Accuracy of Complex Mathematical Operations and Functions in Single and Double Precision

Paul Caprioli, Vincenzo Innocente, Paul Zimmermann

▶ **To cite this version:**

Paul Caprioli, Vincenzo Innocente, Paul Zimmermann. Accuracy of Complex Mathematical Operations and Functions in Single and Double Precision. 2024. hal-04714173

HAL Id: hal-04714173
https://inria.hal.science/hal-04714173v1

Preprint submitted on 30 Sep 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Accuracy of Complex Mathematical Operations and Functions in Single and Double Precision

Paul Caprioli[*], Vincenzo Innocente[†], Paul Zimmermann[‡]

September 2024

1 Introduction

The IEEE Standard for Floating-Point Arithmetic [2] requires correct rounding for basic arithmetic operations (addition, subtraction, multiplication, division, and square root) on real floating-point numbers, but there is no such requirement for the corresponding operations on complex floating-point numbers. Furthermore, while the accuracy of mathematical functions has been studied in various libraries for different IEEE real floating-point formats [8], we are unaware of similar studies for complex-valued functions.

The C programming language [3] specifies that complex floating point types have the same representation and alignment requirements as an array type containing exactly two elements corresponding, respectively, to the real and imaginary parts of the complex number. The C++ programming language [1] does the same. Their use of Cartesian coordinates facilitates addition and subtraction but does lead to some non-trivial challenges for multiplication and division. In fact, some implementations do not attempt to avoid intermediate overflows nor NaN-values resulting from Inf − Inf.

Complex mathematical functions may involve branch cuts [10], curves in the complex plane along which the function is discontinuous. The best known example is probably the complex square root, for which

$$\begin{aligned}\mathrm{csqrt}(-1+0i) &= i\\ \mathrm{csqrt}(-1-0i) &= -i\end{aligned}$$

since the sign of the IEEE floating-point zero provides the extra bit of information needed to resolve the discontinuity along the negative real axis. Similarly, the same convention is adopted for the complex logarithm:

$$\begin{aligned}\mathrm{clog}(-1+0i) &= \pi i\\ \mathrm{clog}(-1-0i) &= -\pi i.\end{aligned}$$

[*]High Performance Kernels LLC
[†]CERN
[‡]Université de Lorraine, CNRS, Inria, LORIA

Consistency among complex functions motivates the following definition of complex cube root:

$$\mathrm{ccbrt}(z) = \mathrm{cexp}\big(\mathrm{clog}(z)/3\big).$$

However, this choice is discordant with the real-valued cube root function for negative numbers (i.e., on the branch cut chosen for clog). For example,

$$\mathrm{ccbrt}(-8 + 0i) = 1 + \sqrt{3}i$$

whereas

$$\mathrm{cbrt}(-8) = -2.$$

Despite the challenges, complex floating-point is widely used in high performance computing. For example, the Fast Fourier Transform is an essential algorithm to many application areas [5], and the accuracy both of the underlying complex arithmetic as well as of the complex exponential function is necessary for its overall accuracy [11].

1.1 Theoretical background

In this section, let $u = 2^{-p}$ for binary floating-point of precision p. For example, $u = 2^{-24}$ for single precision binary floating-point, and $u = 2^{-53}$ for double precision binary format. Also, let $\circ(\cdot)$ denote rounding to nearest, and assume that overflow, underflow, and denormals do not occur.

Then, the complex sum as computed by

$$\begin{aligned} z_0 + z_1 &= (a + bi) + (c + di) \\ &\approx \circ(a + c) + \circ(b + d)\,i \end{aligned}$$

is, by definition, correctly rounded in each of its components. Therefore, letting $z = z_0 + z_1 = x + yi$ be the infinitely precise result and $z' = x' + y'i$ be the rounded-to-nearest sum as computed above,

$$\begin{aligned} |z' - z| &= \sqrt{(x' - x)^2 + (y' - y)^2} \\ &< \sqrt{(u\,|x|)^2 + (u\,|y|)^2} \;=\; u\sqrt{|x|^2 + |y|^2} \;=\; u\,|z|, \end{aligned}$$

so the relative normwise error, $|z' - z|/|z|$, of the computed sum is less than u. Furthermore,

$$\begin{aligned} |z' - z| &\le \sqrt{\left(\frac{1}{2}\mathrm{ulp}(x)\right)^2 + \left(\frac{1}{2}\mathrm{ulp}(y)\right)^2} \\ &\le \sqrt{\left(\frac{1}{2}\mathrm{ulp}(|z|)\right)^2 + \left(\frac{1}{2}\mathrm{ulp}(|z|)\right)^2} \;=\; \frac{\sqrt{2}}{2}\,\mathrm{ulp}(|z|). \end{aligned}$$

Note that satisfying the inequalities above does not imply that z' is correctly rounded. For example, suppose $z = 1/8 + i$ and $z' = (1/8 + u/2) + i$. Then the real component of z' is not correctly rounded. Nevertheless, since $|z' - z| = u/2$ and $|z| > 1$, we have $|z' - z| < u\,|z|$, and also, since $\mathrm{ulp}(|z|) \ge 2u$, we have $|z' - z| \le \big(\sqrt{2}/2\big)\,\mathrm{ulp}(|z|)$.

Consider the standard implementation of a complex product without using a fused multiply-add (FMA) operation:

$$\begin{aligned} z_0 \cdot z_1 &= (a + bi) \cdot (c + di) \\ &\approx \circ\big(\circ(ac) - \circ(bd)\big) + \circ\big(\circ(ad) + \circ(bc)\big)\, i. \end{aligned}$$

Brent, Percival, and Zimmermann [4] showed that, for the algorithm above, the maximum relative normwise error is $\sqrt{5}u$.

Using an FMA, the product can better be implemented as follows:

$$\begin{aligned} z_0 \cdot z_1 &= (a + bi) \cdot (c + di) \\ &\approx \circ\big(ac - \circ(bd)\big) + \circ\big(ad + \circ(bc)\big)\, i. \end{aligned}$$

Jeannerod, Kornerup, Louvet, and Muller [9] showed that the maximum relative normwise error of this algorithm is $2u$.

1.2 Alternatives

Fred Tydeman proposed in 2009 another way to do complex multiply and divide in the C language [13]. His FPCE test suite (`http://www.tybor.com`) has tests of accuracy of many complex mathematical functions, as well as the elementary operations, but unfortunately, it is not free (only a free sample is available, with no complex results). The ISO/IEC standard 10967-3 (LIA) specifies complex integer and floating-point arithmetic and complex elementary numerical functions; it introduces a very strong zero, denoted 00, which yields, for example, $00 \cdot \mathrm{NaN} = 00$.

2 Methodology

In this research, we only consider the Cartesian representation of complex numbers (as in the C standard); in particular, we don't consider the polar representation. We don't consider directed rounding modes from IEEE 754 (only the rounding to nearest mode roundTiesToEven). We don't check exception flags are correctly set (underflow, overflow, invalid, division by zero, inexact).

Basic complex operations (addition, multiplication, division) might be dealt with by the C compiler, or by a library call. It is not always possible to know which case happens, which moreover might depend on some compiler options. We use the same compiler options for all operations, as this would typically be the case when building an application program.

We use the algorithm described in Section 3.1 of [8] to search for large errors. One difference is that for complex univariate functions, we have two inputs (the real and imaginary parts), and for bivariate functions, we have four inputs. As a consequence, it is no longer possible to perform an exhaustive search for single precision.

2.1 How to measure the error?

While for real values there is a consensus to measure the error in ulps (units-in-last-place), for complex values there might be different ways to measure the error. Assume $z = (x, y)$ is the result of a computation with infinite precision, and $z' = (x', y')$ is the compiler or library result.

Component-wise error. One way to measure the error is to compute the error e_x in ulps between x and x' (see [8] for a precise definition), the error e_y in ulps between y and y', and to take the maximum of both values. However, if say y is very small compared to x, one might wonder if a large ulp-error on y really makes sense. Consider for example the product in single precision of (`0x1.994c36p-125`, `-0x1.e588c4p-124`) by (`-0x1.5a43d2p+49`, `-0x1.9ac2c8p+50`), where GCC yields (`-0x1.cabaeap-73`, `-0x1p-97`) instead of (`-0x1.cabaeap-73`, `-0x1.7cb4p-106`).

Normwise error. Here we compute the complex difference $\delta = (x' - x, y' - y)$, and compare δ to z. In the literature, one can find bounds of the form $|\delta| < e \cdot u \cdot |z|$ for various constants e, where $u = 2^{-p}$ for p the binary precision, and $|\cdot|$ denotes the Euclidean norm. However, this kind of formula does not work well when $|z|$ is in the subnormal range. We thus prefer to bound δ in terms of $\mathrm{ulp}(|z|)$:

$$|\delta| \leq e \cdot \mathrm{ulp}(|z|). \tag{1}$$

This formula nicely extends the one used in the real case [8], where the absolute value is replaced by the Euclidean norm. In this article we use normwise error e as defined in Eq. (1). If some table entry contains say $e = 0.708$, this means the largest normwise error we found is $|\delta| \leq 0.708 \cdot \mathrm{ulp}(|z|)$, and we found an example with $|\delta| > 0.707 \cdot \mathrm{ulp}(|z|)$. Inputs attaining the corresponding bounds can be found in the source files `complex.c` (for univariate functions) and `complex2.c` (for bivariate functions) in the `binary64` directory of the `math_accuracy` git project [14] (using revision `4924948` for this document).

2.2 Additional notes

Our experiments were done on an Intel Xeon Silver 4214.

About GCC and GNU libc. We used GCC 14.2.0 and GNU libc 2.40 (which was configured with the default settings, i.e., without `-mfma`). The complex multiplication is handled directly by the compiler. We used the compiler options `-mfma` and `-O3`; without `-mfma`, GCC does not emit FMA (fused multiply-add) for the complex multiplication.

About the Intel compiler. We used the Intel compiler 2023.2.1, which calls the corresponding Intel math library. We used the following compiler options: `-ffp-model=precise`, `-mfma`, `-no-ftz`, and `-O3`.

About clang. We use clang 16.0.6 with the GNU libc; thus results are identical to GCC/GNU libc for mathematical functions implemented by library calls. We used the following compiler options: `-mfma`, `-ffp-model=precise`, `-O3`. The compiler option `-fcomplex-arithmetic=promoted`, available in clang 20.0.0git [12], is not compatible with `-ffp-model=precise`. It does give better results for division; however multiplying (`-0x1p+74`, `0x1.5ecce4p+97`) by (`0x1.2ccad2p+98`, `0x1.4a5346p+97`) in single precision yields NaN for the imaginary part instead of +Inf.

About NaN values. When intermediate overflows happens in a complex operation, the real or imaginary part of the result might be NaN. An example is when multiplying (`-0x1p+64`, `0x1p+64`) by (`0x1p+64`, `0x1p+64`) in single precision: the Intel compiler and clang yield (-Inf,NaN) instead of (-Inf,0). Since this would hide more interesting results, we don't consider library results containing

NaN. However, we hope that the compiler or mathematical libraries will fix such issues, at least if the user provides some compiler option.

About Inf values. Sometimes the library returns ±Inf for the real or imaginary part, whereas the exact value is a normal number. One example is multiplying (`-0x1.f9a182p+6`, `-0x1.fb7ea6p+5`) by (`-0x1.038p+121`, `-0x1.01fe26p+120`) in single precision, where the correctly rounded result is (`0x1.80aebap+127`, Inf), but the Intel compiler and clang yield +Inf for the real part. If the Intel compiler rounds to +Inf, it considers the real part would be rounded to 2^{128} or larger with an extended exponent range; we thus consider the Intel compiler result to be (at least) 2^{128}, which yields a huge error. Since this would also hide more interesting results, we don't consider library results when one of the real or imaginary parts is ±Inf (resp. a normal value), and the corresponding part computed by MPC is a normal value (resp. ±Inf). (When both values are ±Inf, if they are of the same sign, we consider there is no error, otherwise we consider the error is Inf.) Like for NaN values, we hope that the compiler or mathematical libraries will fix such issues, at least if the user provides some compiler option.

About the power function. For the power function x^y, when the imaginary part of the exponent y is large, it requires an expensive argument reduction to get an accurate result, which does not seem to be implemented in current libraries. For example, consider in single precision x = (`0x1.6dfff4p+81`, `0x1.2a01bp-90`) and y = (`-0x1.704858p-66`, `0x1.386c1ep+72`), then the Intel math library yields (`0x1.b8d186p-1`, `0x1.046dcp-1`) instead of (`-0x1.b8cf4ep-1`, `-0x1.04718p-1`). A similar issue exists for the GNU libc. As a consequence, for the power function we limit the real and imaginary parts of the exponent to 2^4, for both single and double precision.

Tested mathematical functions. We test the mathematical functions from C99 against the implementation in GNU MPC [6], which assumes there is an implementation both in the C library and in GNU MPC. For `cbrt`, `exp10`, `exp2`, `expm1`, `log1p`, `log2`, `atan2` and `hypot`, there is no implementation in the latest release MPC 1.3.1. GNU libc 2.40 does not provide `exp10` nor `exp2`. The results with `exp10`, `exp2`, `tan` and `tanh` were obtained with the development version of MPC (for `tan` and `tanh`, the MPC 1.3.1 implementation is too slow when the imaginary part is huge).

3 Single Precision

The following table shows the results for the complex bivariate operations in single precision. Note that "gcc" might mean either GCC or the GNU libc, depending on whether the operation is computed by the compiler or the library. The same applies to clang, which might call the GNU libc.

	gcc	icx	clang
caddf	**0.708**	**0.708**	**0.708**
cmulf	**1.415**	1.768	1.768
cdivf	**0.708**	**0.708**	**0.708**
cpowf	4041.1	**0.708**	4041.1

The following table shows the results for the complex univariate functions in single precision:

	GNU libc	icx
cacosf	7.229	**0.708**
cacoshf	7.255	**0.708**
casinf	5.394	**0.708**
casinhf	5.405	**0.708**
catanf	5.658	**0.708**
catanhf	5.625	**0.708**
ccosf	2.881	**0.708**
ccoshf	2.859	**0.708**
cexpf	1.918	**0.708**
cexp10f	n/a	**4.660**
cexp2f	n/a	**0.708**
clogf	3.118	**0.708**
clog10f	4.370	**0.708**
csinf	2.760	**0.708**
csinhf	2.839	**0.708**
csqrtf	1.820	**0.723**
ctanf	7.062	**0.708**
ctanhf	7.062	**0.708**

We see that except for `cmulf`, `cexp10f`, and `csqrtf`, all entries for the Intel math library are 0.708, which is the rounding up of $\sqrt{2}/2$, the best possible bound for the normwise error (see §1.1).

4 Double Precision

The following table shows the results for the complex bivariate operations in double precision:

	gcc	icx	clang
cadd	**0.708**	**0.708**	**0.708**
cmul	**1.415**	1.768	1.768
cdiv	**3.545**	**3.545**	**3.545**
cpow	3.25e+4	**8.681**	3.25e+4

The following table shows the results for the complex univariate functions in double precision:

	GNU libc	icx
cacos	1.469	**0.708**
cacosh	1.469	**0.708**
casin	5.377	**0.709**
casinh	5.400	**0.708**
catan	5.680	**0.724**
catanh	5.679	**0.724**
ccos	2.822	**0.721**
ccosh	2.795	**0.722**
cexp	1.895	**0.751**
cexp10	n/a	**4.61e+9**
cexp2	n/a	**0.707**
clog	3.185	**0.709**
clog10	4.355	**0.726**
csin	2.834	**0.722**
csinh	2.819	**0.721**
csqrt	**1.809**	Infinity
ctan	6.355	**0.730**
ctanh	6.664	**0.728**

The Intel library's "Infinity" error for `csqrt` is obtained for (`-0x1.004p+1020`, `0x1.03f8p+961`) where the Intel library yields NaN for the real part, while the large error for `cexp10` is obtained for the input point (`-0x1.8dd58ed1692afp-74`, `0x1.93992e5e34b55p+29`).

5 Conclusion

Some libraries publish largest known errors in complex mathematical functions, for example the GNU libc [7]. In that document we see the largest known error for `casinf` on `x86_64` is $1+2i$, which we interpret as a maximal error of 1 ulp on the real part, and 2 ulps on the imaginary part. However, we found an error of 5.34 ulps in section 3 using normwise error Eq. (1). Investigating further, for the input point (`0x1.332904p-9`, `0x1.01167ep-1`) GNU libc yields (`0x1.127fbp-9`, `0x1.eeb51p-2`), with an error of 1.98 ulps on the real part and 5.34 ulps on the imaginary part, which clearly shows the values from [7] are underestimated.

References

[1] ISO/IEC 14882:2020. 2020. *Programming Languages—C++*. International Organization for Standardization, Geneva, Switzerland.

[2] IEEE Std 754-2019. 2019. *IEEE Standard for Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers, New York, NY, USA. 84 pages. https://doi.org/10.1109/IEEESTD.2019.8766229

[3] ISO/IEC 9899:2017. 2018. *Programming Languages—C*. International Organization for Standardization, Geneva, Switzerland.

[4] Richard P. Brent, Colin Percival, and Paul Zimmermann. 2007. Error Bounds on Complex Floating-Point Multiplication. *Mathematics of Computation* 76 (2007), 1469–1481. https://inria.hal.science/inria-00120352

[5] Paul Caprioli and Robby Jenkins. 2023. High Performance Kernels for FFT via Modern C++. https://doi.org/10.5281/zenodo.8253863

[6] Andreas Enge, Philippe Théveny, and Paul Zimmermann. 2022. *MPC: Multiple Precision Complex Library* (1.3.1 ed.). INRIA. http://www.multiprecision.org/mpc/

[7] Free Software Foundation, Inc. 2024. GNU libc: Known Maximum Errors in Math Functions. http://www.gnu.org/software/libc/manual/html_node/Errors-in-Math-Functions.html

[8] Brian Gladman, Vincenzo Innocente, John Mather, and Paul Zimmermann. 2024. Accuracy of Mathematical Functions in Single, Double, Extended Double and Quadruple Precision. https://members.loria.fr/PZimmermann/papers/accuracy.pdf. Version of August, 26 pages.

[9] Claude-Pierre Jeannerod, Peter Kornerup, Nicolas Louvet, and Jean-Michel Muller. 2017. Error bounds on complex floating-point multiplication with an FMA. *Mathematics of Computation* 86, 304 (2017), pp. 881–898. https://doi.org/10.1090/mcom/3123

[10] William Kahan. 1987. Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit. In *The State of the Art in Numerical Analysis*. Clarendon Press, Oxford, 165–211.

[11] James C. Schatzman. 1996. Accuracy of the Discrete Fourier Transform and the Fast Fourier Transform. *SIAM Journal on Scientific Computing* 17, 5 (1996), 1150–1166. https://doi.org/10.1137/S1064827593247023

[12] The Clang Team. 2024. *Clang Compiler User's Manual*. LLVM. https://clang.llvm.org/docs/UsersManual.html#cmdoption-fcomplex-arithmetic

[13] Fred J. Tydeman. 2009. Complex Multiply and Complex Divide, taking into account IEEE-754 signed zeros, signed infinities, NaN, and C99 Imaginary I. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1399.htm

[14] Paul Zimmermann et al. 2024. Math Accuracy Git Project. https://gitlab.inria.fr/zimmerma/math_accuracy.git