Correctly Rounded Math Libraries without Worrying about the Application's Rounding Mode

SEHYEOK PARK, Rutgers University, USA
JUSTIN KIM, Rutgers University, USA
SANTOSH NAGARAKATTE, Rutgers University, USA

Our RLibm project has recently proposed methods to generate a single implementation for an elementary function that produces correctly rounded results for multiple rounding modes and representations with up to 32-bits. They are appealing for developing fast reference libraries without double rounding issues. The key insight is to build polynomial approximations that produce the correctly rounded result for a representation with two additional bits when compared to the largest target representation and with the "non-standard" round-to-odd rounding mode, which makes double rounding the RLibm math library result to any smaller target representation innocuous. The resulting approximations generated by the RLibm approach are implemented with machine supported floating-point operations with the *round-to-nearest* rounding mode. When an application uses a rounding mode other than the round-to-nearest mode, the RLibm math library saves the application's rounding mode, changes the system's rounding mode to round-to-nearest, computes the correctly rounded result, and restores the application's rounding mode. This frequent change of rounding modes has a performance cost.

This paper proposes two new methods, which we call rounding-invariant outputs and rounding-invariant input bounds, to avoid the frequent changes to the rounding mode and the dependence on the round-to-nearest mode. First, our new rounding-invariant outputs method proposes using the round-to-zero rounding mode to implement RLibm's polynomial approximations. We propose fast, error-free transformations to emulate a round-to-zero result from any standard rounding mode without changing the rounding mode. Second, our rounding-invariant input bounds method factors any rounding error due to different rounding modes using interval bounds in the RLibm pipeline. Both methods make a different set of trade-offs and improve the performance of resulting libraries by more than 2×.

CCS Concepts: • **Mathematics of computing → Mathematical software**.

Additional Key Words and Phrases: RLIBM, rounding-to-zero emulation, rounding-invariant bounds

ACM Reference Format:
Sehyeok Park, Justin Kim, and Santosh Nagarakatte. 2025. Correctly Rounded Math Libraries without Worrying about the Application's Rounding Mode. *Proc. ACM Program. Lang.* 9, PLDI, Article 229 (June 2025), 24 pages. https://doi.org/10.1145/3729332

1 Introduction

Math libraries provide implementations of commonly used elementary functions. The outputs of these elementary functions are irrational values for almost all inputs and cannot be represented exactly in a finite precision floating-point (FP) representation. The correctly rounded result of an elementary function for a given input is the result produced after computing the result with infinite precision and then rounded to the target representation. The problem of generating correctly

Authors' Contact Information: Sehyeok Park, Rutgers University, Piscataway, USA, sp2044@cs.rutgers.edu; Justin Kim, Rutgers University, Piscataway, USA, jk1849@scarletmail.rutgers.edu; Santosh Nagarakatte, Rutgers University, Piscataway, USA, santosh.nagarakatte@cs.rutgers.edu.

ACM 2475-1421/2025/6-ART229
https://doi.org/10.1145/3729332

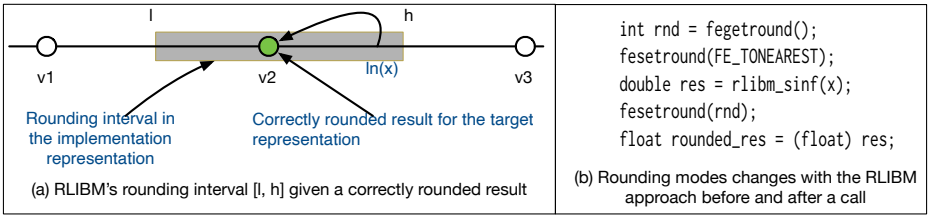


Fig. 1. (a) The three representable FP values (v1, v2, and v3) in the target representation. RLibm's rounding interval $[l, h]$ (shown in gray) in the implementation representation such that any value in this interval rounds to v2 assuming the rounding mode is round-to-nearest-ties-to-even. (b) Changes to the rounding mode performed by the RLibm approach before and after calling the elementary function.

rounded results for arbitrary target representations is known to be challenging (*i.e.*, Table Maker's dilemma [26]). Hence, the IEEE-754 standard recommends but does not mandate correctly rounded results for elementary functions. Recent efforts, such as the CORE-MATH project [50] and our RLibm project [31, 32, 35], have demonstrated that fast and correctly rounded libraries are feasible. There is also a working group discussion to require correctly rounded implementations in the upcoming 2029 IEEE-754 standard [9]. The minimax approximation method is the most well-known method for building correctly rounded libraries. Effectively, these methods generate polynomial approximations that minimize the maximum error across all inputs with respect to the real value (see Chapter 3 of [37]). Subsequently, the error in the polynomial evaluation methods are bounded to ensure that numerical errors do not change the rounding decision.

Our RLibm project. Unlike traditional minimax methods, our RLibm project makes a case for directly approximating the correctly rounded result [31, 32, 35]. The insight is to split the task of generating the oracle and the task of generating an efficient implementation given an oracle such as the MPFR library [21]. When building a correctly rounded library for the 32-bit FP representation (*i.e.*, the target representation), the RLibm project implements the library using the 64-bit FP representation (*i.e.*, the implementation representation). Then, there is an interval of values in the implementation representation around the correctly rounded result of the target representation such that any value in this interval rounds to the correctly rounded result. Figure 1(a) shows the correctly rounded result (*i.e.*, v2) and the rounding interval. Given this interval $[l, h]$, the task of producing a correctly rounded result for an input x with a polynomial of degree d can be expressed as a linear constraint: $l \leq C_0 + C_1x + C_2x^2 + C_3x^3 + \ldots + C_dx^d \leq h$. The size of the rounding interval is 1 ULP (units in the last place) for all inputs.

Multiple representations and rounding modes. Low precision representations are becoming mainstream especially with accelerators (*e.g.*, `bfloat16` [56], `tensorfloat32` [40], and `FP8`). They are increasingly used in scientific computing apart from machine learning. Further, the IEEE-754 standard specifies four distinct rounding modes for the binary FP representation: round-to-nearest-ties-to-even (*RN*), round-towards-zero (*RZ*), round-up (*RU*), and round-down (*RD*). Each rounding mode is attractive in specific domains. For example, the computational geometry algorithms library (CGAL) uses different rounding modes. Similarly, some hardware accelerators use the round-to-zero (*RZ*) mode because it can be implemented efficiently. Rather than designing a custom math library for each such rounding mode and representation, generating a single math library that handles all these rounding modes and new representations is attractive as a reference library. Existing correctly rounded libraries for a single representation such as CORE-MATH [50] and CR-LIBM [15] do not produce correctly rounded results when they are repurposed for these new

representations because of double rounding errors [36]. The first rounding happens when the real value is rounded to the representation the math library was originally designed for and the second rounding happens when the result from the math library is rounded to the target representation.

RLIBM's method to handle multiple representations. The RLIBM project includes an appealing method to generate a single implementation that produces correctly rounded results for multiple representations and rounding modes [35]. When the goal is to generate correctly rounded results for all representations with up to n-bits, the RLIBM project's approach is to approximate the correctly rounded result of a $(n+2)$-bit representation with a non-standard rounding mode called `round-to-odd`. In the round-to-odd mode, the real value that is not exactly representable is rounded to the nearest FP value whose bit-pattern is odd. When a real value is exactly representable in the FP representation, it is represented with that FP value. When the round-to-odd result with the $(n+2)$-bit representation is subsequently rounded to any target representation with n or fewer bits, it produces correctly rounded results. Effectively, RLIBM's approach of computing the round-to-odd (RO) result with a $(n+2)$-bit representation makes double rounding harmless.

Range reduction, output compensation, and polynomial evaluation with FP arithmetic. Typically, polynomial approximations are feasible over domains much smaller than the dynamic range of a 32-bit FP representation. For example, it is much more effective to approximate $log(x)$ over inputs $x \in [0, 1/128)$ rather than over the entire range of 32-bit floats where $|x| \in [2^{-149}, 2^{128})$. As a first step, each input x in the original domain is transformed to a value in a smaller domain x' through a process known as range reduction. The RLIBM implementations represent each range reduced input x' as a 64-bit, double-precision FP number, which is the internal representation used for all subsequent FP operations. Subsequently, a polynomial approximation computes the result for the input in the small domain (*i.e.*, $y' = P(x')$). Additional operations that are collectively known as the output compensation function map the result y' to produce the result for the original input (*i.e.*, $y = OC(y', x)$). Range reduction, polynomial evaluation, and output compensation are performed with FP operations and can accumulate rounding error.

To generate a polynomial approximation with guaranteed correctness, the RLIBM pipeline first computes the *reduced input* x' for each input x with the range reduction algorithm. Using the 34-bit RO oracle of $f(x)$, it computes the round-to-odd rounding interval $[l, h]$ for every input. The RLIBM pipeline subsequently identifies for each reduced input the widest possible *reduced interval* $[l', h']$ such that $\forall y' \in [l', h'], l \leq OC(y', x) \leq h$. Finally, it solves a system of linear inequalities $l' \leq P(x') \leq h'$ to generate the polynomial approximation $P(x')$. Given the manner in which each reduced interval $[l', h']$ is derived (*i.e.*, $\forall y' \in [l', h'], l \leq OC(y', x) \leq h$), a polynomial evaluation result that satisfies $l' \leq P(x') \leq h'$ is guaranteed to satisfy $l \leq OC(P(x'), x) \leq h$.

RLIBM uses round-to-nearest as the implementation rounding mode. The RLIBM project uses the RN mode for its implementations. When an application uses a rounding mode other than RN, the RLIBM project saves the application's rounding mode, changes the default rounding mode of the system to RN, computes the output of the math library implementation, and restores the application's rounding mode before the final rounding to the target representation as shown in Figure 1(b). Each rounding mode change can incur up to 40 cycles on a modern Linux machine. Specifically, RLIBM implementations are only guaranteed to produce intermediate values within the rounding intervals of the final results when they are invoked using the RN mode. Hence, not changing the rounding mode to RN as shown in Figure 1(b) may lead to the wrong results.

This paper. This paper proposes two new methods, which we call rounding-invariant outputs and rounding-invariant input bounds, to completely eliminate the rounding mode changes necessitated by the RLIBM approach while maintaining correctness under all application-level rounding modes.

Rounding-invariant outputs by emulating round-to-zero results. In our rounding-invariant outputs method, we propose to use round-to-zero (RZ) as the underlying implementation rounding

mode instead of round-to-nearest (*RN*). Our key insight is that it is possible to emulate the *RZ* result under any of the four rounding modes without having to explicitly change the application's rounding mode. We design new algorithms that compute the *RZ* result irrespective of the application's rounding mode using error-free transformations that adjust the error in an FP operation using a sequence of auxiliary FP operations and bit manipulations (see Section 3.1). This method requires very few changes to the RLibm pipeline except performing reduced interval and polynomial generation with the *RZ* mode. However, it requires wrapping every rounding mode-dependent addition and multiplication in the final implementations with our new algorithms to adjust the initial results, which is faster than changing the rounding mode but still entails noticeable overhead.

Rounding-invariant input bounds for measuring variability induced by various rounding modes. The key idea is to bound the range of values that can arise across different rounding modes given the different possible combinations of rounding error associated with a series of FP operations (see Section 3.2). Under this new approach, we no longer consider the final output of a sequence of FP operations to be a single FP value, as it would be if all operations adhered to a single rounding mode. Instead, we treat the result as an interval to account for the varying effects of different rounding rules. This approach is based on the property that for any given faithfully rounded FP arithmetic operation on finite operands (*i.e.*, operands that are neither infinity nor *NaN*), the round-down (*RD*) result is the lower bound and the round-up (*RU*) result is the upper bound. Leveraging this property, we compose the bounds on the result of each FP operation bottom-up using interval arithmetic to deduce the final bounds for a target sequence. In doing so, we can identify the minimum and maximum possible outputs a candidate implementation could ultimately produce for a given input across all rounding modes and confirm that both values satisfy the associated correctness constraints. This approach requires non-trivial changes to the RLibm pipeline as it involves correctness constraints that are different from those derivable from a single rounding mode. The main advantage of this approach is that the outputs of the FP operations in the resulting implementations do not require any adjustments to satisfy correctness.

Our resulting library is more than 2× faster than the existing RLibm prototypes. Our prototype is the first math library that produces correctly rounded results for all inputs across multiple representations for all four standard rounding modes, regardless of the application-level rounding mode.

2 Background

Rounding modes. The IEEE-754 standard provides the specification for four rounding modes for the binary FP representation. These are round-to-nearest (*RN*), round-to-zero (*RZ*), round-up (*RU*), and round-down (*RD*). Since this paper is about implementing math libraries that can operate directly under any application level rounding mode, we provide background to understand the behavior of various rounding modes and their interactions with FP arithmetic.

$$pred(r) = max\{t \in \mathbb{T}, t < r\} \quad (1)$$
$$succ(r) = min\{t \in \mathbb{T}, r < t\} \quad (2)$$

(a) The pred and succ functions

$$RZ(r) = \begin{cases} r, & \text{if } r \in \mathbb{T} \\ succ(r), & \text{else if } r < 0 \\ pred(r) & \text{else} \end{cases} \quad (3)$$

(b) The round-to-zero (RZ) mode

$$RD(r) = \begin{cases} r, & \text{if } r \in \mathbb{T} \\ pred(r) & \text{else} \end{cases} \quad (4)$$

(c) The round-down (RD) mode

$$RU(r) = \begin{cases} r, & \text{if } r \in \mathbb{T} \\ succ(r) & \text{else} \end{cases} \quad (5)$$

(d) The round-up (RU) mode

Fig. 2. (a) The pred and succ functions used for faithful rounding of a real number r. The *RZ*, *RD*, and *RU* rounding modes defined using the pred and succ functions are shown in (b), (c), and (d), respectively.

Let $\mathbb{R}$ represent the set of all real numbers and $\mathbb{T}$ represent the set of all the numbers in a given FP representation. With respect to the outputs and operands of FP operations in the RLibm project, $\mathbb{T}$ is the set of all 64-bit FP numbers. For a given number $r \in \mathbb{R}$, its neighbors in the target representation $\mathbb{T}$, denoted $pred(r)$ and $succ(r)$, can be defined through the equations in Figure 2(a).

For a given rounding function to be faithful, it must return either $pred(r)$ and $succ(r)$ whenever the input r is not exactly representable in $\mathbb{T}$. The four rounding modes considered by RLibm, which are *RZ*, *RN*, *RD*, and *RU*, all adhere to this requirement. We apply the notation $rnd(r)$ to denote a rounding function that applies any of these four rounding modes. For the purposes of this paper, we restrict the domain of all functions $rnd(r)$ to *non-zero real numbers*.

The round-to-zero (*RZ*) mode. For our rounding-invariant outputs approach, we propose to use *RZ* as the default implementation rounding mode and simulate its result across all application rounding modes. The rounding function $RZ(r)$, which applies rounding via *RZ*, is defined as shown in Figure 2(b).

The round-down (*RD*) and the round-up (*RU*) mode. Our proposed rounding-invariant input bounds method uses the *RD* and *RU* modes to bound the variability induced by the various rounding modes. The rounding functions $RD(r)$ and $RU(r)$ can be defined as shown in Figure 2(c) and Figure 2(d), respectively.

The definitions of $pred(r)$ and $succ(r)$ in Figure 2(a) (Equations 1 and 2) along with Equations 4 and 5 in Figure 2 reveal the following properties for *RD* and *RU*: $\forall r \in \mathbb{R} \setminus \{0\}, pred(r) \leq RD(r)$ and $\forall r \in \mathbb{R} \setminus \{0\}, RU(r) \leq succ(r)$. Given these properties, one could define *RD* and *RU* in the following manner.

Definition 1. For all $r \in \mathbb{R} \setminus \{0\}$, $RD(r)$ is the largest number $t \in \mathbb{T}$ such that $t \leq r$.

Definition 2. For all $r \in \mathbb{R} \setminus \{0\}$, $RU(r)$ is the smallest number $t \in \mathbb{T}$ such that $t \geq r$.

Using the properties and definitions of faithful rounding, *RD*, and *RU*, we state the following lemma providing the expected bounds on the faithfully rounded outputs of rounding functions.

Lemma 1. *Let rnd be any rounding function that faithfully rounds a number* $r \in \mathbb{R} \setminus \{0\}$ *to a number* $t \in \mathbb{T}$. $\forall r \in \mathbb{R} \setminus \{0\}, RD(r) \leq rnd(r) \leq RU(r)$.

The lemma is directly derivable from the definition of faithful rounding. Lemma 1 guarantees that when a non-zero real number r is rounded using a faithful rounding function *rnd*, $RD(r)$ and $RU(r)$ will respectively serve as the lower and upper bounds of $rnd(r)$. Lemma 2 and 3 detail the well-established monotonically non-decreasing properties of faithful rounding functions [38], specifically with regard to *RD* and *RU*.

Lemma 2. $\forall a, \forall b \in \mathbb{R} \setminus \{0\}, a \leq b \implies RD(a) \leq RD(b)$

Lemma 3. $\forall a, \forall b \in \mathbb{R} \setminus \{0\}, a \leq b \implies RU(a) \leq RU(b)$

Preservation of signs with faithful rounding. The final property of interest pertains to the preservation of signs.

Definition 3. For all $v \in (\mathbb{R} \setminus \{0\}) \cup \mathbb{T}$ where $\mathbb{T}$ is a FP representation, we define $sign(v)$ to be 0 for positive numbers and 1 for negative numbers. For the FP numbers $+0, -0 \in \mathbb{T}$, we define the sign as $sign(+0) = 0$ and $sign(-0) = 1$.

Lemma 4. *Let rnd be any rounding function that faithfully rounds a number* $r \in \mathbb{R} \setminus \{0\}$ *to a FP number* $t \in \mathbb{T}$. *For all* $r \in \mathbb{R} \setminus \{0\}$ *and for all rnd,* $sign(r) = sign(rnd(r))$.

Under our definition of *sign*, Lemma 4 signifies the sign preserving property of faithful rounding for non-zero values. The proof for Lemma 4 is available in our extended technical report [43].

$$a \oplus_{rnd} b = \begin{cases} +0, & \text{if } a \text{ is } +0 \text{ and } b \text{ is } +0 \\ -0, & \text{if } a \text{ is } -0 \text{ and } b \text{ is } -0 \\ +0, & \text{if } a \text{ is } -b \text{ and } rnd \neq RD \\ -0, & \text{if } a \text{ is } -b \text{ and } rnd = RD \\ rnd(a+b) & \text{else} \end{cases} \quad (6)$$

(a)

$$a \otimes_{rnd} b = \begin{cases} +0, & \text{if } (a \text{ is } +0 \text{ or } a \text{ is } -0) \text{ and } sign(a) = sign(b) \\ +0, & \text{if } (b \text{ is } +0 \text{ or } b \text{ is } -0) \text{ and } sign(a) = sign(b) \\ -0, & \text{if } (a \text{ is } +0 \text{ or } a \text{ is } -0) \text{ and } sign(a) \neq sign(b) \\ -0, & \text{if } (b \text{ is } +0 \text{ or } b \text{ is } -0) \text{ and } sign(a) \neq sign(b) \\ rnd(a \times b) & \text{else} \end{cases} \quad (7)$$

(b)

Fig. 3. (a) The rounded addition $\oplus_{rnd}$ for any rounding mode $rnd \in \{RN, RZ, RD, RU\}$. (b) The rounded multiplication $\otimes_{rnd}$ for any rounding mode rnd.

Having introduced pertinent properties of faithful rounding and our definition of $sign(r)$, we refer back to Equations 3 through 5 and elaborate upon crucial intricacies. We constrain the domains of the rounding functions $rnd(r)$ for all $rnd \in \{RN, RZ, RD, RU\}$ to non-zero real numbers. This is because all FP representations $\mathbb{T}$ considered in this paper treat +0 and −0 as separate FP numbers while equating both to 0 in the context of real arithmetic. Distinguishing +0 and −0 creates ambiguity as to which of the two numbers a rounding function should return for 0. This ambiguity can be resolved for rounded FP arithmetic by defining what an FP operation $a \odot b$ should return when its real number counterpart $a \cdot b = 0$. For the FP operations of concern, which are addition and multiplication, the choice between +0 or −0 is dependent on both the rounding mode environment and the sign of the operands. We provide IEEE-754 standard-compliant definitions of $a \oplus_{rnd} b$ and $a \otimes_{rnd} b$ in Figure 3, which represent the output of a FP addition and multiplication under a given rounding mode $rnd \in \{RN, RZ, RD, RU\}$. Henceforth, we reserve the notations $\oplus$ and $\otimes$ for FP addition and multiplication respectively while using + and $\times$ solely for real arithmetic operations. When the rounding rule being applied is relevant, we apply the notations $\oplus_{rnd}$ and $\otimes_{rnd}$. We restrict the domain of the equations for $\oplus_{rnd}$ and $\otimes_{rnd}$ to non-NaN, non-infinity operands.

In the context of Equation 6 in Figure 3(a), $rnd(a + b)$ represents the output obtained from subjecting the real arithmetic result of $a + b$ to a rounding function $rnd(r)$ as exemplified in Equations 3 through 5. The expression $rnd(a \times b)$ in Equation 7 can be interpreted analogously to $rnd(a + b)$. The first four cases of Equation 6 detail different scenarios under which $a + b = 0$. Likewise, the first four cases of Equation 7 cover situations where $a \times b = 0$. Based on these equations, we infer that $a \oplus_{rnd} b = rnd(a + b)$ whenever $a + b \neq 0$ and $a \otimes_{rnd} b = rnd(a \times b)$ whenever $a \times b \neq 0$. By construction, $a \oplus_{rnd} b$ and $a \otimes_{rnd} b$ return faithfully rounded versions of $a + b$ and $a \times b$ respectively. These operations therefore possess the properties of faithful rounding detailed in Lemma 1 as it pertains to RD's and RU's roles in producing the lower and upper bounds respectively. The two operations also reflect the monotonic properties of rounding detailed in Lemmas 2 and 3. We highlight the properties of $\oplus_{rnd}$ and $\otimes_{rnd}$ most important to our theorems through the following lemmas.

Lemma 5. $\forall a, b \in \mathbb{T} \setminus \{NaN, \pm\infty\}, \forall rnd \in \{RN, RZ, RD, RU\}, a \oplus_{RD} b \leq a \oplus_{rnd} b \leq a \oplus_{RU} b.$

Lemma 6. $\forall a, b \in \mathbb{T} \setminus \{NaN, \pm\infty\}, \forall rnd \in \{RN, RZ, RD, RU\}, a \otimes_{RD} b \leq a \otimes_{rnd} b \leq a \otimes_{RU} b.$

Lemma 7. $\forall a, b, c, d \in \mathbb{T} \setminus \{NaN, \pm\infty\}, a + b \leq c + d \implies (a \oplus_{RD} b \leq c \oplus_{RD} d) \wedge (a \oplus_{RU} b \leq c \oplus_{RU} d).$

Lemma 8. $\forall a, b, c, d \in \mathbb{T} \setminus \{NaN, \pm\infty\}, a \times b \leq c \times d \implies (a \otimes_{RD} b \leq c \otimes_{RD} d) \wedge (a \otimes_{RU} b \leq c \otimes_{RU} d).$

Propagating bounds using interval arithmetic. Our rounding-invariant input bounds approach treats both the operands and output of each FP operation as a *range* of values rather than a single value. It relies on the properties of interval arithmetic listed below to identify the expected lower and upper bounds on the result of a FP operation when its operands are represented as

ranges of FP numbers excluding NaNs and $\pm\infty$. Specifically, we use these properties to identify the output bounds of the *real arithmetic* counterpart of an FP operation given the ranges of its FP operands. We denote the lower and upper bounds of the final output a of an ordered sequence of FP operations as $\underline{a}$ and $\overline{a}$, respectively.

Lemma 9. $\forall a \in [\underline{a}, \overline{a}], \forall b \in [\underline{b}, \overline{b}], \underline{a} + \underline{b} \leq a + b \leq \overline{a} + \overline{b}$.

Lemma 10. $\forall a \in [\underline{a}, \overline{a}], \forall b \in [\underline{b}, \overline{b}], \min(\underline{a} \times \underline{b}, \underline{a} \times \overline{b}, \overline{a} \times \underline{b}, \overline{a} \times \overline{b}) \leq a \times b \wedge a \times b \leq \max(\underline{a} \times \underline{b}, \underline{a} \times \overline{b}, \overline{a} \times \underline{b}, \overline{a} \times \overline{b})$.

3 Rounding Mode Independence Using Our Approach

Our goal is to develop implementations for elementary functions that produce correctly rounded results for all FP inputs across multiple representations with up to 32-bits. We seek to achieve correctness for all four standard rounding modes (*e.g.*, *RN*, *RZ*, *RU*, and *RD*) and any faithful rounding mode potentially used by the invoking applications and to do so without requiring any explicit rounding mode changes. Using the RLibm approach, we attempt to generate polynomial approximations over the reduced inputs, which when used with the output compensation function can produce 64-bit values within the rounding interval of the 34-bit round-to-odd (*RO*) result. When a 64-bit FP value in the rounding interval of the 34-bit *RO* result is *double rounded* to any target representation less than or equal to 32-bits, it is guaranteed to produce the correctly rounded result regardless of the rounding rule used for the final rounding.

The main task is thus ensuring that a candidate approximation can produce such 64-bit values for all inputs regardless of the invoking application's rounding mode. The key challenge in this endeavor is that the range reduction, polynomial evaluation, and output compensation processes involve FP arithmetic, which can experience different rounding errors depending on the rounding mode. Hence, an implementation's 64-bit intermediate output prior to the final rounding may differ depending on the rounding mode under which it was produced. The RLibm pipeline generates candidate approximations that produce correct results with round-to-nearest, which is the default rounding mode used by its generators. Inevitably, the RLibm implementations necessitate rounding mode changes to *RN* to ensure correctness. Saving and restoring the application's rounding mode as required by the RLibm prototype requires up to 40 cycles for each input.

This paper proposes two new methods to completely remove the rounding mode changes required by the RLibm approach. First, we make a case for using the round-to-zero (*RZ*) mode as the implementation rounding mode. We design error-free transformations that compute the error in a FP operation to simulate the *RZ* mode result when the application uses any other rounding mode (see Section 3.1). Our second method computes the bounds on the full range of outputs that are possible when evaluating polynomial approximations and output compensation functions under different rounding modes. We propose a new method that recursively defines the lower and upper bounds for individual FP operations to account for rounding errors across different rounding modes and composes them bottom-up using interval arithmetic (see Section 3.2) to represent the results obtained from evaluating polynomials and output compensation functions. Subsequently, we change the reduced interval and polynomial generation processes in the RLibm pipeline to generate polynomials that satisfy the new constraints for correctness stemming from the incorporation of rounding induced variability.

Both these approaches have their own trade-offs. The first method requires augmenting every FP operation with steps that collectively compute its rounding error, assess whether its output conforms to *RZ*, and make necessary adjustments. In contrast, the second method requires extensive changes to the RLibm pipeline with regard to generating reduced intervals and polynomials. It does

not require any additional operations for evaluating polynomials in the resultant implementations once the polynomial approximations are generated.

3.1 Rounding Independence with Round-to-Zero Emulation

To produce the *RZ* result across all rounding modes of interest, we design error-free transformations, which are a sequence of FP operations that compute the error in a given operation using FP arithmetic. Using the error-free transformations, we design a decision procedure that determines whether the original result produced under the application's rounding mode needs to be adjusted to match the result expected under *RZ*.

Intuition for computing the *RZ* result. From the perspective of implementing RLibm's core elementary functions with 64-bit FP operations, the operations of concern are addition and multiplication. Given two non-NaN, non-infinity 64-bit FP numbers a and b, we want to compute $a \oplus_{RZ} b$ and $a \otimes_{RZ} b$ whenever $a \oplus_{rnd} b$ and $a \otimes_{rnd} b$ are computed with different rounding modes. The key challenge in computing the *RZ* results is that neither $a \oplus_{rnd} b$ nor $a \otimes_{rnd} b$ provides any direct indication as to whether $a \oplus_{rnd} b \neq a + b$ or $a \otimes_{rnd} b \neq a \times b$. The goal is thus to create a sequence of FP operations that compute the rounding error in $a \oplus_{rnd} b$ (or $a \otimes_{rnd} b$) with respect to the real result $a + b$ (or $a \times b$) and adjust the original output to match the *RZ* result based on the error. We preface the discussion of our algorithms by emphasizing that they are intended to target non-*NaN*, non-infinity FP operands for which neither the sum nor the product induces overflow. The FP numbers used with these algorithms in the final implementations satisfy these criteria.

Relationship between $\oplus_{RZ}$ and other $\oplus_{rnd}$. Given two FP numbers a and b that need to be summed, we need to adjust the output $a \oplus_{rnd} b \in \mathbb{T}$ whenever $a \oplus_{rnd} b$ differs from $a \oplus_{RZ} b$. Given Equations 3 (Figure 2) and 6 (Figure 3), this occurs when either a is $-b$ and $rnd = RD$ or when $a + b$ is not exactly representable. Addressing the former case entails setting $a \oplus_{rnd} b$ to be +0 whenever a is $-b$. In the latter case, $a \oplus_{RZ} b$ and $a \oplus_{rnd} b$ deviate whenever $a \oplus_{RZ} b = pred(a+b)$ and $a \oplus_{rnd} b = succ(a+b)$, or vice versa. The value $a \oplus_{rnd} b$ is equal to $succ(a+b)$ when $a \oplus_{RZ} b = pred(a+b)$ if $a + b \notin \mathbb{T}$ and $0 < a + b$, while the opposite scenario can occur when $a + b \notin \mathbb{T}$ and $a + b < 0$. Producing $a \oplus_{RZ} b$ thus entails adjusting $a \oplus_{rnd} b$ whenever $a + b$ is not exactly representable (*i.e.*, $a + b \neq a \oplus_{rnd} b$) and $|a + b| < |a \oplus_{rnd} b|$.

Based on the definitions of *pred* and *succ*, the absolute difference between $a \oplus_{rnd} b$ and $a \oplus_{RZ} b$ when $|a + b| < |a \oplus_{rnd} b|$ is 1 ULP, which is the distance between two adjacent FP numbers around $a + b$. We use the definition of ULP as described by Overton [41]: $\forall t \in \mathbb{T}$, $ulp(t) = 2^{e-p+1}$, where e represents the exponent of t and p represents the available precision (*i.e.*, $p = 53$ for 64-bit doubles). We note that because $a \oplus_{rnd} b$ and $a \oplus_{RZ} b$ are faithful roundings of $a + b$, $pred(|a \oplus_{rnd} b|) = |a \oplus_{RZ} b|$ when $|a + b| < |a \oplus_{rnd} b|$. Using Overton's definition of $ulp(t)$, we can thus determine that when $|a + b| < |a \oplus_{rnd} b|$, $|(a \oplus_{rnd} b) - (a \oplus_{RZ} b)| = ulp(pred(|a \oplus_{rnd} b|))$. Given the definition of *ulp* and the relationship between $a \oplus_{RZ} b$ and $a \oplus_{rnd} b$, we define the function $RZA(a, b, a \oplus_{rnd} b)$ that returns $a \oplus_{RZ} b$ as follows.

$$RZA(a, b, a \oplus_{rnd} b) = \begin{cases} +0, & \text{if } a \text{ is } -b \\ a \oplus_{rnd} b - (-1)^{sign(a \oplus_{rnd} b)} \times ulp(pred(|a \oplus_{rnd} b|)), & \text{if } |a + b| < |a \oplus_{rnd} b| \\ a \oplus_{rnd} b & \text{else} \end{cases} \tag{8}$$

The equation above is a specification for obtaining the *RZ* result of FP addition given the operands and the original $a \oplus_{rnd} b$. However, the real arithmetic result of $a + b$ is not directly computable in FP arithmetic. Hence, we design Algorithm 1 to implement the specification in Equation 8.

1 **Function** ***RZA(double a, double b)***
2 **if** $bit(a)$ ***xor*** $bit(b) == 0x8000000000000000$ **then**
3 return +0.0;
4 **end**
5 *double* $s = a + b$;
6 **if** $|b| > |a|$ **then**
7 $a, b = b, a$;
8 **end**
9 *double* $z = s - a$;
10 *double* $t = b - z$;
11 **if** $(bit(t) << 1 \neq 0)$ ***and*** $(bit(t)$ ***xor*** $bit(s) \geq 0x8000000000000000)$ **then**
12 $s = s - (-1)^{sign(s)} \times ulp(pred(|s|))$;
13 **end**
14 return s;
15 **end**

Algorithm 1: Our high level algorithm for computing the *RZ* result for addition (*i.e.*, $a \oplus_{RZ} b$) given any rounding mode *rnd*. All FP operations are performed using double-precision FP arithmetic. Hence, $s = a \oplus_{rnd} b$, $z = (a \oplus_{rnd} b) \oplus_{rnd} (-a)$, and $t = b \oplus_{rnd} (-((a \oplus_{rnd} b) \oplus_{rnd} (-a)))$. Here, $bit(t)$ returns the IEEE-754 64-bit bit-string, and $sign(s)$ returns 0 if s is positive and 1 otherwise. We compute $s = s - (-1)^{sign(s)} \times ulp(pred(|s|))$ using bitwise operations, specifically by decrementing $bit(s)$ and then converting the resulting bit-pattern into a 64-bit FP number.

Emulating the *RZ* result for addition. Algorithm 1 describes the procedure to compute $a \oplus_{RZ} b$ given two 64-bit non-*NaN*, non-infinity FP operands a and b and the rounding mode *rnd*. Using lines 2 through 4, the algorithm applies the first case of Equation 8 and handles the cases where $a \oplus_{rnd} b \neq a \oplus_{RZ} b$ because a is $-b$ and $rnd = RD$. The expressions $bit(a)$ and $bit(b)$ in line 2 provide the bit-strings of the numbers a and b in the 64-bit IEEE-754 representation as 64-bit integers. The bit-strings of negative numbers are greater than or equal to $0x8000000000000000$ (*i.e.*, the sign bit is 1). When two terms have different signs (*i.e.*, $sign(a) = 1$ and $sign(b) = 0$, or vice versa), the sign bit of the **xor** result will be set to 1. The condition $bit(a)$ **xor** $bit(b) == 0x8000000000000000$ indicates that a and b have the same absolute value but with different signs, the exact circumstance under which $a \oplus_{RZ} b$ should always return +0. The remaining steps of the algorithm address the second case of Equation 8, in which $a + b$ is not exactly representable and $|a + b| < |a \oplus_{rnd} b|$. These steps are based on the following two theorems. Our extended technical report provides proofs for these two theorems along with a proof of *RZA* [43].

Theorem 4. *Let a and b be two non-NaN, non-infinity floating-point numbers such that $a \oplus_{rnd} b$ does not overflow for any rounding mode. If $a + b - (a \oplus_{rnd} b) \neq 0$, $a \oplus_{rnd} b$ and $a + b - (a \oplus_{rnd} b)$ have different signs if and only if $|a + b| < |a \oplus_{rnd} b|$.*

From Theorem 4 we conclude that testing whether $a + b - (a \oplus_{rnd} b)$ is a non-zero value, which would indicate $a + b \neq a \oplus_{rnd} b$, and whether its sign differs from that of $a \oplus_{rnd} b$ is sufficient for determining if $a \oplus_{rnd} b$ needs to be adjusted according to Equation 8. The focal point of Algorithm 1 is checking the condition $a + b \neq a \oplus_{rnd} b$ without having direct access to $a + b$. Lines 5 through 10 in Algorithm 1 employ the steps in Dekker's *FastTwoSum* algorithm [19] to compute the value t, which is an approximation of the rounding error of the initial FP addition (*i.e.* $a+b-(a \oplus_{rnd} b)$). Since all operations are performed in FP arithmetic, the outputs of the subtractions in line 9 and 10 are subject to rounding. As a result, $z = s \oplus_{rnd} (-a) = (a \oplus_{rnd} b) \oplus_{rnd} (-a)$ and $t = b \oplus_{rnd} (-z) = b \oplus_{rnd} (-((a \oplus_{rnd} b) \oplus_{rnd} (-a)))$. Proving the viability of *RZA* thus requires

affirming that t is an appropriate proxy for $a + b - (a \oplus_{rnd} b)$ with respect to applying Theorem 4 under all possible modes of *rnd*. Specifically, t must be sufficient for the purposes of assessing whether $a + b - (a \oplus_{rnd} b) \neq 0$ and $sign(a + b - (a \oplus_{rnd} b)) \neq sign(a \oplus_{rnd} b)$.

An important point to note is that t exactly represents the error $a + b - (a \oplus_{rnd} b)$ when the rounding mode is *RN* and is a faithfully rounded FP value of the error $a + b - (a \oplus_{rnd} b)$ for other rounding modes [6]. The suitability of t as an approximation of $a + b - (a \oplus_{rnd} b)$ follows directly from Theorem 5.

Theorem 5. *Let $t \in \mathbb{T}$ be a faithful rounding of the error $a + b - (a \oplus_{rnd} b) \in \mathbb{R}$. Then, t is neither $+0$ nor -0 if and only if $a + b - (a \oplus_{rnd} b) \neq 0$.*

Given that t is a faithful rounding of the real error $a + b - (a \oplus_{rnd} b)$ (see our proofs in the extended technical report [43] and [6]), Theorem 5 asserts that the comparison $(bit(t) << 1) \neq 0$, which checks if t is neither $+0$ nor -0, is sufficient for determining if $a + b - (a \oplus_{rnd} b) \neq 0$. Given Theorem 5, $a + b - (a \oplus_{rnd} b) \neq 0$ must hold when $(bit(t) << 1) \neq 0$ is true. Subsequently, $(bit(t) << 1) \neq 0$ implies t is a faithful rounding of a non-zero real number, and thus the equality $t = rnd(a + b - (a \oplus_{rnd} b))$ holds for all rounding functions $rnd \in \{RN, RZ, RD, RU\}$. Hence, one can apply Lemma 4, which states the preservation of sign with faithful rounding for non-zero real numbers, to conclude that $(bit(t) << 1) \neq 0$ implies $sign(t) = sign(a + b - (a \oplus_{rnd} b))$. The expression $(bit(t)$ **xor** $bit(s)) \geq 0x8000000000000000$, which checks if $sign(t) \neq sign(s = a \oplus_{rnd} b)$, can thus accurately assess if $sign(a + b - (a \oplus_{rnd} b)) \neq sign(a \oplus_{rnd} b)$ when $(bit(t) << 1) \neq 0$. Therefore, the conditions in line 11 of Algorithm 1 associated with t are sufficient for the purposes of determining if $a + b - (a \oplus_{rnd} b) \neq 0$ and $sign(a + b - (a \oplus_{rnd} b)) \neq sign(a \oplus_{rnd} b)$. When such conditions are met, our algorithm *RZA* modifies the value of $a \oplus_{rnd} b$ through line 12. In summary, our algorithm identifies the presence of rounding error, determines whether the error indicates $|a+b| < |a \oplus_{rnd} b|$, and accordingly adjusts $a \oplus_{rnd} b$ to match $a \oplus_{RZ} b$. Figure 4a provides an example of how our algorithm *RZA* adjusts a non-*RZ* FP addition result based on Algorithm 1.

Relationship between $\otimes_{RZ}$ and other $\otimes_{rnd}$. Our objective for handling $a \otimes_{rnd} b$ is to adjust its value whenever it is not equal to $a \otimes_{RZ} b$. A notable difference between FP addition and multiplication is that the latter operation does not entail any corner cases involving $+0$ or -0 - whenever $a \times b = 0$, $a \otimes_{rnd} b$ displays the same behavior across all *rnd*. Due to this distinction, the only cases of concern are those in which $a \times b$ is not exactly representable. As detailed in our premise for the algorithm *RZA*, $a \otimes_{rnd} b$ can differ from $a \otimes_{RZ} b$ when the two numbers are distinct FP neighbors of a product $a \times b$ that is not exactly representable. Because $a \otimes_{RZ} b$ should always return the FP neighbor of $a \times b$ with the smaller absolute value, $a \otimes_{rnd} b$ must be the FP neighbor with the larger absolute value for $a \otimes_{RZ} b \neq a \otimes_{rnd} b$ to hold. Given how a real number's FP neighbors are defined in Equations 1 and 2 (Figure 2), $a \otimes_{RZ} b \neq a \otimes_{rnd} b$ implies $|a \otimes_{RZ} b| = |a \otimes_{rnd} b| - ulp(pred(|a \otimes_{rnd} b|))$. We define the specification of $RZM(a, b, a \otimes_{rnd} b)$ that returns $a \otimes_{RZ} b$ under all rounding modes as follows.

$$RZM(a, b, a \otimes_{rnd} b) = \begin{cases} a \otimes_{rnd} b - (-1)^{sign(a \otimes_{rnd} b)} \times ulp(pred(|a \otimes_{rnd} b|)), & \text{if } |a \times b| < |a \otimes_{rnd} b| \\ a \otimes_{rnd} b & \text{else} \end{cases} \tag{9}$$

Emulating the *RZ* result for multiplication. As is the case with FP addition, simulating the *RZ* result for FP multiplication is non-trivial because the real value output $a \times b$ is not directly observable. Algorithm 2 describes our approach to computing $a \otimes_{RZ} b$ given two non-*NaN*, non-infinity 64-bit FP numbers a and b for which the product doesn't cause overflow. Much like Algorithm 1, it computes a faithful rounding of the error in the original FP operation to assess whether the original product needs to be adjusted to match the *RZ* result. As we describe in our proofs for *RZM*, however, a

1 **Function** ***RZM (double a, double b)***
2 $\quad$ *double* $m = a \times b$;
3 $\quad$ *double* $c1 = fma(a, b, -m)$;
4 $\quad$ *double* $c2 = fma(-a, b, m)$;
5 $\quad$ **if** $bit(c1) \neq bit(c2)$ ***and*** $(bit(c1)$ ***xor*** $bit(m)) \geq 0x8000000000000000$ **then**
6 $\quad\quad$ $m = m - (-1)^{sign(m)} \times ulp(pred(|m|))$;
7 $\quad$ **end**
8 $\quad$ return m;
9 **end**

Algorithm 2: Our algorithm for computing $a \otimes_{RZ} b$ given any rounding mode *rnd*. All FP operations are performed using double-precision FP arithmetic. Hence, $m = a \otimes_{rnd} b$, $c1 = fma_{rnd}(a, b, -m)$, and $c2 = fma_{rnd}(-a, b, m)$. Given non-*NaN*, non-infinity operands a, b, and c, the fused-multiply-add instruction $fma_{rnd}(a, b, c)$ returns a faithful rounding of $a \times b + c$. Here, $c1$ is a faithful rounding of $a \times b - m$ and $c2$ is a faithful rounding of $(-a) \times b + m$. The remaining details are analogous to those found in Algorithm 1.

faithful rounding of $a \times b - (a \otimes_{rnd} b)$ could be equal to +0 or −0 even when $a \times b - (a \otimes_{rnd} b) \neq 0$ because rounding error induced by multiplication is susceptible to underflow. We therefore rely on the following two theorems to confirm the presence and nature of the rounding error in $a \otimes_{rnd} b$.

Theorem 6. *Let a and b be two non-NaN, non-infinity floating-point numbers such that $a \otimes_{rnd} b$ does not overflow for any rounding mode. If $a \times b - (a \otimes_{rnd} b) \neq 0$, $a \otimes_{rnd} b$ and $a \times b - (a \otimes_{rnd} b)$ have different signs if and only if $|a \times b| < |a \otimes_{rnd} b|$.*

Theorem 7. *Let a and b be two non-NaN, non-infinity floating-point numbers such that $a \otimes_{rnd} b$ does not overflow for any rounding mode. Let $bit(f)$ be a function that returns the bit-string of any floating-point number f. Then, for any rounding mode rnd, $bit(fma_{rnd}(a, b, -(a \otimes_{rnd} b))) \neq bit(fma_{rnd}(-a, b, a \otimes_{rnd} b))$ if and only if $a \times b - (a \otimes_{rnd} b) \neq 0$.*

Lines 2 and 3 in Algorithm 2 compute $m = a \otimes_{rnd} b$ and $c1 = fma_{rnd}(a, b, -(a \otimes_{rnd} b))$. The output of a fused-multiply-add (FMA) operation in the form of $fma_{rnd}(a, b, c)$ is the result of performing a faithfully rounded FP addition $\oplus_{rnd}$ using the operands $a \times b$ and c with no intermediate rounding for the multiplication. Based on the rules of $\oplus_{rnd}$ detailed in Equation 6 (Figure 3), $c1 = fma_{rnd}(a, b, -m)$ is guaranteed to be a faithful rounding of $a \times b - m = a \times b - (a \otimes_{rnd} b)$. Line 4 of the algorithm computes $c2 = fma_{rnd}(-a, b, m)$, which makes $c2$ a faithful rounding of $(-a) \times b + m = -(a \times b) + (a \otimes_{rnd} b)$. The algorithm computes $c2$ to compare its bit-pattern against that of $c1$ to utilize Theorem 7, which guarantees that $a \times b - (a \otimes_{rnd} b) \neq 0$ whenever the condition $bit(c1) \neq bit(c2)$ in line 5 is true.

Once it is established that $a \times b - (a \otimes_{rnd} b) \neq 0$, Theorem 6 affirms that examining the sign of $a \times b - (a \otimes_{rnd} b)$ relative to $a \otimes_{rnd} b$ is sufficient for assessing whether $|a \times b| < |a \otimes_{rnd} b|$. Here, $c1$ is a faithful rounding of a non-zero real value $a \times b - (a \otimes_{rnd} b)$. From Lemma 4, the sign preserving properties of faithful rounding with respect to non-zero real numbers ensures that $sign(c1) = sign(a \times b - (a \otimes_{rnd} b))$. Under such conditions, *RZM* can thus apply Theorem 6 and use $c1$ as a proxy for $a \times b - (a \otimes_{rnd} b)$. Our algorithm *RZM* applies Theorem 6 by confirming $a \times b - (a \otimes_{rnd} b) \neq 0$ through the condition $bit(c1) \neq bit(c2)$ and checking if $sign(c1) \neq sign(m = a \otimes_{rnd} b)$ through the expression $(bit(c1)$ **xor** $bit(m)) \geq 0x8000000000000000$. In conclusion, Theorems 6 and 7 in conjunction with Lemma 4 guarantee that the conditions $bit(c1) \neq bit(c2)$ and $(bit(c1)$ **xor** $bit(m)) \geq 0x8000000000000000$ are appropriate for testing whether $a \times b - (a \otimes_{rnd} b) \neq 0$, $sign(a \times b - (a \otimes_{rnd} b)) \neq sign(a \otimes_{rnd} b)$, and $|a \times b| < |a \otimes_{rnd} b|$. When such conditions are met, line 6 of Algorithm 2 produces $a \otimes_{RZ} b$ by adjusting the value of $a \otimes_{rnd} b$ in accordance with Equation 9. We provide detailed proofs for the algorithm *RZM* and its associated theorems in our

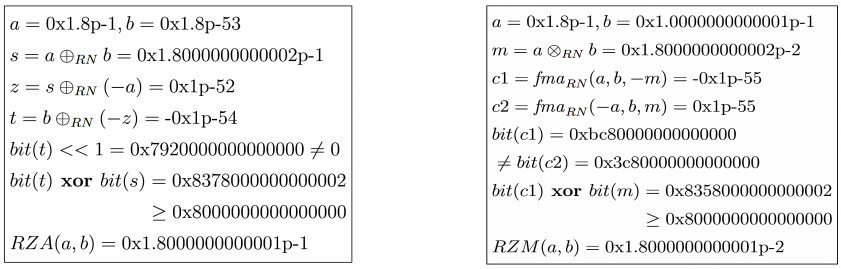


Fig. 4. (a) An illustration of how our *RZA* function adjusts the output of a double-precision FP addition result $s = a \oplus_{rnd} b$ when $s \neq a \oplus_{RZ} b$. (b) An illustration of how our *RZM* function adjusts the output of a double-precision FP multiplication result $m = a \otimes_{rnd} b$ when $m \neq a \otimes_{RZ} b$. All FP values are shown as hex floats. All integers (*e.g.*, $bit(t)$) are shown in hexadecimal representation.

extended technical report [43]. Figure 4b provides an example of how *RZM* adjusts a non-*RZ* FP multiplication result based on Algorithm 2.

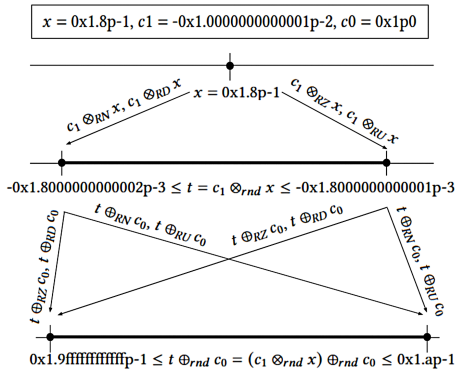
Changes to the RLibm pipeline. The rounding-invariant outputs approach requires minimal changes to the RLibm pipeline. The only change needed is to set the rounding mode to *RZ* instead of *RN* when performing range reduction, reduced interval generation, and polynomial generation. For the resultant implementations, this new method entails replacing all FP additions and multiplications, which are the only FP operations in the code that induce rounding-related variability, with *RZA* and *RZM*, respectively. These modifications lead to implementations that can produce correctly rounded results for all inputs across multiple representations and rounding modes without requiring explicit changes to the application-level rounding mode.

3.2 Rounding Independence by Deducing Rounding-Invariant Input Bounds

The underlying principle behind our round-to-zero emulation in Section 3.1 is the internal enforcement of a single rounding mode (*i.e.*, *RZ*) such that the range reduction, polynomial evaluation, and output compensation steps in a given RLibm implementation will always produce the same outputs regardless of the invoking application's rounding mode. This approach requires augmenting each FP addition and multiplication with operations designed to produce the *RZ* result, which can add some overhead to the final implementations. To address this issue, we propose an alternative method that bypasses the overheads required for accomplishing **rounding mode-invariant outputs** by deducing the **rounding mode-invariant bounds** on the polynomial evaluation and output compensation results for the reduced inputs encountered. This method moves the overhead from the final math library implementation to the process of generating it.

The range reduction, polynomial evaluation, and output compensation processes involve FP arithmetic and are potentially sensitive to rounding modes. Among them, range reduction is the least sensitive because the algorithms used generally produce rounding mode-independent results. We make all range reduction algorithms used with our new approach produce results that conform to *RZ*. We enforce this requirement in the resultant implementations by replacing all rounding mode-dependent FP additions and multiplications used for range reduction with the functions *RZA* and *RZM* discussed in Section 3.1.

Accounting for rounding mode induced variability. To account for the variability in the FP evaluation of polynomials and output compensation functions under different rounding modes, we deduce new bounds for the reduced intervals given to the polynomial generator and for the



(a)

$setround(RD)$
$t1\_lb = c_2 \otimes x$
$t2\_lb = t1\_lb \oplus c_1$
$setround(RU)$
$t1\_ub = c_2 \otimes x$
$t2\_ub = t1\_ub \oplus c_1$
$setround(RD)$
$t3\_lb = \min(t2\_lb \otimes x, t2\_ub \otimes x)$
$poly\_lb = t3\_lb \oplus c_0$
$setround(RU)$
$t3\_ub = \max(t2\_lb \otimes x, t2\_ub \otimes x)$
$poly\_ub = t3\_ub \oplus c_0$

(b)

Fig. 5. (a) The figure illustrates the intuition behind how our rounding-invariant input bounds approach identifies the lower and upper bounds of a polynomial output across all rounding modes using a polynomial of the form $P(x) = (c_1 \otimes_{rnd} x) \oplus_{rnd} c_0$ as an example. We highlight that the lower bounds for both the $\otimes_{rnd}$ and $\oplus_{rnd}$ operations involve the *RD* results. Similarly, the upper bounds involve the *RU* results. (b) The figure illustrates the steps the rounding-invariant input bounds approach takes to compute the lower bound (*poly_lb*) and upper bound (*poly_ub*) of a polynomial $P(x) = (((c_2 \otimes_{rnd} x) \oplus_{rnd} c_1) \otimes_{rnd} x) \oplus_{rnd} c_0$ during the library generation process.

expected results of the polynomial approximations. Our key idea is to consider the polynomial approximations and output compensation functions as returning not a single value, but rather a **range of values**. Representing the output of a sequence of FP operations as a range of values (*i.e.*, an interval) accounts for the variability in the results stemming from the different rounding modes. We use interval arithmetic to identify bounds for such ranges, which the new pipeline uses during the reduced interval and polynomial generation stages. We construct this new strategy based on the insight that the round-down (*RD*) and round-up (*RU*) results serve as the lower and upper bounds for a given FP operation's output range across various rounding modes. Figure 5a illustrates the intuition behind this method. We now describe our approach for deducing the bounds of polynomial evaluation and output compensation results in the context of our reduced interval and polynomial generation processes.

Accounting for rounding induced variability in polynomial evaluation. Let $P(x) = \sum_{i=0}^{d} c_i x^i = c_d x^d + c_{d-1}x^{d-1} + \cdots + c_2 x^2 + c_1 x + c_0$ where $P(x) \in \mathbb{R}$ and $\forall i, c_i \in \mathbb{T}$ represent the typical polynomial approximation considered in the RLIBM project. If a polynomial $P(x)$ has at least one non-constant term, the result of evaluating $P(x)$ on a given input x via real arithmetic can be expressed as either $P(x) = P_1(x) \times P_2(x)$ or $P(x) = P_1(x) + P_2(x)$ where $P_1(x), P_2(x) \in \mathbb{R}$ represent the appropriate intermediate terms. If the evaluation is performed using FP arithmetic, the final output may differ depending on the order in which operations are performed since FP operations are non-associative. For our purposes, we assume throughout the paper that every polynomial is associated with a unique, predetermined FP evaluation scheme. The variability induced by the different orderings that are possible when translating a polynomial to a sequence of FP operations is outside the scope of this paper. Even when evaluating $P(x)$ using a fixed order of FP operations, one may observe different outputs depending on the rounding rule applied to each operation. Henceforth, we use $P(x)$ to represent the final output obtained from evaluating a given polynomial using n ordered FP operations each subject to a rounding rule $rnd_i \in \{RN, RZ, RD, RU\}$. Recognizing the impact of rounding on the final output, we formulate $P(x)$ through the following definition.

Definition 8.

$$P(x) = \begin{cases} c, & \text{if } P(x) \text{ is a FP constant} \\ P_1(x) \oplus_{RN} P_2(x) & \text{if } flop_n = \oplus \text{ and } rnd_n = RN \\ P_1(x) \oplus_{RZ} P_2(x) & \text{if } flop_n = \oplus \text{ and } rnd_n = RZ \\ \cdots & \\ P_1(x) \otimes_{RD} P_2(x) & \text{if } flop_n = \otimes \text{ and } rnd_n = RD \\ P_1(x) \otimes_{RU} P_2(x) & \text{if } flop_n = \otimes \text{ and } rnd_n = RU \end{cases}$$

The definition of $P(x)$ presented above is intended to be recursive: $P_1(x)$ and $P_2(x)$ both represent the result of a specific sequence of intermediate FP operations that are each subject to different rounding rules. Here, rnd_n represents the rounding rule employed by the last FP addition or multiplication, which we collectively denote as $flop_n$. The subscript in rnd_n indicates that each intermediate FP operation $flop_i$ is executed under the corresponding rounding mode rnd_i. Typically, the rounding rule used by each FP operation would be fixed to the application's rounding mode. Note that for the purposes of defining $P(x)$ in this paper, we do not require that all instances of rnd_i are identical.

Given a set of reduced inputs and their reduced intervals, $(x'_i, [l'_i, h'_i])$, the goal of RLibm's polynomial generator is to find a polynomial with an FP output $P(x)$ as defined under Definition 8 such that $l'_i \leq P(x'_i) \leq h'_i$ holds for all inputs x_i. In the original RLibm approach, every intermediate FP output encountered while computing $P(x')$ for a given reduced input is the result of rounding via RN (*i.e.*, $\forall i, rnd_i = RN$). In our rounding-invariant outputs approach, every $rnd_i = RZ$. In both cases, $P(x'_i)$ would be a single value given the absence of rounding-induced variability in each intermediate FP output. Figure 6(a) shows the constraints provided to the polynomial generator when the output of the polynomial evaluation is a single value.

Constraints for polynomial generation when outputs are treated as ranges of values. Given the same values for $P_1(x'_i)$ and $P_2(x'_i)$, the final value of $P(x'_i)$ may differ depending on the choice of rnd_n. Thus, evaluating a single version of $P(x'_i)$ derived from a fixed rnd_n provides limited information. For example, it could be the case that the constraint $l'_i \leq P(x'_i) \leq h'_i$ holds for a reduced input x'_i and its reduced interval $[l'_i, h'_i]$ when $rnd_n = RN$ but not when $rnd_n = RU$. The problem is further exacerbated by the fact that a fully-evaluated $P(x'_i)$ is the result of a specific sequence of intermediate roundings rnd_i. Different combinations of the intermediate rnd_i instances prior to rnd_n could lead to different values for $P_1(x'_i)$ and $P_2(x'_i)$. To account for this variability, we adopt a new viewpoint regarding $P(x)$. *We no longer consider the final polynomial evaluation output $P(x)$ for a particular reduced input x'_i to be a single FP number resulting from a fixed sequence of rnd_i. Rather, we treat it as a function of the n variable instances of rnd_i, the output range of which we denote* $[\underline{P(x'_i)}, \overline{P(x'_i)}]$. In the case that $l'_i \leq \underline{P(x'_i)} \leq \overline{P(x'_i)} \leq h'_i$, one can conclude that regardless of the rounding rule rnd_i used for each operation $flop_i$, the final result of $P(x'_i)$ will satisfy its associated constraint. Therefore, the goal of the new polynomial generator would be to find a polynomial such that $l'_i \leq \underline{P(x'_i)} \leq \overline{P(x'_i)} \leq h'_i$ for as many x_i's as possible. Figure 6(b) displays how the polynomial generator would apply constraints to a candidate polynomial approximation under the new approach.

Defining the bounds of polynomial evaluation. Computing $\underline{P(x)}$ and $\overline{P(x)}$ for an arbitrary input x necessitates identifying an appropriate rounding rnd_i to apply to each FP operation and identifying operands along with their respective interval bounds. We use the monotonicity properties of faithfully rounded FP operations and identify the interval bounds using interval arithmetic rules presented in Lemmas 9 and 10. Given the definition of $P(x)$ in Definition 8, we now present the definitions of $\underline{P(x)}$ and $\overline{P(x)}$ through Theorems 9 and 10 and their proofs.

$-1.5 \le P(0.25) \le -1.375 \quad 0.5 \le P(1.75) \le 0.625$	$-1.5 \le \underline{P(0.25)} \le \overline{P(0.25)} \le -1.375 \quad 0.5 \le \underline{P(1.75)} \le \overline{P(1.75)} \le 0.625$
$-0.75 \le P(0.5) \le -0.625 \quad 0.625 \le P(2.0) \le 0.75$	$-0.75 \le \underline{P(0.5)} \le \overline{P(0.5)} \le -0.625 \quad 0.625 \le \underline{P(2.0)} \le \overline{P(2.0)} \le 0.75$
$-0.375 \le P(0.75) \le -0.25 \quad 0.875 \le P(2.5) \le 1.0$	$-0.375 \le \underline{P(0.75)} \le \overline{P(0.75)} \le -0.25 \quad 0.875 \le \underline{P(2.5)} \le \overline{P(2.5)} \le 1.0$
$0.125 \le P(1.25) \le 0.25 \quad 1.0 \le P(3.0) \le 1.125$	$0.125 \le \underline{P(1.25)} \le \overline{P(1.25)} \le 0.25 \quad 1.0 \le \underline{P(3.0)} \le \overline{P(3.0)} \le 1.125$
$0.375 \le P(1.5) \le 0.5 \quad 1.25 \le P(3.5) \le 1.375$	$0.375 \le \underline{P(1.5)} \le \overline{P(1.5)} \le 0.5 \quad 1.25 \le \underline{P(3.5)} \le \overline{P(3.5)} \le 1.375$
(a) Constraints when the polynomial returns a single value	(b) Constraints when the polynomial returns a range of values with our new approach

Fig. 6. (a) Constraints used to produce the correctly rounded results when $P(x'_i)$ is considered a single FP value as in the original RLIBM and our rounding-invariant outputs approaches. (b) Constraints used to produce the correctly rounded results when $P(x'_i)$ is considered to be any FP value in the range $[\underline{P(x'_i)}, \overline{P(x'_i)}]$.

Theorem 9.

$$\underline{P(x)} = \begin{cases} c, & \text{if FP constant} \\ \underline{P_1(x)} \oplus_{RD} \underline{P_2(x)}, & \text{if } flop_n = \oplus \\ min(\underline{P_1(x)} \otimes_{RD} \underline{P_2(x)}, \underline{P_1(x)} \otimes_{RD} \overline{P_2(x)}, \overline{P_1(x)} \otimes_{RD} \underline{P_2(x)}, \overline{P_1(x)} \otimes_{RD} \overline{P_2(x)}) & \text{if } flop_n = \otimes \end{cases}$$

Theorem 10.

$$\overline{P(x)} = \begin{cases} c, & \text{if FP constant} \\ \overline{P_1(x)} \oplus_{RU} \overline{P_2(x)}, & \text{if } flop_n = \oplus \\ max(\underline{P_1(x)} \otimes_{RU} \underline{P_2(x)}, \underline{P_1(x)} \otimes_{RU} \overline{P_2(x)}, \overline{P_1(x)} \otimes_{RU} \underline{P_2(x)}, \overline{P_1(x)} \otimes_{RU} \overline{P_2(x)}) & \text{if } flop_n = \otimes \end{cases}$$

Proof. Given the symmetry between the definitions of $\underline{P(x)}$ and $\overline{P(x)}$ presented above, we primarily focus on proving Theorem 9. The first case in which $\overline{P(x)}$ is a FP constant requires no proof as it does not involve any FP operations and $P(x) = c$ in this case across any rounding mode. We thus move on to proving the definition for $\underline{P(x)}$ when $flop_n = \oplus$. Let $\underline{P(x)} = p_1^* \oplus_{rnd_n^*} p_2^*$ where $p_1^* \in [\underline{P_1(x)}, \overline{P_1(x)}]$, $p_2^* \in [\underline{P_2(x)}, \overline{P_2(x)}]$, and $rnd_n^* \in \{RN, RZ, RD, RU\}$. We prove the correctness of Theorem 9's definition for this case by affirming that $rnd_n^* = RD$ and that this equality implies $p_1^* = \underline{P_1(x)}$ and $p_2^* = \underline{P_2(x)}$. Suppose that there exists some $rnd_n^* \in \{RN, RZ, RU\}$ such that $p_1^* \oplus_{rnd_n^*} p_2^* < p_1^* \oplus_{RD} p_2^*$. Such an assumption would directly violate the bounds of $\oplus_{rnd}$ established in Lemma 5, thereby establishing $rnd_n^* = RD$. Having established $rnd_n^* = RD$, we also prove that $rnd_n^* = RD$ implies $p_1^* = \underline{P_1(x)}$ and $p_2^* = \underline{P_2(x)}$ via contradiction. Suppose $rnd_n^* = RD$ and there exist a p_1^* and a p_2^* such that $p_1^* \oplus_{RD} p_2^* < \underline{P_1(x)} \oplus_{RD} \underline{P_2(x)}$. By the rules of interval addition detailed in Lemma 9, it must be the case that $\underline{P_1(x)} + \underline{P_2(x)} \le p_1^* + p_2^*$. The lower bound of the real arithmetic sum indicates that the existence of a pair of operands p_1^* and p_2^* such that $p_1^* \oplus_{RD} p_2^* < \underline{P_1(x)} \oplus_{RD} \underline{P_2(x)}$ would directly contradict the monotonic properties of $\oplus_{RD}$ detailed in Lemma 7. The resulting contradiction indicates that $rnd_n^* = RD$ implies $\underline{P(x)} = \underline{P_1(x)} \oplus_{RD} \underline{P_2(x)}$, thereby concluding our proof for the case in which $flop_n = \oplus$.

Our proof for the case in which $flop_n = \otimes$ largely follows the same structure. Let $\underline{P(x)} = p_1^* \otimes_{rnd_n^*} p_2^*$ where $p_1^* \in [\underline{P_1(x)}, \overline{P_1(x)}]$, $p_2^* \in [\underline{P_2(x)}, \overline{P_2(x)}]$, and $rnd_n^* \in \{RN, RZ, RD, RU\}$. Similar to how we use Lemma 5 to show that $\underline{P(x)}$ involves $\oplus_{RD}$ when $flop_n = \oplus$, one can leverage $\otimes_{RD}$'s role in defining the lower bound of FP multiplication detailed in Lemma 6 to conclude that $rnd_n^* = RD$ when $flop_n = \otimes$. We thus focus on establishing that there cannot exist any pair of operands (p_1^*, p_2^*) outside the Cartesian product $\{\underline{P_1(x)}, \overline{P_1(x)}\} \times \{\underline{P_2(x)}, \overline{P_2(x)}\}$ such that $p_1^* \otimes_{RD} p_2^* < min(\underline{P_1(x)} \otimes_{RD} \underline{P_2(x)}, \underline{P_1(x)} \otimes_{RD} \overline{P_2(x)}, \overline{P_1(x)} \otimes_{RD} \underline{P_2(x)}, \overline{P_1(x)} \otimes_{RD} \overline{P_2(x)})$. Henceforth, we refer to the right hand side of $\underline{P(x)}$'s definition for the $flop_n = \otimes$ case as fp_prod_{min}. One can deduce

from the rules of interval multiplication detailed in Lemma 10 that for any $p_1 \in [\underline{P_1(x)}, \overline{P_1(x)}]$ and $p_2 \in [\underline{P_2(x)}, \overline{P_2(x)}]$, the real arithmetic product $p_1 \times p_2$ is equal to or greater than $\min(\underline{P_1(x)} \times \underline{P_2(x)}, \underline{P_1(x)} \times \overline{P_2(x)}, \overline{P_1(x)} \times \underline{P_2(x)}, \overline{P_1(x)} \times \overline{P_2(x)})$, which we subsequently refer to as $real\_prod_{min}$. Let $p^*_{1,real}$ and $p^*_{2,real}$ be two FP numbers such that $(p^*_{1,real}, p^*_{2,real}) \in \{\underline{P_1(x)}, \overline{P_1(x)}\} \times \{\underline{P_2(x)}, \overline{P_2(x)}\}$ and $p^*_{1,real} \times p^*_{2,real} = real\_prod_{min}$. Given that $p^*_{1,real} \times p^*_{2,real} \leq p_1 \times p_2$ for any $p_1 \in [\underline{P_1(x)}, \overline{P_1(x)}]$ and $p_2 \in [\underline{P_2(x)}, \overline{P_2(x)}]$, the monotonic properties of $\otimes_{RD}$ detailed in Lemma 8 indicates that the bound $p^*_{1,real} \otimes_{RD} p^*_{2,real} \leq p_1 \otimes_{RD} p_2$ must hold for all pairs of p_1 and p_2, including those outside the aforementioned Cartesian product. This concludes our proof that no pair of operands $(p^*_1, p^*_2) \notin \{\underline{P_1(x)}, \overline{P_1(x)}\} \times \{\underline{P_2(x)}, \overline{P_2(x)}\}$ can lead to an FP product less than fp_prod_{min} when $rnd^*_n = RD$ as well as our proof that $\underline{P(x)} = fp\_prod_{min}$ when $flop_n = \otimes$.

The proof for Theorem 10 largely mirrors that of Theorem 9 due to the symmetry between them. Akin to the manner in which Lemmas 5 and 6 were applied to justify $\oplus_{RD}$'s and $\otimes_{RD}$'s roles in defining $\underline{P(x)}$, the same lemmas can be applied to validate the use of $\oplus_{RU}$ and $\otimes_{RU}$ in defining $\overline{P(x)}$. As is the case with Theorem 9, one can apply the interval addition and multiplication rules detailed in Lemmas 9 and 10 to justify the operands associated with $\oplus_{RU}$ and $\otimes_{RU}$ in Theorem 10's definition for the non-trivial cases. □

The polynomial generator from the RLibm pipeline now has to solve the stricter constraints resulting from Theorems 9 and 10 to ensure correctness while incurring no overheads in the final implementations. Figure 5b provides an example of the steps taken by our polynomial generator to compute the lower and upper bounds of each FP operation's result under various rounding modes. It uses the *RD* mode to compute the lower bound and then uses the *RU* mode to compute the upper bound based on the specifications in Theorems 9 and 10.

Accounting for rounding induced variability during reduced interval generation. The polynomials produced by RLibm's generators are approximations over the variable x' drawn from a reduced version of the target function's original domain (*i.e.*, its reduced range). Consequently, each polynomial requires a set of operations that form the output compensation function, which maps the polynomial outputs to the original target range. Within the RLibm pipeline, the reduced interval generator accounts for the effects of evaluating an output compensation function via FP arithmetic, the result of which we denote $OC(y', x)$ for any given y'. The goal of the reduced interval generator is to find for each input x's reduced input x' the maximal interval $[l', h']$ such that $\forall y' \in [l', h'], l \leq OC(y', x) \leq h$. The values l and h denote the bounds of x's target rounding interval. By using the resultant $[l', h']$ as a constraint for the polynomial generator, the RLibm pipeline ensures for a given reduced input x' that a polynomial with an output $P(x')$ that satisfies $l' \leq P(x') \leq h'$ also satisfies $l \leq OC(P(x'), x) \leq h$. By satisfying the constraint $l \leq OC(P(x'), x) \leq h$, $OC(P(x'), x)$ is guaranteed to correctly round to the oracle result for the original input x.

The reduced interval generators in the original RLibm pipeline evaluate output compensation functions with the rounding mode set to *RN*, thereby only accounting for the effects of rounding under a specific mode. We note that the final post-output compensation result $OC(y', x)$ for a specific instance of y' can end up as a different number within the range $[\underline{OC(y', x)}, \overline{OC(y', x)}]$ depending on the rounding rule applied to each component FP operation. As such, the goal of the interval generator under the rounding-invariant input bounds approach is to find for each x' the maximal $[l', h']$ such that $\forall y' \in [l', h'], l \leq \min_{y' \in [l', h']} \underline{OC(y', x)} \leq \max_{y' \in [l', h']} \overline{OC(y', x)} \leq h$. We note that every output compensation function used by RLibm can be expressed using only addition and multiplication. In essence, the output compensation functions with respect to the reduced

inputs x' are polynomial compositions of the form $\hat{y} = Q(P(x'))$ evaluated with FP additions and multiplications. The reduced interval generator can therefore compute $min_{y' \in [l', h']} \underline{OC(y', x)}$ and $\max_{y' \in [l', h']} \overline{OC(y', x)}$ by directly applying Theorems 9 and 10. By producing constraints for the polynomial generator in this manner, the reduced interval generator under the rounding-invariant input bounds approach can validate the following statement: for all $P(x')$ such that $l' \leq \underline{P(x')} \leq \overline{P(x')} \leq h'$, $l \leq \min_{P(x') \in [\underline{P(x')}, \overline{P(x')}]} \underline{OC(P(x'), x)} \leq \max_{P(x') \in [\underline{P(x')}, \overline{P(x')}]} \overline{OC(P(x'), x)} \leq h$ holds by default. In summary, the reduced interval generator for the rounding-invariant input bounds approach applies Theorems 9 and 10 to ensure that an output compensation function can map polynomial outputs to their final target rounding intervals under any rounding mode without requiring rounding mode adjustments in the final implementations.

Correctness under all faithful rounding modes. As previously established, $\underline{P(x)}$ is the smallest possible value of $P(x)$ when any of the four rounding modes of concern can be applied to any of the intermediate operations. $\overline{P(x)}$ is the upper bound counterpart. The definitions of $\underline{P(x)}$ and $\overline{P(x)}$ are founded on Lemma 1, which establishes *RD*'s and *RU*'s roles as the lower and upper bounds of the rounding modes considered. One must note that their roles as bounds are not limited to the set $\{RN, RZ, RD, RU\}$ as Lemma 1 pertains to all faithful rounding modes. Naturally, $\oplus_{RD}$ and $\oplus_{RU}$ (or $\otimes_{RD}$ and $\otimes_{RU}$) each define the lower and upper bounds of any faithful FP addition (or multiplication) operation given the same pair of operands. We note that the range $[\underline{P(x)}, \overline{P(x)}]$ subsumes every possible assignment of either $\oplus_{RD}$, $\oplus_{RU}$, $\otimes_{RD}$, or $\otimes_{RU}$ to each FP operation composing the evaluation of $P(x)$. As such, $\underline{P(x')}$ and $\overline{P(x')}$ are the bounds for all the possible values of $P(x')$ when the rnd_i applied to each operation could be **ANY** faithful rounding mode (i.e. round-away-from-zero, round-to-odd, etc). By combining the polynomials generated under this new approach with range reduction and output compensation algorithms that have rounding-invariant correctness guarantees, we can create implementations that can produce correct results under any faithful rounding mode (without the overhead of additional instructions required by our round-to-zero emulation methods). In essence, the new implementations built using the rounding-invariant input bounds approach by default provide correctness guarantees for applications using non-standard faithful rounding modes.

4 Experimental Evaluation

We report the results from experimentally evaluating the two proposed methods with respect to correctness and performance relative to the original RLibm prototypes.

Prototype. We used the publicly available code from the RLibm project [39] and built the two proposed methods. To build the prototype that applies the rounding-invariant outputs method's round-to-zero emulation, we changed the default rounding mode for the RLibm project's generators to the round-to-zero (*RZ*) mode. To build the prototype for the rounding-invariant input bounds approach, we made multiple changes: (1) changed the rounding-sensitive range reduction operations to produce *RZ* results, (2) rewrote the reduced interval generation process to deduce new intervals that account for rounding-induced variability in the output compensation functions, and (3) updated the polynomial generator to evaluate polynomials using interval bounds and generate polynomials that satisfy stricter correctness constraints. Our prototype uses the MPFR library [21] and RLibm's algorithm to compute the Oracle 34-bit round-to-odd (*RO*) result for each input [35]. Our prototype's polynomial generator uses an exact rational arithmetic LP solver, SoPlex, and RLibm's publicly available randomized LP solver [2, 4] to solve the constraints. Using these methods, we have developed a new math library that has a collection of twenty-four new implementations targeting twelve elementary functions (sin, $sinh$, $sinpi$, cos, $cosh$, $cospi$, log, $log2$, $log10$, exp, $exp2$, and $exp10$).

They are designed to directly produce correctly rounded results for all representations with up to 32-bits with respect to all four standard rounding modes, irrespective of the application's rounding mode. Our prototype is open source, and the artifact is publicly available [42].

Methodology. To evaluate correctness, we run the new implementations on every input from each target representation (10-bits to 32-bits). For every input, we run a given implementation under all four rounding modes and compare the outputs rounded to the target representation against the Oracle result generated using the MPFR library. As a specific example, testing correctness for the 32-bit representation involves evaluating the results for each of the 4 billion inputs under all four rounding modes. To test the prototype's ability to produce correct results while maintaining the application-level rounding mode, we call `fesetround` with each rounding mode before invoking the implementation being tested. To compare the performance of the implementations built through the different approaches, we use `rdtsc` to count the number of cycles taken to compute the result for each 32-bit input. We aggregate these counts to compute the total time taken by a given elementary function implementation to produce outputs for the entire 32-bit FP domain. We compile the test harnesses with the `-march=native -frounding-math -fsignaling-nans` flags. We performed the experiments on Ubuntu 24.04 and an AMD EPYC 7313P CPU with a 3.0 GHz base frequency.

Ability to produce correctly rounded results with multiple rounding modes. The 24 new functions built using our rounding-invariant outputs and rounding-invariant input bounds approaches produce correctly rounded results for all inputs in the domain of every target representation with up to 32-bits across all rounding modes $rnd \in \{RN, RZ, RD, RU\}$. Furthermore, the new implementations directly produce correct results under the application-level rounding mode without requiring calls to `fesetround`. Table 1 reports the ability of the libraries to produce correctly rounded results while using the application's rounding mode. For this evaluation, we disabled the rounding modes changes performed by RLibm's default library with `fesetround`, which is indicated by RLibm* in Table 1. When we do not change the rounding mode, RLibm* produces incorrect outputs for approximately 100 inputs. These incorrect outputs are primarily due to rounding mode induced variability. The RLibm prototype with rounding mode changes to *RN* before each call produces correctly rounded results for all inputs similar to our methods but incurs performance overheads. CORE-MATH fails to produce correctly rounded results for representations smaller than 32-bits due to double rounding issues. The math libraries for double-precision from `glibc` do not produce correctly rounded results for any representation. CORE-MATH and `glibc`'s double libm produce incorrect (*i.e.*, not a correctly rounded result) results for more than 200 million inputs for the `sin` function for a 31-bit representation because of double rounding errors.

Table 1. The table lists whether the libraries generate correctly rounded results for all inputs in FP representations with 10 to 32-bits under each of the four standard rounding modes. We compare the functions from our new library (both methods), RLibm's library without rounding mode changes (RLibm*), Core-Math's 32-bit float libm, and glibc's 64-bit double libm. We use ✓ to indicate that a function produces correct results for all inputs across all representations considered under all standard rounding modes. We use ✗ otherwise. N/A indicates that a function is not implemented in the tested library.

$f(x)$	Ours	RLibm*	Core-Math	glibc
$ln(x)$	✓	✗	✗	✗
$log_2(x)$	✓	✗	✗	✗
$log_{10}(x)$	✓	✗	✗	✗
e^x	✓	✗	✗	✗
2^x	✓	✗	✗	✗
10^x	✓	✗	✗	✗
$sin(x)$	✓	✗	✗	✗
$cos(x)$	✓	✗	✗	✗
$sinh(x)$	✓	✗	✗	✗
$cosh(x)$	✓	✗	✗	✗
$sinpi(x)$	✓	✗	✗	N/A
$cospi(x)$	✓	✗	✗	N/A

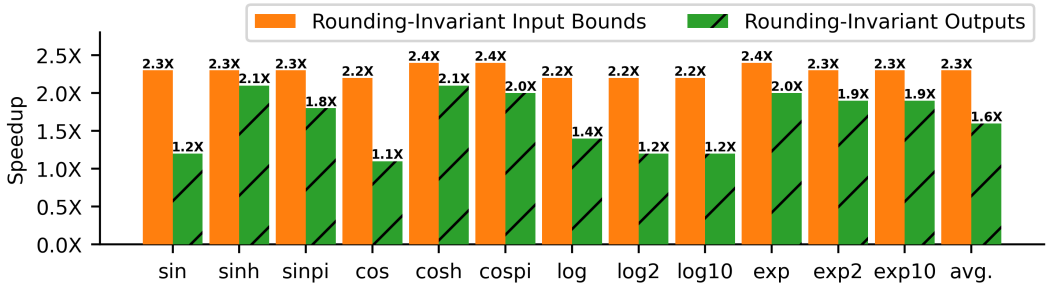


Fig. 7. Speedup (*i.e.*, ratio of the execution times) achieved by the implementations produced using our two approaches, rounding-invariant input bounds and rounding-invariant outputs, when compared to the original RLibm counterparts.

Improved performance of generated math libraries. Figure 7 reports the speedup (*i.e.*, the ratio of the execution times) of the implementations produced through our rounding-invariant input bounds and rounding-invariant outputs approaches over the default RLibm functions (*i.e.*, two bars for each function). The default RLibm implementations contain the latency of the two separate calls to `fesetround`, one used to switch from the application-level rounding mode to *RN* and one used to revert back to the original rounding mode. For all twelve functions, the implementation built using the rounding-invariant input bounds approach exhibits the best performance improvement. On average, functions built using our rounding-invariant inputs method are 2.3× faster than the RLibm functions. Our rounding-invariant input bounds method produces implementations that do not need any changes to the rounding mode while performing similar amount of useful computation as the default RLibm implementations, which is the primary reason for performance improvement. This significant performance improvement also highlights the degree to which calls to `fesetround` degrade the performance of the existing implementations.

In contrast to the rounding-invariant input bounds method, our rounding-invariants outputs method that does round-to-zero emulation is on average only 1.6× faster than the original RLibm functions. While the performance gains of the rounding-invariant input bounds implementations are generally even across all the functions considered, the results associated with the rounding-invariant outputs method can vary significantly. The cost of round-to-zero emulation depends on the number of additions and multiplications in the final implementation, which depends on the degree and the number of terms of the polynomial and can vary across functions. The custom *RZ* operations are alternatives to the basic FP addition and multiplication operations and are primarily used within the polynomial evaluation and output compensation portions of the implementations. Each elementary function has a subset of inputs for which the outputs can be directly approximated with a constant, which obviates range reduction, polynomial evaluation, and output compensation. The number of inputs that undergo polynomial evaluation and output compensation varies widely between the different functions. Consequently, the number of inputs subject to the overheads introduced by the custom *RZ* operations also varies widely.

Rounding mode changes with hardware instructions. The cost of changing the rounding mode depends on the architecture and the implementation of the floating-point environment. While reporting the performance improvements above, we use the `fesetround` function because that is the most portable way to use the implementations across architectures. On x86-64, the fesetround function does three things: checks that the rounding mode is valid, sets the rounding mode for the x87 environment, and sets the rounding mode for the SSE instructions. Each call has an overall latency of 40 cycles on average. If we can assume that the SSE rounding mode and the FP `x87`

rounding mode can be out of sync in the FP environment (*i.e.*, any interim fegetround call will be wrong) and the rounding mode is valid, we can set the rounding mode by setting the `mxcsr` registers in the `x86-64` ISA. Here is the snippet of the assembly code that we used to change the rounding mode with just hardware `x86-64` instructions.

```
unsigned int xcw;
__asm__("stmxcsr %0": "=m" (*&xcw);
xcw &= ~0x6000;
xcw |= round << 3;
__asm__("ldmxcsr %0": : "m" (*&xcw);
```

The average total latency for changing the rounding mode with the above snippet is 15 cycles. Our final implementations from the rounding-invariant inputs bounds method are 1.4× faster than the original RLibm's implementations using hardware assembly instructions for changing the rounding mode.

In summary, the new functions generated using our two methods are not only faster on average than their RLibm counterparts but can also produce correctly rounded results directly with the rounding modes used by the applications.

5 Related Work

Approximating elementary functions. There is a long line of work on approximating elementary functions for FP representations [7, 15–18, 21, 37, 46, 55, 58]. Range reduction is a key component of this task [5, 14, 51–54]. Subsequently, the task is to produce a polynomial approximation over the reduced domain. A well-known and popular tool is Sollya [11]. Using a modified Remez algorithm [8], Sollya can generate polynomials with FP coefficients that minimize the infinity norm. There is also subsequent work to compute and prove the error bound on polynomial evaluation using interval arithmetic [10, 12]. Sollya is an effective tool for creating polynomial approximations using the minimax method. There have been efforts to prove bounds on the results of math libraries [22–24, 29, 48]. Recent efforts have focused on repairing individual outputs of math libraries [57, 59]. Muller's seminal book on elementary functions is an authoritative source on this topic [37].

Correctly rounded libraries. CR-LIBM [15] provides implementations of functions that produce correctly rounded results for double-precision for a single rounding mode. CR-LIBM provides four distinct implementations, one corresponding to each rounding mode. Further, when CR-LIBM's results are double rounded to target representations, it can produce wrong results due to double rounding errors [35]. The CORE-MATH project [50] is also building a collection of correctly rounded elementary functions. It uses the worst-case inputs needed for correct rounding and uses the error bound required for those inputs while generating a minimax polynomial with Sollya. However, they produce correctly rounded results for a specific representation.

Comparison to our prior work on RLibm. This paper builds upon our prior work in the RLibm project [2, 3, 31, 32, 35], which approximates the correctly rounded result using an LP formulation. We use RLibm's method for generating oracles, RLibm's randomized LP algorithm for full rank systems, and its fast polynomial generation. We use the RLibm project's idea of approximating the correctly rounded result for a 34-bit representation with the round-to-odd mode to generate correctly rounded results for all inputs with multiple representations and rounding modes with a single polynomial approximation. We enhance the RLibm approach and the entire pipeline to avoid the rounding mode changes necessary due to the reliance on the round-to-nearest-mode as its implementation rounding mode, which improves the performance by 2×.

An alternative to our approach is to develop four different implementations using the RLibm method where each implementation is tailored to a specific rounding mode similar to CR-LIBM's

implementations. It just requires checking the rounding mode and choosing the correct implementation. In our interactions with various math library developers, they were concerned about the code bloat and concomitant software maintenance issues. Our collaborators at Intel, who were interested in the RLIBM project, also favored a single implementation that is usable as a reference library. If the range reduction and output compensation methods are rounding-mode oblivious, then it is possible to generate different polynomial coefficients for each rounding mode rather than distinct implementations. However, the range reduction and output compensation functions used in the RLIBM project are not rounding-mode oblivious [1, 30, 33, 34, 44]. Hence, we use the rounding-invariant outputs method to make range reduction rounding mode oblivious. Subsequently, we use either the rounding-invariant outputs method or the rounding-invariant input bounds method to address rounding-mode induced variability in output compensation. Further, our algorithms to produce the round-to-zero result from any rounding mode can be independently useful in many applications.

Accounting for error. We compute the error in an FP operation to emulate the round-to-zero result using error-free transformations, which have been used for compensated summation [25, 47], compensated Horner Scheme [28], and robust geometric algorithms [49]. The idea of Fast2Sum was used in accurate summation by Kahan [25] and Dekker [19]. Fast2Sum is fast and requires three FP operations and a branch instruction. Subsequently, it has been modified to remove the branch with TwoSum [27]. Boldo *et al.* [6] have analyzed Fast2Sum and TwoSum with respect to overflows and underflows. They have shown that Fast2Sum is almost immune to overflow. The design of error-free transformations for other rounding modes has been explored in a dissertation by Priest [45]. Dekker introduced an algorithm to identify the error in a multiplication operation based on Veltkamp splitting [38]. Fused-multiply-add (FMA) instructions make the computation of errors easier with just two instructions [38]. This paper builds on these results to design new decision procedures to produce the round-to-zero result given an addition and a multiplication operation performed in any rounding mode.

These error-free transformations (EFTs) have also been used recently for debugging numerical code [13, 20]. Shaman [20] uses EFTs as an oracle and implements a C++ library using operator overloading. EFTSanitizer [13] uses compile-time instrumentation to add these error-free transformations for primitive operations and computes the rounding error. It uses the MPFR library to measure the error in a call to an elementary function. These tools are primarily focused on the round-to-nearest rounding mode and did not explore other rounding modes.

6 Conclusion

This paper proposes two methods, rounding-invariant outputs and rounding-invariant input bounds, to design a single implementation that produces correctly rounded results for all inputs with multiple representations and rounding modes while using the application's rounding mode. The key idea in the rounding-invariant outputs method is to emulate the round-to-zero result for all rounding modes by augmenting each FP addition and multiplication. We design new algorithms to produce the round-to-zero result and provide their associated correctness proofs. Through the rounding-invariant input bounds method, we deduce the bounds on the output of a sequence of FP operations to account for variability induced by rounding modes and augment the RLIBM pipeline to incorporate these bounds. Our math library can serve as a fast reference library for multiple representations with up to 32-bits because it makes double rounding innocuous. It is a step in our effort to make correct rounding mandatory in the next version of the IEEE-754 standard and enhance the portability of applications using such libraries. In the future, we want to explore extending this approach to develop correctly rounded math libraries for the GPU ecosystem, the 64-bit representation, and other extended precision representations.

Acknowledgments

We thank the PLDI reviewers, Bill Zorn, and members of the Rutgers Architecture and Programming Languages (RAPL) lab for their feedback on this paper. This material is based upon work supported in part by the research gifts from the Intel corporation and the National Science Foundation with grants: 2110861 and 2312220. 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 Intel corporation or the National Science Foundation.

References

[1] Mridul Aanjaneya, Jay P. Lim, and Santosh Nagarakatte. 2021. RLIBM-Prog: Progressive Polynomial Approximations for Correctly Rounded Math Libraries. arXiv:2111.12852 Rutgers Department of Computer Science Technical Report DCS-TR-758.

[2] Mridul Aanjaneya, Jay P. Lim, and Santosh Nagarakatte. 2022. Progressive Polynomial Approximations for Fast Correctly Rounded Math Libraries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 552–565. doi:10.1145/3519939.3523447

[3] Mridul Aanjaneya and Santosh Nagarakatte. 2023. Fast Polynomial Evaluation for Correctly Rounded Elementary Functions Using the RLIBM Approach. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal, QC, Canada) *(CGO 2023)*. Association for Computing Machinery, New York, NY, USA, 95–107. doi:10.1145/3579990.3580022

[4] Mridul Aanjaneya and Santosh Nagarakatte. 2024. Maximum Consensus Floating Point Solutions for Infeasible Low-Dimensional Linear Programs with Convex Hull as the Intermediate Representation. *Proc. ACM Program. Lang.* 8, PLDI, Article 197 (Jun 2024), 25 pages. doi:10.1145/3656427

[5] 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. doi:10.1109/TC.2008.216

[6] Sylvie Boldo, Stef Graillat, and Jean-Michel Muller. 2017. On the Robustness of the 2Sum and Fast2Sum Algorithms. *ACM Trans. Math. Softw.* 44, 1, Article 4 (Jul. 2017), 14 pages. doi:10.1145/3054947

[7] Ian Briggs, Yash Lad, and Pavel Panchekha. 2024. Implementation and Synthesis of Math Library Functions. *Proc. ACM Program. Lang.* 8, POPL, Article 32 (Jan 2024), 28 pages. doi:10.1145/3632874

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

[9] Nicolas Brisebarre, Guillaume Hanrot, Jean-Michel Muller, and Paul Zimmermann. 2024. Correctly-rounded evaluation of a function: why, how, and at what cost? (May 2024). https://hal.science/hal-04474530 working paper or preprint.

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

[11] 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. doi:10.1007/978-3-642-15582-6_5

[12] 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. doi:10.1109/QSIC.2007.4385491

[13] Sangeeta Chowdhary and Santosh Nagarakatte. 2022. Fast shadow execution for debugging numerical errors using error free transformations. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 190 (Oct 2022), 28 pages. doi:10.1145/3563353

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

[15] 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. doi:10.1117/12.505591

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

[17] 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. doi:10.1109/TC.2010.128

[18] 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. doi:10.1145/1141277.1141584

[19] T. J. Dekker. 1971. A floating-point technique for extending the available precision. *Numer. Math.* 18, 3 (Jun. 1971), 224–242. doi:10.1007/BF01397083

[20] Nestor Demeure. 2020. *Compromise between precision and performance in high-performance computing.* Ph. D. Dissertation. Université Paris-Saclay. https://tel.archives-ouvertes.fr/tel-03116750

[21] 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). doi:10.1145/1236463.1236468

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

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

[24] 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. doi:10.1007/978-3-642-03359-9_4

[25] William Kahan. 1965. Pracniques: Further Remarks on Reducing Truncation Errors. In *Communications of the ACM*, Vol. 8. ACM, New York, NY, USA. doi:10.1145/363707.363723

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

[27] Donald E. Knuth. 1998. *The Art of Computer Programming Volume 2: Seminumerical Algorithms.* Addison-Wesley.

[28] Philippe Langlois, Stef Graillat, and Nicolas Louvet. 2006. Compensated Horner Scheme. In *Algebraic and Numerical Algorithms and Computer-assisted Proofs (Dagstuhl Seminar Proceedings (DagSemProc), Vol. 5391)*, Bruno Buchberger, Shin'ichi Oishi, Michael Plum, and Sigfried M. Rump (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany. doi:10.4230/DagSemProc.05391.3

[29] 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. doi:10.1145/3158135

[30] Jay P. Lim, Mridul Aanjaneya, John Gustafson, and Santosh Nagarakatte. 2020. A Novel Approach to Generate Correctly Rounded Math Libraries for New Floating Point Representations. arXiv:2007.05344 Rutgers Department of Computer Science Technical Report DCS-TR-753.

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

[32] Jay P. Lim and Santosh Nagarakatte. 2021. High Performance Correctly Rounded Math Libraries for 32-bit Floating Point Representations. In *42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'21)*. doi:10.1145/3453483.3454049

[33] Jay P Lim and Santosh Nagarakatte. 2021. RLIBM-32: High Performance Correctly Rounded Math Libraries for 32-bit Floating Point Representations. arXiv:2104.04043 Rutgers Department of Computer Science Technical Report DCS-TR-754.

[34] Jay P. Lim and Santosh Nagarakatte. 2021. RLIBM-ALL: A Novel Polynomial Approximation Method to Produce Correctly Rounded Results for Multiple Representations and Rounding Modes. arXiv:2108.06756 Rutgers Department of Computer Science Technical Report DCS-TR-757.

[35] Jay P. Lim and Santosh Nagarakatte. 2022. One Polynomial Approximation to Produce Correctly Rounded Results of an Elementary Function for Multiple Representations and Rounding Modes. *Proceedings of the ACM on Programming Languages* 6, POPL, Article 3 (Jan. 2022), 28 pages. doi:10.1145/3498664

[36] David Monniaux. 2008. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.* 30, 3, Article 12 (May 2008), 41 pages. doi:10.1145/1353445.1353446

[37] Jean-Michel Muller. 2016. *Elementary Functions: Algorithms and Implementation.* Sprinder, 3rd edition. doi:10.1007/978-1-4899-7983-4

[38] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. 2018. *Handbook of Floating-Point Arithmetic* (2nd ed.). Birkhäuser Basel. doi:10.1007/978-3-319-76526-6

[39] Santosh Nagarakatte, Sehyeok Park, Mridul Aanjaneya, and Jay P. Lim. 2024. *The RLIBM Project.* https://www.cs.rutgers.edu/~santosh.nagarakatte/rlibm/

[40] 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/

[41] Michael Overton. 2001. *Numerical computing with IEEE floating point arithmetic.* SIAM, Society for Industrial and Applied Mathematics. doi:10.1137/1.9780898718072

[42] Sehyeok Park, Justin Kim, and Santosh Nagarakatte. 2025. Artifact for Correctly Rounded Math Libraries Without Worrying about the Application's Rounding Mode. doi:10.5281/zenodo.15066862
[43] Sehyeok Park, Justin Kim, and Santosh Nagarakatte. 2025. RLIBM-MultiRound: Correctly Rounded Math Libraries Without Worrying about the Application's Rounding Mode. arXiv:2504.07409 Rutgers Department of Computer Science Technical Report DCS-TR-759.
[44] Sehyeok Park and Santosh Nagarakatte. 2025. Fast Trigonometric Functions using the RLIBM Approach. In *Proceedings of the International Workshop on Verification of Scientific Software (VSS 2025)*.
[45] Douglas M. Priest. 1992. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. Ph. D. Dissertation. USA. UMI Order No. GAX93-30692.
[46] 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.
[47] Siegfried M. Rump. 2009. Ultimately Fast Accurate Summation. *SIAM Journal on Scientific Computing* 31, 5 (2009), 3466–3502. doi:10.1137/080738490
[48] Jun Sawada. 2002. Formal verification of divide and square root algorithms using series calculation. In *3rd International Workshop on the ACL2 Theorem Prover and its Applications*.
[49] Jonathan Shewchuk. 1996. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete and Computational Geometry* 18 (Jul 1996). doi:10.1007/PL00009321
[50] Alexei Sibidanov, Paul Zimmermann, and Stéphane Glondu. 2022. The CORE-MATH Project. In *ARITH 2022 - 29th IEEE Symposium on Computer Arithmetic*. virtual, France. https://hal.inria.fr/hal-03721525
[51] Shane Story and Ping Tak Peter Tang. 1999. New algorithms for improved transcendental functions on IA-64. In *Proceedings 14th IEEE Symposium on Computer Arithmetic*. 4–11. doi:10.1109/ARITH.1999.762822
[52] Ping-Tak Peter Tang. 1989. Table-Driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic. *ACM Trans. Math. Software* 15, 2 (June 1989), 144–157. doi:10.1145/63522.214389
[53] 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. doi:10.1145/98267.98294
[54] P. T. P. Tang. 1991. Table-lookup algorithms for elementary functions and their error analysis. In *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic*. 232–236. doi:10.1109/ARITH.1991.145565
[55] Lloyd N. Trefethen. 2012. *Approximation Theory and Approximation Practice (Other Titles in Applied Mathematics)*. Society for Industrial and Applied Mathematics, USA. doi:10.1137/1.9781611975949
[56] 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
[57] 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. doi:10.1145/3290369
[58] Abraham Ziv. 1991. Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit. *ACM Trans. Math. Software* 17, 3 (Sept. 1991), 410–423. doi:10.1145/114697.116813
[59] 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. doi:10.1145/3371128

Received 2024-11-14; accepted 2025-03-06