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 $\text{imag}(w)$, $w \in [32, 64]$
- 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 $\iota$ for $\sqrt{-1}$
- These days, mathematical texts (and C++) use $i$, or $\text{i}$, or $\text{i}$ for $\iota$
- Some engineering & physics texts (and Python) use $j$, or $\text{j}$, or $\text{j}$ for $\iota$
- Chapel and this presentation use $\text{i}$
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) $\imath = \sqrt{-1}$ - 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 $\text{i}$, e. g. $9\text{i}$
  - sometimes called “a (purely) imaginary number”
Given $a = 0 + 2i$ and $b = \infty - 3i$

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

- Computationally, $a \times b$ is
  
  $0 \times \infty - 0 \times 3i + 2i \times \infty - 6i^2$

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

  $\text{NaN} + i \times \infty \quad \ldots \text{Antisocial}$

- But with $a$ purely imaginary, $a \times$ is
  
  $2i \times (\infty - 3i) = 2\infty i - 6i^2$

  $\infty i - 6 \times (-1) = 6 + \infty i \quad \ldots \text{Desirable}$
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 $\sqrt{\pm y \, i}$ where $y \geq 0$
- We call $y$ the multiplier

- Its formula is just $s \pm s \, i$ where $s = \sqrt{\frac{1}{2} y} = \sqrt{\frac{y}{2}}$
- 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

```plaintext
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
}
```
inline proc pix(param f : real(?w)) param // the compile time value of $\pi \times f$
{
    // 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 $z = x + yi$ in polar form
  - $x + yi = r \times e^{i\theta}$ where $e^{i\theta} = \cos(\theta) + \sin(\theta) \times i$ (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 $yi$, i.e. $x \equiv 0$, we have:
  - the magnitude of $yi$ or $|yi| \equiv \text{abs}(y)$ - defined for all numeric types in Chapel
  - $\text{phase}(\pm yi) = \pm \frac{\pi}{2}$ where $y \neq 0$
  - $\text{phase}(\pm yi) = \pm y$ where $y \equiv 0$ or $y$ is a NaN
  - $e^{yi} = \exp(yi) = \cos(y) + \sin(y) \times i$
  - $\log(yi) = \log(|y|) + \text{phase}(yi) \times i$
Square Root and Phase

// $\sqrt{\pm yi} = s \pm si$ where $s = \frac{1}{\sqrt{2}}y$

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)
}

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

// $\text{phase}(\pm y i) = \pm y$ 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 $\pi \times$ 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

\[
\begin{align*}
\cos(yi) &= \cosh(y) \\
\sin(yi) &= \sinh(y)i \\
\tan(yi) &= \tanh(y)i \\
\arcsin(yi) &= \text{asinh}(y)i \\
\arccos(yi) &= \frac{\pi}{2} - \arcsin(yi), \text{ complex} \\
\arctan(yi) &= \text{atanh}(y)i, \text{ limited domain}
\end{align*}
\]

Note:

\[-1 \leq r = \tanh(x) \leq +1, \ |x| \leq \infty\]
Trigonometrics

```plaintext
inline proc \cos(u : \text{imag}(\mathbb{C})) : \text{real}(\mathbb{C}) \\
{ 
    return \cosh(u : \text{real}(\mathbb{C})); 
}

inline proc \sin(u : \text{imag}(\mathbb{C})) \\
{ 
    return \sinh(u : \text{real}(\mathbb{C})):\text{imag}(\mathbb{C}); 
}

inline proc \tan(u : \text{imag}(\mathbb{C})) \\
{ 
    return \tanh(u : \text{real}(\mathbb{C})):\text{imag}(\mathbb{C}); 
}

inline proc \asinv(u : \text{imag}(\mathbb{C})) \\
{ 
    return \text{asinh}(u : \text{real}(\mathbb{C})):\text{imag}(\mathbb{C}); 
}

inline proc \acos(u : \text{imag}(\mathbb{C})) // Eq(16) \\
{ 
    param half = 0.5:real(\mathbb{C}); // ensure correct type 
    param p = pix(half)); // returns \pi \times half 
    // asin (defined earlier) is \text{imag}(\mathbb{C}) by definition 
    return \text{cmplx}(p, -\text{asinv}(u):\text{real}(\mathbb{C})); 
}

inline proc \atan(u : \text{imag}(\mathbb{C})) // Eq(17) \\
{ 
    return \text{atan}(\text{cmplx}(u)); // atan(0 + u) 
}
```

Elementary Functions - Hyperbolics

\[
\cosh(y \, i) = \cos(y) \\
\sinh(y \, i) = \sin(y) \, i \\
\tanh(y \, i) = \tan(y) \, i \\
\text{asinh}(y \, i) = \text{asin}(y) \, i , \text{ limited domain}
\]

\[
\text{acosh}(\pm y \, i) = \pm \text{acos}(\pm y \, i) \, i , \text{ complex}
\]

\[
\text{atanh}(y \, i) = \text{atan}(y) \, i
\]

Note:

\[-1 \leq r = \sin(x) \leq +1 , |x| \leq \infty\]
Hyperbolics

```plaintext
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 // \(-\infty \leq y \leq -0.0\)
    else
        z.im = -z.im; // \(+0.0 \leq y \leq +\infty\)
    return cmplx(z.im, z.re);
}

inline proc atanh(u : imag(\(w\)))
{
    return atan(u:real(\(w\))):imag(\(w\));
}
```

```
Performance

Performance Comparison

- Complex
- Imag
- Real

Samples

0 500 1000 1500 2000

0 0.5 1 1.5 2 2.5
Chapel now has \texttt{imag}(w) elementary functions

- Completeness now exists across all floating point types
  - \texttt{imag}(w) argument handling consistent with \texttt{real}(w) or \texttt{complex}(w)
- Implementation was straightforward (mathematics occasionally not so)
  - Chapel easily handled generic arguments
  - The mathematics was mostly done with existing \texttt{real}(w) functions
  - Special case handling is done for us by these same \texttt{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 \texttt{real}(w) variants of cosine and sine

- Hopefully others might benefit from our work

\[ \text{THANK YOU} \quad \text{\ldots the full version is available} \]