Department of Computer Science, University of Otago

Technical Report OUCS-2017-07

Complex Arithmetic is Complex

Author:

Richard O'Keefe
Department of Computer Science, University of Otago, New Zealand

---

Department of Computer Science,
University of Otago, PO Box 56, Dunedin, Otago, New Zealand

http://www.cs.otago.ac.nz/research/techreports.php

Complex arithmetic is complex

Richard A. O'Keefe
Computer Science, Otāgo

October 2017

Abstract

Floating-point arithmetic is tricky. This report examines the apparently simple case of basic complex arithmetic in order to illustrate this.

1 Audience

This report is intended for 3rd year Computer Science students at the University of Otago. In COSC326 they have to complete a lab task probing their knowledge of floating-point arithmetic and its problems. This report looks at complex arithmetic, provided as standard in Fortran for more than 50 years, in Lisp for about 30 years, and in C for nearly 20 years. It is not concerned with uses of complex arithmetic, but with the question "how hard is it to get the basic operations right", so it is really another look at the problems of floating-point arithmetic.

2 Introduction

Floating-point calculations can run into several problems:

overflow A result is too big to represent.

underflow A result is too small to represent.

division by zero Dividing any number by zero.

undefined Such as the square root of -1.

catastrophic cancellation Subtracting two numbers that are very close can leave very few significant digits.

IEEE arithmetic introduces some special values: $+\infty$, $-\infty$, -0, and NaN. I shall ignore NaN, assuming that undefined calculations raise an exception, and catastrophic cancellation goes un-noticed instead of raising the IEEE "inexact" exception. This is a possible configuration for an IEEE system. I shall also ignore the difference between +0.0 and -0.0, concentrating on overflow, underflow, and division by zero.

3 Two representations

Complex numbers are commonly represented in one of two ways:

- Rectangular form: $z = x + \mathbf{i}y = (x, y)$
- Polar form: $z = \rho e^{\mathbf{i}\theta}$

We can convert between these representations using

$$\begin{aligned} x &= \rho \cos\theta \\ y &= \rho \sin\theta \\ \rho &= \sqrt{x^2 + y^2} \\ \theta &= \text{atan2}(y, x) \end{aligned}$$

4 Comparison

For the rectangular representation, $(x_1, y_1) = (x_2, y_2)$ iff $x_1 = x_2$ and $y_1 = y_2$. This is precise.

The polar representation is trickier. $(\rho_1, \theta_1) = (\rho_2, \theta_2)$ iff $\rho_1 = \rho_2 = 0$ or $\rho_1 = \rho_2 > 0$ and $\theta_1 = \theta_2$. The way $(0, \theta_1) = (0, \theta_2)$ even when $\theta_1 \neq \theta_2$ is structurally analogous to the way $+0.0 = -0.0$ in IEEE arithmetic.

We can define two versions of approximate equality, based on but not identical to the ternary predicates Donald Knuth described in section 4.2.2 of *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.*

$u \sim v \quad (\epsilon)$ if and only if $|v - u| \leq \epsilon \max(|v|, |u|)$
$u \approx v \quad (\epsilon)$ if and only if $|v - u| \leq \epsilon \min(|v|, |u|)$

These definitions work uniformly for real numbers, complex numbers, and quaternions. However, we need to worry about overflow and so on. My current implementation is

```
approximates: other tolerance: ε
    |a b c d m u v w|
    a ← self re.
    b ← self im.
    c ← other re.
    d ← other im.
    m ← (|a| ∨ |b|) ∨ (|c| ∨ |d|).
    (m between: √2μ and: (√(0 · 5Ω)))
        ifTrue: [
            m = 0 ifTrue: [↑true]]
        ifFalse: [
            a ← a/m.
            b ← b/m.
            c ← c/m.
            d ← d/m].
```

$u \leftarrow \sqrt{(a^2 + b^2)}$.
$v \leftarrow \sqrt{(c^2 + d^2)}$.
$w \leftarrow \sqrt{((a - c)^2 + (b - d)^2)}$.
$\uparrow (u \vee v) \times \epsilon \geq w$

where Ω is the largest finite float and μ is the smallest postive normalised float. That avoids overflow, underflow, and division by zero.

If we used a different norm for complex numbers, this could be much simpler. Define $|(x, y)|_1$ to be $|x| \vee |y|$. Then

approximates: other tolerance: ϵ
$|a\ b\ c\ d\ m\ w|$
$a \leftarrow$ self re.
$b \leftarrow$ self im.
$c \leftarrow$ other re.
$d \leftarrow$ other im.
$m \leftarrow (|a| \vee |b|) \vee (|c| \vee |d|)$.
$w \leftarrow (|a - c| \vee |b - d|)$.
$\uparrow m \times \epsilon \geq w$

This avoids overflow, division by zero, and underflow. It deserves further investigation.

We are at liberty to define a total order on complex numbers. However, it is provably impossible to define a total order for the complex numbers which is compatible with the laws of an ordered field[1].

The obvious total order is $(x_1, y_1) < (x_2, y_2)$ iff $x_1 < x_2$ or $x_1 = x_2$ and $y_1 < y_2$. This is lexicographic order on the rectangular representation. It is consistent with equality, but not, as noted above, consistent with the laws of an ordered field.

GNU Smalltalk makes Complex a subclass of Number, which means that it has to define $<$. The definition it uses is $(\rho_1, \theta_1) < (\rho_2, \theta_2)$ iff $\rho_1 < \rho_2$. This is not consistent with equality, because it leads to $1 \approx -1$.

The best approach is to ensure that $<$ cannot be used with complex numbers at all.

5 Simple operations

- conjugate: $\bar{z} = (x, -y) = \rho e^{-\mathbf{i}\theta}$
- negation: $-z = (-x, -y) = \rho e^{\mathbf{i}(\theta+\pi)}$
- absolute value: $|z| = \rho$

In rectangular form, conjugate and negation are just sign manipulation, which is exact. Conjugate is also exact in polar form, but negation involves one rounding. Absolute value is exact in polar form.

[1] https://proofwiki.org/wiki/Complex_Numbers_cannot_be_Totally_Ordered

In rectangular form, the absolute value requires the `hypot`() function in C. Abraham Ziv proved a sharp bound on the accuracy attainable in IEEE arithmetic[2] of 1.222 units in the last place using a straightforward algorithm, and the reported worst case error in the glibc hypot function[3] is 1 unit in the last place using a different algorithm.

Computing $\sqrt{x^2 + y^2}$ directly involves the risk of the computation overflowing or underflowing when the result is actually representable. The reason the `hypot`() function exists is to avoid this. Ziv pointed out that the division used in the usual algorithm can be replaced by scaling (using C's `frexp`() and `ldexp`(), for example)[4]

6 Addition and subtraction

Addition and subtraction in rectangular form are easy:

$$(a, b) \pm (c, d) = (a \pm c, b \pm d)$$

The error in each coördinate is at most 1 unit in the last place, for an overall error of $\sqrt{2}$ units in the last place. An overflow will occur if and only if the result cannot be represented.

In polar form, it is sufficiently tricky that people normally convert to rectangular form and back again instead. Given $z_1 = \rho_1 e^{\mathbf{i}\theta_1}$ and $z_2 = \rho_2 e^{\mathbf{i}\theta_2}$, we have $z_3 = z_1 + z_2 = \rho_3 e^{\mathbf{i}\theta_3}$, where

$$\begin{aligned} x_3 &= \rho_1 \cos\theta_1 + \rho_2 \cos\theta_2 \\ y_3 &= \rho_1 \sin\theta_1 + \rho_2 \sin\theta_2 \\ \rho_3 &= \mathrm{hypot}(x_3, y_3) \\ \theta_3 &= \mathrm{atan2}(y_3, x_3) \end{aligned}$$

At first sight, this appears to require two sine evaluations and two cosine evaluations. However, some libraries, including Solaris, Linux, and Cygwin, include a `sincos`() function that computes the sine and cosine together in significantly less time than computing them separately. So we can implement this as

```
sincos(theta1, &s1, &c1);
sincos(theta2, &s2, &c2);
x3 = c1*r1 + c2*r2;
y3 = s1*r1 + s2*r2;
r3 = hypot(x3, y3);
theta3 = atan2(y3, x3);
```

[2] `http://www.ams.org/journals/mcom/1999-68-227/S0025-5718-99-01103-5/S0025-5718-99-01103-5.pdf`

[3] `http://www.gnu.org/software/libc/manual/html_node/Errors-in-Math-Functions.html`

[4] Although a call to `ldexp` is much slower than a division

which could incur up to 4 units in the last places of error in x3 and y3.

Here is one "direct" formula for addition:

$$\begin{aligned} z &= z_1 + z_2 \\ &= \rho e^{\mathbf{i}\phi} \\ \rho &= \sqrt{\rho_1^2 + \rho_2^2 + 2\rho_1\rho_2 cos(\phi_2 - \phi_1)} \\ \phi &= \phi_1 + \operatorname{atan2}(\rho_2 \sin(\phi_2 - \phi_1), \rho\_1 + \rho_2 \cos(\phi_2 - \phi_1) \end{aligned}$$

This needs only one call to a `sincos`() function. Avoiding overflow and underflow in computing ρ is not much harder than implementing `hypot`(), but you are not going to match the accuracy of the rectangular form.

7 Multiplication and division

Multiplication and division in polar form are easy:

$$\begin{aligned} \rho e^{\mathbf{i}\theta} \times \sigma e^{\mathbf{i}\phi} &= (\rho \times \sigma) e^{\mathbf{i}(\theta+\phi)} \\ \rho e^{\mathbf{i}\theta} \div \sigma e^{\mathbf{i}\phi} &= (\rho \div \sigma) e^{\mathbf{i}(\theta-\phi)} \end{aligned}$$

If the angles are represented as binary fixed-point fractions, the angle addition and subtraction can be exact. (Some C libraries have `sinpi`(), `cospi`(), and so on, and this family of trig functions is in the ISO 18661 extensions for C. That makes working with angles as multiples of pi convenient.) The multiplication and division are just one operation, with a worst case error of 1 unit in the last place, overflowing or underflowing only when the result cannot be represented.

This time it is rectangular form which poses problems.

$$\begin{aligned} (x_1, y_1) \times (x_2, y_2) &= (x_1x_2 - y_1y_2, x_1y_2 + x_2y_1) \\ (x_1, y_1) \div (x_2, y_2) &= \left(\frac{x_1x_2 + y_1y_2}{x_2^2 + y_2^2}, \frac{y_1x_2 - x_1y_2}{x_2^2 + y_2^2} \right) \end{aligned}$$

Taking multiplication first, we are essentially computing two-dimensional dot products. Computing $ab \pm cd$ takes two multiplications and an addition or subtraction, each of which may contribute rounding. Brent, Percival, and Zimmerman proved that $fl(z \times w) = (zw)(1 + \delta)$ where $|delta \leq \sqrt{5}$ units in the last place, which is worse than real multiplication, but not so very much worse. Accuracy is not the problem.

The multiplications may overflow, even when the result does not. For example, let $(x_1, y_1) = (x_2, y_2) = (\sqrt{\Omega} + \alpha, \beta)$ where Ω is the largest finite floating-point number, and α and β are chosen such that $\beta > \sqrt{\alpha^2 + 2\alpha\sqrt{\Omega}}$. For example, take $\alpha = 1.0 \times 10^{150}$ and $\beta = 1.5 \times 10^{152}$. Then $(x_1, y_1)^2 = (\Omega - 4.3166 \times 10^{303}, 4.0226 \times 10^{306})$, which is representable, but a straightfoward implementation will start by computing $(\alpha + \sqrt{\Omega})^2$, which overflows.

There is an operation in C and in hardware on some machines that allows dot products to be calculated with fewer roundings. The fused-multiply-add operation, $\text{fma}(x, y, z)$, computes $xy + z$ with a single rounding, and overflows only when the result is not representable as a finite number. We can use this to compute the square of (x, y) as

$$(|x| > |y| \ ? \ \text{fma}(x, x, -y^2) : \text{fma}(y, -y, x^2), \ 2xy)$$

which is slightly better, but not better enough.

The implementation of complex multiplication in one Unix library says "This implementation can raise spurious underflow, overflow, invalid operation, and inexact exceptions. C99 allows this." And this is in one of the "primitive" operations of a numeric type. The Fortran 08 draft of 2010-06-07 has nothing to say about the accuracy of complex arithmtic. What is at issue here is that we *can* do better than the direct implementation, but it costs longer. The UNIX library mentioned above implements multiplication like this:

$x \leftarrow a \times c - b \times d$
$y \leftarrow a \times d + b \times c$
if $(x = x$ or $y = y)$ return (x, y)
fiddle with NaN and Inf

The seldom-used bulky code make inlining this function unattractive, so the C compiler for the Unix system in question has a command line option saying to ignore IEEE quibbles about infinities and NaNs, allowing the compiler to generate simple fast in-line code. The measured speedup for computing the dot product of long complex vectors is **6.4** times. Using clang on another machine with another version of Unix, the measured speedup was **3.6** times. Either way. this is a huge price to pay for getting the non-finite corner cases right. It seems absurd to be so picky about non-finite cases when the core algorithm is allowed to return spurious non-finite results.

This does set a standard for "safer" multiplication. It might be acceptable if it isn't *too* much slower.

There is another algorithm[5] which does fewer multiplications:

$a \leftarrow x_2 \times (x_1 + y_1)$
$b \leftarrow x_1 \times (y_2 - x_2)$
$c \leftarrow y_1 \times (x_2 + y_2)$
return $(a - c, a + b)$

This is no less susceptible to overflow, underflow, and cancellation than the usual algorithm, and may be less accurate. What's more, it will be slower on a modern machine, and unlike fused multiply-add, there is no $x(y + z)$-with-a-single-roundoff operation.

One open-source Smalltalk uses Ungar's algorithm[6].

[5] https://en.wikipedia.org/wiki/Multiplication_algorithm, citing Volume 2 of Knuth's Art of Computer Programming

[6] *loc. cit*

$a \leftarrow x_1 \times x_2$.
$b \leftarrow y_1 \times y_2$.
$c \leftarrow (x_1 + y_1) * (x_2 + y_2)$.
↑Complex real: $a - b$ imaginary: $c - a - b$

Like the previous algorithm, this takes three multiplies and 5 adds-or-subtracts, and has no obvious advantage over the direct algorithm on a modern machine.

The best I have been able to come up with is

$x \leftarrow a \times c - b \times d$
$y \leftarrow a \times d + b \times c$
if (x is finite and y is finite) return (x, y)
$m \leftarrow |a| \vee |b| \vee |c| \vee |d|$
$a \leftarrow a/m$
$b \leftarrow b/m$
$c \leftarrow c/m$
$d \leftarrow d/m$
$x \leftarrow (a \times c - b \times d) \times m$
$y \leftarrow (a \times d + b \times c) \times m$
return $(x \times m, y \times m)$

which doesn't get the IEEE details right, but in the absence of NaNs, only bothers with scaling when the direct algorithm might have run into spurious overflow.

Reducing the number of multiplications is interesting because if you are implementing an algorithm in hardware, you can use three multiplier units instead of four, so reducing error. This might be done for a signal processing application using fixed point arithmetic, for example. Reducing the number of multiplications is not meant to improve the behaviour of floating-point algorithms on general purpose computers.

7.1 Division

Michael Baudin, in a 2011 draft paper called "Error bounds of complex arithmetic", showed that $fl(z/y) = (x/y)(1 + \delta)$ where $|\delta| \leq 6$ units in the last place using the naïve algorithm and about 9.9 units in the last place for Smith's algorithm, which is described next.

G. W. Stewart, "A Note on Complex Division", *ACM Transactions on Mathematical Software 11*, 3 (Sept. 1985), pp 238–241, starts by introducing Smith's algorithm:

if ($|x_2| \geq |y_2|$) {
    $t \leftarrow y_2/x_2$
    $d \leftarrow x_2 + t \times y_2$
    $u \leftarrow (x_1 + t \times y_1)/d$
    $v \leftarrow (y_2 - t \times x_2)/d$
} else {

$t \leftarrow x_2/y_2$
$d \leftarrow y_2 + t \times x_2$
$u \leftarrow (y_1 + t \times x_1)/d$
$v \leftarrow (x_1 - t \times y_1)/d$
}
return (u, v)

and says "If the operations are performed in the order indicated by the parentheses, the resulting algorithm is remarkably robust in the presence of exponent exceptions, provided underflows are denomralized" (as they are in IEEE arithmetic). "... when the algotithm works, it returns a computed value $\tilde{z}$ satisfying $|\tilde{z} - z| \leq \epsilon|z|$" where z is the exact value and "ϵ is of the same order of magnitude as the rounding unit for the arithmetic in question. Moreover, the algorithm works for virtually all problems in which the numerator, denominator, and quotient are representable as normalized floating-point numbers."

This sounds like exactly what we want. Sadly, it is not that simple. Stewart goes on to say "However, $\tilde{z}$ being accurate in the sense" that $|\tilde{z} - z| \leq \epsilon|z|$ "does not insure the accuracy of its real and imaginary components" and points out that x_2/y_2 or y_2/x_2 may underflow to zero.

Stewart chooses to exploit a rather interesting fact about floating-point numbers.

> Let $x_1, x_2, \ldots, x_n > 0$ be representable numbers and suppose that $x_1 \times x_2 \times \cdots \times x_n$ is representable. Then $\max\{x_i\} \times \min\{x_i\}$ is also representable.

This means, for example, that xyz can be computed by sorting so that $|x'| \geq |y'| \geq |z'|$ and computing $(x' \times z') \times y'$, and this will overflow or underflow only when overflow or underflow cannot be avoided.

Stewart's algorithm is

$p(u, v, w) =$
  if $|u| \geq |v|$ then $v \times (u \times w)$ else
  if $|u| \geq |w|$ then $u \times (v \times w)$ else
    $w \times (v \times u)$

flip $\leftarrow |y_2| \geq |x_2|$
if flip then $x_2 \leftrightarrow y_2$, $x_1 \leftrightarrow y_1$
$s \leftarrow 1/x_2$
$t \leftarrow 1/(x_2 + y_2 \times (y_2 \times s))$
if $|y_2| \geq |s|$ then $y_2 \leftrightarrow s$
$u \leftarrow t \times (x_1 + p(y_1, s, y_2))$
$v \leftarrow t \times (y_1 - p(x_1, s, y_2))$
if flip then $v \leftarrow -v$
return (u, v)

where $x \leftrightarrow y$ means to swap x and y.

This is complicated enough that I have probably mangled it in transcription. Indeed, Stewart had to issue a corrigendum to say that three + signs in the original version of the code should have been − signs, but forgot to say which three.

In"A Robust Complex Division in Scilab"[7], Baudin and Smith were not as sanguine about Smith's method as Stewart. They wrote that "Smith's method may fail more often than expected" and "it is easy to find particular complex divisions where Smith's method fails ... the failure is *complete* in the sense that none of the digits in the result are correct." Their grounds for this claim are "randomized numerical experiments".

How good are their algorithms? They wrote, "We compared our method with other algorithms and found that most algorithms which claimed ... improved accuracy or improved performance were in fact significantly less accurate than expected. More precisely, our numerical experiments suggest that the rate of failure of our improved algorithm is as low as Stewart's, and might be *4 orders of magnitude lower* than a naive implementation and *2 orders of magnitude lower* than the other known implementations." (My emphasis.) But they warn that "This improved algorithm can still fail in a significant number of situations."

The first of their algorithms can be expressed in C thus:

```
#define internal(a, b, c, d, e, f) { \
    double const r = d/c;            \
    double const t = c + d*r;        \
    double u, v;                     \
    if (r != 0.0) {                  \
        u = b*r,     v = a*r;        \
    } else {                         \
        u = (b/c)*d, v = (a/c)*d;    \
    }                                \
    e = (a + u)/t;                   \
    f = (b - v)/t;                   \
}
double complex cxdiv(double complex x, double complex y) {
    double const a = creal(x);
    double const b = cimag(x);
    double const c = creal(y);
    double const d = cimag(y);
    double       e, f;
    if (fabs(d) <= fabs(c)) {
        internal(a, b, c, d, e, f);
    } else {
        internal(b, a, d, c, e, f);
        f = -f;
    }
    return e + f * I;
```

[7]https://arxiv.org/abs/1210.4539

```
}
```

Smith's algorithm, Stewart's, and this one all try to avoid overflow in $x_2^2 + y_2^2$ in very much the same way as the hypot() function does.

Their "robust" algorithm is considerably more complicated. They claim that algorithm is quite accurate as well as robust.

8 Exponentiation

Raising a complex number to an integer power depends only on multiplication and division. In polar form it is trivial:

$$(\rho, \theta)^n = (\rho^n, (n\theta) \bmod 2\pi)$$

In rectangular form, we can use the usual logarithmic time algorithm:

if $n < 0$ then $n \leftarrow -n$, $z \leftarrow (1, 0)/z$
$r \leftarrow z$ if n is odd else $(1, 0)$
while $(n \leftarrow \lfloor n/2 \rfloor) \neq 0$ do
    $z \leftarrow z^2$
    if n is odd then $r \leftarrow r \times z$
return r

Measuring over the range n=-9 to n=9, using this function was 6 to 46 times faster than the C99 cpow() function.

Raising to a real or complex power would require considering complex logarithms and exponentials, making this a good place to stop.