Multiprecision Translation and Execution of Fortran Programs

David H. Bailey

February 3, 1993

Ref: *ACM Trans. on Mathematical Software*, vol. 19, no. 3 (Sept. 1993), pg. 288–319

Abstract

This paper describes two Fortran utilities for multiprecision computation. The first is a package of Fortran subroutines that perform a variety of arithmetic operations and transcendental functions on floating point numbers of arbitrarily high precision. This package is in some cases over 200 times faster than that of certain other packages that have been developed for this purpose.

The second utility is a translator program, which facilitates the conversion of ordinary Fortran programs to use this package. By means of source directives (special comments) in the original Fortran program, the user declares the precision level and specifies which variables in each subprogram are to be treated as multiprecision. The translator program reads this source program and outputs a program with the appropriate multiprecision subroutine calls.

This translator supports multiprecision integer, real and complex datatypes. The required array space for multiprecision data types is automatically allocated. In the evaluation of computational expressions, all of the usual conventions for operator precedence and mixed mode operations are upheld. Furthermore, most of the Fortran-77 intrinsics, such as `ABS, MOD, NINT, COS, EXP` are supported and produce true multiprecision values.

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

Introduction

Section 1 of this paper gives an introduction to the problem of multiprecision computation, including a number of specific applications. Section 2 describes in moderate detail the multiprecision function package (MPFUN), emphasizing the algorithms and computational techniques used. For full details, including a listing of the subroutines and argument definitions, see [4].

Section 3 describes the multiprecision translator program (TRANSMP), including instructions for usage. It is not necessary to understand details of the MPFUN package in order to effectively use it via the translator program. Readers mainly interested in the translator may skip directly to section 3.

Full documentation and the actual source code for both of these programs may be obtained either from the ACM TOMS database or directly from the author by sending e-mail to `dbailey@nas.nasa.gov`.

1.0 Applications of Multiprecision Computation

Multiprecision computation (i.e. computation using numeric precision beyond the single or double precision ordinarily provided in hardware) has been performed on electronic computers since the earliest models were introduced over forty years ago. One question that is frequently raised in this context is what applications justify a multiprecision facility. Actually, there are quite a number of applications, ranging from the highly theoretical to the completely practical.

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. 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 γ is believed to be transcendental, it has not been proven that γ 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 δ constant ($4.669201609 \cdots$) [13, 19] and the Bernstein β 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 α 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, 20, 21]. 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 α. 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.

The author has performed some computations of this type [3, 5], and others are in progress. Some recent results include the following: if γ satisfies an integer polynomial of degree 50 or less, then the Euclidean norm of the coefficients must exceed 7×10^{17}; if Feigenbaum's δ constant satisfies an integer polynomial of degree 20 or less, then the Euclidean norm of the coefficients must exceed 2×10^{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 ζ 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 2×10^{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 $4913 + 2108t^2 - 604t^3 - 977t^4 + 8t^5 + 44t^6 + 392t^7 - 193t^8 - 40t^9 + 48t^{10} - 12t^{11} + t^{12}$ (verified to a precision level of over 1000 digits). 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 π 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 π 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 [1]).

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.

2.0 Overview of the MPFUN 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 2.1 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.

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 some minimal accuracy 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).

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. Versions of this program have on numerous occasions disclosed hardware and software bugs in scientific computer systems.

2.1 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.

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 [23]. 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 [23]. 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:

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

$$\begin{aligned}
&\cdot \\
z_{2n-3} &= x_{n-1}y_{n-2} + x_{n-2}y_{n-1} \\
z_{2n-2} &= x_{n-1}y_{n-1} \\
z_{2n-1} &= 0
\end{aligned}$$

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 [2]. 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.

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 $1/b$:

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

Multiplying the final approximation to $1/b$ by a gives the quotient. Note that this algorithm involves only an addition and a subtraction, plus two multiplications, which can be performed using the FFT-based technique mentioned above. Also note that the term in parentheses is small. In fact, the product of x_k with this term can be performed with half the normal level of precision.

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 2.5 times the cost of the final iteration. The total cost of a multiprecision division is only about five times the cost of a multiprecision multiplication operation of equivalent size.

2.2 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 previous 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 $1/\sqrt{a}$:

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

Multiplying the final approximation to $1/\sqrt{a}$ by a gives the square root. 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 $a^{-2/3}$:

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

Multiplying the final approximation to $a^{-2/3}$ by a gives the cube root.

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 [23]. 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 $a^{-1/n}$:

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

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 section 1.0, 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 [21], 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. Neither algorithm will be presented here. Interested readers are referred to the respective papers.

2.3 Computing π

The computation of π to high precision has a long and colorful history. Interested readers are referred to [6] for discussion of the classical history of computing π. Recently a number of advanced algorithms have been discovered for the computation of π 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 π. 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.

2.4 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 $-0.001 < r \le 0.001$, 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

$$\begin{aligned} a_{k+1} &= \frac{a_k + b_k}{2} \\ b_{k+1} &= \sqrt{a_k b_k} \end{aligned}$$

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 by the Borweins in [7, 8]. 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 $-\pi/32 < s \le \pi/32$, 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 $e^{ix} = \cos x + i \sin x$).

2.5 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 π 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.

2.6 MPFUN 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 described below, 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.

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 π 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 package and Brent's package. The algorithm used by Mathematica is not mentioned in the Mathematica reference book [32], but it is probably either the Salamin-Brent algorithm or one of the Borwein algorithms. The algorithm used by Brent's package for computing π is the Salamin-Brent algorithm, basically the same as described in section 2.3.

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 MFLOPS and a Linpack performance of 4.9 MFLOPS (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 MFLOPS and a Linpack performance of 90 MFLOPS. 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.

It can be seen from these results that the MPFUN package is the fastest of the three at all precision levels on both systems. On the SGI system, MPFUN is nearly twice as fast as Mathematica and three times as fast as Brent's package for the lowest 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 seven times faster than Mathematica and 18 times faster than Brent's package.

m	Digits	MPFUN	Mathematica	Brent
3	60	0.011	0.020	0.033
4	115	0.024	0.040	0.062
5	230	0.053	0.100	0.174
6	460	0.139	0.340	0.543
7	925	0.444	1.190	2.150
8	1,850	1.420	4.570	8.610
9	3,700	4.880	17.410	34.550
10	7,400	14.150	68.240	145.330
11	14,795	41.800	270.600	619.160
12	29,590	146.890	1080.000	2661.640
13	59,185	504.090		
14	118,370	1361.190		

Table 1: SGI Workstation Performance Results (seconds)

m	Digits	MPFUN	Brent
4	105	0.006	0.019
5	210	0.010	0.037
6	425	0.018	0.088
7	845	0.037	0.229
8	1,695	0.088	0.815
9	3,390	0.177	3.176
10	6,780	0.386	13.040
11	13,565	0.772	54.620
12	27,125	1.598	230.800
13	54,250	3.450	975.600
14	108,505	7.503	
15	217,010	16.710	
16	434,020	36.490	
17	868,045	81.690	
18	1,736,090	173.300	

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

On the Cray Y-MP, the MPFUN package is three times faster than Brent's package at the lowest precision level and 280 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 MFLOPS, 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.

3.0 Overview of the Multiprecision Translator

Conversion of a conventional scientific application program to use the MPFUN routines is generally straightforward, but it is often tedious and error prone. For example, if the slightest error is made in any of the arguments to the many subroutine calls, not only will the results be in error, but the program may abort with little information to guide the programmer. As a result of these difficulties, few serious scientific programs have been manually converted to use the MPFUN routines. Similar difficulties have plagued programmers who have attempted to use other multiprecision systems, such as Brent's package [11].

To facilitate such conversions, the author has developed a translator program that accepts as input a conventional Fortran-77 program to which has been added certain special comments that declare the desired level of precision and specify which variables in each subprogram are to be treated as multiprecision. This translator then parses the input code and generates an output program that has all of the calls to the appropriate MPFUN routines. This output program may then be compiled and linked with the MPFUN package for execution.

This translation program allows one to extend the Fortran-77 language with the datatypes `MULTIP INTEGER, MULTIP REAL` and `MULTIP COMPLEX`. These datatypes can be used for integer, floating point or complex numbers of an arbitrarily high, 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` constructs, function calls, `READ` and `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.

This processor translates programs to use the standard MP routines from the author's MPFUN package. If one wishes to utilize this translator in connection with the extra-high precision routines of this package, which are designed for precision levels greater than about 1,000 digits, contact the author for instructions.

3.1 Operation of the Translator Program

This translator program should run on any Fortran-77 system that supports recursive subroutine references. On some systems, including Sun and IBM workstations, a minor source modification and/or a special compiler option must be enabled to permit the program to run correctly. Detailed instructions for compiling and testing the translator on various systems are given in a "read-me" file that accompanies the program code. The translator has been successfully implemented on Cray supercomputers, Sun workstations, SGI workstations, IBM workstations and mainframes (AIX operating system), DEC workstations, HP workstations and Intel parallel computers.

The translator is in effect a compiler in the sense that it identifies and analyzes every input statement. It develops a symbol table that contains type and dimension information for all variables used in a subprogram. A number of Fortran statements, such as `DO, CONTINUE` and `OPEN` statements, are not modified by the translator. Most other statements are analyzed in detail, including type declarations, `IMPLICIT, COMMON, DIMENSION, PARAMETER, READ, WRITE` and `CALL` statements, as well as all assignment statements.

If any input statement is modified or translated, the original statement is included in the output file as a comment, starting with the string `CMP>`. The comment `CMP<` is placed after the translated lines. Warnings and error messages are also written in the output file. Warnings are issued as comments starting with `CMP*`. Fatal error messages start with `***`. When a fatal error is detected, the message is output on the output file, and processing is terminated. Thus to make sure that the translation of an input program was successful, check the end of the output file to make sure there is no fatal error message. It is also strongly recommended that the output program be scanned for `CMP*` warning messages before it is compiled and executed.

3.2 Precision Level and Explicit Type Directives

In the following, an MP statement will be defined as a statement that has at least one MP variable. An MP subprogram will be defined as a subprogram with at least one MP variable. Table 3 gives several datatype abbreviations that will be used hereafter in this paper.

At the beginning of a file containing a conventional Fortran-77 code to be translated, before any program or subroutine statement, a directive (i.e. special comment) of the following form must be inserted:

```
CMP+ PRECISION LEVEL 120
```

IN	Integer
SP	Single precision real
DP	Double precision real
CO	Single precision complex
DC	Double precision complex (non-ANSI extension of Fortran-77)
MPI	Multiprecision integer
MPR	Multiprecision real
MPC	Multiprecision complex
MP	Denotes the three multiprecision types collectively

Table 3: Datatype Abbreviations

This denotes that the maximum precision level to be employed in this program is 120 digits. Only one such declaration is allowed in a single file, and Fortran-77 files whose translated routines later will be linked together must have equivalent precision level declarations. This directive must precede any of the other `CMP+` directives to be described below. This and all other MP directives described in this paper may alternately be written with lower case alphabetics.

Variables in a subprogram of the input Fortran-77 program file that are to be treated as MP by the translator program may be declared by explicit MP type directives, such as the following:

```
CMP+ MULTIP INTEGER IA, IPR, KMAX3
CMP+ MULTIP REAL SUM, TOL34, X, Y
CMP+ MULTIP COMPLEX W, ZAB
```

An MP variable must be declared prior to any appearance of that variable in the subprogram, including any appearance in a type declaration, `DIMENSION` or `COMMON` statement. An exception to this rule is that MP variable names appearing in the argument list of a `FUNCTION` or `SUBROUTINE` statement may be afterwards declared. However, if the function name of a function subprogram is to have an MP type, this name must be declared with an MP type directive immediately preceding the `FUNCTION` statement. The dimensions for MP variables are not included in MP type directives. These dimensions will be taken from the standard type declaration, `DIMENSION` or `COMMON` statement where these dimensions are defined in the original program.

3.3 Implicit Type Directives

Many Fortran-77 codes utilize implicit typing of variables, either with the default convention or with `IMPLICIT` statements. For example, many programmers use an `IMPLICIT` statement to automatically declare all variables whose first letters are in the ranges `A-H` and `O-Z` to be DP. To simplify the translation of such code, implicit MP type directives may be used, as in these examples:

```
CMP+ IMPLICIT MULTIP REAL (A-H, O-Z)
CMP+ IMPLICIT MULTIP INTEGER (M)
CMP+ IMPLICIT MULTIP COMPLEX (C, Z)
```

Implicit MP type directives should appear at the beginning of a subprogram, just like standard `IMPLICIT` statements. An implicit MP type directive overrides any standard `IMPLICIT` statement, but it does not override either an explicit MP type directive or a standard type statement. In function subprograms, an implicit MP type directive may not be used to declare the type of the function name. Use an explicit MP type directive for this purpose, placed immediately before the `FUNCTION` statement.

3.4 The `SAFE` and `FAST` Options

Expressions involving MP variables and constants are evaluated using the operator precedence conventions of Fortran-77, and using predictable extensions of the Fortran-77 mixed mode conventions. There are two options for the evaluation of mixed mode operations: `FAST` and `SAFE`. The difference between these conventions may be seen with the following example, where `A` and `B` are MPR and `N` is an ordinary integer variable:

```
      B = A + 1.D0 / N
```

With the `FAST` option, the subexpression `1.D0 / N` is evaluated using DP arithmetic, and the result temporary has type DP. With the `SAFE` option, which is the default, `1.D0 / N` is performed using MP arithmetic, and the result temporary has type MPR. As the name signifies, the `FAST` option produces somewhat more efficient translated code, but it may also give unexpectedly inaccurate answers, for instance if `N` in the above example has the value 7.

An exception to the `SAFE` option is in the argument lists of subroutine calls or non-intrinsic function references. Expressions appearing in these lists are always evaluated using the `FAST` option, since this corresponds more closely to the Fortran convention that most users expect. Thus in the statement

```
      B = 3 * FUN (N - 1, A)
```

the subexpression `N - 1` is always evaluated using ordinary integer arithmetic, and the result temporary, which is passed to `FUN`, has type IN.

The user may switch between these options by inserting one of the following directives in the declaration section of any subprogram:

```
CMP+ MIXED MODE FAST
CMP+ MIXED MODE SAFE
```

For the operators `+ - * /`, Tables 4 and 5 give the types of results with these two options. Table 6 lists the argument types and results defined for the `**` operator. In Table 6, if a particular combination is not listed, or if its position in the table is blank, then it is not defined. Comparison operations (i.e. `.EQ.`, `.LT.`, etc.), where one or both of the operands are MP, are permitted both in logical `IF` statements and in logical assignment statements. If one of the operands has type CO, DC or MPC, only `.EQ.` and `.NE.` comparisons are permitted.

3.5 Multiprecision Constants

With the `SAFE` option, all IN constants appearing in MP statements are considered MPI constants and are converted to full precision, and all SP or DP constants in MP statements are considered MPR constants and are converted to full precision.

With the `FAST` option, IN, SP and DP constants are recognized and treated as such by the translator. They are merely passed unchanged to the output program and are converted to binary by the underlying Fortran system. For modest sized whole numbers and exact binary fractions, these constants are converted exactly and produce accurate results when they appear in expressions with MP variables. However, SP or DP constants that cannot be precisely converted (i.e. `1.01D0`), or IN, SP or DP constants that have more significant digits than can be exactly accommodated in these datatypes, may result in inaccurate MP calculations.

To avoid such difficulties with the `FAST` option, the user may explicitly specify that a constant in the input program will be treated as an MP constant for the output program. This is done by appending `+0` to the constant, as in the following examples:

```
3+0
-12345678901234567890+0
3.141592653589793+0
1.2345678901234567890D-13+0
```

The first two constants have type MPI, and the last two have type MPR. Embedded blanks are allowed anywhere in these constants. MP constants must appear in a context where the plus operation would actually be performed between the two components of the MP constant if interpreted according to the standard Fortran rules for evaluating expressions. For example, the expression `N*12345+0` is not treated as containing an MP constant. Write this as `N*(12345+0)` if so intended. MP constants are recognized as such only in MP statements.

There is no definition of this sort for MPC constants, but MPC constants may be defined by using the special conversion function `DPCMPL` (see section 3.6), where the two arguments are MPR constants.

MP constants may be defined symbolically using `PARAMETER` statements. The parameter assignment expression for an MP variable may reference previously defined MP and non-MP parameters, and it may also include intrinsic function references. All such assignments are performed upon entry to the subprogram the first time it is called.

3.6 Intrinsic Functions

Table 7 lists Fortran intrinsic functions that are supported by this translator with MP arguments. References to these functions will be automatically translated to call

Arg. 1 / Arg. 2	IN	SP	DP	CO	DC	MPI	MPR	MPC
IN	IN	SP	DP	CO	DC	MPI	MPR	MPC
SP	SP	SP	DP	CO	DC	MPR	MPR	MPC
DP	DP	DP	DP	DC	DC	MPR	MPR	MPC
CO	CO	CO	CO	CO	DC	MPC	MPC	MPC
DC	DC	DC	DC	DC	DC	MPC	MPC	MPC
MPI	MPI	MPR	MPR	MPC	MPC	MPI	MPR	MPC
MPR	MPR	MPR	MPR	MPC	MPC	MPR	MPR	MPC
MPC	MPC	MPC	MPC	MPC	MPC	MPC	MPC	MPC

Table 4: Results of Mixed Mode Arithmetic Operations with the `FAST` option

Arg. 1 / Arg. 2	IN	SP	DP	CO	DC	MPI	MPR	MPC
IN	MPI	MPR	MPR	MPC	MPC	MPI	MPR	MPC
SP	MPR	MPR	MPR	MPC	MPC	MPR	MPR	MPC
DP	MPR	MPR	MPR	MPC	MPC	MPR	MPR	MPC
CO	MPC	MPC	MPC	MPC	MPC	MPC	MPC	MPC
DC	MPC	MPC	MPC	MPC	MPC	MPC	MPC	MPC
MPI	MPI	MPR	MPR	MPC	MPC	MPI	MPR	MPC
MPR	MPR	MPR	MPR	MPC	MPC	MPR	MPR	MPC
MPC	MPC	MPC	MPC	MPC	MPC	MPC	MPC	MPC

Table 5: Results of Mixed Mode Arithmetic Operations with the `SAFE` option (default)

		Result	
Arg. 1	Arg. 2	`FAST`	`SAFE`
IN	IN	IN	MPI
IN or SP	SP	SP	MPR
IN, SP or DP	DP	DP	MPR
IN, SP or CO	CO	CO	
IN, SP, DP, CO or DC	DC	DC	
IN	MPI	MPI	MPI
IN, SP or DP	MPR	MPR	MPR
CO	IN	CO	MPC
CO	SP	CO	
CO	DP or DC	DC	
DC	IN	DC	MPC
DC	SP, DP, CO or DP	DC	
MPI	IN or MPI	MPI	MPI
MPI	SP, DP or MPR	MPR	MPR
MPR	IN, SP, DP, MPI or MPR	MPR	MPR
MPC	IN	MPC	MPC

Table 6: Defined Combinations for the `**` Operator

the appropriate routines from the MPFUN package, provided the arguments are of the appropriate MP type. If the `SAFE` option is in effect, non-MP arguments are first converted to MP, so that true MP results are always returned. If the user requires either a function not listed here or a function with an argument type not listed here, contact the author.

Note that Table 7 does not include any of the (obsolescent) type-specific Fortran-77 functions (i.e. `AMOD, DABS, MIN0`, etc.). This is in keeping with the Fortran-77 convention that these are defined only for specific IN, SP and DP argument types. References to these functions are not permitted in MP statements. Use the equivalent generic Fortran-77 functions (i.e. `MOD, ABS, MIN`, etc.) instead.

Also note in Table 7 that the conversion intrinsics of Fortran-77, namely `INT, CMPLX, DBLE, DCMPLX` and `REAL`, return results of types IN, CO, DP, DC and SP, respectively, even though the arguments have MP types. This is in keeping with the conventions of Fortran-77. If one wishes to truncate an MPR number to MPI, form an MPC number from two MPR numbers, or extract the MPR real and imaginary components of an MPC number, the special functions `MPINT, DPCMPL, DPREAL, DPIMAG` (see Table 8) should be used instead. These special functions are not defined for ordinary SP, DP, CO or DC arguments in the translated program (although they may be in the input program). Thus, for example, `DPREAL` cannot be used to convert a DP number to MPR. Type conversions such as this can be performed either by simple assignment statements, or else by defining an external MP function.

To preserve comparable functionality between an input Fortran-77 program that uses one of these four special conversion functions and the output MP program, equivalent SP or DP function subprograms should be included in the input file. Table 9 has some examples of equivalent definitions for these functions that use DP and DC datatypes. If your program uses ordinary SP and CO datatypes instead, these sample subprograms need to be changed accordingly.

Do not place any MP directives in these function subprograms. If another subprogram references one of these functions, it should declare the argument and function names to be of the appropriate types (i.e. IN, DP or DC). However, the names `MPINT, DPCMPL, DPREAL` and `DPIMAG` do not need to be declared with MP type directives in the subprograms where they are referenced. In the output program, MP results will be automatically be returned with types according to Table 8, and these sample subprograms will be ignored.

With the `FAST` option, non-MP arguments to intrinsic functions appearing in MP statements are passed without change to the non-MP intrinsic functions. For non-MP arguments the translator recognizes the following "generic" intrinsic function names and assigns result types according to argument types, in accordance with the standard Fortran conventions:

```
ABS, ACOS, AINT, AIMAG, ANINT, ASIN, ATAN, ATAN2, CHAR, CMPLX, CONJG,
COS, COSH, DBLE, DCMPLX, DIM, DIMAG, DREAL, EXP, ICHAR, INDEX, INT, LEN,
LOG, LOG10, MAX, MIN, MOD, NINT, REAL, SIGN, SIN, SINH, SQRT, TAN, TANH.
```

Note that this list, like Table 7, does not include any of the type-specific Fortran-77 intrinsic functions (i.e. `AMOD, DABS, MIN0`, etc.). References to these functions are not permitted

Function	Arg. 1	Arg. 2	Result
`ABS`	MPI		MPI
	MPR		MPR
	MPC		MPR
`ACOS`	MPR		MPR
`AINT`	MPR		MPR
`ANINT`	MPR		MPR
`ASIN`	MPR		MPR
`ATAN`	MPR		MPR
`ATAN2`	MPR	MPR	MPR
`CMPLX`	MPC		CO
`CONJG`	MPC		MPC
`COS`	MPR		MPR
`COSH`	MPR		MPR
`DBLE`	MPI		DP
	MPR		DP
	MPC		DP
`DCMPLX`	MPC		DC
`EXP`	MPR		MPR
`INT`	MPI		IN
	MPR		IN
	MPC		IN
`LOG`	MPR		MPR
`LOG10`	MPR		MPR
`MAX`	MPI	MPI	MPI
	MPR	MPR	MPR
`MIN`	MPI	MPI	MPI
	MPR	MPR	MPR
`MOD`	MPI	MPI	MPI
	MPR	MPR	MPR
`NINT`	MPI		MPI
	MPR		MPR
`REAL`	MPI		SP
	MPR		SP
	MPC		SP
`SIGN`	MPI	MPI	MPI
	MPR	MPR	MPR
`SIN`	MPR		MPR
`SINH`	MPR		MPR
`SQRT`	MPR		MPR
	MPC		MPC
`TAN`	MPR		MPR
`TANH`	MPR		MPR

Table 7: Fortran Intrinsics Supported with MP Arguments

Function	Arg. 1	Arg. 2	Result
`MPINT`	MPR		MPI
`DPCMPL`	MPR	MPR	MPC
`DPREAL`	MPC		MPR
`DPIMAG`	MPC		MPR

Table 8: Special MP Conversion Functions

```
FUNCTION MPINT (X)
DOUBLE PRECISION X
MPINT = INT (X)
RETURN
END

FUNCTION DPCMPL (A, B)
DOUBLE COMPLEX DPCMPL
DOUBLE PRECISION A, B
DPCMPL = DCMPLX (A, B)
RETURN
END

FUNCTION DPREAL (C)
DOUBLE PRECISION DPREAL
DOUBLE COMPLEX C
DPREAL = DBLE (C)
RETURN
END

FUNCTION DPIMAG (C)
DOUBLE PRECISION DPIMAG
DOUBLE COMPLEX C
DPIMAG = DIMAG (C)
RETURN
END
```

Table 9: DP Equivalents of the Special Conversion Functions

in MP statements. Use the equivalent generic Fortran-77 functions (i.e. `MOD, ABS, MIN`, etc.) instead.

3.7 Other Special Functions and Constants

Whenever the translator encounters a reference to `COS` or `SIN` in the source program, it inserts a call to the MPFUN routine `MPCSSN`. However, in many instances the user's code requires both function values for a single argument, often computed in adjacent lines of code. Since `MPCSSN` actually returns both the cosine and sine of the input argument at no extra cost, the two calls to `MPCSSN` are redundant and may represent a significant waste of computing time.

If run-time performance is an issue in such programs, the user may optionally replace the separate references to `COS` and `SIN` with a single call to the special MP subroutine `DPCSSN`, which has three arguments: the first is the input value, and the second and third are the output cosine and sine values. The translator recognizes this subroutine name and will substitute a call to `MPCSSN` to produce MP results. For compatibility purposes, a functional equivalent of `DPCSSN` should be included in the program file. A DP example is shown in Table 10. The analogous subroutine name recognized for the hyperbolic functions `COSH` and `SINH` is `DPCSSH` (see Table 10).

Another operation of this nature is root extraction, i.e. `B = A ** (1.D0 / N)`, for which the efficient routine `MPNRT` exists in the MPFUN package. Thus it is recommended (for improved run-time performance) that any code in the input program that performs root extraction using the `**` operator be changed to reference the function `DPNRT` instead, i.e. `B = DPNRT (A, N)`. A DP equivalent of `DPNRT` is shown in Table 10.

One additional special function that many users may find useful produces pseudorandom MPR numbers. The routine `MPRAND` in the MPFUN package generates pseudorandom numbers uniformly in the range $(0, 1)$. To access this routine by means of the translator, one references the special function `DPRAND`. This function has no arguments. One references it by means of statements such as `A = 3 * DPRAND ()`. It is not possible to write a completely equivalent DP version of this routine. However, the basic pseudorandom number functionality can be reproduced by means of a simple routine such as the one shown in Table 10.

The sample program definitions for `DPCSSN, DPCSSH, DPNRT` and `DPRAND` in Table 10, like the definitions of the special conversion functions in Table 9, are only for the purpose of providing comparable functionality when the input program is run with ordinary SP or DP arithmetic, and are ignored in the translated program. Do not place any MP directives in any of these sample subprograms. If another subprogram references either `DPNRT` or `DPRAND`, it should declare the function name to be of the appropriate type (DP in the examples above). However, the names `DPNRT` and `DPRAND` do not need to be declared with an MP type directive in subprograms that reference them.

The constants $\log 2 = 0.69314\cdots$, $\log 10 = 2.30258\cdots$ and $\pi = 3.14159\cdots$ are computed in the program initialization and are available in any subprogram that contains MP variables. These values may be referenced by the user by means of the special variable

```
      SUBROUTINE DPCSSN (A, X, Y)
      DOUBLE PRECISION A, X, Y
      X = COS (A)
      Y = SIN (A)
      RETURN
      END

      SUBROUTINE DPCSSH (A, X, Y)
      DOUBLE PRECISION A, X, Y
      X = COSH (A)
      Y = SINH (A)
      RETURN
      END

      FUNCTION DPNRT (A, N)
      DOUBLE PRECISION A, DPNRT
      DPNRT = A ** (1.D0 / N)
      RETURN
      END

      FUNCTION DPRAND ()
C
C   This routine returns a pseudorandom DP floating number nearly uniformly
C   distributed between 0 and 1 by means of a linear congruential scheme.
C   2^28 pseudorandom numbers with 30 bits each are returned before repeating.
C
      IMPLICIT DOUBLE PRECISION (A-H, O-Z)
      PARAMETER (F7 = 78125.D0, R30 = 0.5D0 ** 30, T30 = 2.D0 ** 30)
      SAVE SD
      DATA SD/314159265.D0/
C
      T1 = F7 * SD
      T2 = AINT (R30 * T1)
      SD = T1 - T30 * T2
      DPRAND = R30 * SD
C
      RETURN
      END
```

Table 10: Suggested DP Equivalents of `DPCSSN`, `DPCSSH`, `DPNRT` and `DPRAND`

names `DPL02, DPL10` and `DPPIC`. Whenever any of these names appears in a statement, the translator substitutes the MP value. For compatibility purposes, any subprogram that references one of these constants should declare it to be SP or DP and set its approximate decimal value in a parameter statement. Example:

```
      DOUBLE PRECISION DPPIC
      PARAMETER (DPPIC = 3.141592653589793D0)
```

This parameter statement will be ignored in the output program, and the MP value will be used instead. The names `DPL02, DPL10` and `DPPIC` do not need to be declared with an MP type directive. Do not attempt to define any of these values by means of assignments or function calls.

3.8 Input and Output of MP Numbers

MP variables may appear in `READ` or `WRITE` statements only with the following two special forms:

```
      WRITE (6, *) VAR1, VAR2(I), VAR3(I,J)
      READ (11) VAR1, VAR2, VAR3
```

Either form may be a `READ` or `WRITE`, but neither may employ implied `DO` loops. Convert implied `DO` loops to explicit `DO` loops instead. The unit numbers may be integer variables instead of integer constants. Non-MP variables and constants may be included in the list, in which case they are handled using ordinary Fortran I/O.

The first form is used for input and output of individual MP numbers (not entire unsubscripted arrays) in ordinary decimal form. The digits of the number may span more than one line. A comma at the end of the last line denotes the end of an MP number. Input lines may not exceed 120 characters in length, but embedded blanks are allowed anywhere. The exponent is optional in an input number, but if present it must appear first, as in the following example:

```
   10 ^ -4 x  3.14159 26535 89793 23846 26433 83279
   50288 41971 69399 37510,
```

MPC numbers are input or output as two consecutive MPR numbers. The output of an MP write operation is in the correct form for a subsequent MP read operation. By default, all digits of an MP number are output. The user can control the number of mantissa digits output by including a directive such as

```
CMP+ OUTPUT PRECISION 200
```

in the declaration section of any subprogram. It remains in effect until the end of file or until another such directive is encountered.

The second form of `READ/WRITE` statement above is used to perform binary I/O of entire MP arrays. Subscripted variables are not allowed in the second form.

3.9 Controlling the Multiprecision "Epsilon" and Precision Level

Many programs need to control the MP "epsilon" for performing comparisons. To this end, the user can reference the special MP constant `DPEPS`. For compatibility purposes, any subprogram that uses `DPEPS` should declare it to be SP or DP and set it to some nominal small value in a parameter statement. Example:

```
      DOUBLE PRECISION DPEPS
      PARAMETER (DPEPS = 1D-16)
```

Whenever this name appears in a subprogram that contains MP variables, the translator substitutes the MP "epsilon" value, which by default is 10^{7-D}, where D is the number of digits of precision specified in the precision level directive. `DPEPS` does not need to be declared with an MP type directive. The MP epsilon value may be modified (independent of the precision level directive) by inserting a directive such as

```
CMP+ EPSILON 1E-200
```

in the declaration section of any subprogram (for instance, adjacent to the parameter statement in which `DPEPS` is defined). It remains in effect until the end of file or until another such directive is encountered.

The number of mantissa words allocated by the translator for MP numbers is approximately one seventh the number of digits specified in the precision level directive. The first dimension of MP arrays is this number plus 4. The user may access the number of mantissa words in the special constant `MPNWP`. For compatibility purposes, any subprogram that uses `MPNWP` should be declare it to be of type IN and set it to some nominal integer value in a parameter statement. Example:

```
      INTEGER MPNWP
      PARAMETER (MPNWP = 1)
```

`MPNWP`, like `MPL02` and `MPPIC`, is considered a constant and may not be changed. If one wishes to dynamically change the working precision level within a program (which is not recommended for novice users), this may be done by calling the MPFUN routines `MPSETP` and `MPINQP`, as follows:

```
      CALL MPSETP ('NW', 35)
      CALL MPINQP ('NW', NX)
```

The first line sets the working precision level to 35 words. This value must not be greater than the value of `MPNWP`. The second line sets `NX` to be the value of the current working precision. If the user is not concerned about possible name conflicts, the same functions can be accomplished by simply including the MPFUN common block

```
      COMMON /MPCOM1/ NW, IDB, LDB, IER, MCR, IRD, ICS, IHS, IMS
```

in the subprogram and directly modifying the variable `NW`.

3.10 Single Precision Scratch Space for the MPFUN Package

The maximum amount of SP scratch space in common block `MPCOM3` (see the documentation for the MPFUN package [4]), cannot be determined in advance by the translator program. The MPFUN package allocates 1024 SP cells in this block, which for most programs is sufficient. If the “insufficient single precision scratch space” error is encountered during execution of the resulting MP program, place a directive of the form

```
CMP+ SCRATCH SPACE 2000
```

at the beginning of the input file, before the `PROGRAM` statement but after the precision level directive. The number placed on this line should be at least the size mentioned in the error message.

3.11 Other Restrictions and Limitations

It should be emphasized again that the Fortran-77 language is not perfectly or completely supported by the translator. In addition to the restrictions already mentioned, a number of other limitations apply. A complete list is included below. However, note that in almost every case there is a simple change that can be made to the input program to make it acceptable to this translation program, while retaining both its functionality and Fortran-77 compliance. The majority of these restrictions are merely good programming practice.

1. A number of identifiers beginning with `DP` and `MP` are reserved for use by the translator, and the translator will flag an error if any of these appears in the user's input program. To be safe, do not use such names in your program, other than as instructed in this paper.

2. `ENTRY`, typed `FUNCTION` (i.e. `INTEGER FUNCTION`), assigned `GOTO`, arithmetic `IF`, `READ` or `WRITE` without parentheses, and `PRINT` statements are not allowed. Please replace these constructs, which in most cases are obsolescent, with more conventional alternatives: `FUNCTION` statements followed by type statements, normal subroutine calls, computed or ordinary `GOTO` statements, logical `IF` statements and normal `READ` or `WRITE` statements, respectively.

3. References to the (obsolescent) type-specific Fortran-77 intrinsic functions (i.e. `AMOD, DABS, MIN0`, etc.) are not allowed in MP statements. Use the equivalent generic Fortran-77 functions (i.e. `MOD, ABS, MIN`, etc.) instead.

4. Statement functions may not be used to define MP functions. Convert these into MP function subprograms or subroutines.

5. MP variables may not appear in `DATA` statements. Convert these into parameter or assignment statements.

6. MP variables or constants may not appear in `DO` statements, array dimensions or array subscripts.
7. An MP statement may not be the terminal line of a `DO` loop. Place the line number on a `CONTINUE` line immediately following the statement. If the line number is also the target of a `GOTO`, the `DO` loop must be changed to use a separate terminal line number.
8. MP variables or constants may not appear in formatted `READ` or `WRITE` statements, and other restrictions apply to the I/O of MP data. See section 3.8 for details.
9. The logical operators `.NOT.`, `.EQV.` and `.NEQV.` may not appear in MP statements. Rewrite such statements using `.AND.` and `.OR.` operators, or move such subexpressions to a separate statement.
10. Complex constants [i.e. (3., 2.)] may not appear in MP statements. Either use the intrinsic functions `CMPLX` or `DCMPLX`, or else assign such constants to CO or DC variables in separate statements.
11. Except for variables in the argument list, variables that appear in a type statement or an MP type directive may not have previously appeared in the subprogram.
12. A single `IMPLICIT` statement may be used to declare the initial letter(s) for only one datatype. A single `COMMON` statement may be used to declare only one common block.
13. `DATA` statements and `FORMAT` statements may appear only after the end of the specification section of the program, i.e. only after type declaration, `DIMENSION` statements, `COMMON` statements, etc.
14. Fortran keywords (i.e. `CALL, DO, IF, READ, RETURN`, etc.) may not be used as identifiers.
15. Embedded blanks may not appear in Fortran keywords, line numbers, variable names, comparison operators and logical operators. Exceptions: `DOUBLE PRECISION, DOUBLE COMPLEX, ELSE IF, END DO, END IF, GO TO` are permitted.
16. Fortran keywords must be followed by a blank or an operator. Also, a blank must follow the line number in a `DO` statement.
17. If an integer constant is followed by a comparison or logical operator, the constant and the operator must be separated by a blank (i.e. `12340 .LE. X`).
18. Input code must be in the standard 72 column format. Comments up to 80 characters long are correctly copied to the output file.
19. Tab characters are not allowed. Convert these to blanks with a text editor.

On the other hand, this program will correctly process code with the following features, which do not comply with the Fortran-77 standard, provided the user's Fortran compiler also supports such constructs:

1. Both upper and lower case alphabetics may be used in identifiers and Fortran keywords.
2. Long variable names (up to 16 characters long) are permitted.
3. Character strings may be delimited with pairs of quotation marks ["] instead of apostrophes ['].
4. The double complex (DC) datatype is supported, including DC intrinsics.
5. The `IMPLICIT NONE` statement is supported. Untyped variables found in executable MP statements will be flagged as errors.
6. The datatypes `INTEGER*4, REAL*8`, etc. are supported. `REAL*8` is interpreted as DP; `COMPLEX*16` is interpreted as DC.
7. `.T.` and `.F.` may be used in place of the logical constants `.TRUE.` and `.FALSE.`.
8. `DO-ENDDO` constructs are permitted.
9. Recursive subroutine calls are permitted.

3.12 Error Checking

More than 100 error conditions are checked by the translator program, and if any of these is encountered, an error message is output, together with the line number of the statement in the input file where the error was detected. An attempt has been made to cover all of the prohibited situations mentioned in this paper, as well as many violations of the standard rules of Fortran. In some cases, certain possible Fortran errors are not checked by the translator, because if they do occur, they will certainly be trapped when an attempt is made to compile the output program.

One example of an error condition that is checked by the translator is any type mismatch between the argument list of a reference to a subroutine or function and its definition (provided both are in the same file). Such errors can easily occur when, for example, a DP constant is used as an argument, but the defining subprogram expects a MPR value. These errors can also occur if the name of an MPR function is not declared to be of type MPR in the subprograms where it is referenced.

Although this is certainly not a recommended programming practice, type mismatches between argument lists do exist in some working Fortran programs. For example, some codes pass a scratch array of type real to a subroutine when a complex scratch array is expected. Because in some cases it may be difficult to remove type mismatches from an existing code, and since the resulting code may work correctly anyway, a provision has been made for the translator to toggle type error trapping on and off. This is done by inserting one of the following directives in the declaration section of any subprogram:

```
CMP+ TYPE ERROR ON
CMP+ TYPE ERROR OFF
```

It remains in effect until the end of file or until another such directive is encountered. When type error trapping is disabled with the `OFF` option, a non-fatal warning message is included in the output file for the programmer's information.

3.13 Performance of Translated Code

A number of fairly large programs have been successfully translated with this program. These include the Linpack benchmark [18], both a real and a complex FFT benchmark [2], a vortex analysis code [24], a Feigenbaum number calculation [13], implementations of Ferguson's PSOS and PSLQ integer relation algorithms [5, 20], and an implementation of the RSA public-key cryptosystem [27]. All appear to work correctly.

In most cases where the author had previously coded the application by hand using the MPFUN routines, the performance of the translated code (using the `FAST` option) is not significantly different. Thus it appears that in most cases there will not be a performance penalty for using the translator. Partly this is due to the fact that in translating arithmetic expressions, the translator program separately handles each of the many mixed mode cases, as opposed to merely handling all cases in a stock fashion.

However, users should be prepared for a substantial slowdown, compared with conventional IN, SP or DP code. See the performance results in section 2.6 for details.

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.

References

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

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

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

[4] D. H. Bailey, "A Portable High Performance Multiprecision Package", Technical Report RNR-90-022, NASA Ames Research Center, 1990.

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

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

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

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

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

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

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

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

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

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

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

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

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

[18] J. J. Dongarra, "The Linpack Benchmark: An Explanation", *SuperComputing* (Spring 1988), p. 10 - 14.

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

[20] H. R. P. Ferguson and D. H. Bailey, "A Polynomial Time, Numerically Stable Integer Relation Algorithm", Technical Report RNR-91-032, NAS Applied Research Branch, NASA Ames Research Center, Moffett Field, CA 94035, March 1992.

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

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

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

[24] R. Krasny, "Desingularization of Periodic Vortex Sheet Roll-Up", *Journal of Computational Physics*, vol. 65, no. 2 (August 1986), p. 292 - 313.

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

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

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

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

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

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

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

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