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

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]. Furthermore, 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 float16 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., \( \sinp(x) \)). Finally, we need to ensure that output compensation does not experience pathological cancellation errors (e.g., \( \cosp(x) \)).

**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., \( \sinp(x) \) 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., \( \cosp(x) \) in Section 5).
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 ∈ (−∞, ∞). There are four billion inputs with a 32-bit float. There are three kinds of special cases:

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

For the first class of special cases, we compute \( \pi x \) 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,

\[
\text{sinpi}(J) = (-1)^K \times \text{sinpi}(L)
\]

Given that \( \text{sinpi} \) between [0.5, 1) is a mirror image of values between [0, 0.5), we further reduce as follows:

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

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 round to the correctly rounded result. We call this interval the rounding interval. If our
polynomial approximation produces a value in the rounding interval, the rounded result is the correct result. Consider the inputs:

\[
\begin{align*}
x_1 &= 1.95312686264514923095703125 \times 10^{-3} \\
x_2 &= 2.148437686264514923095703125 \times 10^{-2}
\end{align*}
\]

We show the correctly rounded result of \(\sin pi(x)\) 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 \(\sin pi(x)\), we need to compute both \(\sin pi(R)\) and \(\cos pi(R)\) (i.e., multiple elementary functions). The result that we produce for \(\sin pi(R)\) and \(\cos pi(R)\) should allow us to compute the correct result for \(\sin pi(x)\) (i.e., produce a value within the rounding interval \([l, h]\) of input \(x\)).

To compute \(\sin pi(x)\), we will generate piecewise polynomials for \(\sin pi(R)\) and \(\cos pi(R)\). 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 \([l', h']\) for the output of \(\sin pi(R)\) and an interval \([l'h']\) for the output of \(\cos pi(R)\) such that the result of output compensation produces a value within the rounding interval for \(x\). We compute the correctly rounded value \((o)\) of \(\sin pi(R)\) in double using the oracle and set it as our initial guess for \([l', h']\) (i.e., \([o, o]\) a singleton). Similarly, we compute the interval \([l'h']\) for \(\cos pi(R)\). 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 \(\sin pi(R)\) and \(\cos pi(R)\) and check if output compensation produces the correct result for all inputs. Similarly, we deduce the upper bound for both \(\sin pi(R)\) and \(\cos pi(R)\). The reduced interval for \(\sin pi(R)\) from the perspective of \(x_1\) is \([ls', hs']\). Similarly, the reduced interval for \(\sin pi(R)\) from the perspective of \(x_2\) is \([ls'', hs'']\). These reduced intervals for \(\sin pi(R)\) 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 \(\sin pi(R)\) and \(\cos pi(R)\). We illustrate this process with \(\sin pi(R)\). It is similar for \(\cos pi(R)\). Even after range reduction, there are approximately 110 million unique reduced inputs for \(R \in [0, \pi]\). 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 \([0, \pi]\). We use the bit-pattern of the reduced input \(R\) in double to identify the sub-domain. Although the domain of \(R\) is \([0, \pi]\), the value of \(R\) in our reduced inputs ranges between \([2^{-32}, 2^{-32}]\) 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
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 \( RN_T(f(x)) \) to denote the rounding function that rounds \( f(x) \) computed with real numbers to target representation \( T \). 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 \( T \). 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 \( RR_{\mathbb{E}}(x) \), 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 \( OC_{\mathbb{E}}(y', x) \), 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 \( \sin \) and \( \cos \) after range reduction in Section 2). We need to generate polynomials \( \Psi_i \) 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 \( T \) (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($f, X, RR_{\mathbb{H}}, OC_{\mathbb{H}}, d)$:
2 $Y \leftarrow \emptyset$
3 foreach $x \in X$ do
4     $y \leftarrow RN_{\mathbb{H}}(f(x))$
5     $[l, h] \leftarrow$ RoundingInterval($y, T, \mathbb{H}$)
6     $Y \leftarrow (x, [l, h])$
7 $L \leftarrow$ ReducedIntervals($Y, RR_{\mathbb{H}}, OC_{\mathbb{H}}$)
8 Result $\leftarrow \emptyset$
9 foreach $(f_i, L_i) \in L$ do
10     if $L_i \subseteq \emptyset$ then return $\emptyset$
11     $\Psi_i \leftarrow$ GenApproxFunc($L_i, d$)
12     Result $\leftarrow (f_i, \Psi_i) \cup$ Result
13 return Result

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 functions. It produces the correctly rounded result of $f(x)$ for each input $x \in X$. RoundingInterval computes the rounding interval $[l, h] \subseteq \mathbb{H}$ of $y$ where all values in the interval round to $y$ in $T$. ReducedIntervals is shown in Algorithm 2 while GenApproxFunc is shown in Algorithm 3.

For 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 $T$.

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 $T$. As $\mathbb{H}$ has higher precision than $T$, there is a range of values in $\mathbb{H}$ that our approach can produce and still round to the correctly rounded result in $T$. 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).

1 Function ReducedIntervals($Y, RR_{\mathbb{H}}, OC_{\mathbb{H}}$):
2 if $OC_{\mathbb{H}}$ is not a monotonic function then return $\emptyset$
3 $F \leftarrow$ {list of functions used in $OC_{\mathbb{H}}$}
4 foreach $f_i \in F$ do $L_i \leftarrow \emptyset$
5 foreach $(x, [l, h]) \in Y$ do
6     $r \leftarrow RR_{\mathbb{H}}(x)$
7     $V \leftarrow$ {RN$_{\mathbb{H}}(f_i(r)) | f_i \in F}$
8     if $OC_{\mathbb{H}}(V, x) \notin [l, h]$ then return $\emptyset$
9     //Set initial reduced range for each $f_i(r)$
10     $I' \leftarrow \{[v, v] | v \in V\}$
11     //Decrease the lower bounds $I'_l$ simultaneously
12     while true
13         $A \leftarrow$ {GetPrev($I'_l$, $\mathbb{H}$) | $I'_l$ $\in$ $I'$}
14         if $OC_{\mathbb{H}}(A, x) \notin [l, h]$ then break
15         $I' \leftarrow$ {GetPrev($I'_l$, $\mathbb{H}$) $I'_l$ $\in$ $I'$}
16     //Increase the upper bounds $I'_h$ simultaneously
17     while true
18         $B \leftarrow$ {GetNext($I'_h$, $\mathbb{H}$) $I'_h$ $\in$ $I'$}
19         if $OC_{\mathbb{H}}(B, x) \notin [l, h]$ then break
20         $I' \leftarrow$ {GetNext($I'_h$, $\mathbb{H}$) $I'_h$ $\in$ $I'$}
21     foreach $[l', h']$ $\in$ $I'$ do
22         $L_i \leftarrow L_i \cup (r, [l', h'])$
23 return $\{(f_i, L_i) | f_i \in F\}$

Algorithm 2: ReducedIntervals computes the reduced interval $[l', h']$ 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 $[l', h']$, then we can generate the correctly rounded result for $x$. ReducedIntervals returns a list with $(r, [l', h'])$ 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 $\mathbb{H}$.

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 $T$ and the largest value $h \in \mathbb{H}$ that rounds to $y$ when rounded to $T$. This search procedure can be efficiently implemented either using a binary search or by leveraging the properties of $T$ 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 \((OC)\) 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 \((OC)\) (line 4). Then, we compute the correctly rounded result \(r_i\) 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., \(u_i\)) 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 \([u_i, u_i]\) (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 \(\mathcal{L}_i\).

### Algorithm 3: GenApproxFunc

```plaintext
1 Function GenApproxFunc(L, d):
2 \(\mathcal{L}^- \leftarrow \{(r, [l'_i, h'_i]) \in \mathcal{L} | r < 0\}
3 \mathcal{L}^+ \leftarrow \{(r, [l'_i, h'_i]) \in \mathcal{L} | r \geq 0\}
4 \Psi^- \leftarrow GenApproxHelper(\mathcal{L}^-, d)
5 \Psi^+ \leftarrow GenApproxHelper(\mathcal{L}^+, d)
6 return \{\Psi^-, \Psi^+\}
```

```plaintext
1 Function GenApproxHelper(L, d):
2 n \leftarrow 0
3 while true do
4 \(\Delta = SplitDomain(L, n)\)
5 \((status, \Psi) = GenPiecewise(\Delta, d)\)
6 if status = true then return \Psi
7 n \leftarrow n + 1
```

```plaintext
1 Function GenPiecewise(\Delta, d):
2 \Psi \leftarrow \emptyset
3 foreach \(\Delta_j \in \Delta\) do
4 \((status, \Psi_j) = GenPolynomial(\Delta_j, d)\)
5 if status = false then return \(false, \emptyset\)
6 \Psi \leftarrow \Psi \cup \Psi_j
7 return \(true, \Psi\)
```

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 \((r, [l'_1, h'_1])\) and \((r, [l'_2, h'_2])\) for the same reduced input \(r\) corresponding to original inputs \(x_1\) and \(x_2\). The reduced intervals \([l'_1, h'_1]\) and \([l'_2, h'_2]\) 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 \((r, [l'_1, h'_1])\) to produce the correctly rounded result for \(x_1\) and \((r, [l'_2, h'_2])\) to produce the correctly rounded result for \(x_2\). 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 \((r, [l', h'])\) in \(L\) 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 \(L_i\). 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 \(H\).

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 \(H\) 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: \(L^-\) that contains negative reduced inputs and \(L^+\) that contains non-negative reduced inputs (lines 2-3). We create polynomial approximations for each \(L^-\) and \(L^+\) (lines 4-5). This step also allows us to subsequently group the reduced input into sub-domains in an efficient manner.

If \(L\) contains only negative or positive reduced inputs, we try to generate a single polynomial of degree \(d\) that satisfies all reduced constraints in \(L\) (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 \(L\) into multiple sub-domains (lines 9-13 in GenApproxPer). We iteratively split the domain of reduced inputs into 2^n sub-domains based on the bit-pattern of \(r\) in \(H\) (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 \(H\) 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 (\(\Delta\) returned by SplitDomain). We try to generate a polynomial of degree \(d\) that satisfies all reduced constraints in \(\Delta_j\) for all \(\Delta_j\)’s belonging to \(f_i\) (lines 16-19). Using bit-patterns of the reduced input in \(H\) allows us to efficiently identify the sub-domain for the piecewise polynomial with two bitwise operations (and and a shift).

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., \( \Delta_j \)) for generating a polynomial approximation for an elementary function \( f_j \) 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 \( \epsilon \), 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:

\[
\begin{bmatrix}
l_1^1 & l_2^1 & \cdots & l_n^1 & 1 & r_1 & \cdots & r_n & c_0 \\
l_1^2 & l_2^2 & \cdots & l_n^2 & 1 & r_1 & \cdots & r_n & c_1 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
l_1^n & l_2^n & \cdots & l_n^n & 1 & r_1 & \cdots & r_n & c_d \\
\end{bmatrix}
\begin{bmatrix}
c_0 \\
c_1 \\
\vdots \\
c_d \\
\end{bmatrix}
\leq
\begin{bmatrix}
h_1^1 \\
h_2^1 \\
\vdots \\
h_n^1 \\
\end{bmatrix}
\]

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 \( L_i \) into even smaller sub-domains and repeat this process.

If we successfully generate a polynomial \( \Psi_j \) that satisfies all constraints in the sample, then we check whether this polynomial satisfies all constraints in \( \Delta_j \) (line 10-15). If \( \Psi_j \) satisfies all constraints, then we return the polynomial (line 7). If there is any constraint not satisfied by \( \Psi_j \) 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 \( \Delta_j \). 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 [15, 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 X. For each X, we show the number of inputs with wrong results. N/A indicates that the implementation is not available.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ln(x)</td>
<td>✓</td>
<td>✓ (4.2E5)</td>
<td>X (5)</td>
<td>X (1060)</td>
<td>X (5)</td>
<td>X (5)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>log2(x)</td>
<td>✓</td>
<td>X (3.1E5)</td>
<td>✓ (3.1E5)</td>
<td>✓ (276)</td>
<td>✓ (3.1E5)</td>
<td>✓ (276)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>log10(x)</td>
<td>✓</td>
<td>X (3.0E7)</td>
<td>X (1)</td>
<td>X (1.5E5)</td>
<td>X (1)</td>
<td>X (1)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>exp(x)</td>
<td>✓</td>
<td>X (1.7E5)</td>
<td>✓ (1.7E5)</td>
<td>✓ (2.5E5)</td>
<td>✓ (1.7E5)</td>
<td>✓ (2.5E5)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>exp2(x)</td>
<td>✓</td>
<td>X (1.7E5)</td>
<td>X (2)</td>
<td>X (7.2E5)</td>
<td>X (2)</td>
<td>X (2)</td>
<td>X (5.1E8)</td>
<td>X (5.1E8)</td>
</tr>
<tr>
<td>sinh(x)</td>
<td>✓</td>
<td>X (7.1E7)</td>
<td>X (2)</td>
<td>X (2.5E5)</td>
<td>X (2)</td>
<td>X (2)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>cosh(x)</td>
<td>✓</td>
<td>X (1.8E7)</td>
<td>✓ (1.8E7)</td>
<td>✓ (1.4E5)</td>
<td>✓ (1.8E7)</td>
<td>✓ (1.4E5)</td>
<td>✓ (1.1E7)</td>
<td>✓ (1.1E7)</td>
</tr>
<tr>
<td>sinpi(x)</td>
<td>✓</td>
<td>N/A</td>
<td>N/A</td>
<td>X (3.4E5)</td>
<td>✓</td>
<td>✓</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>cospi(x)</td>
<td>✓</td>
<td>N/A</td>
<td>N/A</td>
<td>X (3.8E5)</td>
<td>✓</td>
<td>✓</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**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 for 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 X.

<table>
<thead>
<tr>
<th>posit32 functions</th>
<th>Using RLibm-32</th>
<th>Using glibc double</th>
<th>Using Intel double</th>
<th>Using CR-LIBM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ln(x)</td>
<td>✓</td>
<td>X(22)</td>
<td>X(22)</td>
<td>X(22)</td>
</tr>
<tr>
<td>log2(x)</td>
<td>✓</td>
<td>X(19)</td>
<td>X(18)</td>
<td>X(18)</td>
</tr>
<tr>
<td>log10(x)</td>
<td>✓</td>
<td>X(26)</td>
<td>X(23)</td>
<td>X(23)</td>
</tr>
<tr>
<td>Exp(x)</td>
<td>✓</td>
<td>X(4.4E8)</td>
<td>X(4.4E8)</td>
<td>X(4.4E8)</td>
</tr>
<tr>
<td>Exp2(x)</td>
<td>✓</td>
<td>X(4.0E8)</td>
<td>X(4.0E8)</td>
<td>N/A</td>
</tr>
<tr>
<td>Exp10(x)</td>
<td>✓</td>
<td>X(5.2E8)</td>
<td>X(5.2E8)</td>
<td>N/A</td>
</tr>
<tr>
<td>Sinh(x)</td>
<td>✓</td>
<td>X(4.4E8)</td>
<td>X(4.4E8)</td>
<td>X(4.4E8)</td>
</tr>
<tr>
<td>Cosh(x)</td>
<td>✓</td>
<td>X(4.4E8)</td>
<td>X(4.4E8)</td>
<td>X(4.4E8)</td>
</tr>
</tbody>
</table>

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 $\infty$ 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.

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.

<table>
<thead>
<tr>
<th>f(x)</th>
<th>Gen. Time (Minutes)</th>
<th>Reduced Inputs</th>
<th># of Polynomials</th>
<th>Degree</th>
<th># of Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>float functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ln(x)</td>
<td>218</td>
<td>7.2E6</td>
<td>2$^{10}$</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>log2(x)</td>
<td>251</td>
<td>7.2E6</td>
<td>2$^8$</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>log10(x)</td>
<td>429</td>
<td>7.2E6</td>
<td>2$^8$</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>exp(x)</td>
<td>117</td>
<td>5.2E8</td>
<td>2$^{7}$</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>exp2(x)</td>
<td>86</td>
<td>3.0E8</td>
<td>2$^{3}$</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>exp10(x)</td>
<td>169</td>
<td>5.2E8</td>
<td>2$^{9}$</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>sinh(x)</td>
<td>28</td>
<td>1.5E8</td>
<td>2$^7$</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>cosh(x)</td>
<td>24</td>
<td>1.5E8</td>
<td>2$^6$</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>sinpi(x)</td>
<td>30</td>
<td>1.2E8</td>
<td>2$^{8}$</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>cospi(x)</td>
<td>19</td>
<td>1.2E8</td>
<td>1$^{4}$</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>posit32 functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ln(x)</td>
<td>264</td>
<td>1.1E8</td>
<td>2$^{11}$</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>log2(x)</td>
<td>288</td>
<td>1.1E8</td>
<td>2$^6$</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>log10(x)</td>
<td>685</td>
<td>1.1E8</td>
<td>2$^{12}$</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>exp(x)</td>
<td>1089</td>
<td>3.5E9</td>
<td>2$^{12}$</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>exp2(x)</td>
<td>814</td>
<td>7.9E8</td>
<td>2$^{12}$</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>exp10(x)</td>
<td>1528</td>
<td>3.4E9</td>
<td>2$^{13}$</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>sinh(x)</td>
<td>461</td>
<td>1.6E9</td>
<td>2$^{14}$</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>cosh(x)</td>
<td>528</td>
<td>1.7E9</td>
<td>2$^{14}$</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

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.

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
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 20 (i.e., a single polynomial) to 212. Figure 5 reports the performance of log2(x) and log10(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 2d sub-domains. It requires 6KB for storing coefficients of piecewise polynomials.
We describe the case study with \( x \) of \( \cos pi(x) \) to be approximated. Similar to range reduction for \( \pi \) cases, there are more than 600 million float inputs that need for even integers and \( \pi \) elementary function to treat the importance of carefully designing range reduction \( \pi \) increase in the number of sub-domains when compared to a single \( \pi \). Figure 5.

5 Case Study with \( \cos pi(x) \) for Float

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

Special cases. There are three kinds of special cases:

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

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

Range reduction of \( \cos pi(x) \). After excluding special cases, there are more than 600 million float inputs that need to be approximated. Similar to range reduction for \( \sin pi(x) \) (Section 2.1), we use periodicity and trigonometric identities of \( \cos pi(x) \) 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, \( \cos pi(x) = \cos pi(J) \). 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, \( \cos pi(J) \) can be computed with,

\[
\cos pi(J) = (-1)^K \cos pi(L)
\]

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

\[
M = \begin{cases} 
0 & \text{if } L \leq 0.5 \\
1 & \text{if } L > 0.5
\end{cases}
\]

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

We have \( \cos pi(L) = (-1)^M \cos pi(L') \). 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 \( [0, 0.5] \). 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 \( [0, 0.5] \). One possible method to compute \( \cos pi(L') \) is to use the trigonometric identity \( \cos pi(a + b) = \cos pi(a)\cos pi(b) - \sin pi(a)\sin pi(b) \),

\[
\cos pi\left(\frac{N}{512} + Q\right) = \cos pi\left(\frac{N}{512}\right)\cos pi(Q) - \sin pi\left(\frac{N}{512}\right)\sin pi(Q)
\]

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

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:

\[
N' = \begin{cases} 
0 & \text{if } N = 0 \\
N + 1 & \text{otherwise}
\end{cases}
\]

\[
R = \begin{cases} 
\frac{Q}{512} - Q & \text{if } N = 0 \\
\frac{Q}{512} & \text{otherwise}
\end{cases}
\]

Then, we can compute \( \cos pi(L') = \cos pi\left(\frac{N'}{512} - R\right) \) using the trigonometric identity \( \cos pi(a - b) = \cos pi(a)\cos pi(b) + \sin pi(a)\sin pi(b) \) as follows,

\[
\cos pi(L') = \begin{cases} 
\cos pi(R) & \text{if } N = 0 \\
\cos pi\left(\frac{N'}{512}\right)\cos pi(R) + \sin pi\left(\frac{N'}{512}\right)\sin pi(R) & \text{if } N \neq 0
\end{cases}
\]

This output compensation is monotonic and does not experience cancellation error. The values of \( N' \) ranges from 0 to 256 and \( R \in [0, 0.5] \). The computation \( \frac{Q}{512} - Q \) 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 \( \cos p i (\frac{N}{512}) \) and \( \sin p i (\frac{N}{512}) \) in lookup tables (i.e., 514 values in total). We create polynomial approximations for \( \sin p i (R) \) and \( \cos p i (R) \) 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 \( \sin p i (R) \) and a single 4\(^{th}\) degree even polynomial for \( \cos p i (R) \). Finally, we can compute the result for \( \cos p i (x) \) with the output compensation function,

\[
\cos p i (x) = \begin{cases} 
S \times \cos p i (R) & \text{ if } N = 0 \\
S \times (c p n \times \cos p i (R) + s p n \times \sin p i (R)) & \text{ if } N \neq 0
\end{cases}
\]

where \( S = (-1)^{K} \times (-1)^{M} \), \( c p n = \cos p i (\frac{N}{512}) \), and \( s p n = \sin p i (\frac{N}{512}) \). These polynomials combined with the output compensation functions produce correctly rounded results for all inputs for \( \sin p i (x) \) and \( \cos p i (x) \).

### 6 Related Work

Multiple decades of seminal work in this area has advanced the state-of-the-art on creating approximations for FP representations [7, 14, 15, 24, 37, 39, 47, 49]. Further, seminal work on range reduction has made such approximations 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 \( \pi \). 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


Report DCS-TR-753.


