A Fortran-90 Based Multiprecision System

David H. Bailey

RNR Technical Report RNR-94-013
June 6, 1994

Abstract

The author has developed a new version of his Fortran multiprecision computation system that is based on the Fortran-90 language. With this new approach, a translator program is not required — translation of Fortran code for multiprecision is accomplished by merely utilizing advanced features of Fortran-90, such as derived data types and operator extensions. This approach results in more reliable translation and also permits programmers of multiprecision applications to utilize the full power of the Fortran-90 language.

Three multiprecision datatypes are supported in this system: multiprecision integer, real and complex. All the usual Fortran conventions for mixed mode operations are supported, and many of the Fortran intrinsics, such as `SIN, EXP` and `MOD`, are supported with multiprecision arguments.

This paper also briefly describes an interesting application of this software, wherein new number-theoretic identities have been discovered by means of multiprecision computations.

Author's address: NAS Applied Research Branch, NASA Ames Research Center, Moffett Field, CA 94035-1000; `dbailey@nas.nasa.gov`.

1. Introduction

Readers may be familiar with the author's previous multiprecision system [4], which consists of the TRANSMP translator program and the MPFUN package of multiprecision (MP) computation routines. Together they permit one to write straightforward Fortran-77 code that can be executed using an arbitrarily high level of numeric precision.

From its inception, the TRANSMP program was intended only as an interim tool until Fortran-90 was available. This is because advanced Fortran-90 features such as derived data types and operator extensions permit one to implement multiprecision translation in a much more natural way. Now that day has arrived — Fortran-90 is currently available on several computer systems, and it soon will be available from all major vendors of scientific computers. Accordingly, the author has written a set of Fortran-90 modules that permit the user to handle MP data like any other Fortran data type.

With the new Fortran-90 based system, one declares variables to be of type MP integer, MP real or MP complex using Fortran-90 type statements. With a few exceptions, one can then write ordinary Fortran-90 code involving these variables. In particular, arithmetic operations involving these variables are performed with a numeric precision level that can be set to an arbitrarily high level. Also, most of the Fortran intrinsic functions, such as `SIN`, `EXP` and `MOD`, are defined with MP arguments.

In comparison to the TRANSMP approach, there are a few disappointments. To begin with, one has to give up the ability to run MP source code, without change, as a standard single precision or double precision program. Also, features such as read/write statements are not as elegant in the new system — subroutines must now be called for formatted MP read and write.

On the other hand, features such as generic functions work much better in the Fortran-90 version. Also, the coverage of Fortran features is more complete with the Fortran-90 version than with TRANSMP — programmers can now utilize the full power of the Fortran-90 language in a MP application. Another important advantage of the Fortran-90 approach is that a very reliable translation is produced, since the process of translation is performed by the Fortran-90 compiler itself, rather than by the TRANSMP program.

This article gives an overview of this new software, including a brief summary of the instructions for usage. It also describes an interesting application of this software to mathematical number theory, showing how MP calculations can be used to discover new mathematical identities.

This software is available by sending electronic mail to `mp-request@nas.nasa.gov`. Include `send index` as either the subject line or the text of the first message to this address. It is also available by using Mosaic software at the address `http://www.nas.nasa.gov/RNR/software.html`.

2. The Fortran-90 MP Translation Modules

The new MP translator is a set of Fortran-90 modules. These translation modules serve as a link between the user's program and MPFUN, the library of MP computation routines. To utilize the MP translation facility, one inserts the following line in the main program of the user's application code, as well as in any subprogram that performs MP operations:

```
USE MPMODULE
```

This line must be placed after the `PROGRAM`, `SUBROUTINE`, `FUNCTION` or `MODULE` statement, but before any intrinsic or type statement. This `USE` statement connects the subprogram with the Fortran-90 translation modules that define the MP datatypes and operator extensions.

At the beginning of the executable portion of the user's main program, even if the main program itself performs no MP operations, one inserts the line

```
CALL MPINIT
```

The routine `MPINIT` sets parameters in the MPFUN library and precomputes π, $\log(2)$ and $\log(10)$, which are needed in a number of transcendental function routines.

Three derived types are defined in the translation modules: `MP_INTEGER`, `MP_REAL` and `MP_COMPLEX`. In an application program, one may explicitly specify MP variables using Fortran-90 type statements, such as

```
TYPE (MP_INTEGER) IA, IB, IC
TYPE (MP_REAL) A, B, C, D, E
TYPE (MP_COMPLEX) (Z)
```

Alternatively, one may implicitly declare variables to be of one of the three MP types by using an `IMPLICIT` statement, such as

```
IMPLICIT TYPE (MP_REAL) (A-H, O-Z)
```

MP constants are handled a bit differently than with TRANSMP. These are now specified as literal constants, i.e. `'1.23456789'`. One may directly assign a MP constant to a MP variable, but if a MP constant appears in an expression, it must be as the argument to `MPINT`, `MPREAL` or `MPCMPL`, depending on whether it to be treated as MP integer, MP real or MP complex. Examples:

```
IA = '3333333333333333333333333333'
A = '1.4142135623 7309504880 168872420 E-10'
B = MPREAL ('1.25') / N
Z = 2 * MPCMPL ('1.2345', '6.7890')
```

Note that without the quotes to indicate an MP constant, the integer constant in the first line would overflow, and the floating constant in the second line would not be converted with full MP accuracy.

Quotes are not really required in the third line, since 1.25 can be converted exactly with ordinary arithmetic. However, note that simply writing `B = 1.25 / N` would not give a fully accurate result if, for example, `N` is an ordinary integer with the value 7 (although it would be fine if `N` is 8). This is because the division operation would be performed using ordinary single precision arithmetic, and the inaccurate result would then be converted to MP and stored in `B`. The usage of the function `MPREAL` in the third line insures that the division is performed with MP arithmetic. This is an example of the care one must exercise in programming to insure that intermediate calculations are performed with MP arithmetic when required. In this respect, the new Fortran-90 translation system is like the `FAST` option of the TRANSMP program.

The expressions in lines three and four are examples of mixed mode operations. Virtually all such operations are allowed, and the result is of the type that one would expect. For example, the product of a MP real variable and an integer constant is of type MP real, and the sum of a complex variable and a MP real variable is of type MP complex. The only combinations that are not currently allowed are some exponentiations involving MP complex entities — these are defined only when the exponent is an integer.

Unformatted read and write statements with MP variables in the I/O list, such as `WRITE (11) A, B`, are handled as expected. But formatted and list-directed read and write statements, i.e. `WRITE (6, *) A, B`, will not produce the expected results for MP variables. These operations must now be handled using the special subroutines `MPREAD` and `MPWRITE`. The first argument of either routine is the unit number. Arguments 2-10 are the list of MP variables to be input or output. Within a single call to either routine, the MP variables in the list must all be the same type, either MP integer, MP real or MP complex. Examples:

```
CALL MPREAD (5, IA)
CALL MPWRITE (6, A, B, C, D, E)
```

An example of the format for input or output of MP numbers is

```
10 ^ 40 x -3.14159265358979323846264338327950288841971,
```

On input, the exponent field is optional, and blanks may appear anywhere, but a comma must appear at the end of the last line of mantissa digits.

By default, only the first 56 mantissa digits of a MP number are output by `MPWRITE`, so that the output is contained on a single line. This output precision level can be changed by the user, either as a default setting or dynamically during execution (see section four).

It should be noted that the Fortran-90 translation modules generate calls to the standard arithmetic routines of the MPFUN library. If one wishes to utilize the "advanced" routines, which are intended for precision levels above 1000 digits (see section five), contact the author.

3. Multiprecision Functions

The functions `MPINT`, `MPREAL` and `MPCMPL` were mentioned in the previous section in the context of MP constants. These three functions are actually defined for all numeric argument types, ordinary and MP. The result is MP integer, MP real or MP complex, respectively, no matter what the type of the argument. Thus one may use `MPREAL (A)` to convert the ordinary floating point variable `A` to MP real.

The corresponding Fortran type conversion functions `INT, REAL, DBLE, CMPLX`, and `DCMPLX` have also been extended to accept MP arguments. The result, in accordance with Fortran language conventions, is of type default integer, real, double precision, complex and double complex, respectively. Note that `REAL (IA)`, where `IA` is MP integer, is not of type MP real — if that is required, then `MPREAL` should be used instead.

Many of the other common Fortran intrinsics have been extended to accept MP arguments, and they return true MP values as appropriate. A complete list of the Fortran intrinsic functions that have been extended to MP is given in Table 1. In this table, the abbreviations I, R, D, C, DC, MPI, MPR, MPC denote default integer, real, double precision, double complex, MP integer, MP real and MP complex, respectively.

Some additional MP functions and subroutines that users may find useful are demonstrated in the following examples. Here `N` is an ordinary integer variable, and `A, B` and `C` are MP real.

```
A = MPRANF ()
B = MPNRTF (A, N)
CALL MPCSSNF (A, B, C)
CALL MPCSSHF (A, B, C)
```

These calls which produce a pseudorandom number in $(0,1)$, the `N`-th root of `A`, both the cos and sin of `A`, and both the cosh and sinh of `A`, respectively. The above call to `MPNRTF` is equivalent to, but significantly faster than, the expression `A ** (MPREAL (1) / N)`. This is because the latter expression requires log and exp calculations, whereas `MPNRTF` uses an efficient Newton iteration scheme. Similarly, the above call to `MPCSSNF` executes faster than `B = COS (A)` and `C = SIN (A))`, although the results are the same. A similar comment applies to `MPCSSHF`.

4. Global Variables

There are a number of global variables defined in the MP translation modules and in the MPFUN package. These variables, which are listed in Table 2, can be accessed by any user subprogram that includes a `USE MPMODULE` statement. The entries in the column labeled "Dynam. change" indicates whether the values of these variables may be dynamically changed by the user during execution of the program.

The first three global variables listed in Table 2 are set by the user in `PARAMETER` statements at the beginning of the file containing the MP translation modules. `MPIPL` is the initial precision level, in digits, and is often the only parameter that needs to be changed. `MPIOU`, the initial output precision level, is ordinarily set to 56, although it may be set to as high as `MPIPL` if desired. The parameter `MPIEP`, the initial MP "epsilon" level, is typically set to `10 - MPIPL` or so.

The call to `MPINIT` at the start of the user's main program automatically sets initial values for the next six variables in the list. The final set of global variables in Table 2 are

Function Name	Arg. 1	Arg. 2	Result	Function Name	Arg. 1	Arg. 2	Result
`ABS`	MPI		MPI	`INT`	MPI		I
	MPR		MPR		MPR		I
	MPC		MPR		MPC		I
`ACOS`	MPR		MPR	`LOG`	MPR		MPR
`AIMAG`	MPC		MPR	`LOG10`	MPR		MPR
`AINT`	MPR		MPR	`MAX`	MPI	MPI	MPI
`ANINT`	MPR		MPR		MPR	MPR	MPR
`ASIN`	MPR		MPR	`MIN`	MPI	MPI	MPI
`ATAN`	MPR		MPR		MPR	MPR	MPR
`ATAN2`	MPR	MPR	MPR	`MOD`	MPI	MPI	MPI
`CMPLX`	MPI	MPI	C		MPR	MPR	MPR
	MPR	MPR	C	`NINT`	MPR		MPI
	MPC		C	`REAL`	MPI		R
`CONJG`	MPC		MPC		MPR		R
`COS`	MPR		MPR		MPC		R
`COSH`	MPR		MPR	`SIGN`	MPI	MPI	MPI
`DBLE`	MPI		D		MPR	MPR	MPR
	MPR		D	`SIN`	MPR		MPR
	MPC		D	`SINH`	MPR		MPR
`DCMPLX`	MPI	MPI	DC	`SQRT`	MPR		MPR
	MPR	MPR	DC		MPC		MPC
	MPC		DC	`TAN`	MPR		MPR
`EXP`	MPR		MPR	`TANH`	MPR		MPR

Table 1: MP Extensions of Fortran Intrinsic Functions

Variable Name	Type	Dynam. Change	Initial Value	Description
MPIPL	Integer	No	User sets	Maximum and initial precision, in digits.
MPIOU	Integer	No	User sets	Initial output precision, in digits.
MPIEP	Integer	No	User sets	$\log_{10}$ of initial MP epsilon.
MPWDS	Integer	No	$\frac{\texttt{MPIPL}}{7.22472} + 1$	Maximum and initial precision, in words.
MPOUD	Integer	Yes	MPIOU	Current output precision, in digits.
MPEPS	MP real	Yes	$10^{\texttt{MPIEP}}$	Current MP epsilon value.
MPL02	MP real	No	$\log 2$	
MPL10	MP real	No	$\log 10$	
MPPIC	MP real	No	π	
MPNW	Integer	Yes	MPWDS	Current precision level, in words.
MPIDB	Integer	Yes	0	MPFUN debug level.
MPLDB	Integer	Yes	6	Logical unit for debug output.
MPNDB	Integer	Yes	22	No. of words in debug output.
MPIER	Integer	Yes	0	MPFUN error indicator.
MPMCR	Integer	Yes	7	Cross-over point for advanced routines.
MPIRD	Integer	Yes	1	MPFUN rounding option.
MPKER	Int. array	Yes	0	Array of error options.

Table 2: Global Variables

used in the MPFUN package and assume the values as shown. See [5] for additional details on the meaning and usage of these variables.

As noted in Table 2, `MPNW` is the current numeric precision level, measured in machine words. On IEEE and most other systems, the corresponding number of digits is given by `7.22472 * (MPNW - 1)`. If one wishes to perform the same computation with a variety of precision levels without recompiling the translation modules, or if one needs to dynamically change the working precision level during the course of a calculation, this may be done by directly modifying the parameter `MPNW` in the user program, as in

```
MPNW = 127
```

But be careful not to change `MPNW` to a value larger than `MPWDS`, or else array overwrite errors may occur. The parameter `MPWDS` is defined at the beginning of the MP translation modules with the value `MPIPL / 7.22472 + 1`, where `MPIPL` is the user-defined initial precision level in digits (see Table 2). On Cray vector systems, the constant 7.22472 in Table 2 and in the above discussion should be replaced by 6.62266.

With regards to the MP epsilon `MPEPS`, quotes should be used when changing the value of this variable, as in

```
MPEPS = '1E-500'
```

The quotes here insure that the constant is converted with full multiple precision. Without quotes, the constant will not be accurately converted, and in fact a constant of such a small size would result in an underflow condition on IEEE arithmetic systems.

5. The Fortran-90 MPFUN Package

The new Fortran-90 translation modules, like the older TRANSMP program, generate calls to the MPFUN library, which contains all of the subroutines that perform MP operations. With the advent of Fortran-90, the MPFUN library has also been updated to use some of the advanced features of the this language. Among the changes in the new MPFUN package are the elimination of common blocks and the dynamic allocation of scratch space. Thus the user never needs to worry about "insufficient scratch space" error messages.

One algorithmic change in the Fortran-90 version of the MPFUN library is the substitution of the author's PSLQ integer-relation finding algorithm [8, 2] for the HJLS algorithm [9] that was used in subroutine `MPINRL` of the Fortran-77 MPFUN. The PSLQ algorithm does not exhibit the catastrophic numerical instabilities that are a characteristic of the HJLS algorithm. With PSLQ, integer relations can be reliably detected when the precision level is set to only slightly higher than that of the input data.

Another algorithmic change in the Fortran-90 MPFUN is the utilization of an improved fast Fourier transform (FFT) algorithm [3], which is used by the advanced MP multiplication routine of MPFUN. This new FFT algorithm, which is variously called the "factored" or "four-step" FFT, exhibits significantly improved performance on computers that employ cache memory systems.

That this new FFT scheme is significantly more efficient on modern RISC systems can be seen from Table 3, which compares the performance of the new Fortran-90 MPFUN with the author's previous Fortran-77 MPFUN. These timings were performed on an IBM RS6000/590 workstation and compare the run time required to compute the constant π to the specified precision levels (excluding binary to decimal conversion). The numbers of digits shown in the second column correspond to 2^m numbers of words, which are convenient precision levels for the FFT-based multiplication routine. Note that the new MPFUN package is up to four times faster than the old on this computation, even though the FFT routine only constitutes part of the operations being performed.

One addition to the Fortran-90 MPFUN package is a routine to perform binary to decimal string conversion for extra-high precision arguments. This routine, named MPOUTX, employs a divide-and-conquer scheme, which together with the extra-high precision multiplication and division routines, permits rapid conversion of MP numbers whose precision ranges from roughly 1000 digits to millions of digits. MPOUTX uses the same calling sequence as the existing routine MPOUTC, which suffices for more modest precision levels.

	Prec. Level	CPU Time	
m	(Digits)	Old MPFUN	New MPFUN
4	115	0.0039	0.0037
5	231	0.0077	0.0073
6	462	0.0183	0.0172
7	924	0.0494	0.0480
8	1849	0.1251	0.1279
9	3699	0.3094	0.3295
10	7398	0.6668	0.7141
11	14796	1.4613	1.5639
12	29592	3.2861	3.5167
13	59184	13.3865	8.0605
14	118369	55.1227	18.2515
15	236739	150.3923	40.9832
16	473479	393.6824	94.5782

Table 3: Time to Compute π on an IBM RS6000/590 Workstation

Extra scratch space is required for MPOUTX, but this space is automatically allocated by the routine when required.

6. An Application of the Fortran-90 Multiprecision System

In April 1993, Enrico Au-Yeung, an undergraduate at the University of Waterloo, brought to the attention of the author's colleague Jonathan Borwein the curious fact that

$$\begin{aligned}\sum_{k=1}^{\infty}\left(1+\frac{1}{2}+\cdots+\frac{1}{k}\right)^2 k^{-2} &= 4.59987\cdots \\ &\approx \frac{17}{4}\zeta(4) = \frac{17\pi^4}{360}\end{aligned}$$

based on a computation to 500,000 terms. Borwein's reaction was to compute the value of this constant to a higher level of precision in order to dispel this conjecture. Surprisingly, his computation to 30 digits affirmed it. The present author then computed this constant to 100 decimal digits, and the above equality was still affirmed.

Intrigued by this empirical result, the author, J. Borwein and R. Girgensohn have researched several classes of iterated sums. We have termed these Euler sums, since Euler first studied them in a letter to Goldbach. One class of Euler sums is the following:

$$\begin{aligned}s_{\mathrm{h}}(m,n) &= \sum_{k=1}^{\infty}\left(1+\frac{1}{2}+\cdots+\frac{1}{k}\right)^m (k+1)^{-n} \\ &= \sum_{k=1}^{\infty} h^m(k)(k+1)^{-n}\end{aligned}$$

where $h(k) = 1 + 1/2 + \cdots + 1/k$. By applying the Euler-Maclaurin summation formula [1], one can show that

$$\begin{aligned} h(k) &= \gamma + \ln k + \frac{1}{2k} - \frac{1}{12k^2} + \frac{1}{120k^4} - \frac{1}{252k^6} + \frac{1}{240k^8} \\ &\quad - \frac{1}{132k^{10}} + \frac{691}{32760k^{12}} - \frac{1}{12k^{14}} + \frac{3617}{8160k^{16}} + O(k^{-18}). \end{aligned} \tag{1}$$

We will use $\bar{h}(k)$ to denote this particular approximation (i.e., (1) without the error term).

Let c be a large integer, and let $g(t) = \bar{h}^m(t)(t+1)^{-n}$. By applying the Euler-Maclaurin summation formula again, one can show that

$$\begin{aligned} s_{\mathrm{h}}(m,n) &= \sum_{k=1}^{c} \left(1 + \frac{1}{2} + \cdots + \frac{1}{k}\right)^m (k+1)^{-n} \\ &\quad + \sum_{k=c+1}^{\infty} \left(1 + \frac{1}{2} + \cdots + \frac{1}{k}\right)^m (k+1)^{-n} \\ &= \sum_{k=1}^{c} h^m(k)(k+1)^{-n} + \int_{c+1}^{\infty} g(t)\,dt + \frac{1}{2} g(c+1) \\ &\quad - \sum_{k=1}^{9} \frac{B_{2k}}{(2k)!} D^{2k-1} g(c+1) + O(c^{-18}). \end{aligned} \tag{2}$$

where B_{2k} denotes the Bernoulli constants and D denotes the differentiation operator.

This formula suggests the following computational scheme. First, explicitly evaluate the sum $\sum_{k=1}^{c} h^m(k)(k+1)^{-n}$ for $c = 10^8$, using a numeric working precision of 150 digits. Secondly, perform the symbolic integration and differentiation steps indicated in formula (2). Finally, evaluate the resulting expression, again using a working precision of 150 digits. The final result should be equal to $s_{\mathrm{h}}(m,n)$ to approximately 135 significant digits.

The author has performed many computations of this type [2]. The integration and differentiation operations required in (2) can be handled using a symbolic mathematics package, such as Maple [10]. The explicit summation of the first c terms, as indicated above, has been performed using both the author's TRANSMP translator and the new Fortran-90 multiprecision system.

Once a highly accurate numerical value of one of these sums has been obtained, one can ask whether the sum satisfies some simple formula involving basic mathematical constants. This can be done with an integer relation algorithm, such as the PSLQ algorithm that was developed by H. R. P. Ferguson and the author [8]. I will present but one example of these computations here. Consider

$$\begin{aligned} s_{\mathrm{h}}(2,7) &= \sum_{k=1}^{\infty} \left(1 + \frac{1}{2} + \cdots + \frac{1}{k}\right)^2 (k+1)^{-7} \\ &= 0.0091346205773347893705892376775215252409185580168153 78\cdots \end{aligned}$$

Based on experience with other constants, J. Borwein and the author conjectured that this constant satisfies a relation involving the constants $\zeta(9)$, $\zeta(4)\zeta(5)$, $\zeta(3)\zeta(6)$, $\zeta^3(3)$, and

$\zeta(2)\zeta(7)$. When $s_{\mathrm{h}}(2,7)$ is augmented with this set of terms, all computed to 135 decimal digits accuracy, and the resulting 6-long vector is input to the PSLQ algorithm, it detects the relation $(6, -1, 3, 9, -2, -6)$ at iteration 46. Solving this relation for $s_{\mathrm{h}}(2,7)$, we obtain the formula

$$\begin{aligned} s_{\mathrm{h}}(2,7) &= \frac{1}{6}\zeta(9) - \frac{1}{2}\zeta(4)\zeta(5) - \frac{3}{2}\zeta(3)\zeta(6) + \frac{1}{3}\zeta^3(3) + \zeta(2)\zeta(7) \\ &= \frac{1}{6}\zeta(9) - \frac{\pi^4}{180}\zeta(5) - \frac{\pi^6}{630}\zeta(3) + \frac{1}{3}\zeta^3(3) + \frac{\pi^2}{6}\zeta(7) \end{aligned}$$

(recall that $\zeta(2n) = (2\pi)^{2n}|B_{2n}|/\ [2(2n)!]$). See [2] for full details and numerous other experimentally discovered results of this type.

Certainly such computation results, even if confirmed to very high precision, do not constitute formal proofs of the resulting identities. However, since these computations were first performed, formal proofs have been found for several of these experimental results [6]. The experimental results thus pointed the way to the formal results.

References

[1] K. E. Atkinson, *An Introduction to Numerical Analysis*, John Wiley, New York, 1989.

[2] D. H. Bailey, J. M. Borwein and R. Girgensohn, "Experimental Evaluation of Euler Sums," NASA Ames RNR Technical Report RNR-93-014, to appear in *Experimental Mathematics*.

[3] D. H. Bailey, "FFTs in External or Hierarchical Memory," *Journal of Supercomputing*, vol. 4, no. 1 (March 1990), p. 23 – 35.

[4] D. H. Bailey, "Multiprecision Translation and Execution of Fortran Programs," *ACM Transactions on Mathematical Software*, vol. 19, no. 3, Sept. 1993, p. 288 - 319.

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

[6] D. Borwein, J. M. Borwein and R. Girgensohn, "Explicit Evaluation of Euler Sums," to appear in *Proc. Edinburgh Mathematical Society*.

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

[8] H. R. P. Ferguson and D. H. Bailey, "A Polynomial Time, Numerically Stable Integer Relation Algorithm," RNR-91-032.

[9] 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.

[10] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, S. M. Watt, *Maple V Language Reference Manual*, Springer-Verlag, New York, 1991.