15 Complex arithmetic primitives

GAUSS[1] ESTABLISHED THE MODERN THEORY OF NUMBERS,
GAVE THE FIRST CLEAR EXPOSITION OF COMPLEX NUMBERS,
AND INVESTIGATED THE FUNCTIONS OF COMPLEX VARIABLES.

— *The Columbia Encyclopedia* (2001).

Apart from Fortran and symbolic-algebra systems, few programming languages support complex arithmetic. The 1998 ISO C++ Standard adds a standard header file, `<complex>`, to provide a `complex` data-type template, and prototypes of complex versions of a dozen or so elementary functions. The 1999 ISO C Standard goes further, offering a standard header file, `<complex.h>`, and a new language keyword, `_Complex`. When `<complex.h>` is included, the name `complex` can be used as a synonym for the new keyword, allowing declaration of objects of type `float complex`, `double complex`, and `long double complex`. The header file declares nearly two dozen function prototypes for each of those three data types.

The C99 Standard also specifies a built-in pure imaginary data-type modifier, called `_Imaginary`, and a new keyword, `_Imaginary_I`, representing the constant $i = \sqrt{-1}$ as one solution of the equation $i^2 = -1$. The `<complex.h>` header file defines macro synonyms `imaginary` for the type and `I` for the constant. However, the Standard makes the imaginary type optional: it is available if, and only if, the macros `imaginary` and `_Imaginary_I` are defined. If the imaginary type is not supported, then `I` is defined to be the new keyword `_Complex_I`, which has the value of the imaginary unit, and type of `float _Complex`. Because of its optional nature, we avoid use in the `mathcw` library of the imaginary type modifier; the complex data types are sufficient for our needs.

The IEEE *Portable Operating System Interface (POSIX) Standard* [IEEE01] requires the same complex arithmetic support as C99, but defers to that Standard in the event of differences.

Annex G of the C99 Standard contains this remark:

> *A complex or imaginary value with at least one infinite part is regarded as an* infinity *(even if its other part is a NaN). A complex or imaginary value is a* finite number *if each of its parts is a finite number (neither infinite nor NaN). A complex or imaginary value is a* zero *if each of its parts is a zero.*

That first statement is surprising, because it implies different computational rules that might ignore, or fail to propagate, the NaN. That is contrary to the usual behavior of NaNs in the operations and functions of real arithmetic, and in this author's view, likely to be more confusing than useful.

The C99 Standard requires implementations of the complex data type to satisfy this condition [C99, §6.2.5, ¶13, p. 34]:

> *Each complex type has the same representation and alignment requirements as an array type containing exactly two elements of the corresponding real type; the first element is equal to the real part, and the second element to the imaginary part, of the complex number.*

The mandated storage layout follows the practice in Fortran, and means that, in C99, a complex argument passed by address can be received as a pointer to a two-element array. That simplifies interlanguage communication.

The C89 Standard adds support for `struct` return values, and the C99 Standard allows complex return values from functions. However, the C language has never permitted arrays to be returned from functions, nor does it support operator overloading. The only solution seems to be implementation of the type as a `struct` instead of an array. Otherwise, it is *impossible* to retrofit full support for complex types into older C implementations without modifying the compiler itself.

[1]The German scientist Carl Friedrich Gauss (1777–1855) was one of the most influential mathematicians in history, with important contributions in algebra, astronomy, complex analysis, geodesy, geometry, magnetism, number theory, numerical quadrature, probability, statistics, and telegraphy. The unit of magnetic induction is named after him. He was a child prodigy, a mental calculating savant, and a polyglot. He also invented the *heliotrope*, a device for using mirrors to reflect sunlight over long distances in land surveying.

For that reason, the complex-arithmetic primitives that we describe in this chapter take a different approach: when a complex function result is required, it appears as the *first* argument. The function is then declared to be of type `void`, and the code guarantees that the result argument can overlap with an input argument, because that often proves convenient in a chain of complex-arithmetic operations.

In addition, to avoid collisions with the C99 function names, we use prefix letters `cx` instead of the standard prefix letter `c`. Thus, our `cxabs()` corresponds to C99's `cabs()`.

Our implementation of support for complex arithmetic then provides a *completely portable* set of functions. When the compiler can support the C99-style extensions, as indicated by the compile-time definition of the standard macro `__STDC_IEC_559_COMPLEX__`, it is easy to provide the new functions as wrappers around calls to our own set. Although the C99 Standard requires the enabling macro to be defined by the language, at the time of writing this, some implementations that claim at least partial support of that Standard require `<complex.h>` to be included to have the symbol defined. That is backwards, because the symbol should indicate the availability of the header file, not the reverse. Our code caters to the problem by using instead a private compile-time symbol, `HAVE_COMPLEX`, to enable code that references the `complex` data type. The `<complex.h>` header file is then expected to be available, and the `mathcw` library version of that file, `complexcw.h`, includes the standard file, and then declares prototypes for our additional functions.

At the time of writing this, the GNU `gcc` and Intel `icc` compilers do not permit combining the `complex` modifier with decimal floating-point types. Compilation of the complex decimal functions is therefore suppressed by omitting them from the decimal source-file macros in the `mathcw` package `Makefile`.

In the following sections, after we define some macros and data types, we present in alphabetical order the dozen or so primitives for portable complex arithmetic, along with their C99 counterparts.

15.1 Support macros and type definitions

To simplify, and hide, the representation of complex-as-real data, we define in the header file `cxcw.h` public types that correspond to two-element arrays of each of the supported floating-point types:

```
typedef float                    cx_float                  [2];
typedef double                   cx_double                 [2];
typedef decimal_float            cx_decimal_float          [2];
typedef decimal_double           cx_decimal_double         [2];
typedef long double              cx_long_double            [2];
typedef __float80                cx_float80                [2];
typedef __float128               cx_float128               [2];
typedef long_long_double         cx_long_long_double       [2];
typedef decimal_long_double      cx_decimal_long_double    [2];
typedef decimal_long_long_double cx_decimal_long_long_double [2];
```

Unlike Fortran, which has built-in functions for creating a complex number from two real numbers, C99 instead uses expressions involving the imaginary value, `I`. That value is defined in `<complex.h>` as a synonym for either the new keyword `_Imaginary_I`, if the compiler supports pure imaginary types, or else the new keyword `_Complex_I`. To clarify the creation of complex numbers from their component parts, the header file `complexcw.h` defines the constructor macro

```
#define CMPLX(x,y)      ((x) + (y) * I)
```

that we use in the remainder of this chapter. That header file also defines the macro

```
#define CTOCX_(result,z) CXSET_(result, CREAL(z), CIMAG(z))
```

for converting from native `complex` to the complex-as-real type `fp_cx_t`.

Our C99 complex functions use a new type, `fp_c_t`, for *floating-point complex* data. It is defined with a `typedef` statement to one of the standard built-in complex types.

The header file `cxcw.h` provides a few public macros for inline access to the components of complex-as-real data objects, and their conversion to native `complex` data:

```
#define CXCOPY_(z,w)     CXSET_(z, CXREAL_(w), CXIMAG_(w))
#define CXIMAG_(z)       (z)[1]
#define CXREAL_(z)       (z)[0]
#define CXSET_(z,x,y)    (CXREAL_(z) = (x), CXIMAG_(z) = (y))
#define CXTOC_(z)        CMPLX(CXREAL_(z), CXIMAG_(z))
```

We use them extensively to eliminate explicit array subscripting in all of the complex functions.

Experiments on several platforms with multiple compilers show that code-generation technology for complex arithmetic is immature. The simple constructor function

```
double complex
cmplx(double x, double y)
{
    return (x + y * I);
}
```

can compile into a dozen or more instructions, including a useless multiplication by one in the imaginary part. With high optimization levels, some compilers are able to reduce that function to a single `return` instruction, when the input argument and the output result occupy the same registers.

For the common case of conversion of `fp_cx_t` data to `fp_c_t` values, we can use the storage-order mandate to reimplement the conversion macro like this:

```
#define CXTOC_(z)        (*(fp_c_t *)(&(CXREAL_(z))))
```

Tests show that our new version eliminates the useless multiply, and produces shorter code.

On most architectures, access to the imaginary or real parts requires only a single load instruction, and the assignments in `CXSET_()` can sometimes be optimized to two store instructions, and on some platforms, to just one double-word store instruction. The conversion to native complex data by `CXTOC_()` can often be reduced to two store or register-move instructions, or one double-word store instruction.

Because most of the functions defined in this chapter are short and time critical, we use the underscore-terminated macros to get inline code. For slower code that requires function calls and allows debugger breakpoints, any of the macros that end with an underscore can be replaced by their companions without the underscore.

15.2 Complex absolute value

A complex number $z = x + yi$ can be represented as a point in the plane at position (x, y), as illustrated in **Figure 15.1** on the following page. The notation $x + yi$ is called the *Cartesian form*. Using simple trigonometry, we can write it in *polar form* as

$$\begin{aligned} z &= r\cos(\theta) + r\sin(\theta)i, \\ &= r\exp(\theta i), && \textit{polar form,} \end{aligned}$$

where r is the distance of the point (x, y) from the origin $(0, 0)$, the angle θ is measured *counterclockwise* from the positive x axis, and the exponential is obtained from the famous *Euler formula* for the imaginary exponential:

$$\boldsymbol{\exp(\theta i) = \cos(\theta) + \sin(\theta)i}, \qquad \textit{Euler formula.}$$

The combination of trigonometric functions on the right-hand side is so common that many textbooks give it the name $\operatorname{cis}(\theta)$. In this book, we prefer the exponential form on the left-hand side.

The substitution $\theta = \pi$ in the Euler formula produces

$$\begin{aligned} \exp(\pi i) &= \cos(\pi) + \sin(\pi)i \\ &= -1 + 0i \\ &= -1. \end{aligned}$$

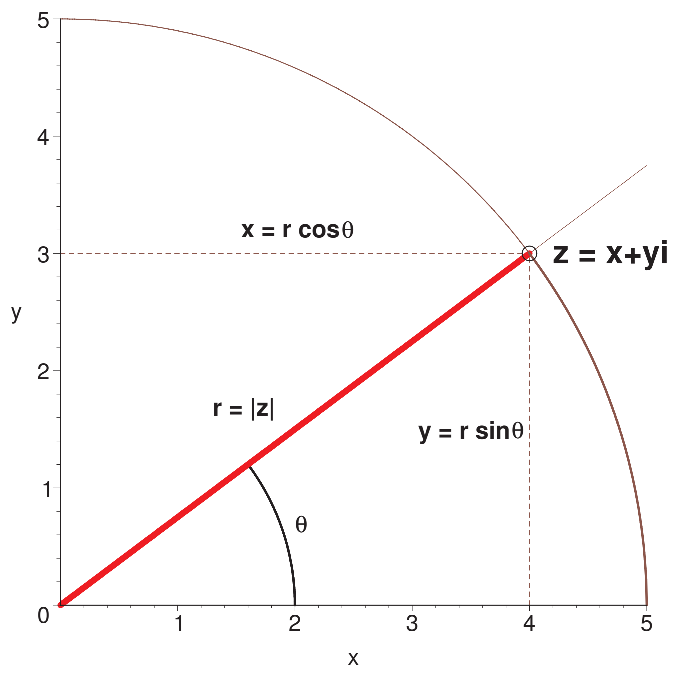

Figure 15.1: Cartesian and polar forms of a point in the complex plane. The angle θ is positive when measured counterclockwise from the positive x axis.

That can be rearranged to produce the *Euler identity* that connects the two most important transcendental numbers in mathematics with the complex imaginary unit and the digits of the binary number system:

$$e^{\pi i} + \mathbf{1} = \mathbf{0}, \qquad \textit{Euler identity.}$$

The absolute value of a complex number $x + yi$ is the *Euclidean distance* of the point (x, y) from the origin: $|z| = r = \sqrt{x^2 + y^2}$. Because the standard `hypot()` function already provides that computation, the C99 Standard mandates use of that function, or equivalent code:

```
fp_t
CXABS(const fp_cx_t z)
{   /* complex absolute value: return abs(z) */
    /* WARNING: this function can overflow for component magnitudes
       larger than FP_T_MAX / sqrt(2): rescale carefully! */

    return (HYPOT(CXREAL_(z), CXIMAG_(z)));
}
```

No special handling of Infinity and NaN components in the complex number is needed, because `HYPOT()` does that work for us.

The commented warning about possible overflow is significant, because the real function family `ABS()` is *never* subject to overflow. Whenever an algorithm requires the complex absolute value, it is essential to provide suitable scaling to prevent premature overflow.

The code for the C99 complex functions, and their function prototypes, is bracketed with preprocessor conditionals that emit code only if `HAVE_COMPLEX` is defined. However, we omit those conditionals in the code presented in this book.

With a few exceptions where efficiency is imperative, and the code is simple, we implement the C99-style functions in terms of our portable functions, as here for the complex absolute value:

```
fp_t
CABS(fp_c_t z)
{   /* complex absolute value: return abs(z) */
    /* WARNING: this function can overflow for component magnitudes
       larger than FP_T_MAX / sqrt(2): rescale carefully! */
    fp_cx_t zz;

    CTOCX_(zz, z);

    return (CXABS(zz));
}
```

15.3 Complex addition

The addition operation for complex numbers is simple: just sum the real and imaginary parts separately:

```
void
CXADD(fp_cx_t result, const fp_cx_t x, const fp_cx_t y)
{  /* complex addition: result = x + y */
    CXSET_(result, CXREAL_(x) + CXREAL_(y), CXIMAG_(x) + CXIMAG_(y));
}
```

The code works correctly even when `result` is the same as either, or both, of `x` or `y`.

The C99-style companion function for complex addition is not likely to be used, but we include it for completeness, because it facilitates machine-assisted translation to C99 code from code that uses the portable complex arithmetic primitives:

```
fp_c_t
CADD(fp_c_t x, fp_c_t y)
{   /* complex addition: return x + y */
    return (x + y);
}
```

15.4 Complex argument

The *argument*, or *phase*, of a complex number $x + iy$ is the angle in radians between the positive x axis and a line from the origin to the point (x, y). That is exactly what the two-argument arc tangent function computes, so the code is easy:

```
fp_t
CXARG(const fp_cx_t z)
{   /* complex argument: return argument t of z = r * exp(i*t) */
    return (ATAN2(CXIMAG_(z), CXREAL_(z)));
}
```

The C99 function is a simple wrapper that calls the portable function:

```
fp_t
CARG(fp_c_t z)
{   /* complex argument: return argument (angle in radians) of z */
    fp_cx_t zz;

    CTOCX_(zz, z);

    return (CXARG(zz));
}
```

From the polar form of complex numbers, it is easy to see that the argument of a product is the sum of the arguments:

$$\arg(w \times z) = \arg(w) + \arg(z).$$

The argument of a quotient is the difference of the arguments, and the argument of a reciprocal is the negation of the argument:

$$\begin{aligned}\arg(w/z) &= \arg(w) - \arg(z),\\ \arg(1/z) &= -\arg(z).\end{aligned}$$

15.5 Complex conjugate

The *conjugate* of a complex number $x + iy$ is just $x - iy$, so the code is easy:

```
void
CXCONJ(fp_cx_t result, const fp_cx_t z)
{   /* complex conjugate: result = complex_conjugate(z) */
    CXSET_(result, CXREAL_(z), -CXIMAG_(z));
}
```

We implement the C99 function with inline code:

```
fp_c_t
CONJ(fp_c_t z)
{   /* complex conjugate: return complex_conjugate(z) */
    return (CMPLX(CREAL(z), -CIMAG(z)));
}
```

15.6 Complex conjugation symmetry

There are two common notations for complex conjugation of a variable or expression in mathematical texts: a superscript star, $z^\star$, or an overbar, $\bar{z}$. In this chapter, we use the star notation because it is easier to see.

The conjugate of a complex number is the reflection of its point on the complex plane across the real axis. In the Cartesian form of complex numbers, we have

$$\begin{aligned} w &= u + vi, & z &= x + yi, & &\textit{for real } u, v, x, \textit{ and } y,\\ w^\star &= u - vi, & z^\star &= x - yi, & &\textit{complex conjugate}. \end{aligned}$$

In the equivalent polar form, we have

$$\begin{aligned} z &= r\exp(\theta i), & &\textit{for real } r \textit{ and } \theta,\\ z^\star &= r\exp(-\theta i), & &\textit{by reflection across the real axis}. \end{aligned}$$

The operation of complex conjugation appears in an important symmetry relation for many complex functions of a single variable, $f(z)$:

$$f(z^\star) = \big(f(z)\big)^\star, \qquad \textit{symmetry under complex conjugation}.$$

To understand the origin of that special symmetry of some complex functions, we look first at how complex conjugation behaves with the low-level operations of complex arithmetic:

$$\begin{aligned}(-z)^\star &= (-x - yi)^\star\\ &= (-x + yi)\\ &= -(x - yi)\end{aligned}$$

$$\begin{aligned}
&= -(z^\star), && \textit{symmetry under negation},\\
(w+z)^\star &= \big((u+x)+(v+y)i\big)^\star \\
&= (u+x)-(v+y)i \\
&= (u-vi)+(x-yi) \\
&= w^\star + z^\star, && \textit{symmetry under addition},\\
(w\times z)^\star &= \big((ux-vy)+(uy+vx)i\big)^\star \\
&= (ux-vy)-(uy+vx)i \\
&= (u-iv)(x-iy) \\
&= w^\star \times v^\star, && \textit{symmetry under multiplication},\\
(1/z)^\star &= \big(1/(x+yi)\big)^\star \\
&= \big((x-yi)/(x^2+y^2)\big)^\star \\
&= (x+yi)/(x^2+y^2) \\
&= 1/(x-yi) \\
&= 1/(z^\star), && \textit{symmetry under reciprocation},\\
(w/z)^\star &= w^\star/z^\star, && \textit{symmetry under division}.
\end{aligned}$$

The last relation follows by combining the symmetry rules for multiplication and reciprocation, but can also be derived by tedious expansion and rearrangement of the numerator and denominator. Thus, the operation of complex conjugation distributes over negations, sums, differences, products, and quotients. Because it holds for products, we conclude that it also holds for integer powers:

$$(z^n)^\star = (z^\star)^n, \qquad \textit{for } n = 0, \pm 1, \pm 2, \ldots.$$

Because it applies for products and sums, it is also valid for polynomials and convergent series with *real* coefficients, p_k:

$$\begin{aligned}
\big(\mathcal{P}(z)\big)^\star &= (p_0 + p_1 z + p_2 z^2 + p_3 z^3 + \cdots)^\star \\
&= (p_0)^\star + (p_1 z)^\star + (p_2 z^2)^\star + (p_3 z^3)^\star + \cdots \\
&= p_0^\star + p_1^\star z^\star + p_2^\star (z^\star)^2 + p_3^\star (z^\star)^3 + \cdots \\
&= p_0 + p_1 z^\star + p_2 (z^\star)^2 + p_3 (z^\star)^3 + \cdots \\
&= \mathcal{P}(z^\star).
\end{aligned}$$

If a function with a single complex argument has a convergent Taylor-series expansion about some complex point z_0 given by

$$f(z) = c_0 + c_1(z-z_0) + c_2(z-z_0)^2 + c_3(z-z_0)^3 + \cdots,$$

then as long as the coefficients c_k are real, we have the conjugation symmetry relation $\big(f(z)\big)^\star = f(z^\star)$. The elementary functions that we treat in this book have that property, and it is of utmost importance to design computational algorithms to ensure that the symmetry property holds for all complex arguments z, whether finite or infinite. However, NaN arguments usually require separate consideration.

The product of a complex number with its conjugate is the square of its absolute value. We can show that in both Cartesian form and in polar form:

$$\begin{aligned}
zz^\star &= (x+yi)\times(x-yi) & zz^\star &= r\exp(\theta i)\times r\exp(-\theta i)\\
&= x^2 - xyi + yxi + y^2 & &= r^2\exp(\theta i - \theta i)\\
&= x^2+y^2 & &= r^2\\
&= |z|^2, & &= |z|^2.
\end{aligned}$$

Rearranging the final identity produces a simple formula for the complex reciprocal that is helpful for converting complex division into complex multiplication and a few real operations:

$$\frac{1}{z} = \frac{z^\star}{|z|^2}.$$

From the Cartesian form, these further relations are evident:

$$\begin{aligned} z &= z^\star, && \textit{if, and only if, } z \textit{ is real}, \\ \mathrm{real}(z) &= \tfrac{1}{2}(z + z^\star), \\ \mathrm{imag}(z) &= -\tfrac{1}{2}(z - z^\star)i. \end{aligned}$$

15.7 Complex conversion

The type-conversion macros defined on page 442 make it easy to provide companion functions:

```
void
CTOCX(fp_cx_t result, fp_c_t z)
{   /* convert native complex z to complex-as-real */
    CTOCX_(result, z);
}


fp_c_t
CXTOC(fp_cx_t z)
{   /* convert complex-as-real z to native complex */
    return (CXTOC_(z));
}
```

Those macros and functions reduce the need to reference the imaginary and real parts separately, shorten code that uses them, and make the type conversions explicit.

15.8 Complex copy

Because C does not support array assignment, we need a primitive for the job, so that user code can avoid referring to array subscripts, or individual components:

```
void
CXCOPY(fp_cx_t result, const fp_cx_t z)
{   /* complex copy: result = z */
    CXCOPY_(result, z);
}
```

The C99-style companion is unlikely to be needed, except for machine-assisted code translation, but we provide it for completeness:

```
fp_c_t
CCOPY(fp_c_t z)
{   /* complex copy: return z */
    return (z);
}
```

15.9 Complex division: C99 style

The most difficult operation in the complex primitives is division. If $x = a + ib$ and $y = c + id$, then complex division is defined by introducing a common factor in the numerator and denominator that reduces the complex denominator to a real number that can then divide each component:

$$\begin{aligned} x/y &= (a+bi)/(c+di) \\ &= \big((a+bi)(c-di)\big)/\big((c+di)(c-di)\big) \\ &= \big((ac+bd)+(bc-ad)i\big)/(c^2+d^2) \\ &= \big((ac+bd)/(c^2+d^2)\big) + \big((bc-ad)/(c^2+d^2)\big)i. \end{aligned}$$

In the last result, we can readily identify two serious problems for implementation with computer arithmetic of finite precision and range: significance loss in the additions and subtractions in the numerator, and premature overflow and underflow in both the numerator and the denominator.

There are more difficulties lurking, however. We also have to consider the possibility that one or more of the four components a, b, c, and d are Infinity, a NaN, or zero.

Infinities introduce problems, because IEEE 754 arithmetic requires that subtraction of like-signed Infinities, and division of Infinities, produce a NaN. Thus, even though we expect mathematically that $(1+i)/(\infty + i\infty)$ should evaluate to zero, the IEEE 754 rules applied to the definition of division produce a NaN result:

$$\begin{aligned} (1+i)/(\infty+i\infty) &= ((\infty+\infty)+(\infty-\infty)i)/(\infty^2+\infty^2) \\ &= (\infty + \mathrm{NaN}i)/\infty \\ &= \mathrm{NaN} + \mathrm{NaN}i. \end{aligned}$$

Division by zero is also problematic. Consider a finite numerator with positive parts. We then have three different results, depending on whether we divide by a complex zero, a real zero, or an imaginary zero:

$$\begin{aligned} (a+bi)/(0+0i) &= 0/0 + (0/0)i \\ &= \mathrm{NaN} + \mathrm{NaN}i, \\ (a+bi)/0 &= a/0 + (b/0)i \\ &= \infty + \infty i, \\ (a+bi)/(0i) &= b/0 - (a/0)i \\ &= \infty - \infty i. \end{aligned}$$

Thus, a complex-division routine that checks for zero real or imaginary parts to simplify the task to two real divides gets different answers from one that simply applies the expansion of x/y given at the start of this section.

Experiments with complex division in C99 and Fortran on various platforms show that their handling of Infinity is inconsistent. The ISO standards for those languages offer no guidance beyond a recommended algorithm for complex division in an informative annex of the C99 Standard [C99, §G.5.1, p. 469]. However, that annex notes in its introduction:

> *This annex supplements annex F to specify complex arithmetic for compatibility with IEC 60559 real floating-point arithmetic. Although these specifications have been carefully designed, there is little existing practice to validate the design decisions. Therefore, these specifications are not normative, but should be viewed more as recommended practice.*

The possibility of Infinity and NaN components could require extensive special casing in the division algorithm, as well as multiple tests for such components. That in turn makes the algorithm slower for all operands, even those that require no special handling.

To eliminate most of the overhead of special handling, the algorithm suggested in the C99 Standard follows the policy of *compute first, and handle exceptional cases later*, which the IEEE 754 nonstop model of computation easily supports. Older architectures may require additional coding, however, to achieve documented and predictable results for complex division.

Here is our implementation of the C99 algorithm for complex division:

```
void
CXDIV(fp_cx_t result, const fp_cx_t x, const fp_cx_t y)
{   /* complex division: result = x / y */
    fp_t a, b, c, d, logb_y, denom;
    fp_pair_t ac_bd, bc_ad;
    volatile fp_t e, f;
    int ilogb_y;

    ilogb_y = 0;
    a = CXREAL_(x);
    b = CXIMAG_(x);
    c = CXREAL_(y);
    d = CXIMAG_(y);

    logb_y = LOGB(FMAX(QABS(c), QABS(d)));

    if (ISFINITE(logb_y))
    {
        ilogb_y = (int)logb_y;
        c = SCALBN(c, -ilogb_y);
        d = SCALBN(d, -ilogb_y);
    }

    denom = c * c + d * d;
    fast_pprosum(ac_bd, a, c, b, d);
    fast_pprosum(bc_ad, b, c, -a, d);

    e = SCALBN(PEVAL(ac_bd) / denom, -ilogb_y);
    STORE(&e);
    f = SCALBN(PEVAL(bc_ad) / denom, -ilogb_y);
    STORE(&f);

    if (ISNAN(e) && ISNAN(f))
    {
        fp_t inf;

        /* Recover infinities and zeros that computed as NaN +
           I*NaN. The only cases are nonzero/zero, infinite/finite,
           and finite/infinite */
        if ((denom == ZERO) && (!ISNAN(a) || !ISNAN(b)))
        {
            inf = INFTY();
            e = COPYSIGN(inf, c) * a;
            f = COPYSIGN(inf, c) * b;
        }
        else if ((ISINF(a) || ISINF(b)) && ISFINITE(c) && ISFINITE(d))
        {
            inf = INFTY();
            a = COPYSIGN(ISINF(a) ? ONE : ZERO, a);
            b = COPYSIGN(ISINF(b) ? ONE : ZERO, b);
            e = inf * (a * c + b * d);
            f = inf * (b * c - a * d);
        }
        else if (ISINF(logb_y) && ISFINITE(a) && ISFINITE(b))
        {
            c = COPYSIGN(ISINF(c) ? ONE : ZERO, c);
            d = COPYSIGN(ISINF(d) ? ONE : ZERO, d);
            e = ZERO;
            f = ZERO;
            e *= (a * c + b * d);
            f *= (b * c - a * d);
        }
    }

    CXSET_(result, e, f);
}
```

The complex division algorithm is complicated, and several subtle points are worth noting:

- Premature overflow and underflow are avoided by scaling the denominator and the numerator. Mathematically, that could be done by dividing each component by the larger magnitude, $|c|$ or $|d|$. However, that division introduces into the scaled components at least one rounding error each, and for older architectures, even more, because division was often less accurate than addition and multiplication. The C99 algorithm trades execution time for better accuracy by scaling by the power of the base near the larger of $|c|$ and $|d|$. That scaling is *exact*, so no additional rounding error is introduced. The C99 `LOGB()` and `SCALBN()` functions provide the needed tools, and we can use them on all systems because they are also members of the `mathcw` library.
- Even though division is more expensive than multiplication, the computation of the result components, e and f, uses two divisions, rather than precomputing the reciprocal of `denom` and then using two multiplications. Doing so would introduce additional rounding errors that are unnecessary if we pay the cost of an extra division.
- The biggest loss of accuracy in the division comes from the product sums $ab + bd$ and $bc - ad$. The algorithm recommended by the C99 Standard computes them directly, but we replace that computation by calls to the functions `PPROSUM()` and `PEVAL()` for enhanced accuracy. We comment more on that problem later in **Section 15.13** on page 455 and **Section 15.16** on page 458.
- If the result components are finite or Infinity, or just one of them is a NaN, no further computation is needed.
- Otherwise, both components are a NaN, and three separate cases of corrective action are required, each of which involves from two to six property checks with the `ISxxx()` functions.

C99 does not provide a function for complex division, because that operation is built-in. For completeness, we provide a C99-style function that uses the code in `CXDIV()`:

```
fp_c_t
CDIV(fp_c_t x, fp_c_t y)
{   /* complex division: return x / y */
    fp_cx_t xx, yy, result;

    CTOCX_(xx, x);
    CTOCX_(yy, y);
    CXDIV(result, xx, yy);

    return (CXTOC_(result));
}
```

15.10 Complex division: Smith style

The first published algorithm for complex division known to this author is Robert L. Smith's *ACM Algorithm 116* [Smi62], which addresses the overflow and underflow problems by scaling the denominator by the larger of its two components. If $|c| \geq |d|$, we rewrite the division given at the start of the previous section with two intermediate variables r and s like this:

$$x/y = (a + bi)/(c + di)$$

$$\begin{aligned}
&= \big((ac+bd)/(c^2+d^2)\big) + \big((bc-ad)/(c^2+d^2)\big)i,\\
&= \big((a+bd/c)/(c+d^2/c)\big) + \big((b-ad/c)/(c+d^2/c)\big)i,\\
r &= d/c,\\
s &= dr + c,\\
x/y &= \big((a+br)/s\big) + \big((b-ar)/s\big)i.
\end{aligned}$$

Otherwise, when $|c| < |d|$, similar steps produce

$$\begin{aligned}
r &= c/d,\\
s &= cr + d,\\
x/y &= \big((ar+b)/s\big) + \big((br-a)/s\big)i,
\end{aligned}$$

The total floating-point work is 2 `fabs()` operations, 1 compare, 3 adds, 3 divides, and 3 multiplies.

The inexact scaling contaminates both components of the result with an additional rounding error. With somewhat more work, that scaling can be made exact [LDB+00, Appendix B, page 61], but we do not discuss it here because we can do even better.

The fused multiply-add operation was not invented until much later, but it clearly could be of use here, speeding the calculations, and largely eliminating subtraction loss.

15.11 Complex division: Stewart style

About two decades after *ACM Algorithm 116* [Smi62], G. W. Stewart revisited Smith's method and pointed out that premature overflow and underflow can be made less likely by rearranging the computation to control the size of intermediate products [Ste85]. The revised algorithm requires additional control logic, and looks like this when rewritten to use the same variables for real and imaginary parts as we have in `CXDIV()`:

```
#include <stdbool.h>

#define SWAP(x,y)       (temp = x, x = y, y = temp)

void
stewart_cxdiv(fp_cx_t result, const fp_cx_t z, const fp_cx_t w)
{   /* complex-as-real division: set result = z / w */
    fp_t a, b, c, d, e, f, s, t, temp;
    bool flip;

    a = CXREAL_(z);
    b = CXIMAG_(z);
    c = CXREAL_(w);
    d = CXIMAG_(w);
    flip = false;

    if (QABS(d) > QABS(c))
    {
        SWAP(c, d);
        SWAP(a, b);
        flip = true;
    }

    s = ONE / c;
    t = ONE / (c + d * (d * s));

    if (QABS(d) > QABS(s))
        SWAP(d, s);
    if (QABS(b) >= QABS(s))
        e = t * (a + s * (b * d));
    else if (QABS(b) >= QABS(d))
        e = t * (a + b * (s * d));
    else
        e = t * (a + d * (s * b));

    if (QABS(a) >= QABS(s))
        f = t * (b - s * (a * d));
    else if (QABS(a) >= QABS(d))
        f = t * (b - a * (s * d));
    else
        f = t * (b - d * (s * a));

    if (flip)
        f = -f;

    CXSET_(result, e, f);
}
```

In the first `if` block, Stewart exploits the complex-conjugation symmetry rule for division:

$$\begin{aligned}\big((a+bi)/(c+di)\big)^\star &= (a+bi)^\star/(c+di)^\star \\ &= (a-bi)/(c-di), && \textit{then multiply by } i/i, \\ &= (b+ai)/(d+ci).\end{aligned}$$

If necessary, the real and imaginary components are swapped to ensure that $|c| \geq |d|$, and the `flip` variable records that action. Subsequent operations then compute the conjugate of the desired result, and the sign is inverted in the final `if` statement.

Stewart's code requires parentheses to be obeyed, but as we recorded in **Section 4.4** on page 64, that was not true in C before the 1990 ISO Standard.

The floating-point operation count is 8 to 12 `QABS()` tests, 4 to 6 compares, 3 adds, 2 divides, 8 multiplies, and possibly 1 negation. Unless divides are exceptionally slow, Stewart's algorithm is likely to be somewhat slower than Smith's, but it has better numerical behavior at the extremes of the floating-point range.

Extensive tests of both Smith's and Stewart's algorithms on several platforms against more accurate code for complex division show that, with millions of random arguments, the relative error of the quotient lies below 3.0 ulps. However, it is certainly possible with specially chosen components to exhibit cases that suffer catastrophic subtraction loss.

Although we do not show the needed code, special handling of Infinity and NaN arguments is required, because both Smith's and Stewart's algorithms produce NaN results, instead of Infinity, for ∞/finite and zero for finite/∞.

15.12 Complex division: Priest style

Two decades more passed before complex division was again revisited. In a lengthy article with complicated numerical analysis [Pri04], Douglas Priest shows that the inexact scaling in the Smith [Smi62] and Stewart [Ste85] methods can be eliminated *without* the overhead of the `logb()` and `scalbn()` calls used in the C99 algorithm, provided that the programmer is willing to commit to a particular floating-point format, and grovel around in the bits of the floating-point representation. Priest's detailed examination shows that there is some flexibility in the choice of scale factor, that absolute-value operations can be eliminated by bit masking, and that a two-step scaling can eliminate *all* premature underflow and overflow.

The major difficulty for the programmer is decoding the floating-point representation and choosing suitable bit masks and constants to accomplish the scaling. Priest exhibits code only for IEEE 754 64-bit arithmetic, and hides the architecture-dependent byte storage order by assuming that `long long int` is a supported 64-bit integer data type. With the same variable notation as before, here is what his code looks like:

```
void
priest_cxdiv(cx_double result, const cx_double z, const cx_double w)
{   /* set result = z / w */
    union
    {
        long long int i;    /* must be 64-bit integer type */
        double d;           /* must be 64-bit IEEE 754 type */
    } aa, bb, cc, dd, ss;
    double a, b, c, d, t;
    int ha, hb, hc, hd, hz, hw, hs; /* components of z and w */

    a = CXREAL_(z);
    b = CXIMAG_(z);
    c = CXREAL_(w);
    d = CXIMAG_(w);
    aa.d = a; /* extract high-order 32 bits to estimate |z| and |w| */
    bb.d = b;
    ha = (aa.i >> 32) & 0x7fffffff;
    hb = (bb.i >> 32) & 0x7fffffff;
    hz = (ha > hb) ? ha : hb;
    cc.d = c;
    dd.d = d;
    hc = (cc.i >> 32) & 0x7fffffff;
    hd = (dd.i >> 32) & 0x7fffffff;
    hw = (hc > hd) ? hc : hd;   /* compute the scale factor */

    if (hz < 0x07200000 && hw >= 0x32800000 && hw < 0x47100000)
    {                   /* |z| < 2^-909 and 2^-215 <= |w| < 2^114 */
        hs = (((0x47100000 - hw) >> 1) & 0xfff00000) + 0x3ff00000;
    }
    else
        hs = (((hw >> 2) - hw) + 0x6fd7ffff) & 0xfff00000;

    ss.i = (long long int)hs << 32; /* scale c & d, & get quotient */
    c *= ss.d;
    d *= ss.d;
    t = ONE / (c * c + d * d);
    c *= ss.d;
    d *= ss.d;

    CXSET_(result, (a * c + b * d) * t, (b * c - a * d) * t);
}
```

The variables `ha` through `hd` hold the top 32 bits of the four components with the sign bits masked to zero, and the larger of each pair are then used to determine `hs`, which has the top 32 bits of the scale factor. That scale factor is an exact power of the base, and is constructed in the structure element `ss.i`, and used as its memory overlay `ss.d`. The scale-factor selection is intricate and takes three journal pages to describe; see [Pri04, §2.2] for the details.

Determining the scale factor requires only fast 32-bit and 64-bit integer operations, and once it is available, the final result is constructed with 3 adds, 1 divide, and 12 multiplies.

Priest observes that the product-sums in the last statement are subject to catastrophic subtraction loss, but does not attempt to correct that problem.

Instead of laboriously deriving new scale-factor code for a `float` version of Priest's method for complex division, it is more sensible to promote the `float` operands to `double` and call `priest_cxdiv()`, since all internal products are then exact, and subtraction loss is eliminated. The `long double` type is more difficult to handle, since it too needs new scale-factor code, and there are 80-, 96-, and 128-bit storage conventions for the 80-bit type, a paired-double 128-bit format, and a separate 128-bit representation, plus differing byte-addressing practices.

Priest claims that his algorithm also correctly handles the case of complex infinite operands: when the exact result is infinite, at least one component of the result is Infinity, and the other may be a NaN, as permitted by C99 and noted at the beginning of this chapter.

Timing tests on several architectures with arguments of random signs, and magnitudes drawn from both uniform and logarithmic distributions, show that Priest's algorithm is always faster than Stewart's, and is much faster than the C99 algorithm that we present in **Section 15.9** on page 449.

15.13 Complex division: avoiding subtraction loss

The problem of catastrophic subtraction loss remains in the four algorithms for complex division ([C99, §G.5.1, p. 469], [Smi62], [Ste85], and [Pri04]) that we have presented in the preceding sections. It is a requirement of IEEE 754 arithmetic that results of the five basic operations of real arithmetic are always correctly rounded. Even though complex arithmetic is more difficult than real arithmetic, a library implementation of the basic complex operations should guarantee *relative errors* that are no worse than a few units in the last place for *all possible* operands.

In each algorithm, subtraction loss lurks in the expression forms $ab + cd$ and $ab + c$. In **Section 15.9** on page 449, we proposed handling the first form with our pair-precision product-sum function family, `PPROSUM()`. In **Section 15.10** on page 451, we suggested using the `FMA()` fused multiply-add family for the second form.

When we recall that each product can be represented *exactly* as a sum of pairs, then we can apply our `VSUM()` primitive for accurate vector summation:

```
fp_t v[4], result;
v[3] = a * b;                       /* hi(a * b) */
v[2] = c * d;                       /* hi(c * d) */
v[1] = FMA(a, b, -v[3]);            /* lo(a * b) */
v[0] = FMA(c, d, -v[2]);            /* lo(c * d) */
result = VSUM((ft_t)NULL, 4, v);    /* a * b + c * d, accurately */
```

When a fast `fma()` operation is available, the problem expression can, and should, be computed that way. However, when the `fma()` function is comparatively slow, it is better to use it only when subtraction loss is known to happen: in a binary base, the terms must be of opposite sign, and ratio of their magnitudes must lie in $[\frac{1}{2}, \frac{3}{2}]$ (see **Section 4.19** on page 89).

We therefore replace the call to `PPROSUM()` with a call to this faster version:

```
static void
fast_pprosum(fp_pair_t result, fp_t a, fp_t b, fp_t c, fp_t d)
{   /* compute result = a * b + c * d accurately and quickly */
    fp_t ab, ab_abs, cd, cd_abs;

    ab = a * b;
    cd = c * d;
    result[1] = ZERO;

    if ((ab >= ZERO) && (cd >= ZERO))           /* same signs */
        result[0] = ab + cd;
    else if ((ab < ZERO) && (cd < ZERO))        /* same signs */
        result[0] = ab + cd;
    else                                        /* opposite signs */
    {
        ab_abs = QABS(ab);
        cd_abs = QABS(cd);

        if ( ((cd_abs + cd_abs) < ab_abs) || ((ab_abs + ab_abs) < cd_abs) )
            result[0] = ab + cd;
        else                                    /* certain loss */
        {
            fp_t err_ab;
            err_ab = FMA(a, b, -ab);
            result[0] = FMA(c, d, ab);
            result[1] = err_ab;
        }
    }
}
```

In that code, PSET_() is a macro that expands inline to set both components of its first argument, without worrying about the sign of zero in the second component, as the function PSET() does.

15.14 Complex imaginary part

The imaginary part of a complex number is just its second component, so retrieving it is simple:

```
fp_t
CXIMAG(const fp_cx_t z)
{   /* complex imaginary part: return imag(z) */
    return (CXIMAG_(z));
}
```

The C99 companion function exploits the storage mandate cited earlier on page 441 to cast a complex-value pointer to an array-element pointer via our conversion macro:

```
fp_t
CIMAG(fp_c_t z)
{   /* complex imaginary part: return imag(z) */
    return (CXIMAG_((fp_t *)&z));
}
```

15.15 Complex multiplication

After division, the next most difficult operation in the complex primitives is multiplication. If $x = a + ib$ and $y = c + id$, then complex multiplication is defined like this:

$$\begin{aligned} xy &= (a + ib)(c + id) \\ &= (ac - bd) + i(ad + bc). \end{aligned}$$

That looks straightforward, but as happens with division, the problems of significance loss and premature overflow and underflow, and the introduction of spurious NaN results from subtraction and division of Infinity, must be dealt with.

Our algorithm follows the procedure recommended in a non-binding annex of the C99 Standard [C99, §G.5.1, p. 468], and like the division algorithm, it computes first, and handles exceptional cases later:

```
void
CXMUL(fp_cx_t result, const fp_cx_t x, const fp_cx_t y)
{   /* complex multiply: result = x * y */
    fp_t a, b, c, d;
    fp_pair_t ac_bd, ad_bc;

    a = CXREAL_(x);
    b = CXIMAG_(x);
    c = CXREAL_(y);
    d = CXIMAG_(y);

    PPROSUM(ac_bd, a, c, -b, d);
    PPROSUM(ad_bc, a, d, b, c);
```

```
CXSET_(result, PEVAL(ac_bd), PEVAL(ad_bc));

if (ISNAN(CXREAL_(result)) && ISNAN(CXIMAG_(result)))
{
    int recalc;

    recalc = 0;

    if ( ISINF(a) || ISINF(b) )     /* x is infinite */
    {   /* Box the infinity and change NaNs in other factor to 0 */
        a = COPYSIGN(ISINF(a) ? ONE : ZERO, a);
        b = COPYSIGN(ISINF(b) ? ONE : ZERO, b);

        if (ISNAN(c))
            c = COPYSIGN(ZERO, c);

        if (ISNAN(d))
            d = COPYSIGN(ZERO, d);

        recalc = 1;
    }

    if  ( ISINF(c) || ISINF(d) )    /* y is infinite */
    {    /* Box infinity and change NaNs in other factor to 0 */
        c = COPYSIGN(ISINF(c) ? ONE : ZERO, c);
        d = COPYSIGN(ISINF(d) ? ONE : ZERO, d);

        if (ISNAN(a))
            a = COPYSIGN(ZERO, a);

        if (ISNAN(b))
            b = COPYSIGN(ZERO, b);

        recalc = 1;
    }

    if (!recalc && (ISINF(a * c) || ISINF(b * d) || ISINF(a * d) || ISINF(b * c)))
    {   /* Recover infinities from overflow: change NaNs to zero */
        if (ISNAN(a))
            a = COPYSIGN(ZERO, a);

        if (ISNAN(b))
            b = COPYSIGN(ZERO, b);

        if (ISNAN(c))
            c = COPYSIGN(ZERO, c);

        if (ISNAN(d))
            d = COPYSIGN(ZERO, d);

        recalc = 1;
    }

    if (recalc)
    {
        fp_t inf;
```

```
            inf = INFTY();
            CXSET_(result, inf * ( a * c - b * d ),
                           inf * ( a * d + b * c ));
        }
    }
}
```

The code deals with the common case quickly, but when both components of the result are found to be NaNs, further testing, and possible computation of properly signed infinities, is needed. We improve upon the recommended algorithm by using `PPROSUM()` and `PEVAL()` for the computation of $ac - bd$ and $ad + bc$. In practice, we can use our `fast_pprosum()` private function (see **Section 15.13** on page 455) instead of `PPROSUM()`.

As with division, C99 does not provide a function for complex multiplication, because that operation too is built-in. Following our practice with `CDIV()`, we provide a C99-style function that uses the code in `CXMUL()`:

```
fp_c_t
CMUL(fp_c_t x, fp_c_t y)
{   /* complex multiply: return x * y */
    fp_cx_t xx, yy, result;

    CTOCX_(xx, x);
    CTOCX_(yy, y);
    CXMUL(result, xx, yy);

    return (CXTOC_(result));
}
```

15.16 Complex multiplication: error analysis

The accuracy of multiplication with complex arithmetic has been studied with detailed mathematical proofs that occupy about ten journal pages [BPZ07]. That work has been recently extended to algorithms using fused multiply-add operations [JKLM17], slightly improving the bounds cited here. The authors show that for floating-point base β under the conditions

- subnormals, overflow, and underflow are avoided,
- *round-to-nearest* mode is in effect for real arithmetic operations,
- the number of significand bits is at least five, and
- expressions of the form $ab \pm cd$ are computed accurately,

then complex multiplication has a maximum relative error below $\frac{1}{2}\sqrt{5}\beta^{1-t}$, where t is the number of significand digits. For binary IEEE 754 arithmetic, their bound corresponds to 1.118 ulps. Importantly, their analysis leads to simple test values that produce the *worst-case* errors:

$$\begin{aligned}
\beta &= 2,\\
t &= 24 \text{ or } 53,\\
e &= \tfrac{1}{2}\beta^{1-t},\\
z_0 &= 3/4 + \big((3(1-4e))/4\big)i, && \textit{32-bit IEEE 754 format,}\\
&= \frac{3}{4} + \frac{12\,582\,909}{16\,777\,216}i\\
&= \texttt{0x1.8p-1 + 0x1.7fff\_fap-1 * I},\\
z_1 &= \big(2(1+11e)\big)/3 + \big((2(1+5e))/3\big)i
\end{aligned}$$

$$
\begin{aligned}
&= \frac{5\,592\,409}{8\,388\,608} + \frac{5\,592\,407}{8\,388\,608} i \\
&= \texttt{0x1.5555\_64p-1 + 0x1.5555\_5cp-1 * I,} \\
\text{exact } z_0 z_1 &= (5e + 10e^2) + (1 + 6e - 22e^2)i \\
&= \frac{41\,943\,045}{140\,737\,488\,355\,328} + \frac{140\,737\,538\,686\,965}{140\,737\,488\,355\,328} i \\
&= \texttt{0x1.4000\_028p-22 + 0x1.0000\_05ff\_ffeap+0 * I,} \\
w_0 &= \big(3(1+4e)\big)/4 + (3/4)i, \qquad \textit{64-bit IEEE 754 format,} \\
&= \frac{6\,755\,399\,441\,055\,747}{9\,007\,199\,254\,740\,992} + \frac{3}{4} i \\
&= \texttt{0x1.8000\_0000\_0000\_3p-1 + 0x1.8p-1 * I,} \\
w_1 &= \big(2(1+7e)\big)/3 + \big(2(1+e)/3\big)i \\
&= \frac{3\,002\,399\,751\,580\,333}{4\,503\,599\,627\,370\,496} + \frac{3\,002\,399\,751\,580\,331}{4\,503\,599\,627\,370\,496} i \\
&= \texttt{0x1.5555\_5555\_5555\_ap-1 +} \\
&\quad \texttt{0x1.5555\_5555\_5555\_6p-1 * I,} \\
\text{exact } w_0 w_1 &= (5e + 14e^2) + (1 + 6e + 2e^2)i \\
&= \frac{22\,517\,998\,136\,852\,487}{40\,564\,819\,207\,303\,340\,847\,894\,502\,572\,032} + \\
&\quad \frac{40\,564\,819\,207\,303\,367\,869\,492\,266\,795\,009}{40\,564\,819\,207\,303\,340\,847\,894\,502\,572\,032} i \\
&= \texttt{0x1.4000\_0000\_0000\_1cp-51 +} \\
&\quad \texttt{0x1.0000\_0000\_0000\_3000\_0000\_0000\_008p+0 * I.}
\end{aligned}
$$

Despite the factors of $2/3$ in the expressions for z_1 and w_1, all of the components are *exactly representable* in binary arithmetic.

Using those worst-case values for the `float` and `double` formats, this author wrote two short test programs, `tcmul2.c` and `tcmul3.c`, in the `exp` subdirectory. The first uses native complex arithmetic and is run with the native math library. The second replaces the complex multiplications by calls to our `CMUL()` family members, and thus requires the `mathcw` library. The programs were then run on several architectures, including GNU/LINUX (Alpha, AMD64, IA-32, IA-64, PowerPC, and SPARC), FREEBSD (IA-32), OPENBSD (IA-32), and SOLARIS (AMD64 and SPARC), with multiple C99-level compilers.

All of the tests show relative errors below 0.539 ulps for the 32-bit native complex multiply. For the 64-bit native complex multiply, all systems produce a correctly rounded imaginary part, but *almost all* of the test systems lose 49 of the 53 significand bits for the real part of the product! Only on the FREEBSD IA-32 and SOLARIS AMD64 tests, and with one uncommon commercial compiler on GNU/LINUX AMD64, is the real part correctly rounded. By contrast, the test program that uses the mathcw library routines produces correctly rounded results for those tests on all platforms. The lesson is that *complex arithmetic in C is not yet trustworthy*.

15.17 Complex negation

The negative of a complex number $x + iy$ is just $-x - iy$, so the implementation is simple:

```
void
CXNEG(fp_cx_t result, const fp_cx_t z)
{   /* complex negation: result = -z */
    CXSET_(result, -CXREAL_(z), -CXIMAG_(z));
}
```

There is no C99 Standard function for complex negation, because the operation is built-in, but we provide a companion function that uses inline code, rather than calling `CXNEG()`:

```
fp_c_t
CNEG(fp_c_t z)
{   /* complex negation: return -z */
    return (-z);
}
```

15.18 Complex projection

The projection function may be unique to C99, and the Standard describes it this way [C99, §7.3.9.4, p. 179]:

> *The* `cproj()` *functions compute a projection of* `z` *onto the Riemann*[2] *sphere:* `z` *projects to* `z` *except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If* `z` *has an infinite part, then* `cproj(z)` *is equivalent to*
>
> *`INFINITY + I * copysign(0.0, cimag(z)).`*

That description readily leads to obvious code:

```
void
CXPROJ(fp_cx_t result, const fp_cx_t z)
{   /* complex projection of z onto Riemann sphere: result = proj(z) */
    if (ISINF(CXREAL_(z)) || ISINF(CXIMAG_(z)))
        CXSET_(result, INFTY(), COPYSIGN(ZERO, CXIMAG_(z)));
    else
        CXSET_(result, CXREAL_(z), CXIMAG_(z));
}
```

The C99 companion function uses `CXPROJ()` for the real work:

```
fp_c_t
CPROJ(fp_c_t z)
{   /* complex projection of z onto Riemann sphere: return proj(z) */
    fp_cx_t zz, result;

    CTOCX_(zz, z);
    CXPROJ(result, zz);

    return (CXTOC_(result));
}
```

If a sphere of radius one is centered at the origin on the complex plane (see **Figure 15.2**), then a line from any point $w = (u, v)$ on the plane outside the sphere to the North Pole of the sphere intersects the sphere in exactly two places, like an arrow shot through a balloon. Thus, every finite point (u, v) has a unique image point on the sphere at the first intersection. As we move further away from the origin, that intersection point approaches the North Pole. The two intersections with the sphere coincide only for points where one or both components of the point on the plane are infinitely far away. That is, all complex infinities project onto a single point, the North Pole, of the Riemann sphere.

15.19 Complex real part

The real part of a complex number is just its first component, so retrieving it is easy:

```
fp_t
CXREAL(const fp_cx_t z)
{   /* complex real part: return real(z)  */
    return (CXREAL_(z));
}
```

[2]See the footnote in **Section 11.2** on page 303.

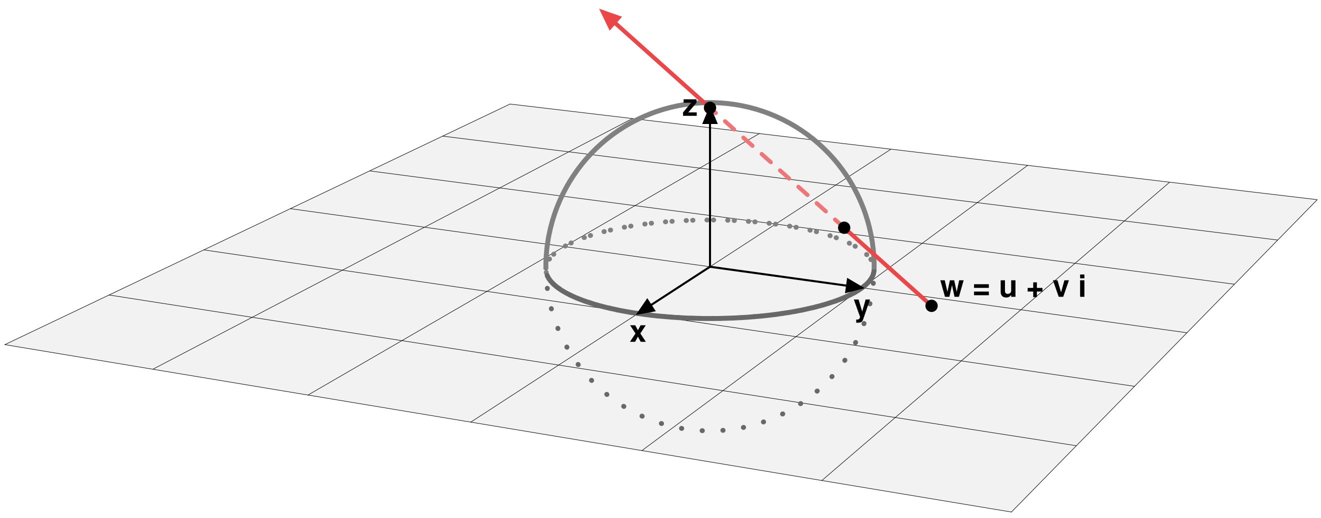

Figure 15.2: Projecting a complex point onto the Riemann sphere.

As with `CIMAG()`, the corresponding C99 function for the real part uses the storage requirement to convert a complex-value pointer to an array-element pointer:

```
fp_t
CREAL(fp_c_t z)
{   /* complex real part: return real(z) */
    return (CXREAL_((fp_t *)&z));
}
```

15.20 Complex subtraction

The subtraction operation for complex numbers simply requires computing the difference of their real and imaginary components, so the code is obvious:

```
void
CXSUB(fp_cx_t result, const fp_cx_t x, const fp_cx_t y)
{   /* complex subtraction: result = x - y */
    CXSET_(result, CXREAL_(x) - CXREAL_(y), CXIMAG_(x) - CXIMAG_(y));
}
```

There is no C99 function for complex subtraction, because it is a built-in operation, but we can easily provide a C99-style function. Because the operation is so simple, we code it directly, rather than calling `CXSUB()` to do the work:

```
fp_c_t
CSUB(fp_c_t x, fp_c_t y)
{   /* complex subtraction: return x - y */
    return (x - y);
}
```

15.21 Complex infinity test

The peculiar C99 definition of complex infinite values cited on page 441 suggests that we provide a primitive for testing for a complex infinity, even though the ISO Standard does not specify test functions for complex types. The code for our two families of complex types is short:

```
int
ISCXINF(const fp_cx_t z)
{   /* return 1 if z is a complex Infinity, else 0 */
    return (ISINF(CXREAL_(z)) || ISINF(CXIMAG_(z)));
}
```

```
int
ISCINF(fp_c_t z)
{   /* return 1 if z is a complex Infinity, else 0 */
    return (ISINF(CREAL(z)) || ISINF(CIMAG(z)));
}
```

15.22 Complex NaN test

The C99 definition of complex infinite values complicates NaN tests, so we extend the ISO Standard with test functions to hide the mess. Their code looks like this:

```
int
ISCXNAN(const fp_cx_t z)
{   /* return 1 if z is a complex NaN, else 0 */
    fp_t x, y;
    int result;

    x = CXREAL_(z);
    y = CXIMAG_(z);

    if (ISINF(x) || ISINF(y))
        result = 0;
    else if (ISNAN(x) || ISNAN(y))
        result = 1;
    else
        result = 0;

    return (result);
}
```

```
int
ISCNAN(fp_c_t z)
{   /* return 1 if z is a complex NaN, else 0 */
    fp_cx_t zz;

    CTOCX_(zz, z);
    return (ISCXNAN(zz));
}
```

15.23 Summary

At the time of writing this, support for complex arithmetic in compilers for the C language family is limited, and on many systems, of doubtful quality. Consequently, the functions for complex arithmetic described in this chapter have not received extensive use.

Most of the functions are simple, and good candidates for inline expansion by compilers. We encouraged such expansion by using the macros `CXIMAG_()`, `CXREAL_()`, `CXSET_()`, `CXTOC_()` and `CTOCX_()` to produce inline code instead of a function call, and conceal all array-element references.

The functions for complex absolute value and complex argument are numerically troublesome, but can easily be expressed in terms of the `HYPOT()` and `ATAN2()` families which are carefully implemented to produce high accuracy.

The functions for multiplication and division are subject to premature underflow and overflow, and to massive subtraction loss. We discussed four different algorithms for complex division, and showed how careful handling of expressions of the forms $a \times b + c \times d$ and $a \times b + c$ with the occasional help of fused multiply-add operations can eliminate subtraction loss in both complex division and complex multiplication. Since the mathcw package and this book were completed, several additional papers on the problem of accurate complex multiplication and division have appeared [BS12, WE12, JLM13b, JLM13a, Mul15, Jea16, JLMP16, JKLM17], but comparison with our code remains a project for future study.

The last of those papers also points out an issue that we have not treated in this chapter: computer algorithms for complex multiplication may not obey the expected mathematical property of commutativity $(w \times z \equiv z \times w)$, and thus, may incorrectly produce a nonzero imaginary part in the computation $z \times z^{\star} = (x + yi) \times (x - yi) = x^2 - x \times yi + y \times xi + y^2 = x^2 + y^2$, a value that, mathematically, is purely real.

The difficulties in producing fast and correctly rounded complex multiply and divide cry out for a hardware solution. If hardware provided multiple functional units with extended range and double-width results, then complex absolute value, complex division, and complex multiplication could be done quickly, and without premature underflow or overflow, or unnecessary subtraction loss. Sadly, only a few historical machines seem to have addressed that need:

- Bell Laboratories Model 1 through Model 4 relay computers with fixed-point decimal arithmetic (1938–1944) (ten digits, only eight displayed) [Sti80],
- Bell Laboratories Model 5 relay computer with seven-digit floating-point decimal arithmetic (1945) [Sti80], and
- Lawrence Livermore National Laboratory S-1 (see **Appendix H.7**),

No current commercially significant computer architecture provides complex arithmetic in hardware, although there are several recent papers in the chip-design literature on that subject.

Unlike Fortran, C99 does not offer any native I/O support for complex types. The real and imaginary parts must be handled explicitly, forcing complex numbers to be constructed and deconstructed by the programmer for use in calls to input and output functions for real arithmetic. Maple provides a useful extension in `printf()` format specifiers that could be considered for future C-language standardization: the letter `Z` is a numeric format modifier character that allows a single format item to produce two outputs, like this:

```
% maple
> printf("%Zg\n", 1 + 2*I);
1+2I
> printf("%5.1Zf\n", 1 + 2*I);
  1.0 +2.0I
```

The `_Imaginary` data type is a controversial feature of C99, and its implementation by compilers and libraries was therefore made optional. That decision makes the type practically unusable in portable code. Although mathematicians work with numbers on the real axis, and numbers in the complex plane, they rarely speak of numbers that are restricted to the imaginary axis. More than four decades of absence of the imaginary data type from Fortran, the only widely used, and standardized, language for complex arithmetic, suggests that few programmers will find that type useful. The primary use of an imaginary type may be to ensure that an expression like `z * I` is evaluated without computation as `-cimag(z) + creal(z) * I` to match mathematics use, instead of requiring an explicit complex multiplication that gets the wrong answer when one or both of the components of `z` is a negative zero, Infinity, or NaN.

We provided complex companions for the real `ISxxx()` family only for Infinity and NaN tests, because they are nontrivial, and likely to be programmed incorrectly if done inline. Most of the relational operations do not apply to complex values, and it is unclear whether tests for complex finite, normal, subnormal, and zero values are useful. We have not missed them in writing any of the complex-arithmetic routines in the `mathcw` library. Should they prove desirable in some applications, they could easily be generated inline by private macros like these:

```
#define ISCFINITE(z)     ISFINITE(CREAL(z))      && ISFINITE(CIMAG(z))
#define ISCNORMAL(z)     ISNORMAL(CREAL(z))      && ISNORMAL(CIMAG(z))
#define ISCSUBNORMAL(z)  ISSUBNORMAL(CREAL(z))   && ISSUBNORMAL(CIMAG(z))
#define ISCZERO(z)       (CREAL(z) == ZERO)      && (CIMAG(z) == ZERO)

#define ISCXFINITE(z)    ISFINITE(CXREAL_(z))    && ISFINITE(CXIMAG_(z))
#define ISCXNORMAL(z)    ISNORMAL(CXREAL_(z))    && ISNORMAL(CXIMAG_(z))
#define ISCXSUBNORMAL(z) ISSUBNORMAL(CXREAL_(z)) && ISSUBNORMAL(CXIMAG_(z))
#define ISCXZERO(z)      (CXREAL_(z) == ZERO)    && (CXIMAG_(z) == ZERO)
```

However, the programmer needs to decide what should be done about cases where one component is, say, subnormal, and the other is not. For example, it might be desirable to change && to `||` in the definition of `ISCXSUBNORMAL()`.

We defer presentation of the computation of complex versions of the elementary functions required by C99 to **Chapter 17** on page 475, because now is a good time to apply the primitives of this chapter to one of the simplest problems where complex arithmetic is required: solution of quadratic equations, the subject of the next chapter.