A Fortran Multiple-Precision Arithmetic Package

RICHARD P. BRENT
Australian National University

A collection of ANSI Standard Fortran subroutines for performing multiple-precision floating-point arithmetic and evaluating elementary and special functions is described. The subroutines are machine independent and the precision is arbitrary, subject to storage limitations. The design of the package is discussed, some of the algorithms are described, and test results are given.

Key Words and Phrases: arithmetic, multiple precision, extended precision, floating point, elementary function evaluation, Euler's constant, gamma function, polyalgorithm, software package, Fortran, machine-independent software, special function evaluation, Bessel functions, exponential integral, logarithmic integral, Bernoulli numbers, zeta function, portable software
CR Categories: 3 15, 4.49, 5.11, 5 12, 5 15, 5 19, 5 25
The Algorithm: MP, A Fortran Multiple-Precision Arithmetic Package. *ACM Trans. Math. Software 4*, 1 (March 1978), 71–81.

1. INTRODUCTION

MP is a collection of Fortran subroutines for performing multiple-precision floating-point arithmetic. The package is almost completely machine independent, and the consequent loss of efficiency is not excessive. MP works with t-digit normalized floating-point numbers with base b, where $t \geq 2$, $b \geq 2$, and $8b^2 - 1$ is representable as a single-precision integer. The base and number of digits may be varied dynamically.

Several multiple-precision arithmetic packages are available [1, 4, 7, 8, 18–20, 26–28, 34–36, 38, 40, 45, 47, 49, 51], but MP appears to be the only one which does not suffer from at least one of the following disadvantages: machine dependence, use of fixed-point rather than floating-point arithmetic, fixed or bounded precision, no routines for elementary and special functions (ln, exp, sin, Bessel functions, etc.) or constants (π, γ, etc.).

MP is designed for floating-point calculations. In some applications it is essential that all operations should be performed exactly, using multiple-precision integers or rational numbers. For these applications, a package which uses a linked-

Author's address: Computer Centre, Australian National University, Box 4, Canberra, ACT 2600, Australia.

list representation of variable-length multiple-precision integers is preferable to MP. The MP subroutines are intended for applications such as checking the accuracy of floating-point library routines or generating accurate constants to be used in such routines. See, for example, [43–45].

Since MP is machine independent it is necessarily inefficient at a low level. However, we have attempted to make it efficient at a high level by implementing good algorithms. Some of the algorithms are described in Section 6, and test results are given in Section 7. We chose Fortran because it is widely available and relatively efficient, although a language such as Algol 68 has some obvious advantages [45].

2. DESIGN OF THE PACKAGE

A t-digit floating-point number is represented in an integer array of dimension at least $t + 2$. The first word is used for the sign (0, +1, or −1), the second word for the exponent, and the third to $(t + 2)$th words for the normalized (base b) fraction. Such a number is called an "*mp* number" below. Zero is represented by a zero sign, with words 2 to $t + 2$ undefined. The exponent lies in $[-m, m]$, where m is set by the user, with the restriction that $4m$ is representable as a single-precision integer. If the result of an operation underflows (i.e. the exponent is less than $-m$), it is set to zero, but overflow (exponent greater than m) is treated as a fatal error.

The assumption that $8b^2-1$ is representable as an integer makes it easy to perform multiplication of *mp* numbers using single-precision integer arithmetic, but is rather wasteful of space. Without this assumption much time would be spent in packing and unpacking the digits of *mp* numbers, and it seems that time is more important than space in most applications. Routines for packing *mp* numbers into integer arrays of dimension $\lceil \frac{1}{2}(t + 2) \rceil$, and for unpacking such "compressed" numbers, are provided for use when space is critical, for example, when large arrays of multiple-precision numbers need to be stored. MP could be modified to work with compressed numbers, but execution times would be increased by about 50 percent because of the increased complexity of the lower level routines. The problem could easily be overcome if Fortran supported operations on double-length integers.

Arithmetic operations on *mp* numbers are performed by subroutine calls. Thus, instead of Z = X + Y we need to write CALL MPADD (X, Y, Z). A precompiler in the style of [27] or [52] could generate the appropriate subroutine calls.[1] Sufficient working space for the MP routines must be declared in COMMON in the main program. The parameters b, t, etc., are also transmitted to the MP routines in COMMON.

3. CAPABILITIES OF MP

The present version of MP contains 101 subroutines and four main programs. The capabilities of the subroutines include the following:

(1) conversion of integer, real, and double-precision numbers to multiple-precision format, and vice versa;

[1] A precompiler using an extension of MORTRAN2 has been written and is currently being tested.

(2) multiplication and division of *mp* numbers by small integers;
(3) addition, subtraction, multiplication, and division of *mp* numbers.
(4) powers and roots of *mp* numbers,
(5) elementary functions of *mp* numbers (log, exp, sin, tan, arcsin, arctan, sinh, cosh, tanh),
(6) some special functions and constants (Bernoulli numbers, Bessel functions of the first kind, error and complementary error functions, exponential and logarithmic integrals, Dawson's integral, gamma function, π, γ, $\zeta(n)$, etc.);
(7) fixed and floating-point decimal output and free-field decimal input of *mp* numbers;
(8) integer and fractional parts of *mp* numbers;
(9) routines for error handling, testing, and debugging;
(10) miscellaneous: comparison of *mp* numbers, storing, packing and unpacking *mp* numbers, etc.

The four main programs are designed for testing purposes and are described in Sections 7 and 8.

The list of special functions could obviously be extended. In fact, it would be useful to have arbitrary-precision subroutines for all the commonly used special functions. We invite others to contribute such subroutines.

4. DEPARTURES FROM ANSI FORTRAN

We have attempted to use ANSI Standard Fortran [3] as far as possible. The only known violation of the standard is that arrays used as arguments of MP subroutines (or in COMMON) are declared with dimension 1 in the subroutines, e.g.

```
SUBROUTINE A (X, Y)
INTEGER X(1), Y(1)
```

It is assumed that X, Y, etc., are declared with sufficiently large dimension (usually at least $t + 2$) in the main program, and that the compiler does not check whether array bounds as declared in the example shown above are violated. The reason for this deviation from the standard is that it makes it possible to increase the precision of a computation merely by changing the main program—the MP routines do not need to be changed or recompiled.

To conform to the standard, it would be necessary to pass the dimensions of the arrays X, Y, etc., as extra arguments, e.g.

```
SUBROUTINE A (X, Y, M, N)
INTEGER X(M), Y(N)
```

However, this would increase both the space and time required by the MP routines, and the nonstandard usage is accepted by most Fortran compilers (see Section 7).

5. ERROR HANDLING

There is no way that the MP routines can ensure that arrays used as arguments have been declared with sufficiently large dimensions in the main program. Thus overwriting may occur, with unpredictable results, if the user is careless. However, MP attempts to detect errors and makes it easy for the user to avoid them. For

example, there is a subroutine MPSET (LUNIT, IDECPL, ITMAX2, MAXDR) which provides a convenient way for the user to set the base, the number of digits, and other necessary parameters in COMMON before calling any other MP routines. Here LUNIT is a logical unit to be used for subsequent error messages from MP routines, and IDECPL is the "equivalent" number of floating decimal places desired by the user. MPSET determines the largest machine-representable integer of the form $2^k - 1$, sets $m = 2^{k-2} - 1$, and sets b and t to the optimal legal values such that

$$(t - 1)\log_{10} b \geq \text{IDECPL}.$$

The arguments ITMAX2 and MAXDR should equal the dimensions of arrays to be used for MP numbers and for working space, respectively. MPSET checks that $t + 2 \leq$ ITMAX2, and informs the user if ITMAX2 is too small. MAXDR is saved in COMMON, and if the user subsequently calls a routine which needs more space, he is informed of this.

The MP routines check for various error conditions. For example, if an integer which should be positive becomes negative, then it is probable that integer overflow has occurred because b is too large, and an informative message is printed. If the sign of an *mp* number is not 0 or ± 1, or a digit is not in the range $0, 1, \ldots, b-1$, overwriting has probably occurred, and the user is informed. Routines such as MPSIN and MPPI convert their result to (single precision) real and check that it is reasonable, and routines which use Newton's method check that the iteration is converging as it should.

If an MP routine is about to generate a multiple-precision result X whose exponent lies outside the allowable range, then MPUNFL(X) or MPOVFL(X) is called. At present MPUNFL sets X to zero and returns, while MPOVFL prints an error message, but these actions could easily be modified. For example, it might be desirable to terminate execution after a certain number of multiple-precision underflows, since these are probably caused by a programming error if the allowable exponent range is large.

To make the MP routines as intelligible and easy to modify as possible, we have followed most of the suggestions of Kernighan and Plauger [30].

6. SOME ALGORITHMS USED IN MP

6.1 Addition

This is straightforward, and is performed much as described in Knuth [32]. We use R^*-rounding [15, 33] after postnormalization, with four guard digits.

6.2 Multiplication

The classical $O(t^2)$ method is used because it seems difficult to implement faster algorithms [29, 32, 46] in a machine-independent manner in Fortran. Also, the faster algorithms would actually be slower for small t. The subroutine MPMUL could be modified without affecting the other routines in MP. We denote the time required for t-digit multiplication by $M(t)$, so $M(t)$ is of order t^2 with our implementation of multiplication, but $M(t) = O(t \cdot \log(t) \log\log(t))$ is theoretically attainable. Multiplication (or division) of an *mp* number by a single-precision integer is an important special case which requires time $O(t)$.

6.3 Reciprocals, Square Roots, Etc.

It is well known that, if x is a positive number and n is a positive integer, then $y = x^{-1/n}$ may be computed by Newton's method without using any divisions except by n. The iteration is

$$y_{j+1} = y_j + y_j(1 - xy_j^n)/n.$$

MPROOT implements this iteration, starting with a small value of t and approximately doubling it at each step, as described in [14, 17]. Thus the time required by MPROOT is $O(M(t))$. MPREC implements the special case $n = 1$. Division and square roots require one additional multiplication after the computation of x^{-1} or $x^{-1/2}$, respectively, so they also require time $O(M(t))$.

Numbers of the form $(i/j)^{p/q}$, where i, j, p, and q are small integers, often occur in multiple-precision calculations. MPQPWR uses MPROOT to compute such numbers efficiently.

6.4 exp(x)

MPEXP1 evaluates $\exp(x) - 1$ for small $|x|$ by an algorithm described in [17]. The idea is to use the power series for $\exp(x) - 1$ if $|x|$ is sufficiently small, and otherwise to use the relation

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

to reduce $|x|$. The time required is $O(t^{1/2}M(t))$. Methods which require time $O(\log(t)M(t))$ are known [11, 14], but turn out to be slower for small and moderate t.

MPEXP uses the identity $\exp(x) = e^n \exp(f)$, where n is the integer part of x, and $f = x - n$, to evaluate $\exp(x)$ (using MPEXP1 to evaluate $\exp(f)$ and the well-known series for e or e^{-1}). The hyperbolic functions are easily evaluated using MPEXP (or sometimes MPEXP1 to avoid cancellation).

6.5 ln(x)

MPLNS evaluates $\ln(1 + x)$ for small $|x|$ by Newton's method, using MPEXP1. Thus the time required is $O(t^{1/2}M(t))$. Asymptotically faster methods are known [14], but are practically useful only if t is very large. MPLNGS implements one such method (useful mainly for testing purposes).

MPLN evaluates $\ln(x)$ by first obtaining a rough approximation y, evaluating $\exp(y)$ accurately using MPEXP, and then evaluating $\ln(x/\exp(y))$ accurately using MPLNS.

MPLNI computes $\ln(n)$ for small integer n and is faster than MPLN. MPLNI calls MPL235, which uses well-known series for $\ln(16/15)$, $\ln(25/24)$, and $\ln(81/80)$ to compute $\ln(m)$, where m is an integer of the form $2^i3^j5^k$. MPLNI chooses such an m close to n and then uses a series for $\ln(n/m)$ to obtain $\ln(n)$. MPL235 and MPLNI require time $O(t^2)$.

6.6 sin(x) and tan(x)

The function $\sin(x)$ is evaluated from the Taylor series if $|x| < 1$, so the time required is $O(tM(t)/\log(t))$. This could be reduced to $O(t^{1/2}M(t))$ or even

$O(\log(t)M(t))$, as described in [11, 14, 17], but the simpler algorithm is faster for small t. If $|x| \geq 1$, various identities are used to reduce $|x|$.

The function $\tan(x)$ is evaluated from the identity $\tan(x) = \sin(x)[1 - \sin^2(x)]^{-1/2}$ if $|x| < \pi/4$ and from similar identities if $|x| \geq \pi/4$.

6.7 arctan(x) and arcsin(x)

MPATAN evaluates $\arctan(x)$ from the Taylor series if $|x| < \frac{1}{2}$, so the time required is $O(tM(t))$. (This could be reduced for large t, as for $\sin(x)$.) If $|x| \geq \frac{1}{2}$, the identity $\arctan(x) = 2\arctan\{x/[1 + (1 + x^2)^{1/2}]\}$ is used to reduce $|x|$. The asymptotically faster method described in [11] has been implemented for testing purposes, but is not included in the MP package, as it is competitive with MPATAN only for very large t.

MPASIN evaluates $\arcsin(x)$ from the identity $\arcsin(x) = \arctan\{x[1 - x^2]^{-1/2}\}$ if $|x| < 1$.

6.8 Evaluation of π

MPPI uses Machin's well-known identity $\pi/4 = 4\arctan(1/5) - \arctan(1/239)$, where $\arctan(1/5)$ and $\arctan(1/239)$ are obtained from the Taylor series (not using MPATAN). Thus the time required is $O(t^2)$.

MPPIGL uses the Gauss-Legendre algorithm [11, 14, 42], which requires time $O(\log(t)M(t))$. Since our implementation of MPMUL has $M(t)$ of order t^2, MPPIGL is always slower than MPPI, but it would be faster (for large t) if an algorithm faster than $O(t^2/\log(t))$ were used in MPMUL. Subroutine MPPIGL may be used to test MPPI and also, indirectly, MPSQRT and MPDIV.

6.9 Evaluation of γ

Euler's constant $\gamma = 0.5772\ldots$ is usually defined by

$$\gamma = \lim_{n\to\infty}\left(\sum_{i=1}^{n} 1/i - \ln(n)\right).$$

We may use this definition to evaluate γ, estimating the remainder after a finite n by the asymptotic Euler-Maclaurin series; see, for example, [31]. However, this method requires the Bernoulli numbers which appear as coefficients in the Euler-Maclaurin series, and the number of Bernoulli numbers required increases with t if we demand reasonable efficiency (e.g. time polynomial in t). Thus MPEUL uses a method which does not require any Bernoulli numbers. We outline the method here because similar ideas may be used for other multiple-precision calculations, for example, in the computation of $\Gamma(x)$ (see Section 6.10).

The method was suggested (though not used) by Sweeney [48] and depends on the identity

$$\gamma = S(n) - R(n) - \ln(n),$$

where

$$S(n) = \sum_{k=1}^{\infty} \frac{n^k(-1)^{k-1}}{k!\,k},$$

$$R(n) = \int_n^{\infty} \frac{\exp(-u)}{u}\,du \sim \frac{\exp(-n)}{n}\sum_{k=0}^{\infty}\frac{k!}{(-n)^k},$$

and n is chosen sufficiently large. Using Stirling's approximation, we have

$$\left| R(n) - \frac{\exp(-n)}{n} \sum_{k=0}^{n=2} \frac{k!}{(-n)^k} \right| = O(e^{-2n}),$$

and

$$\left| S(n) - \sum_{k=1}^{\lfloor \alpha n \rfloor} \frac{n^k(-1)^{k-1}}{k!\,k} \right| = O(e^{-2n}),$$

where $\alpha = 4.3191\ldots$ is the positive root of $\alpha + 2 = \alpha \ln \alpha$. Thus, to obtain γ with error $O(b^{-t})$, we need to take $n \sim \frac{1}{2}t \ln(b)$, and the number of operations is $O(t^2)$. This could be reduced to $O(\log^2(t)M(t))$ by grouping the terms in the above series in pairs, etc. (as for the computation of e in [17],) but this is not worthwhile unless t is large and a fast multiplication algorithm is used.

When evaluating the series for $S(n)$, the largest term (in absolute value) corresponds to $k = n - 2$, and $n^{n-2}/[(n-2)!(n-2)] < \exp(n)$. Thus to compensate for cancellation when summing the series it is sufficient to work with approximately $3t/2$ digits. When summing the asymptotic series for $R(n)$, only $t/2$ floating-point digits are necessary.

6.10 Evaluation of $\Gamma(x)$

If $0 < x < 1$ we have

$$\Gamma(x) = \int_0^n u^{x-1}e^{-u}\,du + O(e^{-n})$$

$$= \sum_{k=0}^{\infty} \frac{(-1)^k n^{k+x}}{(k+x)\cdot k!} + O(e^{-n}).$$

The series can be summed in much the same way as the series $S(n)$ above. This avoids the need for Bernoulli numbers, which are required if the asymptotic series for $\ln(\Gamma(j+x))$ is used. Taking $n \sim t\cdot\ln(b)$, it is necessary to work with approximately $2t$ digits to compensate for cancellation. This could be reduced to $3t/2$ digits if the remainder were approximated by an asymptotic series, as in the computation of γ, with $n \sim \frac{1}{2}t\cdot\ln(b)$.

The summation of the above series may be performed more rapidly if x is a rational number, for then a multiple-precision division is not required to evaluate each term. MPGAMQ implements this method to evaluate $\Gamma(p/q)$ for small integers p and q. The argument is reduced to $(0, 1)$ using well-known identities, and the special cases $q = 1$ and $q = 2$ are dealt with separately.

MPLNGM evaluates $\ln(\Gamma(x))$ using Stirling's approximation. The number of terms used in the asymptotic series for the remainder is not fixed, but depends on the precision required. Thus a variable number of Bernoulli numbers have to be generated using MPBERN (see Section 6.11). Actually, $\ln(\Gamma(j+x))$ is computed, where j is an integer chosen to approximately minimize the total computation time, and then $\ln(\Gamma(x))$ is deduced using the well-known recurrence formula. The time required is $O(t^3)$.

MPGAM evaluates $\Gamma(x)$ using MPGAMQ if x is a suitable rational number, or MPLNGM and MPEXP otherwise (combined with the reflection formula if x

is negative). The time and space required in the worst case are considerably greater than for MPGAMQ.

6.11 Bernoulli Numbers

For many multiple-precision computations it is necessary to estimate a remainder using the Euler-Maclaurin formula [2]. To obtain n terms in the asymptotic series for the remainder, we need to know the Bernoulli numbers $B_2, B_4, \ldots, B_{2n}$. MPBERN generates $C_1, \ldots, C_n$ first, where $C_k = B_{2k}/(2k)!$, using the recurrence

$$2C_k(1 - 4^{-k}) + \frac{C_{k-1}}{2!4} + \frac{C_{k-2}}{4!4^2} + \cdots + \frac{C_1}{(2k-2)!4^{k-1}} = \frac{2k-1}{(2k)!4^k}.$$

The relative error in the computed C_k is $O(k^2b^{1-t})$.

Note that we avoid the well-known recurrence [31]

$$\frac{C_k}{1!} + \frac{C_{k-1}}{3!} + \cdots + \frac{C_1}{(2k-1)!} = \frac{k - \frac{1}{2}}{(2k+1)!}$$

because it is numerically unstable: using it in the forward direction would give a relative error $O(4^kb^{1-t})$ in the computed C_k. (This might be unimportant if the terms in the asymptotic series decreased fast enough.) The time required by MPBERN is $O(n^2t + nM(t))$.

6.12 Exponential and Logarithmic Integrals

MPEI computes

$$ei(x) = \int_{-\infty}^{x} (e^u/u)\, du, \qquad x \neq 0,$$

and MPLI computes

$$li(x) = \int_0^x \frac{du}{\ln(u)} = ei(\ln(x)), \qquad x \geq 0,\ x = 1,$$

where both integrals are Cauchy principal value integrals. MPEI uses the asymptotic series if $|x| > t \cdot \ln(b)$, and otherwise uses the power series

$$ei(x) = \gamma + \ln|x| + \sum_{k=1}^{\infty} \frac{x^k}{k!k}.$$

If x is negative there is some cancellation in summing the power series, so the working precision is increased to compensate for this. In the worst case, when $x = -t \cdot \ln(b)$, approximately $3t$ digits have to be used, and the time required is $O(tM(t))$.

6.13 Error Function

MPERF computes the error function $\operatorname{erf}(x) = (2/\pi^{1/2}) \int_0^x \exp(-u^2)\, du$, and MPERFC computes the complementary error function $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$. The methods used are similar to those described for the exponential integral (Section 6.12). If $|x|$ is sufficiently large, the asymptotic series is used; otherwise the power series for $\exp(x^2) \int_0^x \exp(-u^2)\, du$ is used. With this series there is no cancellation, so it is not necessary to increase the working precision in MPERF.

It is, however, necessary to increase the working precision in MPERFC if x is positive but not large enough for the asymptotic series to be used.

6.14 Bessel Functions

MPBESJ computes the Bessel function $J_\nu(x)$ for integer ν and multiple precision x. If x is sufficiently large, Hankel's asymptotic expansion is used [2, sect. 9.2.5]; otherwise, either the power series [2, sect. 9.1.10] or the backward recurrence method [22] is used, depending upon how much cancellation occurs with the power series. The time required is $O(tM(t))$.

7. TEST RESULTS

Most testing has been done using Fortran V on a Univac 1108. Gross errors may be detected by comparing results of multiple-precision calculations with the results of the corresponding calculations performed in single or double precision. Many internal consistency checks are possible. For example, we should have $(x^2)^{1/2} = |x|$, $\sin(\arcsin(x)) = x$, etc. MPREC, MPROOT, MPSQRT, MPDIV, MPQPWR, MPSIN, MPASIN, MPTAN, MPATAN, MPCOSH, etc., were checked in this way for several different choices of b and t (e.g. $b = 10$ and $t = 16$). Because MPLNS uses Newton's method and MPEXP1, a check that the computed value of $\ln(\exp(x))$ is close to x does not necessarily imply that MPLN is correct. However, MPLN was tested by comparison with MPLNI and MPLNGS, and then MPEXP was checked using MPLN. MPATAN was checked using an implementation of the method described in [11], MPPI was checked using MPPIGL as well as published values of π, and MPEUL was checked using published values of γ [31, 48]. The testing of MPEUL actually revealed an error in the most accurate published value of γ [5, 6]. MPGAMQ was checked using published values [21] as well as various identities. MPGAM and MPLNGM were tested using MPGAMQ and various identities. MPEI, MPERF, MPERFC, and MPBESJ were tested for sufficiently large arguments by using both the asymptotic series and the power series methods. MPBESJ was also tested for small arguments by using both the power series and the backward recurrence methods.

All routines have been tested with several different choices of b and t. Our aim was not to provide routines which always give t correctly rounded digits; there is no need for this because t may easily be increased if necessary. Generally the error is bounded by a few units in the $(t - 1)$th digit for moderate arguments and sensible choices of b and t. There are some exceptions, e.g. $\sin(x)$ for x near π. However, in all cases the computed value y of $f(x)$ satisfies $y = (1 + \epsilon_1)f(x(1 + \epsilon_2))$, where ϵ_1 and ϵ_2 are perturbations of order b^{1-t}. If the user wishes to be confident of the accuracy of his results, he should compare them with single- or double-precision results to detect gross errors and then use at least two different values of t (and/or b) to estimate the accuracy of the multiple-precision results.

The main program TEST uses MP to evaluate the 33 constants given to 40 decimal places in [32, appendix B]. TEST calls MPSET to set the base and number of digits to give the equivalent of at least 42 decimal places. Thus b and t depend on the wordlength of the machine. For example, on a Univac 1108 or PDP 10 with

Table I. Runs of TEST Program

Machine	Compiler	Effective wordlength (bits)	Approximate execution time (seconds)	Space (1024 words)[17]
PDP 11/45[1]	Fortran[2]	16	115.2	24
IBM 360/50	Fortran G[3]	32	31.5	21
IBM 360/50	Fortran H[4]	32	22.6	18
IBM 360/91	Fortran H[5]	32	1.63	21
IBM 370/168	Fortran H[6]	32	1.24	21
PDP 10[7]	F40[8]	36	14.1	10[10]
PDP 10[7]	F10[9]	36	11.0	10[10]
Univac 1108	Fortran V[11]	36	4.42	17
Univac 1100/42	Fortran V[12]	36	3.39	12[13]
Univac 1100/42	Ascii Fortran[14]	36	3.28	13[13]
Cyber 76	Fortran 4.2[15]	48[16]	0.43	6

[1] No floating-point hardware, 28K words memory
[2] Fortran V004A,/ER,/ON,/SU under DOS.
[3] Fortran IV G (level 19) under OS/MFT.
[4] Fortran H (level 20.1), opt = 2, under OS/MFT.
[5] Fortran H extended (level 2.1), opt = 2, under OS/MVT release 21 8
[6] Fortran H extended (level 2 1), opt = 2, under OS/VS2 release 1 6.
[7] KA 10 processor.
[8] F40 V27(360)
[9] Fortran V.4A(317),/KA,/NOOPT (Caution: /OPT does not work)
[10] Low segment only. High segment 7K words.
[11] Fortran V (FOR SE1D) under Exec 8 (level 31).
[12] Fortran V (FOR 00T3), 1110 = opt, under Exec 8 (level 32R2A).
[13] Excluding common banks
[14] Ascii Fortran (FTN 6R1), XZ, under Exec 8 (level 32R2B).
[15] Fortran 4 2(+383), opt = 1, under Scope 2.1 (level 185).
[16] Actual wordlength, 60 bits, but effective wordlength for integer arithmetic only 48 bits. Similarly, on Burroughs B6700 the effective wordlength is only 37 bits (and MPSET requires a trivial modification).
[17] Excluding buffers, device drivers, and other system requirements.

a 36-bit word, $b = 65536$, $t = 10$, and the precision is sufficient to give correctly rounded 40D results. TEST calculates $\zeta(3)$ from the rapidly converging series of Gosper [23]:

$$\zeta(3) = \frac{5}{4}\sum_{k=0}^{\infty}\frac{(-1)^k(k!)^2}{(k+1)^2(2k+1)!}.$$

The other constants each require only a few calls to MP routines for their computation.

TEST has been run successfully on various machines with several different wordlengths. Details are given in Table I.

The program TESTV is a slight modification of TEST to allow variable precision, For checking purposes, 1000D results are available [12]. Table II shows how the execution time depends on the precision required.[2]

[2] Some of the results were checked using the package of Wyatt et al. [50]. MP was significantly faster: the ratio of execution times ranged from 1.49 (for 20-place accuracy) to 10.1 (for 200-place accuracy).

Table II. Execution Times of TESTV Program on Univac 1108

Decimal places requested in call to MPSET	Execution time (seconds)
1	0.8
10	1.6
20	2.4
30	3.4
40	4.4
50	5.9
60	7.0
80	10.3
100	14.0
200	42.1
400	161.5
1000	1065.6

The program TEST2 tests the MP routines more thoroughly than does TEST. It computes the constants given in [24, appendix C] and various other constants to 40 significant figures. On a Univac 1108, working with two extra decimal places (i.e. IDECPL = 42 in the call to MPSET), the results are accurate to within half a unit of the fortieth place. Correct 40S results are given in the comments in the program. As for TEST, it is easy to increase the precision of TEST2 if required. TEST2 has been run on several different machines.

Knuth's computation of the continued fraction for γ has been verified and extended using MP. Details will be published separately [10], but it is worth noting that only 38 seconds of U1108 time were needed to compute and verify the 372 quotients given by Knuth [31].

MP has also been used successfully to verify the orders of certain root-finding methods and to determine their asymptotic error constants [16].

8. AN EXAMPLE

The main program, EXAMPLE, given in [9, 13], is intended to give a simple example of the use of MP. It computes π and $\exp(\pi(163)^{1/2}/3)$ and prints them to 100 decimal places. The correct output is

$$\pi = 3.14159265358979 \ldots 1170680$$

and

$$\exp(\pi(163)^{1/2}/3) = 640320.0000000060486 \ldots 6477590.$$

EXAMPLE also computes

$$\exp(\pi(163)^{1/2}) = 262537412640768743.99999999999925007 \ldots$$

and prints it to 90 decimal places. See [25, 37, 39] for the reason these numbers are interesting. EXAMPLE has been run successfully on the same machines and with the same compilers as TEST (see Section 7).

ACKNOWLEDGMENTS

I would like to thank D. Bowers, J.T. Chmura, W.L. Edwards, C.A. Freyberg, R.H. Gonter, R.J. Hurle, H. Kung, M.W. Ray, and M. Saunders for their assistance in testing MP on various machines, and J.P. Abbott, J.R. Ehrman, M. Ginsberg, W.M. Gentleman, and R. Spira for their useful comments. One of the referees kindly ran MP through the PFORT verifier [41].

REFERENCES

1. ABERTH, O. A precise numerical analysis program. *Comm. ACM 17*, 9 (Sept. 1974), 509–513.
2. ABRAMOWITZ, M , AND STEGUN, I.A. *Handbook of Mathematical Functions*. Nat. Bur. of Standards, Washington, D.C., 1964.
3. American National Standard Fortran (ANSI X3.9-1966), Amer. Nat. Standards Inst., New York, 1966. See also *Comm. ACM 12*, 5 (May 1969), 289–294 and *Comm. ACM 14*, 10 (Oct. 1971), 628–642.
4. BAER, R M., AND REDLICH, M.G. Multiple-precision arithmetic and the exact calculation of the 3-*j*, 6-*j*, and 9-*j* symbols. *Comm. ACM 7*, 11 (Nov. 1964), 657–659.
5. BEYER, W.A., AND WATERMAN, M.S. Decimals and partial quotients of Euler's constant and ln(2). Submitted to *Math. Comput.*
6. BEYER, W.A., AND WATERMAN, M S. Error analysis of a computation of Euler's constant. *Math. Comput. 28* (April 1974), 599–604.
7. BLUM, B.I. An extended arithmetic package. *Comm. ACM 8*, (May 1965), 318–320.
8. BOGEN, R. MACSYMA Reference Manual, Ver 8. Mathlab. Group, Project MAC, M.I.T., Cambridge, Mass., 1975.
9. BRENT, R.P. Algorithm 524. MP, A Fortran multiple-precision arithmetic package. *ACM Trans. Math. Software 4*, 1 (March 1978), 71–81.
10. BRENT, R P. Computation of the continued fraction for Euler's constant. *Math. Comput. 31* (July 1977), 771–777.
11. BRENT, R.P. Fast multiple-precision evaluation of elementary functions. *J. ACM 23*, 2 (April 1976), 242–251.
12. BRENT, R.P. Knuth's constants to 1000 decimal and 1100 octal places. Tech. Rep. 47, Comptr. Ctr., Australian National U., Canberra, Sept. 1975.
13. BRENT, R.P. MP users guide Tech Rep. 54, Comptr. Ctr., Australian National U., Canberra, Sept. 1976.
14. BRENT, R.P. Multiple-precision zero-finding methods and the complexity of elementary function evaluation. In *Analytic Computational Complexity*, J.F. Traub, Ed., Academic Press, New York, 1976, pp. 151–176.
15. BRENT, R.P. On the precision attainable with various floating-point number systems. *IEEE Trans. Comptrs. C-22* (June 1973), 601–607.
16. BRENT, R.P. Some high-order zero-finding methods using almost orthogonal polynomials. *J. Austral Math. Soc. Series B, 19* (1975), 1–29.
17. BRENT, R.P. The complexity of multiple-precision arithmetic. In *Complexity of Computational Problem Solving*, R S. Anderssen and R.P. Brent, Eds., U. of Queensland Press, Brisbane, 1976, pp. 126–165.
18. COLLINS, G E. PM, A system for polynomial manipulation. *Comm. ACM 9*, 8 (Aug. 1966), 578–589.
19. DEKKER, T.J. A floating-point technique for extending the available precision. *Numer. Math. 18* (1971), 224–242.
20 EHRMAN, J.R. A multiple-precision floating-point arithmetic package for System/360. Rep. CGTM 18, Stanford Linear Accelerator Ctr., Stanford, Calif., 1967.
21. GALANT, D. C., AND BYRD, P.F. High accuracy gamma function values for some rational arguments. *Math Comput. 22* (1968), 885–887
22. GAUTSCHI, W Algorithm 236. Bessel functions of the first kind. *Comm. ACM 7*, 8 (Aug. 1964), 479–480.

23. GOSPER, R.W Acceleration of series. Memo 304, AI Lab , M.I.T., Cambridge, Mass., March 1974.
24. HART, J.F., ET AL. *Computer Approximations.* Wiley, New York, 1968.
25. HERMITE, C. *Oeuvres de Charles Hermite, Vol. 2.* Gauthier-Villars, Paris, 1908, pp. 38–82.
26. HILL, I D. Algorithm 34, Procedures for the basic arithmetical operations in multiple-length working. *Computer J 11* (Aug. 1968), 232–235.
27. HULL, T.E., AND HOFBAUER, J J. Language facilities for multiple-precision floating-point computation. Dept. Comptr. Sci , U. of Toronto, Toronto, Ont., 1974.
28. JONES, H.S.P. Algorithm 72. Multiple integer arithmetic procedures in Algol. *Computer J. 15* (1972), 281–282.
29. KARATSUBA, A., AND OFMAN, Y. Multiplication of multidigit numbers on automata. *Dokl. Akad. Nauk SSSR 146* (1962), 293–394 (in Russian).
30. KERNIGHAN, B.W , AND PLAUGER, P.J. *The Elements of Programming Style.* McGraw-Hill, New York, 1974.
31. KNUTH, D.E. Euler's constant to 1271 places. *Math. Comput. 16* (1962), 275–281.
32. KNUTH, D.E. *The Art of Computer Programming, Vol 2· Seminumerical Algorithms* Addison Wesley, Reading, Mass , 1969
33 KUKI, H , AND CODY, W.J. A statistical study of the accuracy of floating-point number systems *Comm. ACM 16,* 4 (April 1973), 223–230.
34. LAWSON, C L Basic *Q*-precision arithmetic subroutines including input and output. Tech Memo 170, Jet Propulsion Lab., Pasadena, Calif , Oct. 1967.
35. LAWSON, C.L. *Q*-precision subroutines for the elementary functions and aids for testing single-precision and double-precision function subroutines. Tech. Memo 188, Jet Propulsion Lab , Pasadena, Calif., April 1968.
36. LAWSON, C L Summary of *Q*-precision subroutines as revised in October 1968. Tech Memo 211, Jet Propulsion Lab , Pasadena, Calif., Jan 1969.
37 LEHMER, D.H. Tables to many places of decimals. *Math. Tables Aids Comput. 1* (1943), 30–31 (now *Math Comput.*).
38. MAXIMON, L.C Fortran programs for arbitrary precision arithmetic. Rep. 10563, Nat. Bur. of Standards, Washington, D.C., April 1971.
39. RAMANUJAN, S *Collected Papers of Srinivasa Ramanujan* Cambridge U. Press, Cambridge, 1927, pp 23–39.
40. REID, C E , AND KNOBLE, H D. A multiple precision arithmetic package for the IBM 360/370 systems. SHARE Program Library, March 1974.
41. RYDER, B.G The PFORT verifier *Software—Practice and Experience 4* (1974), 359–377
42 SALAMIN, E. Computation of π using arithmetic-geometric mean. *Math. Comput. 30* (July 1976), 565–570.
43 SCHONFELDER, J.L. The production of special function routines for a multi-machine library. *Software—Practice and Experience 6* (1976), 71–82.
44. SCHONFELDER, J L The testing of mathematical function software in a multi-machine environment. Tech. Rep. 107, Basser Dept Comptr. Sci., U. of Sydney, Sydney, Australia, Nov. 1975.
45 SCHONFELDER, J.L., AND THOMASON, J.T. Applications support by direct language extension—an arbitrary precision arithmetic facility in Algol 68 Computer Ctr., U. of Birmingham, Birmingham, 1975.
46. SCHONHAGE, A , AND STRASSEN, V Schnelle Multiplikation grosser Zahlen. *Computing 7* (1971), 281–292.
47. SPIRA, R. Fortran multiple precision, Pt. 1, 2. Dept. of Math., Michigan State U., East Lansing, Mich , 1973.
48. SWEENEY, D.W. On the computation of Euler's constant. *Math. Comput. 17* (1963), 170–178.
49. TIENARI, M., AND SUOKONAUTIO, V A set of procedures making real arithmetic of unlimited accuracy possible within Algol 60. *BIT 6* (1966), 322–338.
50. WYATT, W.T., LOZIER, D.W., AND ORSER, D.J. A portable extended precision arithmetic package and library with Fortran precompiler. Nat Bur. of Standards, Washington, D C , 1975; *Trans. Math Software 2,* 3 (Sept. 1976), 209–231.

51. CRARY, F.D. Multiple precision arithmetic design with an implementation on the Univac 1108. Tech. Summary Rep. 1123, Mathematics Research Center, U. of Wisconsin, Madison, Wis., 1971.
52 CRARY, F.D. The Augment precompiler, Pt. I—User information. Tech. Summary Rep. 1469, Mathematics Research Center, U of Wisconsin, Madison, Wis. 1974 (revised April 1976)

Received July 1975; revised October 1975, March 1976, and October 1976