



# Implementation and Synthesis of Math Library Functions

IAN BRIGGS, University of Utah, USA

YASH LAD, University of Utah, USA

PAVEL PANCHEKHA, University of Utah, USA

Achieving speed and accuracy for math library functions like  $\exp$ ,  $\sin$ , and  $\log$  is difficult. This is because low-level implementation languages like C do not help math library developers catch mathematical errors, build implementations incrementally, or separate high-level and low-level decision making. This ultimately puts development of such functions out of reach for all but the most experienced experts. To address this, we introduce MegaLibm, a domain-specific language for implementing, testing, and tuning math library implementations. MegaLibm is safe, modular, and tunable. Implementations in MegaLibm can automatically detect mathematical mistakes like sign flips via semantic wellformedness checks, and components like range reductions can be implemented in a modular, composable way, simplifying implementations. Once the high-level algorithm is done, tuning parameters like working precisions and evaluation schemes can be adjusted through orthogonal tuning parameters to achieve the desired speed and accuracy. MegaLibm also enables math library developers to work interactively, compiling, testing, and tuning their implementations and invoking tools like Sollya and type-directed synthesis to complete components and synthesize entire implementations. MegaLibm can express 8 state-of-the-art math library implementations with comparable speed and accuracy to the original C code, and can synthesize 5 variations and 3 from-scratch implementations with minimal guidance.

CCS Concepts: • **Mathematics of computing** → **Numerical analysis**; **Continuous functions**; **Mathematical software performance**; • **Computing methodologies** → *Representation of mathematical functions*; *Special-purpose algebraic systems*.

Additional Key Words and Phrases: Function approximation, libm, DSL, type-directed synthesis, e-graphs

## ACM Reference Format:

Ian Briggs, Yash Lad, and Pavel Panchekha. 2024. Implementation and Synthesis of Math Library Functions. *Proc. ACM Program. Lang.* 8, POPL, Article 32 (January 2024), 28 pages. <https://doi.org/10.1145/3632874>

## 1 INTRODUCTION

Mathematical computations in tasks as diverse as aeronautics, banking, scientific simulations, and data analysis are typically implemented as operations on floating-point numbers. The basic operators—addition, subtraction, multiplication, and possibly division, square roots, and fused multiply-adds—are typically provided by the hardware, but higher-level mathematical functions such as trigonometric or exponential functions are implemented in software in libraries such as libm. The speed and accuracy of these software libraries can have a dramatic impact on applications such as 3D graphics [Briggs and Panchekha 2022].

To maximize performance, math libraries are written in low-level languages like C; Figure 1 shows one example. Ensuring correctness and accuracy is thus challenging. These implementation languages cannot prevent mathematical errors such as mixing up signs or using the wrong

---

Authors' addresses: Ian Briggs, University of Utah, , USA, [ibriggs@cs.utah.edu](mailto:ibriggs@cs.utah.edu); Yash Lad, University of Utah, , USA, [yash.lad@utah.edu](mailto:yash.lad@utah.edu); Pavel Panchekha, University of Utah, , USA, [pavpan@cs.utah.edu](mailto:pavpan@cs.utah.edu).

---



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2024 Copyright held by the owner/author(s).

ACM 2475-1421/2024/1-ART32

<https://doi.org/10.1145/3632874>

transformations, so the developer must be constantly vigilant. Moreover, these implementation languages require interleaving conceptually distinct components of a math library function (such as function approximation and range reduction and reconstruction), so the developer must hold the implementation's complexity in their head. Finally, these implementation languages demand making low-level decisions such as polynomial evaluation scheme or working precision up front, so the developer must have the skill to make these decisions well before even running the code. The vigilance, expertise, and experience required is a high bar for even the most expert programmers.

We propose a new way of implementing math libraries that circumvents these problems. At the core of our approach is a new DSL, MegaLibm, which allows expressing the high-level algorithms behind an implementation without fixing low-level tuning decisions; the MegaLibm implementation of Figure 1 is shown in Figure 9. The implementation in MegaLibm can then be compiled to fast and accurate C code, and low-level control over the compiled code is available through optional tuning parameters orthogonal to the high-level algorithm. Developers can thus make, test, and tweak low-level choices interactively, in a Jupyter notebook, based on measurement and data. Specialized, application-specific math libraries can thus be rapidly written, tested, and tuned.

The key to this workflow are three properties of the MegaLibm DSL: safety, modularity, and tunability. Safety means that mathematical mistakes, such as putting the wrong sign or mis-parenthesizing an expression, are caught at compile time by the type checker via a set of semantic wellformedness rules. Modularity means that complex math function implementation components such as range reduction and reconstruction are expressed via a compositional theory of function identities, allowing complex range reductions to be expressed as combinations of simple ones. And tuning means that MegaLibm carefully separates real-number from floating-point reasoning, allowing low-level speed and accuracy decisions to be made separately from high-level algorithmic ones. These properties are ensured by careful design of the MegaLibm DSL. This DSL also permits the integration of automated synthesis tools. The MegaLibm type system, for example, provides the information needed to invoke function approximation tools such as Sollya, while MegaLibm's semantic wellformedness rules allow for type-directed synthesis of range reduction and reconstruction algorithms. By leveraging the egg e-graph library [Willsey et al. 2021], including new features such as node extraction and e-graph intersection, even synthesis of whole function implementations from scratch becomes practical.

We demonstrate that MegaLibm can support expert, state-of-the-art implementations by replicating 8 function implementations from the AOCL [AMD 2021], fdlibm [Microsystems 1993], and VDT [Piparo et al. 2014] math libraries, achieving comparable accuracy and performance. We then design 5 variations on these implementations, achieving either greater accuracy or performance, to demonstrate that MegaLibm enables experimentation and design space exploration. Finally, we show that MegaLibm can synthesize 3 implementations of related functions `sinpi`, `vercos`, and `sinmx` with no input from the user. Section 8 discusses limitations and future work.

In short, this paper contributes:

- A new DSL, MegaLibm, for safe, modular, and tunable math library implementation (Section 3).
- Interactive tooling to rapidly develop math function implementations in MegaLibm (Section 4).
- Automated synthesis of math function implementations with minimal guidance (Section 5).

Section 2 provides an overview of math library implementation and MegaLibm, Section 6 contains our evaluation of MegaLibm, and Section 7 summarizes the extensive literature on math library implementation.

Copyright (C) 1993-2004 by Sun Microsystems, Inc. All rights reserved. Developed at SunSoft, a Sun Microsystems, Inc. business. Permission to use, copy, modify, and distribute this software is freely granted, provided that this notice is preserved.

```

double __ieee754_log(double x) {
    double hfsq, f, s, z, R, w, t1, t2, dk;
    int k, hx, i, j;
    unsigned lx;
    hx = __HI(x);          /* high word of x */
    lx = __LO(x);         /* low word of x */
    k=0;
    if (hx < 0x00100000) {                               /* x < 2**-1022 */
        if (((hx&0x7fffffff)|lx)==0)
            return -two54/zero;          /* log(+0)=-inf */
        if (hx<0) return (x-x)/zero;    /* log(-#) = NaN */
        k -= 54; x *= two54; /* subnormal number, scale up x */
        hx = __HI(x);          /* high word of x */
    }
    if (hx >= 0x7ff00000) return x+x;
    k += (hx>>20)-1023;
    hx &= 0x000ffff;
    i = (hx+0x95f64)&0x100000;
    __HI(x) = hx|(i^0x3ff00000);          /* normalize x or x/2 */
    k += (i>>20);
    f = x-1.0;
    if((0x000ffff&(2+hx))<3) { /* |f| < 2**-20 */
        if(f==zero) if(k==0) return zero; else {dk=(double)k;
            return dk*ln2_hi+dk*ln2_lo;}
        R = f*f*(0.5-0.3333333333333333*f);
        if(k==0) return f-R; else {dk=(double)k;
            return dk*ln2_hi-((R-dk*ln2_lo)-f);}
    }
    s = f/(2.0+f);
    dk = (double)k;
    z = s*s;
    i = hx-0x6147a;
    w = z*z;
    j = 0x6b851-hx;
    t1= w*(Lg2+w*(Lg4+w*Lg6));
    t2= z*(Lg1+w*(Lg3+w*(Lg5+w*Lg7)));
    i |= j;
    R = t2+t1;
    if(i>0) {
        hfsq=0.5*f*f;
        if(k==0) return f-(hfsq-s*(hfsq+R)); else
            return dk*ln2_hi-((hfsq-(s*(hfsq+R)+dk*ln2_lo))-f);
    } else {
        if(k==0) return f-s*(f-R); else
            return dk*ln2_hi-((s*(f-R)-dk*ln2_lo)-f);
    }
}

```

Fig. 1. The C source code for fdlibm's  $\log(x)$  function. Constant and macro definitions omitted for space.

## 2 OVERVIEW AND BACKGROUND

This overview walks the reader through implementing  $\cos(x)$  in MegaLibm in the interactive style of a Jupyter notebook session.

### 2.1 The MegaLibm DSL

At their core, most math function implementations *approximate* the target function as some easier-to-evaluate function. For example, polynomials are easy to evaluate because they only use addition and multiplication, and by picking the right polynomial, almost any function can be approximated with little error. For example, the venerable Taylor approximation of  $\cos(x)$  is  $1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \dots$ , which is a polynomial with coefficients 1 for  $x^0$ ,  $-\frac{1}{2}$  for  $x^2$ , and  $\frac{1}{24}$  for  $x^4$ . Polynomial approximations are also the simplest kind of MegaLibm implementation, expressed using a polynomial term:<sup>1</sup>

$$a_0 = 1, a_2 = -1/2, a_4 = 1/24$$

$$\text{impl}_1 = \text{polynomial}(\{0 : a_0, 2 : a_2, 4 : a_4\}) : \text{Impl}\langle a_0 + a_2x^2 + a_4x^4, [-\infty, \infty] \rangle$$

Here, `polynomial` takes a dictionary mapping powers of  $x$  to the corresponding coefficient and returns a MegaLibm term of `Impl` type. Strictly speaking, this term represents an *implementation* of a polynomial, with the polynomial itself, and its input domain, given by the parameters of its `Impl` type. Distinguishing between mathematical functions and their implementations is essential in MegaLibm, because there are typically many non-equivalent ways to compute a polynomial in floating-point arithmetic: different evaluation schemes, different or mixed precisions, and so on. These subtleties are accessible in MegaLibm via tuning parameters, which we will discuss later.

Next, we need to inform MegaLibm that this polynomial is intended to implement  $\cos(x)$ . This requires a “cast” from an `Impl` of a polynomial to an `Impl` of  $\cos(x)$ . This is an easy place to make a mistake or a typo, such as mistyping one of the coefficients or dropping a minus sign. MegaLibm’s cast operator, `approx`, prevents these mistakes by requiring an explicit domain and approximation error:

$$\text{impl}_2 = \text{approx}\left(\cos(x), \left[0, \frac{\pi}{2}\right], 0.02, \text{impl}_1\right) : \text{Impl}\left\langle \cos(x), \left[0, \frac{\pi}{2}\right] \right\rangle$$

This allows MegaLibm to check that the polynomial we have chosen is within the given error, 0.02, of the target function,  $\cos(x)$ . These *semantic wellformedness* checks exist for most terms in MegaLibm and prevent typos and mathematical mistakes, a property we call *safety*.

Besides approximating the target function, most math libraries also use *range reduction and reconstruction* to expand the domain on which an implementation is usable. These remap inputs from some wider domain, like  $[-\frac{\pi}{2}, \frac{\pi}{2}]$ , to a narrower one like the  $[0, \frac{\pi}{2}]$  chosen above. This is necessary because polynomial approximations are typically most accurate on a narrow domain. For  $\cos$ , one useful remapping leverages the fact that  $\cos(-x) = \cos(x)$ . This means that to evaluating  $\cos$  on a negative input like  $-1$ , we can instead evaluate  $\cos$  on its negation, 1, and still get the right answer. This technique is represented in MegaLibm via the `left` construct, which takes three arguments: a reduction function (here, negation), the implementation to call on the narrower domain (here, the implementation from above) and a reconstruction function (described later):

$$\text{impl}_3 = \text{left}(-x, \text{impl}_2, y) : \text{Impl}\left\langle \cos(x), \left[-\frac{\pi}{2}, \frac{\pi}{2}\right] \right\rangle$$

Note that the domain has gone from  $[0, \frac{\pi}{2}]$  to the larger range  $[-\frac{\pi}{2}, \frac{\pi}{2}]$ . In C, this `left` term corresponds to the branch `if (x < 0.0) x = -x`. Just like `approx` terms, range reduction terms are checked for semantic wellformedness to prevent errors. For this `left` term, the semantic wellformedness condition requires that  $s(x) = -x$  maps  $[-\frac{\pi}{2}, 0]$  to  $[0, \frac{\pi}{2}]$  and that  $\cos(x) = \cos(-x)$ . If we

<sup>1</sup>Expert readers, fear not: we will replace the Taylor approximation with a better one later in this section.

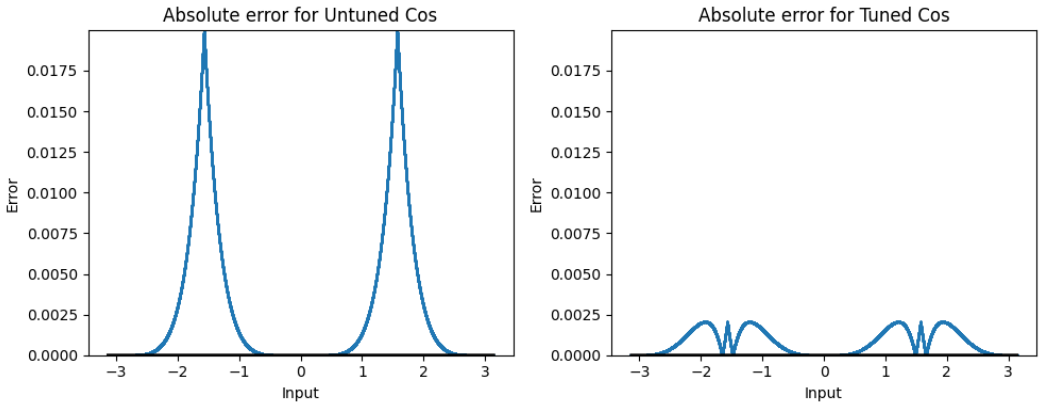


Fig. 2. Absolute error for two implementation of  $\cos$ : on the left, the un-tuned  $\text{impl}_5$ , and on the right, its tuned counterpart. The only difference is the coefficient for  $x^4$ , which is  $\frac{1}{24}$  on the left and a Sollya-synthesized  $0.0387248842917668$  on the right.

made a mistake—mistyping the range reduction function, or using the wrong input range—the user would see a type error at compile time instead of misleading results when running the code.

Real-world range reductions can sometimes require *reconstruction* afterwards. For example, the identity  $\cos(x) = -\cos(\pi - x)$  remaps inputs  $x$  to  $\pi - x$ , but only if the output  $\cos(\pi - x)$  is then *reconstructed* by negating it. The corresponding MegaLibm term is:

$$\text{impl}_4 = \text{right}(\pi - x, \text{impl}_2, -y) : \text{Impl}(\cos(x), [0, \pi])$$

There are many other cosine identities that could be used for range reduction and reconstruction, either through left and right or through MegaLibm’s other range reduction operators. There is thus a dizzying array of possible range reduction and reconstruction algorithms. In MegaLibm, this design space can be explored modularly by nesting range reduction terms. For example, combining the two reductions we’ve already seen produces an implementation valid from  $-\pi$  to  $\pi$ :

$$\text{impl}_5 = \text{left}(-x, \text{right}(\pi - x, \text{impl}_2, -y), y) : \text{Impl}(\cos(x), [-\pi, \pi])$$

This modularity allows complex range reduction and reconstruction algorithms to be built step-by-step and interactively. For now, let’s put more complex range reduction and reconstruction to the side and switch to tuning our implementation.

## 2.2 Tuning MegaLibm Implementations

So far, our implementation just describes a high-level strategy, with many important details left unspecified. That’s on purpose: MegaLibm allows users to express a high-level strategy without having to make low-level decisions. Those low-level decisions can then be tuned by compiling, measuring, and tweaking the implementation to achieve the desired speed and accuracy. To start this tuning cycle, a user first verifies that their implementation is semantically well-formed by calling `verify`. They can then call `compile` to compile the implementation to C source code, and then `measure` to run the compiled code on thousands of inputs to measure both accuracy and speed, presenting the result in a plot. Figure 2 shows the plot for  $\text{impl}_5$  above.

One thing that stands out in the plot is the spike in error near  $\frac{\pi}{2}$ . This is a consequence of using Taylor series coefficients, which are only accurate near 0. A more accurate polynomial is needed. One of the best existing tools for that is Sollya [S. et al. 2010], which implements state-of-the-art

polynomial fitting algorithms such as `remez` and `fpminimax` algorithms. Normally, a math library implementor would invoke `Sollya` to derive polynomial coefficients, and then copy those coefficients into their implementation. `MegaLibm`, however, can automate that process using *holes*, which in `MegaLibm` are written `?Impl(f(x), I)`, with the  $f(x)$  and  $I$  indicating the type of term to be put in the hole. Here, we can replace the `approx` term with a more-accurate, synthesized approximation:

$$\text{impl}_2 = ?\text{Impl}\left(\cos(x), \left[0, \frac{\pi}{2}\right]\right).\text{synthesize}(\text{"remez"}, \text{powers}=[4], \text{fixed}=\{0 : a_0, 2 : a_2\})$$

The first argument to `synthesize` names the synthesis tool to use, and the other parameters control its behavior. Here, we ask `MegaLibm` to synthesize only an  $x^4$  term, while keeping the  $x^0$  and  $x^2$  terms fixed at their current values. This takes 2.1 seconds, and the new implementation, with the filled hole, is returned and can then be compiled and remeasured, with the results shown in Figure 2. `MegaLibm` supports all of `Sollya`'s other polynomial fitting tools, and could be easily extended to call out to other software packages like `Mathematica`, `Maple`, or `Matlab`.

`MegaLibm` also allows the precise computation to be tuned using tuning parameters. For example, when evaluating a polynomial, it is sometimes useful to evaluate the first few terms in a higher precision than later terms. `MegaLibm`'s optional `split` and `split_precision` options allow tuning this. For example, to evaluate the first two terms of the polynomial in double-precision arithmetic, but the last term in single-precision arithmetic, one could write:

$$\text{impl}_1 = \text{polynomial}\left(\left[0 : 1, 2 : -\frac{1}{2}, 4 : \frac{1}{24}\right], \text{precision}=\text{fp32}, \text{split}=2, \text{split\_precision}=\text{fp64}\right)$$

Users can likewise tune the polynomial evaluation scheme (using `Estrin` or `Horner` form, which matters for longer polynomials), how reductions are executed, and numerous other low-level decisions. While tuning parameters can affect the speed and accuracy of the resulting code, they can't affect syntactic or semantics wellformedness and don't affect the high-level algorithm.

Moreover, `MegaLibm` provides a couple of special `MegaLibm` constructors that are useful for more complex tuning operations. For example, near 0,  $\cos(x)$  is exactly equal to 1. If lots of such inputs are expected, it might be worth special-casing these inputs, like so:

$$\begin{aligned} I_0 &= [-0.0003, 0.0003], I = [-\pi, \pi] \\ \text{impl}_5 &= \text{approx}(\cos(x), I_0, 2^{-24}, \text{polynomial}([0 : 1])) \\ \text{impl}_6 &= \text{split}([I_0 \mapsto \text{impl}_5, I \mapsto \text{impl}_5]), \end{aligned}$$

Here `impl5` is the degenerate “polynomial” 1, and `impl6` uses that approximation only on special case domain  $I_0$ . Note that the `approx` term's error bound,  $2^{-24}$ , guarantees that this approximation is only used where it introduces no error. Developing these split implementations can be challenging because of the complex interactions between speed and accuracy. But split tests semantic wellformedness just like other `MegaLibm` operators, the user can experiment fearlessly, rely on `MegaLibm` to catch errors as they perform this experimentation. Of course, `split` may or may not ultimately improve speed; only the user can ultimately know if this trade-off, or other tradeoffs like a slightly wider  $I_0$ , is worth it.

### 2.3 Synthesizing MegaLibm Implementations

The `MegaLibm` workflow described so far—where an expert first crafts the high-level implementation, and then adjusts tuning parameters—is perfect for experts with pre-existing numerics knowledge. Less knowledgeable users, however, benefit from the strong automated tooling that `MegaLibm`'s strict type system enables. For example, a user might want an implementation of

$$\begin{aligned}
\langle impl \rangle ::= & \text{polynomial}([n : a, \dots]) \mid \text{approx}(f(x), I, \varepsilon, \langle impl \rangle) \\
& \mid \text{left}(s(x), \langle impl \rangle, t(y)) \mid \text{right}(s(x), \langle impl \rangle, t(y)) \\
& \mid \text{periodic}(p, \langle impl \rangle, t(y, k)) \mid \text{logarithmic}(p, \langle impl \rangle, t(y, k)) \\
& \mid \langle impl \rangle \odot \langle impl \rangle \mid (y = \langle impl \rangle); \langle impl \rangle \\
& \mid \text{split}([I \mapsto \langle impl \rangle, \dots]) \mid \text{rewrite}(\langle impl \rangle, f(\dots) \mapsto g(\dots))
\end{aligned}$$

$k, n : \mathbb{Z}; \quad a, b, c, p, \varepsilon : \mathbb{R}; \quad I : \mathbb{R} \times \mathbb{R}; \quad \odot \in \{+, -, \cdot, /\}$   $f, g, s, t \in \text{mathematical expressions}$

Fig. 3. The MegaLibm grammar. Intervals, constants, and real-valued expressions are all represented symbolically. polynomial and approx are used to build polynomial and rational approximations. left and right remap one half of the domain to the other half. periodic and logarithmic do additive and multiplicative range reduction. split and rewrite to add special cases or tweak MegaLibm’s generated code.

$1 - \cos(x)$  but not be comfortable writing it themselves. MegaLibm allows them to get started anyway:

$$?Impl\langle 1 - \cos(x), [-\pi, \pi] \rangle .\text{synthesize}(\text{“tds”}, \text{“remez”})$$

Here, the synthesize method is passed not only remez, for finding polynomial approximations, but also tds, MegaLibm’s *type-directed synthesis* algorithm. Type-directed synthesis works by “reversing MegaLibm’s type rules”, and can synthesize range reduction terms like the left and right terms above, as well as more complex periodic and logarithmic reductions. Moreover, type-directed synthesis can generate multiple variations of a desired function from scratch, or help complete partial implementations. The synthesized terms can then be used directly or tuned by the user.

### 3 DSL AND TYPE SYSTEM

The MegaLibm DSL expresses math functions implementations safely, modularly, and tunably.

#### 3.1 Syntax and Semantics

The most important terms in the MegaLibm DSL have type  $Impl\langle f(x), I \rangle$ , where  $f$  is the *target function* in argument  $x$ , and the *domain*  $I$  is an interval  $[x_{lo}, x_{hi}]$  of real values that  $x$  is expected to take on. Both the function and the interval are represented symbolically by an AST containing operators (addition, subtraction, multiplication, and division), functions (square root, trigonometric, and exponential functions), constants (explicit and symbolic), and the variable  $x$ .<sup>2</sup> Typically, the user does not need to explicitly write types, which for each term can be computed from its arguments.

The full grammar of MegaLibm is given in in Figure 3. The polynomial term is the only leaf term, while all other terms combine other implementation terms. To guarantee safety, the MegaLibm DSL is typed, and each type rule is split into *syntactic* and *semantic* wellformedness. Syntactic wellformedness merely propagates target functions and domains. Semantic wellformedness checks that the implementation would work correctly over the real numbers and guarantees the absence of mathematical mistakes, sign flips, and typos. Formally, every MegaLibm type describes a real-valued function, while each term denotes a set of real-valued functions; the set structure is required to support approx. Semantic well-formedness ensures that the function represented by the type is included in the set represented by the term. The full type system for MegaLibm terms is given in Figure 4; checking semantic wellformedness is discussed in Section 4.1.

<sup>2</sup>This symbolic representation of intervals is important for implementing functions like logarithm where domains like  $[2^{-1/2}, 2^{1/2}]$  need to be represented exactly.

$$\begin{array}{c}
p(x) = \sum_i a_i x^{n_i} \quad I = [-\infty, \infty] \\
\hline
\text{polynomial}([n_i : a_i]) : \text{Impl}\langle p(x), I \rangle
\end{array}
\qquad
\begin{array}{c}
e : \text{Impl}\langle g(x), J \rangle \quad I \subseteq J \\
\forall x \in I, |f(x) - g(x)| < \varepsilon \\
\hline
\text{approx}(f(x), I, \varepsilon, e) : \text{Impl}\langle f(x), I \rangle
\end{array}$$

$$\begin{array}{c}
e : \text{Impl}\langle f(x), [m, b] \rangle \quad m = (a+b)/2 \\
\forall x \in [a, m], s(x) \in [m, b] \wedge t(f(s(x))) = f(x) \\
\hline
\text{left}(s(x), e, t(y)) : \text{Impl}\langle f(x), [a, b] \rangle
\end{array}
\qquad
\begin{array}{c}
e : \text{Impl}\langle f(x), [a, m] \rangle \quad m = (a+b)/2 \\
\forall x \in [m, b], s(x) \in [a, m] \wedge t(f(s(x))) = f(x) \\
\hline
\text{right}(s(x), e, t(y)) : \text{Impl}\langle f(x), [a, b] \rangle
\end{array}$$

$$\begin{array}{c}
P = [0, p] \quad e : \text{Impl}\langle f(x), P \rangle \\
\forall x \in P, t(f(x), k) = f(x + pk) \\
\hline
\text{periodic}(p, e, t(y, k)) : \text{Impl}\langle f(x), [-\infty, \infty] \rangle
\end{array}
\qquad
\begin{array}{c}
P = [p^{-1/2}, p^{1/2}] \quad e : \text{Impl}\langle f(x), P \rangle \\
\forall x \in P, t(f(x), k) = f(p^k x) \\
\hline
\text{logarithmic}(p, e, t(y, k)) : \text{Impl}\langle f(x), [0, \infty] \rangle
\end{array}$$

$$\begin{array}{c}
p : \text{Impl}\langle f(x), I \rangle \quad q : \text{Impl}\langle g(y), J \rangle \\
\forall x \in I, f(x) \in J \wedge g(f(x)) = h(x) \\
\hline
(y = p); q : \text{Impl}\langle h(x), I \rangle
\end{array}$$

$$\begin{array}{c}
e_i : \text{Impl}\langle f(x), I_i \rangle \quad \bigcup_i I_i = I \\
\hline
\text{split}([I_i \mapsto e_i]_i) : \text{Impl}\langle f(x), I \rangle
\end{array}
\qquad
\begin{array}{c}
e : \text{Impl}\langle f(x), I \rangle \\
\forall \vec{x} \in \mathbb{R}^n, a(\vec{x}) = b(\vec{x}) \\
\hline
\text{rewrite}(e, a \mapsto b) : \text{Impl}\langle f(x), I \rangle
\end{array}$$

Fig. 4. MegaLibm’s type rules. Each type rule’s antecedents include first a set of syntactic wellformedness conditions, which guarantee that an implementation implements the correct function on the correct range, and then, on the next line, a set of semantic wellformedness conditions, which guarantee the correctness of the implementation over the reals.

### 3.2 Approximation Terms

Dozens of function approximation schemes exist including Taylor series, Chebyshev approximations, Remez exchange, including its variations for non-Haar spaces, LLL for rounding polynomial coefficients, Caratheodory-Fejer approximations, Padé approximations, and many others, including active research such as the RLibM project [Aanjaneya et al. 2022; Aanjaneya and Nagarakatte 2023; Lim and Nagarakatte 2022]. MegaLibm must therefore provide the flexibility to use any polynomial or other function approximation while still guaranteeing safety. MegaLibm accomplishes this through its combination of approx and polynomial terms.<sup>3</sup>

A polynomial term takes a list of powers and coefficients, and is typed as an implementation of that polynomial over the full range  $[-\infty, \infty]$ . Rational polynomials and other more-complex polynomial forms can be defined using constructions such as:<sup>4</sup>

$$\text{polynomial}([1 : 16\pi, 2 : -16]) / \text{polynomial}([0 : 5\pi^2, 1 : 4\pi, 2 : -4]).$$

The polynomial coefficients are provided directly by the user, providing total flexibility to compute those coefficients using whatever method seems best. Short polynomial terms can be used to represent constants or simple linear approximations for special cases. The polynomial coefficients are symbolic and are rounded to the appropriate precision during compilation.

<sup>3</sup>MegaLibm also provides shorthands for some common non-polynomial approximations, which can be thought of as combinations of polynomial terms.

<sup>4</sup>This particular term is Bhaskara I’s famous approximation of the sine function.

There are many ways to evaluate any given polynomial in floating-point arithmetic. The precision is configurable using tuning parameters, as is the polynomial evaluation form (Horner or Estrin). The split tuning parameter also enables a limited but common form of mixed-precision evaluation. And rewrite terms (described below) can be used to leverage polynomial factoring. These choices are part of tuning and don't affect type checking.

Function approximations must then be wrapped with an `approx` term to cast them to the appropriate `Impl` type. The `approx` term declares the target function being approximated, the domain over which it is being approximated, and the maximum absolute approximation error over that domain (see Figure 4). This serves to catch typos like mixing up the signs of coefficients or entering the wrong powers of  $x$ . When synthesizing function approximations (see Section 5), the error bound is automatically filled in by the synthesis tool. The `approx` term does not generate any code and is present solely for safety and type checking.

### 3.3 Range Reduction Terms

Range reduction and reconstruction algorithms remap inputs and outputs from one domain to another, typically smaller, domain. These algorithms are specific to the function being implemented, and state-of-the-art math libraries use a great variety of different approaches. Moreover, range reduction and reconstruction typically contain a mix of multiple steps, iteratively increasing the domain the function is evaluated on. MegaLibm must therefore provide a flexible and modular framework for range reduction and reconstruction.

MegaLibm relates range reduction and reconstruction to identities of the target function. For example, a `sin` implementation might split the sign and magnitude of  $x$ , computing `sin` of the magnitude and then adding back the sign afterwards. In MegaLibm, this range reduction and reconstruction algorithm is seen to derive from the identity  $\sin(x) = -\sin(-x)$ . This identity implies that `sin` can be computed either by evaluating it directly or by *reducing* the input  $x$  to  $-x$ , then evaluating `sin`, and then *reconstructing* the output  $y$  to  $-y$ . An implementation can make use of this by using the  $-\sin(-x)$  implementation for negative  $x$  and the  $\sin(x)$  implementation for positive  $x$ , and thereby guaranteeing that core `sin` approximation is only called on positive inputs. More abstractly, each range reduction and reconstruction algorithm derives from a pair  $s/t$  of functions such that the identity  $t(f(s(x))) = f(x)$  holds. Importantly, such  $s/t$  pairs can be composed: if  $t_1(f(s_1(x))) = f(x)$  and  $t_2(f(s_2(x))) = f(x)$ , then by substitution the composition  $t_1(t_2(f(s_2(s_1(x)))))) = f(x)$  holds as well. Range reductions therefore form a monoid, which allows composing simple range reductions to construct more complex ones.

This abstract model is expressed in MegaLibm via four range reduction operators: `left`, `right`, `periodic`, and `logarithmic`. The `left` and `right` MegaLibm operators simply correspond to a branch that applies  $t(f(s(x)))$  on one half of the domain and  $f(x)$  on the other half.<sup>5</sup> For example, `right` has the type signature:

$$\frac{e : \text{Impl}\langle f(x), [a, m] \rangle \quad m = (a + b)/2}{\forall x \in [m, b], s(x) \in [a, m] \wedge t(f(s(x))) = f(x)} \quad \text{right}(s(x), e, t(y)) : \text{Impl}\langle f(x), [a, b] \rangle$$

The syntactic condition requires that  $e$  can handle inputs from the left half of the domain, while the semantic condition requires that  $s$  reduces inputs from the right half of the domain to its left half and that the  $s/t$  identity holds. This prevents mathematical errors in either  $s$  or  $t$  and ensures that

<sup>5</sup>left and right could be made more general: the midpoint  $m$  could be user-provided. This extension would not substantially change the design of MegaLibm, but we didn't find a need it for any of the functions that we wanted to implement.

the valid domains for each computation are propagated correctly. Like the rest of the MegaLibm type rules, these semantic conditions are purely mathematical and can be checked statically.

MegaLibm also provides periodic and logarithmic operators, which correspond to repeatedly composing a range reduction with itself. For example, the `exp` function satisfies the identity  $\exp(x) = \exp(x + a)/e^a$ , yielding  $s(x) = x + a$  and  $t(y) = y/e^a$ . By repeatedly applying  $s(x)$ , or its inverse, any input can be reduced to the range  $[0, a]$ .<sup>6</sup> More generally, this corresponds to the  $k$ -times-iterated identity  $t^k(f(s^k(x))) = f(x)$ . In MegaLibm, two such  $k$ -times-iterated identities are supported:  $s(x) = x + a$  via periodic and  $s(x) = x \cdot a$  via logarithmic. In each case, the user supplies the function  $t^k(x)$ , written  $t(x, k)$ , instead of just  $t(x)$ . Allowing the user to specify  $t^k(x)$  makes MegaLibm quite general; for example, the same operator, periodic, is used to define not just the periodic function `sin` but also the exponential function `exp`:

$$\text{exp\_impl} = \text{periodic}(\log(2), \dots, \text{ldexp}(y, k))$$

Nesting range reduction operators composes their  $s/t$  pairs. The individual range reductions can therefore be checked independently, but safety is guaranteed for the range reduction as a whole. This makes the implementation easier to read and allows constructing and testing range reductions incrementally. While not strictly speaking a range reduction algorithm, the MegaLibm composition operation  $(y = p); q$  is also commonly used to change input domains by computing intermediate values. This expression represents the function  $q \circ p$ , reversing the order of composition to match traditional programming notation, and also introduces the variable `y`, which can be used within `q`. It can be, and often is, freely mixed with other range reduction steps.

Just like polynomial terms, range reductions can be tuned. The periodic and logarithmic operators do not specify working precision or the specific reduction algorithm. For example, to implement  $\cos(x)$  over a large range, one might want to use Payne-Hanek reduction,<sup>7</sup> while  $\cos(x)$  over a smaller range might use Cody-Waite reduction and  $\cos(\pi x)$  might use simple division; the method tuning parameter can specify this without changing the real-number semantics of the reduction. Likewise, the precision of reduction and reconstruction in left and right can be specified by tuning parameters. The floating-point reduction bounds, periods, and multi-precision constants are all computed automatically by MegaLibm during compilation and do not have to be specified by the user.

### 3.4 Tuning for Speed and Accuracy

MegaLibm functions have fixed semantics over the real numbers, but their behavior when compiled is more complex: many low-level decisions become relevant for the speed and accuracy of the resulting code. In MegaLibm, these low-level decisions are accessible as optional tuning parameters, which don't affect the syntax or semantics of MegaLibm terms but do affect their compilation to C and thereby speed and accuracy. This design guarantees safety by ensuring that tuning parameters can be freely adjusted by users and also separates high-level and low-level reasoning, creating a form of modularity. Moreover, tuning parameters can be freely added as MegaLibm is extended, giving users intimate control over speed and accuracy, a property we term *tunability*.

A complete list of MegaLibm's current tuning parameters is given in Figure 5. All MegaLibm terms<sup>8</sup> have a `prec` tuning parameter to select the working precision of intermediate operations. For example, in left terms, this working precision is used for the evaluating the reduction and reconstruction functions  $s(x)$  and  $t(y)$ . MegaLibm supports the standard `fp32` and `fp64` precisions

<sup>6</sup>Of course, implementations don't actually contain a while loop that repeatedly adds or subtracts. Implementations instead use algorithms like Cody-Waite reduction to compute a high-precision modulus operation.

<sup>7</sup>The current MegaLibm prototype does not implement Payne-Hanek reduction, though we hope to add it in the future.

<sup>8</sup>Except no-op terms like `approx`.

Operation	Parameter	Effect
All polynomial	prec	Precision for operations: fp32, fp64, ...
	method	Polynomial evaluation method: horner, estrin, ...
	split	Number of leading terms to evaluate separately
periodic	split_prec	Precision to evaluate leading terms in
	method	Reduction method: naive, cody-waite, ...
	cw_bits	Bits per element in the Cody-Waite constant
	cw_len	Elements in the Cody-Waite constant

Fig. 5. Tuning parameters for MegaLibm operations. Each impacts the generated code coming, but does not affect syntactic or semantic wellformedness. Additional tuning parameters can therefore be easily added without affecting the MegaLibm core or the behavior of existing MegaLibm implementations. Some parameter settings are invalid and raise error: splitting more terms than a polynomial contains, or using cody-waite reduction without specifying a cw\_bits or cw\_len.

as well as compensated “double-double” operations, which are used in many math libraries. Double-double operations compile to calls to a custom library based on the CRLibm project [Daramy et al. 2003] and David Bailey’s Fortran library [Hida et al. 2008], which provides the standard addition and subtraction operations, multiplication and division operations based on fma, and a square root operation extracted from Sun’s fdlibm library [Microsystems 1993].

Other tuning parameters are specific to individual terms. For example, a polynomial term evaluates a polynomial in Horner form by default,

$$a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot (a_3 + x \cdot (a_4 + x \cdot (a_5 + x \cdot (a_6 + x \cdot a_7))))))$$

However, other evaluation schemes exist. Estrin form uses a tree structure for evaluation:

$$((a_0 + x \cdot a_1) + x^2(a_2 + x \cdot a_3)) + x^4((a_4 + x \cdot a_5) + x^2(a_6 + x \cdot a_7))$$

This is typically mildly less accurate than Horner form, but can be faster on super-scalar processors because it has a critical path of length  $O(\log N)$ , not  $O(N)$ , where  $N$  is the number of polynomial terms.<sup>9</sup> It is can also be valuable to split out leading terms, like so with three split terms:

$$a_0 + (x \cdot a_1 + (x^2 \cdot a_2 + x^3(a_3 + x(a_4 + x(a_5 + x(a_6 + x \cdot a_7))))))$$

All of these options are available to the user via the method and split tuning parameters on polynomial, and when leading terms are split out, they can be computed in a different (typically higher) precision using the split\_prec tuning parameter. periodic likewise has various tuning parameters for adjusting how the additive range reduction is actually done. If Cody-Waite range reduction is chosen, the cw\_bits and cw\_len parameters define the precision of the various constants used, which affects speed and accuracy in non-trivial ways. Some tuning parameters do additional checks and raise errors if those checks fail; for example, if a polynomial term is asked to split more terms than the polynomial actually has, or if the split argument is negative. Because these mistakes are easy to diagnose and fix, we think of these as syntactic, not semantic, errors. Tuning parameters also provide a measure of extensibility and future-proofing to MegaLibm. New algorithms, or variations of existing algorithms, are easy to add to MegaLibm as tuning parameters. As long as the tuning parameters have sensible defaults, existing implementations would not be affected.

Some accuracy and performance tweaks, however, cannot be easily represented with tuning parameters because the changes they make to the implementation are too invasive. MegaLibm

<sup>9</sup>Moreover, on processors with vector units, some of the operations can be efficiently vectorized.

provides the split and rewrite operators to handle these cases. The split operator allows an implementation to use different approximations for different portions of the input domain, such as small-angle approximations in  $\sin$ . An implementation can also use split to switch reduction algorithms or polynomial schemes depending on the input, such as using naive range reduction on inputs near 0 and Cody-Waite range reduction on inputs further away when implementing  $\sin$ .

The rewrite operator allows rewriting one mathematical expression into another, presumably with higher accuracy. For example, in Sun `fdlibm`'s  $\log$  implementation, the remapping  $s = \frac{f}{2+f}$  is performed so that  $2s = f - s \cdot f$ . It turns out that  $s$  has rounding error but  $f$  is exact, so replacing  $2s$  by  $\text{fma}(-s, f, f)$  improves accuracy. A rewrite operator allows the user to make this replacement in one place. To help support complex rewrites, the MegaLibm composition operator ( $y = p$ );  $q$  introduces a name,  $y$ , for the intermediate value, and rewrite can refer to this name.

Both split and rewrite have semantic wellformedness rules that guarantee that their use preserves the mathematical correctness of the implementation, so even when making such invasive changes to an implementation, a MegaLibm user can still rely on the MegaLibm type system to rule out typos and mathematical mistakes.

## 4 THE MEGALIBM TOOLCHAIN

MegaLibm is typically used via a Jupyter notebook. The user constructs the AST directly, with each term in the DSL having a corresponding AST constructor and returning an object of type `Impl`. Once a MegaLibm user writes a complete implementation, that implementation needs to be type-checked and can then be compiled to C to measure its speed and accuracy.

### 4.1 Type Checking

Terms in MegaLibm must be checked for both syntactic and semantic wellformedness. While syntactic wellformedness can typically be checked using a simple top-down recursive type checking procedure, semantic equivalence requires reasoning about real numbers and therefore requires more complex techniques. MegaLibm therefore performs syntactic type-checking automatically,<sup>10</sup> as terms are constructed, but defers semantic wellformedness checks to a separate check method. This check method traverses the MegaLibm term, gathers all semantic wellformedness constraints given by the type rules in Figure 4, and attempts to prove each.

Each constraint takes the form of a comparison or equality between two real-valued expressions with variables drawn from intervals. Unfortunately, determining equality for arbitrary real-valued expressions is known to be hard—dependent on unproven mathematical conjectures [Boehm 2020], and possibly undecidable. MegaLibm therefore cannot provide a complete decision procedure for this problem; instead, it provides access to several different backends, which differ in their soundness and completeness: `egg`, `sympy`, `dirtyinfnorm`, and `sampling`. New verification tools such as `Mathematica` or `Maple` could be easily added in the future.

The `egg` backend can prove equalities (but not inequalities) using the `egg` e-graph library [Willsey et al. 2021] and a custom set of rewrite rules initially drawn from Herbie [Panchekha et al. 2015] but with unsound rules removed. To prove two expressions equal, both are added to a single e-graph, and then the rewrite rules are run for a user-configurable number of iterations or until the e-graph reaches a user-configurable size bound. If the e-nodes corresponding to the two expressions are in the same e-class at the end of the iteration, the expressions are equal; otherwise, the `egg` backend reports an error. The `egg` backend is fast and, to the best of our knowledge, sound, but is unable to

<sup>10</sup>Since MegaLibm intervals are represented symbolically, syntactic wellformedness can require reasoning about real numbers. The only tests necessary, however, are comparisons. Our prototype therefore just uses a high-precision evaluation, but deferring these checks to the semantic type-checking phase would also be possible.

Node	Input(s)	Output(s)	Additional Arguments
cast	1	1	c_type
cody_waite	1	2	period, bits_per, entries
decompose	1	2	
expression	any	1	expr
estrin	1	1	monomials, coefficients, split
horner	1	1	monomials, coefficients, split
if_less	1	1	bound, t_val, f_val
mod_switch	2	1	mod_to_blocks
set_exp	2	1	
additive	1	2	period
split_dom	1	1	dom_to_block

Fig. 6. The nodes used in MegaLibm’s intermediate representation. Inputs and outputs to nodes always result in C-level variable names. Additional arguments are mandatory and vary in type.

prove some more complex identities. The sympy backend uses the simplify method from Python’s SymPy library. Sympy internally uses a normalization procedure instead of rewrites, and is mature and well-tested; it can prove both equalities as well as some inequalities. Like the egg backend, the sympy backend is sound (at least, assuming no bugs in SymPy) but not complete.

MegaLibm also provides some unsound checking algorithms. The dirtyinfnorm backend checks comparisons, such as the approx operator’s wellformedness condition. It uses interval arithmetic at evenly-spaced points along the domain as well as some form of first-order search to find possible minima and maxima. It typically yields tight bounds, but in theory is not sound. The sampling backend can verify any kind of condition by sampling inputs in the domain, evaluating all terms in high precision, and testing all conditions up to some tolerance; the precision and tolerance can be configured by the user. Since the user chooses the set of verification tools to use, the soundness and completeness of the result is ultimately up to the user.

## 4.2 Compilation

Each term in the MegaLibm DSL can be compiled to C by invoking its generate\_c method. Internally, this occurs in two steps: first, converting the MegaLibm term into an intermediate representation, which requires lowering all real-number constants to floating-point ones, and then converting that intermediate representation into C. The intermediate representation takes the form of an ordered sequence of untyped computation nodes linked together in a graph, analogous to a “sea of nodes” SSA IR. Each node has a fixed count of input and output variables, plus named configuration parameters that typically reflect tuning parameters in the MegaLibm DSL. The first input and output parameter is special: it always represents the transformed input  $x$  or output  $y$ . For instance, the cast block takes a single input, produces a single output, and has a c\_type parameter to define the underlying type. The intermediate representation has no representation of control flow: all IR nodes are computed in order, one after another. This is possible because branches in math function implementations are typically short, more like conditional moves than true branches. Avoiding control flow dramatically simplifies the intermediate representation.

Figure 6 lists all of MegaLibm’s intermediate representation nodes. The cast block handles casts between MegaLibm’s supported number representations. The decompose and set\_exp nodes convert between a floating-point number and the integer fields that comprise it; they are analogous to the C standard frexp and ldexp functions, but directly use bit tricks because MegaLibm assumes

that NaN and infinite values are handled separately. The `estrin` and `horner` nodes implement the corresponding polynomial evaluation schemes. The `additive` and `cody_waite` nodes handle periodic terms with naive or cody-waite methods; the `cody_waite` term has additional arguments to configure the high precision constant. The `expression` node is used for reduction and reconstruction functions and evaluates arbitrary mathematical expressions by the corresponding C operator. Finally, MegaLibm's intermediate representation has several specialized conditionals. The `if_less` node handles conditionals in left and right terms; specializing `if` to comparisons avoids having to explicitly represent boolean operations. However, the `mod_switch` node is a specialized conditional for branching on the lower bits of an integer constant  $k$  using C's `switch` operator, which is useful in periodic reconstruction functions. `split_dom` uses a chain of `if/else` conditions to handle the interval testing in split terms. These specialized conditional nodes are automatically inserted by MegaLibm, and are important for achieving good performance.

To compile an implementation, its terms are traversed in post-order and each term generates nodes following a template. Terms do not examine the compiled form of other nodes, except in the case of `rewrite`. Input and output types (`fp64`, `fp32`, or `dd` for double-double) are tracked during compilation and casts are automatically inserted to account for `prec` tuning parameters. After all the nodes are generated, names are chosen for each intermediate variable and C code is produced for the sequence of nodes. While MegaLibm currently only supports C, in the future we hope to develop additional backends. The IR-based architecture should simplify that extension. MegaLibm does not perform any kind of optimization to the IR, and frequently generates code with common subexpressions and dead code. MegaLibm therefore relies on the C compiler's optimizer to eliminate redundancy; because the IR node typically contains simple floating-point operations, simple control flow, and `always_inline` functions, the C compiler's optimizer is typically sufficient and the resulting assembly code is acceptably fast.

### 4.3 Measurement

Once a MegaLibm implementation is compiled to C, it can be run on randomly sampled points to estimate its speed and accuracy. Importantly, this measurement step is integrated into the Jupyter workflow, facilitating the user rapidly compiling, testing, and tweaking their implementation.

To measure a compiled implementation, MegaLibm generates and compiles a series of drivers. Each driver randomly generates inputs uniformly distributed over an interval; by default, the domain of the implementation itself is used, but the user can optionally provide a subset of the domain, which is useful for "zooming in" on inputs of interest, such as when developing split terms. After sampling points in the domain and saving them in memory, MegaLibm then invokes the implementation on each point in turn, saving the results for later analysis. The total execution time is saved as a measure of the implementation's speed. Naturally, execution speed can vary depending on the execution context, machine state, and the specific distribution of inputs used. However, since math function implementations are typically arithmetic-heavy and control-flow-light, simple measurement over uniformly sampled inputs tends to provide a meaningful measure of speed.

To measure the implementation's accuracy, each output is compared to an oracular result computed by MPFR. The number of points is configurable by the user. For each output, we compute the absolute error compared to the oracular result and plot all of the outputs in a dot plot, such as the one in Figure 2. Note that this produces an error plot, not a worst-case analytic bounds. While analytic bounds are the ultimate goal of math function implementation, it is often important, while a function is being developed, to focus on how error varies with input and how common inaccurate inputs are (which often provides a hint how to fix them). Verification of math function accuracy is still an active area of research [Das et al. 2020; Lee et al. 2017]. As the tools in this area mature, we plan to integrate them into MegaLibm.

Debugging speed or accuracy is helped by the fact that MegaLibm implementations are modular. This means that users can typically measure the accuracy of individual pieces—for example, just the function approximation, or just one of several range reductions—to get insight into what specific step is inaccurate or slow. MegaLibm’s tuning parameters then provide a set of options to achieve greater speed or accuracy for that step. For example, a user might find that their implementation is inaccurate on a certain input. They would then proceed to test, say, their core polynomial approximation, and if that is accurate, they may then try increasing the precision of a range reduction step. Key to this incremental workflow is the fact that individual implementation steps are easy to separate in the MegaLibm DSL and that the type system ensures safety, avoiding misleading mistakes during the testing and tuning loop.

## 5 SYNTHESIS IN THE MEGALIBM DSL

MegaLibm’s safety, modularity, and tunability also makes it a good target for synthesis. To that end, MegaLibm provides easy access to several synthesis algorithms to automatically generate part or all of a MegaLibm implementation for a given mathematical function.

### 5.1 Synthesizing Approximations

Math library implementors already commonly use specialized tools to compute polynomial approximations. Popular tools include Maple, Mathematica, Matlab, and Sollya, and the topic is an area of ongoing research [Lim and Nagarakatte 2022]. Sollya in particular is free, open source, and implements multiple algorithms including Taylor series, Chebyshev approximation, Remez exchange, and shortest-vector search. It also has a rigorous approach to safety, making it a good match for MegaLibm. MegaLibm therefore provides access to Sollya for synthesizing polynomial approximations.

To synthesize MegaLibm implementations, the user uses *hole* terms. Holes are typed; a hole for type  $\text{Impl}\langle f(x), I \rangle$  is written  $?\text{Impl}\langle f(x), I \rangle$ . Implementations then have a *synthesize* method, which takes in a list of synthesis tools and attempts to apply each tool to each hole in turn, returning a list of synthesized implementations. This *synthesize* method provides flexibility for users: they can synthesize each hole individually, passing custom tuning parameters for each one, or construct a partial implementation containing potentially multiple holes and then calling *synthesize* with a generic set of options. Incorporating synthesis into MegaLibm reduces errors from copying functions and coefficients between tools, and supports MegaLibm’s interactive workflow. Moreover, the MegaLibm synthesis method is designed for extensibility. Both Maple and Mathematica, for example, can synthesize rational function approximations to a function. Since we do not have access to either (pricey) package, MegaLibm does not currently have the ability to call them, but these algorithms would be easy to add to MegaLibm and then invoke via *synthesize*.

To invoke Sollya, users request the *taylor*, *chebyshev*, *remez*, and *fpminimax* synthesis tools. These tools synthesize  $\text{approx}\langle f(x), I, \epsilon, \text{polynomial}(C) \rangle$  terms, where the target function  $f(x)$  and interval  $I$  come from the hole’s type, while the error bound  $\epsilon$  and coefficient list  $C$  are computed by Sollya. The user can customize the polynomial by passing arguments to *synthesize*, such as specifying the number of polynomial terms, the powers of  $x$  to use, the precision of the coefficients (for *fpminimax*), or any number of fixed initial terms. If the function is known to be odd or even (see below), Sollya synthesizers such as *remez* and *fpminimax* automatically attempt to synthesize coefficients only for the odd or even powers (configurable by the user). If Sollya fails to synthesize an approximation, which happens in a variety of complicated and hard-to-predict cases, MegaLibm attempts minor fix-ups, including increasing the working precision or slightly expand the input domain. Users can also provide lists for parameters such as the number of polynomial terms to

Template	$s(x)$	Generated term
$\text{flip}(a)$	$a - x$	left and right
$\text{shift}(a)$	$a + x$	left, right, and periodic
$\text{scale}(a)$	$a \cdot x$	left, right, and logarithmic

Fig. 7. Reduction templates for MegaLibm terms, including the template name, the reduction function  $s(x)$ , and the MegaLibm terms that can use such reductions

synthesize multiple alternative implementations for each hole. The same occurs if the user specifies multiple function approximation tools such as both chebyshev and remez.

## 5.2 Type-directed Synthesis

Function implementations typically require range reductions as well as function approximations in order to be competitive. While expert users may prefer to design range reductions by hand, less-expert users benefit from automation. MegaLibm thus provides a type-directed synthesis algorithm, invoked as `tds`, to automatically suggest range reduction and reconstruction steps.

The basic idea of `tds` is to “reverse the type rules”, using the type of the requested hole to determine which range reductions and reconstructions are possible. For example, consider synthesizing an  $\text{Impl}\langle 1 - \sin(x), [-\pi, \pi] \rangle$ , via a left operator. This requires synthesizing matching functions  $s$  and  $t$  such that  $t(1 - \sin(s(x))) = 1 - \sin(x)$ , where moreover  $s([-\pi, 0]) \in [0, \pi]$ . One such mapping is  $s(x) = -x$  and  $t(y) = 2 - y$ , which then transforms the initial term into

$$\text{Impl}\langle 1 - \sin(x), [-\pi, \pi] \rangle \rightsquigarrow \text{left}(-x, \text{Impl}\langle 1 - \sin(x), [0, \pi] \rangle, 2 - y).$$

The expanded implementation has another hole, which is expanded again by `tds`, repeating until a full term is built. Typically the user invokes MegaLibm’s synthesis method with both `tds` and some polynomial synthesis tool like `remez`, which is used to fill in leaf terms in the implementation. The type rules, especially semantic wellformedness, heavily constrain the possible implementations and therefore cut down the search space.

The key challenge in this type-directed synthesis approach is efficiently finding  $s/t$  pairs for arbitrary target functions. The `tds` tool leverages a key observation about these pairs: the reduction operation  $s(x)$  is typically one of a small number of operations (flips, shifts, and scales) while the reconstruction operation  $t(y)$  can be quite complex and specific to the function being implemented. MegaLibm exploits this asymmetry by considering a fixed set of reduction templates  $s(x)$  and attempting to synthesize a matching  $t(y)$  for each. For example, to synthesize the term  $\text{left}(-x, \dots, y)$ , MegaLibm would attempt to instantiate the flip template  $s(x) = a - x$  and then find a  $t(y)$  such that  $t(1 - \sin(a - x)) = 1 - \sin(x)$ . The full set of templates can be found in Figure 7.

To find the reconstruction function for each possible template, MegaLibm uses equivalence graphs. Equivalence graphs explore and compactly store many similar, equivalent programs [Kozen 1977], and MegaLibm uses the popular egg library [Willsey et al. 2021] (via its Python interface `snake_egg`) for working with e-graphs. When using egg, the user typically provides a starting expression and a set of rewrite rules, and egg then “grows” the e-graph, exploring the space of expressions equivalent to the starting point and reachable via the rewrite rules. MegaLibm initializes the e-graph with the term  $\text{thefunc}(x)$ , and rewrite rules including sound mathematical rewrite rules, the function definition  $\text{thefunc}(x) \leftrightarrow f(x)$ , and a rewrite rule of the form  $\text{thefunc}(s_T(x)) \rightsquigarrow T$  for each reduction template  $T$ . For example, the  $\text{flip}(a)$  template generates the rewrite rule  $f(a - x) \rightsquigarrow \text{flip}(a)$ . Given a starting point like  $\text{thefunc}(x) = \sin(x)$ , the e-graph will find an equivalent form like  $-\sin(0 - x)$  using the mathematical rewrite rules, convert it into  $-\text{thefunc}(0 - x)$  via the definition rule, and then rewrite it into  $-\text{flip}(0)$  using the reduction template rewrite rule. Note that this

final form does not use the input  $x$ ; such expressions, without  $x$ , are then split into a reduction function  $s(x)$ , based on the reduction template, and a reconstruction function  $t(x)$  based on the rest of the expression. For example, the  $-\text{flip}(0)$  discovered for  $\sin(x)$  is then split into  $s(x) = 0 - x$  and  $t(y) = -y$ , and terms  $\text{left}(0 - x, ?, -y)$  and  $\text{right}(0 - x, ?, -y)$  are proposed.

Type-directed synthesis has several limitations. It can only generate left, right, periodic, and logarithmic terms; even adding also approx and polynomial using Sollya, there are still many MegaLibm constructs that it cannot synthesize. The set of reduction templates is also limited, and does not include some more-complex range reductions used in some state-of-the-art math function implementations. Finally, type-directed synthesis also depends on the mathematical rewrites provided to the e-graph library. MegaLibm’s rewrite rules are based on the rewrite database of Herbie [Panchekha et al. 2015], with unsound rules audited by hand and removed. These rewrite rules include identities for core functions such as  $\sin(x)$  and  $\log(x)$ , so synthesizing a sin or log implementation using type-directed synthesis is more an exercise in recall than synthesis. However, tds shines when synthesizing implementations of compound operations such as  $\sin(\pi x)$ ,  $1 - \cos(x)$ , or  $\sin(x) - x$ . These operations are common in real-world code and do not achieve the best possible speed and accuracy when implemented naively. Type-directed synthesis thus provides users with get acceptable implementations without relying on user expertise.

### 5.3 Making Type-Directed Synthesis Practical

Once the e-graph is grown, it must be searched for the expressions of interest, specifically expressions equivalent to the starting point  $f(x)$  and that do not contain the variable  $x$ , meaning that all calls to  $f(x)$  have been replaced with reduction templates. However, the very nature of e-graphs implies that the e-graph likely stores a huge number of such terms, possibly exponentially or even infinitely large. To extract a large but potentially useful set of rewrites, MegaLibm uses *node extraction*, where one representative expression is extracted from the e-graph for each “e-node” in the “e-class” corresponding to  $f(x)$ . In effect, this extracts the largest set of expressions where any two expression in the set either differ at the root operator (such as  $0 - \text{flip}(0)$  and  $0 + \text{flip}(1)$ ) or have the arguments to the root operator that are non-equivalent (such as  $0 + \text{flip}(0)$  and  $1 + \text{flip}(1)$ ). The extraction procedure is configured to heavily penalize terms with an  $x$  variable, and extracted terms that use that variable are discarded.

Node extraction is fast and typically results in dozens to hundreds of extracted expressions. However, far from all of the extracted expressions are useful. Some restate trivial identities; for example, the term  $\text{shift}(0)$  stands for to expression  $f(x + 0)$ , meaning that it represents the identity  $f(x) = f(x + 0)$ . Others restate each other, such as the terms  $\text{flip}(0)$  and  $1 \cdot \text{flip}(0)$ . In these terms have different root operators, meaning all of these duplicates will be extracted. To filter out these duplicates, we construct a second e-graph, insert all of the extracted terms, and again grow that e-graph. However, this e-graph is not provided with the definition of  $\text{thefunc}(x)$ . In effect, this procedure asks the e-graph to identify all extracted terms that would be equivalent for *any* function  $f(x)$ , and which therefore cannot provide a useful guide toward implementing  $f(x)$  in particular. This cuts the number of generated identities substantially, typically to only a handful. Type-directed synthesis can then rapidly refer to just this small handful of possible identities and rapidly enumerate possible implementations.

Discovering identities such as  $\sin(x) \rightsquigarrow -\text{flip}(0)$  by growing an e-graph can take a long time—typically tens of seconds. It is therefore important that the e-graph is grown as few times as possible. But since type-directed synthesis adds terms that themselves contain holes, type-directed synthesis can be invoked repeatedly. MegaLibm’s type-directed synthesis engine thus grows the e-graph when filling the first hole, and then caches the resulting identities.

One more challenge arises with periodic and logarithmic terms. These are generated from shift and scale templates, via terms like  $\sin(x) \rightsquigarrow -\text{shift}(\pi)$  and  $\log(x) \rightsquigarrow \text{scale}(2) + \log(2)$ . The issue is that while these terms contain  $t(y)$ , the periodic and logarithmic terms require  $t(y, k) = t^k(y)$ . This requires solving an inductive equation:  $t(y, 0) = y$  and  $t(y, n + 1) = t(t(y, n))$ . To solve this inductive equation, MegaLibm uses e-graph intersection [Gulwani et al. 2005]. Specifically, MegaLibm constructs and grows a series of e-graphs  $E_1, E_2, \dots$  where e-graph  $E_k$  contains:

- The term  $t(y, k)$  and equality  $t(y, k) = t(t(\dots(y)))$
- The variable  $n$  and equality  $n = k$

This series of e-graphs is then *intersected*, resulting in a new e-graph  $E^*$  that only contains terms and equalities that were true in each intersected e-graph  $E_k$ . For example, consider  $t(y) = y + \log(2)$ , which is needed to synthesize the logarithmic reduction for  $\log(x)$ . Here, the e-graph  $E_1$  initially contains the equalities  $t(y, 1) = y + \log(2)$  and  $n = 1$ . After growing this e-graph, it will also contain the equality  $t(y, n) = y + n \cdot \log(2)$ . The e-graph  $E_2$  initially contains the equalities  $t(y, 2) = (y + \log(2)) + \log(2)$  and  $n = 2$ , but after growing, it too contains  $t(y, n) = y + n \cdot \log(2)$ . E-graph intersection can thus, in some cases, solve the required inductive equation. Luckily, the inductive equations required are typically simple, and this synthesis method is often successful, requiring only a few e-graphs  $E_k$  grown for only a few iterations.

## 6 EVALUATION

Our evaluation aims to answer three research questions about MegaLibm:

**RQ1** Can MegaLibm implement state-of-the-art math libraries safely, modularly, and tunably?

**RQ2** Is MegaLibm able to tune math libraries for greater speed and accuracy?

**RQ3** Is MegaLibm able to synthesize math libraries automatically?

To this end, we perform three tests, first recreating an array of existing math libraries, then tuning those libraries for greater speed and accuracy, and finally synthesizing implementations of related functions from scratch. All experiments were performed on an 2020 Apple MacBook Pro with an M1 processor, 16GB of RAM, Python 3.11.3, Sympy 1.12, Sollya 8.0, and Clang 14.0.3 run with `-O3 -mtune=native -DNDEBUG`. Error and runtime measurements are performed using MegaLibm's native tools, as described in Section 4.3.

### 6.1 Re-creating existing libraries (RQ1)

Existing math libraries are time-tested, with both algorithms and tuning parameters persisting for decades. To determine whether MegaLibm can express the techniques required in state-of-the-art implementations, we chose 8 implementations from three widely-used libraries (Sun's `fdlibm`, perhaps the most widely copied math library; VDT, a modern variant of the well-known Cephes library; and AMD's `Optimizing CPU Libraries`) and attempted to reimplement each in MegaLibm. The libraries, functions, and implementations are listed in Figure 8, along with the speed and accuracy of both the original implementation and MegaLibm's reimplementations. For each function, we started by reading code comments, which typically had an English-language description of the high-level algorithm behind the implementation. We then implemented that algorithm in MegaLibm and tweaked the tuning parameters until we could match the original implementation's accuracy and speed. When the original library used techniques not currently implemented in MegaLibm, or when we found flaws in the original library, we prioritized keeping to the spirit of the original implementation over accuracy or speed. MegaLibm's reimplementations are roughly the same speed and accuracy, averaging 26% more accurate but 14% slower. Most of that slow-down is from two related implementations; if those are excluded, MegaLibm's reimplementations are 0.5% slower on average. Moreover, MegaLibm's reimplementations are much shorter than the originals, as shown

Library	Func	Domain	Error		Runtime		Source Lines	
			Original	MLM	Original	MLM	Original	MLM
AOCL	fast_asin	[0, 0.5]	<b>7.50e-17</b>	<b>7.50e-17</b>	7.34	<b>7.26</b>	56	<b>24</b>
		[-1, 1]	<b>2.69e-16</b>	2.70e-16	<b>17.93</b>	18.47		
	fast_asinf	[0, 0.5]	<b>3.63e-8</b>	<b>3.63e-8</b>	7.41	<b>7.33</b>	27	<b>14</b>
		[-1, 1]	<b>1.60e-7</b>	<b>1.60e-7</b>	19.02	<b>18.78</b>		
	opt_asinf	[0, 0.5]	<b>3.78e-8</b>	<b>3.78e-8</b>	7.32	<b>7.27</b>	40	<b>18</b>
		[-1, 1]	<b>5.96e-8</b>	<b>5.96e-8</b>	18.10	<b>17.62</b>		
ref_asin	[0, 0.5]	6.49e-17	<b>6.40e-17</b>	7.24	<b>12.51</b>	41	<b>21</b>	
	[-1, 1]	1.66e-16	<b>1.44e-16</b>	<b>17.22</b>	25.43			
fdlibm	asin	[0, 0.5]	<b>6.02e-17</b>	6.03e-17	<b>7.20</b>	11.22	52	<b>25</b>
		[-1, 1]	1.66e-16	<b>1.44e-16</b>	<b>16.37</b>	25.62		
	log	[1.7, 2.4]	7.43e-17	<b>7.04e-17</b>	7.44	<b>7.31</b>	61	<b>23</b>
		[1, 50]	<b>2.64e-16</b>	2.76e-16	<b>7.04</b>	8.24		
VDT	cos	[0, 0.7]	<b>1.34e-16</b>	1.38e-16	7.46	<b>7.29</b>	50	<b>27</b>
		[-4, 4]	<b>1.42e-16</b>	1.44e-16	7.17	<b>7.05</b>		
	exp	[-0.3, 0.3]	<b>2.69e-16</b>	<b>2.69e-16</b>	7.39	<b>7.27</b>	30	<b>16</b>
		[-20, 20]	<b>6.40e-8</b>	<b>6.40e-8</b>	<b>7.04</b>	7.05		
		[0, 50]	<b>6.77e5</b>	<b>6.77e5</b>	7.55	<b>7.55</b>		

Fig. 8. Maximum absolute error and average runtime for each of the implementations reimplemented using the MegaLibm DSL. MegaLibm’s variation reached the exact same output for tested point in VDT’s  $\exp(x)$  and two of AOCL’s  $\sin^{-1}(x)$  variations. Other implementations have very close error values. Performance is matched across most implementations with AOCL’s reference and fdlibm’s versions of  $\sin^{-1}(x)$  being slower when reimplemented in MegaLibm.

by the line of code counts in Figure 8. These line of code counts understate MegaLibm’s advantage, because the C code of the original libraries often uses complex features such as pointers, unions, casts, and macros, visible in Figure 1, while MegaLibm’s DSL does not.<sup>11</sup> Many of MegaLibm’s lines of code, by contrast, are simply coefficients for polynomial terms. To get a better sense of MegaLibm’s flexibility and how it can match state-of-the-art implementations, we describe the case studies in Figure 8 in more detail.

*fdlibm’s log.* Sun’s fdlibm library uses a complex and clever implementation of the simple natural logarithm  $\log$ . Like most implementations, it begins by first extracting and saving the exponent bits, reducing the input range to  $[2^{-1/2}, 2^{1/2}]$  (implemented with MegaLibm’s logarithmic operator) and then subtracting 1 from the input (implemented with MegaLibm’s composition operator), which is exact, thanks to Sterbenz’s lemma [Sterbenz 1974]. That leaves the task of implementing  $\log(f + 1)$  on a range of approximately  $[-0.3, 0.4]$ , where  $f = x - 1$ . Next, fdlibm performs the substitution  $s = f/(2 + f)$ , transforming the target function into  $\log(1 + s) - \log(1 - s)$  on the range  $[-0.18, 0.17]$ ; this is again represented in MegaLibm with a composition. The advantage of this transformation is that the new target function is odd, meaning that it can be approximated by a polynomial with only odd powers of  $x$ . Fdlibm then uses a polynomial approximation, organized in a somewhat strange way with the terms split modulo 4 and the leading term split out; in MegaLibm, this strange organization can be replicated with addition terms, though perhaps it could also be represented by a new value for the method tuning parameter for the polynomial term. Finally, to combat rounding error in the computation of  $s$ , fdlibm uses a rewrite of  $2s \rightsquigarrow f - s \cdot f$ , iterated once for most of

<sup>11</sup>Of course MegaLibm’s generated C code does use these features because they are necessary for maximum performance.

```

R = polynomial([4 : ...]) + polynomial([2 : ...])
core = approx(log(1 + s) - log(1 - s), [-0.17, 0.17], 5E-19, 2 · s + s · R),
logarithmic(2,
  (f = x - 1); split([
    [-9.5E-7, 9.5E-7] ↦
      approx(log(f + 1), [-9.5E-7, 9.5E-7], 3E-25,
        polynomial([1 : 1, 2 : -0.5, 3 : 0.33])),
    [-0.29, 0.38] ↦
      (s = f / (2 + f)); rewrite(core,
        (2 · s) ↦ fma(-s, f, f)),
    [0.38, 0.41] ↦
      (s = f / (2 + f)); rewrite(core,
        (2 · s) ↦ (f - f · f · 0.5 + (s · f · f · 0.5))))],
  y + k · log(2))

```

Fig. 9. A implementation of `fdlibm`'s  $\log(x)$  function in MegaLibm. Polynomial coefficients and tuning parameters are omitted for clarity, and various numeric constants are shortened.

the range and twice for a particularly difficult interval from 0.38 to 0.41. This rewrite is expressed in MegaLibm with the `rewrite` operator, which has the advantage of verifying that the rewrite is mathematically correct. Finally, the reconstruction operation is implemented in the standard way, with the original exponent bits multiplied by  $\log(2)$  and added to the final value. MegaLibm's reimplementaion of `fdlibm`'s  $\log$  is shown in Figure 9, with tuning parameters hidden to highlight the overall algorithm.

One technique in `fdlibm`'s  $\log$  implementation, however, could not be replicated in MegaLibm. In `fdlibm`'s  $\log$ , the reconstruction operation  $y + k \cdot \log(2)$  and the polynomial summation  $2s + s \cdot R$  are interleaved, with terms summed from smallest to largest. This reduces compounding rounding error, but interferes with MegaLibm's "black-box" approach to compilation. To approximate the effects of reordering, our MegaLibm implementation uses double-double arithmetic [Dekker 1971], which achieves similar accuracy to `fdlibm` but at the cost of additional floating-point operations. In the future, we hope to add support for this technique by adding an uninterpreted-sum data type to MegaLibm. Adding uninterpreted sums would simply concatenate compile-time lists of terms, and converting an uninterpreted sum into a floating-point value would perform interval analysis to sort terms by magnitude before adding them together. Preliminary tests suggest that this would allow MegaLibm to match `fdlibm`'s  $\log$  exactly.

*VDT's cos.* Like many cosine implementations, VDT reduces the input range to  $[0, \frac{\pi}{4}]$  using a Cody-Waite additive range reduction (implemented using MegaLibm's periodic operator) and reconstructing the result using the count  $k$  of the reduction modulo 4. On the reduced range, VDT uses a polynomial approximation. Most of the terms of this polynomial approximation are generated by Sollya's `fpminimax` algorithm. However, the first two terms of this polynomial are fixed to 1 and  $-\frac{1}{2}$ , which are the first two terms of the Taylor expansion for cosine; this common practice guarantees higher accuracy near 0. Moreover, instead of using the standard Horner form,  $a_0 + x \cdot (a_1 + x \cdot (a_2 + \dots))$ , VDT computes the first two terms of the polynomial separately, as  $1.0 - zz * .5 + zz * zz * \text{get\_cos\_px}(zz)$ , where  $zz$  is equal to  $x^2$  and where `get\_cos\_px` implements the rest of the polynomial.

Splitting out the first term of the polynomial like this can reduce rounding error, and MegaLibm offers it via the `split` tuning parameter for polynomial terms. However, seemingly as the result

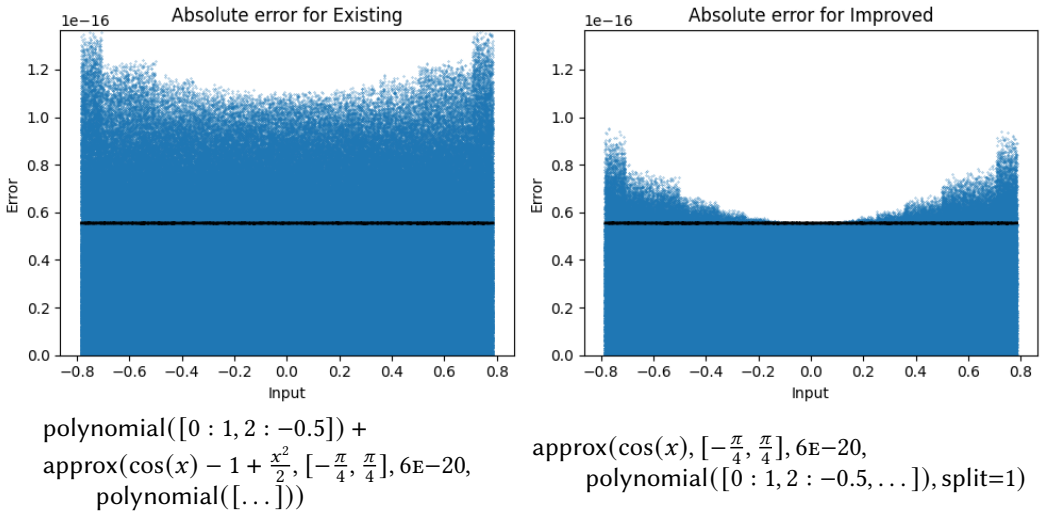


Fig. 10. Absolute error graphs and MegaLibm code for the core of the VDT cos implementation. On the left, our attempt to replicate VDT’s actual implementation; on the right, a more natural MegaLibm implementation that fixes a seeming bug in VDT’s cos. The bug fails to parenthesize  $1.0 - zz * .5$  in  $1.0 - zz * .5 + zz * zz * \text{get\_cos\_px}(zz)$ .

of an oversight, the VDT implementation adds the terms largest-to-smallest instead of smallest-to-largest.<sup>12</sup> In C, this is an easy mistake to make, since it depends on the associativity of the  $-$  and  $+$  operations. However, it dramatically increases the error (see Figure 10), with seemingly no benefits.<sup>13</sup> MegaLibm’s `split` tuning parameter does not have this bug, and achieves better accuracy; to replicate the actual VDT implementation for Figure 8 requires complex contortions (see Figure 10) and is still not exactly identical to VDT.

*AOCL’s many asins.* The AMD Optimizing CPU Libraries include not just one but four `asin` implementations: `ref` and `opt` variants with greater accuracy, and `fast` variants for lower accuracy but greater speed. We re-implemented all four in MegaLibm; all four share similar high-level algorithms, though the `fast` variants use fewer polynomial terms and the `ref` variant uses a modified Padé approximation instead of a polynomial approximation. Some variants even share polynomial coefficients. We also implemented the very similar `fdlibm` `asin` function. However, the five implementations are tuned differently: the `fast_asin` variant, for example, adds a `split` term, while the `opt_asinf` variant does most of the computation in double precision before casting to a single-precision result.

There is one aspect of these implementations that we were unable to replicate in MegaLibm. All four implementations use the identity  $\frac{\pi}{2} - 2 \cdot \text{asin}\left(\sqrt{(1-x)/2}\right)$  for range reduction and reconstruction. MegaLibm’s `right` operator ably expresses this reduction; however, the core polynomial approximation used for `asin` uses only even powers of the reduced value  $x$ . On range-reduced inputs, that would normally mean computing  $\sqrt{\dots}^2$ , an expression that introduces unnecessary error. The AOCL implementations save the  $(1-x)/2$  value before performing the square root to avoid this. MegaLibm’s `rewrite` operator cannot be used to similar effect because it operates on one

<sup>12</sup>We have reported this bug to the VDT developers.

<sup>13</sup>Changing the order of summation does shorten the arithmetic critical path, so could hypothetically lead to speed ups, but we were unable to measure this effect; in any case, our MegaLibm reimplementation is faster.

Library	Func	Domain	Error		Runtime		+
			Original	MLM	Original	MLM	
AOCL	fast_asin_better	[0.0, 0.5]	7.50e-17	<b>6.35e-17</b>	7.27	<b>7.20</b>	2
		[-1.0, 1.0]	<b>2.69e-16</b>	2.83e-16	<b>17.96</b>	18.52	
VDT	cos_better	[0.0, 0.78]	1.34e-16	<b>9.53e-17</b>	7.37	<b>7.19</b>	0
		[-4.0, 4.0]	1.42e-16	<b>1.12e-16</b>	7.44	<b>7.29</b>	
	cos_faster	[0.0, 0.78]	1.34e-16	<b>1.13e-16</b>	7.43	<b>7.25</b>	4
		[-4.0, 4.0]	1.42e-16	<b>1.12e-16</b>	7.17	<b>7.05</b>	
	exp_better	[-0.34, 0.34]	2.69e-16	<b>1.54e-16</b>	7.29	<b>7.23</b>	2
		[-20.0, 20.0]	6.40e-8	<b>4.45e-8</b>	<b>7.04</b>	<b>7.04</b>	
	exp_faster	[0.0, 50.0]	6.77e5	<b>5.81e5</b>	7.56	<b>7.55</b>	2
		[-0.34, 0.34]	2.69e-16	<b>1.53e-16</b>	7.38	<b>7.25</b>	
		[-20.0, 20.0]	6.40e-8	<b>4.96e-8</b>	<b>7.04</b>	<b>7.04</b>	
		[0.0, 50.0]	6.77e5	<b>5.81e5</b>	7.54	<b>7.54</b>	

Fig. 11. Variations on some of the reimplemented library functions. The “better” variations were made to decrease error, while the “faster” versions were made to increase speed. The “+” column counts lines added in the variation. This column does not subtract lines removed; typically, more lines are added than removed, because polynomial coefficients are re-synthesized.

IR node at a time, whereas the square root and the square operation are located in different nodes. In the future we hope to add cross-block matching to rewrite, allowing it to express this trick.

Unfortunately, our reimplementations of AOCL’s `ref_asin` function, and the similar `fdlibm asin` function, are dramatically slower than the original implementation. The slow-down is due to a fairly simple cause: the MegaLibm implementation compiles the whole reduction operation  $\sqrt{(1-x)}/2$  to double-double precision, but in fact the subtraction and division operations are exact and can be implemented with standard double precision. While it is possible to address this issue in MegaLibm, it would require changes to the implementation beyond simple tweaking of tuning parameters, and we felt such changes went against the spirit of MegaLibm. In the future, we hope to detect exact operations like this using interval analysis, leveraging the intervals available in MegaLibm’s type system, similar to Lee et al. [2017].

## 6.2 Tuning Implementations for Speed and Accuracy (RQ2)

To test that MegaLibm makes it easy to tune math library implementations, we construct 5 variations of the implementations in Figure 8 that use simple, widely applicable techniques to improve speed or accuracy. These variations change only a few lines, and never add more lines than they remove, and almost always result in implementations that are both faster and more accurate.

To improve the accuracy of AOCL’s `fast_asin`, we resynthesize the core function approximation using Sollya’s `fpmnimax` algorithm via MegaLibm. AOCL’s `fast_asin` actually already used coefficients synthesized in this way; however, instead of being directly synthesized for `fast_asin`, they were just the coefficients from `opt_asinf` but truncated to form a shorter polynomial. This is not quite as accurate as resynthesizing the polynomial. The result was somewhat more accurate over the core  $[0, 0.5]$  domain, but actually less accurate over the whole domain  $[-1, 1]$ . The resynthesized polynomial could still be useful (if, say, the user knows their inputs are in  $[-0.5, 0.5]$ ) but it also demonstrates an important aspect of MegaLibm’s design: floating point error is fundamentally not composable, so tuning and experimentation is often the best path. MegaLibm therefore focuses on making it easy to measure, tune, and test floating-point speed and accuracy.

Func	Domain	Error		Runtime	
		Lib	MLM	Lib	MLM
$1 - \cos(x)$	[-1.0, 1.0]	9.12e-17	<b>5.78e-17</b>	12.17	<b>7.50</b>
	[-2.0, 2.0]	<b>1.41e-16</b>	2.91e-16	15.87	<b>7.49</b>
	[-8.0, 8.0]	<b>1.97e-16</b>	8.72e-16	19.11	<b>7.48</b>
	[-32.0, 32.0]	<b>2.04e-16</b>	1.39e-15	19.08	<b>7.40</b>
$\sin(x) - x$	[-1.0, 1.0]	9.31e-17	<b>8.79e-17</b>	12.29	<b>7.54</b>
	[-2.0, 2.0]	<b>1.75e-16</b>	3.63e-16	16.36	<b>7.51</b>
	[-8.0, 8.0]	<b>5.31e-16</b>	9.70e-15	18.99	<b>7.61</b>
	[-32.0, 32.0]	<b>1.86e-15</b>	1.18e-14	18.63	<b>7.40</b>
$\sin(\pi x)$	[-1.0, 1.0]	<b>3.46e-16</b>	1.10e-15	22.36	<b>8.16</b>
	[-2.0, 2.0]	<b>6.85e-16</b>	1.03e-15	19.82	<b>7.49</b>
	[-8.0, 8.0]	2.73e-15	<b>1.12e-15</b>	19.52	<b>7.50</b>
	[-32.0, 32.0]	1.08e-14	<b>1.05e-15</b>	19.43	<b>7.51</b>

Fig. 12. Fully synthesized implementations of compound math functions and their straightforward C counterparts. All "lib" implementations are using the system libm.

We made a more substantial change to VDT's exp. The original implementation used an unusual rational polynomial  $1 + 2\frac{p}{q-p}$ , where  $p$  has three terms and  $q$  has four. We synthesized polynomial approximations using `remez`<sup>14</sup> with between 7 and 12 terms, and found that an 11-term polynomial approximation had comparable speed to the existing approximation. Note that, while this polynomial has more terms, it avoids an expensive division operation, meaning more terms can be used. The synthesized polynomial therefore has comparable speed while improving error substantially. We also tested a faster version, which changes the polynomial evaluation scheme from Horner to Estrin. We expected this to produce a speedup, because 11-term polynomials are long enough for critical path length to become important. However, we were unable to measure any difference, again underscoring the importance of measurement.

Finally, noting the seeming oversight in VDT's cos implementation, we implemented a "fixed" version of VDT cos, which is substantially more accurate than the original. We also experimented with a faster version, which resynthesizes the polynomial coefficients, uses Estrin polynomial evaluation, and splits out one term instead of two. This was slightly faster on the full range, likely the result of Estrin's shorter critical path, but less accurate on the narrow range. This again demonstrates the benefit of MegaLibm's empirical, interactive design: we do not know of a systematic way to estimate the costs and benefits of Estrin over Horner form, but testing both options made the trade-offs clear.

### 6.3 Automated Synthesis of New Libraries (RQ3)

To test the MegaLibm's combined type-directed and Sollya-based synthesis, we constructed implementations of  $1 - \cos(x)$ ,  $\sin(x) - x$ , and  $\sin(\pi x)$  entirely from scratch, and compared them to the naive implementation of these functions where, for example,  $\sin(\pi x)$  is implemented as `sin(M_PI * x)`, with `sin` referencing the Glibc standard library. To synthesize each implementation, we constructed a hole of the appropriate type and called `synthesize`. Polynomials were set to be 14 terms and created using the `remez` algorithm. No further tuning was performed; when MegaLibm produced multiple expressions, the most accurate was chosen. Results are shown in Figure 12.

<sup>14</sup>While `fminimax` is typically more accurate, `remez` is much faster so a better match for testing many sizes.

All synthesized implementations are significantly faster than the naive library-based implementation. For  $1 - \cos(x)$  and  $\sin(x) - x$ , the MegaLibm-synthesized implementations are more accurate on narrow domains, but become less accurate as the domain widens; this is because MegaLibm-synthesized implementations by default use the naive method for periodic terms instead of the more-accurate cody-waite method. For  $\sin(\pi x)$ , by contrast, the MegaLibm-synthesized implementation shows a constant accuracy as the domain widens, while the naive library-based implementation grows progressively less accurate as the domain grows wider. This is because in the naive library-based implementation, multiplying by  $\pi$  and then reducing modulo  $2\pi$  introduces error, while the MegaLibm-synthesized implementation can perform a fast periodic reduction by 2, which is exact. Each of these implementations can be further tuned by the user, including tuning the periodic reduction method, introducing higher accuracy, or switching to `fpminimax` polynomial approximations. While MegaLibm’s synthesized methods cannot match hand-tuned expert-written functions, they provide an alternative for users without the expertise to write their own.

## 7 RELATED WORK

Mathematical function implementation, at its core, replaces a hard-to-compute function and with simpler operations that are easily computable. Work in this area starts before electric computers with the work of people like Taylor [Taylor 1717], Chebyshev [Tchébychev 1858], and Remez [Remez and Gavriilyuk 1968], who created the foundations of approximation theory. This work gained practical importance with the advent of vacuum tube computers [Hastings 1955]. As computers increased in speed, control over accuracy became important, with important contributions by Dekker [1971], Kahan [1967, 2004], and Higham [2002]. The roots of modern math libraries go back to Cody and Waite’s “Software Manual for the Elementary Functions” [Cody and Waite 1980], which provided a detailed guide for many common functions. Modern references on math function implementation include the “Mathematical-Function Computation Handbook” [Beebe 2017] and “Elementary Functions” [Muller 2016].

Many extant math libraries trace coefficients and code to a couple of especially important historic libraries, chief among them Sun’s `fdlibm` library [Microsystems 1993]. Other widely-used libraries include GNU’s `GLibC` math functions [FSF 2020], the `OpenLibm` project [Julia Math Project 2021], `Cephes` [Moshier 1992], `VDT` [Piparo et al. 2014], Intel’s `MKL` [Intel 2020], and AMD’s `AOCL` [AMD 2021]. All of these libraries achieve accuracy of about 1 ULP of error for most functions [Free Software Foundation 2020; Zimmermann 2021]. Four functions from Intel’s `MKL` library have had their accuracy verified with semi-automated methods [Lee et al. 2017], though it is unclear whether this approach scales to other libraries or functions. A variety of tools support the development of these libraries. `Sollya` [S. et al. 2010] provides several polynomial approximation algorithms, and the `Metalibm` Python library [Kupriianova and Lauter 2014] provides a programmatic interface to it. Specialized math software such as `Magma` [Bosma et al. 1997], `Maple` [Maplesoft, a division of Waterloo Maple Inc.. 2019], `Mathematica` [Research Inc. 2023], `Matlab` [Inc. 2022] (via the `Chebfun` package [Driscoll et al. 2014]), and `Sage Math` [The Sage Developers 2023] also provide polynomial or rational function approximation packages. Other tools, such as `Flopoco` [de Dinechin 2019], are specialized to hardware implementation. None of these tools, however, provide a high-level language for math function implementation that provides safety, modularity, and tunability.

More recently, new research has suggested paths toward *correctly rounded* implementations, which achieve the lowest possible error of  $\frac{1}{2}$  ULP on all inputs. The `CRLibm` library [Daramy et al. 2003] is one product of this line of work, as are the `RLibm` project [Aanjaneya et al. 2022; Aanjaneya and Nagarakatte 2023; Lim et al. 2021; Lim and Nagarakatte 2022] and the in-progress `CORE-Math` libraries [Sibidanov et al. 2022]. The `MPFR` library also offers correctly rounding implementations of the core functions at arbitrary precision [Fousse et al. 2007], though at high

runtime cost, and libraries that track their precision, such as Arb [Johansson 2017], can in some cases play a similar role. Recent advances allows modern correctly rounded libraries to match the performance of classic libraries such as Glibc or fdlibm while also providing the ultimate accuracy guarantee [Group 2023]. All of these implementations still require careful design of range reduction and reconstruction algorithms, and could potentially be written in MegaLibm.

Even an accurate library function does not guarantee accurate user code. Many researchers have thus attempted to develop tools to bound the worst-case error from a given computation [Bard et al. 2019; Darulova et al. 2018; Das et al. 2020; Solovyev et al. 2018] or even to discover specific high-error inputs [Chiang et al. 2014; Guo and Rubio-González 2020]. Those high-error inputs could then be debugged [Benz et al. 2012; Chowdhary et al. 2020; Sanchez-Stern et al. 2018] and the code rewritten to improve accuracy. The Herbie tool [Panchekha et al. 2015] attempts to automatically rewrite math expressions for higher accuracy and, in recent version, greater speed [Saiki et al. 2021]. Other packages, such as Salsa [Damouche and Martel 2018], Daisy [Darulova et al. 2018], Precimonious [Rubio-González et al. 2013], FPTuner [Chiang et al. 2017], HiFPTuner [Guo and Rubio-González 2018], and POP [Ben Khalifa and Martel 2022] provide automated precision adjustment. Some authors have proposed tools to select [Briggs and Panchekha 2022] or synthesize [Darulova and Volkova 2019] the optimal function implementation for a particular call site. These tools must be used in concert with MegaLibm to achieve the desired accuracy and speed for application code.

## 8 CONCLUSION

Implementing math functions has long been challenging because the programming languages necessary for high performance did not provide safety, modularity, or tunability. This implied a top-down, monolithic, one-shot implementation style that demanded experience and expertise. MegaLibm addresses this with a high-level DSL where math library implementations can be expressed, tuned, and then compiled to highly accurate, highly performant code. Implementations in MegaLibm are automatically checked for syntactic and semantic wellformedness, and tuning parameters allow low-level control of speed and accuracy while remaining separate from high-level algorithm design. MegaLibm thus allows users—even, thanks to synthesis, users without extensive expertise—to interactively write and tune math function implementations. We used MegaLibm to re-implement 8 state-of-the-art math libraries with comparable (often better) speed and accuracy, make 5 improved versions, and synthesize 3 implementations from scratch.

In the future, we hope to extend MegaLibm to incorporate more techniques used in state-of-the-art math libraries. The biggest gap in MegaLibm is the use of lookup tables in computations, which have become increasingly important. Support for tabular approximations is necessary for new implementations such as RLibm [Lim et al. 2021] and CORE-Math [Sibidanov et al. 2022]. A key design question is how to support mixed methods such as tables of polynomial coefficients or polynomial interpolation schemes. RLibm could be integrated as a synthesis tool for tabular approximations. We also hope to integrate algorithms such as Payne-Hanek reduction or compensated Horner evaluation. We would like to support number representations such as half-precision, bfloat16 [Wang and Kanwar 2019], posits [Behnam and Bojnordi 2020], and FP8 [Micievicus et al. 2022], and improve support for compound representations such as triple-/quadruple-double support. We'd also like to use interval analysis and uninterpreted sums to improve the existing double-double support. Finally, we'd like to add formal verification backends to MegaLibm, so that users could send MegaLibm implementations to Gappa [Gustafson and Yonemoto 2017], FPTaylor [Solovyev et al. 2018], Satire [Das et al. 2020], or Daisy [Darulova et al. 2018]. While full formal verification of a math library implementation is likely still beyond the state of the art, these tools would be useful during interactive development.

## ACKNOWLEDGMENTS

We also thank our anonymous reviewers for their guidance and valuable suggestions while preparing the final version of this paper. This work was supported by NSF award 2119939. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, ComPort: Rigorous Testing Methods to Safeguard Software Porting, under Award Number 10061193.

## DATA AVAILABILITY STATEMENT

The source code for MegaLibm has been made available at the following GitHub page <https://github.com/IanBriggs/megalibm>. Examples of using Jupyter notebook to drive the system are given in the examples directory.

## REFERENCES

- Mridul Aanjaneya, Jay P. Lim, and Santosh Nagarakatte. 2022. Progressive Polynomial Approximations for Fast Correctly Rounded Math Libraries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) (PLDI 2022). Association for Computing Machinery, New York, NY, USA, 552–565. <https://doi.org/10.1145/3519939.3523447>
- Mridul Aanjaneya and Santosh Nagarakatte. 2023. Fast Polynomial Evaluation for Correctly Rounded Elementary Functions Using the RLIBM Approach. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal, QC, Canada) (CGO 2023). Association for Computing Machinery, New York, NY, USA, 95–107. <https://doi.org/10.1145/3579990.3580022>
- AMD. 2021. AMD Math Library (LibM). <https://developer.amd.com/amd-aocl/amd-math-library-libm/>
- Joachim Bard, Heiko Becker, and Eva Darulova. 2019. Formally Verified Roundoff Errors Using SMT-based Certificates and Subdivisions. In *Formal Methods – The Next 30 Years*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer International Publishing, Cham, 38–44.
- Nelson H. F. Beebe. 2017. The Mathematical-Function Computation Handbook. In *Cambridge International Law Journal*.
- Payman Behnam and Mahdi Bojnordi. 2020. Posit: A Potential Replacement for IEEE 754. <https://www.sigarch.org/posit-a-potential-replacement-for-ieee-754/>
- Dorra Ben Khalifa and Matthieu Martel. 2022. Constrained Precision Tuning. In *8th International Conference on Control, Decision and Information Technologies, CoDIT 2022, Istanbul, Turkey, May 17-20, 2022*. IEEE, 230–236. <https://doi.org/10.1109/CoDIT55151.2022.9804011>
- Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A Dynamic Program Analysis to Find Floating-Point Accuracy Problems. *SIGPLAN Not.* 47, 6 (jun 2012), 453–462. <https://doi.org/10.1145/2345156.2254118>
- Hans-J. Boehm. 2020. Towards an API for the Real Numbers. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 562–576. <https://doi.org/10.1145/3385412.3386037>
- Wieb Bosma, John Cannon, and Catherine Playoust. 1997. The Magma algebra system. I. The user language. *J. Symbolic Comput.* 24, 3–4 (1997), 235–265. <https://doi.org/10.1006/jsc.1996.0125> Computational algebra and number theory (London, 1993).
- Ian Briggs and Pavel Panchekha. 2022. Choosing Mathematical Function Implementations for Speed and Accuracy. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) (PLDI 2022). Association for Computing Machinery, New York, NY, USA, 522–535. <https://doi.org/10.1145/3519939.3523452>
- Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (POPL 2017). Association for Computing Machinery, New York, NY, USA, 300–315. <https://doi.org/10.1145/3009837.3009846>
- Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamarić, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 43–52.
- Sangeeta Chowdhary, Jay P. Lim, and Santosh Nagarakatte. 2020. Debugging and Detecting Numerical Errors in Computation with Posits. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 731–746. <https://doi.org/10.1145/3385412.3386004>

- William James Cody and William Waite. 1980. *Software Manual for the Elementary Functions (Prentice-Hall series in computational mathematics)*. Prentice-Hall, Inc., USA.
- Nasrine Damouche and Matthieu Martel. 2018. Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs. In *Kalpa Publications in Computing*, Vol. 5. EasyChair, 63–76. <https://doi.org/10.29007/j2fd> ISSN: 2515-1762.
- Catherine Daramy, David Defour, Florent Dinechin, and Jean-Michel Muller. 2003. CR-LIBM: A correctly rounded elementary function library. *Proceedings of SPIE - The International Society for Optical Engineering* 5205 (Dec. 2003). <https://doi.org/10.1117/12.505591>
- Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. 2018. *Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper)*. 270–287. [https://doi.org/10.1007/978-3-319-89960-2\\_15](https://doi.org/10.1007/978-3-319-89960-2_15)
- Eva Darulova and Anastasia Volkova. 2019. Sound Approximation of Programs with Elementary Functions. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 174–183. [https://doi.org/10.1007/978-3-030-25543-5\\_11](https://doi.org/10.1007/978-3-030-25543-5_11)
- Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panekha. 2020. Scalable yet Rigorous Floating-Point Error Analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Atlanta, Georgia) (SC '20)*. IEEE Press, Article 51, 14 pages.
- Florent de Dinechin. 2019. Reflections on 10 years of FloPoCo. In *26th IEEE Symposium of Computer Arithmetic (ARITH-26)*.
- T. J. Dekker. 1971. A floating-point technique for extending the available precision. *Numer. Math.* 18, 3 (1971), 224–242. <https://doi.org/10.1007/BF01397083>
- T. A Driscoll, N. Hale, and L. N. Trefethen. 2014. *Chebfun Guide*. Pafnuty Publications. <http://www.chebfun.org/docs/guide/>
- Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Software* 33, 2 (June 2007), 13–es. <https://doi.org/10.1145/1236463.1236468>
- Free Software Foundation. 2020. Errors in Math Functions (The GNU C Library). [https://www.gnu.org/software/libc/manual/html\\_node/Errors-in-Math-Functions.html](https://www.gnu.org/software/libc/manual/html_node/Errors-in-Math-Functions.html)
- FSF. 2020. The GNU C Library. <https://www.gnu.org/software/libc/manual/>
- LLVM Developer Group. 2023. Math Functions. <https://libc.llvm.org/math/index.html>
- Sumit Gulwani, Ashish Tiwari, and George C. Necula. 2005. Join Algorithms for the Theory of Uninterpreted Functions. In *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, Kamal Lodaya and Meena Mahajan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 311–323.
- Hui Guo and Cindy Rubio-González. 2020. Efficient Generation of Error-Inducing Floating-Point Inputs via Symbolic Execution. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (Seoul, South Korea) (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1261–1272. <https://doi.org/10.1145/3377811.3380359>
- Hui Guo and Cindy Rubio-González. 2018. Exploiting community structure for floating-point precision tuning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 333–343. <https://doi.org/10.1145/3213846.3213862>
- John L. Gustafson and Isaac T. Yonemoto. 2017. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomput. Front. Innov.* 4, 2 (2017), 71–86. <https://doi.org/10.14529/jsfi170206>
- Cecil Hastings. 1955. Approximations for Digital Computers. *Science* 122, 3170 (1955), 602–602. <https://doi.org/10.1126/science.122.3170.602.a> arXiv:<https://www.science.org/doi/pdf/10.1126/science.122.3170.602.a>
- Yozo Hida, Sherry Li, and David Bailey. 2008. Library for Double-Double and Quad-Double Arithmetic. (01 2008).
- Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics. <https://doi.org/10.1137/1.9780898718027>
- MathWorks Inc. 2022. *MATLAB version: 9.13.0 (R2022b)*. Natick, Massachusetts, United States. <https://www.mathworks.com>
- Intel. 2020. Intel-Optimized Math Library for Numerical Computing. <http://software.intel.com/en-us/intel-mkl>
- F. Johansson. 2017. Arb: efficient arbitrary-precision midpoint-radius interval arithmetic. *IEEE Trans. Comput.* 66 (2017), 1281–1292. Issue 8. <https://doi.org/10.1109/TC.2017.2690633>
- Julia Math Project. 2021. JuliaMath/OpenLibm. <https://github.com/JuliaMath/openlibm>
- William Kahan. 1967. 7094-11 System support for numerical analysis. In *Proceedings*. US Army Research Office., 175.
- William Kahan. 2004. A Logarithm Too Clever by Half. <http://people.eecs.berkeley.edu/~wkahan/LOG10HAF.TXT>
- Dexter Kozen. 1977. Complexity of Finitely Presented Algebras. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing (Boulder, Colorado, USA) (STOC '77)*. Association for Computing Machinery, New York, NY, USA, 164–177. <https://doi.org/10.1145/800105.803406>
- Olga Kupriianova and Christoph Lauter. 2014. Metalibm: A Mathematical Functions Code Generator. In *Mathematical Software – ICMS 2014 (Lecture Notes in Computer Science)*, Hoon Hong and Chee Yap (Eds.). Springer, Berlin, Heidelberg, 713–717. [https://doi.org/10.1007/978-3-662-44199-2\\_106](https://doi.org/10.1007/978-3-662-44199-2_106)
- Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2017. On automatically proving the correctness of math.h implementations. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec. 2017), 47:1–47:32. <https://doi.org/10.1145/3158135>

- Jay P. Lim, Mridul Aanjaneya, John Gustafson, and Santosh Nagarakatte. 2021. An Approach to Generate Correctly Rounded Math Libraries for New Floating Point Variants. *Proc. ACM Program. Lang.* 5, POPL, Article 29 (jan 2021), 30 pages. <https://doi.org/10.1145/3434310>
- Jay P. Lim and Santosh Nagarakatte. 2022. One Polynomial Approximation to Produce Correctly Rounded Results of an Elementary Function for Multiple Representations and Rounding Modes. *Proc. ACM Program. Lang.* 6, POPL, Article 3 (jan 2022), 28 pages. <https://doi.org/10.1145/3498664>
- Maplesoft, a division of Waterloo Maple Inc.. 2019. *Maple*. Waterloo, Ontario. <https://www.maplesoft.com>
- Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. 2022. FP8 Formats for Deep Learning. arXiv:2209.05433 [cs.LG]
- Sun Microsystems. 1993. A Freely Distributable Libm. (1993). <https://www.netlib.org/fdlibm/>
- Stephen L Moshier. 1992. Cephes mathematical library. <http://www.netlib.org/cephes/>
- Jean-Michel Muller. 2016. *Elementary Functions: Algorithms and Implementation* (3 ed.). Birkhäuser Basel. <https://doi.org/10.1007/978-1-4899-7983-4>
- Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) (PLDI '15). Association for Computing Machinery, New York, NY, USA, 1–11. <https://doi.org/10.1145/2737924.2737959>
- Danilo Piparo, Vincenzo Innocente, and Thomas Hauth. 2014. Speeding up HEP experiment software with a library of fast and auto-vectorisable mathematical functions. *Journal of Physics: Conference Series* 513, 5 (June 2014), 052027. <https://doi.org/10.1088/1742-6596/513/5/052027> Publisher: IOP Publishing.
- Evgenii Yakovlevich Remez and Vera Timofeevna Gavrilyuk. 1968. A theorem on interpolation functions which provides a fundamental approach to the treatment of general analogs of the method of successive Chebyshev interpolations. In *Doklady Akademii Nauk*, Vol. 183. Russian Academy of Sciences, 750–753.
- Wolfram Research Inc. 2023. Mathematica, Version 13.3. <https://www.wolfram.com/mathematica> Champaign, IL.
- C. Rubio-González, Cuong Nguyen, Hong Diep Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. <https://doi.org/10.1145/2503210.2503296> ISSN: 2167-4337.
- Chevillard S., Joldeş M., and Lauter C. 2010. Sollya: An Environment for the Development of Numerical Codes. In *Mathematical Software - ICMS 2010 (Lecture Notes in Computer Science, Vol. 6327)*, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama (Eds.). Springer, Heidelberg, Germany, 28–31.
- Brett Saiki, Oliver Flatt, Chandrakana Nandi, Pavel Panchekha, and Zachary Tatlock. 2021. Combining Precision Tuning and Rewriting. In *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*.
- Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2018. Finding Root Causes of Floating Point Error. *SIGPLAN Not.* 53, 4 (jun 2018), 256–269. <https://doi.org/10.1145/3296979.3192411>
- Alexei Sibidanov, Paul Zimmermann, and Stéphane Gloudu. 2022. The CORE-MATH Project. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*. 26–34. <https://doi.org/10.1109/ARITH54963.2022.00014>
- Alexey Solovyyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Trans. Program. Lang. Syst.* 41, 1, Article 2 (dec 2018), 39 pages. <https://doi.org/10.1145/3230733>
- Pat H Sterbenz. 1974. *Floating-point computation*. Prentice-Hall, Englewood Cliffs, NJ. <https://cds.cern.ch/record/268412>
- Brook Taylor. 1717. *Methodus incrementorum directa & inversa*. Inny.
- P Tchébychev. 1858. *Sur les questions de minima qui se rattachent a la représentation aproximative des fonctions*. Imprimerie de l'Academie Impériale des Sciences.
- The Sage Developers. 2023. *SageMath, the Sage Mathematics Software System (Version x.y.z)*. <https://www.sagemath.org>
- Shibo Wang and Pankaj Kanwar. 2019. BFloat16: The secret to high performance on Cloud TPUs. <https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus/>
- Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021), 29 pages. <https://doi.org/10.1145/3434304>
- Paul Zimmermann. 2021. Accuracy of Mathematical Functions in Single, Double, Extended Double and Quadruple Precision. (Feb. 2021). <https://hal.inria.fr/hal-03141101> working paper or preprint.

Received 2023-07-11; accepted 2023-11-07