Complex Arithmetic for IEEE Implementations

An extension to Annex X: IEEE standard floating-point arithmetic

WG14/N518 X3J11/95-119 (Draft 12/21/95)

Jim Thomas
Taligent, Inc.
10201 N. DeAnza Blvd.
Cupertino, CA 95014-2233
jim_thomas@taligent.com

This is a proposal to extend the annex proposed as part of the integration of FPCE into C9X in order to specify complex arithmetic for IEEE implementations.

X.11 Complex arithmetic

X.11.1 Binary operators

For most operand types, the result of a binary operator with an imaginary or complex operand is completely determined, with reference to real arithmetic, by the usual mathematical formula. For some operand types, the usual mathematical formula is problematic because of its treatment of infinities and because of undue overflow or underflow; in these cases the results are required to satisfy certain properties, but are not completely determined.

> [Calculated with the usual mathematical formula for multiplication, the result of raising any infinity to the fourth power has NaNs in both parts. The sum-of-squares denominator in the usual mathematical formula for division is particularly prone to overflow and underflow.]

X.11.1.1 Multiplication operators

Semantics

If the operands are not both complex, then the result of the * operator is defined by the usual mathematical formula:

*	real x	imaginary $y*I$	complex $x + y*I$
real u	$x*u$	$(y*u)*I$	$(x*u) + (y*u)*I$
imaginary $v*I$	$(x*v)*I$	$-y*v$	$(-y*v) + (x*v)*I$
complex $u + v*I$	$(x*u) + (x*v)*I$	$(-y*v) + (y*u)*I$	

If the second operand is not complex, then the result of the / operator is defined by the usual mathematical formula:

/	x	y*I	x + y*I
u	x/u	(y/u)*I	(x/u) + (y/u)*I
v*I	(-x/v)*I	y/v	(y/v) + (-x/v)*I

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. The * and / operators satisfy the following infinity properties for all real, imaginary, and complex operands[1]:

- If one operand is an infinity and the other operand is a nonzero finite number, then the result of the * operator is an infinity.
- If the first operand is an infinity and the second operand is a finite number, then the result of the / operator is an infinity.
- If the first operand is a finite number and the second operand is an infinity, then the result of the / operator is a zero.
- If the first operand is a nonzero number (and not a NaN) and the second operand is a zero, then the result of the / operator is an infinity.

[These properties, together with the `proj` function in `<complex.h>` and the user paradigm of ignoring differences in infinities, largely supports the 1-infinity, Riemann sphere model for complex numbers.]

[1] These properties are already implied for those cases covered in the tables, but are required for all cases.

Examples

1. Multiplication of **double complex** operands could be implemented with

```
#include <math.h>
#include <complex.h>
#define isinf(x) (fabs(x)==INFINITY)

// To "box" infinities ...
static double complex box(double complex z)
{
   return copysign(isinf(real(z)) ? 1.0 : 0.0, real(z)) +
      I * copysign(isinf(imag(z)) ? 1.0 : 0.0, imag(z));
}

// Multiply z * w ...
double complex _Cmultd(double complex z, double complex w)
{
   double a, b, c, d, x, y;
   a = real(z); b = imag(z); c = real(w); d = imag(w);
   x = a * c - b * d;
   y = a * d + b * c;
   if (isnan(x) && isnan(y))
   {
      int recalc = 0;
      if (isinf(a) || isinf(b))  // if z is infinite
      {
         double complex zz;
         zz = box(z);
         a = real(zz);
         b = imag(zz);
         recalc = 1;
      }
      if (isinf(c) || isinf(d))  // if w is infinite
      {
         double complex ww;
         ww = box(w);
         c = real(ww);
         d = imag(ww);
         recalc = 1;
      }
      if (recalc)
      {
         x = INFINITY * (a * c - b * d);
         y = INFINITY * (a * d + b * c);
      }
   }
   return x + I * y;
}
```

In ordinary (finite) cases, the cost to satisfy the infinity property for the * operator is only one **isnan** test. This implementation opts for performance over guarding against undue overflow and underflow.

171

For multiplication, whose speed is critical for certain important, large matrix problems, which are know not to encounter infinities, even the one isnan test is expected to be too costly. This problem could be solved by a compiler switch, say cx_infinities, roughly in the style of fp_contract, whose off state would allow unspecified treatment of infinities.

[Undue overflow for multiplication is less a problem than for division, because it occurs in more limited cases. Generally, undue underflow from complex multiplication is not a serious problem. Ideally, all multiplications and divisions would be correctly rounded, as are the operations specified in the tables; however, the costs would be substantial, in the absence of special hardware support.]

2. Division of two **double complex** operands could be implemented with

```
#include <math.h>
#include <complex.h>
// box and isinf are as in example 1 above

// Divide z / w ...
double complex _Cdivd(double complex z, double complex w)
{
   double a, b, c, d, logbw, denom, x, y;
   long llogbw = 0;
   a = real(z); b = imag(z); c = real(w); d = imag(w);
   logbw = logb(fmax(fabs(c), fabs(d)));
   if (isfinite(logbw))
   {
      llogbw = (long)logbw;
      c = scalb(c, -llogbw);
      d = scalb(d, -llogbw);
   }
   denom = c * c + d * d;
   x = scalb((a * c + b * d) / denom, -llogbw);
   y = scalb((b * c - a * d) / denom, -llogbw);
   if (isnan(x) && isnan(y))
   {
      double fact = 1;
      int recalc = 0;
      if (isinf(a) || isinf(b))  // if z is infinite
      {
         double complex zz;
         zz = box(z);
         fact = INFINITY;
         a = real(zz);
         b = imag(zz);
         recalc = 1;
      }
```


172

```
        if (logbw == INFINITY)  // if w is infinite
        {
            double complex ww;
            ww = box(w);
            fact /= INFINITY;
            c = real(ww);
            d = imag(ww);
            denom = c * c + d * d;
            recalc = 1;
        }
        if (recalc)
        {
            x = fact * scalb((a * c + b * d) / denom, -llogbw);
            y = fact * scalb((b * c - a * d) / denom, -llogbw);
        }
    }
    return x + I * y;
}
```

Scaling the denominator alleviates the main overflow and underflow problem, which is more serious than for multiplication. In the spirit of the multiplication example above, this code does not defend against overflow and underflow in the calculation of the numerator. Scaling with the `scalb` function, instead of with division, provides better roundoff characteristics.

[Because of the roundoff characteristics preserved by the use of `scalb`, division of Gaussian integers is exact when it should be.]

X.11.1.2 Additive operators

Semantics

In all cases the result of a + or - operator is defined by the usual mathematical formula:

±	x	$y*I$	$x + y*I$
u	$x \pm u$	$\pm u + y*I$	$(x \pm u) + y*I$
$v*I$	$x \pm v*I$	$(y \pm v)*I$	$x + (y \pm v)*I$
$u + v*I$	$(x \pm u) \pm v*I$	$\pm u + (y \pm v)*I$	$(x \pm u) + (y \pm v)*I$

[The usual mathematical formulas for the + and - operators are inconsistent with the 1-infinity, Riemann sphere interpretation for complex numbers in that the sum of infinities can be an infinity, whereas a value with two NaN parts would be better. This is thought to be a relatively minor problem, because programs that add the results of two calculations that overflow generally have more serious problems.]

X.11.2 <complex.h>

This subclause contains specification for the `<complex.h>` functions and overloading macros that is particularly suited to IEEE implementations.

The functions (including those invoked by overloading macros) are continuous onto both sides of their branch cuts, taking into account the sign of zero. For example, `sqrt(-2 ± 0*I) == ±sqrt(2)*I`.

Since complex and imaginary values are composed of real values, each function may be regarded as computing real values from real values. The functions treat real infinities, NaNs, signed zeros, and subnormals in a manner consistent with the specification for real functions in X.9.

If each part of the argument for a one-parameter function is a NaN the function returns its argument.

The functions **`conj`**, **`imag`**, **`proj`**, and **`real`** are fully specified for all implementations, including IEEE ones, in 7.x.3.4. These functions raise no exceptions.

For other functions, the following subclauses specify behavior for special cases, including treatment of the invalid and divide-by-zero exceptions.[2]

> Subclauses are required for acos, asin, atan, cos, sin, tan, acosh, asinh, atanh, cosh, sinh, tanh, exp, log, sqrt, pow, fabs arg, exp, log, sqrt, pow, fabs, arg. Samples for asin and cosh follow.

X.11.2.? The asin macro

- **`asin`** is symmetric about the real and imaginary axes.
- **`asin`**`(+∞+iNaN)` returns `NaN±i∞` (where the sign of the imaginary part is unspecified) and optionally raises the invalid exception.
- **`asin`**`(+0+iNaN)` returns +0+iNaN.
- **`asin`**`(x+iNaN)` returns NaN+iNaN, and optionally raises the invalid exception, for nonzero finite x.
- **`asin`**`(NaN+i∞)` returns NaN+i∞.
- **`asin`**`(+0+i∞)` returns +0+i∞.
- **`asin`**`(x+i∞)` returns NaN+i∞ and raises the invalid exception, for nonzero, finite or infinite x.
- **`asin`**`(NaN+iy)` returns NaN+iNaN and optionally raises the invalid exception, for finite y.
- **`asin`**`(∞+i0)` returns π/2+i∞.
- **`asin`**`(∞+iy)` returns NaN+i∞ and raises the invalid exception, for nonzero finite y.

X.11.2.? The cosh macro

The following defines a portable implementation of a **`double complex`** function that could be invoked by the **`cosh`** overloading macro.

---

[2] Sample implementations implicitly specify behavior for special cases.

```
double complex _Ccoshd(double complex z)
{
   double x, y;
   x = real(z);
   y = imag(z);
   return cosh(x) * cos(y) + I * sinh(x) * sin(y);
}
```