Fast Polynomial Evaluation for Correctly Rounded Elementary Functions using the RLIBM Approach

Mridul Aanjaneya
Department of Computer Science
Rutgers University
United States
mridul.aanjaneya@rutgers.edu

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

Abstract

This paper proposes fast polynomial evaluation methods for correctly rounded elementary functions generated using our RLIBM approach. The resulting functions produce correct results for all inputs with multiple representations and rounding modes. Given an oracle, the RLIBM approach approximates the correctly rounded result rather than the real value of an elementary function. A key observation is that there is an interval of real values around the correctly rounded result such that any real value in it rounds to the correct result. This interval is the maximum freedom available to RLIBM's polynomial generation procedure. Subsequently, the problem of generating correctly rounded elementary functions using these intervals can be structured as a linear programming problem. Our prior work on the RLIBM approach uses Horner's method for polynomial evaluation.

This paper explores polynomial evaluation techniques such as Knuth's coefficient adaptation procedure, parallel execution of operations using Estrin's procedure, and the use of fused multiply-add operations in the context of the RLIBM approach. If we take the polynomial generated by the RLIBM approach and subsequently perform polynomial evaluation optimizations, it results in incorrect results due to rounding errors during polynomial evaluation. Hence, we propose to integrate the fast polynomial evaluation procedure in the RLIBM's polynomial generation process. Our new polynomial evaluation procedure that combines parallel execution with fused multiply-add operations outperforms the Horner's method used by RLIBM's correctly rounded functions. We show the resulting polynomials for 32-bit `float` are not only correct but also faster than prior functions in RLIBM by 24%.

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

Keywords: RLIBM, correctly rounded, fused-multiply-add, Horner's method, coefficient adaptation, Estrin's procedure

ACM Reference Format:
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 (CGO '23), February 25 – March 1, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579990.3580022

1 Introduction

The IEEE standard for the floating point (FP) representation mandates correct rounding for primitive operations. It recommends, but does not mandate, correct rounding for elementary functions [27]. Correctly rounded elementary functions improve the portability and reproducibility of applications that use them. Unsurprisingly, there is seminal prior work on generating correctly rounded elementary functions [12, 19, 34]. The classic approaches to generate such correctly rounded functions use (near) minimax approximations, where the goal is to minimize the maximum error across all inputs with respect to the real value [10]. Subsequently, the coefficients of the polynomial are adjusted such that they can be evaluated with a finite precision representation supported in hardware [7, 8]. Typically, the resulting correctly rounded functions produce correct results for a single FP representation and a single rounding mode.

The RLIBM project. In contrast to the minimax methods, our RLIBM project makes a case for approximating the correctly rounded result [1, 2, 20–24, 26]. The RLIBM project splits the problem of generating correctly rounded elementary functions into two tasks: (1) task of generating the oracle result and (2) the task of generating efficient implementations given the oracle result. The RLIBM project focuses on the latter task, which allows them to sidestep the table maker's dilemma [17]. Given the correctly rounded result for an input in the target representation, there is an interval of real values around the correctly rounded result such that producing any real value in this interval rounds to the correctly rounded result (see Figure 2). The RLIBM project shows that this interval is the freedom available to the polynomial generation step,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CGO '23, February 25 – March 1, 2023, Montréal, QC, Canada

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0101-6/23/02…$15.00
https://doi.org/10.1145/3579990.3580022

which is larger than the freedom available with minimax methods, and structures the task of generating a polynomial of degree d that produces correctly rounded results for all inputs as a linear program (*i.e.*, a system of linear inequalities). It uses an LP solver to identify the coefficients. Numerical errors experienced during range reduction and output compensation are addressed by constraining the intervals.

Supporting multiple rounding modes and representations. The above method produces correctly rounded results for a single FP representation and a single rounding mode [22, 23]. The IEEE-754 standard has multiple rounding modes: round-to-nearest-ties-to-even (rn), round-to-nearest-ties-to-away (ra), round-towards-zero (rz), round-towards-positive-infinity (ru), and round-towards-negative-infinity (rd). Furthermore, there are new representations such as `bfloat16` and `tensorfloat32` that make trade-offs between the dynamic range and the precision.

We have recently proposed a method in the RLibm project to generate a single polynomial approximation that produces correctly rounded results for multiple floating point representations and all standard rounding modes [25, 26]. To generate correctly rounded results for FP representations with up to n-bits that have E-bits for the exponent, the key insight is to generate a polynomial approximation that produces correctly rounded results for a representation with $(n+2)$-bits with the *round-to-odd* rounding mode [26]. When such a result is double rounded to the target representation, it produces correct results for representations with k bits, where $E+2 \le k \le n$, and for all standard rounding modes.

Fast polynomial evaluation. We use Horner's method by default for polynomial evaluation in the RLibm project. We observe that we can further improve the performance of the single polynomial generated using the RLibm approach that produces correct results for multiple representations and rounding modes by performing parallel evaluation with SIMD and fused multiply-add (FMA) operations.

For the sake of exposition, consider the following polynomial with 5 terms, where $x \in \mathbb{R}$ is a variable:

$$u(x) = -6 + 6x + 42x^2 + 18x^3 + 2x^4$$

It can be efficiently evaluated with Horner's method using 4 addition and 4 multiplication operations, as follows:

$$u(x) = -6 + x \times (6 + x \times (42 + x \times (18 + x \times 2)))$$

Although Horner's method is optimal in terms of the number of operations, it serializes the computation by creating a longer chain of dependent instructions. Our goal in this paper is to explore methods to reduce the number of operations, enable the execution of the operations in parallel, and improve performance using fused multiply-add operations for polynomial evaluation.

Adapting coefficients. Knuth [18] had proposed a systematic method for *adapting* the polynomial coefficients to reduce the number of operations. This approach favors additions in place of multiplication operations. Adapting the coefficients using Knuth's procedure for the above polynomial will yield:

$$y = (x+4)x - 1, \quad u(x) = ((y + x + 3)y - 1)2$$

This alternate expression requires only 3 multiplications, but 5 additions. We describe the procedure for obtaining adapted coefficients in Section 3. Generating such adapted coefficients for a polynomial of degree n, where $n \ge 5$, requires the solution of a non-linear equation of degree $\lceil n/2 \rceil$ [18].

Parallel execution of subexpressions. We can leverage the instruction-level parallelism (ILP) in modern processors by dividing the problem of polynomial evaluation into subexpressions that are independent of each other.

$$u(x) = (-6 + 6x) + x^2(42 + 18x + 2x^2) \tag{1}$$

In the above example, two subexpressions (*i.e.*, $(-6+6x)$ and $(42 + 18x + 2x^2)$) can execute in parallel using the ILP in the machine. The above process is also known as Estrin's procedure [27]. We describe Estrin's method in Section 4.

Reducing rounding errors with fused multiply-add operations. Another consideration with polynomial evaluation is the number of rounding operations performed. Each rounding operation adds more error to the polynomial approximation generated for an elementary function. This rounding error can be reduced by using fused multiply-add (FMA) operations. A fused multiply-add operation, denoted as $fma(x, y, z)$, computes $x \times y + z$ with only one rounding. In contrast, a multiply followed by an add operation would have performed two rounding operations.

The above example in equation (1) can be implemented with fused multiply-add operations as follows:

$$\begin{aligned} t1 &= fma(x, 6, -6) \\ t2 &= x^2 \\ t3 &= fma(18, x, 42) \\ t4 &= fma(2, t2, t3) \\ u(x) &= fma(t2, t4, t1) \end{aligned}$$

This paper. In this paper, we explore the problem of fast polynomial evaluation for polynomial approximations generated with the RLibm method for a 32-bit floating point input that produces correctly rounded results for multiple FP representations and rounding modes. Although Knuth's coefficient adaptation method and Estrin's parallel procedure have been known to the community, they were not previously used in the context of producing correctly rounded

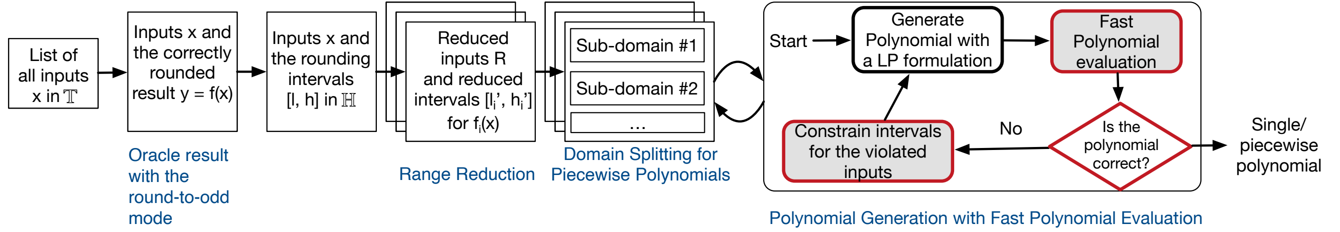

Figure 1. Our modifications to the RLibm pipeline to generate polynomial approximations with fast polynomial evaluation (Knuth's procedure for adapting the coefficients, Estrin's method for parallelizing operations, and the use of fused multiply-add operations to reduce the rounding error), which produces correctly rounded results for all inputs with multiple rounding modes and multiple floating point representations. Our changes are highlighted in red color.

elementary functions because of rounding errors in the resulting polynomial evaluation procedure. A naïve approach for incorporating fast polynomial evaluation methods (*i.e.*, adapt the coefficients, parallelize it, or use *fma* instructions) at the end of the RLibm process will not work. For some inputs, rounding errors in the adapted coefficients can produce incorrect results.

To address this issue, we propose to incorporate the procedure for fast polynomial evaluation into the RLibm pipeline (see Figure 1). Our method is described as follows: Given an oracle, we generate the correctly rounded result with the round-to-odd rounding mode for every input with a representation that has two additional bits of precision compared to the largest representation we wish to support. Then, we compute the rounding interval for each correctly rounded result in the round-to-odd mode. The rounding interval is computed in double precision. Next, we perform range reduction to obtain the reduced input and infer the reduced interval using the inverse of the output compensation. The above steps produce a set of range reduced inputs and the corresponding reduced intervals. To generate a polynomial of degree d, these reduced inputs and intervals define a system of linear inequalities. We use RLibm's method to solve a system of linear inequalities of low dimensions to obtain the candidate polynomial [2].

Next, we explore fast polynomial evaluation techniques (*i.e.*, coefficient adaptation, parallel execution, and the use of *fma* operations) on the candidate polynomial. If the resulting polynomial produces a value in the corresponding reduced intervals for all inputs, we are done with the process. Otherwise, we identify all inputs that when evaluated with the candidate polynomial, produce a value outside the rounding interval. We shrink the rounding interval for such inputs and re-run the entire process a bounded number of times. This iterative process of generating the polynomial, identifying coefficients with the fast procedure for polynomial evaluation, validating the resulting polynomials, and constraining the violated inputs enables us to handle the non-linearity associated with fast polynomial evaluation using an LP-based approach and still generate correctly rounded implementations.

We extend our publicly available RLibm prototype to add fast polynomial evaluation. The resulting prototype is also publicly available [3]. We find that coefficient adaptation with Knuth's method provides some performance gains over Horner's method. We discover Estrin's method when used in tandem with the fused multiply-add operations and our iterative procedure significantly speeds up the resulting elementary functions. The functions generated with this approach are faster than the previous RLibm functions by 24% on average.

2 The RLibm Approach

We describe our RLibm method in detail because we extend it to incorporate fast polynomial evaluation methods.

2.1 Key Insights of the RLibm Approach

The RLibm approach [2, 20, 22, 23, 26] assumes the existence of an oracle that provides correctly rounded results and focuses on generating efficient implementations. Hence, the RLibm project makes a case for approximating the correctly rounded result rather than the real value of the elementary function.

Typically, the implementation of a correctly rounded function for a representation $\mathbb{T}$ uses a representation $\mathbb{H}$ with more precision than $\mathbb{T}$. For example, a correctly rounded library for single precision (*i.e.*, a 32-bit float) is implemented in double precision (*i.e.*, a 64-bit type). The RLibm project observes that for a given input in $\mathbb{T}$, there is an interval of values in representation $\mathbb{H}$ around the correctly rounded result such that any value in that interval rounds to the correctly rounded value (see Figure 2), which is called the *rounding interval*.

The rounding interval can be specified as $[l, h]$, where l is the lower bound and h is the upper bound. When the goal is to generate a polynomial of degree d with $d + 1$ terms, the rounding interval specifies the following constraint on the result of the polynomial evaluation for a given input x:

$$l \leq C_0 + C_1x + C_2x^2 + .. + C_dx^d \leq h$$

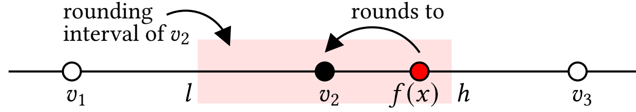

Figure 2. Values v_1, v_2, v_3 are representable in precision $\mathbb{T}$. The real value $f(x)$ for input x is not representable in $\mathbb{T}$ and is rounded to v_2. The RLibm approach identifies v_2's rounding interval (shaded box) in representation $\mathbb{H}$.

This produces a system of linear inequalities corresponding to all inputs in representation $\mathbb{T}$ and their corresponding rounding intervals. The goal is to identify the coefficients (*i.e.*, C_i's) of the polynomial that satisfy all these inequalities. In summary, the RLibm project frames the problem of generating correctly rounded elementary functions as a linear program and uses an LP solver to solve them. The RLibm project demonstrates that the polynomial generation procedure has more freedom by approximating the correctly rounded result in comparison to minimax approximation methods.

The above description provides the key insights about the RLibm approach. There are other important details, such as range reduction, to consider while generating polynomial approximations for a 32-bit float. The original input x is range reduced to x'. Subsequently, the polynomial that we wish to generate approximates the result for x', which is then used inside output compensation to compute the final correctly rounded output for x. Both range reduction and output compensation happen in $\mathbb{H}$ and can experience numerical errors, which should not affect the generation of correctly rounded results. Thus, it is necessary to deduce intervals for the reduced domain such that the polynomial evaluation over the reduced input produces the correct results for the original inputs. The RLibm project uses the inverse of the output compensation function to infer the reduced rounding intervals. The reduced rounding interval is further constrained to account for numerical errors that may arise during range reduction, polynomial evaluation, and output compensation. Finally, a system of linear inequalities corresponding to the reduced inputs and the reduced rounding intervals are solved using an LP solver to identify coefficients of a polynomial of degree d. The RLibm project iteratively increases the degree when it fails to find a polynomial after some threshold number of iterations. The RLibm project also generates piecewise polynomials to limit the increase in the degrees of the generated polynomials, which improves performance. The resulting polynomials are significantly faster than mainstream and correctly rounded libraries [2, 23, 26].

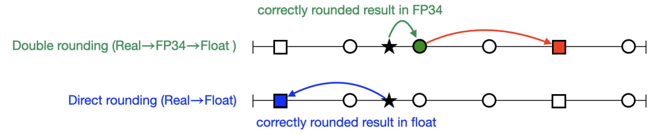

Figure 3. Illustration of double rounding errors. The values in a 32-bit float are represented with boxes. The values in a 34-bit representation FP34, which has two additional bits of precision compared to a 32-bit float, are represented with circles. The top row shows the result of rounding the original real value (black star) to FP34 (green circle) and subsequently rounding the result to a 32-bit float (red box). It can be observed that this result is different from directly rounding the original real value to a 32-bit float (blue box).

2.2 Single Polynomial Approximation for Multiple Representations and Rounding Modes

A naïve approach to generate a single approximation for multiple representations is to use a correctly rounded elementary function designed for a higher precision representation. However, it produces wrong results for some inputs because of double rounding errors. Figure 3 illustrates why double rounding errors occur.

A recent result in the RLibm project, RLibm-All [25, 26], generates a single polynomial approximation that can produce correctly rounded results for all FP representations up to n-bits. The key idea is to generate a polynomial for the (n+2)-bit representation (which has 2 additional precision bits compared to the n-bit representation) using the *round-to-odd* rounding mode. The two additional precision bits as well as the round-to-odd mode retain all necessary information to produce the correct results for any representation with n-bits or fewer.

Round-to-odd is a non-standard rounding mode that has been used in a few niche cases, such as correctly rounding decimal numbers to binary numbers [15] and computing primitive operations in extended precision and still producing correct results for smaller representation [5, 6]. The round-to-odd mode works as follows: if a real value is exactly representable by the target representation, we represent it with that value. Otherwise, the value is rounded to an adjacent floating point value where the bit-pattern is odd when interpreted as an integer. Figure 4 illustrates the round-to-odd mode.

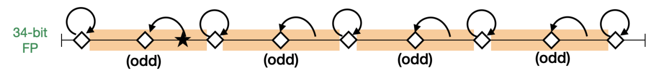

Figure 4. The round-to-odd rounding mode. If a real value is exactly representable, it rounds to that value. Otherwise, it rounds to the nearest value whose bit-pattern is odd.

To understand why double rounding with the round-to-odd mode produces correct results, we have to first understand how rounding works. Typically, we need three pieces of information when rounding a real value to an n-bit floating point representation: (1) the first n-bits of the real value in the binary representation, (2) the $(n+1)^{th}$-bit, known as the *rounding bit*, and (3) the result of the bitwise OR of all the remaining bits, known as the *sticky bit*. When we round a value to a (n+2)-bit representation using the round-to-odd mode, the round-to-odd result precisely maintains the same three pieces of information as rounding the real value directly to the target representation of a lower bitwidth, as shown in Figure 5. Hence, subsequent rounding of the round-to-odd result to a representation with n-bits or smaller produces the correct result.

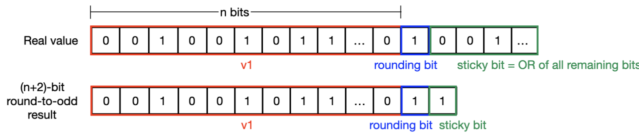

Figure 5. Intuition on why the round-to-odd result with 34-bit FP produces correct results for representations with 10 bits to 32-bits and for all standard rounding modes. The round-to-odd result precisely maintains the three pieces of information as directly rounding the real value to a target representation.

Hence, the key idea in RLibm-All is to generate a polynomial approximation that produces the correctly rounded round-to-odd result in the (n+2)-bit representation. It is feasible because the RLibm project approximates the correctly rounded result. For each input in the n-bit representation, it computes the correctly rounded result of $f(x)$ in the (n+2)-bit representation with the round-to-odd mode using an oracle. It uses the same LP formulation described earlier to generate polynomials.

In summary, RLibm-All produces a polynomial approximation that computes the correctly rounded result in the round-to-odd mode for the 34-bit representation. When this round-to-odd 34-bit FP's result is eventually rounded to any FP representation with 10-bits to 32-bits and for any standard rounding mode, it produces correct results for all inputs.

3 Polynomial Coefficient Adaptation

Knuth's coefficient adaptation procedure [18] is a polynomial evaluation technique that reformulates the algebraic operations in polynomial evaluation to favor addition operations in place of multiplication operations. It is feasible to adapt the coefficients of any polynomial of degree greater than 3 [18]. Next, we describe the process of adapting the coefficients for polynomials of degrees 4, 5, and 6 because RLibm generates polynomial approximations of degree at most 6 and generates piecewise polynomials by splitting the domain into sub-domains when the degree exceeds 6.

3.1 Coefficient Adaptation for Polynomials of Degree 4

Consider the following polynomial equation with 5 terms, where $u_i \in \mathbb{R}$ for all $0 \le i \le 4$, and $x \in \mathbb{R}$ is a variable:

$$u(x) = u_0 + u_1 x + u_2 x^2 + u_3 x^3 + u_4 x^4 \tag{2}$$

Using Knuth's method [18] for adapting the polynomial coefficients in equation (2) will yield (assuming $u_4 \neq 0$):

$$y = (x + \alpha_0)x + \alpha_1, \quad u(x) = ((y + x + \alpha_2)y + \alpha_3)\alpha_4 \tag{3}$$

where α_i, $0 \le i \le 4$, are the "adapted" coefficients. This alternate expression requires only 3 multiplications, but 5 additions. In contrast, Horner's method requires 4 multiplications. Equating equations (2) and (3) gives the following formulas for computing the α_j's in terms of the u_k's:

$$\begin{aligned}\alpha_0 &= \frac{1}{2}(u_3/u_4 - 1), \quad \beta = u_2/u_4 - \alpha_0(\alpha_0 + 1),\\ \alpha_1 &= u_1/u_4 - \alpha_0\beta, \alpha_2 = \beta - 2\alpha_1,\\ \alpha_3 &= u_0/u_4 - \alpha_1(\alpha_1 + \alpha_2), \quad \alpha_4 = u_4\end{aligned} \tag{4}$$

Although in this case the adapted coefficients can be derived in closed-form, polynomials of degree-n, for $n \ge 5$, require the solution of a non-linear equation of degree $\lceil n/2 \rceil$ [18].

3.2 Coefficient Adaptation for Polynomials of Degree 5

Any polynomial of degree-5 can be evaluated using 4 multiplications, 6 additions, and 1 temporary variable by using the rule: $u(x) = U(x)x + u_0$, where $U(x) = u_5x^4 + u_4x^3 + u_3x^2 + u_2x + u_1$ is evaluated using the adaptation scheme shown in equation (3). However, a degree-5 polynomial can also be evaluated with 4 multiplications, 5 additions, and 3 temporaries (in registers) by using the following alternate rule:

$$\begin{aligned} y &= (x + \alpha_0)^2 \\ u(x) &= (((y + \alpha_1)y + \alpha_2)(x + \alpha_3) + \alpha_4)\alpha_5 \end{aligned} \tag{5}$$

Let $p = u_3/u_5$ and $q = u_4/u_5$. The determination of α_0 requires computing a real root of the following cubic equation:

$$pq - 2(p + 2q^2)\alpha_0 + 24q\alpha_0^2 - 40\alpha_0^3 = u_2/u_5 \tag{6}$$

Note that equation (6) is guaranteed to have a real root because the cubic polynomial approaches $-\infty$ for large positive values of α_0, and it approaches $+\infty$ for large negative values of α_0. Thus, by continuity, it must assume the zero value somewhere in between. After solving equation (6), the remaining adapted coefficients α_i, where $1 \le i \le 4$, can be computed as follows:

$$\alpha_1 = p - 4q\alpha_0 + 10\alpha_0^2, \quad \alpha_3 = q - 4\alpha_0$$
$$\alpha_2 = u_1/u_5 - \alpha_0^2(\alpha_1 + \alpha_0^2) - 2\alpha_0\alpha_3(\alpha_1 + 2\alpha_0^2) \quad (7)$$
$$\alpha_4 = u_0/u_5 - \alpha_2\alpha_3 - \alpha_0^2\alpha_3(\alpha_1 + \alpha_0^2)$$

3.3 Coefficient Adaptation for Polynomials of Degree 6

A polynomial $u(x) = u_6x^6 + \ldots + u_1x + u_0$ of degree-6 can be evaluated using 4 multiplications, 7 additions, and 2 temporaries (in registers) using the following scheme:

$$z = (x + \alpha_0)x + \alpha_1, \quad w = (x + \alpha_2)z + \alpha_3,$$
$$u(x) = ((w + z + \alpha_4)w + \alpha_5)\alpha_6 \quad (8)$$

Equation (8) saves two of the 6 multiplications required by Horner's method. Since $\alpha_6 = u_6$, let us assume for the sake of simplicity that $u_6 = 1$, which amounts to dividing all the other coefficients u_i, for $0 \le i \le 5$, by u_6. Now, the goal is to compute the adapted coefficients (*i.e.*, α_i where $0 \le i < 6$). We know that $\alpha_6 = u_6$. Further, assume:

$$\beta_1 = (u_5 - 1)/2, \quad \beta_2 = u_4 - \beta_1(\beta_1 + 1),$$
$$\beta_3 = u_3 - \beta_1\beta_2, \;\; \beta_4 = \beta_1 - \beta_2, \;\; \beta_5 = u_2 - \beta_1\beta_3 \quad (9)$$

This case also requires the computation of a real root of the following cubic equation for determining the value of β_6:

$$2y^3 + (2\beta_4 - \beta_2 + 1)y^2 + (2\beta_5 - \beta_2\beta_4 - \beta_3)y$$
$$+(u_1 - \beta_2\beta_5) = 0 \quad (10)$$

Equation (10) is guaranteed to always have a real root, since the cubic polynomial approaches $+\infty$ for large positive values of y, and it approaches $-\infty$ for large negative values of y. Thus, by continuity, it must assume a value of zero somewhere in between. By defining β_7 and β_8 as shown below:

$$\beta_7 = \beta_6^2 + \beta_4\beta_6 + \beta_5, \quad \beta_8 = \beta_3 - \beta_6 - \beta_7 \quad (11)$$

the adapted coefficients can be computed as follows:

$$\alpha_0 = \beta_2 - 2\beta_6, \quad \alpha_2 = \beta_1 - \alpha_0, \quad \alpha_1 = \beta_6 - \alpha_0\alpha_2,$$
$$\alpha_3 = \beta_7 - \alpha_1\alpha_2, \;\; \alpha_4 = \beta_8 - \beta_7 - \alpha_1, \;\; \alpha_5 = u_0 - \beta_7\beta_8 \quad (12)$$

4 Estrin's Method

Modern processors can exploit instruction-level parallelism for increased efficiency by executing those instructions in parallel that do not depend on each other. While Horner's method is optimal in terms of minimizing the number of additions and multiplications required to evaluate an arbitrary polynomial, the series of operations depend sequentially on each other, and so cannot execute in parallel. In contrast, Estrin's method can overcome this limitation, while still keeping the number of arithmetic operations reasonably close to the optimal. The key idea behind Estrin's method is to break a polynomial $u(x) = u_0 + \ldots + u_nx^n$ as follows:

$$u(x) = \underbrace{u_0 + \ldots + u_{m-1}x^{m-1}}_{u_L(x)} + x^m \cdot \underbrace{(u_m + \ldots + u_nx^m)}_{u_R(x)} \quad (13)$$

```
1  Function EvaluatePolynomial(u, x):
2      if n%2 == 0 then
3          u_{n+1} ← 0; v_{n/2} ← u_n
4      end
5      if n ≤ 1 then
6          return u_0
7      end
8      foreach 0 ≤ i ≤ ⌊n/2⌋ do
9          v_i ← u_{2i} + u_{2i+1} x
10     end
11     y ← x^2
12     v(y) ← v_0 + v_1 y + v_2 y^2 + ... + v_{⌊n/2⌋} y^{⌊n/2⌋}
13     return EvaluatePolynomial(v, y)
```

Algorithm 1: Polynomial evaluation using Estrin's method given a polynomial u in variable x of degree n. The method identifies subterms that can be executed in parallel. The computation of various $u_{2i} + u_{2i+1}x$ (lines 8-10) can be computed in parallel. Further, fused multipy-add operations can be used to compute each $u_{2i} + u_{2i+1}x$. At the end of this process, we are now left with a polynomial v in variable y, where $y = x^2$, with degree $\lfloor n/2 \rfloor$. The procedure is recursively invoked on the resultant polynomial until there is only one term left.

where $m = 2^{\lceil \log_2 n \rceil - 1}$. The polynomials u_L and u_R are of degree at most m and are evaluated *recursively*. This definition is a top-down view of Estrin's method. The reader can gain more intuition by analyzing the operations in a bottom-up fashion. Specifically, $u(x)$ can be viewed as a polynomial in x^2 by grouping all sub-expressions of the form $(A + Bx)$ as:

$$v(x^2) = (u_0 + u_1x) + (u_2 + u_3x)\,x^2 + (u_4 + u_5x)\,x^4 + \ldots \quad (14)$$

Each of these sub-expressions can be computed in parallel. The grouping process can be subsequently repeated $\lfloor \log_2 n \rfloor + 1$ times to obtain polynomials in x^4, x^8, and so on. For efficiency, the monomials $x^2, x^4, \ldots$ can be computed in a pre-processing step. Algorithm 1 provides pseudocode for evaluating a polynomial using Estrin's method.

Fused multiply-add operations. Once we have grouped terms in the form $(A+Bx)$, then they can be computed using fused multiply-add operations as *fma*(B, x, A). All iterations of the loop corresponding to lines 8-10 in Algorithm 1 can be computed in parallel using *fma* operations. There are two-fold benefits by combining Estrin's method with *fma* and the RLibm approach. First, *fma* operations reduce the rounding error. When combined with the RLibm procedure,

the use of *fma* can help generate a lower degree polynomial. Second, the *fma* operations can be executed in parallel using the SIMD extensions tailored for *fma* on modern machines.

5 The RLibm Approach with Fast Polynomial Evaluation

Our goal in this paper is to explore incorporating coefficient adaptation and Estrin's method with *fma* operations into the RLibm approach to facilitate faster polynomial evaluation while generating correctly rounded results. The coefficient adaptation procedure described in Section 3 solves a cubic equation that introduces some rounding errors. Furthermore, the resulting coefficients will have some rounding error when they are represented in floating point. Similarly, rounding errors can get amplified by the parallel recursive evaluation procedure in Estrin's method. If we just use these methods on the polynomial generated by the RLibm approach at the end of the process, it may not produce correctly rounded results for all inputs (and other representations/rounding modes), as described in Section 6.

Hence, we propose an integrated iterative process in the RLibm pipeline that consists of the following steps. First, we generate a candidate polynomial with the RLibm approach. Second, we use the coefficient adaptation procedure or the parallel recursive evaluation for the candidate polynomial. Third, we check if the fast polynomial evaluation satisfies the rounding intervals for all inputs. Fourth, when there are certain inputs for which evaluating the polynomial on them produces a result outside the rounding interval, we constrain the rounding intervals for them and repeat the above process. This process that generates and adapts the polynomial and then verifies and constrains the rounding intervals enables us to incorporate fast polynomial evaluation procedures in the RLibm approach, which relies on a linear programming (LP) based formulation.

Our method is motivated by the fact that rounding the polynomial coefficients from exact rational arithmetic generated by an LP solver to double precision, during the verification step of RLibm, is already a non-linear process. Since RLibm has been able to produce correctly rounded versions of several elementary functions despite this non-linearity, our insight is to piggy-back the fast polynomial evaluation procedure with the generate-check-constrain loop to it. Algorithm 2 provides a sketch of our iterative process within the RLibm pipeline, which is also pictorially shown in Figure 1.

Generating the candidate polynomial. We generate the candidate polynomial exactly similar to the RLibm method described in Section 2. Our objective is to generate correctly rounded results for all representations from 10-bits to a 32-bit float and for all rounding modes in the IEEE standard. Hence, we create a polynomial for a 34-bit representation with the round-to-odd mode. Similarly, the rounding intervals are for the round-to-odd oracle result. Range reduction, output compensation, and coefficient adaptation are performed in double precision. We use range reduction to create reduced inputs and infer the reduced intervals. Subsequently, we use RLibm's algorithm for solving a system of linear inequalities of low dimensions to obtain the candidate polynomial.

```
1  Function AdaptPolynomial(f, X, RR_H, OC_H):
2      Y ← ∅
       /* Compute the rounding interval      */
3      foreach (x, T) ∈ X do
4          y ← RN_T(f(x))
5          [l, h] ← RoundingInterval(y, T, H)
6          Y ← (x, [l, h])
7      end
8      L ← ReducedIntervals(Y, RR_H, OC_H)
9      i ← 0
10     while i < N do
11         poly ← RlibmLPSolve(L, d))
12         apoly ←
             AdaptCoeffsOrParallelFMA(poly)
13         foreach (x, [l, h]) ∈ L do
14             if PolyEval(apoly, x, d) ∉ [l, h] then
15                 L ← ConstrainInterval(x, L)
16             end
17         end
18     end
19     return apoly
```

Algorithm 2: A sketch of our procedure to adapt the coefficients of polynomial approximations with modifications to the RLibm approach for a function f given a set of inputs X in the largest representation that we wish to support ($\mathbb{T}$) (*i.e.*, 32-bit `float` in our prototype). Range reduction ($RR_{\mathbb{H}}$) and output compensation ($OC_{\mathbb{H}}$) are performed in representation $\mathbb{H}$ (*i.e.*, $\mathbb{H}$ is double precision in our prototype). The maximum number of iterations in our process is specified by N. We represent the oracle round-to-odd result obtained by rounding the real value of $f(x)$ to representation $\mathbb{T}$ by $RN_{\mathbb{T}}(f(x))$. The function `RoundingInterval` computes the round-to-odd result's rounding interval. The function `ReducedIntervals` computes the reduced inputs and infers the reduced intervals. The function `RlibmLPSolve` generates a single polynomial or a piecewise polynomial that satisfies the constraints of all inputs. The function `AdaptCoeffsOrParallelFMA` adapts the coefficients of the candidate polynomial based on Knuth's coefficient adaptation procedure or the parallel recursive evaluation with Estrin's method. The function `ConstrainInterval` shrinks the rounding intervals for those inputs that evaluate to a result outside the rounding interval with the adapted polynomial.

Incorporating coefficient adaptation and parallel recursive evaluation. Once we have the candidate polynomial, we perform coefficient adaptation using the procedure described in Section 3 or the parallel recursive polynomial

evaluation with *fma* operations described in Section 4. During Knuth's coefficient adaptation, for candidate polynomials of degree 4, we use the closed form formula to adapt the coefficients. For candidate polynomials of degree 5 or degree 6, we solve a cubic equation to compute the adapted coefficients. We use an external cubic solver in double precision. For parallel recursive evaluation with Estrin's method, we explore variants with and without *fma* operations.

Checking the result of fast polynomial evaluation. After generating the candidate polynomial with fast polynomial evaluation, we check if the adapted polynomial, when evaluated on a reduced input, produces a value outside the corresponding rounding interval. We identify all such points. Lines 13-17 in Algorithm 2 perform this step.

Constraining intervals. If the adapted polynomial, when evaluated on a reduced input, produces a value outside the rounding interval, we constrain the rounding interval as follows. Let us consider the constraint, $(x', [l', h']) \in \mathcal{L}$, where x' is the reduced input, l' is the lower bound of the rounding interval, and h' is the upper bound of the rounding interval. If evaluating the adapted polynomial on x' produces a value that is smaller than the lower bound (*i.e.*, l'), then we constrain the rounding interval by setting the new lower bound to the value succeeding l' in double precision. This causes the RLibm polynomial generator to generate new coefficients for the candidate polynomial. Similarly, if evaluating the adapted polynomial on x' produces a value that is larger than the upper bound of the interval constraint (*i.e.* h'), then we set the new upper bound to the value preceding h' in double precision. This also forces the RLibm polynomial generator to generate a new candidate polynomial with different coefficients.

We repeat this process of generating a new candidate polynomial with the refined constraints until it satisfies all constraints or exceeds a user-specified number of iterations. When the RLibm method generates a piecewise polynomial, we adapt the coefficients of each piece with this process.

6 Experimental Results

We describe the results of our experiments for checking the correctness and performance of our polynomial approximations for elementary functions.

6.1 Prototype and Methodology

We extended the publicly available RLibm prototype to build a polynomial generator with adapted coefficients and parallel recursive evaluation. In this process, we created a collection of correctly rounded elementary functions for six functions (*i.e.*, e^x, 2^x, 10^x, $ln(x)$, $log_2(x)$, and $log_{10}(x)$). Recently, we have also developed fast polynomial evaluation methods for trigonometric functions.

To perform evaluation for this paper, we created three new versions of each correctly rounded elementary function (e^x, 2^x, 10^x, $ln(x)$, $log_2(x)$, and $log_{10}(x)$) that incorporates our fast polynomial evaluation techniques into RLibm, which we call RLibm-Knuth, RLibm-Estrin, and RLibm-Estrin+FMA. A single polynomial approximation for each function produces the correctly rounded result for the 34-bit FP representation that has 8-bits for the exponent with the round-to-odd mode. It produces correctly rounded results for all FP representations starting from 10-bits to 32-bits for all five rounding modes in the IEEE standard.

Our prototype uses the MPFR library [14] and RLibm's algorithm to compute the oracle round-to-odd value of $f(x)$ for each input x. It uses an LP solver with exact rational arithmetic, SoPlex, to solve constraints. We use range reduction and output compensation from the RLibm project [2, 22, 23, 26]. We perform range reduction, polynomial evaluation, and output compensation using double precision.

Methodology: We compare the three new versions of our elementary functions with the default version of the functions in the RLibm project. The functions in the RLibm project produce correctly rounded results for all n-bit FP representations, where $10 \leq n \leq 32$, and all five rounding modes. We do not use mainstream libraries (*i.e.*, GLIBC's libm, and Intel's libm) in this paper because they do not produce correctly rounded results for all inputs.

We conducted our experiments on a 2.10GHz Intel Xeon Gold 6230R server with 192GB of RAM running Ubuntu 20.04 that has both Intel turbo boost and hyper-threading disabled to minimize perturbation. The test harness for comparing performance is built using the gcc-9.3.0 compiler with `-O3 -frounding-math -fsignaling-nans` flags. For measuring the performance, we use `rdtscp` to count the number of cycles taken to compute the result for each input. Subsequently, we aggregate these counts for computing the total time taken for computing the elementary function for all inputs.

6.2 Properties of the Generated Polynomial Approximations

Table 1 provides details on the polynomial approximations generated by our method (*i.e.*, three versions: RLibm-Knuth, RLibm-Estrin, and RLibm-Estrin+FMA) in comparison to RLibm. Among the 6 elementary functions, constraints for $log_2(x)$ form a full-rank system. Hence, we generate a single polynomial of degree 5. Other functions are not full-rank, we identify some minimal number of special case inputs that require explicit handling. Both 2^x and 10^x use a single polynomial of degree-5 with few special case inputs since the system of linear equations is not full-rank. The memory usage required by RLibm and our method are very similar for these functions, although our implementation is faster. For $ln(x)$ and $log_{10}(x)$, we could not obtain polynomial approximations that are faster than those in RLibm by using Knuth's coefficient adaptation procedure.

For e^x, RLibm uses a piecewise polynomial with two pieces to minimize the degree of the polynomial approximations

Table 1. Details of polynomial approximations generated by RLibm, RLibm-Knuth, RLibm-Estrin, and RLibm-Estrin+FMA.

		RLibm		**RLibm-Knuth**		**RLibm-Estrin**		**RLibm-Estrin+FMA**	
$f(x)$	# of poly.	Max poly. degree	# of special inputs	Max poly. degree	# of special inputs	Max poly. degree	# of special inputs	Max poly. degree	# of special inputs
$\mathbf{e^x}$	2	4,5	3,0	4,4	5,4	4,4	5,3	4,4	3,3
$\mathbf{2^x}$	1	5	3	5	2	5	2	5	1
$\mathbf{10^x}$	1	5	4	5	3	5	3	5	4
$\ln(\mathbf{x})$	2	4,4	4,6	N/A	N/A	4,4	5,7	4,4	5,6
$\log_2(\mathbf{x})$	1	5	0	5	0	5	0	5	0
$\log_{10}(\mathbf{x})$	4	4,4,4,5	0,1,2,0	N/A	N/A	4,4,4,4	0,1,3,3	4,4,4,4	0,1,2,2

for each subdomain. We do not adapt the first polynomial, which is of degree-4, as we noticed an increase in the number of special case inputs from 3 to 5 with polynomial adaptation. However, for the second subdomain, our method produces a polynomial of degree-4, in contrast to the polynomial of degree-5 generated by RLibm. Similar to RLibm-Knuth, RLibm-Estrin and RLibm-Estrin+FMA versions also reduced the degree of the resulting polynomials. Further, they also reduced the number of special case inputs compared to RLibm-Knuth. This shows that our method can sometimes decrease the degree of the polynomial also, apart from reducing the number of multiplications. Furthermore, the use of *fma* operations in RLibm-Estrin+FMA version reduces rounding error and decreases the number of special case inputs.

For $ln(x)$, RLibm generates a piecewise polynomial with 2 pieces. Our RLibm-Estrin version generates a piecewise polynomial with 2 pieces but has one more special case input for each piece. Using the *fma* operation reduces the number of special case inputs in the RLibm-Estrin+FMA version. For $log_{10}(x)$, RLibm generates a piecewise polynomial with 4 pieces. Our versions, RLibm-Estrin and RLibm-Estrin+FMA, reduce the degree of the resulting pieces, which improves performance.

6.3 Ability to Produce Correctly Rounded Results

Mainstream libraries such as GLIBC's libm and Intel's libm do not produce correctly rounded results for all inputs. They produce several million incorrect results even for a 32-bit float with the float version of the functions. Their double versions are better in producing correctly rounded results for the 32-bit float representation but still do not produce correctly rounded results for all inputs for the six functions that we consider in this paper.

In contrast, RLibm and versions proposed in this paper, RLibm-Knuth, RLibm-Estrin, and RLibm-Estrin+FMA, produce correctly rounded results for all n-bit FP representations, where $10 \leq n \leq 32$, with 8-bit exponent and all five rounding modes with a single polynomial approximation.

Adapting RLibm polynomials as a post-process: We observed an increase in the number of special case inputs that are required to ensure correctly rounded results for all inputs if the coefficients of the polynomial approximations from RLibm were adapted as a post-process. For example, RLibm uses a degree-5 polynomial for 10^x with 4 special case inputs. Adapting the coefficients of this polynomial produces incorrectly rounded results for 4 additional inputs that are different from the 4 special cases, requiring 8 special cases altogether. In contrast, our RLibm-Knuth version produces an adapted polynomial of degree-5, where only 3 inputs need to be treated as special cases in total, and produces correctly rounded results for all inputs. Similarly, RLibm uses a polynomial of degree-5 for 2^x, where 3 inputs are treated as special cases in order to produce correctly rounded results for all inputs. Simply adapting this polynomial produces incorrectly rounded results for 3 additional inputs that are different from the 3 special cases, requiring 6 special cases altogether. RLibm-Knuth, in contrast, produces an adapted degree-5 polynomial, where only 2 inputs are treated as special cases in total to produce correctly rounded results for all inputs.

Table 2. Speedup achieved by implementations in RLibm-Knuth, RLibm-Estrin, and RLibm-Estrin+FMA in comparison to RLibm.

$f(x)$	Speedup of RLIBM-Knuth vs RLIBM	Speedup of RLIBM-Estrin vs RLIBM	Speedup of RLIBM-Estrin-FMA vs RLIBM
$\mathbf{e^x}$	**2.9%**	**13.7%**	**25.3%**
$\mathbf{2^x}$	**7.9%**	**19.7%**	**31.2%**
$\mathbf{10^x}$	**7.5%**	**17.9%**	**27.3%**
$\ln(\mathbf{x})$	N/A	**14.3%**	**20.6%**
$\log_2(\mathbf{x})$	**4.4%**	**17.6%**	**22.7%**
$\log_{10}(\mathbf{x})$	N/A	**9.1%**	**15.9%**

6.4 Performance Improvement with Fast Polynomial Evaluation

Figure 6 and Table 2 report the improvement in performance obtained with our fast polynomial evaluation methods in

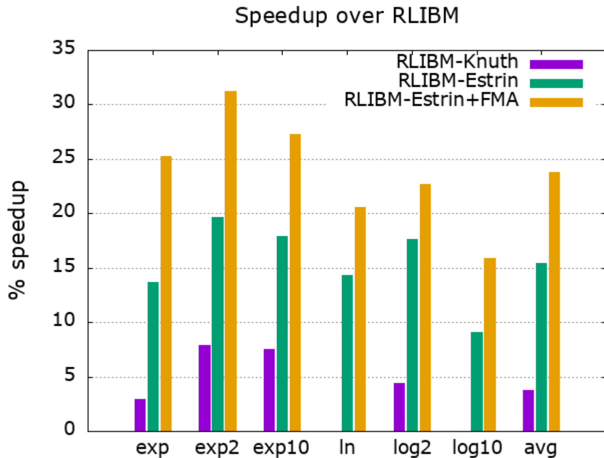

Figure 6. Speedup of the polynomial approximations generated by RLibm-Knuth, RLibm-Estrin, and RLibm-Estrin+FMA methods in comparison to functions in the RLibm project.

comparison to functions in the RLibm project. On average, the RLibm-Estrin+FMA method generates implementations that are 24% faster when compared to functions in the RLibm project. In contrast, the RLibm-Estrin method generates implementations that are 15% faster than the functions in the RLibm project. The functions generated by the RLibm-Knuth method are only 4% faster than the functions in the RLibm project. The better performance of RLibm-Estrin and RLibm-Estrin+FMA functions highlights the benefits of parallel execution using the instruction-level parallelism on the machine. Further, the use of *fma* operations improves performance with a reduction in the degree of the polynomial and with a reduction in the number of overall operations.

In summary, our fast polynomial evaluation techniques enable the generation of single polynomial approximations that not only produce correct results for multiple representations and multiple rounding modes but are also significantly faster than the default functions in the RLibm project.

7 Related Work

Advances in range reduction techniques [4, 11, 30–33] made it feasible to approximate elementary functions [12, 14, 16, 28, 34]. Many correctly rounded math libraries have also been developed [2, 12, 22, 23, 26, 34]. We refer the reader to Muller's seminal book [27] for a detailed survey. Below, we briefly review the most closely related prior work.

CR-LIBM [12, 13] is a correctly rounded math library for double precision, that provides implementations for four out of the five rounding modes in the IEEE standard. It uses Sollya [10] for generating near mini-max polynomial approximations. CR-LIBM computes provable error bounds on polynomial evaluation using interval arithmetic [9]. CR-LIBM's results can suffer from double rounding errors when rounded down to 32-bit floating point values [2, 22, 23, 26].

The CORE-MATH project [29] is also building a collection of fast 32-bit floating point functions. It uses the worst-case inputs needed for correct rounding and uses the error bound required for those inputs while generating a mini-max polynomial with Sollya [10]. The functions in the CORE-MATH project also use *fma* operations. The functions in the CORE-MATH project produce correctly rounded results for a specific representation. When the result of the function is double rounded to a target representation that has fewer than 32-bits, it produces incorrect results.

We build on our prior work on the RLibm project [2, 22, 23, 26], which approximates the correctly rounded result using an LP formulation. We use RLibm's range reduction, output compensation functions, and the idea of creating a single polynomial approximation that produces correctly rounded results for multiple representations and rounding modes from RLibm-All [26]. We use the fast randomized algorithm recently proposed for fast polynomial generation with RLibm [2]. Our main contribution in this paper is a method for integrating fast polynomial evaluation techniques, such as Knuth's procedure for adapting polynomial coefficients, Estrin's method, and Estrin's method with fused multiply-add operations, inside RLibm's polynomial generation process for producing polynomial evaluations that are faster than Horner's method, while guaranteeing correctly rounded results for all inputs and all rounding modes.

8 Conclusion and Future Work

This paper explores the problem of faster polynomial evaluation with the RLibm approach using Knuth's coefficient adaptation procedure and parallel recursive polynomial evaluation with *fma* operations. We show that simply changing polynomial evaluation at the end as a post-process is not sufficient to guarantee correctly rounded results for all inputs. Thus, we propose to integrate the procedure for faster polynomial evaluation in RLibm's polynomial generation process. The resulting polynomials with 32-bit float inputs are not only correct, but also faster than state-of-the-art libraries. We plan to explore this method for other elementary functions as future work.

Acknowledgments

We thank Steve Cannon for suggesting us to explore FMAs with the RLibm approach on Twitter. This material is based upon work supported in part by the National Science Foundation under Grant No. 1908798 and Grant No. 2110861 and a research gift from Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

A Artifact Description

Our artifact with the correctly rounded functions and the polynomial generator for them is available open source and publicly available at https://github.com/rutgers-apl/cgo23-artifact. It is available with the MIT license. The artifact includes: (1) 24 correctly rounded implementations for six elementary functions using Horner's method, the coefficient adaptation procedure by Knuth, Estrin's method, and Estrin's method with fused multiply-add operations for polynomial evaluation, (2) correctness testing framework for the 24 functions, (3) performance testing framework to demonstrate the performance improvements over RLibm, and (4) polynomial generator for generating the polynomials using the Estrin's method with fused multiply-add operations.

A.1 Setup

Requirements. To replicate our experiments, we need a Linux machine with gcc compiler on x86-64 machine. To accurately reproduce our performance experiments, it is recommended to disable turbo-boost and hyper-threading in the BIOS. To run the polynomial generator, it is recommended to have a machine with at least 16 GB of RAM. We need about 50 GB of storage if all the auxiliary files are unpacked. You also need *Python2* to run the scripts.

Installation. There are four main things that are necessary to download the artifact. First, download the sources from the github repository as follows:

```
git clone
https://github.com/rutgers-apl/cgo23-artifact.git
```

Second, download the oracle files for the 6 elementary functions at https://go.rutgers.edu/m6ex2hnc using the browser and save them to a directory. Let us call the directory: ORACLE. We will use these oracle files to test the correctness of the 24 functions. One can use the MPFR library to individually test the correctness. However, the process takes a very long time (close to a day). Hence, we provide the pre-generated oracle files.

Use the gunzip command to unzip the oracle files in the ORACLE directory. Each unzipped oracle file is 12GB. We recommend unzipping only the oracle files corresponding to the specific function being tested.

For example, unzip the oracle files for Log2 as follows:

```
cd <ORACLE>
gunzip Log2Oracle.gz
cd ..
```

Third, download the interval files for the 6 functions for the polynomial generator at https://rutgers.box.com/s/aeaclmvnez8z0rjjsvcspoflslcu1nof. These can be generated using the RLibm-ALL infrastructure. However, it would take a long time. Hence, we provide these interval files. Let us call the directory with these interval files: INTERVALS. Use the gunzip command to unzip the intervals file in the INTERVALS directory. Each unzipped file can be up to 5GB. We recommend unzipping only the interval file corresponding to the function being tested.

For example, unzip the interval files for Log2 as follows:

```
cd <INTERVALS>
gunzip Float34ROLog2Intervals.gz
cd ..
```

Fourth, you need to download Soplex-4.0.1 to run the polynomial generator. Download Soplex-4.0.1 using the browser at https://go.rutgers.edu/6oxgah4a. Untar Soplex-4.0.1 as follows:

```
tar -xvf soplex-4.0.1.tar.gz
cd soplex-4.0.1
make clean
make
export SOPLEX_INCLUDE=<SOPLEX_PATH>/src/
export SOPLEX_LIB=<SOPLEX_PATH>/build/lib/libsoplex.a
cd ..
```

Now, we are ready to replicate the results.

A.2 Reproducing Results

Correctness test. You can test the correctness of all the functions generated from our libraries using the correctness test infrastructure as follows:

```
cd cgo23_artifact/libm
make
cd ../correctness_test
make
```

To test out the default RLibm function, you can execute the following command, which checks if the implementation produces correctly rounded results for all inputs.

```
./Log2 <ORACLE>/Log2Oracle
```

For testing *log2f* with the coefficient adaptation procedure proposed by Knuth for polynomial evaluation, you can execute the following command:

```
./Log2-adapt <ORACLE>/Log2Oracle
```

For testing *log2f* with Estrin's method for polynomial evaluation, you can execute the following command:

```
./Log2-estrin <ORACLE>/Log2Oracle
```

To test *log2f* with Estrin's method along with *fma* operations for the polynomial evaluation and to check if it produces correct results for all inputs, you can execute the following command:

```
./Log2-estrin-fma <ORACLE>/Log2Oracle
```

You should see an output like the following, which indicates that the function produces correct results for all inputs and for multiple representations:

```
Wrong results: 000 (0)
rlibm-latest   wrong result: 0
```

It can take up to 10 minutes for testing each implementation as described above. Similarly you can test out the other functions.

Performance. We provide an automated script to test the performance of RLIBM-Knuth, RLIBM-Estrin, and RLIBM-ESTRIN-FMA with respect to the default RLIBM implementations. To run the performance testing framework, execute the following command:

```
cd cgo23_artifact/performance_test
sh runRLIBMAll.sh
```

It automatically executes all the 24 implementations of the 6 functions and creates text files with the timing data. This script takes close to 40 minutes to complete the execution. To see similar results as reported in the paper, it is necessary to run the script on a machine with hyper-threading and turbo-boost disabled. It is advised to not execute other programs simultaneously with the script.

After the script completes, you can run the analysis script that prints out the speedup with various configurations as follows:

```
python SpeedupOverRLIBM.py
```

It will print out the output as follows:

```
Speedup of RLIBM-Knuth over RLIBM
log: 0.00%
log2: 4.03%
log10: 0.00%
exp: 3.00%
exp2: 7.85%
exp10: 7.27%
Average speedup of RLIBM-Knuth over RLIBM: 3.65%
Speedup of RLIBM-Estrin over RLIBM
log: 12.59%
log2: 16.89%
log10: 8.87%
exp: 12.73%
exp2: 17.80%
exp10: 17.54%
Average speedup of RLIBM-Estrin over RLIBM: 14.36%
Speedup of RLIBM-Estrin-FMA over RLIBM
log: 17.82%
log2: 21.05%
log10: 15.23%
exp: 22.55%
exp2: 28.74%
exp10: 25.01%
Average speedup of RLIBM-Estrin-FMA over RLIBM: 21.66%
```

Generating Polynomials with the RLIBM-Estrin-FMA method. We illustrate the process of generating polynomials by incorporating Estrin's method with FMA operations within the RLIBM polynomial generation framework. To use the polynomial generator, it needs reduced intervals generated using the RLIBM framework, which we provide as intervals. It also requires Soplex installed with the the SOPLEX_INCLUDE and SOPLEX_LIB environment variables set as described above. We illustrate polynomial generation for the Log2 function.

To generate the polynomial for Log2 with Estrin's method with FMA operations, execute the following commands:

```
cd cgo23_artifact/polynomial_generator
make
./polygen-estrin-fma log2-estrin-fma.txt
<INTERVALS>/Float34ROLog2Intervals
```

The configurations for Log2 polynomials that we generate are in the `log2-estrin-fma.txt` file. At the end, the polynomial generator prints out the polynomial.

Log2 is a full-rank system. There is a single polynomial with no special inputs. Other systems run for a large number of iterations and the goal is to find a polynomial with the minimum number of points. Hence, it is recommended to redirect the output of the polynomial generator to a file. Those functions can take close to 3-4 hours to perform such an exhaustive search.

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 *43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'22)*. https://doi.org/10.1145/3519939.3523447

[3] Mridul Aanjaneya and Santosh Nagarakatte. 2022. *Artifact for "Fast Polynomial Evaluation for Correctly Rounded Elementary Functions using the RLIBM Approach"*. https://doi.org/10.5281/zenodo.7369395

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

[5] Sylvie Boldo and Guillaume Melquiond. 2005. When double rounding is odd. In *17th IMACS World Congress*. Paris, France, 11.

[6] Sylvie Boldo and Guillaume Melquiond. 2008. Emulation of a FMA and Correctly Rounded Sums: Proved Algorithms Using Rounding to Odd. *IEEE Transations on Computing* 57, 4 (April 2008), 462–471. https://doi.org/10.1109/TC.2007.70819

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

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

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

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

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

//doi.org/10.1137/1024023

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

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

[14] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Software* 33, 2, Article 13 (June 2007). https://doi.org/10.1145/1236463.1236468

[15] David Goldberg. 1991. What Every Computer Scientist Should Know About Floating-point Arithmetic. In *ACM Computing Surveys*, Vol. 23. ACM, New York, NY, USA, 5–48. https://doi.org/10.1145/103162.103163

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

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

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

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

[20] Jay Lim. 2021. *Novel Polynomial Approximation Methods for Generating Correctly Rounded Elementary Functions*. Ph. D. Dissertation. Rutgers University.

[21] 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.

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

[23] 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)*. https://doi.org/10.1145/3453483.3454049

[24] 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.

[25] 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 [abs] Rutgers Department of Computer Science Technical Report DCS-TR-757.

[26] 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. https://doi.org/10.1145/3498664

[27] Jean-Michel Muller. 2016. *Elementary Functions: Algorithms and Implementation*. Sprinder, 3rd edition.

[28] 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.

[29] 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

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

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

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

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

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

Received 2022-09-02; accepted 2022-11-07