An Approach to Generate Correctly Rounded Math Libraries for New Floating Point Variants

JAY P. LIM, Rutgers University, United States
MRIDUL AANJANEYA, Rutgers University, United States
JOHN GUSTAFSON, National University of Singapore, Singapore
SANTOSH NAGARAKATTE, Rutgers University, United States

Given the importance of floating point (FP) performance in numerous domains, several new variants of FP and its alternatives have been proposed (*e.g.*, `Bfloat16`, `TensorFloat32`, and `posits`). These representations do not have correctly rounded math libraries. Further, the use of existing FP libraries for these new representations can produce incorrect results. This paper proposes a novel approach for generating polynomial approximations that can be used to implement correctly rounded math libraries. Existing methods generate polynomials that approximate the real value of an elementary function $f(x)$ and produce wrong results due to approximation errors and rounding errors in the implementation. In contrast, our approach generates polynomials that approximate the correctly rounded value of $f(x)$ (*i.e.*, the value of $f(x)$ rounded to the target representation). It provides more margin to identify efficient polynomials that produce correctly rounded results for all inputs. We frame the problem of generating efficient polynomials that produce correctly rounded results as a linear programming problem. Using our approach, we have developed correctly rounded, yet faster, implementations of elementary functions for multiple target representations.

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

Additional Key Words and Phrases: floating point, posits, correctly rounded math libraries

ACM Reference Format:
Jay P. Lim, Mridul Aanjaneya, John Gustafson, and Santosh Nagarakatte. 2021. An Approach to Generate Correctly Rounded Math Libraries for New Floating Point Variants. *Proc. ACM Program. Lang.* 5, POPL, Article 29 (January 2021), 30 pages. https://doi.org/10.1145/3434310

1 INTRODUCTION

Approximating real numbers. Every programming language has primitive data types to represent numbers. The floating point (FP) representation, which was standardized with the IEEE-754 standard [Cowlishaw 2008], is widely used in mainstream languages to approximate real numbers. For example, every number in JavaScript is a FP number! There is an ever-increasing need for improved FP performance in domains such as machine learning and high performance computing (HPC). Hence, several new variants and alternatives to FP have been proposed recently such as `Bfloat16` [Tagliavini et al. 2018], posits [Gustafson 2017; Gustafson and Yonemoto 2017], and `TensorFloat32` [NVIDIA 2020].

Authors' addresses: Jay P. Lim, Computer Science, Rutgers University, United States, jpl169@cs.rutgers.edu; Mridul Aanjaneya, Computer Science, Rutgers University, United States, mridul.aanjaneya@rutgers.edu; John Gustafson, Computer Science, National University of Singapore, Singapore, john.gustafson@nus.edu.sg; Santosh Nagarakatte, Computer Science, Rutgers University, United States, santosh.nagarakatte@cs.rutgers.edu.

2475-1421/2021/1-ART29
https://doi.org/10.1145/3434310

`Bfloat16` [Tagliavini et al. 2018] is a 16-bit FP representation with 8-bits of exponent and 7-bits for the fraction. It is already available in Intel FPGAs [Intel 2019] and Google TPUs [Wang and Kanwar 2019]. `Bfloat16`'s dynamic range is similar to a 32-bit `float` but has lower memory traffic and footprint, which makes it appealing for neural networks [Kalamkar et al. 2019]. Nvidia's `TensorFloat32` [NVIDIA 2020] is a 19-bit FP representation with 8-bits of exponent and 10-bits for the fraction, which is available with Nvidia's Ampere architecture. `TensorFloat32` provides the dynamic range of a 32-bit `float` and the precision of `half` data type (*i.e.*, 16-bit float), which is intended for machine learning and HPC applications. In contrast to FP, posit [Gustafson 2017; Gustafson and Yonemoto 2017] provides tapered precision with a fixed number of bits. Depending on the value, the number of bits available for representing the fraction can vary. Inspired by posits, a tapered precision log number system has been shown to be effective with neural networks [Bernstein et al. 2020; Johnson 2018].

Correctly rounded math libraries. Any number system that approximates real numbers needs a math library that provides implementations for elementary functions [Muller 2005] (*i.e.*, $log(x)$, $exp(x)$, $sqrt(x)$, $sin(x)$). The recent IEEE-754 standard recommends (although it does not require) that the programming language standards define a list of math library functions and implement them to produce the correctly rounded result [Cowlishaw 2008]. Any application using an erroneous math library will produce erroneous results.

A correctly rounded result of an elementary function f for an input x is defined as the value produced by computing the value of $f(x)$ with real numbers and then rounding the result according to the rounding rule of the target representation. Developing a correct math library is a challenging task. Hence, there is a large body of work on accurately approximating elementary functions [Brisebarre et al. 2006; Brunie et al. 2015; Bui and Tahar 1999; Chevillard et al. 2011, 2010; Chevillard and Lauter 2007; Gustafson 2020; Jeannerod et al. 2011; Kupriianova and Lauter 2014; Lefèvre et al. 1998; Lim et al. 2020b], verifying the correctness of math libraries [Boldo et al. 2009; Daumas et al. 2005; de Dinechin et al. 2011; de Dinechin et al. 2006; Harrison 1997a,b; Lee et al. 2017; Sawada 2002], and repairing math libraries to increase the accuracy [Yi et al. 2019]. There are a few correctly rounded math libraries for `float` and `double` types in the IEEE-754 standard [Daramy et al. 2003; Fousse et al. 2007; IBM 2008; Microsystems 2008; Ziv 1991]. Widely used math libraries (*e.g.*, `libm` in `glibc` or Intel's math library) do not produce correctly rounded results for all inputs.

New representations lack math libraries. The new FP representations currently do not have math libraries specifically designed for them. One stop-gap alternative is to promote values from new representations to a `float`/`double` value and use existing FP libraries for them. For example, we can convert a `Bfloat16` value to a 32-bit `float` and use the FP math library. However, this approach can produce wrong results for the `Bfloat16` value even when we use the correctly rounded `float` library (see Section 2.6 for a detailed example). This approach also has suboptimal performance as the math library for `float`/`double` types probably uses a polynomial of a large degree with many more terms than necessary to approximate these functions.

Prior approaches for creating math libraries. Most prior approaches use minimax approximation methods (*i.e.*, Remez algorithm [Remes 1934] or Chebyshev approximations [Trefethen 2012]) to generate polynomials that have the smallest error compared to the real value of an elementary function. Typically, range reduction techniques are used to reduce the input domain such that the polynomial only needs to approximate the elementary function for a small input domain. Subsequently, the result of the polynomial evaluation on the small input domain is adjusted to produce the result for the entire input domain, which is known as output compensation. Polynomial evaluation, range reduction, and output compensation are implemented in some finite representation that has higher precision than the target representation. The approximated result is finally rounded to the target representation.

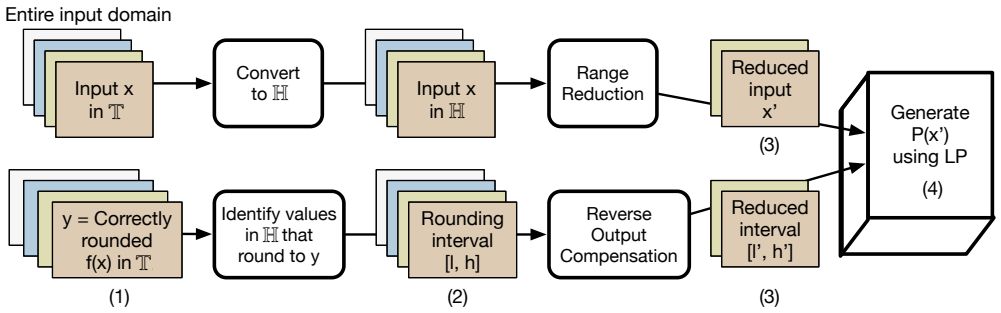


Fig. 1. Our approach to generate correctly rounded elementary functions for a target representation ($\mathbb{T}$). The math library is implemented in representation $\mathbb{H}$. The goal is to synthesize a polynomial $P(x')$ using linear programming such that the final result after range reduction and output compensation is the correctly rounded result of $f(x)$ in $\mathbb{T}$. (1) For each input x in $\mathbb{T}$, we compute the correctly rounded value of $f(x)$ (denoted as y) using an oracle. (2) Based on y, we identify an interval ($[l, h]$) where all values in the interval round to y. (3) Then, we compute the reduced input x' using range reduction and the reduced interval ($[l', h']$) such that when the output of the polynomial on the reduced input x' is adjusted (*i.e.*, output compensation), it produces the result for the original input and it is in $[l, h]$. (4) Finally, we synthesize $P(x')$ that produces a value in the reduced interval $[l', h']$ for each reduced input x'.

When the result of an elementary function $f(x)$ with reals is extremely close to the rounding-boundary (*i.e.*, $f(x)$ rounds to a value v_1 but $f(x) + \epsilon$ rounds to a different value v_2 for very small value ϵ), then the error of the polynomial must be smaller than ϵ to ensure that the result of the polynomial produces the correctly rounded value [Lefèvre and Muller 2001]. This probably necessitates a polynomial of a large degree with many terms. Further, there can be round-off errors in polynomial evaluation with a finite precision representation. Hence, the result produced may not be the correctly rounded result.

Our approach. This paper proposes a novel approach to generate correctly rounded implementations of elementary functions by framing it as a linear programming problem. In contrast to prior approaches that generate polynomials by minimizing the error compared to the real value of an elementary function $f(x)$, we propose to generate polynomials that directly approximate the correctly rounded value of $f(x)$ inspired by the Minefield approach [Gustafson 2020]. Specifically, we identify an interval of values for each input that will result in a correctly rounded output and use that interval to generate the polynomial approximation. For each input x_i, we use an oracle to generate an interval $[l_i, h_i]$ such that all real values in this interval round to the correctly rounded value of $f(x_i)$. Using these intervals, we can subsequently generate a set of constraints, which is given to a linear programming solver, to generate a polynomial that computes the correctly rounded result for all inputs. The interval $[l_i, h_i]$ for correctly rounding the output of input x_i is larger than $[f(x_i) - \epsilon, f(x_i) + \epsilon]$ where ϵ is the maximum error of the polynomial generated using prior methods. Hence, our approach has larger freedom to generate polynomials that produce correctly rounded results and also provide better performance.

Handling range reduction. Typically, generating polynomials for a small input domain is easier than a large input domain. Hence, the input is reduced to a smaller domain with range reduction. Subsequently, polynomial approximation is used for the reduced input. The resulting value is adjusted with output compensation to produce the final output. For example, the input domain for $log_2(x)$ is $(0, \infty)$. Approximating this function with a polynomial is much easier over the domain $[1, 2)$ when compared to the entire input domain $(0, \infty)$. Hence, we range reduce the

input x into z using $x = z * 2^e$, where $z \in [1, 2)$ and e is an integer. We compute $y' = log_2(z)$ using our polynomial for the domain $[1, 2)$. We compute the final output y using the range reduced output y' and the output compensation function, which is $y = y' + e$. Polynomial evaluation, range reduction, and output compensation are performed with a finite precision representation (*e.g.*, `double`) and can experience numerical errors. Our approach for generating correctly rounded outputs has to consider the numerical error with output compensation. To account for rounding errors with range reduction and output compensation, we constrain the output intervals that we generated for each input x in the entire input domain (see Section 4). When our approach generates a polynomial, it is guaranteed that the polynomial evaluation along with the range reduction and output compensation can be implemented with finite precision to produce a correctly rounded result for all inputs of an elementary function $f(x)$. Figure 1 pictorially provides an overview of our methodology.

RLibm. We have developed a collection of correctly rounded math library functions, which we call RLibm, for `Bfloat16`, posits, and floating point using our approach. RLibm is open source [Lim and Nagarakatte 2020a,b]. Concretely, RLibm contains twelve elementary functions for `Bfloat16`, eleven elementary functions for 16-bit posits, and $log_2(x)$ function for a 32-bit `float` type. We have validated that our implementation produces the correctly rounded result for all inputs. In contrast, `glibc`'s $log_2(x)$ function for a 32-bit float produces wrong results for more than fourteen million inputs. Similarly, Intel's math library also produces wrong results for 276 inputs. We also observed that re-purposing `glibc`'s and Intel's float library for `Bfloat16` produces a wrong result for 10^x.

Our library functions for `Bfloat16` are on average 2.02× faster than the `glibc`'s `double` library and 1.39× faster than the `glibc`'s `float` library. Our library functions for `Bfloat16` are also 1.44× and 1.30× faster than the Intel's `double` and `float` math libraries, respectively.

Contributions. This paper makes the following contributions.

- Proposes a novel approach that generates polynomials based on the correctly rounded value of an elementary function rather than minimizing the error between the real value and the approximation.
- Demonstrates that the task of generating polynomials with correctly rounded results can be framed as a linear programming problem while accounting for range reduction.
- Demonstrates RLibm, a library of elementary functions that produce correctly rounded results for all inputs for various new alternatives to floating point such as `Bfloat16` and posits. Our functions are faster than state-of-the-art libraries.

2 BACKGROUND AND MOTIVATION

We provide background on the FP representation and its variants (*i.e.*, `Bfloat16`), the posit representation, the state-of-the-art for developing math libraries, and a motivating example illustrating how the use of existing libraries for new representations can result in wrong results.

2.1 Floating Point and Its Variants

The FP representation $\mathbb{F}_{n,|E|}$, which is specified in the IEEE-754 standard [Cowlishaw 2008], is parameterized by the total number of bits n and the number of bits for the exponent $|E|$. There are three components in a FP bit-string: a sign bit s, $|E|$-bits to represent the exponent, and $|F|$-bits to represent the mantissa F where $|F| = n - 1 - |E|$. Figure 2(a) shows the FP format. If $s = 0$, then the value is positive. If $s = 1$, then the value is negative. The value represented by the FP bit-string is a normal value if the bit-string E, when interpreted as an unsigned integer, satisfies $0 < E < 2^{|E|} - 1$. The normal value represented with this bit-string is $(1 + \frac{F}{2^{|F|}}) \times 2^{E-bias}$, where bias is $2^{|E|-1} - 1$. If $E = 0$, then the FP value is a denormal value. The value of the denormal value is $(\frac{F}{2^{|F|}}) \times 2^{1-bias}$.

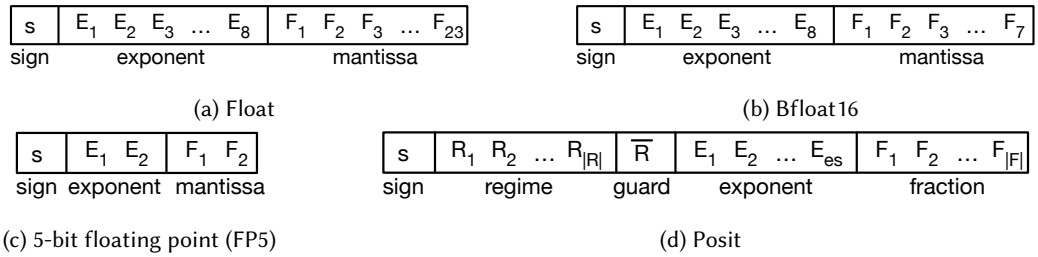


Fig. 2. (a) The bit-string for a 32-bit FP format (`float`). (b) The bit-string for the `Bfloat16` representation. (c) a 5-bit FP format used for illustration in the paper. It has 2 bits for the exponent and 2 bits for the fraction. (d) The bit pattern for a posit representation.

When $E = 2^{|E|} - 1$, the FP bit-strings represent special values. If $F = 0$, then the bit-string represents $\pm\infty$ depending on the value of s and in all other cases, it represents *not-a-number* (NaN).

IEEE-754 specifies a number of default FP types: 16-bit ($\mathbb{F}_{16,5}$ or `half`), 32-bit ($\mathbb{F}_{32,8}$ or `float`), and 64-bit ($\mathbb{F}_{64,11}$ or `double`). Beyond the types specified in the IEEE-754 standard, recent extensions have increased the dynamic range and/or precision. `Bfloat16` [Tagliavini et al. 2018], $\mathbb{F}_{16,8}$, provides increased dynamic range compared to FP's `half` type. Figure 2(b) illustrates the `Bfloat16` format. Recently proposed `TensorFloat32` [NVIDIA 2020], $\mathbb{F}_{19,8}$, increased both the dynamic range and precision compared to the `half` type.

2.2 The Posit Representation

Posit [Gustafson 2017; Gustafson and Yonemoto 2017] is a new representation that provides tapered precision with a fixed number of bits. A posit representation, $\mathbb{P}_{n,es}$, is defined by the total number of bits n and the maximum number of bits for the exponents es. A posit bit-string consists of five components (see Figure 2(d)): a sign bit s, a number of regime bits R, a regime guard bit $\overline{R}$, up to es-bits of the exponent E, and fraction bits F. When the regime bits are not used, they can be re-purposed to represent the fraction, which provides tapered precision.

Value of a posit bit-string. The first bit is a sign bit. If $s = 0$, then the value is positive. If $s = 1$, then the value is negative and the bit-string is decoded after taking the two's complement of the remaining bit-string after the sign bit. Three components R, $\overline{R}$, and E together are used to represent the exponent of the final value. After the sign bit, the next $1 \leq |R| \leq n-1$ bits represent the regime R. Regime bits consist of consecutive 1's (or 0's) and are only terminated if $|R| = n-1$ or by an opposite bit 0 (or 1), which is known as the regime guard bit ($\overline{R}$). The regime bits represent the super exponent. Regime bits contribute $useed^r$ to the value of the number where $useed = 2^{2^{es}}$ and $r = |R| - 1$ if R consists of 1's and $r = -|R|$ if R consists of 0's.

If $2 + |R| < n$, then the next $min\{es, n - 2 - |R|\}$ bits represent the exponent bits. If $|E| < es$, then E is padded with 0's to the right until $|E| = es$. These $|es|$-bits contribute 2^E to the value of the number. Together, the regime and the exponent bits of the posit bit-string contribute $useed^r \times 2^E$ to the value of the number. If there are any remaining bits after the es-exponent bits, they represent the fraction bits F. The fraction bits are interpreted like a normal FP value, except the length of F can vary depending on the number of regime bits. They contribute $1 + \frac{F}{2^{|F|}}$. Finally, the value v represented by a posit bit-string is,

$$v = (-1)^s \times (1 + \frac{F}{2^{|F|}}) \times useed^r \times 2^E = (-1)^s \times (1 + \frac{F}{2^{|F|}}) \times 2^{2^{es} \times r + E}$$

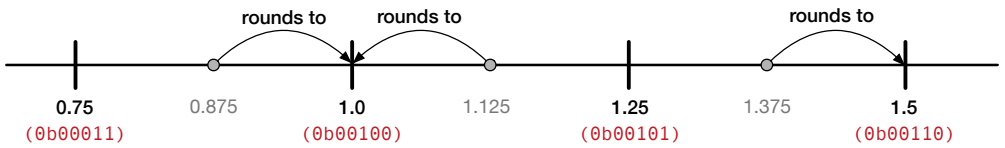


Fig. 3. Illustration of Round to Nearest with ties to Even (RNE) rounding mode with our 5-bit FP representation (FP5). There are two FP5 values (0.75 and 1.0) adjacent to the real number 0.875, but both 0.75 and 1.0 are equidistant from 0.875. In this case, RNE mode specifies that 0.875 should round to 1.0 because the bit representation of 1.0 (`0b00100`) is an even number when interpreted as an integer. Similarly, the real number 1.125 rounds to 1.0 and 1.375 rounds to 1.5.

There are two special cases. A bit-string of all 0's represents 0. A bit-string of 1 followed by all 0*s*'s represents *Not-a-Real* (NaR).

Example. Consider the bit-string `0000011011000000` in the $\mathbb{P}_{16,1}$ configuration. Here, $useed = 2^{2^1} = 2^2$. Also $s = \texttt{0}$, $R = \texttt{0000}$, $\overline{R} = 1$, $E = 1$, and $F = \texttt{011000000}$. Hence, $r = -|R| = -4$. The final exponent resulting from the regime and the exponent bits is $(2^2)^{-4} \times 2^1 = 2^{-7}$. The fraction value is 1.375. The value represented by this posit bit-string is 1.375×2^{-7}.

2.3 Rounding and Numerical Errors

When a real number x cannot be represented in a target representation $\mathbb{T}$, it has to be rounded to a value $v \in \mathbb{T}$. The FP standard defines a number of rounding modes but the default rounding mode is the *round-to-nearest-tie-goes-to-even* (RNE) mode. The posit standard also specifies RNE rounding mode with a minor difference that any non-zero value does not underflow to `0` or overflow to `NaR`. We describe our approach with the RNE mode but it is applicable to other rounding modes.

In the RNE mode, the rounding function $v = RN_{\mathbb{T}}(x)$, rounds $x \in \mathbb{R}$ (Reals) to $v \in \mathbb{T}$, such that x is rounded to the nearest representable value in $\mathbb{T}$, *i.e.* $\forall_{v' \in \mathbb{T}} |x - v| \leq |x - v'|$. In the case of a tie, where $\exists v_1, v_2 \in \mathbb{T}, v_1 \neq v_2$ such that $|x - v_1| = |x - v_2|$ and $\forall_{v' \in \mathbb{T}} |x - v_1| \leq |x - v'|$, then x is rounded to v_1 if the bit-string encoding the value v_1 is an even number when interpreted as an integer and to v_2 otherwise. Figure 3 illustrates the RNE mode with a 5-bit FP representation from Figure 2(c).

The result of primitive operations in FP or any other representation experiences rounding error when it cannot be exactly represented. Modern hardware and libraries produce correctly rounded results for primitive operations. However, this rounding error can get amplified with a series of primitive operations because the intermediate result of each primitive operation must be rounded. As math libraries are also implemented with finite precision, numerical errors in the implementation should also be carefully addressed.

2.4 Background on Approximating Elementary Functions

The state-of-the-art methods to approximate an elementary function $f(x)$ for a target representation ($\mathbb{T}$) involves two steps. First, approximation theory (*e.g.*, minimax methods) is used to develop a function $A_{\mathbb{R}}(x)$ that closely approximates $f(x)$ using real numbers. Second, $A_{\mathbb{R}}(x)$ is implemented in a finite precision representation that has higher precision than $\mathbb{T}$.

Generating $A_{\mathbb{R}}(x)$. Mathematically deriving $A_{\mathbb{R}}(x)$ can be further split into three steps. First, identify inputs that exhibit special behavior (*e.g.*, $\pm\infty$). Second, reduce the input domain to a smaller interval, $[a', b']$, with range reduction techniques and perform any other function transformations. Third, generate a polynomial $P(x)$ that approximates $f(x)$ in the domain $[a', b']$.

There are two types of special cases. The first type includes inputs that produce undefined values or $\pm\infty$ when mathematically evaluating $f(x)$. For example, in the case of $f(x) = 10^x$, $f(x) = \infty$ if

$x = \infty$. The second type consists of interesting inputs for evaluating $RN_{\mathbb{T}}(f(x))$. These cases include a range of inputs that produce interesting outputs such as $RN_{\mathbb{T}}(f(x)) \in \{\pm\infty, 0\}$. For example, while approximating $f(x) = 10^x$ for `Bfloat16` ($\mathbb{B}$), all values $x \in (-\infty, -40.5]$ produce $RN_{\mathbb{B}}(10^x) = 0$, inputs $x \in [-8.46 \cdots \times 10^{-4}, 1.68 \cdots \times 10^{-3}]$ produce $RN_{\mathbb{B}}(10^x) = 1$, and $x \in [38.75, \infty)$ produces $RN_{\mathbb{B}}(10^x) = \infty$. These properties are specific to each $f(x)$ and $\mathbb{T}$.

Range reduction. It is mathematically simpler to approximate $f(x)$ for a small domain of inputs. Hence, most math libraries use range reduction to reduce the entire input domain into a smaller domain before generating the polynomial. Given an input $x \in [a, b]$ where $[a, b] \subseteq \mathbb{T}$, the goal of range reduction is to reduce the input x to $x' \in [a', b']$, where $[a', b'] \subset [a, b]$. We represent this process of range reduction with $x' = RR(x)$. Then, the polynomial P approximates the output y' for the range reduced input (*i.e.*, $y' = P(x')$). The output (y') of the range reduced input (x') has to be compensated to produce the output for the original input (x). The output compensation function, $OC(y', x)$, produces the final result by compensating the range reduced output y' based on the range reduction performed for input x.

For example, consider the function $f(x) = log_2(x)$ where the input domain is defined over $(0, \infty)$. One way to range reduce the original input is to use the mathematical property $log_2(a \times 2^b) = log_2(a) + b$. We decompose the input x as $x = x' \times 2^e$ where $x' \in [1, 2)$ and e is an integer. Approximating $log_2(x)$ is equivalent to approximating $log_2(x' \times 2^e) = log_2(x') + e$. Thus, we can range reduce the original input $x \in (0, \infty)$ into $x' \in [1, 2)$. Then, we approximate $log_2(x')$ using $P(x')$, which needs to only approximate $log_2(x)$ for the input domain $[1, 2)$. To produce the output of $log_2(x)$, we compensate the output of the reduced input by computing $P(x') + e$, where e is dependent on the range reduction of x.

Polynomial approximation $P(x)$. A common method to approximate an elementary function $f(x)$ is with a polynomial function, $P(x)$, which can be implemented with addition, subtraction, and multiplication operations. Typically, $P(x)$ for math libraries is generated using the minimax approximation technique, which aims to minimize the maximum error, or L_∞-norm,

$$||P(x) - f(x)||_\infty = \sup_{x \in [a,b]} |P(x) - f(x)|$$

where sup represents the supremum of a set. The minimax approach is attractive because the resulting $P(x)$ has a bound on the error (*i.e.*, $|P(x) - f(x)|$). The most well-known minimax approximation method is the Remez algorithm [Remes 1934]. Both CR-LIBM [Daramy et al. 2003] and Metalibm [Kupriianova and Lauter 2014] use a modified Remez algorithm to produce polynomial approximations [Brisebarre and Chevillard 2007].

Implementation of $A_{\mathbb{R}}(x)$ with finite precision. Finally, mathematical approximation $A_{\mathbb{R}}(x)$ is implemented in finite precision to approximate $f(x)$. This implementation typically uses a higher precision than the intended target representation. We use $A_{\mathbb{H}}(x)$ to represent that $A_{\mathbb{R}}(x)$ is implemented in a representation with higher precision ($\mathbb{H}$) where $\mathbb{T} \subset \mathbb{H}$. Finally, the result of the implementation $A_{\mathbb{H}}(x)$ is rounded to the target representation $\mathbb{T}$.

2.5 Challenges in Building Correctly Rounded Math Libraries

An approximation of an elementary function $f(x)$ is defined to be a correctly rounded approximation if for all inputs $x_i \in \mathbb{T}$, it produces $RN_{\mathbb{T}}(f(x_i))$. There are two major challenges in creating a correctly rounded approximation. First, $A_{\mathbb{H}}(x)$ incurs error because $P(x)$ is an approximation of $f(x)$. Second, the evaluation of $A_{\mathbb{H}}(x)$ has numerical error because it is implemented in a representation with finite precision (*i.e.*, $\mathbb{H}$). Hence, the rounding of $RN_{\mathbb{T}}(A_{\mathbb{H}}(x))$ can result in a value different from $RN_{\mathbb{T}}(f(x))$, even if $A_{\mathbb{H}}(x)$ is arbitrarily close to $f(x)$ for some $x \in \mathbb{T}$.

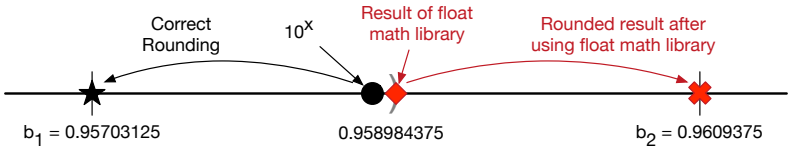


Fig. 4. Using a correctly rounded 32-bit FP math library to approximate 10^x for `Bfloat16` results in wrong results. Horizontal axis represents a real number line. Given an input $x = -0.0181884765625$ that is exactly representable in `Bfloat16`, b_1 and b_2 represent the two closest `Bfloat16` values to the real value of 10^x. The correctly rounded `Bfloat16` value is b_1 (black star). When we use the 32-bit FP library to compute 10^x, it produces the value shown with red diamond, which then rounds to b_2 producing an incorrect result.

As $A_{\mathbb{R}}(x)$ uses a polynomial approximation of $f(x)$, there is an inherent error of $|f(x) - A_{\mathbb{R}}(x)| > 0$. Further, the evaluation of $A_{\mathbb{H}}(x)$ experiences an error of $|A_{\mathbb{H}}(x) - A_{\mathbb{R}}(x)| > 0$. It is not possible to reduce both errors to 0. The error in approximating the polynomial can be reduced by using a polynomial of a higher degree or a piecewise polynomial. The numerical error in the evaluation of $A_{\mathbb{H}}(x)$ can be reduced by increasing the precision of $\mathbb{H}$. Typically, library developers make trade-offs between error and performance of the implementation.

Unfortunately, there is no known general method to analyze and predict the bound on the error for $A_{\mathbb{H}}(x)$ that guarantees $RN_{\mathbb{T}}(A_{\mathbb{H}}(x)) = RN_{\mathbb{T}}(f(x))$ for all x because the error may need to be arbitrarily small. This problem is widely known as *table-maker's dilemma* [Kahan 2004]. It states that there is no general method to predict the amount of precision in $\mathbb{H}$ such that the result is correctly rounded for $\mathbb{T}$.

2.6 Why Not Use Existing Libraries for New Representations?

An alternative to developing math libraries for new representations is to use existing libraries. We can convert the input $x \in \mathbb{T}$ to $x' = RN_{\mathbb{T}'}(x)$, where $\mathbb{T}$ is the representation of interest and $\mathbb{T}'$ is the representation that has a math library available (*e.g.*, `double`). Subsequently, we can use a math library for $\mathbb{T}'$ and round the result back to $\mathbb{T}$. This strategy is appealing if a correctly rounded math library for $\mathbb{T}'$ exists and $\mathbb{T}'$ has significantly more precision bits than $\mathbb{T}$.

However, using a correctly rounded math library designed for $\mathbb{T}'$ to approximate $f(x)$ for $\mathbb{T}$ can produce incorrect results for values in $\mathbb{T}$. We illustrate this behavior by generating an approximation for the function $f(x) = 10^x$ in the `Bfloat16` ($\mathbb{B}$) representation (Figure 4). Let's consider the input $x = -0.0181884765625 \in \mathbb{B}$. The real value of $f(x) \approx 0.95898435797\ldots$ (black circle in Figure 4). This oracle result cannot be exactly represented in `Bfloat16` and must be rounded. There are two `Bfloat16` values adjacent to $f(x)$, $b_1 = 0.95703125$ and $b_2 = 0.9609375$. Since b_1 is closer to $f(x)$, the correctly rounded result is $RN_{\mathbb{B}}(10^x) = b_1$, which is represented by a black star in Figure 4.

If we use the correctly rounded `float` math library to approximate 10^x, we get the value, $y' = 0.958984375$, represented by red diamond in Figure 4. From the perspective of a 32-bit `float`, y' is a correctly rounded result, *i.e.* $y' = RN_{\mathbb{F}_{32,8}}(10^x) = 0.958984375$. Because $y' \notin \mathbb{B}$, we round y' to `Bfloat16` based on the rounding rule, $RN_{\mathbb{B}}(y') = b_2$. Therefore, the `float` math library rounds the result to b_2 but the correctly rounded result is $RN_{\mathbb{B}}(10^x) = b_1$.

Summary. Approximating an elementary function for representation $\mathbb{T}$ using a math library designed for a higher precision representation $\mathbb{T}'$ does not guarantee a correctly rounded result. Further, the math library for $\mathbb{T}'$ probably requires higher accuracy than the one for $\mathbb{T}$. Hence, it uses a higher degree polynomial, which causes it to be slower than the math library tailored for $\mathbb{T}$.

3 HIGH-LEVEL OVERVIEW

We provide a high-level overview of our methodology to generate correctly rounded math libraries. We will illustrate this methodology with an end-to-end example that creates correctly rounded results for $ln(x)$ with FP5 (*i.e.*, a 5-bit FP type shown in Figure 2(c)).

3.1 Our Methodology for Generating Correctly Rounded Elementary Functions

Given an elementary function $f(x)$ and a target representation $\mathbb{T}$, our goal is to synthesize a polynomial that when used with range reduction (*RR*) and output compensation (*OC*) function produces the correctly rounded result for all inputs in $\mathbb{T}$. The evaluation of the polynomial, range reduction, and output compensation are implemented in representation $\mathbb{H}$, which has higher precision than $\mathbb{T}$.

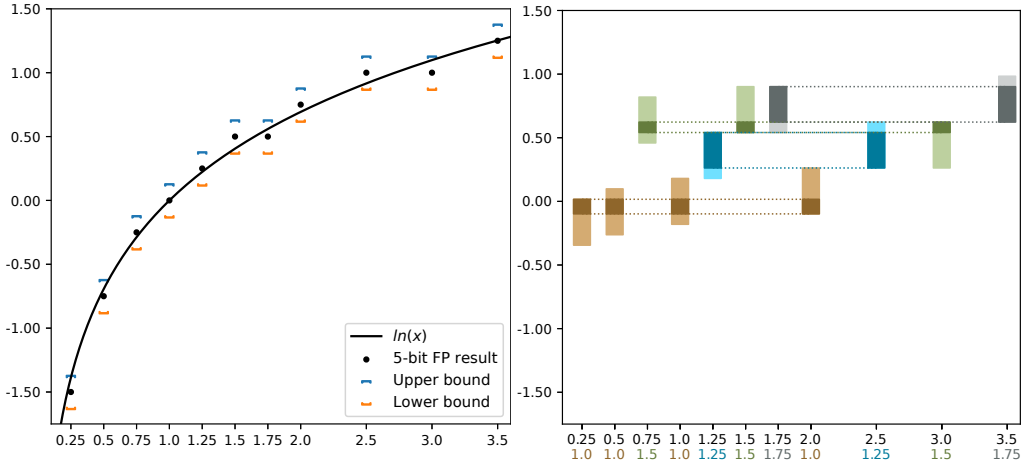
Our methodology for generating correctly rounded elementary functions is shown in Figure 1. Our methodology consists of four steps. First, we use an oracle (*i.e.*, MPFR [Fousse et al. 2007] with a large number of precision bits) to compute the correctly rounded result of the function $f(x)$ for each input $x \in \mathbb{T}$. In this step, a small sample of the entire input space can be used rather than using all inputs for a type with a large input domain.

Second, we identify an interval $[l, h]$ around the correctly rounded result such that any value in $[l, h]$ rounds to the correctly rounded result in $\mathbb{T}$. We call this interval the *rounding interval*. Since the eventual polynomial evaluation happens in $\mathbb{H}$, the rounding intervals are also in the $\mathbb{H}$ representation. The internal computations of the math library evaluated in $\mathbb{H}$ should produce a value in the rounding interval for each input x.

Third, we employ range reduction to transform input x to x'. The generated polynomial will approximate the result for x'. Subsequently, we have to use an appropriate output compensation code to produce the final correctly rounded output for x. Both range reduction and output compensation happen in the $\mathbb{H}$ representation and can experience numerical errors. These numerical errors should not affect the generation of correctly rounded results. Hence, we infer intervals for the reduced domain so that the polynomial evaluation over the reduced input domain produces the correct results for the entire domain. Given x and its rounding interval $[l, h]$, we can compute the reduced input x' with range reduction. The next task before polynomial generation is identifying the reduced rounding interval for $P(x')$ such that when used with output compensation it produces the correctly rounded result. We use the inverse of the output compensation function to identify the reduced interval $[l', h']$. Any value in $[l', h']$ when used with the implementation of output compensation in $\mathbb{H}$ produces the correctly rounded results for the entire domain.

Fourth, we synthesize a polynomial of a degree d using an arbitrary precision linear programming (LP) solver that satisfies the constraints (*i.e.*, $l' \leq P(x') \leq h'$) when given a set of inputs x'. Since the LP solver produces coefficients for the polynomial in arbitrary precision, it is possible that some of the constraints will not be satisfied when evaluated in $\mathbb{H}$. In such cases, we refine the reduced intervals for those inputs whose constraints are violated and repeat the above step. If the LP solver is not able to produce a solution, then the developer of the library has to either increase the degree of the polynomial or reduce the input domain.

If the inputs were sampled in the first step, we check whether the generated polynomial produces the correctly rounded result for all inputs. If it does not, then the input is added to the sample and the entire process is repeated. At the end of this process, the polynomial along with range reduction and output compensation when evaluated in $\mathbb{H}$ produces the correctly rounded outputs for all inputs in $\mathbb{T}$.



(a) (b)

$$
\begin{aligned}
-0.098315\cdots \le P(1.00) \le 0.016294\ldots \\
0.262358\cdots \le P(1.25) \le 0.541010\ldots \\
0.541010\cdots \le P(1.50) \le 0.623031\ldots \\
0.623031\cdots \le P(1.75) \le 0.901684\ldots
\end{aligned}
$$

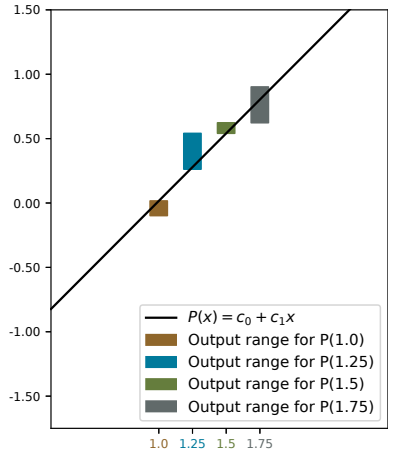
(c)

$$
\begin{bmatrix} -0.09831\ldots \\ 0.26235\ldots \\ 0.54101\ldots \\ 0.62303\ldots \end{bmatrix} \le \begin{bmatrix} 1.0 & 1.0 \\ 1.0 & 1.25 \\ 1.0 & 1.5 \\ 1.0 & 1.75 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \le \begin{bmatrix} 0.01629\ldots \\ 0.54101\ldots \\ 0.62303\ldots \\ 0.90168\ldots \end{bmatrix}
$$

(d)

$$
\begin{aligned}
P(x) &= c_0 + c_1 x \\
c_0 &= -1.033138324333696456136522101587615907 19\ldots \\
c_1 &= 1.049432643111371854516278290248010307550\ldots
\end{aligned}
$$

(e)



(f)

Fig. 5. Our approach for $ln(x)$ with FP5. (a) For each input x in FP5, we accurately compute the correctly rounded result (black circle) and identify intervals around the result so that all values round to it. (b) For each input and corresponding interval computed in (a), we perform range reduction to obtain the reduced input. The number below a value on the x-axis represents the reduced input. The reduced interval to account for rounding errors in output compensation is also shown. Multiple distinct inputs can map to the same reduced input after range reduction (intervals with the same color). In such scenarios, we combine the reduce intervals by computing the common region in the intervals (highlighted in bold for each color with dotted lines). (c) The set of constraints that must be satisfied by the polynomial for the reduced input. (d) LP formulation for the generation of a polynomial of degree one. (e) The coefficients generated by the LP solver for the polynomial. (f) Generated polynomial satisfies the combined intervals.

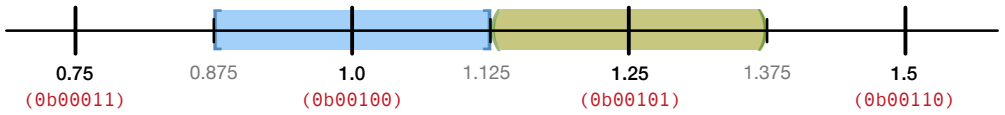


Fig. 6. This figure shows the real number line and a number of adjacent FP5 values, 0.75, 1.0, 1.25, and 1.5. Any real value in the blue interval $[0.875, 1.125]$, rounds to 1.0 in FP5 with RNE rounding mode. Similarly, any value in the green interval $(1.125, 1.375)$ rounds to 1.25 in FP5.

3.2 Illustration of Our Approach with $ln(x)$ for FP5

We provide an end-to-end example of our approach by creating a correctly rounded result of $ln(x)$ for the FP5 representation shown in Figure 2(c) with the RNE rounding mode. The $ln(x)$ function is defined over the input domain $(0, \infty)$. There are 11 values ranging from 0.25 to 3.5 in FP5 within $(0, \infty)$. We show the generation of the polynomial with FP5 for pedagogical reasons. With FP5, it is beneficial to create a pre-computed table of correctly rounded results for the 11 values.

Our strategy is to approximate $ln(x)$ by using $log_2(x)$. Hence, we perform range reduction and output compensation using the properties of logarithm: $ln(x) = \frac{log_2(x)}{log_2(e)}$ and $log_2(x \times y^z) = log_2(x) + zlog_2(y)$. We decompose the input x as $x = x' \times 2^n$ where x' is the fractional value represented by the mantissa, *i.e.* $x' \in [1, 2)$, and n is the exponent of the value. We use $ln(x) = \frac{log_2(x')+m}{log_2(e)}$ for our range reduction. We construct the range reduction function $RR(x)$ and the output compensation function $OC(y', x)$ as follows,

$$RR(x) = fr(x), \qquad OC(y', x) = \frac{y' + exp(x)}{log_2(e)}$$

where $fr(x)$ returns the fractional part of x (*i.e.*, $x' \in [1, 2)$) and $exp(x)$ returns the exponent of x (*i.e.*, n). Then, our polynomial approximation $P(x')$ should approximate the function $log_2(x)$ for the reduced input domain $x' \in [1, 2)$. The various steps of our approach are illustrated in Figure 5.

Step 1: Identifying the correctly rounded result. There are a total of 11 FP5 values in the input domain of $ln(x)$, $(0, \infty)$. These values are shown on the x-axis in Figure 5(a). Other values are special cases. They are captured by the precondition for this function (*i.e.*, $x = 0$ or $x = \infty$). Our goal is to generate the correctly rounded results for these 11 FP5 values. For each of these 11 inputs x, we use an oracle (*i.e.*, MPFR math library) to compute y, which is the correctly rounded value of $ln(x)$. Figure 5(a) shows the correctly rounded result for each input as a black dot.

Step 2: Identifying the rounding interval $[l, h]$. The range reduction, output compensation, and polynomial evaluation are performed with the `double` type. The `double` result of the evaluation is rounded to FP5 to produce the final result. The next step is to find a rounding interval $[l, h]$ in the `double` type for each output. Figure 5(a) shows the rounding interval for each FP5 output using the blue (upper bound) and orange (lower bound) bracket.

Let us suppose that we want to compute the rounding interval for $y = 1.0$, which is the correctly rounded result of $ln(2.5)$. To identify the lower bound l of the rounding interval for $y = 1.0$, we first identify the preceding FP5 value, which is 0.75. Then we find a value v between 0.75 and 1.0 such that values greater than or equal to v rounds to 1.0. In our case, $v = 0.875$, which is the lower bound. Similarly, to identify the upper bound h, we identify the FP5 value succeeding 1.0, which is 1.25. We find a value v such that any value less than or equal to v rounds to 1.0. In our case, the upper bound is $h = 1.125$. Hence, the rounding interval for $y = 1.0$ is $[0.875, 1.125]$. Figure 6 shows the intervals for a small subset of FP5.

Step 3-a: Computing the reduced input x' and the reduced interval $[l', h']$. We perform range reduction and generate a polynomial that computes $log_2(x)$ for all reduced inputs in $[1, 2)$. The next step is to identify the reduced input and the rounding interval for the reduced input such that it accounts for any numerical error in output compensation. Figure 5(b) shows the reduced input (number below the value on the x-axis) and the reduced interval for each input.

To identify the reduced rounding interval, we use the inverse of the output compensation function, which exists if OC is continuous and bijective over real numbers. For example, for the input $x = 3.5 = 1.75 \times 2^1$, the output compensation function is,

$$OC(y', 3.5) = \frac{y' + 1}{log_2(e)}$$

The inverse is

$$OC^{-1}(y, 3.5) = ylog_2(e) - 1$$

We use the inverse of the output compensation function to compute the candidate reduced interval $[l', h']$ by computing $l' = OC^{-1}(l, x)$ and $h' = OC^{-1}(h, x)$. Then, we verify that the output compensation result of l' (*i.e.*, $OC(l', x)$) and h' (*i.e.*, $OC(h', x)$), when evaluated in `double` lies in $[l, h]$. If it does not, then we iteratively refine the reduced interval by restricting $[l', h']$ to a smaller interval until both $OC(l', x)$ and $OC(h', x)$ evaluated in `double` results lie in $[l, h]$. The vertical bars in Figure 5(b) show the reduced input for each x and its corresponding reduced rounding interval.

Step 3-b: Combining the reduced intervals. Multiple inputs from the original input domain can map to the same reduced input after range reduction. In our example, both $x_1 = 1.25$ and $x_2 = 2.5$ reduce to $x' = 1.25$. However, the reduced intervals that we compute for x_1 and x_2 are $[l'_1, h'_1]$ and $[l'_2, h'_2]$, respectively. They are not exactly the same. In Figure 5(b), the reduced intervals corresponding to the original inputs that map to the same reduced input are colored with the same color. The reduced intervals for $x_1 = 1.25$ and $x_2 = 2.5$ are colored in blue.

The reduced interval for x_1 indicates that $P(1.25)$ must produce a value in $[l'_1, h'_1]$ such that the final result, after evaluating the output compensation function in `double`, is the correctly rounded value of $ln(1.25)$. The reduced interval for x_2 indicates that $P(1.25)$ must produce a value in $[l'_2, h'_2]$ such that the final result is the correct value of $ln(2.5)$. To produce the correctly rounded result for both inputs x_1 and x_2, $P(1.25)$ must produce a value that is in both $[l'_1, h'_1]$ and $[l'_2, h'_2]$. Thus, we combine all reduced intervals that correspond to the same reduced input by computing the common interval. Figure 5(b) shows the common interval for a given reduced input using a darker shade. At the end of this step, we are left with one combined interval for each reduced input.

Step 4: Generating the Polynomial for the reduced input. The combined intervals specify the constraints on the output of the polynomial for each reduced input, which when used with output compensation in `double` results in a correctly rounded result for the entire domain. Figure 5(c) shows the constraints for $P(x')$ for each reduced input.

To synthesize a polynomial $P(x')$ of a particular degree (the degree is 1 in this example), we encode the problem as a linear programming (LP) problem that solves for the coefficients of $P(x')$. We look for a polynomial that satisfies constraints for each reduced input (Figure 5(d)). We use an LP solver to solve for the coefficients and find $P(x')$ with the coefficients in Figure 5(e). The generated polynomial $P(x')$ satisfies all the linear constraints as shown in Figure 5(f). Finally, we also verify that the generated polynomial when used with range reduction and output compensation produces the correctly rounded results for all inputs in the original domain.

4 OUR METHODOLOGY FOR GENERATING CORRECTLY ROUNDED LIBRARIES

Our goal is to create approximations for an elementary function $f(x)$ that produces correctly rounded results for all inputs in the target representation ($\mathbb{T}$).

Definition 4.1. A function that approximates an elementary function $f(x)$ is a correctly rounded function for the target representation $\mathbb{T}$ if it produces $y = RN_{\mathbb{T}}(f(x))$ for all $x \in \mathbb{T}$.

Intuitively, the result produced by the approximation should be same as the result obtained when $f(x)$ is evaluated with infinite precision and then rounded to the target representation. It may be beneficial to develop precomputed tables with correctly rounded results of elementary functions for small data types (*e.g.*, FP5). However, it is infeasible (due to memory overheads) to store such tables for every elementary function even with modestly sized data types.

We propose a methodology that produces polynomial approximation and stores a few coefficients for evaluating the polynomial. There are three main challenges in generating a correctly rounded result with polynomial approximations. First, we have to generate polynomial approximations that produce the correct result and are efficient to evaluate. Second, the polynomial approximation should consider rounding errors with range reduction and output compensation that are implemented in some finite precision representation. Third, the polynomial evaluation also is implemented with finite precision and can experience numerical errors.

We will use $A_{\mathbb{H}}(x)$ to represent the approximation of the elementary function $f(x)$ produced with our methodology while using a representation $\mathbb{H}$ to perform polynomial evaluation, range reduction, and output compensation. The result of $A_{\mathbb{H}}(x)$ is rounded to $\mathbb{T}$ to produce the final result. Hence, $A_{\mathbb{H}}(x)$ is composed of three functions: $A_{\mathbb{H}}(x) = OC_{\mathbb{H}}(P_{\mathbb{H}}(RR_{\mathbb{H}}(x)), x)$ where $y' = P_{\mathbb{H}}(x')$ is the polynomial approximation function, $x' = RR_{\mathbb{H}}(x)$ is the range reduction function, and $OC_{\mathbb{H}}(y', x)$ is the output compensation function. All three functions, $RR_{\mathbb{H}}(x)$, $P_{\mathbb{H}}(x')$, and $OC_{\mathbb{H}}(y', x)$ are evaluated in $\mathbb{H}$. Given $RR_{\mathbb{H}}(x)$ and $OC_{\mathbb{H}}(y', x)$ for a particular elementary function $f(x)$, the task of creating an approximation that produces correctly rounded results involves synthesizing a polynomial $P_{\mathbb{H}}(x)$ such that final result generated by $A_{\mathbb{H}}(x)$ is a correctly rounded result for all inputs x.

Our methodology for identifying $A_{\mathbb{H}}(x)$ that produces correctly rounded outputs is pictorially shown in Figure 1. In our approach, we assume the existence of an oracle, which generates the correct real result, to generate the polynomial approximation for a target representation $\mathbb{T}$. We can use existing MPFR libraries with large precision as an oracle. Typically, the polynomial approximation is closely tied to techniques used for range reduction and the resulting output compensation. We also require that the output compensation function (OC) is invertible (*i.e.*, continuous and bijective). The degree of the polynomial is an input provided by the developer of the math library. The top-level algorithm shown in Figure 7 identifies a polynomial approximation of degree d. If it is unable to find one, the developer of the math library should explore one with a higher degree.

Our approach has four main steps. First, we compute $y \in \mathbb{T}$, the correctly rounded result of $f(x)$, *i.e.* $y = RN_{\mathbb{T}}(f(x))$ for each input x (or a sample of the inputs for a large data type) using our oracle. Then, we identify the rounding interval $I = [l, h] \subseteq \mathbb{H}$ where all values in the interval round to y. The pair (x, I) specifies that $A_{\mathbb{H}}(x)$ must produce a value in I such that $A_{\mathbb{H}}(x)$ rounds to y. The function `CalcRndIntervals` in Figure 7 returns a list L that contains a pair (x, I) for all inputs x.

Second, we compute the reduced input x' using range reduction and a reduced interval $I' = [l', h']$ for each pair $(x, I) \in L$. The reduced interval $I' = [l', h']$ ensures that any value in I' when used with output compensation code results in a value in I. This pair (x', I') specifies the constraints for the output of the polynomial approximation $P_{\mathbb{H}}(x')$ so $A_{\mathbb{H}}(x)$ rounds to the correctly rounded result. The function `CalcRedIntervals` returns a list L' with such reduced constraints for all inputs x.

Third, multiple inputs from the original input domain will map to the same input in the reduced domain after range reduction. Hence, there will be multiple reduced constraints for each reduced input x'. The polynomial approximation, $P_{\mathbb{H}}(x')$, must produce a value that satisfies all the reduced constraints to ensure that $A_{\mathbb{H}}(x)$ produces the correct value for all inputs when rounded. Thus,

```
1  Function CorrectlyRoundedPoly(f, 𝕋, ℍ, X, RR_ℍ, OC_ℍ, d):
2      L ← CalcRndIntervals(f, 𝕋, ℍ, X)
3      if L = ∅ then return (false, DNE)
4      L' ← CalcRedIntervals(L, ℍ, RR_ℍ, OC_ℍ)
5      if L' = ∅ then return (false, DNE)
6      Λ ← CombineRedIntervals(L')
7      if Λ = ∅ then return (false, DNE)
8      S, P_ℍ ← GeneratePoly(Λ, d)
9      if S = true then return (true, P_ℍ)
10     else return (false, DNE)
```

Input Description:
f: *The oracle that computes the result of* $f(x)$ *in arbitrary precision.*
$\mathbb{T}$: *Target representation of math library.*
$\mathbb{H}$: *Higher precision representation.*
X: *Input domain of* $A_{\mathbb{H}}(x)$.
$RR_{\mathbb{H}}$: *The range reduction function.*
$OC_{\mathbb{H}}$: *The output compensation function.*
d: The degree of polynomial to generate.

Fig. 7. Our approach to generate a polynomial approximation $P_{\mathbb{H}}(x)$ that produces the correctly rounded result for all inputs. On successfully finding a polynomial, it returns (true, $P_{\mathbb{H}}$). Otherwise, it returns (false, DNE) where DNE means that the polynomial Does-Not-Exist. Functions, `CalcIntervals`, `CalcRedIntervals`, `CombineRedIntervals`, and `GeneratePoly` are shown in Figure 8, Figure 9, and Figure 10, respectively.

```
1  Function CalcRndIntervals(f, 𝕋, ℍ, X):
2      L ← ∅
3      foreach x ∈ X do
4          y ← RN_𝕋(f(x))
5          I ← GetRndInterval(y, 𝕋, ℍ)
6          if I = ∅ then return ∅
7          L ← L ∪ {(x, I)}
8      end
9      return L

10 Function GetRndInterval(y, 𝕋, ℍ):
11     t_l ← GetPrecVal(y, 𝕋)
12     l ← min{v ∈ ℍ | v ∈ [t_l, y] and RN_𝕋(v) = y}
13     t_u ← GetSuccVal(y, 𝕋)
14     h ← max{v ∈ ℍ | v ∈ [y, t_u] and RN_𝕋(v) = y}
15     return [l, h]
```

Fig. 8. For each input $x \in X$, `CalcRndIntervals` identifies the interval $I = [l, h]$ where all values in I round to the correctly rounded result. The `GetRndInterval` function takes the correctly rounded result y and returns the interval $I \subseteq \mathbb{H}$ where all values in I round to y. `GetPrecValue`(y, $\mathbb{T}$) returns the value preceeding y in $\mathbb{T}$. `GetSuccValue`(y, $\mathbb{T}$) returns the value succeeding y in $\mathbb{T}$.

we combine all reduced intervals for each unique reduced input x' and produce the pair (x', Ψ) where Ψ represents the combined interval. Function `CombineRedIntervals` in Figure 7 returns a list Λ containing the constraint pair (x', Ψ) for each unique reduced input x'. Finally, we generate a polynomial of degree d using linear programming so that all constraints $(x', \Psi) \in \Lambda$ are satisfied. Next, we describe these steps in detail.

4.1 Calculating the Rounding Interval

The first step in our approach is to identify the values that $A_{\mathbb{H}}(x)$ must produce so that the rounded value of $A_{\mathbb{H}}(x)$ is equal to the correctly rounded result of $y = f(x)$, *i.e.* $RN_{\mathbb{T}}(A_{\mathbb{H}}(x)) = RN_{\mathbb{T}}(y)$, for each input $x \in X$. *Our key insight is that it is not necessary to produce the exact value of* y *to produce a correctly rounded result.* It is sufficient to produce any value in $\mathbb{H}$ that rounds to the correct result. For a given rounding mode and an input, we are looking for an interval $I = [l, h]$ around the oracle result that produces the correctly rounded result. We call this the rounding interval.

Given an elementary function $f(x)$ and an input $x \in X$, we define an interval I that is representable in $\mathbb{H}$ such that $RN_{\mathbb{T}}(v) = RN_{\mathbb{T}}(f(x))$ for all $v \in I$. If $A_{\mathbb{H}}(x) \in I$, then rounding the result of $A_{\mathbb{H}}(x)$ to $\mathbb{T}$ produces the correctly rounded result (*i.e.*, $RN_{\mathbb{T}}(A_{\mathbb{H}}(x)) = RN_{\mathbb{T}}(f(x))$). For each input x, if $A_{\mathbb{H}}(x)$ can produce a value that lies within its corresponding rounding interval, then it will

produce a correctly rounded result. Thus, the pair (x, I) for each input x defines constraints on the output of $A_{\mathbb{H}}(x)$ such that $RN_{\mathbb{T}}(A_{\mathbb{H}}(x))$ is a correctly rounded result.

Figure 8 presents our algorithm to compute constraints (x, I). For each input x in our input domain X, we compute the correctly rounded result of $f(x)$ using an oracle and produce y. Next, we compute the rounding interval of y where all values in the interval round to y. The rounding interval can be computed as follows. First, we identify t_l, the preceding value of y in $\mathbb{T}$ (line 11 in Figure 8). Then we find the minimum value $l \in \mathbb{H}$ between t_l and y where l rounds to y (line 12 in Figure 8). Similarly for the upper bound, we identify t_u, the succeeding value of y in $\mathbb{T}$ (line 13 in Figure 8), and find the maximum value $h \in \mathbb{H}$ between y and t_u where h rounds to y (line 14 in Figure 8). Then, $[l, h]$ is the rounding interval of y and all values in $[l, h]$ round to y. Thus, the pair $(x, [l, h])$ specifies a constraint on the output of $A_{\mathbb{H}}(x)$ to produce the correctly rounded result for input x. We generate such constraints for each input in the entire domain (or for a sample of inputs) and produce a list of such constraints (lines 7-9 in Figure 8).

4.2 Calculating the Reduced Input and Reduced Interval

After the previous step, we have a list of constraints, (x, I), that need to be satisfied by our approximation $A_{\mathbb{H}}(x)$ to produce correctly rounded outputs. If we do not perform any range reduction, then we can generate a polynomial that satisfies these constraints. However, it is necessary to perform range reduction (RR) in practice to reduce the complexity of the polynomial and to improve performance. Range reduction is accompanied by output compensation (OC) to produce the final output. Hence, $A_{\mathbb{H}}(x) = OC_{\mathbb{H}}(P_{\mathbb{H}}(RR_{\mathbb{H}}(x)), x)$. Our goal is to synthesize a polynomial $P_{\mathbb{H}}(x')$ that operates on the range reduced input x' and $A_{\mathbb{H}}(x) = OC_{\mathbb{H}}(P_{\mathbb{H}}(RR_{\mathbb{H}}(x)), x)$ produces a value in I for each input x, which rounds to the correct output.

To synthesize this polynomial, we have to identify the reduced input and the reduced interval for an input x such that $A_{\mathbb{H}}(x)$ produces a value in the rounding interval I corresponding to x. The reduced input is available by applying range reduction $x' = RR(x)$. Next, we need to compute the reduced interval corresponding to x'. The output of the polynomial on the reduced input will be fed to the output compensation function to compute the output for the original input. For the reduced input x' corresponding to the original input x, $y' = P_{\mathbb{H}}(x')$, $A_{\mathbb{H}}(x) = OC_{\mathbb{H}}(y', x)$, and $A_{\mathbb{H}}(x)$ must be within the interval I for input x to produce a correct output. Hence, our high-level strategy is to use the inverse of the output compensation function to compute the reduced interval, which is feasible when the output compensation function is continuous and bijective. In our experience, all commonly used output compensation functions are continuous and bijective.

However, the output compensation function is evaluated in $\mathbb{H}$, which necessitates us to take any numerical error in output compensation with $\mathbb{H}$ into account. Figure 9 describes our algorithm to compute reduced constraint (x', I') for each $(x, I) \in L$ when the output compensation is performed in $\mathbb{H}$.

To compute the reduced interval I' for each constraint pair $(x, [l, h]) \in L$, we evaluate the values $v_1 = OC^{-1}_{\mathbb{H}}(l, x)$ and $v_2 = OC^{-1}_{\mathbb{H}}(h, x)$ and create an interval $[\alpha, \beta] = [v_1, v_2]$ if $OC_{\mathbb{R}}(y', x)$ is an increasing function (lines 5-6 in Figure 9) or $[v_2, v_1]$ if $OC_{\mathbb{R}}(y', x)$ is a decreasing function (line 7 in Figure 9). The interval $[\alpha, \beta]$ is a candidate for I'. Then, we verify that the output compensated value of α is in $[l, h]$ (*i.e.*, I). If it is not, we replace α with the succeeding value in $\mathbb{H}$ and repeat the process until $OC_{\mathbb{H}}(\alpha, x)$ is in I (lines 8-11 in Figure 9). Similarly, we verify that the output compensated value of β is in $[l, h]$ and repeatedly replace β with the preceding value in $\mathbb{H}$ if it is not (lines 12-15 in Figure 9). If $\alpha > \beta$ at any point during this process, then it indicates that there is no polynomial $P(x')$ that can produce the correct result for all inputs. As there are only finitely many values between $[\alpha, \beta]$ in $\mathbb{H}$, this process terminates. In the case when our algorithm is not able to

```
1  Function CalcRedIntervals(L, 𝕀, RR_𝕀, OC_𝕀):
2      L' ← ∅
3      foreach (x, [l, h]) ∈ L do
4          x' ← RR_𝕀(x)
5          if OC_𝕀 is an increasing function then
6              [α, β] ← [OC_𝕀^{-1}(l, x), OC_𝕀^{-1}(h, x)]
7          else [α, β] ← [OC_𝕀^{-1}(h, x), OC_𝕀^{-1}(l, x)]
8          while OC_𝕀(α, x) ∉ [l, h] do
9              α ← GetSuccVal(α, 𝕀)
10             if α > β then return ∅
11         end
12         while OC_𝕀(β, x) ∉ [l, h] do
13             β ← GetPrecVal(β, 𝕀)
14             if α > β then return ∅
15         end
16         L' ← L' ∪ {(x', [α, β])}
17     end
18     return L'

19 Function CombineRedIntervals(L'):
20     X̂ ← {x' | (x', I') ∈ L'}
21     Λ ← ∅
22     foreach x̂ ∈ X̂ do
23         Ω ← {I' | (x̂, I') ∈ L'}
24         Ψ ← ⋂_{I'∈Ω} I'
25         if Ψ = ∅ then return ∅
26         Λ ← Λ ∪ {(x̂, Ψ)}
27     end
28     return Λ
```

Fig. 9. `CalcRedIntervals` computes the reduced input x' and the reduced interval I' for each constraint pair (x, I) in L. The reduced constraint pair (x', I') specifies the bound on the output of $P_{\mathbb{H}}(x')$ such that it produces the correct value for the input x. `CombineRedIntervals` combines any reduced constraints with the same reduced input, *i.e.* (x'_1, I'_1) and (x'_2, I'_2) where $x'_1 = x'_2$ into a single combined constraint (x_1, Ψ) by computing the common interval range in I'_1 and I'_2.

find a polynomial, the user can provide either a different range reduction/output compensation function or increase the precision to be higher than $\mathbb{H}$.

If the resulting interval $[\alpha, \beta] \neq \emptyset$, then $I' = [\alpha, \beta]$ is our reduced interval. The reduced constraint pair, $(x', [\alpha, \beta])$ created for each $(x, I) \in L$ specifies the constraint on the output of $P_{\mathbb{H}}(x')$ such that $A_{\mathbb{H}}(x) \in I$. Finally, we create a list L' containing such reduced constraints.

4.3 Combining the Reduced Constraints

Each reduced constraint $(x'_i, I'_i) \in L'$ corresponds to a constraint $(x_i, I_i) \in L$. It specifies the bound on the output of $P_{\mathbb{H}}(x'_i)$ (*i.e.*, $P_{\mathbb{H}}(x'_i) \in I'_i$ should be satisfied), which ensures $A_{\mathbb{H}}(x_i)$ produces a value in I_i. Range reduction reduces the original input x_i in the entire input domain of $f(x)$ to a reduced input x'_i in the reduced domain. Hence, multiple inputs in the entire input domain can be range reduced to the same reduced input. More specifically, there can exist multiple constraints $(x_1, I_1), (x_2, I_2), \cdots \in L$ such that $RR_{\mathbb{H}}(x_1) = RR_{\mathbb{H}}(x_2) = \hat{x}$. Consequently, L' can contain reduced constraints $(\hat{x}, I'_1), (\hat{x}, I'_2) \cdots \in L'$. The polynomial $P_{\mathbb{H}}(\hat{x})$ must produce a value in I'_1 to guarantee that $A_{\mathbb{H}}(x_1) \in I_1$. It must also be within I'_2 to guarantee $A_{\mathbb{H}}(x_2) \in I_2$. Hence, for each unique reduced input $\hat{x}$, $P_{\mathbb{H}}(\hat{x})$ must satisfy all reduced constraints corresponding to $\hat{x}$, *i.e.* $P_{\mathbb{H}}(\hat{x}) \in I'_1 \cap I2'$.

The function `CombineRedIntervals` in Figure 9 combines all reduced constraints with the same reduced input by identifying the common interval (Ψ in line 24 in Figure 9). If such a common interval does not exist, then it is infeasible to find a single polynomial $P_{\mathbb{H}}(x')$ that produces correct outputs for all inputs before range reduction. Otherwise, we create a pair $(\hat{x}, \Psi)$ for each unique reduced interval $\hat{x}$ and produce a list of constraints Λ (line 26 in Figure 9).

```
1  Function GeneratePoly(Λ, ℍ d):
2      ϒ ← Λ
3      while true do
4          C ← LPSolve(ϒ, d)
5          if C = ∅ then return (false, DNE)
6          P_ℍ ← CreateP(C, d, ℍ)
7          ϒ ← Verify(P_ℍ, Λ, ϒ, ℍ)
8          if ϒ = ∅ then return (true, P_ℍ)
9      end
10 Function Verify(P_ℍ, Λ, ϒ, ℍ):
11     Z ← {(x', Ψ, ψ) | (x', Ψ) ∈ Λ, (x', ψ) ∈ ϒ}
12     foreach (x', [l', h'], [σ, μ]) ∈ Z do
13         if P_ℍ(x') < l' then
14             ϒ ← ϒ − {(x', [σ, μ])}
15             σ' ← GetSuccVal(σ, ℍ)
16             return ϒ ∪ {(x', [σ', μ])}
17         else if P_ℍ(x') > h' then
18             ϒ ← ϒ − {(x', [σ, μ])}
19             μ' ← GetPrecVal(μ, ℍ)
20             return ϒ ∪ {(x', [σ, μ'])}
21         end
22     end
23     return ∅
```

Fig. 10. The function `GeneratePoly` generates a polynomial $P_{\mathbb{H}}(x')$ of degree d that satisfies all constraints in Λ when evaluated in $\mathbb{H}$. If it cannot generate such a polynomial, then it returns *false*. The function `LPSolve` solves for the real number coefficients of a polynomial $P_{\mathbb{R}}(x)$ using an LP solver where $P_{\mathbb{R}}(x)$ satisfies all constraints in Λ when evaluated in real number. `CreateP` creates $P_{\mathbb{H}}(x)$ that evaluates the polynomial $P_{\mathbb{R}}(x)$ in $\mathbb{H}$. The `Verify` function checks whether the generated polynomial $P_{\mathbb{H}}(x)$ satisfies all constraints in Λ when evaluated in $\mathbb{H}$ and refines the constraints to a smaller interval for each constraint that $P_{\mathbb{H}}(x)$ does not satisfy.

4.4 Generating the Polynomial Using Linear Programming

Each reduced constraint $(x', [l', h']) \in \Lambda$ requires that $P_{\mathbb{H}}(x')$ satisfy the following condition: $l' \leq P_{\mathbb{H}}(x') \leq h'$. This constraint ensures that when $P_{\mathbb{H}}(x')$ is combined with range reduction and output compensation, it produces the correctly rounded result for all inputs. When we are trying to generate a polynomial of degree d, we can express each of the above constraints in the form:

$$l' \leq c_0 + c_1 x' + c_2 (x')^2 + \ldots + c_d (x')^d \leq h'$$

The goal is to find coefficients for the polynomial evaluated in $\mathbb{H}$. Here, x', l' and h' are constants from perspective of finding the coefficients. We can express all constraints $(x'_i, [l'_i, h'_i]) \in \Lambda$ in a single system of linear inequalities as shown below, which can be solved using a linear programming (LP) solver.

$$\begin{bmatrix} l'_1 \\ l'_2 \\ \vdots \\ l'_{|\Lambda|} \end{bmatrix} \leq \begin{bmatrix} 1 & x'_1 & \ldots & (x'_1)^d \\ 1 & x'_2 & \ldots & (x'_2)^d \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x'_{|\Lambda|} & \ldots & (x'_{|\Lambda|})^d \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_d \end{bmatrix} \leq \begin{bmatrix} h'_1 \\ h'_2 \\ \vdots \\ h'_{|\Lambda|} \end{bmatrix}$$

Given a system of inequalities, the LP solver finds a solution for the coefficients with real numbers. The polynomial when evaluated in real (*i.e.* $P_{\mathbb{R}}(x')$) satisfies all constraints in Λ. However, numerical errors in polynomial evaluation in $\mathbb{H}$ can cause the result to not satisfy Λ. We propose a *search-and-refine* approach to address this problem. We use the LP solver to solve for the coefficients of $P_{\mathbb{R}}(x')$ that satisfy Λ and then check if $P_{\mathbb{H}}(x')$ that evaluates $P_{\mathbb{R}}(x')$ in $\mathbb{H}$ satisfies the constraints in Λ. If $P_{\mathbb{H}}(x')$ does not satisfy a constraint $(x', [l', h']) \in \Lambda$, then we refine the reduced interval $[l', h']$ to a smaller interval. Subsequently, we use the LP solver to generate the coefficients of $P_{\mathbb{R}}(x')$ for the refined constraints. This process is repeated until either $P_{\mathbb{H}}(x')$ satisfies all reduced constraints in Λ or the LP solver determines that there is no polynomial that satisfies all the constraints.

Figure 10 provides the algorithm used for generating the coefficients of the polynomial using the LP solver. Υ tracks the refined constraints for $P_{\mathbb{H}}(x')$ during our search-and-refine process. Initially, Υ is set to Λ (line 2 in Figure 10). Here, Υ is used to generate the polynomial and Λ is used to to verify that the generated polynomial satisfies all constraints. If the generated polynomial does not satisfy Λ, we restrict the intervals in Υ.

We use an LP solver to solve for the coefficients of the $P_{\mathbb{R}}(x')$ that satisfy all constraints in Υ (line 4 in Figure 10). If the LP solver cannot find the coefficients, our algorithm concludes that it is not possible to generate a polynomial and terminates (line 5 in Figure 10). Otherwise, we create $P_{\mathbb{H}}(x')$ that evaluates $P_{\mathbb{R}}(x')$ in $\mathbb{H}$ by rounding all coefficients to $\mathbb{H}$ and perform all operations in $\mathbb{H}$ (line 6 in Figure 10). The resulting $P_{\mathbb{H}}(x')$ is a candidate for the correct polynomial for $A_{\mathbb{H}}(x)$.

Next, we verify that $P_{\mathbb{H}}(x')$ satisfies all constraints in Λ (line 7 in Figure 10). If $P_{\mathbb{H}}(x')$ satisfies all constraints in Λ, then our algorithm returns the polynomial. If there is a constraint $(x', [l', h']) \in \Lambda$ that is not satisfied by $P_{\mathbb{H}}(x')$, then we further restrict the interval $(x', [\sigma, \mu])$ in Υ corresponding to the reduced input x'. If $P_{\mathbb{H}}(x')$ is smaller than the lower bound of the interval constraint in Λ (*i.e.* l'), then we restrict the lower bound of the interval constraint σ in Υ to the value succeeding σ in $\mathbb{H}$ (lines 13-16 in Figure 10). This forces the next coefficients for $P_{\mathbb{R}}(x')$ that we generate using the LP solver to produce a value larger than l'. Likewise, if $P_{\mathbb{H}}(x')$ produces a value larger than the upper bound of the interval constraint in Λ (*i.e.* h'), then we restrict the upper bound of the interval constraint μ in Υ to the value preceding μ in $\mathbb{H}$ (lines 17-20 in Figure 10).

We repeat this process of generating a new candidate polynomial with the refined constraints Υ until it satisfies all constraints in Λ or the LP solver determines that it is infeasible. If a constraint $(x', [\sigma, \mu]) \in \Upsilon$ is restricted to the point where $\sigma > \mu$ (or $[\sigma, \mu] = \emptyset$), then the LP solver will determine that it is infeasible to generate the polynomial. When we are successful in generating a polynomial, then $P_{\mathbb{H}}(x)$ used in tandem with range reduction and the output compensation in $\mathbb{H}$ is checked to ascertain that it produces the correctly rounded results for all inputs.

5 EXPERIMENTAL EVALUATION

This section describes our prototype for generating correctly rounded elementary functions and the math library that we developed for `Bfloat16`, posit, and `float` data types. We present case studies for approximating elementary functions 10^x, $ln(x)$, $log_2(x)$, and $log_{10}(x)$ with our approach for various types. We also evaluate the performance of our correctly rounded elementary functions with state-of-the-art approximations.

5.1 RLibm Prototype and Experimental Setup

Prototype. We use RLibm to refer to our prototype for generating correctly rounded elementary functions and the resulting math libraries generated from it. RLibm supports `Bfloat16`, `Posit16` (16-bit posit type in the Posit standard [Gustafson 2017]), and the 32-bit `float` type in the FP representation. The user can provide custom range reduction and output compensation functions. The prototype uses the MPFR library [Fousse et al. 2007] with 2,000 precision bits as the oracle to compute the real result of $f(x)$ and rounds it to the target representation. Although there is no bound on the precision to compute the oracle result (*i.e.*, Table-maker's dilemma), prior work has shown around 160 precision bits in the worst case is empirically sufficient for the `double` representation [Lefèvre and Muller 2001]. Hence, we use 2,000 precision bits with the MPFR library to compute the oracle result. The prototype uses SoPlex [Gleixner et al. 2015, 2012], an exact rational LP solver as the arbitrary precision LP solver for polynomial generation from constraints.

RLibm's math library contains correctly rounded elementary functions for multiple data types. It contains twelve functions for `Bfloat16` and eleven functions for `Posit16`. The library produces

the correctly rounded result for all inputs. To show that our approach can be used with large data types, RLIBM also includes a correctly rounded $log_2(x)$ for the 32-bit `float` type.

RLIBM performs range reduction and output compensation using the `double` type. We use state-of-the-art range reduction techniques for various elementary functions. Additionally, we split the reduced domain into multiple disjoint smaller domains using the properties of specific elementary functions to generate efficient polynomials. We evaluate all polynomials using the Horner's method, *i.e.* $P(x) = c_0 + x(c_1 + x(c_2 + \ldots))$ [Borwein and Erdelyi 1995], which reduces the number of operations in polynomial evaluation. Our technical report provides details on range reduction, output compensation, and the polynomial generated and the coefficients for each function for each data type [Lim et al. 2020a].

The entire RLIBM prototype is written in C++. RLIBM is open-source [Lim and Nagarakatte 2020a,b]. Although we have not optimized RLIBM for a specific target, it already has better performance than state-of-the-art approaches.

Experimental setup. We describe our experimental setup to check the correctness and performance of RLIBM. There is no math library specifically designed for `Bfloat16` available. To compare the performance of our `Bfloat16` elementary functions, we convert the `Bfloat16` input to a `float` or a `double`, use `glibc`'s (and `Intel`'s) `float` or `double` math library function, and then convert the result back to `Bfloat16`. We use SoftPosit-Math library [Leong 2019] to compare our `Posit16` functions. We also compare our `float` $log_2(x)$ function to the one in `glibc`/`Intel`'s library.

For our performance experiments, we compiled the functions in RLIBM with g++ at the `O3` optimization level. All experiments were conducted on a machine with 4.20GHz Intel i7-7700K processor and 32GB of RAM, running the Ubuntu 16.04 LTS operating system. We count the number of cycles taken to compute the correctly rounded result for each input using hardware performance counters. We use both the average number of cycles per input and total cycles for all inputs to compare performance.

5.2 Correctly Rounded Elementary Functions in RLIBM

Table 1(a) shows that RLIBM produces the correctly rounded result for all inputs with numerous elementary functions for the `Bfloat16` representation. In contrast to RLIBM, we discovered that re-purposing existing `glibc`'s or `Intel`'s `float` library for `Bfloat16` did not produce the correctly rounded result for all inputs. The case with input $x = -0.0181884765625$ for $exp10(x)$ was already discussed in Section 2.6. This case is interesting because both `glibc`'s and `Intel`'s `float` math library produces the correctly rounded result of $exp10(x)$ with respect to the `float` type. However, the result for `Bfloat16` is wrong. We found that both `glibc`'s and `Intel`'s `double` library produce the correctly rounded result for all inputs for `Bfloat16`. Our experience during this evaluation illustrates that a correctly rounded function for $\mathbb{T}'$ does not necessarily produce a correctly rounded library for $\mathbb{T}$ even if $\mathbb{T}'$ has more precision that $\mathbb{T}$.

Table 1(b) reports that RLIBM produces correctly rounded results for all inputs with elementary functions for `Posit16`. We found that SoftPosit-Math functions also produce the correctly rounded result for the available functions. However, functions $log10(x)$, $exp10(x)$, $sinh(x)$, and $cosh(x)$ are not available in the SoftPosit-Math library.

Table 1(c) reports that RLIBM produces the correctly rounded results for $log2(x)$ for all inputs with the 32-bit float data type. The corresponding function in `glibc`'s and `Intel`'s `double` library produces the correct result for all inputs. However, `glibc`'s and `Intel`'s `float` math library does not produce the correctly rounded result for all inputs. We found approximately fourteen million inputs where `glibc`'s `float` library produces the wrong result and 276 inputs where `Intel`'s `float` library produces the wrong result. In summary, we are able to generate correctly rounded results for many elementary functions for various representations using our proposed approach.

Table 1. (a) The list of `Bfloat16` functions used for our evaluation. The second column shows whether RLibm produces the correct result for all inputs. The third column and fourth column shows whether `glibc`'s float and `Intel`'s float library produces the correct result for all `Bfloat16` inputs. We use (✓) to indicate correctly rounded results and ✗, otherwise. (b) The list of `Posit16` functions used. The second column shows whether RLibm produces the correct results for all inputs. The third column shows whether the functions in SoftPosit-Math produces correctly rounded results for all inputs. `N/A` indicates that function is not available in SoftPosit-Math. (c) The `float` function used. First column indicates whether RLibm produces the correctly rounded result for all inputs. In the second and third column, we show whether `glibc`'s float and `Intel`'s float math library produce the correct result for all inputs.

Bfloat16 Functions	Using RLibm	Using glibc float	Using Intel float
$ln(x)$	✓	✓	✓
$log2(x)$	✓	✓	✓
$log10(x)$	✓	✓	✓
$exp(x)$	✓	✓	✓
$exp2(x)$	✓	✓	✓
$exp10(x)$	✓	✗	✗
$sinpi(x)$	✓	N/A	✓
$cospi(x)$	✓	N/A	✓
$sqrt(x)$	✓	✓	✓
$cbrt(x)$	✓	✓	✓
$sinh(x)$	✓	✓	✓
$cosh(x)$	✓	✓	✓

(a) Correctly rounded results with Bfloat16

Posit16 Functions	Using RLibm	Using SoftPosit-Math
$ln(x)$	✓	✓
$log2(x)$	✓	✓
$log10(x)$	✓	N/A
$sinpi(x)$	✓	✓
$cospi(x)$	✓	✓
$sqrt(x)$	✓	✓
$exp(x)$	✓	N/A
$exp2(x)$	✓	✓
$exp10(x)$	✓	✓
$sinh(x)$	✓	N/A
$cosh(x)$	✓	N/A

(b) Correctly rounded results with Posit16

float Functions	Using RLibm	Using glibc float	Using Intel float
$log2(x)$	✓	✗	✗

(c) Correctly rounded result with 32-bit float

Table 2 provides details on the polynomials for each elementary function and for each data type. For some elementary functions, we had to generate piecewise polynomials using a trial-and-error approach. As the degree of the generated polynomials and the number of terms in the polynomial are small, the resulting libraries are faster than the state-of-the-art libraries. The time taken by our tool to generate the resulting polynomials depends on the bit-width and the degree of the polynomial. It ranges from a few seconds to a few minutes.

5.3 Performance Evaluation of Elementary Functions in RLibm

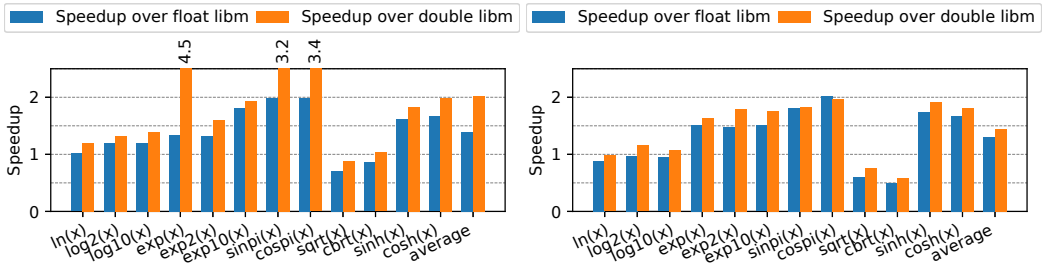
We empirically compare the performance of the functions in RLibm for `Bfloat16`, `Posit16`, and a 32-bit `float` type to the corresponding ones in `glibc`, `Intel`, and SoftPosit-Math libraries.

5.3.1 Performance of `Bfloat16` *Functions in RLibm.* To measure performance, we measure the amount of time it takes for RLibm to produce a `Bfloat16` result given a `Bfloat16` input for all inputs. Similarly, we measure the time taken by `glibc` and `Intel` libraries to produce a `Bfloat16` output given a `Bfloat16` input. As $sinpi(x)$ and $cospi(x)$ are not available in `glibc`'s libm, we transform $sinpi(x) = sin(\pi x)$ and $cospi(x) = cos(\pi x)$ before using `glibc`'s *sin* and *cos* functions. `Intel`'s libm provides implementations of $sinpi(x)$ and $cospi(x)$.

Figure 11(a) shows the speedup of RLibm's functions for `Bfloat16` compared to `glibc`'s `float` math library (left bar in the cluster) and the `double` library (right bar in the cluster). On average, RLibm's functions are 1.39× faster when compared to `glibc`'s `float` library and 2.02× faster over

Table 2. Details about the generated polynomials. For each elementary function, we report the total number of inputs in the target representation, number of special inputs, total number of reduced intervals, the number of intervals that we encoded in the LP query, the total time taken to generate the polynomials, the number of polynomials generated, the degree of the generated polynomial, and the number of terms in the polynomial.

Elementary Functions	Total # of Inputs	Special Inputs	Reduced Intervals	Intervals Used in LP	Total Time (Seconds)	# of Poly-nomials	Degree	# of Terms
Bfloat16 functions								
$ln(x)$	2^{16}	32897	128	128	0.84	1	7	4
$log2(x)$	2^{16}	32897	128	128	8.65	1	5	3
$log10(x)$	2^{16}	32897	128	128	1.63	1	5	3
$exp(x)$	2^{16}	61716	3820	3820	2.9	1	4	5
$exp2(x)$	2^{16}	61548	1937	1937	0.89	1	4	5
$exp10(x)$	2^{16}	61696	3840	3840	3	1	4	5
$sinpi(x)$	2^{16}	30976	16129	16129	32	2	1 7	1 4
$cospi(x)$	2^{16}	30976	16129	16129	32.2	3	0 6 0	1 4 1
$sqrt(x)$	2^{16}	32897	256	256	0.07	1	4	5
$cbrt(x)$	2^{16}	257	384	384	0.16	1	6	7
$sinh(x)$	2^{16}	63084	422	422	0.27	3	5 0 6	3 1 4
$cosh(x)$	2^{16}	62980	471	471	0.27	2	5 6	3 4
Posit16 functions								
$ln(x)$	2^{16}	32769	4096	4096	3.32	1	9	5
$log2(x)$	2^{16}	32769	4096	4096	5.69	1	9	5
$log10(x)$	2^{16}	32769	4096	4096	6.51	1	9	5
$exp(x)$	2^{16}	8165	57371	1740	6.17	1	6	7
$exp2(x)$	2^{16}	7160	24201	805	5.15	1	6	7
$exp10(x)$	2^{16}	12430	53106	1879	11.97	1	6	7
$sinpi(x)$	2^{16}	1	12289	12289	37.99	2	1 9	1 5
$cospi(x)$	2^{16}	1	12289	12289	85.74	3	0 8 0	1 5 1
$sqrt(x)$	2^{16}	32769	8192	8192	77.08	2	6 6	7 7
$sinh(x)$	2^{16}	14804	13044	13044	37.44	2	7 6	4 4
$cosh(x)$	2^{16}	11850	14400	14400	391.94	4	1 7 6 6	1 4 4 4
32-bit float function								
$log2(x)$	2^{32}	2155872257	7165657	7775	220.59	1	5	5



(a) Speedup over Glibc's math library (b) Speedup over Intel's math library

Fig. 11. (a) Speedup of RLibm's elementary functions compared to a baseline using Glibc's float math library (left bar) and Glibc's double math library (right bar). (b) Speedup of RLibm's elementary functions compared to a baseline using Intel's float math library (left bar) and Intel's double math library (right bar). These functions take a Bfloat16 input and produce a Bfloat16 output.

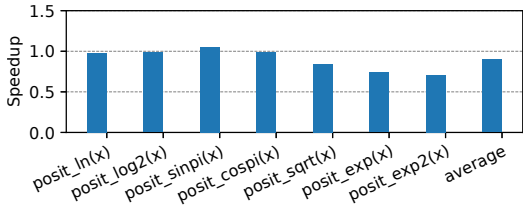


Fig. 12. Performance speedup of RLibm's functions compared to SoftPosit-Math library when the input is available as a double. It avoids the cast from Posit16 to double with RLibm. SoftPosit-Math takes as input a Posit16 value that is internally represented as an integer.

glibc's double math library. Figure 11(b) shows the speedup of RLibm's functions for Bfloat16 compared to Intel's float math library (left boar in the cluster) and the double library (right bar in the cluster). On average, RLibm's functions are 1.30× faster when compared to Intel's float library and 1.44× faster compared to Intel's double math library.

For $sqrt(x)$, RLibm's version has a slowdown because both glibc and Intel math library likely utilize the hardware instruction, FSQRT, to compute $sqrt(x)$ whereas RLibm performs polynomial evaluation. Our $cbrt(x)$ function is slower than both the glibc and Intel's math library and our logarithm functions are slower than Intel's float math library. It is likely that they use sophisticated range reduction and has a lower degree polynomial. Overall, RLibm's functions for Bfloat16 not only produce correct results for all inputs but also are faster than the existing libraries re-purposed for Bfloat16.

5.3.2 Performance of Posit16 *Elementary Functions in RLibm.* Figure 12 shows the speedup of RLibm's functions when compared to a baseline that uses SoftPosit-Math functions. The Posit16 input is cast to the double type before using RLibm. We did not measure the cost of this cast, which can incur additional overhead. SoftPosit-Math library does not have an implementation for $log10(x)$, $exp10(x)$, $sinh(x)$, and $cosh(x)$ functions. Hence, we do not report them. On average, RLibm has 11% slowdown compared to SoftPosit-Math. RLibm's $log(x)$, $log2(x)$, $cospi(x)$, and $sinpi(x)$ have similar performance compared to SoftPosit-Math, while the super-optimized implementations of SoftPosit-Math show higher performance for $exp(x)$ and $exp2(x)$ even though both libraries use polynomials of similar degree. Finally, SoftPosit-Math library computes $sqrt(x)$ using the

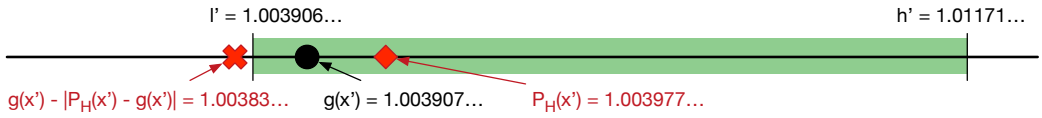


Fig. 13. More freedom in generating a polynomial for 10^x with our approach. The reduced interval $[l', h']$ (in green box) corresponds to the reduced input $x' = 0.0056264\ldots$. We show the real value of $g(x')$ (black circle) and the result produced by the polynomial generated with our approach (red diamond). If we approximated the real result $g(x')$ instead of the correctly rounded result, the margin of error for any such polynomial would be lower.

Newton-Raphson refinement method and produces a more efficient function. We plan to explore integer operations for internal computation to further improve RLIBM's performance.

5.3.3 Performance Evaluation of Elementary Functions for `Float`. RLIBM's $log2(x)$ function for the 32-bit `floating point` type has a 1.32× speedup over `glibc`'s `float` math library, which produces wrong results for 14 million inputs. Compared to `glibc`'s `double` math library which produces the correctly rounded result for all `float` inputs, RLIBM has 1.36× speedup. RLIBM's $log2(x)$ function for `float` has 1.1× and 1.2× speedup over `Intel`'s `float` and `double` math library, respectively. `Intel`'s `float` math library produces wrong results for 276 inputs.

5.4 Case Studies of Correctly Rounded Elementary Functions

We provide case studies to show that our approach (1) has more freedom in generating better polynomials, (2) generates different polynomials for the same underlying elementary function to account for numerical errors in range reduction and output compensation, and (3) generates correctly rounded results even when the polynomial evaluation is performed with the `double` type.

5.4.1 Case Study with 10^x *for* `Bfloat16`. The 10^x function is defined over the input domain $(-\infty, \infty)$. There are four classes of special cases:

$$\text{Special cases of } 10^x = \begin{cases} 0.0 & \text{if } x \leq -40.5 \\ 1.0 & \text{if } -8.4686279296875 \times 10^{-4} \leq x \leq 1.68609619140625 \times 10^{-3} \\ \infty & \text{if } x \geq 38.75 \\ NaN & \text{if } x = NaN \end{cases}$$

A quick initial check returns their result and reduces the overall input that we need to approximate.

We approximate 10^x using 2^x, which is easier to compute. We use the property, $10^x = 2^{xlog_2(10)}$ to approximate 10^x using 2^x. Subsequently, we perform range reduction by decomposing $xlog_2(10)$ as $xlog_2(10) = i + x'$, where i is an integer and $x' \in [0, 1)$ is the fractional part.

Now, 10^x decomposes to

$$10^x = 2^{xlog_2(10)} = 2^{i+x'} = 2^i 2^{x'}$$

The above decomposition requires us to approximate $2^{x'}$ where $x' \in [0, 1)$. Multiplication by 2^i can be performed using integer operations. The range reduction, output compensation, and the function we are approximating $g(x')$ is as follows:

$$RR(x) = x' = xlog_2(10) - \lfloor xlog_2(10) \rfloor \quad OC(y', x) = y'2^i = y'2^{\lfloor xlog_2(10) \rfloor} \quad g(x') = 2^{x'}$$

Our approach generated a 4^{th} degree polynomial that approximates $2^{x'}$ in the input domain $[0, 1)$. Our polynomial produces the correctly rounded result for all inputs in the entire domain for 10^x when used with range reduction and output compensation.

We are able to generate a lower degree polynomial because our approach provides more freedom to generate the correctly rounded results. We illustrate this point with an example. Figure 13 presents a reduced interval ($[l', h']$ in green region) for the reduced input ($x' = 0.00562\ldots$) in our approach. The real value of $g(x')$ is shown in black circle. In our approach, the polynomial that approximates $g(x')$ has to produce a value in $[l', h']$ such that the output compensated value produces the correctly rounded result of 10^x for all input x that reduce to x'. The value of $g(x')$ is extremely close to l' with a margin of error $\epsilon = |g(x') - l'| \approx 1.31 \times 10^{-6}$. In contrast to our approach, if we approximated the real value of $g(x')$, then we must generate a polynomial with an error of at most ϵ, *i.e.* the polynomial has to produce a value in $[g(x') - \epsilon, g(x') + \epsilon]$, which potentially necessitates a higher degree polynomial. The polynomial that we generate produces a value shown in Figure 13 with red diamond. This value has an error of $|P_{\mathbb{H}}(x') - g(x')| \approx 7.05 \times 10^{-5}$, which is much larger than ϵ. Still, the 4^{th} degree polynomial generated by our approach produces the correctly rounded value when used with the output compensation function for all inputs.

5.4.2 Case Study with $ln(x)$, $log_2(x)$, and $log_{10}(x)$ for `Bfloat16`. While creating the `Bfloat16` approximations for functions $ln(x)$, $log_2(x)$, and $log_{10}(x)$, we observed that our approach generates different polynomials for the same underlying elementary function to account for numerical errors in range reduction and output compensation. We highlight this observation in this case study.

To approximate these functions, we use a slightly modified version of the Cody and Waite range reduction technique [Cody and Waite 1980]. As a first step, we use mathematical properties of logarithms, $log_b(x) = \frac{log_2(x)}{log_2(b)}$ to approximate all three functions $ln(x)$, $log_2(x)$, and $log_{10}(x)$ using the approximation for $log_2(x)$. As a second step, we perform range reduction by decomposing the input x as $x = t \times 2^e$ where $t \in [1, 2)$ is the fractional value represented by the mantissa and e is an integer representing the exponent of the value. Then, we use the mathematical property of logarithms, $log_b(x \times y^z) = log_b(x) + z log_b(y)$, to perform range reduction and output compensation. Now, any logarithm function $log_b(x)$ can be decomposed to $log_b(x) = \frac{log_2(t)+e}{log_2(b)}$.

As a third step, to ease the job of generating a polynomial for $log_2(t)$, we introduce a new variable $x' = \frac{t-1}{t+1}$ and transform the function $log_2(t)$ to a function with rapidly converging polynomial expansion:

$$g(x') = log_2\left(\frac{1+x'}{1-x'}\right)$$

where the function $g(x')$ evaluates to $log_2(t)$.

The above input transformation, attributed to Cody and Waite [Cody and Waite 1980], enables the creation of a rapidly convergent odd polynomial, $P(x) = c_1 x + c_3 x^3 \ldots$, which reduces the number of operations. In contrast, the polynomial would be of the form $P(x) = c_0 + c_1 x + c_2 x^2 \ldots$ in the absence of above input transformation, which has terms with both even and odd degrees.

When the input x is decomposed into $x = t * e$ where $t \in [1, 2)$ and e is an integer, the range reduction function $x' = RR(x)$, the output compensation function $y = OC(y', x)$, and the function that we need to approximate, $y' = g(x')$ are as follows,

$$RR(x) = x' = \frac{t-1}{t+1}, \qquad OC(y', x) = \frac{y'+e}{log_2(b)} \qquad g(x') = log_2\left(\frac{1+x'}{1-x'}\right)$$

Hence, we approximate the same elementary function for $ln(x)$, $log_2(x)$ and $log_{10}(x)$ (*i.e.*, $g(x')$). However, the output compensation functions are different for each of them.

We observed that our approach produced different polynomials that produced correct output for $ln(x)$, $log_2(x)$, and $log_{10}(x)$ functions for `Bfloat16`, which is primarily to account for numerical errors in each output compensation function. We produced a 5^{th} degree odd polynomial for $log2(x)$, a 5^{th} degree odd polynomial with different coefficients for $log_{10}(x)$, and a 7^{th} degree odd polynomial

for $ln(x)$. Our technique also determined that there was no correct 5^{th} degree odd polynomial for $ln(x)$. Although these polynomials approximate the same function $g(x')$, they cannot be used interchangeably. For example, our experiment show that the 5^{th} degree polynomial produced for $log_2(x)$ cannot be used to produce the correctly rounded result of $ln(x)$ for all inputs.

5.4.3 Case Study with $log_2(x)$ for a 32-bit `Float`. To show that our approach is scalable to data types with numerous inputs, we illustrate a correctly rounded $log_2(x)$ function for a 32-bit `float` type. Even with state-of-the-art range reduction for $log_2(x)$ [Tang 1990], there are roughly seven million reduced inputs and its corresponding intervals. Solving an LP problem with seven million constraints is infeasible with our LP solver. Hence, we sampled five thousand reduced inputs and generated a polynomial that produces correct result for the sampled inputs. Next, we validated whether the generated polynomial produces the correctly rounded result for all inputs. We added any input where the polynomial did not produce the correctly rounded result to the sample and re-generated the polynomial. We repeated the process until the generated polynomial produced the correctly rounded result for all inputs.

We were able to generate a 5^{th} degree polynomial that produces the correct result for all inputs by using 7, 775 reduced inputs. This case study shows that our approach can be adapted for generating correctly rounded functions for data types with numerous inputs.

6 DISCUSSION

We discuss alternatives to polynomial approximation for computing correctly rounded results for small data types, design considerations with our approach, and opportunities for future work.

Look-up tables. A lookup table is an attractive alternative to polynomial approximation for data types with small bit-widths. However, it requires additional space to store these tables for each function (*i.e.*, space versus latency tradeoff). In the case of embedded controllers, computing the function in a few cycles with polynomial approximation can be appealing because lookup tables can have non-deterministic latencies due to memory footprint issues. Further, lookup tables are likely infeasible for 32-bit float or posit values.

Scalability with large data types. Our goal is to eventually generate the correctly rounded math library for FP types with larger bit-widths. The LP solver can become a bottleneck when the domain is large. In the case of `Bfloat16` and posit16, we can use all inputs to generate intervals. We observed that it is not necessary to add every interval to the LP formulation. Only highly constrained intervals need to be added. We plan to explore systematic sampling of intervals to generate polynomials for data types with larger bit-widths.

When our approach cannot generate a single polynomial that produces correctly rounded results for all inputs, we currently use a trial-and-error approach to generate piecewise polynomials (*e.g.*, *sinpi*, *cospi*, *sinh*, and $cosh(x)$ in Section 5). We plan to explore a systematic approach to generate piecewise polynomials as future work.

Validation of correctness for all inputs. In our approach, we enumerate each possible input and obtain the oracle result for each input using the same elementary function in the MPFR library that is computed with 2000 bits of precision. This MPFR result is rounded to the target representation. We validate that the polynomial generated by our approach produces exactly the same oracle result by evaluating it with each input. Although it is possible to validate whether a particular polynomial produces the correctly rounded output for the `float` data type by enumeration, it is not possible for the `double` type. Validating the correctness of the result produced by a polynomial for the `double` type is an open research question.

Importance of Range reduction. Efficient range reduction is important when the goal is to produce correctly rounded results for all inputs with the best possible performance. The math

library designer has to choose an appropriate range reduction technique for various elementary functions with our approach. Fortunately, there is a rich body of prior work on range reduction for many elementary functions, which we use. In the absence of such customized range reduction techniques, it is possible to generate polynomials that produce correctly rounded results with our approach. However, it will likely not be efficient. Further, effective range reduction techniques are important to decrease the condition number of the LP problem and to avoid overflows in polynomial evaluation. We plan to explore if we can automatically generate customized range reduction techniques as future work.

Handling multivariate functions. Currently, our approach does not handle multivariate functions such as $pow(x, y)$. The key challenge lies in encoding the constraints of multivariate functions as linear constraints, which we are exploring as part of future work.

7 RELATED WORK

Correctly rounded math libraries for FP. Since the introduction of the floating point standard [Cowlishaw 2008], a number of correctly rounded math libraries have been proposed. For example, the IBM LibUltim (or also known as MathLib) [IBM 2008; Ziv 1991], Sun Microsystem's LibMCR [Microsystems 2008], CR-LIBM [Daramy et al. 2003], and the MPFR math library [Fousse et al. 2007]. MPFR produces the correctly rounded result for any arbitrary precision.

CR-LIBM [Daramy et al. 2003; Lefèvre et al. 1998] is a correctly rounded double math library developed using Sollya [Chevillard et al. 2010]. Given a degree d, a representation $\mathbb{H}$, and the elementary function $f(x)$, Sollya generates polynomials of degree d with coefficients in $\mathbb{H}$ that has the minimum infinity norm [Brisebarre and Chevillard 2007]. Sollya uses a modified Remez algorithm with lattice basis reduction to produce polynomials. It also computes the error bound on the polynomial evaluation using interval arithmetic [Chevillard et al. 2011; Chevillard and Lauter 2007] and produces Gappa [Melquiond 2019] proofs for the error bound. Metalibm [Brunie et al. 2015; Kupriianova and Lauter 2014] is a math library function generator built using Sollya. MetaLibm is able to automatically identify range reduction and domain splitting techniques for some transcendental functions. It has been used to create correctly rounded elementary functions for the `float` and `double` types.

A number of other approaches have been proposed to generate correctly rounded results for different transcendental functions including square root [Jeannerod et al. 2011] and exponentiation [Bui and Tahar 1999]. A modified Remez algorithm has also been used to generate polynomials for approximating some elementary functions [Arzelier et al. 2019]. It generates a polynomial that minimizes the infinity norm compared to an ideal elementary function and the numerical error in the polynomial evaluation. It can be used to produce correctly rounded results when range reduction is not necessary. Compared to prior techniques, our approach approximates the correctly rounded value $RN_{\mathbb{T}}(f(x))$ and the margin of error is much higher, which generates efficient polynomials. Additionally, our approach also takes into account numerical errors in range reduction, output compensation, and polynomial evaluation.

Posit math libraries. SoftPosit-Math [Leong 2019] has a number of correctly rounded `Posit16` elementary functions, which are created using the Minefield method [Gustafson 2020]. The Minefield method identifies the interval of values that the internal computation should produce and declares all other regions as a minefield. Then the goal is to generate a polynomial that avoids the mines. The polynomials in the minefield method were generated by trial and error. Our approach is inspired by the Minefield method. It generalizes it to numerous representations, range reduction, and output compensation. Our approach also automates the process of generating polynomials by encoding the mines as linear constraints and uses an LP solver. In our prior work [Lim et al. 2020b], we have

used the CORDIC method to generate approximations to trigonometric functions for the `Posit32` type. However, they do not produce the correctly rounded result for all inputs.

Verification of math libraries. As performance and correctness are both important with math libraries, there is extensive research to prove the correctness of math libraries. Sollya verifies that the generated implementations of elementary functions produce correctly rounded results with the aid of Gappa [Daumas et al. 2005; de Dinechin et al. 2011; de Dinechin et al. 2006]. It has been used to prove the correctness of CR-LIBM. Recently, researchers have also verified that many functions in Intel's `math.h` implementations have at most 1 ulp error [Lee et al. 2017]. Various elementary function implementations have also been proven correct using HOL Light [Harrison 1997a,b, 2009]. Similarly, CoQ proof assistant has been used to prove the correctness of argument reduction [Boldo et al. 2009]. Instruction sets of mainstream processors have also been proven correct using proof assistants (*e.g.*, division and $sqrt(x)$ instruction in IBM Power4 processor [Sawada 2002]). RLIBM validates that the reported polynomial produces the correctly rounded result for all inputs. We likely have to rely on prior verification efforts to check the correctness of RLIBM's polynomials for the `double` type.

Rewriting tools. Mathematical rewriting tools are other alternatives to create correctly rounded functions. If the rounding error in the implementation is the root cause of an incorrect result, we can use tools that detect numerical errors to diagnose them [Benz et al. 2012; Chowdhary et al. 2020; Fu and Su 2019; Goubault 2001; Sanchez-Stern et al. 2018; Yi et al. 2019; Zou et al. 2019]. Subsequently, we can rewrite them using tools such as Herbie [Panchekha et al. 2015] or Salsa [Damouche and Martel 2018]. Recently, a repair tool was proposed specifically for reducing the error of math libraries [Yi et al. 2019]. It identifies the domain of inputs that result in high error. Then, it uses piecewise linear or quadratic equations to repair them for the specific domain. However, currently, these rewriting tools do not guarantee correctly rounded results for all inputs.

8 CONCLUSION

A library to approximate elementary functions is a key component of any FP representation. We propose a novel approach to generate correctly rounded results for all inputs of an elementary function. The key insight is to identify the amount of freedom available to generate the correctly rounded result. Subsequently, we use this freedom to generate a polynomial using linear programming that produces the correct result for all inputs. The resulting polynomial approximations are faster than existing libraries while producing correct results for all inputs. Our approach can also allow designers of elementary functions to make pragmatic trade-offs with respect to performance and correctness. More importantly, it can enable standards to mandate correctly rounded results for elementary functions with new representations.

ACKNOWLEDGMENTS

We thank the POPL reviewers for their feedback. This material is based upon work supported by the National Science Foundation under Grant No. 1908798, Grant No. 1917897, and Grant No. 1453086. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Denis Arzelier, Florent Bréhard, and Mioara Joldes. 2019. Exchange Algorithm for Evaluation and Approximation Error-Optimized Polynomials. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. 30–37. https://doi.org/10.1109/ARITH.2019.00014

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

Jeremy Bernstein, Jiawei Zhao, Markus Meister, Ming-Yu Liu, Anima Anandkumar, and Yisong Yue. 2020. Learning compositional functions via multiplicative weight updates. arXiv:2006.14560 [cs.NE]

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

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

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

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

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

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

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

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

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

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

William J Cody and William M Waite. 1980. *Software manual for the elementary functions*. Prentice-Hall, Englewood Cliffs, NJ.

Mike Cowlishaw. 2008. *IEEE Standard for Floating-Point Arithmetic*. IEEE 754-2008. IEEE Computer Society. 1–70 pages. https://doi.org/10.1109/IEEESTD.2008.4610935

Nasrine Damouche and Matthieu Martel. 2018. Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs. In *Automated Formal Methods (Kalpa Publications in Computing, Vol. 5)*, Natarajan Shankar and Bruno Dutertre (Eds.). 63–76. https://doi.org/10.29007/j2fd

Catherine Daramy, David Defour, Florent Dinechin, and Jean-Michel Muller. 2003. CR-LIBM: A correctly rounded elementary function library. In *Proceedings of SPIE Vol. 5205: Advanced Signal Processing Algorithms, Architectures, and Implementations XIII*, Vol. 5205. https://doi.org/10.1117/12.505591

Marc Daumas, Guillaume Melquiond, and Cesar Munoz. 2005. Guaranteed proofs using interval arithmetic. In *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*. 188–195. https://doi.org/10.1109/ARITH.2005.25

Florent de Dinechin, Christopher Lauter, and Guillaume Melquiond. 2011. Certifying the Floating-Point Implementation of an Elementary Function Using Gappa. In *IEEE Transactions on Computers*, Vol. 60. 242–253. https://doi.org/10.1109/TC.2010.128

Florent de Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. 2006. Assisted Verification of Elementary Functions Using Gappa. In *Proceedings of the 2006 ACM Symposium on Applied Computing* (Dijon, France) *(SAC '06)*. Association for Computing Machinery, New York, NY, USA, 1318–1322. https://doi.org/10.1145/1141277.1141584

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

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

Ambros Gleixner, Daniel E. Steffy, and Kati Wolter. 2015. *Iterative Refinement for Linear Programming*. Technical Report 15-15. ZIB, Takustr. 7, 14195 Berlin. https://doi.org/10.1287/ijoc.2016.0692

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

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

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

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

John Gustafson and Isaac Yonemoto. 2017. Beating Floating Point at Its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations: an International Journal* 4, 2 (June 2017), 71–86. https://doi.org/10.14529/jsfi170206

John Harrison. 1997a. 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

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

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

IBM. 2008. *Accurate Portable MathLib*. http://oss.software.ibm.com/mathlib/

Intel. 2019. *Delivering a New Intelligence with AI at Scale*. https://www.intel.com/content/www/us/en/artificial-intelligence/posts/nnp-aisummit.html

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

Jeff Johnson. 2018. *Rethinking floating point for deep learning*. http://export.arxiv.org/abs/1811.01721

William Kahan. 2004. *A Logarithm Too Clever by Half*. https://people.eecs.berkeley.edu/~wkahan/LOG10HAF.TXT

Dhiraj D. Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. 2019. A Study of BFLOAT16 for Deep Learning Training. arXiv:1905.12322

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

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

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

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

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

Jay P. Lim, Mridul Aanjaneya, John Gustafson, and Santosh Nagarakatte. 2020a. A Novel Approach to Generate Correctly Rounded Math Libraries for New Floating Point Representations. arXiv:2007.05344 [cs.MS]

Jay P. Lim and Santosh Nagarakatte. 2020a. *RLibm*. https://github.com/rutgers-apl/rlibm

Jay P. Lim and Santosh Nagarakatte. 2020b. *RLibm-generator*. https://github.com/rutgers-apl/rlibm-generator

Jay P. Lim, Matan Shachnai, and Santosh Nagarakatte. 2020b. 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

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

Sun Microsystems. 2008. *LIBMCR 3 "16 February 2008" "libmcr-0.9"*. http://www.math.utah.edu/cgi-bin/man2html.cgi?/usr/local/man/man3/libmcr.3

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

NVIDIA. 2020. *TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x*. https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/

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

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.

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

Joe 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*.

Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benin. 2018. A transprecision floating-point platform for ultra-low power computing. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1051–1056. https://doi.org/10.23919/DATE.2018.8342167

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

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

Shibo Wang and Pankaj Kanwar. 2019. *BFloat16: The secret to high performance on Cloud TPUs*. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus

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

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

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