Multiple Precision Complex Arithmetic and Functions

David M. Smith

Loyola Marymount University

This paper describes a collection of Fortran routines for multiple precision complex arithmetic and elementary functions. The package provides good exception handling, flexible input and output, trace features, and results that are almost always correctly rounded. For best efficiency on different machines, the user can change the arithmetic type used to represent the multiple precision numbers.

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

General Terms: Algorithms, Performance, Reliability

Additional Key Words and Phrases: Complex arithmetic, multiple precision, accuracy, function evaluation, floating point, Fortran, mathematical library, portable software

---

1. INTRODUCTION

The ZM package is a collection of Fortran subroutines that performs floating point multiple precision evaluation of complex arithmetic and elementary functions. These routines use the FM package [7] for real multiple precision arithmetic, constants, and elementary functions.

Brent's MP package [4] did not support complex arithmetic, and Bailey's more recent MP package [2,3] provides complex arithmetic and some complex elementary functions, and contains a Fortran-90 module defining multiple precision derived types.

ZM also provides a Fortran-90 module that defines three multiple precision data types and provides the interface routines for overriding arithmetic operators and intrinsic Fortran-90 functions. This allows a program to declare variables as multiple precision real, integer, or complex, and then to describe operations on these variables using the normal syntax for arithmetic expressions.

ZM versions are available for the numerical Fortran-90 intrinsic functions, and they provide good speed, rounding, and exception handling. They support flexible input and output conversion,

and have the capability for automatic tracing of arithmetic and functions.

There are three multiple precision data types:

FM	(multiple precision real)
IM	(multiple precision integer)
ZM	(multiple precision complex)

The following table lists the operations defined in the interface module for the three derived types. For each of the operations =, +, -, *, /, **, .EQ., .NE., .GT., .GE., .LT., and .LE., the interface module defines all mixed mode variations involving one of the three multiple precision derived types and another argument having one of the types integer, real, double, complex, complex double, FM, IM, ZM.

This insures that if `A` is type FM, mixed mode expressions such as

```
A = 12
A = A + 1
IF (ABS(A).LT.1.0D-23) THEN
```

are handled correctly.

Not all the named functions are defined for all three multiple precision derived types, so the list below shows which can be used. The labels "real", "integer", and "complex" refer to types FM, IM, and ZM respectively. For functions that accept two or more arguments, like `ATAN2` or `MAX`, all the arguments must be of the same type.

In addition to these types, the three conversion functions `TO_FM`, `TO_IM`, and `TO_ZM` accept character strings for input arguments, (e.g., `TO_FM('3.45')`), and those functions also accept any of the machine precision data types integer, real, double, complex, or complex double. For converting a multiple precision number to one of the machine precision types, there are five conversion functions: `TO_INT`, `TO_SP`, `TO_DP`, `TO_SPZ`, and `TO_DPZ`.

Table 1. Available operations for multiple precision derived types

=	real	integer	complex	HUGE	real	integer	complex
+	real	integer	complex	INT	real	integer	complex
-	real	integer	complex	LOG	real		complex
*	real	integer	complex	LOG10	real		complex
/	real	integer	complex	MATMUL	real	integer	complex
**	real	integer	complex	MAX	real	integer	
.EQ.	real	integer	complex	MAXEXPONENT	real		
.NE.	real	integer	complex	MIN	real	integer	
.GT.	real	integer	complex	MINEXPONENT	real		
.GE.	real	integer	complex	MOD	real	integer	
.LT.	real	integer	complex	MODULO	real	integer	
.LE.	real	integer	complex	NEAREST	real		

ABS	real	integer	complex	NINT	real	integer	complex
ACOS	real		complex	PRECISION	real		complex
AIMAG			complex	RADIX	real	integer	complex
AINT	real		complex	RANGE	real	integer	complex
ANINT	real		complex	REAL	real	integer	complex
ASIN	real		complex	RRSPACING	real		
ATAN	real		complex	SCALE	real		complex
ATAN2	real			SETEXPONENT	real		
BTEST		integer		SIGN	real	integer	
CEILING	real		complex	SIN	real		complex
CMPLX	real	integer		SINH	real		complex
CONJ			complex	SPACING	real		
COS	real		complex	SQRT	real		complex
COSH	real		complex	TAN	real		complex
DBLE	real	integer	complex	TANH	real		complex
DIGITS	real	integer	complex	TINY	real	integer	complex
DIM	real	integer		TO_FM	real	integer	complex
DINT	real		complex	TO_IM	real	integer	complex
DOTPRODUCT	real	integer	complex	TO_ZM	real	integer	complex
EPSILON	real			TO_INT	real	integer	complex
EXP	real		complex	TO_SP	real	integer	complex
EXPONENT	real			TO_DP	real	integer	complex
FLOOR	real	integer	complex	TO_SPZ	real	integer	complex
FRACTION	real		complex	TO_DPZ	real	integer	complex

2. EFFICIENCY

Prior to Bailey's package, most multiple precision packages stored the numbers in integer arrays. This was faster for machines of the time, but Bailey reported that for some supercomputers and scientific workstations it was better to use double precision. His routines stored the numbers in single precision arrays and used double precision arithmetic to accumulate the multiple precision results.

The advantage of using double precision arithmetic is that a larger base can be used for the multiple precision arithmetic. This reduces the number of array elements required to store a number with a given precision. The base is usually restricted to be less than the square root of the largest representable integer. Using integer arithmetic on a 32-bit computer, each word holds about four decimal digits, compared to seven for double precision.

Multiplication, division, and functions have running times no faster than $O(t^2)$ for t words of precision unless t is large. These times may be faster by a factor of about three if double precision arithmetic is as fast as integer arithmetic. At commonly used lower precisions (20 to 60 decimal

digits), the overhead involved in a general multiple precision package reduces this potential speedup factor.

However, some machines are faster using integer arithmetic. To increase the flexibility of both the FM and ZM packages, they have been written so that the type of arithmetic used for the basic operations can be easily changed. The user can select the type of arithmetic for the packages to use internally. The default is to use double precision. To produce an equivalent integer version, the user duplicates the source code file and uses a text editor to make two (global) string replacements in the code. The details are given in the documentation at the beginning of the new FM package.

The new version of the FM package incorporates several changes that support the ZM routines. There are some new routines, and most of the old ones are faster. The division routine uses a new algorithm [8] that is about twice as fast as the original. In addition, the package now includes a collection of routines for integer multiple precision arithmetic.

Here are some timing comparisons at 50 significant digits and 1000 significant digits that illustrate the effect of using different arithmetic on different machines. For a precision level of 50 digits, the integer runs used 14 digits in base 10^4, while double precision used 8 digits in base 10^7. The reason 14 digits are needed in base 10^4 is that normalization can cause the first word to have only one significant digit, so only about 53 significant digits can be assumed to be present.

Table 2. Complex arithmetic timing (seconds per call on a 604 Macintosh)

	50 S.D.		1000 S.D.	
	ZM Integer	ZM D.P.	ZM Integer	ZM D.P.
add	.000 008	.000 008	.000 053	.000 037
subtract	.000 009	.000 008	.000 052	.000 038
multiply	.000 055	.000 047	.003 130	.001 200
divide	.000 142	.000 074	.011 500	.001 690
sqrt(z)	.000 228	.000 186	.006 330	.002 380
exp(z)	.000 858	.000 842	.062	.027
ln(z)	.002 130	.002 190	.102	.049
sin(z)	.000 908	.000 861	.062	.027
arctan(z)	.002 650	.002 620	.121	.053

Table 3. Complex arithmetic timing (seconds per call on a 68030 Macintosh)

	50 S.D.		1000 S.D.	
	ZM Integer	ZM D.P.	ZM Integer	ZM D.P.
add	.000 280	.000 480	.001 620	.003 380
subtract	.000 303	.000 523	.001 720	.003 500
multiply	.002 750	.003 600	.209	.160
divide	.005 790	.006 120	.450	.307
sqrt(z)	.009 500	.011 600	.360	.279
exp(z)	.032 700	.045 200	3.680	2.900
ln(z)	.073 100	.107 000	5.780	4.810
sin(z)	.033 900	.045 800	3.680	2.860
arctan(z)	.088 900	.127 000	6.620	5.440

These results are average times based on many calls to each routine. The same compiler and optimization settings were used in all cases, as well as the same set of input arguments. These tables are meant to compare the double precision and the integer versions on different machines, and to give a rough idea of the timing for various routines. However, the times can be sensitive to changes in compilers or hardware.

With the more recent (604) machine, double precision is slightly better than integer at 50 digits. For 1000 digits, multiplication and division are much faster in double precision, and this causes the elementary functions to be much faster as well.

The older (68030) machine has slower floating point performance compared to its integer arithmetic. At 50 digits the integer version is much better. The two versions are about equal around 800 digits, and the double precision version is faster for precision much higher than that.

These timing tests were also done using a Sun workstation, 68040 and 601 Macs, and a Pentium PC. The results were similar to Table 2. Although most current machines run the double precision version faster than the integer version, for a machine with double-length integer arithmetic that is at least as fast as its double precision floating-point arithmetic the integer version would be better.

3. ACCURACY

The complex arithmetic and function routines return results that are nearly always correctly

rounded. Precision is raised upon entry to each routine so that the predicted error in the result before the final rounding operation is less than 0.001 units in the last place (measured in terms of the original precision).

The philosophy used for measuring errors is to assume all input arguments are exact. When zero digits are appended to the input arguments as precision is raised at the start of a routine, the values can still be assumed to be exact. This assumption is also used by Hull, Fairgrieve, and Tang [6]. For cases where the the input values really are exact, making this assumption assures that accuracy is not lost needlessly.

Except for cancellation error in addition or subtraction, most algorithms lose accuracy slowly and predictably. A simple guess for how many guard digits are needed will usually prove correct. In rare cases, unexpected cancellation error will mean that the goal of 0.001 ulps of error cannot be achieved using the number of guard digits originally chosen.

With complex arithmetic, even multiplication or division can suffer from cancellation error. The following example using 5 digits in base 10 shows how ZM routines handle this problem, although the actual routines would initially use more than two guard digits in this case.

$(0.63287 + 0.52498\ i) * (0.69301 + 0.83542\ i)$.

If precision is raised to 7 digits and the simple formula for multiplication is used, the result is

6.4000E-6 + 8.9253E-1 i.

The real part is correct to only two significant digits. The multiplication routine detects this loss of significance, raises the precision to a higher level, and tries again. Using at least 10 digits gives the result

6.4471E-6 + 8.9253E-1 i.

This is the correctly rounded answer. Similar cancellation can occur in functions such as

$\ln(\ .77266 + .63483\ i\ ) = 6.3022\text{E-}6 + .68778\ i$.

4. SOME ALGORITHMS USED IN THE PACKAGE

Many of the basic formulas are the ones used by Wynn [11]. Hull, Fairgrieve, and Tang [6] remarked that the formulas for complex arithmetic and elementary functions appear deceptively simple. Many special cases must be handled, including cases where one or more parts of the input arguments are zero, where exceptions (underflow, overflow, or unknown) are generated as output, and where exceptions are encountered as intermediate results but the final results are normal

numbers.

As noted in [6], a good exception-handling facility makes designing the routines much easier. The treatment of exceptions in the FM package simplifies exception-handling for the complex functions.

Because the basic FM real arithmetic is faster than the equivalent complex arithmetic, the elementary complex functions use real functions whenever possible. Some of the complex functions must check for too much cancellation error and on rare occasions must raise the precision and do the calculation a second time.

4.1 Multiplication

For relatively low precision the multiply routine uses the simple formula

$$(a + b\,i)(c + d\,i) = (a\,c - b\,d) + (a\,d + b\,c)\,i.$$

At higher precision the routine first computes $P = (a + b)(c + d)$, then applies the formula

$$(a + b\,i)(c + d\,i) = (a\,c - b\,d) + (P - a\,c - b\,d)\,i.$$

This replaces one $O(t^2)$ multiplication by three $O(t)$ additions.

At low precision, a real multiplication takes about twice as long as a real addition. Because complex multiplication and division each require several real operations, precision must be raised to provide guard digits. The result is that complex multiplication is about seven times slower than real multiplication at low precision, and between three and four times slower at high precision.

4.2 Division

This method is due to Smith [9]. If $|c| < |d|$, let $Q = c/d$ and then use

$$\frac{a + b\,i}{c + d\,i} = \frac{a\,Q + b}{c\,Q + d} + \frac{b\,Q - a}{c\,Q + d}\,i.$$

Otherwise, let $Q = d/c$ and use

$$\frac{a + b\,i}{c + d\,i} = \frac{b\,Q + a}{d\,Q + c} + \frac{b - a\,Q}{d\,Q + c}\,i.$$

This requires three multiplications, three divisions, and three additions. Making $|Q| < 1$ prevents overflow in the intermediate results.

Complex division is about seven times slower than real division at low precision, and about five times slower at high precision.

4.3 Square Root

The algorithm used is due to Friedland [5]. To find $c + d\,i = \sqrt{a + b\,i}$, let

$$t = \sqrt{\frac{|a| + \sqrt{a^2 + b^2}}{2}}.$$

Then if $a \geq 0$, $\quad c = t, \qquad d = \dfrac{b}{2t}.$

Otherwise, $\qquad c = \dfrac{|b|}{2t}, \quad d = \mathrm{Sign}(b)\,t.$

This is about three times slower than a real square root.

4.4 Exponential and Logarithmic Functions

The exponential function uses the formula

$$e^{a+b\,i} = e^a\,\cos(b) + e^a\,\sin(b)\,i.$$

One of the trigonometric functions is computed and the other can be obtained quickly using an identity. The real sine function is about as fast as the exponential, so the complex exponential takes slightly more than twice as long as the real exponential.

For the logarithm, with $|a + b\,i| = \sqrt{a^2 + b^2}$ and $\arg(a + b\,i) = \mathrm{atan2}(b, a)$, then

$$\ln(a + b\,i) = \ln(\,|a + b\,i|\,) + \arg(a + b\,i)\,i.$$

The complex logarithm takes slightly more than twice as long as the real logarithm.

The general exponential, y^x, uses three complex operations: logarithm, multiplication, exponential. The integer power function, y^n, uses the binary method with complex multiplication. The rational power function, $y^{k/n}$, uses Newton iteration to get $y^{1/n}$, followed by a call to the integer power function.

4.5 Trigonometric Functions

$$\cos(a + b\,i) = \cos(a)\,\cosh(b) - \sin(a)\,\sinh(b)\,i$$

$$\sin(a + b\,i) = \sin(a)\,\cosh(b) + \cos(a)\,\sinh(b)\,i$$

$$\tan(a + b\,i) = \frac{\sin(2a)}{\cos(2a) + \cosh(2b)} + \frac{\sinh(2b)}{\cos(2a) + \cosh(2b)}\,i$$

These functions are slower than their real counterparts by factors ranging from 2.5 or 3 at low precision to slightly more than 2 at high precision.

The inverse functions call the complex logarithm and complex square root. The two equivalent forms are used to avoid cases where one or the other suffers from cancellation.

$$\cos^{-1}(z) = -i\,\ln\left(z + i\,\sqrt{1-z^2}\right) = i\,\ln\left(z - i\,\sqrt{1-z^2}\right)$$
$$\sin^{-1}(z) = -i\,\ln\left(\sqrt{1-z^2} + i\,z\right) = i\,\ln\left(\sqrt{1-z^2} - i\,z\right)$$
$$\tan^{-1}(z) = \frac{i}{2}\,\ln\left(\frac{i+z}{i-z}\right)$$

These are two or three times slower than the real versions.

4.6 Hyperbolic Functions

$$\cosh(a + b\,i) = \cosh(a)\,\cos(b) + \sinh(a)\,\sin(b)\,i$$
$$\sinh(a + b\,i) = \sinh(a)\,\cos(b) + \cosh(a)\,\sin(b)\,i$$
$$\tanh(a + b\,i) = \frac{\sinh(2a)}{\cosh(2a) + \cos(2b)} + \frac{\sin(2b)}{\cosh(2a) + \cos(2b)}\,i$$

Timing is similar to the trigonometric functions.

4.7 Conversion and Utility Functions

Routines are provided for complex absolute value, argument, conjugate, real or imaginary parts, integer or nearest integer values, and conversion between real and complex FM formats.

The input conversion routines do free-format conversion from character format to ZM format. Input strings can be in '$(1.23\ ,\ -4.56)$' form or '$1.23\ -\ 4.56\ i$' form. The two numeric parts can be in any integer, fixed, or exponential format. If one part is omitted, input of the form '3.45' or '$-5\,i$' is correctly converted to ZM format with the missing part set to zero.

The output conversion routines can be set for the desired complex format, the number of digits to display, and the real format to use for the individual parts of the complex number.

5. TESTING

To test the accuracy of the functions in this package, many random arguments were generated and results compared at a sequence of increasing precisions.

As independent tests for the functions, ZM results were compared to values tabulated in [1] and to values generated by the Mathematica computer algebra system [10]. Tests were also made comparing ZM results at low precision with values generated by the complex Fortran intrinsic functions.

References

1. Abramowitz, M., and Stegun, I.A. (Eds.) *Handbook of Mathematical Functions*, Dover, New York, 1965.

2. Bailey, D.H. Multiprecision Translation and Execution of FORTRAN Programs. *ACM Trans. Math. Softw.*, 19 (1993), 288–319.

3. Bailey, D.H. A Fortran 90-Based Multiprecision System. *ACM Trans. Math. Softw.*, 21 (1995) 379–387.

4. Brent, R.P. A Fortran Multiple-Precision Arithmetic Package. *ACM Trans. Math. Softw.*, 4 (1978), 57–70.

5. Friedland, P. Absolute Value and Square Root of a Complex Number. *Comm. ACM*, 10 (1967), 665.

6. Hull, T.E., Fairgrieve, T.F., and Tang, P.T.P. Implementing Complex Elementary Functions Using Exception Handling. *ACM Trans. Math. Softw.*, 20 (1994), 215–244.

7. Smith, D.M. A Fortran Package for Floating-Point Multiple-Precision Arithmetic. *ACM Trans. Math. Softw.*, 17 (1991), 273–283.

8. Smith, D.M. A Multiple-Precision Division Algorithm. *Math. Comp.*, 66 (1996), 157–163.

9. Smith, R. L. Complex Division. *Comm. ACM*, 5 (1962), 435.

10. Wolfram, S. *Mathematica: A System for doing Mathematics by Computer*, 2nd ed., Addison-Wesley, Redwood City, Calif., 1991.

11. Wynn, P. An Arsenal of Algol Procedures for Complex Arithmetic. *BIT*, 2 (1962), 232–255.