High Performance Correctly Rounded Math Libraries for 32-bit Floating Point Representations

Jay P. Lim
Department of Computer Science
Rutgers University
United States
jpl169@cs.rutgers.edu

Santosh Nagarakatte
Department of Computer Science
Rutgers University
United States
santosh.nagarakatte@cs.rutgers.edu

Abstract

This paper proposes a set of techniques to develop correctly rounded math libraries for 32-bit float and posit types. It enhances our RLibm approach that frames the problem of generating correctly rounded libraries as a linear programming problem in the context of 16-bit types to scale to 32-bit types. Specifically, this paper proposes new algorithms to (1) generate polynomials that produce correctly rounded outputs for all inputs using counterexample guided polynomial generation, (2) generate efficient piecewise polynomials with bit-pattern based domain splitting, and (3) deduce the amount of freedom available to produce correct results when range reduction involves multiple elementary functions. The resultant math library for the 32-bit float type is faster than state-of-the-art math libraries while producing the correct output for all inputs. We have also developed a set of correctly rounded elementary functions for 32-bit posits.

CCS Concepts: • **Mathematics of computing → Mathematical software**; **Linear programming**; • **Theory of computation → Numeric approximation algorithms**.

Keywords: elementary functions, correctly rounded math libraries, floating point, posits, piecewise polynomials

ACM Reference Format:
Jay P. Lim and Santosh Nagarakatte. 2021. High Performance Correctly Rounded Math Libraries for 32-bit Floating Point Representations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454049

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454049

1 Introduction

Math libraries provide implementations of elementary functions (*e.g.* $log(x)$, e^x, $cos(x)$) [37]. They are crucial components in various domains ranging from scientific computing to machine learning. Designing math libraries is a challenging task because they are expected to provide correct results for all inputs and also have high performance. These elementary functions are typically approximated with some hardware supported representation for performance.

Given a representation $\mathbb{T}$ with finite precision (*e.g.*, `float`), the correctly rounded result of an elementary function f for an input $x \in \mathbb{T}$ is defined as the value of $f(x)$ computed with real numbers and then rounded to a value in the representation $\mathbb{T}$. The IEEE-754 standard recommends the generation of correctly rounded results for elementary functions. Seminal prior work on generating approximations for elementary functions has resulted in numerous implementations that have reduced error significantly [5–10, 15, 24, 25, 29, 49]. Further, numerous correctly rounded libraries have also been developed [13, 15]. Unfortunately, they are not widely used due to performance considerations. Moreover, widely used libraries do not produce correct results for all inputs.

Mini-max approaches. Most prior approaches identify a polynomial that minimizes the maximum error among all input points (*i.e.*, a mini-max approach) compared to the real value of the elementary function using the Weierstrass approximation theorem and the Chebyshev alternation theorem [47]. The Weierstrass approximation theorem states that if f is a continuous real-valued function on $[a, b]$ and $\epsilon > 0$, there exists a polynomial P such that $|f(x) - P(x)| < \epsilon$ for all $x \in [a, b]$. The Chebyshev alternation theorem provides the condition for such a polynomial: a polynomial of degree d that minimizes the maximum error will have at least $d + 2$ points where it has the absolute maximum error and the error alternates in sign. Remez algorithm [37, 39] is a procedure to identify such mini-max polynomials. The maximum approximation error has to be below the error threshold required to produce correct results for all inputs.

As approximating a polynomial in a small domain $[a, b]$ is much easier, the input domain of the function is reduced using range reduction [12, 31, 45]. The approximated result is adjusted to produce the result for the original input (*i.e.*, output compensation). Both range reduction and polynomial

evaluation in a representation with finite precision will have some numerical errors. The combination of approximation errors with the mini-max approach and numerical errors with polynomial evaluation, range reduction, and output compensation can result in wrong results.

RLibm. Our RLibm approach [31, 32] generates polynomials that approximate the correctly rounded result rather than the real value of the elementary function. The generation of the polynomial considers errors in polynomial approximation and numerical errors in polynomial evaluation, range reduction, and output compensation to produce the correctly rounded output for all inputs. The task of generating the polynomial is then structured as a linear programming (LP) problem. The RLibm approach first computes the correctly rounded result for each input in a target representation $\mathbb{T}$ using an oracle (*e.g.*, the MPFR library [15]). Given the correctly rounded result for an input, it finds an interval in `double` precision such that every value in the interval rounds to the correctly rounded result, which is called the rounding interval. The rounding intervals are further constrained to account for numerical errors during range reduction and output compensation. Subsequently, it attempts to generate a polynomial of degree d using an LP solver, which when evaluated with an input produces a result that lies within the rounding interval. Using the RLibm approach, we have been successful in generating correctly rounded libraries with 16-bit types such as `bfloat16` and `posit16`.

Challenges in scaling to 32-bits. To extend our RLibm approach to 32-bit data types, we have to address the following challenges. First, modern LP solvers can handle a few thousand constraints. A naive use of the RLibm approach with 32-bit types will generate more than a billion constraints, which is beyond the capabilities of current LP solvers. Second, it may not be feasible to generate a single polynomial of a reasonable degree given the large number of constraints. Third, LP solvers are sensitive to the condition number of the system of constraints. LP solvers will not be able to solve an ill-conditioned system of constraints. An effective range reduction is a strategy to address it. Although there are excellent books on range reduction [12], these techniques need to be adapted to work with our RLibm approach. Fourth, some range reduction strategies need multiple elementary functions themselves (*e.g.*, $sinpi(x)$). Finally, we need to ensure that output compensation does not experience pathological cancellation errors (*e.g.*, $cospi(x)$).

This paper. Our goal is to generate efficient implementations of elementary functions that produce correctly rounded results for all inputs with 32-bit types. This paper extends our RLibm approach to scale to 32-bit FP types to address the challenges described above. We propose (1) sampling of inputs with counterexample guided polynomial generation to handle the large input space, (2) generation of piecewise polynomials for efficiency, (3) deduction of rounding intervals when a range reduction technique uses multiple elementary functions, and (4) modified range reduction techniques for some elementary functions to address cancellation errors in output compensation. Figure 1 pictorially represents our approach to scale to 32-bit data types.

Counterexample guided polynomial generation. We sample inputs proportional to the number of representable values in a given input domain $[a, b]$ with a 32-bit representation $\mathbb{T}$. To generate polynomials that produce the correctly rounded result for every input, it is not necessary to consider every input and its rounding interval. We primarily need to consider those rounding intervals that are highly constrained. For each input in the sample, we generate the oracle result using the MPFR library. We compute the rounding interval in double precision (*i.e.*, set of values in the `double` type that round to the oracle result). We generate LP constraints to create a polynomial of degree d such that it evaluates to a value in the rounding interval for each input in the sample. If the initial sample generates a polynomial that produces the correctly rounded output for all values in $[a, b]$, then the process terminates. Otherwise, we add counterexamples to the sample and repeat the process. The size of the sample is bounded by the number of constraints that the LP solver can process.

Piecewise polynomials. When either the number of inputs in the sample exceeds our LP constraint threshold or the LP solver is not able to generate a polynomial, we split the input domain $[a, b]$ to $[a, b')$ and $[b', b]$ to generate piecewise polynomials using the above process for each input sub-domain. We choose the splitting point such that we can identify the sub-domain quickly using a few bits of the input, which results in efficient implementations. The ability to generate piecewise polynomials ensures that our resultant polynomials are of a lower degree and provide performance improvements when compared to state-of-the-art libraries.

Range reduction with multiple functions. We propose new algorithms to deduce rounding intervals for a class of range reduction techniques that involve multiple elementary functions. Range reduction reduces the input x to x'. The creation of the polynomial happens with the reduced inputs. The output of the polynomial $P(x')$ should be adjusted to compute the correctly rounded result for x, which is called output compensation. We have to deduce the rounding intervals for the reduced input x' that considers the numerical error in range reduction, polynomial evaluation, and output compensation. We propose new techniques to create reduced rounding intervals when range reduction uses multiple elementary functions (*e.g.*, $sinpi(x)$ in Section 2). These techniques allow us to perform range reduction on functions that otherwise cause condition number issues with the LP formulation (*i.e.*, $sinh(x)$ or $cosh(x)$). Further, we develop modified range reduction techniques for some elementary functions to avoid cancellation errors in output compensation (*e.g.*, $cospi(x)$ in Section 5).

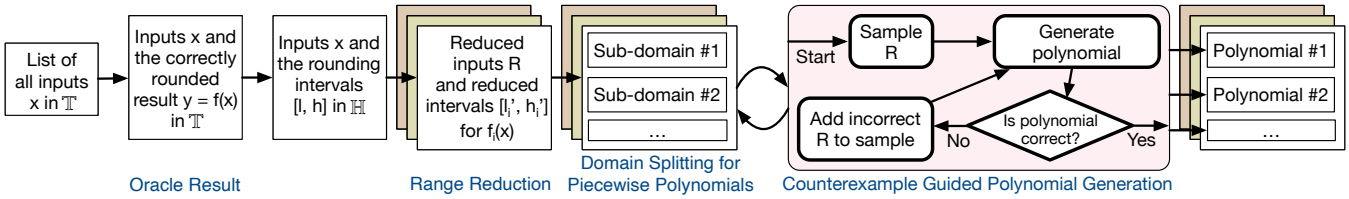

Figure 1. Steps in our approach to generate correctly rounded libraries for 32-bit types ($\mathbb{T}$).

The RLIBM-32 prototype. We have developed library generators and correctly rounded libraries for multiple 32-bit data types: IEEE-754 float and posits. Our elementary functions for `floats` are faster than existing libraries: Intel's libm, Glibc's libm, CR-LIBM [13], and Metalibm [25]. Unlike existing libraries, our functions produce correctly rounded results for all inputs. We have developed the first correctly rounded implementations of functions for 32-bit posits.

2 Overview of Our Approach with $sinpi(x)$

We provide an overview of our approach for generating piecewise polynomials for $sinpi(x)$ (*i.e.*, $sin(\pi x)$) with a 32-bit float. The function $sinpi(x)$ is defined for $x \in (-\infty, \infty)$. There are four billion inputs with a 32-bit float. There are three kinds of special cases:

$$sinpi(x) = \begin{cases} \pi x & \text{if } |x| < 1.173\cdots \times 10^{-7} \\ 0 & \text{if } |x| >= 2^{23} \\ NaN & \text{if } x = NaN \text{ or } x = \pm\infty \end{cases}$$

For the first class of special cases, we compute πx in double and round the result to float, which produces the correctly rounded result for those inputs.

2.1 Our Range Reduction for $sinpi(x)$

After considering special cases, there are close to 800 million float inputs that need to be approximated with polynomials. Using RLIBM's approach directly with an LP solver will fail. Next, we perform range reduction to reduce the domain for polynomial approximation. The key idea is to use periodicity and trigonometric identities of $sinpi(x)$. We transform input x into $x = 2.0 \times I + J$, where I is an integer and $J \in [0, 2)$. As a result of periodicity, $sinpi(x) = sinpi(J)$. Next, we further split J into $J = K + L$ where $K \in \{0, 1\}$ is the integral part of J and $L \in [0, 1)$ is the fractional part. Then, $sinpi(J)$ can be computed as,

$$sinpi(J) = (-1)^K \times sinpi(L)$$

Given that $sinpi$ between $[0.5, 1)$ is a mirror image of values between $[0, 0.5)$, we further reduce as follows:

$$L' = \begin{cases} L & \text{if } L \leq 0.5 \\ 1.0 - L & \text{if } L > 0.5 \end{cases}$$

From Sterbenz lemma [42], the expression $1.0 - L$ can be computed exactly. Hence, $sinpi(L) = sinpi(L')$. Even after reducing the input x to $L' \in [0, 0.5]$, there are around 184 million inputs with a 32-bit float in this reduced domain.

To enable easier polynomial approximation, we further reduce L' to a value between $[0, \frac{1}{512}]$. We split L' as $L' = \frac{N}{512} + R$ where N is an integer in the set $\{0, 1, \ldots, 255\}$ and R is a fraction that lies in $[0, \frac{1}{512}]$. There are 110 million reduced inputs in R ignoring special cases. Now, $sinpi(L')$ can be computed using the trigonometric identity $sinpi(a + b) = sinpi(a)cospi(b) + cospi(a)sinpi(b)$ as follows,

$$sinpi(L') = sinpi(\frac{N}{512})cospi(R) + cospi(\frac{N}{512})sinpi(R)$$

We precompute the values for $sinpi(\frac{N}{512})$ and $cospi(\frac{N}{512})$ in lookup tables (*i.e.*, 512 values in total). Finally, we approximate $sinpi(R)$ and $cospi(R)$ for the reduced input domain $R \in [0, \frac{1}{512}]$. *To approximate $sinpi(x)$ for the entire domain, the range reduction requires us to approximate sinpi and cospi over the reduced domain R.* We can compute the result for $sinpi(x)$ as follows,

$$sinpi(x) = (-1)^K \times (sinpi(\frac{N}{512})cospi(R) + cospi(\frac{N}{512})sinpi(R))$$

2.2 Generating Piecewise Polynomials for $sinpi(x)$

To produce correctly rounded results for $sinpi(x)$, our approach involves the following steps. First, we identify the correctly rounded result and the rounding interval for each input in the entire domain. Second, we identify the reduced rounding interval after range reduction. Third, we split the reduced domain into sub-domains to generate piecewise polynomials. Fourth, we perform counterexample guided polynomial generation for each sub-domain. Finally, we validate the generated piecewise polynomials for the entire input domain.

Step 1: Identifying the correctly rounded result and the rounding interval. For each input x, we first identify the correctly rounded result of $sinpi(x)$ using an oracle. Then, we identify an interval of values $[l, h]$ in `double` where all values in the interval rounds to the correctly rounded result. We call this interval the rounding interval. If our

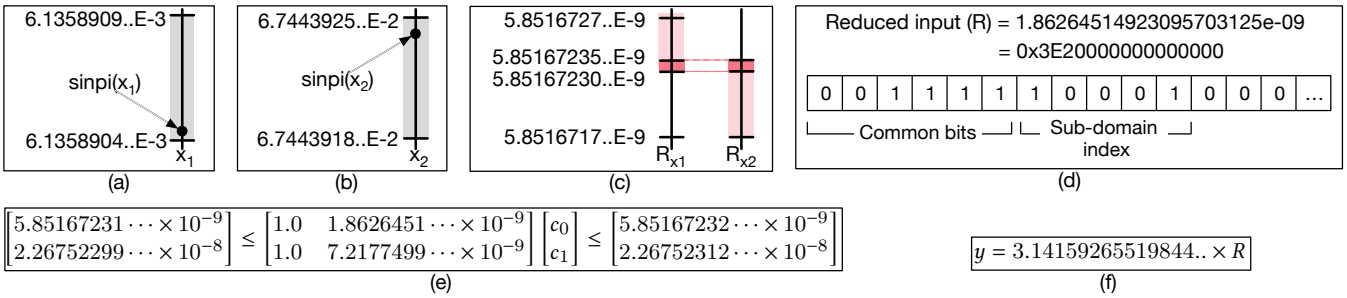

Figure 2. (a) A 32-bit float input $x_1 = 1.95\cdots\times10^{-3}$ and its correctly rounded result of $sinpi(x_1)$ (shown with a black circle). The rounding interval in double is colored gray. (b) Input $x_2 = 2.14\cdots\times10^{-2}$, its correctly rounded result for $sinpi(x_2)$, and the rounding interval. (c) The reduced intervals for $sinpi(R)$ corresponding to x_1 and x_2 so that both x_1 and x_2 produce correctly rounded results, respectively. Both x_1 and x_2 map to the same reduced input R. The common interval for R is highlighted with darker color. (d) To approximate $sinpi(R)$, we create a piecewise polynomial with 32 sub-domains in total. We use the 5-bits in the double representation of R to identify the sub-domain for the piecewise polynomial. (e) Our LP formulation for generating a piecewise polynomial of degree 1 for the sub-domain with bit-pattern (10001) with two reduced inputs in the sample. (f) The resulting coefficients returned by the LP solver.

polynomial approximation produces a value in the rounding interval, the rounded result is the correct result. Consider the inputs:

$$x_1 = 1.95312686264514923095703125 \times 10^{-3}$$
$$x_2 = 2.14843768626451492309570 3125 \times 10^{-2}$$

We show the correctly rounded result of $sinpi(x)$ for these inputs with a black circle in Figure 2(a) and Figure 2(b), respectively. It also shows the rounding interval in gray.

Step 2: Identifying the reduced interval for input R. Range reduction transforms input x into R. To produce the result for $sinpi(x)$, we need to compute both $sinpi(R)$ and $cospi(R)$ (*i.e.*, multiple elementary functions). The result that we produce for $sinpi(R)$ and $cospi(R)$ should allow us to produce the correct result for $sinpi(x)$ (*i.e.*, produce a value within the rounding interval $[l, h]$ of input x).

To compute $sinpi(x)$, we will generate piecewise polynomials for $sinpi(R)$ and $cospi(R)$. Two inputs x_1 and x_2 (Figure 2(a) and 2(b)) map to the same reduced input after range reduction,

$$R = 1.86264514923095703125 \times 10^{-9}$$

Now, we need to deduce an interval $[ls', hs']$ for the output of $sinpi(R)$ and an interval $[lc', hc']$ for the output of $cospi(R)$ such that the result of output compensation produces a value within the rounding interval for x. We compute the correctly rounded value (v) of $sinpi(R)$ in double using the oracle and set it as our initial guess for $[ls', hs']$ (*i.e.*, $[v, v]$ a singleton). Similarly, we compute the interval $[lc', hc']$ for $cospi(R)$. Now, we need to check if these intervals are sufficient to produce the correct output for the original input x. Section 3.2 provides our detailed algorithm. The key idea is to simultaneously lower the lower bound for both $sinpi(R)$ and $cospi(R)$ and check if output compensation produces the correct result for all inputs. Similarly, we deduce the upper bound for both $sinpi(R)$ and $cospi(R)$. The reduced interval for $sinpi(R)$ from the perspective of x_1 is $[ls1', hs1']$. Similarly, the reduced interval for $sinpi(R)$ from the perspective of x_2 is $[ls2', hs2']$. These reduced intervals for $sinpi(R)$ corresponding to x_1 and x_2 are shown in Figure 2(c). They are not identical because our approach considers the numerical error in both range reduction and output compensation.

Step 3: Splitting the reduced domain into sub-domains. Now that we have reduced intervals for all reduced inputs, the next task is to generate polynomials for $sinpi(R)$ and $cospi(R)$. We illustrate this process with $sinpi(R)$. It is similar for $cospi(R)$. Even after range reduction, there are approximately 110 million unique reduced inputs for $R \in [0, \frac{1}{512}]$. Using our counterexample guided polynomial generation strategy (Step 4), we attempt to generate a polynomial for the entire reduced domain. If we cannot generate a polynomial or the polynomial does not satisfy the performance constraints, then we split the reduced input domain into smaller sub-domains to generate piecewise polynomials. We iteratively split the domain into smaller sub-domains until we can produce a polynomial that produces the correct results for all inputs and satisfies the performance criterion.

Let us say we want to generate 32 (*i.e.* 2^5) piecewise polynomials for the domain $[0, \frac{1}{512}]$. We use the bit-pattern of the reduced input R in double to identify the sub-domain. Although the domain of R is $[0, \frac{1}{512}]$, the value of R in our reduced inputs ranges between $[2^{-32}, 2^{-9}]$ along with $R = 0$. There is a large gap of values between the reduced input 0 and 2^{-32}. This is because we have already handled special cases for the original input. Excluding the reduced input $R = 0$, all other reduced inputs in the double representation have the left-most six bits identical. Hence, we use 5-bits

after the six left-most bits to identify the sub-domain for the piecewise polynomial. Figure 2(d) shows the reduced input R, its double bit-pattern, and the 5-bits used to identify the sub-domain.

Step 4: Generating a polynomial for a sub-domain. The final step is to produce a polynomial that approximates $sinpi(R)$ for a particular sub-domain. This polynomial must produce a value within the reduced interval $[ls', hs']$ for each reduced input R in the sub-domain. This requirement can be encoded as a linear constraint for each reduced input,

$$ls' \leq P(R) \leq hs'$$

where $P(R)$ is a polynomial that approximates $sinpi(R)$.

We show the generation of the polynomial for sub-domain with bit-pattern 10001. First, we sample a portion of the reduced inputs (*e.g.*, 2 in Figure 2(e)). Second, we encode the two reduced inputs and reduced intervals as linear constraints to create a LP query (see Figure 2(e)). Third, we use a LP solver to identify coefficients that satisfy the constraints. The generated polynomial is shown in Figure 2(f). Fourth, we check if the generated polynomial produces a value within the reduced interval for all inputs in the sub-domain. In this case, there are two reduced inputs where the generated polynomial does not produce a value within the reduced interval. Fifth, we add both counterexamples (*i.e.*, reduced inputs) to the sample. Next, we create a LP query using these four reduced inputs and intervals. Then, we check if the generated polynomial satisfies all reduced inputs in the sub-domain corresponding to bit-pattern 10001.

After generating polynomials for all 32 sub-domains, we store the coefficients of the piecewise polynomial in a table, which is indexed by the bit-pattern of the reduced input that identifies the sub-domain. The approximation for the elementary function $sinpi(x)$ is now ready. To produce the result for input x, our library will perform range reduction on x, identify the reduced input R, identify the sub-domain based on the bit-pattern of R, evaluate the piecewise polynomial using the coefficients from the table, perform output compensation, and round the result to a 32-bit float to produce the correctly rounded result.

3 Generating Piecewise Polynomials

Our goal is to generate polynomial approximations for elementary functions $f(x)$ that produce the correctly rounded result for all inputs x in 32-bit target representations $\mathbb{T}$. Similar to our prior work on RLibm [31, 32], we approximate the correctly rounded result rather than the real value of $f(x)$. We extend it in three main directions. First, we develop counterexample guided polynomial generation with sampling to make this approach feasible with 32-bit types. Second, we design techniques to generate piecewise polynomials, which provide performance improvements. Third, we develop modified range reduction techniques for a class of elementary functions and develop methods to deduce rounding intervals when range reduction involves multiple functions.

Correctly rounded result. For a given input x and elementary function f, the output of our approximation is the correctly rounded result if it is equal to the value of $f(x)$ computed with real numbers and then rounded to the target representation. We use $RN_{\mathbb{T}}(f(x))$ to denote the rounding function that rounds $f(x)$ computed with real numbers to target representation $\mathbb{T}$. All internal computation such as range reduction, polynomial evaluation, and output compensation is performed in representation $\mathbb{H}$ where $\mathbb{H}$ has higher precision than $\mathbb{T}$. To attain good performance, $\mathbb{H}$ is a representation that is supported in hardware (*e.g.*, double).

Our approach. There are three main tasks in creating polynomial approximations with our approach. First, we need to create a range reduction function, which we denote as $RR_{\mathbb{H}}(x)$, that reduces input x to a reduced input r in a smaller domain. Once we have the result of the elementary function for the reduced input r (let's say $y' = f(r)$), we need to develop an output compensation function, which we denote as $OC_{\mathbb{H}}(y', x)$, to produce the result of $f(x)$ for input x. Second, we need to generate polynomial approximations for each elementary function $f_i(r)$ in the reduced domain (*e.g.*, there were two elementary functions *sinpi* and *cospi* after range reduction in Section 2). We need to generate polynomials Ψ_i for each $f_i(r)$ in the reduced domain when there are millions of reduced inputs in each reduced input domain. We have to ensure that the polynomials generated for each $f_i(r)$ in the reduced domain produce correctly rounded results for all inputs after output compensation and polynomial evaluation is performed in $\mathbb{H}$. Third, we may have to split the reduced input domain to generate piecewise polynomials for each $f_i(r)$ to create efficient implementations.

High-level sketch. Algorithm 1 provides a high-level sketch of our approach. Given an elementary function $f(x)$ and a list of inputs X, we compute the correctly rounded result y in our target representation $\mathbb{T}$ (line 4) and compute the rounding interval of y in $\mathbb{H}$ (lines 14-17). If our approximation of $f(x)$ produces a value in the rounding interval, then the result will round to y. Next, we compute the reduced input r using range reduction. The range reduction may require us to compute multiple elementary functions f_i to produce the result for x. Hence, we identify the range of values that each function f_i should produce such that the result when used with output compensation produces a value in the rounding interval of y (line 7). We call this range of values for the reduced input r as the reduced interval (see Section 3.2). Finally, we approximate each elementary function $f_i(r)$ used in output compensation with piecewise polynomials of degree d (line 11) with counterexample guided polynomial generation and by using an LP solver. A single polynomial for each f_i may not be ideal for performance. To create efficient implementations, we iteratively split the domain of the reduced input into multiple sub-domains (see Section 3.3). Even such

```
1  Function CorrectPolys(f, X, RR_H, OC_H, d):
2      Y ← ∅
3      foreach x ∈ X do
4          y ← RN_T(f(x))
5          [l, h] ← RoundingInterval(y, T, H)
6          Y ← (x, [l, h])
7      L ← ReducedIntervals(Y, RR_H, OC_H)
8      Result ← ∅
9      foreach (f_i, L_i) ∈ L do
10         if L_i ← ∅ then return ∅
11         Ψ_i ← GenApproxFunc(L_i, d)
12         Result ← (f_i, Ψ_i) ∪ Result
13     return Result
14 Function RoundingInterval(y, T, H):
15     l ← min{v ∈ H | v ≤ y and RN_T(v) = y}
16     h ← max{v ∈ H | v ≥ y and RN_T(v) = y}
17     return [l, h]
```

Algorithm 1: `CorrectPolys` computes piecewise polynomials of degree d for each elementary function f_i used in output compensation, $OC_{\mathbb{H}}$, to generate a math library for elementary function f. It produces the correctly rounded result of $f(x)$ for each input $x \in X$. `RoundingInterval` computes the rounding interval $[l, h] \subset \mathbb{H}$ of y where all values in the interval rounds to y in $\mathbb{T}$. `ReducedIntervals` is shown in Algorithm 2 while `GenApproxFunc` is shown in Algorithm 3.

```
1  Function ReducedIntervals(Y, RR_H, OC_H):
2      if OC_H is not a monotonic function then return ∅
3      F ← {list of functions used in OC_H}
4      foreach f_i ∈ F do L_i ← ∅
5      foreach (x, [l, h]) ∈ Y do
6          r ← RR_H(x)
7          V ← {RN_H(f_i(r)) | f_i ∈ F}
8          if OC_H(V, x) ∉ [l, h] then return ∅
9          //Set initial reduced range for each f_i(r)
10         I' ← {[v, v] | v ∈ V}
11         //Decrease the lower bounds l'_i simulataneously
12         while true do
13             A ← {GetPrev(l'_i, H) | [l'_i, h'_i] ∈ I'}
14             if OC_H(A, x) ∉ [l, h] then break
15             I' ← {[GetPrev(I'_i, H), h'_i] | [l'_i, h'_i] ∈ I'}
16         //Increase the upper bounds h'_i simulataneously
17         while true do
18             B ← { GetNext(h'_i, H) | [l'_i, h'_i] ∈ I'}
19             if OC_H(B, x) ∉ [l, h] then break
20             I' ← {[I'_i, GetNext(h'_i, H)] | [l'_i, h'_i] ∈ I'}
21         foreach [l'_i, h'_i] ∈ I' do
22             L_i ← L_i ∪ (r, [l'_i, h'_i])
23     return {(f_i, L_i) | f_i ∈ F}
```

Algorithm 2: `ReducedIntervals` computes the reduced interval $[l'_i, h'_i]$ and the reduced input r corresponding to input x for each function f_i used with output compensation. If our polynomial approximation for f_i produces a value in $[l'_i, h'_i]$, then we can generate the correctly rounded result for x. `ReducedIntervals` returns a list with $(r, [l'_i, h'_i])$ for each f_i. `GetPrev`(p, $\mathbb{H}$) returns the preceding value of p in $\mathbb{H}$ and `GetNext`(p, $\mathbb{H}$) returns the succeeding value of p in $\mathbb{H}$.

sub-domains for the reduced inputs can have millions of reduced inputs. Hence, we create a sample of the reduced inputs, generate constraints to ensure that the polynomial of degree d produces a value in the reduced interval for the reduced inputs in the sample, and query the LP solver to solve for the coefficients. When the LP solver returns the coefficients, we check whether the generated polynomial produces a value within the reduced interval for all inputs in the sub-domain. We add any input that violates the constraints to the sample and repeat this process. We call this process as counterexample guided polynomial generation. At the end of this process, our approach produces piecewise polynomials for each $f_i(r)$, where the results of $f_i(r)$ when used with output compensation produces the correctly rounded result for all inputs when rounded to $\mathbb{T}$.

3.1 Computing Rounding Intervals

Our approach approximates the correctly rounded result rather than the real value. Hence, the first step is to identify the correctly rounded result using an oracle and then identify all values in $\mathbb{H}$ that rounds to the correct result in $\mathbb{T}$. As $\mathbb{H}$ has higher precision than $\mathbb{T}$, there is a range of values in $\mathbb{H}$ that our approach can produce and still round to the correctly rounded result in $\mathbb{T}$. We call this range the rounding interval. Algorithm 1 illustrates our steps to compute the rounding interval for each input $x \in X$ (lines 14-17). We compute the oracle correctly rounded result, $y = f(x)$, using the MPFR math library with a large number of precision bits. To compute the rounding interval, we identify the smallest value $l \in \mathbb{H}$ that rounds to y when rounded to $\mathbb{T}$ and the largest value $h \in \mathbb{H}$ that rounds to y when rounded to $\mathbb{T}$. This search procedure can be efficiently implemented either using a binary search or by leveraging the properties of $\mathbb{T}$ and $\mathbb{H}$. As long as our approach produces a value in the rounding interval $[l, h]$ for input x, it will produce the correctly rounded result.

3.2 Computing Reduced Rounding Intervals

Range reduction is crucial for any technique that generates approximations for elementary functions. It is particularly important with our approach for 32-bit types because the condition number of the LP problem increases drastically if the input domain has both extremely large and small values. Further, large inputs can cause overflows during the evaluation of a polynomial with a large degree in $\mathbb{H}$.

After computing rounding intervals from Algorithm 1, we have a list of constraints $(x, [l, h])$ that our approximation for $f(x)$ needs to satisfy for each input x to produce the correctly rounded result. The range reduction and subsequent output compensation can require us to approximate multiple elementary functions f_i. The next step is to identify reduced inputs to f_i and a range of values that f_i should produce such that the result of the output compensation produces a value in $[l, h]$ for each x. The input to f_i is the reduced input and the range of values that f_i should produce is the reduced interval.

Algorithm 2 shows the steps in deducing the reduced interval. For each constraint $(x, [l, h])$, we can identify the reduced input r by performing range reduction on x (line 6). However, computing the reduced interval is challenging. We present an algorithm to deduce reduced intervals when output compensation ($OC_{\mathbb{H}}$) is monotonic (either increasing or decreasing), which is the case with all range reductions that we explore in the paper.

To compute the reduced interval, we identify all functions f_i used in $OC_{\mathbb{H}}$ (line 4). Then, we compute the correctly rounded result v_i for each $f_i(r)$ in $\mathbb{H}$ using an oracle (line 7). If the result of output compensation using v_i's does not produce a value in the rounding interval for x, then either the range reduction technique should be redesigned or the precision of $\mathbb{H}$ should be increased.

Now, we have a candidate value (*i.e.*, v_i) for each $f_i(r)$ to produce the correctly rounded result of x. We have to deduce the maximum amount of freedom available for each $f_i(r)$. We initially set the reduced intervals $[l_i, h_i]$ for each f_i to be $[v_i, v_i]$ (line 10). Next, we identify if we can decrease the lower bound of the intervals of $f_i(r)$. For a given reduced input r of input x, we check if using the preceding values of l_i in $\mathbb{H}$ for all f_i's with output compensation produces a value in the rounding interval $[l, h]$ of x. If it does, then we widen the reduced interval by replacing each l_i with the preceding value. We repeat the process until the result of output compensation using the preceding values no longer produces a value in $[l, h]$ (lines 12-15). This procedure to compute the lower bound can be efficiently implemented by performing binary search between v_i and the minimum representable value.

Similarly, we identify if we can increase the upper bound of the interval for each $f_i(r)$. For each upper bound h_i of $f_i(r)$, we identify the value that succeeds h_i and check whether the result of output compensation using the succeeding value produces a value in $[l, h]$. If it does, then we widen the reduced interval by replacing each h_i with the succeeding value. We repeat the process until output compensation produces a value outside the interval $[l, h]$ of input x (lines 17-20). The upper bound of the reduced interval can be efficiently computed by performing binary search between v_i and the maximum representable value. Finally, we store the reduced constraints $(r, [l'_i, h'_i])$ for each function f_i in a list $\mathcal{L}_i$.

```
1  Function GenApproxFunc(ℒ, d):
2      ℒ⁻ ← {(r, [l′, h′]) ∈ ℒ | r < 0}
3      ℒ⁺ ← {(r, [l′, h′]) ∈ ℒ | r ≥ 0}
4      Ψ⁻ ← GenApproxHelper(ℒ⁻, d)
5      Ψ⁺ ← GenApproxHelper(ℒ⁺, d)
6      return {Ψ⁻, Ψ⁺}
7  Function GenApproxHelper(ℒ, d):
8      n ← 0
9      while true do
10         Δ = SplitDomain(ℒ, n)
11         (status, Ψ) = GenPiecewise(Δ, d)
12         if status = true then return Ψ
13         n ← n + 1
14 Function GenPiecewise(Δ, d):
15     Ψ ← ∅
16     foreach Δ_j ∈ Δ do
17         (status, Ψ_j) ← GenPolynomial(Δ_j, d)
18         if status = false then return (false, ∅)
19         Ψ ← Ψ ∪ Ψ_j
20     return (true, Ψ)
```

Algorithm 3: `GenApproxFunc` generates piecewise polynomials that produce a value in the reduced interval for all reduced inputs in $\mathcal{L}$. It initially attempts to produce a single polynomial for the entire reduced input domain. If unsuccessful, then it splits the domain into multiple sub-domains. `SplitDomain` (not defined in this algorithm) splits the reduced input domain into sub-domains based on the bit-pattern of the reduced inputs in $\mathbb{H}$. `SplitDomain` returns Δ, which includes a set of reduced constraints for each sub-domain Δ_j. `GenPiecewise` generates a polynomial for each sub-domain, which is shown in Algorithm 4.

Each $\mathcal{L}_i$ corresponding to f_i contains reduced intervals $(r, [l'_i, h'_i])$ for the reduced input r to produce a correct result for input x. As multiple inputs can map to the same reduced input r, there can be multiple reduced constraints $(r, [l1'_i, h1'_i])$ and $(r, [l2'_i, h2'_i])$ for the same reduced input r corresponding to original inputs $x1$ and $x2$. The reduced intervals $[l1'_i, h1'_i]$ and $[l2'_i, h2'_i]$ are not exactly identical to account for numerical errors in range reduction and output compensation. Our polynomial approximation for f_i must satisfy the constraints $(r, [l1'_i, h1'_i])$ to produce the correctly rounded result for $x1$ and $(r, [l1'_i, h1'_i])$ to produce the correctly rounded result for $x2$. Thus, we generate a single combined interval by computing the common interval between them. If there is no common interval between all reduced intervals corresponding to the same reduced input, then it implies that there is no polynomial approximation for f_i that produces the correctly rounded results for all inputs x in the original domain. The library designer will have to redesign range reduction in such cases.

3.3 Efficient Piecewise Polynomials

After the above steps, we have a list of reduced constraints $(r, [l'_i, h'_i])$ in $\mathcal{L}$ for each reduced input r and for each function f_i that we need to approximate. The next step in our approach is to generate polynomials that approximate f_i and satisfy the constraints in $\mathcal{L}_i$. Even after range reduction, there can be hundreds of millions of reduced inputs. The counterexample guided polynomial generation algorithm, which we describe in Section 3.4, can likely generate a single polynomial in many cases. However, it will also have a large degree and may not be efficient. To generate high performance math libraries, we propose the generation of piecewise polynomials. Effectively splitting the domain into smaller domains for the generation of piecewise polynomials is essential to improve performance. Hence, we group the reduced input into sub-domains based on the bit-patterns of the reduced input in $\mathbb{H}$.

Algorithm 3 describes our steps to generate piecewise polynomials. Range reduction techniques for many elementary functions can create both positive and negative reduced inputs (*e.g.*, e^x, 2^x, 10^x). The bit-patterns for positive and negative reduced inputs in $\mathbb{H}$ will not have common bits at the beginning (*e.g.*, the explicit sign bit distinguishes positive and negative values in double). Hence, we separate the reduced inputs (and their intervals) into two groups: $\mathcal{L}^-$ that contains negative reduced inputs and $\mathcal{L}^+$ that contains non-negative reduced inputs (lines 2-3). We create polynomial approximations for each $\mathcal{L}^-$ and $\mathcal{L}^+$ (lines 4-5). This step also allows us to subsequently group the reduced input into sub-domains in an efficient manner.

If $\mathcal{L}$ contains only negative or positive reduced inputs, we try to generate a single polynomial of degree d that satisfies all reduced constraints in $\mathcal{L}$ (line 11 and 17) using our counterexample guided polynomial generation (see Section 3.4). If it cannot generate a polynomial of degree d that satisfies all constraints, then we split the reduced input domain in $\mathcal{L}$ into multiple sub-domains (lines 9-13 in `GenApproxHelper`). We iteratively split the domain of reduced inputs into 2^n sub-domains based on the bit-pattern of r in $\mathbb{H}$ (*i.e.*, `SplitDomain` call in line 10). To split the reduced input domain, we first identify the smallest reduced input R_{min} and the largest reduced input R_{max}. Then, we compute the number of consecutive bits that are identical in the bit-string representation of R_{min} and R_{max} in $\mathbb{H}$ starting from the most significant bit. We use the next n bits to identify the sub-domain for the piecewise polynomial. Subsequently, we group the reduced inputs and reduced intervals based on the bit-pattern of the reduced input into sub-domains (Δ returned by `SplitDomain`). We try to generate a polynomial of degree d that satisfies all reduced constraints in Δ_j for all Δ_j's belonging to f_i (lines 16-19). Using bit-patterns of the reduced input in $\mathbb{H}$ allows us to efficiently identify the sub-domain for the piecewise polynomial with two bitwise operations (`and` and a `shift`).

```
1  Function GenPolynomial(Δ_j, d):
2      S ← Sample(Δ_j)
3      while true do
4          Ψ_j ← GetCoeffsUsingLP(S, d)
5          if Ψ_j = ∅ then return (false, ∅)
6          (Done, S) ← Check(Ψ_j, Δ_j, S)
7          if Done = true then return (true, Ψ_j)
8          if |S| > threshold then return (false, ∅)
9  Function Check(Ψ_j, Δ_j, S):
10     Done ← true
11     foreach (r, [l', h']) ∈ Δ_j do
12         if not l' ≤ Ψ_j(r) ≤ h' then
13             S ← {(r, [l', h'])} ∪ S
14             Done ← false
15     return (Done, S)
```

Algorithm 4: `GenPolynomial` attempts to find a polynomial of degree d that satisfies the reduced input and interval constraints in Δ_j using our counterexample guided sampling approach. If it is infeasible to find a polynomial of degree d or the size of the sample exceeds a threshold, then it returns (false, $\emptyset$). `GetCoeffsUsingLP` generates the coefficients of a polynomial that satisfies all constraints in $\mathcal{S}$ using the LP solver. `Check` validates that the polynomial generated using the sample satisfies all reduced input and interval constraints. We add counterexamples (*i.e.*, all inputs where the polynomial does not satisfy the constraints) to the sample and repeat the process.

Once we generate a polynomial for each sub-domain of every f_i, the coefficients of the polynomial are stored in a table, which is indexed using the bit-pattern of the reduced input for each f_i.

3.4 Counterexample Driven Polynomial Generation

Once we have the reduced input and the reduced intervals, we structure the problem of generating polynomials as a linear programming problem similar to our prior work on RLibm [31, 32]. Even after range reduction and creation of sub-domains for the generation of piecewise polynomials, we need to generate a polynomial approximation when there are several million reduced inputs and reduced intervals in the context of 32-bit types. However, they are beyond the capabilities of modern LP solvers, which can handle a few thousand constraints. To address this issue, we propose counterexample guided polynomial generation with sampling. The key insight is that we do not need to add every reduced input and interval as a constraint in the LP formulation as long as we identify and add the highly constrained intervals.

Our counterexample guided polynomial generation strategy takes as input the set of reduced constraints $(r, [l', h'])$ corresponding to reduced inputs that belong to a particular sub-domain. The goal is to generate a polynomial of degree d that produces a value in the reduced interval $[l', h']$ for each

reduced input r. Each reduced input r and the corresponding interval $[l', h']$ specifies the following linear constraint for a polynomial of degree d that we want to generate:

$$l' \leq c_0 + c_1 r + c_2 r^2 + \cdots + c_d r^d \leq h'$$

The task of the polynomial generator is to find coefficients for the polynomial.

To scale to 32-bit types, we sample a small fraction of the reduced input and intervals. Algorithm 4 reports our counterexample guided polynomial generation process. It takes two inputs: the degree of the polynomial and the set of reduced inputs and intervals (*i.e.*, Δ_j) for generating a polynomial approximation for an elementary function f_i on reduced inputs for sub-domain j. We maintain the reduced inputs and their intervals in increasing order. Then we uniformly sample the reduced inputs based on the distribution of reduced inputs. If there are a large number of reduced inputs in a particular region of the sub-domain, then our method has more samples from that region. We also add highly constrained reduced inputs and intervals (*i.e.*, the correctly rounded result and the lower bound/upper bound is less than ϵ, which is set by the math library designer) to the sample.

Then we express all constraints in the sample (r, $[l', h']$) using a single system of linear inequalities and solve for the coefficients using an LP solver (line 4). If there are n points in the sample, the system of linear inequalities is of the following form:

$$\begin{bmatrix} l'_1 \\ l'_2 \\ \vdots \\ l'_n \end{bmatrix} \leq \begin{bmatrix} 1 & r_1 & \dots & r_1^d \\ 1 & r_2 & \dots & r_2^d \\ \vdots & \vdots & \vdots & \vdots \\ 1 & r_n & \dots & r_n^d \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_d \end{bmatrix} \leq \begin{bmatrix} h'_1 \\ h'_2 \\ \vdots \\ h'_n \end{bmatrix}$$

There are two issues with the polynomial generated using the sampled reduced inputs that we need to address. First, as the LP solver returns coefficients as real numbers, the coefficients are rounded to a value in $\mathbb{H}$. As a result of rounding error, the result of polynomial evaluation for a particular reduced input in the sample may not lie within its rounding interval. Second, the polynomial generated using the sample may not satisfy the constraints for the entire set of reduced inputs and their corresponding intervals.

We address the real coefficients issue with a search-and-refine procedure similar to RLibm. When the LP solver returns real coefficients and we round it to $\mathbb{H}$, we check whether evaluating the polynomial satisfies constraints for every input in the sample. If it does not, then we select the input and reduce its rounding interval (either replace the lower bound with its succeeding value or replace the upper bound with the preceding value). Then we repeat the above process until it generates a polynomial that either satisfies all constraints in the sample when evaluated in $\mathbb{H}$ or cannot find a polynomial of degree d. If we cannot find a polynomial that satisfies all constraints in the sample, then we split the entire reduced domain in $\mathcal{L}_i$ into even smaller sub-domains and repeat this process.

If we successfully generate a polynomial Ψ_j that satisfies all constraints in the sample, then we check whether this polynomial satisfies all constraints in Δ_j (line 10-15). If Ψ_j satisfies all constraints, then we return the polynomial (line 7). If there is any constraint not satisfied by Ψ_j in the entire set of reduced inputs, then we add that reduced input and its interval to the sample (*i.e.*, adding the counterexample in lines 12-13). We repeat the process of generating the polynomial with the new sample. If the number of constraints in the sample exceeds a threshold at any point, then we determine that we cannot generate a polynomial for the sub-domain Δ_j. Our function to generate the coefficients for the polynomial (*i.e.*, `GetCoeffsUsingLP`) using an LP solver generates a polynomial of a lower degree (than input degree d) if it is possible to do so.

4 Experimental Evaluation

We provide details on our prototype, experimental methodology, and the results of our experiments to check the correctness and performance of the generated functions.

4.1 Experimental Setup and Methodology

Prototype. The RLibm-32 prototype generates correctly rounded elementary functions for 32-bit `floats` and `posit32`, which is a 32-bit posit type providing tapered precision (*i.e.*, more precision than float for values near 1) [19]. It contains ten correctly rounded elementary functions for 32-bit floats and eight elementary functions for the posit32 type. To generate correctly rounded elementary functions with good performance, the user can provide custom range reduction functions and specify the degree or the structure of the polynomial (*i.e.*, odd or even). RLibm-32 uses the MPFR library [15] with up to 400 precision bits to compute the oracle for $f(x)$ and rounds it to the target representation, which is good enough to compute the oracle result for double [28]. RLibm-32 uses SoPlex [17], an exact rational LP solver, for generating coefficients for the polynomials with a five minute time limit. We use a threshold of fifty thousand reduced inputs and intervals in the sample for counterexample guided polynomial generation. RLibm-32's math library performs range reduction, polynomial evaluation, and output compensation using double precision. Polynomial evaluation uses the Horner's method [3]. We designed novel extensions to range reduction for many elementary functions, which is inspired by table-based range reduction [13, 44–46]. Our extended technical report provides additional details about range reduction for each elementary function [34]. RLibm-32 is open source and publicly available [33].

Table 1. Generation of correctly rounded results for 32-bit floats with RLibm-32, Intel's libm (float and double), glibc's libm (float and double), CR-LIBM, and MetaLibm (float and double). ✓indicates that the library produces the correctly rounded result for all inputs. Otherwise, we use ✗. For each ✗, we show the number of inputs with wrong results. N/A indicates that the implementation is not available.

float functions	Using RLibm-32	Using glibc float	Using glibc double	Using Intel float	Using Intel double	Using CR-LIBM	Using MetaLibm float	Using MetaLibm double
ln(x)	✓	✗(4.2E5)	✗(5)	✗(1060)	✗(5)	✗(5)	N/A	N/A
log2(x)	✓	✗(3.1E5)	✓	✗(276)	✓	✓	N/A	N/A
log10(x)	✓	✗(3.0E7)	✗(1)	✗(1.5E5)	✗(1)	✗(1)	N/A	N/A
exp(x)	✓	✗(1.7E5)	✓	✗(2.5E5)	✓	✓	✗(5.1E8)	✗(5.1E8)
exp2(x)	✓	✗(1.7E5)	✗(2)	✗(7.2E5)	✗(2)	N/A	✗(6.5E7)	✗(1026)
exp10(x)	✓	✗(1.7E5)	✓	✗(3.9E5)	✓	N/A	N/A	N/A
sinh(x)	✓	✗(7.1E7)	✗(2)	✗(2.5E5)	✗(2)	✗(2)	N/A	N/A
cosh(x)	✓	✗(1.8E7)	✓	✗(1.4E5)	✓	✓	✗(1.1E7)	✓
sinpi(x)	✓	N/A	N/A	✗(3.4E5)	✓	✓	N/A	N/A
cospi(x)	✓	N/A	N/A	✗(3.8E5)	✓	✓	N/A	N/A

Methodology. We test the elementary functions in RLibm-32 on two dimensions: (1) ability to generate correct results and (2) performance in comparison to state-of-the-art libraries. We compare RLibm-32's functions with four libraries: Intel's libm, glibc's libm, CR-LIBM [13], and Metalibm [25]. To use double precision libraries, we convert the float input into double, use the double function, and round the result back to float. Among these libraries, CR-LIBM has correctly rounded functions for double precision. However, CR-LIBM does not produce correctly rounded results for 32-bit floats due to double rounding. There are no math libraries available for posit32. All posit32 values can be exactly represented in double. Hence, we compare our posit32 library with glibc and Intel's double libm and CR-LIBM.

Experimental setup. We performed all our experiments on a 2.10GHz Intel Xeon Gold 6230R machine with 187GB of RAM running Ubuntu 18.04. We disabled Intel turbo boost and hyper-threading to minimize noise. We compiled RLibm-32's math library at the `O3` optimization level. We used Intel's libm from the oneAPI Toolkit and glibc's libm from glibc-2.33. We generated Metalibm implementations with optimizations for `AVX2` extensions enabled. Our test harness that compares glibc's libm, CR-LIBM, and Metalibm with RLibm-32 is built using the `gcc-10` compiler with `-O3 -static -frounding-math -fsignaling-nans` flags. To use Intel's libm, we have to use the Intel compiler. Hence, the test harness that compares Intel libm with RLibm-32 is built using the `icc` compiler with `-O3 -no-ftz -fp-model strict -static` to obtain as many correct results as possible. Further, the size of the executable generated by statically linking RLibm-32 is 2% smaller on average when compared to the executable generated with Intel's double libm.

Measuring performance. To compare performance, we measure the number of cycles taken to compute the result for each input using hardware performance counters. The total time taken is computed as the sum of the time taken by all inputs (*i.e.*, all 2^{32} inputs for a 32-bit representation). We ran the measurements for all inputs for each function six times. Then, we compute the average time taken to compute each elementary function. As Intel's compiler performs vectorization by default at the `O3` optimization level, our above setup does not measure improvements due to vectorization. Hence, we created another test harness that creates an array of 1024 floats (*i.e.*, 2^{10} inputs), populates it with different inputs, and measures the number of cycles taken to compute the results of 2^{10} inputs using hardware performance counters. We repeat this experiment 2^{22} times to compute the result and measure the total time taken for all 2^{32} inputs.

4.2 Generation of Correctly Rounded Results

Table 1 reports the results of our experiments to check the correctness of various elementary functions in RLibm-32 and other mainstream libraries. RLibm-32 produces the correctly rounded results for all inputs for the ten elementary functions for 32-bit floats. In contrast, elementary functions in glibc, Intel, and MetaLibm's float library do not produce the correct result for all inputs. Multiple functions in glibc and MetaLibm's float library produce wrong results for several million inputs. Intel's libm also produces wrong results with several thousand inputs with the float version. When we use double precision version of functions from glibc, Intel's libm, and CR-LIBM, it does not produce the correct result for $ln(x)$, $log10(x)$, $exp2(x)$, and $sinh(x)$. These cases occur when the real value of $f(x)$ is extremely close to the rounding boundary of a floating point value. Even with a smaller mini-max approximation error in the double library compared to their float versions, these libraries do not produce the correctly rounded result for all inputs. CR-LIBM, which is a correctly rounded double library, produces wrong results for float functions due to double rounding. We observed that functions in MetaLibm do not produce correct results even when it internally uses Sollya [9], which can be used to generate correctly rounded implementations.

Table 2. Generation of correctly rounded results with posit32 functions for all inputs by RLIBM-32, Intel and glibc's double libraries, and CR-LIBM. ✓indicates that the library produces the correctly rounded result for all inputs and otherwise, we use ✗.

posit32 functions	Using RLIBM-32	Using glibc double	Using Intel double	Using CR-LIBM
ln(x)	✓	✗(22)	✗(22)	✗(22)
log2(x)	✓	✗(19)	✗(18)	✗(18)
log10(x)	✓	✗(26)	✗(23)	✗(23)
Exp(x)	✓	✗(4.4E8)	✗(4.4E8)	✗(4.4E8)
Exp2(x)	✓	✗(4.0E8)	✗(4.0E8)	N/A
Exp10(x)	✓	✗(5.2E8)	✗(5.2E8)	N/A
Sinh(x)	✓	✗(4.4E8)	✗(4.4E8)	✗(4.4E8)
Cosh(x)	✓	✗(4.4E8)	✗(4.4E8)	✗(4.4E8)

Table 2 reports that RLIBM-32 produces correctly rounded results with all inputs for the eight posit32 functions. All posit32 values are representable in double precision but they cannot be represented in 32-bit floats. Hence, we use CR-LIBM, Intel and glibc's double library to compare with RLIBM-32. These libraries for double precision do not produce correct results for all posit32 inputs. Unlike functions for 32-bit floats, they produce wrong results for several million inputs especially for exponential and hyperbolic functions. One of the key reasons for wrong results is the absence of overflows to ∞ and underflows to 0 with the posit32 type. Instead, extremely large values are rounded to the largest representable value. Similarly, extremely small values are rounded to the smallest non-zero representable value in the posit32 type.

Piecewise polynomials generated by RLIBM-32. Table 3 provides details on the piecewise polynomials generated by RLIBM-32. Our goal is to get the best possible performance within a given storage budget for piecewise polynomials (*i.e.*, number of sub-domains when we split the range of reduced inputs). Hence, we used the RLIBM-32 to generate piecewise polynomials such that the degree of each polynomial was less than or equal to 8 and the number of sub-domains was less than or equal to 2^{14}. The output compensation for $sinh(x)$, $cosh(x)$, $sinpi(x)$, and $cospi(x)$ involves two elementary functions. We generate two piecewise polynomials for each of those elementary functions. There are both positive and negative reduced inputs for $exp(x)$, $exp2(x)$, and $exp10(x)$. Hence, we created two piecewise polynomials: one for the negative reduced inputs and another for positive reduced inputs. Notably, we were able to generate a single polynomial of degree 5 and 4 that satisfies all reduced constraints for $sinpi(r)$ and $cospi(r)$, respectively. Both $sinpi(r)$ and $cospi(r)$ have close to 120 million reduced inputs. Our counterexample driven polynomial generation with sampling was instrumental in creating this efficient polynomial.

Time taken to generate RLIBM-32 functions. Table 3 also reports the time taken to generate the 32-bit float and the posit32 functions in RLIBM-32. It ranges from 19 minutes for $cospi(x)$ for the float type to approximately 25 hours for $exp10(x)$ for the posit32 type. Majority of the total time total time is spent in computing the oracle result and the rounding interval using the MPFR library (*i.e.*, 86% of total time for 32-bit floats and 55% of total time for the posit32 type). In contrast, counterexample guided polynomial generation takes 14% and 45% of the total time for 32-bit floats and the posit32 type, respectively. We noticed that it takes significantly longer to generate posit32 functions. There are fewer special cases, which requires longer oracle computation. Further, RLIBM-32 generates larger piecewise polynomials for posit32 functions to account for higher precision than a 32-bit float and saturating behavior with extremal values.

Table 3. Details about the generated polynomials. For each elementary function, time taken to generate the polynomials in minutes, the size of the piecewise polynomial for approximating $f_i(r)$, the maximum degree of the polynomial, and the number of terms in the polynomial.

$f(x)$	Gen. Time (Minutes)	Reduced Inputs	# of Polynomials	Degree	# of Terms
float functions					
$ln(x)$	218	7.2E6	2^{10}	3	3
$log2(x)$	251	7.2E6	2^{8}	3	3
$log10(x)$	429	7.2E6	2^{8}	3	3
$exp(x)$	117	5.2E8	2^{7} 2^{7}	4 4	5 5
$exp2(x)$	86	3.0E8	2^{4} 2^{3}	4 4	5 5
$exp10(x)$	169	5.2E8	2^{6} 2^{7}	4 3	5 4
$sinh(x)$	28	1.5E8	2^{6}	5	3
$cosh(x)$	24	1.5E8	2^{6}	4	3
$sinpi(x)$	30	1.2E8	1	5	3
$cospi(x)$	19	1.2E8	1	4	3
posit32 functions					
$ln(x)$	264	1.1E8	2^{11}	4	4
$log2(x)$	288	1.1E8	2^{8}	4	4
$log10(x)$	685	1.1E8	2^{12}	3	3
$exp(x)$	1089	3.5E9	2^{12} 2^{12}	3 3	4 4
$exp2(x)$	814	7.9E8	2^{10} 2^{12}	3 3	4 4
$exp10(x)$	1528	3.4E9	2^{13} 2^{13}	3 3	4 4
$sinh(x)$	461	1.6E9	2^{14} 2^{14}	5 4	3 3
$cosh(x)$	528	1.7E9	2^{14} 2^{12}	3 6	2 4

4.3 Performance Evaluation of RLIBM-32

Performance of float functions. Figure 3(a) presents the speedup of RLIBM-32's float functions over glibc's float functions (left bar in each cluster) and double functions (right

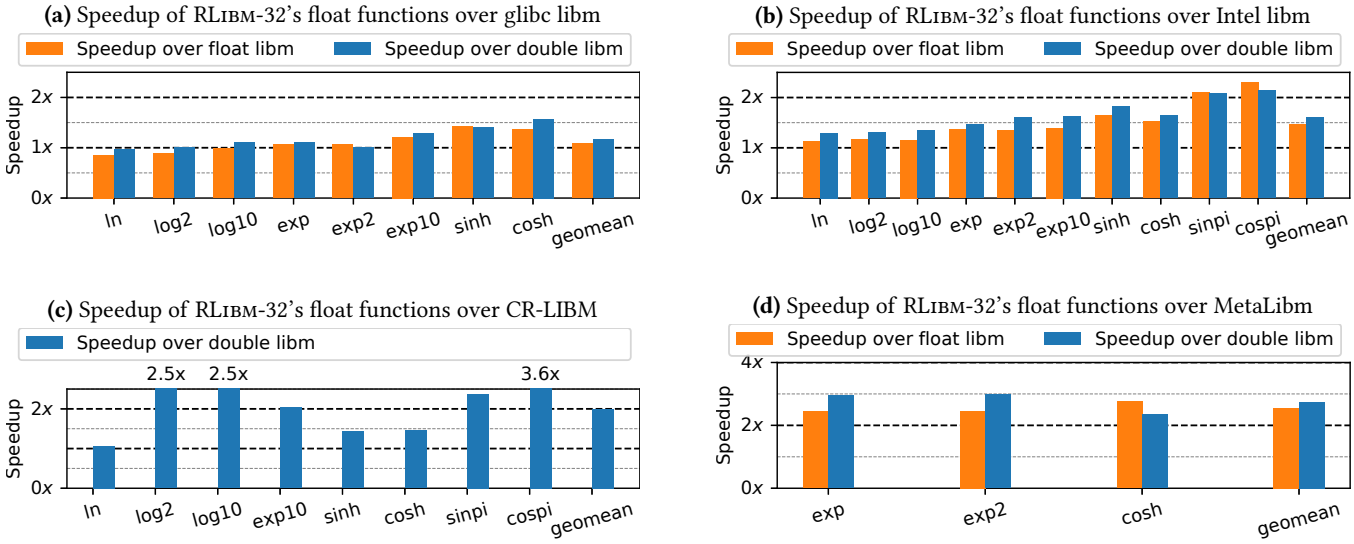

Figure 3. (a) Speedup of RLIBM-32's float functions compared to glibc's float functions (left) and glibc's double functions (right). (b) Speedup of RLIBM-32's functions compared to Intel's float functions (left) and Intel's double functions (right). (c) Speedup of RLIBM-32's functions compared to CR-LIBM functions. (d) Speedup of RLIBM-32's functions compared to MetaLibm's float functions (left) and double functions (right) built with AVX2 optimizations.

bar in each cluster). On average, RLIBM-32's float functions have 1.1× speedup over glibc's float libm and 1.2× speedup over glibc's double libm. Figure 3(b) reports the speedup of RLIBM-32's float functions over Intel's float libm and double libm. RLIBM-32's float functions have an average of 1.5× speedup over Intel's float functions and 1.6× speedup over Intel's double functions. Figure 3(c) reports that RLIBM-32's functions are on average 2× faster than CR-LIBM functions. Figure 3(d) reports the speedup of RLIBM-32's functions over MetaLibm's float and double functions. RLIBM-32's functions are on average 2.5× and 2.7× faster than MetaLibm's float and double functions, respectively. RLIBM-32's functions are faster than all the corresponding functions in Intel libm, CR-LIBM, and MetaLibm. RLIBM-32's functions are faster than glibc's functions except for $ln(x)$, $log_2(x)$, and $log_{10}(x)$ for float and $ln(x)$ for double. However, glibc's libm produces a large number of wrong results for them. RLIBM-32's functions are not only faster but also produce correctly rounded results for all inputs.

Performance of posit32 functions. The graphs in Figure 4(a), Figure 4(b), and Figure 4(c) report the speedup of RLIBM-32's posit32 functions when compared to math libraries created by re-purposing glibc's, Intel's, and CR-LIBM's double functions, respectively. On average, RLIBM-32's posit32 functions are 1.1×, 1.1×, and 1.4× faster than glibc's libm, Intel's libm, and CR-LIBM, respectively. All three re-purposed math libraries produce wrong results for some inputs. RLIBM-32 provides the first correctly rounded functions for the posit32 type.

Vectorization. Intel compiler uses vector instructions to improve performance by default. In our experiments with vectorization using an array of 1024 inputs (see Section 4.1), RLIBM-32 is on average 10% and 5% slower than Intel's float libm and double libm, respectively. However, Intel's compiler produces wrong results for several million inputs (without `-no-ftz -fp-model strict` flags). In contrast, RLIBM-32's functions are almost as fast as vectorized code while producing correct results for all inputs.

Performance impact of piecewise polynomials. To analyze the performance benefits due to piecewise polynomials, we identified elementary functions for which we could generate a single polynomial that produces correctly rounded results for all inputs ($log2(x)$, $log10(x)$, *sinpi*, and *cospi*) . We measured the change in performance with an increase in the number of sub-domains ranging from 2^0 (*i.e.*, a single polynomial) to 2^{12}. Figure 5 reports the performance of $log2(x)$ and $log_{10}(x)$ with an increase in the number of sub-domains when compared to the performance of a single polynomial. We validated that all these polynomials produce the correct result for all inputs. Figure 5 does not report *sinpi* and *cospi* because the single polynomial has the best performance. Initially, there is a small decrease in performance by moving from a single polynomial to a piecewise polynomial because the degree of the piecewise polynomial does not decrease significantly to subsume the overhead of table lookup. On increasing the number of sub-domains, we observed almost 1.2× speedup with piecewise polynomials having 2^8 sub-domains. It requires 6KB for storing coefficients of piecewise polynomials.

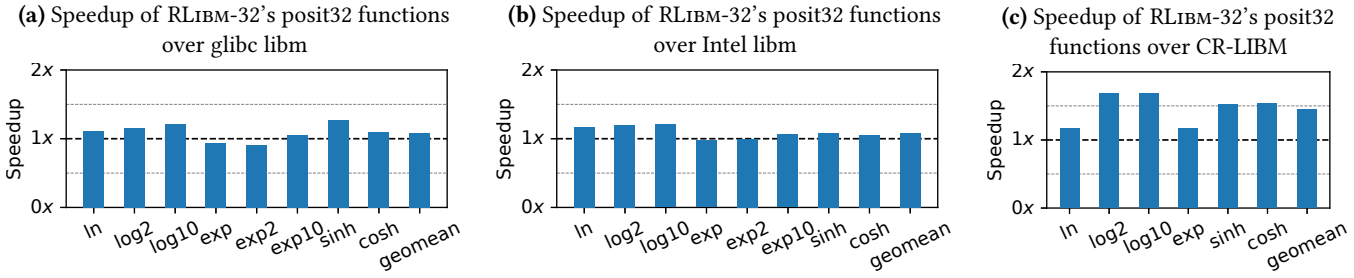

Figure 4. (a) Speedup of RLIBM-32's posit32 functions compared to glibc's double functions. (b) Speedup of RLIBM-32's posit32 functions compared to Intel's double functions. (c) Speedup of RLIBM-32's posit32 functions compared to CR-LIBM functions.

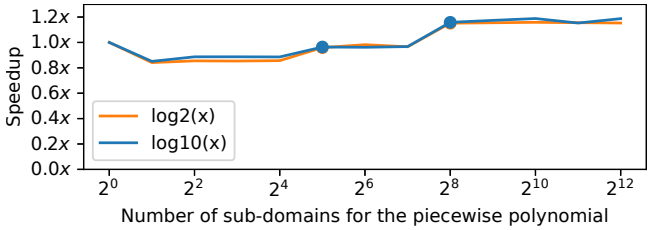

Figure 5. Performance speedup of $log2(x)$ and $log10(x)$ with an increase in the number of sub-domains when compared to a single polynomial generated by RLIBM-32. All these polynomials produce the correctly rounded result for all inputs. A circle represents a decrease in the degree of the piecewise polynomial.

5 Case Study with $cospi(x)$ for Float

We describe the case study with $cospi(x) = cos(\pi x)$ to illustrate the importance of carefully designing range reduction to avoid cancellation errors in output compensation. The elementary function $cospi(x)$ is defined for $x \in (-\infty, \infty)$.

Special cases. There are three kinds of special cases:

$$cospi(x) = \begin{cases} 1.0 & \text{if } |x| < 7.771 \times 10^{-5} \\ (-1)^{(|x| \bmod 2)} \times 1.0 & \text{if } |x| \geq 2^{23} \\ NaN & \text{if } x = NaN \text{ or } x = \pm\infty \end{cases}$$

All float values $\geq 2^{23}$ are integers. Hence, $cospi(x) = 1.0$ for even integers and $cospi(x) = -1.0$ for odd integer inputs.

Range reduction of $cospi(x)$. After excluding special cases, there are more than 600 million float inputs that need to be approximated. Similar to range reduction for $sinpi(x)$ (Section 2.1), we use periodicity and trigonometric identities of $cospi(x)$ to reduce inputs to a smaller domain. We transform input x into $x = 2.0 \times I + J$ where I is an integer and $J \in [0, 2)$. Due to periodicity, $cospi(x) = cospi(J)$. Next, we decompose J into $J = K + L$ where K is the integral part of J ($K \in \{0, 1\}$) and $L \in [0, 1)$ is the fractional part. Then, $cospi(J)$ can be computed with,

$$cospi(J) = (-1)^K cospi(L)$$

To further reduce the range of L, we use the fact that $cospi(x)$ between $[0.5, 1)$ is a mirror image of $cospi(x)$ between $[0, 0.5)$ **with the opposite sign**. We decompose L into M and L' where

$$M = \begin{cases} 0 & \text{if } \leq 0.5 \\ 1 & \text{if } L > 0.5 \end{cases} \qquad L' = \begin{cases} L & \text{if } L \leq 0.5 \\ 1.0 - L & \text{if } L > 0.5 \end{cases}$$

We have $cospi(L) = (-1)^M cospi(L')$. After reducing the input x to $L' \in [0, 0.5]$, there are around 107 million inputs. Thus, we further reduce L' to a value in $[0, \frac{1}{512}]$. We split $L' = \frac{N}{512} + Q$ where N is an integer in the set $\{0, 1, 2, \ldots, 255\}$ and Q is a fractional value in $[0, \frac{1}{512}]$. One possible method to compute $cospi(L')$ is to use the trigonometric identity $cospi(a + b) = cospi(a)cospi(b) - sinpi(a)sinpi(b)$,

$$cospi\left(\frac{N}{512} + Q\right) = cospi\left(\frac{N}{512}\right) cospi(Q) - sinpi\left(\frac{N}{512}\right) sinpi(Q)$$

The above formula is not monotonic and can have cancellation errors if $N \neq 0$ (if $N = 0$, then $cospi(L') = cospi(Q)$).

Creating monotonic output compensation. If $N \neq 0$, we transform N and Q to N' and R such that $L' = \frac{N'}{512} - R$ to create a monotonic output compensation function:

$$N' = \begin{cases} 0 & \text{if } N = 0 \\ N + 1 & otherwise \end{cases} \qquad R = \begin{cases} Q & \text{if } N = 0 \\ \frac{1}{512} - Q & otherwise \end{cases}$$

Then, we can compute $cospi(L') = cospi(\frac{N'}{512} - R)$ using the trigonometric identity $cospi(a - b) = cospi(a)cospi(b) + sinpi(a)sinpi(b)$ as follows,

$$cospi(L') = \begin{cases} cospi(R) & \text{if } N = 0 \\ cospi\left(\frac{N'}{512}\right) cospi(R) + sinpi\left(\frac{N'}{512}\right) sinpi(R) & \text{if } N \neq 0 \end{cases}$$

This output compensation is monotonic and does not experience cancellation error. The values of N' ranges from 0 to 256 and $R \in [0, \frac{1}{512}]$. The computation $\frac{1}{512} - Q$ can be computed exactly with float or double type for all values of Q

that corresponds to $N \neq 0$. There are approximately 40 million values of R. We precompute the values for $cospi\left(\frac{N'}{512}\right)$ and $sinpi\left(\frac{N'}{512}\right)$ in lookup tables (*i.e.*, 514 values in total). We create polynomial approximations for $sinpi(R)$ and $cospi(R)$ for the reduced input domain $R \in [0, \frac{1}{512}]$. Using RLibm-32, we were able to generate a single 5^{th} degree odd polynomial for $sinpi(R)$ and a single 4^{th} degree even polynomial for $cospi(R)$. Finally, we can compute the result for $cospi(x)$ with the output compensation function,

$$cospi(x) = \begin{cases} S \times cospi(R) & \text{if } N = 0 \\ S \times (cpn \times cospi(R) + spn \times sinpi(R)) & \text{if } N \neq 0 \end{cases}$$

where $S = (-1)^K \times (-1)^M$, $cpn = cospi(\frac{N'}{512})$, and $spn = sinpi(\frac{N'}{512})$. These polynomials combined with the output compensation functions produce correctly rounded results for all inputs for $sinpi(x)$ and $cospi(x)$.

6 Related Work

Multiple decades of seminal work has advanced the state-of-the-art on creating approximations for FP representations [7, 14, 15, 24, 37, 39, 47, 49]. Further, seminal research on range reduction has made such approximation feasible [2, 12, 43–46]. Simultaneously, there are verification efforts to prove bounds for math libraries [21–23, 27, 41], identify numerical errors with expressions that can be used in the implementation of math libraries [1, 11, 16, 18, 40], and repair individual outputs of math libraries [38, 48, 50].

Correctly rounded libraries. Numerous groups have developed correctly rounded elementary functions [7, 24]. Some correctly rounded libraries for FP are IBM LibUltim [49], Sun Microsystem's LibMCR, CR-LIBM [13], MPFR math library [15], and RLIBM [31, 32]. CR-LIBM is a correctly rounded double library developed using Sollya [9], which generates mini-max polynomials to approximate elementary functions [4, 5]. Sollya uses the modified Remez algorithm [39] using lattice basis reduction and also computes the error bound of the polynomial [8, 10, 36]. Metalibm [6, 25] builds on Sollya and generates efficient mini-max polynomials with user-defined error bounds. It also uses domain splitting and hardware specific optimizations [26]. Compared to mini-max approaches, our work approximates the correctly rounded result of $f(x)$ and generates polynomials that already account for numerical error in range reduction and output compensation. Hence, it generates efficient and correctly rounded results for all inputs.

This paper extends our prior work on RLibm [31, 32] and John Gustafson's Minefield method [20], which advocate approximating the correctly rounded value rather than real value of an elementary function. Our prior work on RLibm also frames the problem of generating polynomials as an LP problem. We have used RLibm to create correctly rounded functions for 16-bit types: bfloat16 and posit16. This paper extends RLibm to handle 32-bit types with systematic counterexample guided polynomial generation, generation of piecewise polynomials to improve performance, and new techniques to deduce rounding intervals when range reduction involves multiple elementary functions.

Posit libraries. SoftPosit-Math [30] and RLibm libraries provide correctly rounded math functions for 16-bit posits. In our prior work, we have produced approximations for a set of trigonometric functions using the CORDIC method for posit32 [35]. However, it does not produce correct results for all inputs. In this paper, we develop the first set of elementary functions that produce correctly rounded results for all inputs for 32-bit posits.

7 Conclusion and Future Directions

Mainstream math libraries have been designed and improved by numerous researchers spanning multiple decades. Yet, they fail to generate correct results for all inputs. This paper advocates approximating the correctly rounded value instead of the real value similar to our prior work on RLibm. It extends RLibm to scale to 32-bit representations: (a) counterexample guided polynomial generation with an LP solver to handle billions of inputs, (b) generation of constraints to account for multiple elementary functions in range reduction, and (c) generation of piecewise polynomials. The resulting functions produce correct results for all inputs and are also faster than existing libraries for 32-bit floats and posits.

Going forward, we plan to generate approximations for all commonly used elementary functions with 32-bit types, which we believe can be accomplished with our approach. However, it may require us to develop novel extensions to range reduction. Further, it may be necessary to perform range reduction in higher precision for some trigonometric functions such as sine and cosine that use π. Beyond 32-bit types, we also plan to extend this approach to double precision. Our approach can generate a polynomial that produces the correctly rounded result for the sampled points in the double type. Validating the correctness of the result produced by a polynomial generated using our approach for all inputs in the double type is an open research problem. Our long-term goal is to enable the standards of existing and new representations to mandate correctly rounded results.

Acknowledgments

We thank our shepherd Rahul Sharma and the PLDI reviewers for their feedback. We thank John Gustafson for his inputs on the Minefield method and the posit representation. This material is based upon work supported in part by the National Science Foundation under Grant No. 1908798 and Grant No. 1917897.

References

[1] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A Dynamic Program Analysis to Find Floating-point Accuracy Problems. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. ACM, New York, NY, USA, 453–462. https://doi.org/10.1145/2345156.2254118

[2] Sylvie Boldo, Marc Daumas, and Ren-Cang Li. 2009. Formally Verified Argument Reduction with a Fused Multiply-Add. In *IEEE Transactions on Computers*, Vol. 58. 1139–1145. https://doi.org/10.1109/TC.2008.216

[3] Peter Borwein and Tamas Erdelyi. 1995. *Polynomials and Polynomial Inequalities*. Springer New York. https://doi.org/10.1007/978-1-4612-0793-1

[4] Nicolas Brisebarre and Sylvvain Chevillard. 2007. Efficient polynomial L-approximations. In *18th IEEE Symposium on Computer Arithmetic (ARITH '07)*. https://doi.org/10.1109/ARITH.2007.17

[5] Nicolas Brisebarre, Jean-Michel Muller, and Arnaud Tisserand. 2006. Computing Machine-Efficient Polynomial Approximations. In *ACM ACM Transactions on Mathematical Software*, Vol. 32. Association for Computing Machinery, New York, NY, USA, 236–256. https://doi.org/10.1145/1141885.1141890

[6] Nicolas Brunie, Florent de Dinechin, Olga Kupriianova, and Christoph Lauter. 2015. Code Generators for Mathematical Functions. In *2015 IEEE 22nd Symposium on Computer Arithmetic*. 66–73. https://doi.org/10.1109/ARITH.2015.22

[7] Hung Tien Bui and Sofiene Tahar. 1999. Design and synthesis of an IEEE-754 exponential function. In *Engineering Solutions for the Next Millennium. 1999 IEEE Canadian Conference on Electrical and Computer Engineering*, Vol. 1. 450–455 vol.1. https://doi.org/10.1109/CCECE.1999.807240

[8] Sylvain Chevillard, John Harrison, Mioara Joldes, and Christoph Lauter. 2011. Efficient and accurate computation of upper bounds of approximation errors. In *Theoretical Computer Science*, Vol. 412. https://doi.org/10.1016/j.tcs.2010.11.052

[9] Sylvain Chevillard, Mioara Joldes, and Christoph Lauter. 2010. Sollya: An Environment for the Development of Numerical Codes. In *Mathematical Software - ICMS 2010 (Lecture Notes in Computer Science, Vol. 6327)*. Springer, Heidelberg, Germany, 28–31. https://doi.org/10.1007/978-3-642-15582-6_5

[10] Sylvain Chevillard and Christopher Lauter. 2007. A Certified Infinite Norm for the Implementation of Elementary Functions. In *Seventh International Conference on Quality Software (QSIC 2007)*. 153–160. https://doi.org/10.1109/QSIC.2007.4385491

[11] Sangeeta Chowdhary, Jay P. Lim, and Santosh Nagarakatte. 2020. Debugging and Detecting Numerical Errors in Computation with Posits. In *41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. https://doi.org/10.1145/3385412.3386004

[12] William J Cody and William M Waite. 1980. *Software manual for the elementary functions*. Prentice-Hall, Englewood Cliffs, NJ. https://doi.org/10.1137/1024023

[13] Catherine Daramy-Loirat, David Defour, Florent de Dinechin, Matthieu Gallet, Nicolas Gast, Christoph Lauter, and Jean-Michel Muller. 2006. *CR-LIBM A library of correctly rounded elementary functions in double-precision*. Research Report. Laboratoire de l'Informatique du Parallélisme. https://hal-ens-lyon.archives-ouvertes.fr/ensl-01529804

[14] Davide De Caro, Ettore Napoli, Darjn Esposito, Gerardo Castellano, Nicola Petra, and Antonio G. M. Strollo. 2017. Minimizing Coefficients Wordlength for Piecewise-Polynomial Hardware Function Evaluation With Exact or Faithful Rounding. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2017). https://doi.org/10.1109/TCSI.2016.2629850

[15] 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, Article 13 (June 2007). https://doi.org/10.1145/1236463.1236468

[16] Zhoulai Fu and Zhendong Su. 2019. Effective Floating-point Analysis via Weak-distance Minimization. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 439–452. https://doi.org/10.1145/3314221.3314632

[17] Ambros M. Gleixner, Daniel E. Steffy, and Kati Wolter. 2012. Improving the Accuracy of Linear Programming Solvers with Iterative Refinement. In *Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation* (Grenoble, France) *(ISSAC '12)*. Association for Computing Machinery, New York, NY, USA, 187–194. https://doi.org/10.1145/2442829.2442858

[18] Eric Goubault. 2001. Static Analyses of the Precision of Floating-Point Operations. In *Proceedings of the 8th International Symposium on Static Analysis (SAS)*. Springer, 234–259. https://doi.org/10.1007/3-540-47764-0_14

[19] John Gustafson. 2017. *Posit Arithmetic*. https://posithub.org/docs/Posits4.pdf

[20] John Gustafson. 2020. *The Minefield Method: A Uniformly Fast Solution to the Table-Maker's Dilemma*. https://bit.ly/2ZP4kHj

[21] John Harrison. 1997. Floating point verification in HOL light: The exponential function. In *Algebraic Methodology and Software Technology*, Michael Johnson (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 246–260. https://doi.org/10.1007/BFb0000475

[22] John Harrison. 1997. Verifying the Accuracy of Polynomial Approximations in HOL. In *International Conference on Theorem Proving in Higher Order Logics*. https://doi.org/10.1007/BFb0028391

[23] John Harrison. 2009. HOL Light: An Overview. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009 (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer-Verlag, Munich, Germany, 60–66. https://doi.org/10.1007/978-3-642-03359-9_4

[24] Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, and Guillaume Revy. 2011. Computing Floating-Point Square Roots via Bivariate Polynomial Evaluation. *IEEE Trans. Comput.* 60. https://doi.org/10.1109/TC.2010.152

[25] Olga Kupriianova and Christoph Lauter. 2014. Metalibm: A Mathematical Functions Code Generator. In *4th International Congress on Mathematical Software*. https://doi.org/10.1007/978-3-662-44199-2_106

[26] Olga Kupriianova and Christoph Lauter. 2015. Replacing Branches by Polynomials in Vectorizable Elementary Functions. In *Scientific Computing, Computer Arithmetic, and Validated Numerics*, Marco Nehmeier, Jürgen Wolff von Gudenberg, and Warwick Tucker (Eds.). Springer International Publishing, Cham, 14–22. https://doi.org/10.1007/978-3-319-31769-4_2

[27] 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, Article 47 (Dec. 2017), 32 pages. https://doi.org/10.1145/3158135

[28] Vincent Lefèvre and Jean-Michel Muller. 2001. Worst Cases for Correct Rounding of the Elementary Functions in Double Precision. In *15th IEEE Symposium on Computer Arithmetic (Arith '01)*. 111–118. https://doi.org/10.1109/ARITH.2001.930110

[29] Vincent Lefèvre, Jean-Michel Muller, and Arnaud Tisserand. 1998. Toward correctly rounded transcendentals. *IEEE Trans. Comput.* 47, 11 (1998), 1235–1243. https://doi.org/10.1109/12.736435

[30] Cerlane Leong. 2019. *SoftPosit-Math*. https://gitlab.com/cerlane/softposit-math

[31] Jay P. Lim, Mridul Aanjaneya, John Gustafson, and Santosh Nagarakatte. 2020. A Novel Approach to Generate Correctly Rounded Math Libraries for New Floating Point Representations. arXiv:2007.05344 Rutgers Department of Computer Science Technical

Report DCS-TR-753.

[32] Jay P. Lim, Mridul Aanjaneya, John Gustafson, and Santosh Nagarakatte. 2021. An Approach to Generate Correctly Rounded Math Libraries for New Floating Point Variants. *Proceedings of the ACM on Programming Languages* 6, POPL, Article 29 (Jan. 2021), 30 pages. https://doi.org/10.1145/3434310

[33] Jay P. Lim and Santosh Nagarakatte. 2021. *RLibm-32*. https://github.com/rutgers-apl/rlibm-32

[34] Jay P Lim and Rutgers University Santosh Nagarakatte. 2021. RLIBM-32: High Performance Correctly Rounded Math Libraries for 32-bit Floating Point Representations. Rutgers Department of Computer Science Technical Report DCS-TR-754.

[35] Jay P. Lim, Matan Shachnai, and Santosh Nagarakatte. 2020. Approximating Trigonometric Functions for Posits Using the CORDIC Method. In *Proceedings of the 17th ACM International Conference on Computing Frontiers* (Catania, Sicily, Italy) *(CF '20)*. Association for Computing Machinery, New York, NY, USA, 19–28. https://doi.org/10.1145/3387902.3392632

[36] Guillaume Melquiond. 2019. *Gappa*. http://gappa.gforge.inria.fr

[37] Jean-Michel Muller. 2005. *Elementary Functions: Algorithms and Implementation*. Birkhauser. https://doi.org/10.1007/978-1-4899-7983-4

[38] 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*, Vol. 50. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2813885.2737959

[39] Eugene Remes. 1934. Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *Comptes rendus de l'Académie des Sciences* 198 (1934), 2063–2065.

[40] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2018. Finding Root Causes of Floating Point Error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 256–269. https://doi.org/10.1145/3296979.3192411

[41] Jun Sawada. 2002. Formal verification of divide and square root algorithms using series calculation. In *3rd International Workshop on the ACL2 Theorem Prover and its Applications*.

[42] Pat H Sterbenz. 1974. *Floating-point computation*. Prentice-Hall, Englewood Cliffs, NJ.

[43] Shane Story and Ping Tak Peter Tang. 1999. New algorithms for improved transcendental functions on IA-64. In *Proceedings 14th IEEE Symposium on Computer Arithmetic*. 4–11. https://doi.org/10.1109/ARITH.1999.762822

[44] Ping-Tak Peter Tang. 1989. Table-Driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic. *ACM Trans. Math. Software* 15, 2 (June 1989), 144–157. https://doi.org/10.1145/63522.214389

[45] Ping-Tak Peter Tang. 1990. Table-Driven Implementation of the Logarithm Function in IEEE Floating-Point Arithmetic. *ACM Trans. Math. Software* 16, 4 (Dec. 1990), 378–400. https://doi.org/10.1145/98267.98294

[46] P. T. P. Tang. 1991. Table-lookup algorithms for elementary functions and their error analysis. In *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic*. 232–236. https://doi.org/10.1109/ARITH.1991.145565

[47] Lloyd N. Trefethen. 2012. *Approximation Theory and Approximation Practice (Other Titles in Applied Mathematics)*. Society for Industrial and Applied Mathematics, USA.

[48] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2019. Efficient Automated Repair of High Floating-Point Errors in Numerical Libraries. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 56 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290369

[49] Abraham Ziv. 1991. Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit. *ACM Trans. Math. Software* 17, 3 (Sept. 1991), 410–423. https://doi.org/10.1145/114697.116813

[50] Daming Zou, Muhan Zeng, Yingfei Xiong, Zhoulai Fu, Lu Zhang, and Zhendong Su. 2019. Detecting Floating-Point Errors via Atomic Conditions. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 60 (Dec. 2019), 27 pages. https://doi.org/10.1145/3371128