Pacific ESI

Implementing Imaginary Elementary Mathematical Functions (or Leveraging Chapel's imag*(w) primitive types)*

Damian McGuckin & Peter Harding

(Corresponding Author's email: damianm@esi.com.au)

ChapelCon24 – June 5-7 2024

SHORT VERSION FOR PRESENTATION DURING ChapelCon24

Multiples of $\sqrt{-1}$

- These are what (since the 1920s) have been called imaginary numbers
- Chapel supports these with a primitive (or base) floating point type
- It is the parameterized type **imag**(*w*), $w \in [32, 64]$
- Chapel is one of the few HPC compiled languages to support such numbers
- C, D and Ada are the others (neither Fortran nor Julia nor C++ do)

Background (not spoken):

- It was Leonard Euler who first introduced the Greek symbol **ι** for $\sqrt{-1}$
- These days, mathematical texts (and C++) use ***i***, or **i**, or **`i`** for **ι**
- Some engineering & physics texts (and Python) use ***j***, or **j**, or **`j`** for **ι**
- Chapel and this presentation use **`i`**

Why are imaginary types needed? - SKIP

Imaginary numbers as a distinct type are necessary if you are going to do complex arithmetic without weird errors ...(Walter Bright)

Multiplying by y `i` does not have quite the same semantics as multiplying by $0 + y$ `i` …(Walter Bright)

Programming with an imaginary base type allows real and complex arithmetic to mix without forcing unnecessary coercions of real to complex. It also avoids a little wasteful arithmetic (with zero real parts) that compilers can have trouble optimizing away ... (W. Kahan)

+ papers by Kahan, Thomas, Coonen and others

History of Imaginary Numbers - SKIP

- Gardano 1500s - wrote them as $6 + \sqrt{-81}$
- Bombelli 1500s - called $\sqrt{-1}$ plus of minus
- Descartes 1600s unimpressed - coined the term *imaginary numbers*
- Euler 1700s – introduces (iota) $\iota = \sqrt{-1}$ - legitimacy at last!!!

- he spoke in terms of points with rectangular coordinates

- Lots of work on these numbers from late 1700s to late 1800s
- Argand 1805 - *Interpretation of Imaginary Quantities*

Imaginary Numbers => Complex Numbers - SKIP

- Gauss 1830 – he was initially sceptical of such numbers

- saying they were "enveloped in mystery, surrounded by darkness"

- but he did finally accept them, coining the term **complex number**

- the term *imaginary numbers* went out of favour

- 1920s - the term *imaginary number* reappears

- but now used solely for multiples of $\mathbf{\imath}$, e. g. 9$\mathbf{i}$

- sometimes called "a (purely) imaginary number"

Given $a = 0 + 2\mathtt{i}$ and $b = \infty - 3\mathtt{i}$ - **SKIP**

- Mathematically,

- $a \times b = (0 + 2\mathtt{i}) \times (\infty - 3\mathtt{i}) = 6 + \infty\,\mathtt{i}$

- Computationally, $a \times b$ is

- $0 \times \infty - 0 \times 3\mathtt{i} + 2\mathtt{i} \times \infty - 6\mathtt{i}^2$

- NaN $- 0 + \mathtt{i} \times \infty - 6 \times (-1)$

- NaN $+ \mathtt{i} \times \infty$ ... Antisocial

- But with a purely imaginary, $a \times$ is

- $2\,\mathtt{i} \times (\infty - 3\,\mathtt{i}\,) = 2\,\infty\,\mathtt{i} - 6\mathtt{i}^2$

- $\infty\,\mathtt{i} - 6\,(-1) = 6 + \infty\,\mathtt{i}$ …Desirable

Elementary mathematical functions

- These exist for **real**(w) and **complex**(w) arguments
- But those of **imag**(w) arguments are missing?
- If they did, they would return a result which is …

\- an **imag**(w) - often, a **complex**(w) – less often, a **real**(w) - occasionally

- Consider the square root of an imaginary number **sqrt**(±*y*`i`) where $y \geq 0$
- We call y the **multiplier**
- Its formula is just $s \pm s$ `i` where $s = \sqrt{\frac{1}{2} y}$ = **sqrt**($y/2$)
- It needs only to compute s with a single **real**(w) function
- It needs **NO** computations involving complex arithmetic

Supporting Coercion Routines – for readability - SKIP

```
inline proc cmplx(x : real(?w), y : real(w)) //  x + y i where x and y are real
{
    const z : complex(2 * w);
    z.re = x; z.im = y; // split initialization
    return z;
}
inline proc cmplx(u : imag(?w)) // provide 0 + u where u is imaginary
{
     return cmplx(0:real(w), u:real(w)); // avoids type promotion issues
}
```

Compile Time Expression – for Readability - SKIP

inline proc pix(**param** f : **real**(?*w*)) **param** *// the compile time value of* **π** × *f*

{

// value from the On-line Encyclopedia

// of Integer Sequences (OEIS™) [11]

// (should use more digits to be safe)

param A000796 = 3.1415926535897932384626;

return (A000796 * f):**real**(*w*);

}

Some Really Basic Elementary Functions

- Rewriting the polynomial form of a complex number $z = x + y\,\mathtt{i}$ in polar form

- $x + y\mathtt{i} = r \times e^{\mathtt{i}\theta}$ where $e^{\mathtt{i}\theta} = \cos(\theta) + \sin(\theta)\,\mathtt{i}$ (Euler's formula)
- $r = \sqrt{x^2 + y^2}$ or the magnitude (or absolute value) of z
- $\theta = \tan^{-1}(y/x)$ or the phase of z

- Restricting ourselves to an imaginary number, say $y\mathtt{i}$, i.e. $x \equiv 0$, we have:

- the magnitude of $y\mathtt{i}$ or $|y\mathtt{i}| \equiv$ **abs**(y) - defined for all numeric types in Chapel
- **phase**($\pm y\mathtt{i}$) $= \pm\frac{\pi}{2}$ where $y \neq 0$
- **phase**($\pm y\mathtt{i}$) $= \pm y$ where $y \equiv 0$ or y is a NaN
- $e^{y\mathtt{i}}$ = **exp**($y\mathtt{i}$) = **cos**(y) + **sin**(y) $\mathtt{i}$
- **log**($y\mathtt{i}$) = **log**($|y|$) + **phase**($y\mathtt{i}$) $\mathtt{i}$

Square Root and Phase

// **sqrt**(±y**i**) = $s \pm s$ **i** where $s = \sqrt{\frac{1}{2} y}$

```
inline proc sqrt(u : imag(?w))
{
    param half = 0.5:real(w); // ensure correct type
    const y = u:real(w); // grab the multiplier
    const s = sqrt(abs(y) * half);
    // handle NaN and signed zeros appropriately
    const i = if y > 0 then s else if y < 0 then -s else y;

    return cmplx(r, i); // better than (r, i):complex(w+w)
}
```

// **phase**(±y**i**) = $\pm \frac{\pi}{2}$ where $y \neq 0$

// **phase**(±y**i**) = $\pm y$ where $y \equiv 0$ … also handles NaN

```
inline proc phase(u : imag(?w))
{
    param half = 0.5:real(w); // ensure correct type
    param p = pix(half)); // returns π × half
    const y = u:real(w); // grab the multiplier
    // handle NaN and signed zeroes appropriately

    return if y > 0 then p else if y < 0 then -p else y;
}
```

Exponential and Logarithm Routines

```
inline proc exp(u : imag(?w)) // this is just Euler's formula
{
    const y = u:real(w); // grab the multiplier

    return cmplx(cos(y), sin(y)); // this is suboptimal
}
inline proc log(u : imag(?w))
{
    return cmplx(log(abs(u:real(w))), phase(u));
}
```

Elementary Functions -Trigonometrics

$$\mathbf{cos}(y\ \mathtt{i}) = \mathbf{cosh}(y)$$

$$\mathbf{sin}(y\ \mathtt{i}) = \mathbf{sinh}(y)\ \mathtt{i}$$

$$\mathbf{tan}(y\ \mathtt{i}) = \mathbf{tanh}(y)\ \mathtt{i}$$

$$\mathbf{asin}(y\ \mathtt{i}) = \mathbf{asinh}(y)\ \mathtt{i}$$

$$\mathbf{acos}(y\ \mathtt{i}) = \frac{\pi}{2} - \mathbf{asin}(y\ \mathtt{i})\ ,\ \textit{complex}$$

$$\mathbf{atan}(y\ \mathtt{i}) = \mathbf{atanh}(y)\ \mathtt{i}\ ,\ \textit{limited domain}$$

Note:

$$-1 \le r = \tanh(x) \le +1\ ,\ |x| \le \infty$$

Trigonometrics

```
inline proc cos(u : imag(?w)) : real(w)
{
    return cosh(u:real(w));
}
inline proc sin(u : imag(?w))
{
    return sinh(u:real(?w)):imag(w);
}
inline proc tan(u : imag(?w))
{
    return tanh(u:real(w)):imag(w);
}
inline proc asin(u : imag(?w))
{
    return asinh(u:real(w)):imag(w);
}
```

```
inline proc acos(u : imag(?w)) // Eq(16)
{
    param half = 0.5:real(w); // ensure correct type
    param p = pix(half)); // returns π × half
    //  asin (defined earlier) is imag(w) by definition
    return cmplx(p, -asin(u):real(w));
}
inline proc atan(u : imag(?w)) // Eq(17)
{
    return atan(cmplx(u)); // atan(0 + u)
}
```

Elementary Functions - Hyperbolics

$$\mathbf{cosh}(y\ \mathtt{i}) = \mathbf{cos}(y)$$

$$\mathbf{sinh}(y\ \mathtt{i}) = \mathbf{sin}(y)\ \mathtt{i}$$

$$\mathbf{tanh}(y\ \mathtt{i}) = \mathbf{tan}(y)\ \mathtt{i}$$

$$\mathbf{asinh}(y\ \mathtt{i}) = \mathbf{asin}(y)\ \mathtt{i} \quad , \ \textit{limited domain}$$

$$\mathbf{acosh}(\pm y\ \mathtt{i}) = \pm\ \mathbf{acos}(\pm y\ \mathtt{i})\ \mathtt{i} \quad , \textit{complex}$$

$$\mathbf{atanh}(y\ \mathtt{i}) = \mathbf{atan}(y)\ \mathtt{i}$$

Note:

$$-1 \le r = \sin(x) \le +1\ ,\ |x| \le \infty$$

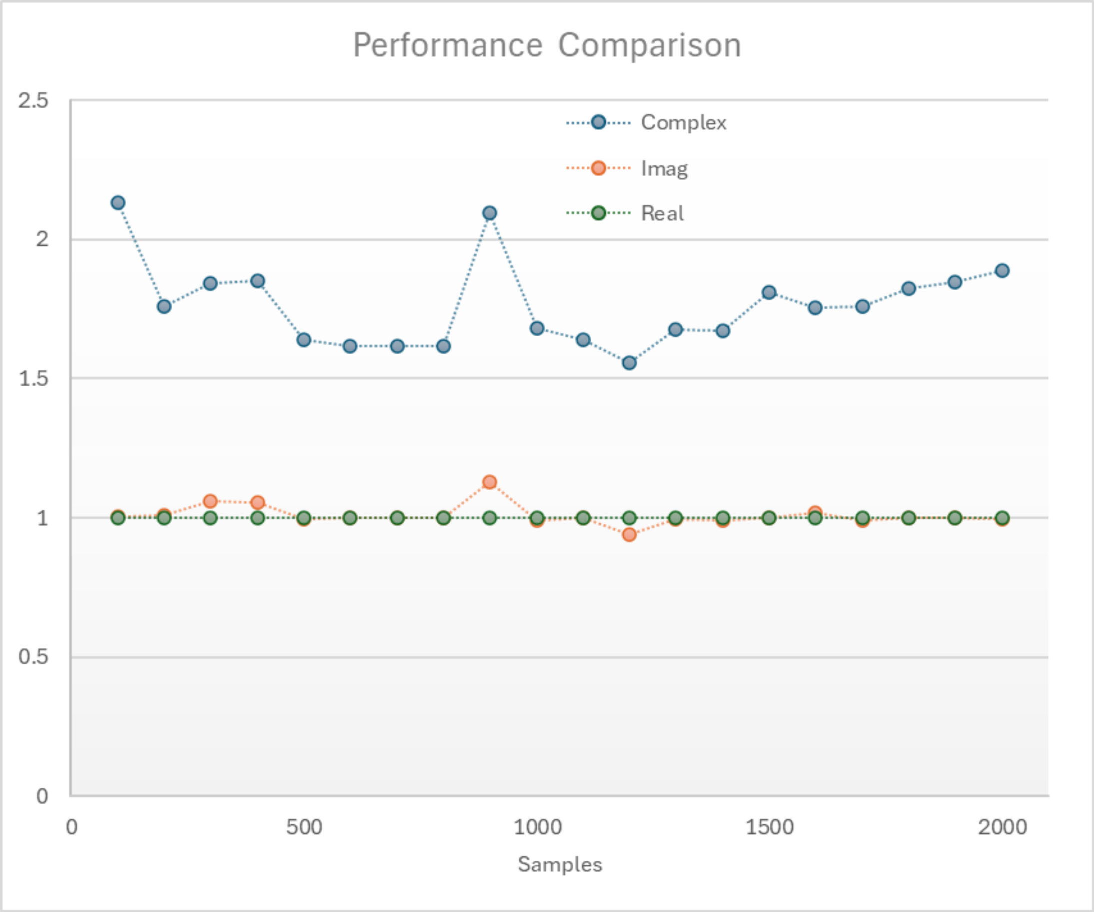
Hyperbolics

```
inline proc cosh(u : imag(?w)) : real(w)
{
   return cos(u:real(w));
}
inline proc sinh(u : imag(?w))
{
   return sin(u:real(w)):imag(w);
}
inline proc tanh(u : imag(?w))
{
   return tan(u:real(w)):imag(w);
}
inline proc asinh(u : imag(?w)) // Eq(21)
{
   return asinh(cmplx(u));
}
```

```
inline proc acosh(u : imag(?w)) // Eq(22)
{
   var z = acos(u);

   if isNegative(u:real(w)) then
      z.re = -z.re // −∞ ≤ y ≤ −0. 0
   else
      z.im = -z.im; // +0. 0 ≤ y ≤ +∞
   return cmplx(z.im, z.re);
}
inline proc atanh(u : imag(?w))
{
   return atan(u:real(w)):imag(w);
}
```

Performance



Chapel now has **imag**(*w*) elementary functions

- Completeness now exists across all floating point types
- **imag**(*w*) argument handling consistent with **real**(*w*) or **complex**(*w*)
- Implementation was straightforward (mathematics occasionally not so)
- Chapel easily handled generic arguments
- The mathematics was mostly done with existing **real**(*w*) functions
- Special case handling is done for us by these same **real**(*w*) functions
- These new routines perform well and as predicted
- Performance gain came from the mathematics not the coding
- Implementation showed a need to rework **real**(*w*) variants of cosine and sine
- **Hopefully others might benefit from our work**

… **THANK YOU** … the full version is available