strings, s and t are - or blank, and x is either x or *. Blanks may be embedded anywhere. The digit string a is limited to nine digits and 80 total characters, including blanks. The exponent portion (i.e. the portion up to and including x) and the period may

optionally be omitted. MPOUTC converts the MP number A to the CHARACTER*1 array B of length N (an output parameter), which then can be output using nA1 format by the user. B must have at least 7.225*NW+30 cells.

9. MPINRL and MPINRX search for integer relations among the entries of the N-long MP vector X. An integer relation is an $n$-long vector $r$ such that $r_1x_1+r_2x_2+\cdots+r_nx_n = 0$. The entries of $x$ are assumed to start at X(1), X(LX+1), X(2*LX+1), etc. MN is the $\log_{10}$ of the maximum Euclidean norm of an acceptable relation. IQ is set to 1 if the routine succeeds in recovering a relation that (1) produces zero to within the relative tolerance 10**MT and (2) has Euclidean norm less than 10**MN. If no relation is found that meets these standards, IQ is set to 0. When a valid relation vector is recovered, it is placed in R, beginning at R(1), R(LR+1), R(2*LR+1), etc., where LR, like LX, is an input parameter. LR should be at least MN/6+3. When IDB is at least 5, norm bounds are output within which no relation can exist. The parameters N1 and N2 in the scratch space column denote 4*N**2 + 5*N + 12 and 4*N**2 + 5*N + 15, respectively.

10. MPPOL and MPPOLX find a real root of the N-th degree polynomial whose MP coefficients are in A by Newton iterations, beginning at the DPE value (X1, NX) and return the MP root in X. The N+1 coefficients $a_0, a_1, ..., a_N$ are assumed to start in locations A(1), A(L+1), A(2*L+1), etc. Typically L = NW+4.

11. MPSORT sorts the entries of the N-long MP vector A into ascending order using the quicksort algorithm. The entries of A are assumed to start at A(1), A(LA+1), A(2*LA+1), etc. The permutation vector that would sort the vector is returned in IP.