Progressive Polynomial Approximations for Fast Correctly Rounded Math Libraries

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

Jay P. Lim
Department of Computer Science
Yale University
United States
jay.lim@yale.edu

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

Abstract

This paper presents a novel method for generating a single polynomial approximation that produces correctly rounded results for all inputs of an elementary function for multiple representations. The generated polynomial approximation has the nice property that the first few lower degree terms produce correctly rounded results for specific representations of smaller bitwidths, which we call *progressive* performance. To generate such progressive polynomial approximations, we approximate the correctly rounded result and formulate the computation of correctly rounded polynomial approximations as a linear program similar to our prior work on the RLIBM project. To enable the use of resulting polynomial approximations in mainstream libraries, we want to avoid piecewise polynomials with large lookup tables. We observe that the problem of computing polynomial approximations for elementary functions is a linear programming problem in low dimensions, *i.e.*, with a small number of unknowns. We design a fast randomized algorithm for computing polynomial approximations with progressive performance. Our method produces correct and fast polynomials that require a small amount of storage. A few polynomial approximations from our prototype have already been incorporated into LLVM's math library.

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

Keywords: RLIBM, round-to-odd, correctly rounded libraries

ACM Reference Format:
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 (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3519939.3523447

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523447

1 Introduction

Correct rounding of primitive arithmetic operations is mandatory for floating-point (FP) implementations since the inception of the IEEE 754 standard. This requirement was not enforced for elementary functions (algebraic functions such as $1/\sqrt{x}$ and transcendental functions such as sin, cos, log, exp, etc.) due to the *Table Maker's Dilemma* [34]. When the output of an elementary function matches the result that is computed with infinite precision and rounded to the target representation, then it is a correctly rounded result. Correctly rounded elementary functions can enhance the reproducibility and portability of software systems. The IEEE 754-2008 standard has recommended (yet not mandated) correct rounding of elementary functions. Research efforts from several groups have shown that correctly rounded elementary functions can be obtained at a "reasonable" cost [11, 12, 22]. Yet, mainstream math libraries for a 32-bit float still do not produce correctly rounded results for all inputs. When correctly rounded libraries for double precision such as CR-LIBM are re-purposed for 32-bit floats, they can produce wrong results due to double rounding errors.

We have been building correctly rounded functions as part of the RLIBM project [23–29, 35]. Our key insight in the RLIBM project is to separate the task of generating the oracle of an elementary function from the task of generating efficient implementations. Given an oracle (*e.g.*, a high precision math library), we make a case for approximating the correctly rounded result rather than the real value of an elementary function to generate efficient implementations. Figure 1(a) shows the real value and the correctly rounded result for an input. There is an interval of real values around the correctly rounded result such that all real values round to it, which is called the *rounding interval.* This interval provides the constraints on the result of the polynomial approximation for a given input (see Figure 1(b)). Next, we formulate the task of identifying the coefficients of a polynomial of a specific degree that produces a value in the rounding interval for all inputs as a system of linear inequalities.

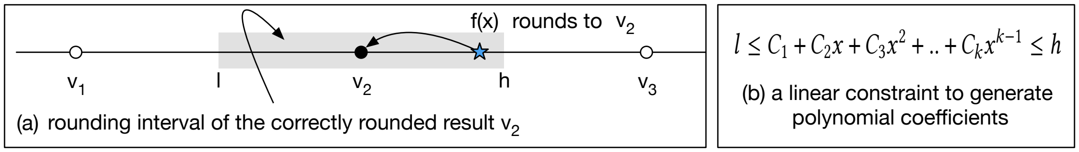

Figure 1. Illustration of the RLibm approach. (a) The values v_1, v_2, and v_3 are representable in the FP representation $\mathbb{T}$. The real value of $f(x)$ for a given input x cannot be exactly represented in $\mathbb{T}$ and is rounded to v_2. The RLibm approach identifies the rounding interval of v_2 and computes polynomial approximations that produce values in this interval. (b) A linear constraint that the polynomial approximation with k terms must satisfy for each rounding interval (*i.e.*, $[l, h]$) for each input x.

By approximating the correctly rounded result, RLibm provides more freedom, allows for lower degree polynomial approximations, and can be realized via a carefully-crafted system of linear inequalities. One challenge with the RLibm approach is that modern LP solvers can only handle a few thousand constraints. Hence, our RLibm prototypes create piecewise polynomials for 32-bit types. Such piecewise polynomials are created for each function and for each representation and rounding mode. We have shown that the resulting functions are both correctly rounded and faster than mainstream libraries such as Intel's libm and glibc's libm [26].

A recent result from our RLibm project, RLibm-All [23, 29], generates a single polynomial approximation that produces correctly rounded results for multiple representations and rounding modes. The key idea behind RLibm-All is to generate a polynomial approximation that produces correctly rounded results for a floating-point (FP) representation with two additional bits of precision (*i.e.*, $n+2$-bits) using the *round-to-odd* mode. The resulting polynomial approximation produces correctly rounded results for all five rounding modes in the standard and for multiple representations with k-bits of precision where $|E| + 1 < k \leq n$ and $|E|$ is the number of exponent bits in the representation. The RLibm-All prototype also generates piecewise polynomials [28, 29]. A single generic polynomial approximation that produces correct results for multiple representations and rounding modes is attractive because it avoids unnecessary code duplication and can enable adoption by mainstream libraries.

Although polynomial approximations resulting from various RLibm prototypes are fast and correct, they had not been incorporated into mainstream libraries because of the large lookup tables required for the piecewise polynomials. Space usage by the mainstream library is an important consideration as these libraries are used in numerous domains ranging from micro-controllers to high performance systems. To enable mainstream usage of polynomial approximations from the RLibm project, we want to avoid generating large piecewise polynomials and the accompanying lookup tables. Further, we want to improve performance for representations with fewer bits rather than every representation having the same performance because low bitwidth representations are increasingly used in various domains [36, 40].

Progressive polynomials. Our goal is to generate a single polynomial that produces correctly rounded results for multiple FP representations with *progressive* performance. The first few lower degree terms of such a polynomial produces correctly rounded results for representations with fewer precision bits and the higher degree terms become necessary for representations with more precision bits, while keeping the *same* lower degree terms. We call such polynomials *progressive polynomials*. They are inspired by Taylor polynomials, which provide better polynomial fits as one uses more terms. These progressive polynomial approximations offer two major benefits. First, they will provide more efficient implementations for representations with fewer precision bits (*e.g.*, bfloat16 [40] or tensorfloat32 [36]) in comparison to RLibm-All that uses the same high degree polynomial approximation across all representations (see Section 4). Second, they will provide a unified approach to implementing math library functions, as representations with less precision bits can *reuse* the implementation for those with more precision bits, while discarding the higher order terms from the polynomial.

For example, consider the case when we want to produce a single approximation for e^x that produces correct results for all inputs in the 32-bit float, 16-bit bfloat16, and 19-bit tensorfloat32 types. For the sake of argument, suppose we generate a 6-term, 5^{th}-degree progressive polynomial ($C_1 + C_2x^1 + C_3x^2 + C_4x^3 + C_5x^4 + C_6x^5$). We use all 6 terms of the polynomial to produce correctly rounded results for a 32-bit float input. We use only the first four terms of the polynomial to produce correct results for a tensorfloat32 input, which is faster than producing the result for a 32-bit float. Similarly, we use only the first three terms to generate correctly rounded results for a bfloat16 input, which is faster than producing results for both tensorfloat32 and float inputs.

Efficient randomized algorithm for solving linear constraints. To generate progressive polynomial approximations and to avoid storing large tables of coefficients for piecewise polynomials, we observe that the problem of computing a polynomial approximation using the linear programming approach of RLibm is a linear program in *low dimensions*, with far fewer unknown variables in comparison to the number of constraints. Inspired by prior work on linear programs in low dimensions [9], we design a fast randomized algorithm for producing progressive polynomial approximations that uses a significantly smaller table of coefficients (by an order of magnitude) compared to RLibm-All.

Given the number of terms for each representation of interest used in the progressive polynomial, our algorithm uses an LP solver to only solve a small set of $6k^2$ constraints, where k is the maximum number of terms used in the progressive polynomial. Given a multi-set of constraints, the algorithm samples $6k^2$ constraints from the entire set of constraints and solves the sample optimally using the LP solver. If the sample solution violates more than $1/3k$ of the multi-set, it discards the sample. If the sample solution violates less than $1/3k$ of the multi-set, it adds the violated constraints once more to the sample. To efficiently implement this algorithm, we use weights to encode the multi-set and use weighted random sampling to create the sample (see Section 3.3).

This process repeats until we find a solution that satisfies all constraints, which happens when the system of linear inequalities is full-rank (*i.e.*, there are k-linearly independent constraints) or when the number of iterations reaches a threshold. Since we do not know the rank of our system, we iteratively increase the number of terms used for the polynomial and have a threshold on the number of iterations. When the system is full-rank, we prove that our algorithm finds the progressive polynomial in $6k \log n$ iterations in expectation (see Section 3.4).

Prototype and results. Our prototype, RLIBM-Prog, provides a single progressive polynomial approximation that produces the correctly rounded results for multiple representations and multiple rounding modes for 10 elementary functions. It has progressive performance with bfloat16 and tensorfloat32 inputs being 25% and 16% faster than evaluating the entire polynomial. The randomized algorithm produces polynomial approximations that require an order of magnitude lower storage than prior RLibm prototypes [26, 29]. RLIBM-Prog's polynomials for the 32-bit float type are faster than all mainstream and/or correctly rounded libraries. Three polynomial approximations ($ln(x)$, $log_2(x)$, and $log_{10}(x)$) generated by our prototype have already been incorporated into LLVM's math library [30–32].

2 Background

We provide background on the FP representation, the process of computing polynomial approximations for elementary functions, and the RLibm approach [23–29].

2.1 The Floating-Point Representation

The IEEE-754 standard specifies the FP representation $\mathbb{F}_{n,|E|}$ that is parameterized based on the total number of bits (n) and the number of bits used for the exponent ($|E|$). The goal is to represent a large range of values (*i.e.*, wider dynamic range) with a reasonable amount of accuracy (*i.e.*, precision) [16]. The sign of a value is specified by a dedicated sign bit (S). To represent a large range of values, the FP representation has an unsigned exponent field (E). Each value is represented as precisely as possible with the mantissa bits (F). Figure 2(c) depicts the bit-string for a 32-bit float.

The values represented by the FP representation are classified into three classes: (a) normal values when the exponent field is neither all zeros nor all ones, (b) subnormal or denormal values when the exponent field is all zeros, and (c) special values when the exponent field is all ones. In the case of normal values, the value represented by the FP bit-string is $(1 + \frac{F}{2^{|F|}}) \times 2^{E-bias}$, where $bias$ is $2^{|E|-1} - 1$. With subnormal values, the value represented by the bit-string is $(\frac{F}{2^{|F|}}) \times 2^{1-bias}$. Subnormal values are used to represent values close to zero. In the case of special values, when the mantissa bits are all zeros, then the bit-string represents positive or negative infinity depending on the sign bit. Otherwise, the bit-string represents not-a-number (NaN).

The common formats specified in the IEEE-754 standard are 16-bit half precision ($\mathbb{F}_{16,5}$), 32-bit single precision float ($\mathbb{F}_{32,8}$), and 64-bit double precision ($\mathbb{F}_{64,11}$).

The bfloat16 and tensorfloat32 formats. Numerous recent variants of the IEEE-754 FP representation increase either the dynamic range or the precision when compared to the existing half precision format. The bfloat16 format [40] is a 16-bit representation with 8 bits for the exponent (*i.e.*, $\mathbb{F}_{16,8}$). Nvidia's `tensorfloat32` [36] is a 19-bit representation with 8-bits for the exponent (*i.e.*, $\mathbb{F}_{19,8}$). It provides the dynamic range of bfloat16 and the precision of the half precision format. Figure 2(a) and Figure 2(b) show the bfloat16 and the tensorfloat32 format.

Rounding mode. When a real value is not exactly representable in the FP representation, it needs to be rounded to a value in the FP representation. The IEEE-754 standard specifies five distinct rounding modes that rounds the real value to one of the two adjacent FP values: 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*). Different rounding modes have different trade-offs in the implementation of various FP operations. The *rn* mode is the widely used rounding mode.

2.2 Approximating Elementary Functions

Elementary functions are functions of a single variable that are typically approximated with polynomial approximations. It is feasible to design polynomial approximations with low error for an elementary function when the input domain is small. Hence, one of the crucial steps in approximating any elementary function is range reduction.

Range reduction and output compensation. Range reduction reduces the domain of an elementary function $f(x)$ to a small input domain using mathematical identities [10]. The range reduction transforms an input x from the original domain of inputs to a reduced input x'. The polynomial

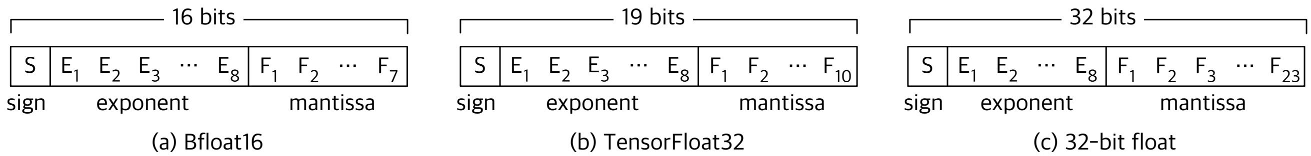

Figure 2. The three FP representations used in this paper: (a) Bfloat16, (b) TensorFloat32, and (c) 32-bit float.

approximations are performed with the reduced inputs (*i.e.*, $y' = P(x')$). After range reduction, the function being approximated with polynomial approximation may be different from the original elementary function (*e.g.*, $ln(x)$ can be approximated with $log_2(x')$). The output (y') has to be adjusted appropriately to produce the output for the original input (x). The output compensation function produces the final result by compensating the range reduced output y' based on the range reduction performed for input x.

Polynomial approximations. The next step is to generate polynomial approximations that take reduced inputs and produce the result of the elementary function in the reduced input domain. One common method to generate such polynomial approximations is to minimize the maximum error of the polynomial approximation with respect to the real value of the elementary function (also known as minimax approximations [34]). A commonly used mini-max approximation is the Remez algorithm [37]. Using real analysis, one can bound the maximum error of such a minimax approximation. CR-LIBM [11, 12], a correctly rounded library for the double precision type, uses this near-minimax approach to generate polynomial approximations.

Range reduction, output compensation, and polynomial evaluation are all implemented in a finite precision representation. Hence, they can experience numerical errors, which when coupled with polynomial approximation errors can cause wrong results.

2.3 The RLibm Approach

We provide a brief background on our prior work in the RLibm project [23–29], where we decouple the problem of generating an oracle from the task of the generating efficient implementations. We assume the existence of an oracle (which may be slow) that provides correctly rounded results. This oracle is only used to compute the correctly rounded result of an elementary function $f(x)$ for each input x in the target representation $\mathbb{T}$. Once there is an oracle result, the RLibm project makes a case for approximating the correctly rounded result rather than the real value of an elementary function [25, 26]. An FP representation can only represent finitely many values accurately. Hence, there is an interval of real values around the correctly rounded result such that all values in the interval round to it. This is the maximum amount of freedom available for the polynomial approximation. The RLibm project has demonstrated that

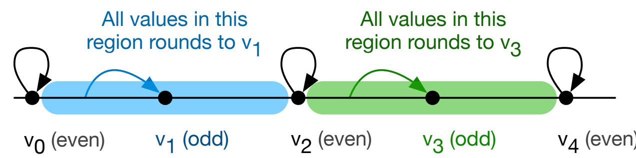

Figure 3. We show the rounding of a real value with the round-to-odd mode. Here, v_0, v_1, v_2, v_3, and v_4 are values representable in a representation $\mathbb{T}$. If the real value is exactly representable in $\mathbb{T}$, then it rounds to that value. Otherwise, it rounds to the nearest value in $\mathbb{T}$ that is odd.

this amount of freedom for polynomial generation by approximating the correctly rounded result is much larger than the one with the minimax approach. Hence, RLibm prototypes provide significant performance benefits when compared to highly optimized libraries [26].

Given the correctly rounded result, the next step is to identify an interval $[l, h]$ around the correctly rounded result such that any value in $[l, h]$ rounds to the correctly rounded result, which is called the *rounding interval*. Figure 1(a) illustrates the rounding interval around the correctly rounded result. Next, range reduction specific to the elementary function is applied to transform an input x to x'. The polynomial approximation will approximate the result for x'. To perform polynomial approximation, one needs the rounding interval that corresponds to the reduced input x'. The RLibm project uses the inverse of the output compensation function to identify the reduced interval $[l', h']$.

Once a set of reduced intervals is available, the next task is to synthesize the coefficients of the polynomial with k terms using an arbitrary precision linear programming (LP) solver such that it satisfies the reduced constraints (*i.e.*, $l' \leq P(x') \leq h'$). Figure 1(b) shows the linear constraint to generate the coefficients of a polynomial with k terms.

Subsequently, the result for the original input x is computed with output compensation. Range reduction, output compensation, and the polynomial evaluation happen in some finite precision representation (*e.g.*, double) and can experience numerical errors. The rounding intervals are further constrained to ensure that the resulting polynomial always produces the correctly rounded results for all inputs.

RLibm-All. The approach described above produces correctly rounded results for all inputs for a specific rounding mode and representation. Our recent work, RLibm-All [29], generates a single polynomial approximation that produces correctly rounded results for multiple representations and multiple rounding modes. When the goal is to create correctly rounded results for a representation with n-bits, the key idea behind RLibm-All is to create polynomial approximations that produce the correctly rounded result of $f(x)$ with the *round-to-odd* mode for a representation with $n+2$-bits (*i.e.*, two additional bits of precision with the same exponent). We have proven that the resulting polynomial produces correctly rounded results for all rounding modes in the standard and all representations with k-bits such that $|E|+1 < k \leq n$, where $|E|$ is the number of exponent bits. The round-to-odd mode is a non-standard rounding mode that avoids double rounding errors and can be described as follows. If the real value is exactly representable in the target representation, then it is rounded to that value. Otherwise, it is rounded to an adjacent value whose bit-string is odd when interpreted as an unsigned integer. Figure 3 pictorially depicts the round-to-odd mode. To correctly round any real value to a FP representation with the standard rounding modes, one needs to identify if the real value is less than, greater than, or equal to the midpoint of two adjacent FP values. The round-to-odd mode preserves this information and avoids double rounding errors [29].

One drawback of the single polynomial approximation with the round-to-odd mode in RLibm-All is that *every* representation must pay the computational cost of the largest representation. Our RLibm-All prototype generated piecewise polynomials with large lookup tables because we were not aware of an effective method to solve a large number of constraints at that point in time and LP solvers cannot automatically solve millions of constraints.

3 Progressive Polynomial Approximations

Our goal is to generate a single polynomial approximation that not only produces correctly rounded results for multiple representations and rounding modes but also has progressively better performance for lower bitwidth representations given a set of representations. We call them progressive polynomials. If we can generate such progressive polynomials, then we can evaluate the first few terms of the polynomial to obtain the correct results for lower bitwidth representations and the entire polynomial for the largest representation.

This paper proposes a novel method to generate progressive polynomial approximations. Building on our prior work in the RLibm project [23–29], we approximate the correctly rounded result and use a linear programming formulation to generate polynomial approximations. In contrast to the prior work in the RLibm project, our setting has a significantly larger number of constraints (a constraint for each input and for each type) because we are generating progressive polynomials. In our prior work in the RLibm project, we were not aware of an effective way to solve an LP problem with millions of constraints. Hence, our prior RLibm prototypes generated piecewise polynomials with large lookup tables to store the polynomial coefficients. The presence of these lookup tables was a barrier for adoption of our polynomial approximations into mainstream math libraries. Hence, we do not want to generate large piecewise polynomials.

A key observation that we make in this paper is that the system of linear inequalities generated by the RLibm approach is a linear program in low dimensions (*i.e.*, a polynomial with a small number of terms k that satisfies millions of constraints). If the set of linear constraints is full-rank, then there exist k linearly independent constraints that identify the polynomial coefficients [9]. Our goal is to develop a fast iterative method for generating progressive polynomials without large lookup tables. One challenge in this setting is that we do not know the rank k of the set of constraints.

3.1 Overview of Our Method

Our approach for generating progressive polynomial approximations consists of the following steps. First, we iteratively explore the number of terms for each individual representation of interest in our progressive polynomial. Second, we use an oracle (*i.e.*, an existing high-precision library) to identify the correctly rounded result for each representation. For the largest representation $\mathbb{T}_i$ of interest, we generate correctly rounded results for a representation with two additional bits of precision ($\mathbb{T}_{i+2}$) with the *round-to-odd* mode inspired by our prior work on RLibm-All [29]. The resulting polynomial approximation produces correctly rounded results for all representations $\mathbb{T}_j$, where $j \leq i$, and for all rounding modes as long as $\mathbb{T}_j$ has the same number of exponent bits as $\mathbb{T}_i$.

Third, we identify an interval of real values that round to the correctly rounded result for every input, which is known as the rounding interval. Fourth, we perform range reduction to identify the reduced input and infer the reduced rounding intervals. Subsequently, we attempt to generate a progressive polynomial from the set of reduced inputs and reduced rounding intervals for each representation. We generate constraints for the largest representation that uses all terms of the polynomial. The polynomial when evaluated should produce a value in the reduced rounding interval. For other representations, we systematically hypothesize a specific number of terms for generating the progressive polynomial. Fifth, we try to generate a polynomial that has k terms and is of degree d given n constraints (*e.g.*, n is 512 million with e^x). We extend Clarkson's method [9] to our context and develop a fast randomized algorithm to identify k linearly independent constraints that identifies the polynomial. If the systems of linear inequalities has full rank, then it has a unique solution.

Our randomized algorithm can be described as follows. Initially, we maintain a multiset M of all n constraints. We sample $6k^2$ constraints from M, where k is the total number of terms for the largest representation in the progressive polynomial. We solve the sample optimally using an LP solver to obtain the solution x^*. If we are able to solve the sample, then we use the resulting polynomial to identify constraints in M that are not satisfied by x^*. While checking whether x^* satisfies the constraint, we evaluate only the specified number of terms as dictated by the configuration of the progressive polynomial. If more than $1/3k$ of the set M of constraints is not satisfied by x^*, then we discard the sample and repeat the above process by creating a new sample. Otherwise, we add each constraint that was not satisfied one additional time to the multiset M. We repeat the above process until we find that x^* for the sample does not violate any constraint in M or the number of iterations exceeds the user-specified cut-off. If there exists a solution (*i.e.*, the system of linear inequalities is full-rank), then the above algorithm finds it in $6k\log(n)$ iterations in expectation.

Algorithm 1 describes our procedure for generating progressive polynomials. Our procedure for computing the oracle result, identifying the rounding intervals, and deducing the reduced rounding intervals is identical to our prior work in the RLibm project [25, 26, 29]. The key difference lies in the manner in which we generate linear constraints for progressive polynomials, the manner in which we evaluate polynomials, and our procedure for generating the polynomial approximation given a set of linear constraints.

3.2 Linear Constraints for Progressive Polynomials

A reduced input x can be present in multiple representations. The rounding interval for each such reduced input will be different depending on the representation(*i.e.*, $[l_x^{\mathbb{T}_1}, h_x^{\mathbb{T}_1}]$ for representation $\mathbb{T}_1$ and $[l_x^{\mathbb{T}_2}, h_x^{\mathbb{T}_2}]$ for representation $\mathbb{T}_2$). A representation with lower bitwidths will have larger rounding intervals as the spacing between adjacent points is relatively larger when compared to a representation with larger bitwidth. We want a single polynomial approximation to satisfy all these bounds of the rounding intervals. Hence,

$$l_x^{\mathbb{T}_1} \le P(x) \le h_x^{\mathbb{T}_1}$$
$$l_x^{\mathbb{T}_2} \le P(x) \le h_x^{\mathbb{T}_2}$$
$$l_x^{\mathbb{T}_3} \le P(x) \le h_x^{\mathbb{T}_3}$$

Progressive performance. We want the resulting single polynomial approximation to have better performance while producing correctly rounded results for lower bitwidths (*i.e.*, progressive performance) when compared to evaluating the entire polynomial for larger bitwidths. Given the number of terms for a representation with a particular bitwidth and the total number of terms for the entire polynomial approximation, we create constraints such that evaluating the first few terms produces a value that lies in the rounding interval corresponding to that representation. Consider the scenario where $\mathbb{T}_1$ is the representation with the largest bitwidth. We are trying to find a polynomial approximation with k_1 terms for it. We also want to find coefficients such that when we evaluate inputs belonging to representations $\mathbb{T}_2$ and $\mathbb{T}_3$ with k_2 and k_3 terms (here $k_1 > k_2 > k_3$), they lie within their respective rounding intervals. The system of linear constraints that we generate for a given input x is as follows,

$$l_x^{\mathbb{T}_3} \le \underbrace{C_1 + C_2 x + \ldots + C_{k_3} x^{k_3 - 1}}_{\mathcal{P}_3(x)} \le h_x^{\mathbb{T}_3}$$
$$l_x^{\mathbb{T}_2} \le \underbrace{\mathcal{P}_3(x) + \ldots + C_{k_2} x^{k_2 - 1}}_{\mathcal{P}_2(x)} \le h_x^{\mathbb{T}_2}$$
$$l_x^{\mathbb{T}_1} \le \underbrace{\mathcal{P}_2(x) + \ldots + C_{k_1} x^{k_1 - 1}}_{\mathcal{P}_1(x)} \le h_x^{\mathbb{T}_1}$$

When we generate constraints for representation $\mathbb{T}_2$, we use the exact same coefficients for the first k_2 terms as we did for representation $\mathbb{T}_1$. Similarly, we use the same coefficients for the first k_3 terms for representation $\mathbb{T}_3$. If we are able to find such polynomials, the resulting polynomial approximation not only produces correctly rounded results for all inputs but also has better performance for representations with lower bitwidth when compared to evaluating all terms in the polynomial. Note that this formulation for generating progressive polynomials creates significantly more constraints (since there is a constraint for each input and each representation). Hence, an efficient method to generate polynomial approximations is crucial.

3.3 A Fast Algorithm for Solving Constraints

To create correctly rounded progressive polynomial approximations, our objective is to generate polynomials of low degree with a few terms. Our prior work on the RLibm project [25, 26, 29] generates piecewise polynomials with approximately 2^{10} sub-domains. Such large tables can interfere with caches in memory-intensive applications and may not be ideal for resource constrained environments such as micro-controllers.

We make a key observation that our system of linear constraints is a linear program of small dimensions [9, 33], which is widely studied. We use ideas from prior work to our setting where we do not know whether the system of linear constraints is "full-rank" (*i.e.*, if there are at least k *linearly independent* constraints).

Algorithm 1 describes our procedure to find a progressive polynomial. As we do not know the rank of our system of linear constraints, we iteratively increase the number of terms for the entire polynomial and for the individual representations. The procedure to identify a small set of key constraints is as follows:

```
Function GenProgPolynomial(f, X, RR_H, OC_H, K, N):
    Y ← ∅
    /* Compute the rounding interval */
    foreach (x, T) ∈ X do
        y ← RN_T(f(x))
        [l, h] ← RoundingInterval(y, T, H)
        Y ← (x, [l, h])
    end
    L ← ReducedIntervals(Y, RR_H, OC_H)
    /* initialize the weights */
    foreach x ∈ L do
        x.w ← 1
        x.u ← random(0, 1)
    end
    i ← 0
    while i < N do
        S ← WeightedRandomSample(L, K)
        (poly, n_v) ← SolveSample(S, L, K)
        i ← i + 1
        if n_v < Limit then
            return (poly, n_v)
        end
    end
    return (∅, 0)
```

Algorithm 1: Our procedure to generate progressive polynomials for an elementary function f given a set of inputs X with their respective representations ($\mathbb{T}$). Range reduction ($RR_{\mathbb{H}}$) and output compensation ($OC_{\mathbb{H}}$) are performed in representation $\mathbb{H}$. Here, K is a vector that provides the number of terms in the progressive polynomial for each representation. The maximum number of iterations is specified by N. We represent the oracle 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 rounding interval. The function `ReducedIntervals` computes the reduced inputs and infers the reduced intervals. The function `WeightedRandomSample` identifies the sample with weighted random sampling. The function `SolveSample` solves the sample and updates the weights of the constraints not satisfied by the solution to the sample, which is described in Algorithm 2.

```
Function SolveSample(S, M, K):
    poly ← LPSolver(S)
    (w_v, w_s, n_v) ← (0, 0, 0)
    k ← max_element(K)
    foreach (x, [l, h]) ∈ M do
        if poly(x, K) ∈ [l, h] then
            /* sum the weights of the satisfied
               constraints */
            w_s ← w_s + x.w
        end
        else
            /* sum the weights of the violated
               constraints */
            w_v ← w_v + x.w
            n_v ← n_v + 1
        end
    end
    /* Check if it is a lucky iteration */
    if w_v ≤ 1/(3k−1) w_s then
        /* Double the weights of violated
           constraints */
        foreach (x, [l, h]) ∈ M and poly(x, K) ∉ [l, h] do
            x.w ← x.w * 2
        end
    end
    return (poly, n_v)
```

Algorithm 2: Given a sample S, the total set of reduced inputs and constraints $\mathcal{M}$, and the degrees of the progressive polynomials, this function `SolveSample` uses the LP solver to solve the sample, identifies whether the iteration happens to be a lucky iteration, and doubles the weights of the violated constraints on a lucky iteration. This function returns the progressive polynomial that solves the sample and the number of constraints violated in $\mathbb{L}$. Here, $poly(x, K)$ evaluates the progressive polynomial using the number of terms specified in K for various representations with input x.

- Let M be a multi-set of constraints.
- **Step 1**: Sample S constraints from M uniformly at random where $|S| = 6k^2$, where k is the number of terms for the largest representation with the progressive polynomial.
- **Step 2**: Solve the sample S optimally using an LP solver to compute x^*.
- **Step 3**: Check how many constraints of M are not satisfied by x^*. If more than $1/3k$ of the constraints in M are not satisfied by x^*, then discard this sample and go to Step 1. Otherwise, add all such constraints not satisfied by x^* another time to M (*i.e.*, M will now have repeated constraints and is a multi-set). We call these iterations **lucky** (*i.e.*, we are making progress towards our goal of identifying the crucial k constraints). Then go back to Step 1.
- Repeat the above until we find a solution x^* that satisfies all constraints in M or the number of iterations exceeds the user-specified threshold.

When we create a sample S with $6k^2$ constraints from the multi-set M and compute the optimum solution x^* for S, then with probability at least $1/2$, x^* will only violate $1/3k$ of the constraints in M. We provide a proof that this algorithm is effective in finding the key constraints necessary to solve the system of linear constraints quickly in Section 3.4.

Removing the multi-set requirement. As we have billions of constraints in M to start with, maintaining a multi-set in memory is challenging. Hence, we logically implement such a multi-set by maintaining weights with each constraint, which are incremented instead of duplicating constraints. We next describe our procedure to find a polynomial with k terms using the weight-based formulation. Initially, each constraint is present only once in the multi-set version. Hence, we set the weight of each constraint to 1. Subsequently, we sample constraints with probability proportional to their weights.

Weighted random sampling. We use weighted random sampling [13] to produce a sample of size $6k^2$ given a set M with n weighted constraints.

1. For each constraint $s_i \in M$ with weight w_i, set $u_i = random(0, 1)$ and $key_i = u_i^{1/w_i}$.
2. Select $6k^2$ items that have the largest values of keys (*i.e.*, key_i) as the sample.

Here, u_1 and u_2 are uniform random variables in (0,1). If $X_1 = u_1^{1/w_1}$ and $X_2 = u_2^{1/w_2}$, then $P(X_1 \leq X_2) = \frac{w_1}{w_1+w_2}$. Hence, selecting the largest $6k^2$ items is equivalent to sampling according to their weights.

Identifying the lucky iteration. The next task in avoiding the multi-set representation lies in identifying the lucky iteration. An invariant with our weight-based representation is that the sum of the weights of all constraints in M is equal to the cardinality of the multi-set.

The constraints in M can be divided into two categories: constraints violated by the sample solution (*i.e.*, $VIO_{x^*}(M)$) and constraints that are satisfied by the sample solution (*i.e.*, $SAT_{x^*}(M)$). To determine if an iteration is a lucky iteration, we need the number of violated constraints to be less than $1/3k$ of the cardinality of the multi-set of constraints. We compute the sum total of the weights of constraints that are satisfied by x^* and the sum total of weights of constraints not satisfied by x^*.

$$\sum_{v \in VIO_{x^*}(M)} v.w \leq \frac{1}{3k}\left(\sum_{v \in VIO_{x^*}(M)} v.w + \sum_{u \in SAT_{x^*}(M)} u.w\right)$$

After rearranging the terms, we have

$$\sum_{v \in VIO_{x^*}(M)} v.w \leq \frac{1}{3k-1}\sum_{u \in SAT_{x^*}(M)} u.w$$

Hence, if the sum of the weights of the violated constraints is less than $\frac{1}{3k-1}$ of the sum of the weights of the satisfied constraints, then it is a lucky iteration. Finally, the task of adding the violated constraint again to the set M is equivalent to doubling the weights of the violated constraints. Algorithm 2 presents our procedure for solving the sample, identifying whether the iteration is a lucky iteration, and updating the weights of the violated constraints.

This entire process repeats until we find a polynomial that satisfies all constraints (*i.e.*, when the system is full-rank) or produces a polynomial that violates at most a few points or exceeds the user-specified threshold for the number of iterations. When the algorithm exceeds the number of iterations without producing a polynomial, we increment the number of terms used for the smaller bitwidth representations in the progressive polynomial. We increase the number of terms used for the largest representation when we are unable to find a progressive polynomial after increasing the terms used for the smaller representations.

When the system of linear equations is full-rank, then the above procedure will find the unique polynomial. In many cases, the system of equations may not be full-rank for a given number of terms in the polynomial. Rather than increasing the number of terms, we also accept a polynomial that satisfies all constraints except a few constraints (*e.g.*, typically 1-4 inputs in our experiments). For some elementary functions, we also split the reduced inputs into two to four sub-domains and generate polynomials for them to reduce the number of terms.

3.4 A Sketch of the Proof

The proof that our algorithm finds the solution for a system of "full-rank" linear constraints and terminates in $6k \log n$ iterations in expectation immediately follows from the proof of the Clarkson's method [9]. We provide a sketch of the proof for completeness. Here, k is the number of terms in the largest representation for the progressive polynomial. The proof specifically relies on the following two lemmas.

Lemma 1. *There exist a set of k constraints such that if we find an optimal solution with respect to them, it will also be a feasible and optimal solution for the entire set of M constraints.*

This lemma holds because the optimum value of a linear program is always located on a vertex, which corresponds to k-strict constraints.

Lemma 2. *Suppose we have a multi-set M with n constraints. If we sample $6k^2$ constraints S from M and compute the optimum solution x^* on S, then with probability at least $1/2$, x^* can only violate $1/3k$ constraints in M.*

Proof that the algorithm solves the system in $6k \log n$ iterations in expectation. Let us consider the basis B for the optimal solution in M, which follows from Lemma 1. Here, $B \subseteq M$. Initially, B has k constraints as the rank of the system of linear constraints is k (*i.e.*, $|B| = k$). On every lucky iteration, we double the constraints violated in B (*i.e.* B is also a multi-set). After k lucky iterations, the number of constraints in B is at least $2k$. Similarly, the number of constraints in B is at least $2^2 \cdot k$ after $2k$ lucky iterations. Generalizing, the number of constraints in B is at least $2^i \cdot k$ after $k \cdot i$ lucky iterations. Hence, $|B| \geq 2^i \cdot k \geq 2^i$.

From Lemma 2, an iteration is lucky with probability $1/2$, where the solution x^* for the sample only violates $1/3k$ or fewer constraints in the multi-set M. Hence, M grows slowly. After $k \cdot i$ lucky iterations, size of the multi-set M is at most

$(1+1/3k)^{k\cdot i}\, n$. From the Taylor's series for e^x, we have $(1+x) \le e^x$ for all x. Hence, $(1+1/3k) \le e^{1/3k}$.

$$|M| \le (1+1/3k)^{k\cdot i}\, n \le \left(e^{1/3k}\right)^{k\cdot i} n \le e^{i/3} n$$

The above two properties imply that there cannot be many lucky iterations without finding a solution x^* that satisfies all constraints. In our setting, multi-set B is a subset of multi-set M. After $k \cdot i$ lucky iterations, the cardinalities of the sets B and M should satisfy $2^i \le ne^{i/3}$. When is $i \ge 3 \log n$, the above inequality is no longer true. Since we are exploring $k \cdot i$ lucky iterations, the algorithm will terminate after $3k \log n$ lucky iterations. Finally each iteration is lucky with probability at least 1/2 from Lemma 2. So, the algorithm terminates by finding a solution that satisfies all constraints after $6k \log n$ iterations in expectation.

Proof of Lemma 2. To construct the proof for this lemma, consider an artificial way of sampling as follows: we first sample $r+1$ constraints S' from M and then throw one of them out uniformly at random to get S. Here, r is the size of the sample S. This way of sampling S' has the same distribution as the original distribution of S. Let $X(S)$ be the number of violated constraints when we sample S. For any constraint $h \in M$, let $X(h, S) = 1$ if and only if constraint h is violated by the optimum solution x^* computed on S. Then, the expected value of $X(S)$ is:

$$E[X(S)] = \sum_S Prob(S) \sum_{h \notin S} X(h,S) = \frac{1}{\binom{|M|}{r}} \sum_S \sum_{h \notin S} X(h,S)$$

because the choice of S is uniform over all r-subsets of M. But interestingly, from our artificial way of sampling, we can also write:

$$\sum_S \sum_{h \notin S} X(h,S) = \sum_{S'} \sum_{h \in S'} X(h, S'-h)$$

Here, S consists of all r-subsets of M and S' consists of all $r+1$-subsets of M.

To understand when $X(h, S'-h) = 1$, fix a basis of S' (as in Lemma 1). Then, $X(h, S'-h) = 1$ only when h belongs to this basis. But there are only k choices of vectors in this basis! So most of the time, the second summand is 0. In particular, we have:

$$E[X(S)] \le \frac{1}{\binom{|M|}{r}} \sum_{S'} k$$

Since the number of choices for S' is $\binom{|M|}{r+1}$, so in total, we have:

$$E[X(S)] \le \frac{\binom{|M|}{r+1}}{\binom{|M|}{r}} k < k \frac{|M|}{r+1}$$

By Markov inequality, the probability that the value of $X(S)$ is at least twice its expectation is at most 1/2. Hence, we have:

$$Prob\left(X(S) > 2k\frac{|M|}{r+1}\right) < \frac{1}{2}$$

Recall that we would like $X(S)$ to be at least $|M|/3k$. To make $\frac{2k|M|}{r+1} < \frac{|M|}{3k}$, we can pick $r = 6k^2$, which is the size of the sample, so that the probability of a lucky iteration is at least 1/2.

4 Experimental Evaluation

We describe the RLIBM-Prog prototype, methodology, and the results of our experiments to check both the correctness and performance of our elementary functions.

Prototype. Our prototype, RLIBM-Prog [1], is a progressive polynomial generator and a collection of correctly rounded elementary functions. RLIBM-Prog contains multiple implementations for ten elementary functions. A single progressive 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 with all five rounding modes in the IEEE standard. It also has progressive performance with bfloat16 and tensorfloat32 types and produces correctly rounded results for all inputs with them. Correct and fast polynomial approximations generated by RLIBM-Prog for $ln(x)$, $log_2(x)$, and $log_{10}(x)$ are already part of LLVM's math library [30–32].

RLIBM-Prog uses the MPFR library [14] to compute the oracle value of $f(x)$ for each representation. It uses an exact rational LP solver, SoPlex [15], to solve constraints. We use range reduction and output compensation functions from our prior work in the RLibm project [25, 26, 29]. While evaluating the progressive polynomial, the bfloat16 and tensorfloat32 inputs use only the first few terms of the progressive polynomial. We perform polynomial evaluation, range reduction, and output compensation using double precision. We use Horner's method to evaluate polynomials [3].

Methodology. We compare RLIBM-Prog's functions with state-of-the-art libraries: Intel's double libm, glibc's double libm, CR-LIBM [11], and RLibm-All. Intel's and glibc's libm are mainstream libraries that are widely used for their performance but do not provide correctly rounded results for all inputs with any one rounding mode. CR-LIBM provides separate implementations for each rounding mode for an elementary function that produce the correctly rounded results for double precision. It has implementations for four out of the five rounding modes in the IEEE standard and does not have an implementation for the round-to-nearest-ties-to-away mode. RLibm-All produces correctly rounded

Table 1. Details of the polynomials generated by RLIBM-Prog in comparison to RLibm-All. For each function generated, we show the size of the piecewise polynomial, the maximum degree, and the number of terms (for bfloat16, tensorfloat32, and float types) in the polynomial. We also report the number of special case inputs to avoid increasing the degree of the polynomial approximation with RLIBM-Prog, and the size of the lookup tables for the coefficients of the generated polynomial approximations in bytes. We report the total reduction in memory for the lookup tables computed with RLIBM-Prog in comparison to RLibm-All.

	RLibm-All				**RLIBM-Prog**									
$f(x)$	# of poly.	Degree	# of terms	Poly. mem. use (B)	# of poly.	Max. poly. degree			# of terms			# of special inputs	Poly. mem. use (B)	**Reduction in mem vs. RLibm-All**
						FP32	TF32	BF16	FP32	TF32	BF16			
$\mathbf{ln(x)}$	2^{10}	3	3	24576	4	5	5	0	5	5	0	13	360	**68×**
$\mathbf{log_2(x)}$	2^8	3	3	6144	1	5	3	0	5	3	0	0	40	**154×**
$\mathbf{log_{10}(x)}$	2^8	3	3	6144	4	6	3	0	6	3	0	3	216	**28×**
$\mathbf{e^x}$	2^8	4	5	10240	4	6	4	3	7	5	4	0	160	**64×**
$\mathbf{2^x}$	2^8	3	4	8192	1	6	3	2	7	4	3	0	56	**146×**
$\mathbf{10^x}$	2^9	3	4	16384	4	6	4	3	7	5	4	1	176	**93×**
$\mathbf{sinh(x)}$	$2^6, 2^5$	5, 4	3, 3	2304	1, 1	5, 4	3, 2	3, 2	3, 3	2, 2	2, 2	1, 1	80	**29×**
$\mathbf{cosh(x)}$	$2^6, 2^5$	5, 4	3, 3	2304	1, 1	5, 4	3, 2	3, 2	3, 3	2, 2	2, 2	0, 0	48	**48×**
$\mathbf{sinpi(x)}$	$2^2, 2^2$	5, 4	3, 3	192	1, 1	5, 4	3, 2	3, 2	3, 3	2, 2	2, 2	0, 0	48	**4×**
$\mathbf{cospi(x)}$	$2^2, 2^2$	5, 4	3, 3	192	1, 1	5, 4	3, 2	3, 2	3, 3	2, 2	2, 2	0, 0	48	**4×**

results for all n-bit FP representations and all five rounding modes, where $10 \leq n \leq 32$.

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. We use the publicly available CR-LIBM and RLibm-All versions. We use Intel's double libm from the oneAPI Toolkit and glibc's double libm from glibc-2.31. The test harness for comparing glibc's libm, CR-LIBM, and RLibm-All is built using the gcc-9.3.0 compiler with `-O0 -frounding-math -fsignaling-nans` flags. The test harness for comparing against Intel's libm is built using the `icc` compiler with `-O0 -fp-model strict -no-ftz` flags because Intel's libm is only available in the Intel's compiler. The performance is measured using the number of cycles taken to compute the result for each input using `rdtscp`. Then, we computed the total time taken to compute the elementary functions for all inputs.

Properties of RLIBM-Prog's polynomials. Table 1 provides details on the various properties of the polynomial approximations generated by RLIBM-Prog in comparison to RLibm-All. With RLIBM-Prog, we tried to generate progressive polynomials with the lowest degree with at most four sub-domains and with at most four special case inputs per sub-domain (*i.e.*, when the system is not full-rank). We chose these thresholds because they can be implemented efficiently with simple branches. The range reduction strategy for $sinh(x)$, $cosh(x)$, $sinpi(x)$, and $cospi(x)$ requires approximations of two functions. We generated two polynomial approximations for each elementary function.

Significant reduction in memory usage. In contrast to RLibm-All, RLIBM-Prog generates a single polynomial or a piecewise polynomial with at most 4 sub-domains. RLIBM-Prog's polynomials require only 123 bytes on average per function. In contrast, RLibm-All's polynomials need 7667 bytes (7.5KB) on average per function. RLIBM-Prog's polynomials reduce total storage needs by 62× on average compared to RLibm-All.

RLIBM-Prog was able to generate a single progressive polynomial that produces correctly rounded results without any special case inputs for $log_2(x)$, 2^x, $cosh(x)$, $sinpi(x)$, and $cospi(x)$, which implies that the system is full-rank. When we experimented with RLibm-All's polynomial generation, it was not able to generate a single polynomial for all functions except $log_2(x)$. RLIBM-Prog generates these progressive polynomials very quickly: only 19 minutes on average per function. This shows the effectiveness of the RLIBM-Prog's fast randomized algorithm for solving the set of constraints.

Terms needed by bfloat16 and tensorfloat32. When RLIBM-Prog generates progressive polynomials, it indicates the number of terms necessary to evaluate to produce correctly rounded results for the bfloat16 and the tensorfloat32 types. Table 1 also reports the number of terms that we need to evaluate in the progressive polynomial to produce the correctly rounded bfloat16 and the tensorfloat32 results. Surprisingly, a single term (first term) is sufficient to produce correctly rounded results for all bfloat16 inputs with $ln(x)$, $log_2(x)$, and $log_{10}(x)$. In contrast, RLibm-All's functions for $ln(x)$, $log_2(x)$, and $log_{10}(x)$ have to evaluate a degree-3 polynomial to produce correctly rounded bfloat16 results. The number of terms needed for bfloat16 and tensorfloat32 are lower than the terms needed for computing correctly

Table 2. This table reports whether a library produces correctly rounded results for all inputs using RLIBM-PROG, glibc's double libm, Intel's double libm, CR-LIBM, and RLIBM-ALL. Each sub-column also reports the ability to generate correctly rounded results for (1) bfloat16 and tensorfloat32 results with the *rn* mode, (2) 32-bit float results with the *rn* mode, and (3) 32-bit float results with all five rounding modes. ✓indicates that the library produces correctly rounded results for the given representation for all inputs. Otherwise, we use ✗.

$f(x)$	**RLIBM-PROG**			glibc double libm			Intel double libm			CR-LIBM			RLIBM-ALL		
	BF16 & TF32 rn	FP32 *rn*	FP32 all rm	BF16 & TF32 rn	FP32 *rn*	FP32 all rm	BF16 & TF32 rn	FP32 *rn*	FP32 all rm	BF16 & TF32 rn	FP32 *rn*	FP32 all rm	BF16 & TF32 rn	FP32 *rn*	FP32 all rm
$\mathbf{ln(x)}$	✓	✓	✓	✓	✗	✗	✓	✗	✗	✓	✗	✗	✓	✓	✓
$\mathbf{log_2(x)}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\mathbf{log_{10}(x)}$	✓	✓	✓	✓	✗	✗	✓	✗	✗	✓	✗	✗	✓	✓	✓
$\mathbf{e^x}$	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓
$\mathbf{2^x}$	✓	✓	✓	✓	✗	✗	✓	✗	✗	N/A	N/A	N/A	✓	✓	✓
$\mathbf{10^x}$	✓	✓	✓	✓	✓	✗	✓	✓	✗	N/A	N/A	N/A	✓	✓	✓
$\mathbf{sinh(x)}$	✓	✓	✓	✓	✗	✗	✓	✗	✗	✓	✗	✗	✓	✓	✓
$\mathbf{cosh(x)}$	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓
$\mathbf{sinpi(x)}$	✓	✓	✓	N/A	N/A	N/A	✓	✓	✗	✓	✓	✗	✓	✓	✓
$\mathbf{cospi(x)}$	✓	✓	✓	N/A	N/A	N/A	✓	✓	✗	✓	✓	✗	✓	✓	✓

rounded results for the 34-bit float with the round-to-odd mode except where tensorfloat32 needs all terms for $ln(x)$.

Does RLIBM-PROG produce correct results? Table 2 reports the summary of our evaluation to check whether RLIBM-PROG and other existing libraries produce correctly rounded results for various representations and rounding modes. All libraries produce correctly rounded results for bfloat16 and tensorfloat32 results using the round-to-nearest-ties-to-even (*rn*) mode. Glibc's double libm, Intel's double libm, and CR-LIBM do not produce correctly rounded results for 32-bit float inputs for several elementary functions and various rounding modes. Even though CR-LIBM is a correctly rounded library for double precision, it produces wrong results when it is re-purposed for 32-bit floats due to double rounding errors. Both RLIBM-PROG and RLIBM-ALL produce correctly rounded float results for all inputs and all standard rounding modes. More importantly, RLIBM-PROG is able to produce correctly rounded bfloat16 and tensorfloat32 results even when evaluating only the first few terms of the generated progressive polynomial approximations.

Performance evaluation of RLIBM-PROG. Figure 4 reports the speedup obtained with RLIBM-PROG's functions when compared to various state-of-the-art libraries. Figure 4(a) presents the speedup of RLIBM-PROG's bfloat16 functions (left bar in each cluster), tensorfloat32 functions (middle bar in each cluster), and float functions (right bar in each cluster) over glibc's double libm. On average, RLIBM-PROG's bfloat16, tensorfloat32, and float functions are 42%, 29%, and 20% faster over glibc's double library, respectively. Similarly, Figure 4(b) presents the speedup of RLIBM-PROG's functions over Intel's double library. On average, RLIBM-PROG's bfloat16, tensorfloat32, and float functions are 74%, 64%, and 49% faster over Intel's double math library. Intel's double library produces more accurate results compared to glibc's double library and is slightly slower compared to glibc's double library. Hence, RLIBM-PROG has more speedup over Intel's double library compared to glibc's double library.

Figure 4(c) reports the speedup with RLIBM-PROG when compared to CR-LIBM. On average, RLIBM-PROG's bfloat16, tensorfloat32, and float functions are 123%, 105%, and 85% faster over CR-LIBM functions.

Figure 4(d) shows the speedup of RLIBM-PROG's functions over RLIBM-ALL. On average, RLIBM-PROG's bfloat16, tensorfloat32, and float functions have 25%, 16%, and 5% speedup over RLIBM-ALL. While RLIBM-PROG and RLIBM-ALL's functions produce correctly rounded results for all inputs, glibc's double libm, Intel's double libm, and CR-LIBM are slower and do not produce correctly rounded results for all inputs.

RLIBM-PROG generates significantly smaller piecewise polynomial approximations compared to RLIBM-ALL, which results in fewer memory accesses, producing speedups even with the float functions. RLIBM-ALL's $ln(x)$ function has a piecewise polynomial of 2^{10} sub-domains whereas RLIBM-PROG's $ln(x)$ function has a piecewise polynomial with 4 sub-domains. Hence, RLIBM-PROG's float functions for $ln(x)$ are 11% faster over RLIBM-ALL. Similarly, RLIBM-PROG's $sinh(x)$ function uses two single polynomials compared to RLIBM-ALL's piecewise polynomials with sizes of $2^6 + 2^5$ (*i.e.*, 96) sub-domains. Hence, RLIBM-PROG's $sinh(x)$ reports 11% speedup over RLIBM-ALL.

Even though the degree of the piecewise polynomials are smaller with RLIBM-ALL when compared to RLIBM-PROG for e^x and 2^x, RLIBM-PROG's functions are 1% and 2% faster because the benefit from storing fewer coefficients subsumes the overhead of evaluating a higher degree polynomial.

Progressive performance. Our performance evaluation demonstrates that RLIBM-PROG's progressive polynomial approximations have better performance for bfloat16 and tensorfloat32 types when compared to the float type. RLIBM-PROG's bfloat16 functions show the highest speedup followed by tensorfloat32, highlighting the progressive nature. RLIBM-PROG's $ln(x)$, $log_2(x)$, and $log_{10}(x)$ functions for bfloat16 are 60%, 61%, and 50% faster over RLIBM-ALL functions, respectively. Although RLIBM-ALL produces correctly rounded

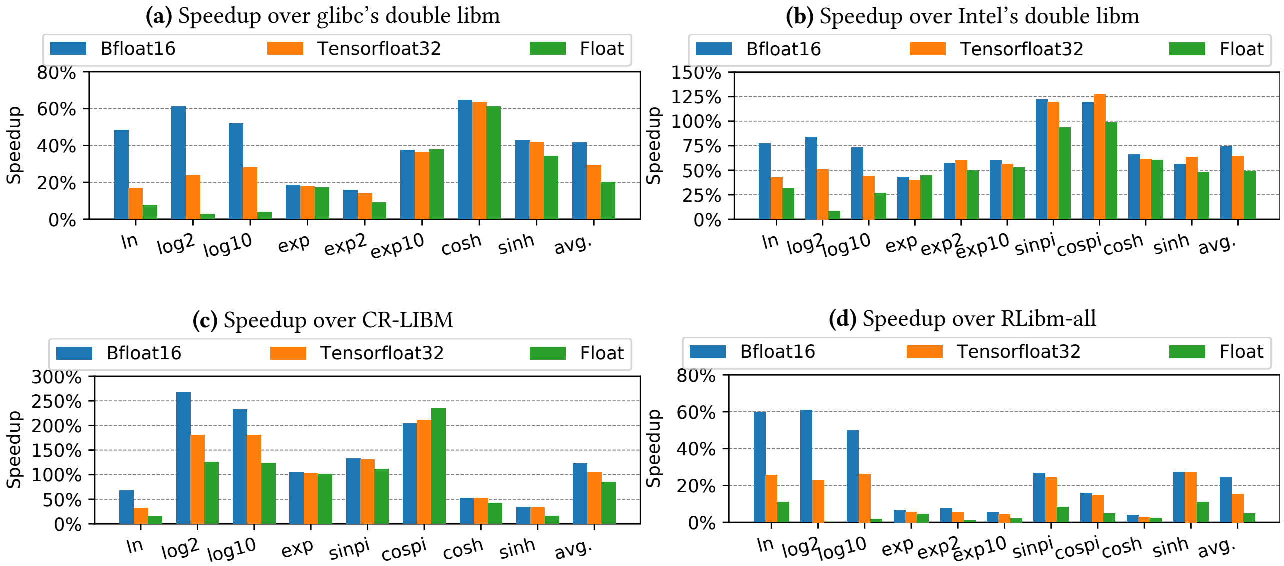

Figure 4. Computational speedup of RLIBM-Prog's progressive polynomial approximations in comparison to (a) glibc's double libm, (b) Intel's double libm, (c) CR-LIBM, and (d) RLibm-All. The left-most bar, middle bar, and the right-most bar in each cluster measures the speedup of RLIBM-Prog's bfloat16, tensorfloat32, and float elementary functions implemented as a progressive polynomial approximation.

results for all bfloat16 inputs, it requires evaluating the entire polynomial that results in some performance loss. In summary, RLIBM-Prog produces a single progressive polynomial approximation that produces correctly rounded results for all inputs with multiple representations and multiple rounding modes. Its float functions are faster than state-of-the-art math libraries. Furthermore, smaller representations are significantly faster demonstrating progressive performance.

5 Related Work

Approximating and validating elementary functions is a well-studied problem [4, 5, 11, 14, 17–21, 34, 37, 38, 44, 45], which has been feasible because of advances in range reduction [2, 10, 39, 41–43]. A number of correctly rounded math libraries have also been developed [11, 25, 26, 45]. A detailed survey is available in Muller's seminal book [34]. We restrict our discussion to the most closely related work.

CR-LIBM [11, 22] is a correctly rounded double library that provides implementations for a subset of the rounding modes. CR-LIBM relies on Sollya [7] to generate near minimax polynomial approximations. CR-LIBM computes and proves the error bound on the polynomial evaluation using interval arithmetic [6, 8]. Double rounding errors can cause wrong results when the CR-LIBM's result is rounded to a 32-bit float.

This paper is closely related to our prior work in the RLibm project [23, 25, 26, 29]. Like the prior work in the RLibm project, we also approximate the correctly rounded result using an LP formulation. We also use RLibm's range reduction strategies. We use the idea of creating a single polynomial approximation that produces correctly rounded results for multiple representations and rounding modes from RLibm-All [29]. We advance ideas from the RLibm project by generating faster polynomial approximations with a novel method for solving linear constraints that provide progressive performance with smaller bitwidth representations.

6 Conclusion

This paper proposes a novel type of polynomial approximations, termed *progressive polynomials*, that produce correctly rounded results for multiple representations and rounding modes. An elegant property of the progressive polynomial is that evaluating the first few terms produces correctly rounded results for smaller representations. To generate such progressive polynomials, we propose a fast algorithm for polynomial generation that generates an order of magnitude smaller lookup tables than the state-of-the-art method. RLIBM-Prog's polynomials are faster than all mainstream and/or correctly rounded libraries. We have already incorporated a few polynomial approximations from this project in mainstream libraries. We believe this is the next logical step in mandating correctly rounded elementary functions at least for representations up to 32-bits.

Acknowledgments

We thank Sepehr Assadi for his assistance with the proof of Clarkson's algorithm. We thank our shepherd Fan Long and the PLDI reviewers for their feedback on the paper. This material is based upon work supported in part by the National Science Foundation under Grant No. 1908798, Grant No. 2110861, and Grant No. 1917897.

References

[1] Mridul Aanjaneya, Jay P. Lim, and Santosh Nagarakatte. 2022. *RLibm-Prog*. https://github.com/rutgers-apl/rlibm-prog

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

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

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

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

[6] 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

[7] 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

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

[9] Kenneth L. Clarkson. 1995. Las Vegas Algorithms for Linear and Integer Programming When the Dimension is Small. *J. ACM* 42, 2 (March 1995), 488–499. https://doi.org/10.1145/201019.201036

[10] 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

[11] 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

[12] 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

[13] Pavlos S. Efraimidis and Paul G. Spirakis. 2006. Weighted random sampling with a reservoir. *Inform. Process. Lett.* 97, 5 (2006), 181–185. https://doi.org/10.1016/j.ipl.2005.11.003

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

[16] 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

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

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

[19] 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)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer-Verlag, Munich, Germany, 60–66. https://doi.org/10.1007/978-3-642-03359-9_4

[20] 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

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

[22] 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

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

[24] 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.

[25] 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

[26] 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

[27] 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.

[28] 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.

[29] 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

[30] Tue Ly. 2021. *Implement correctly rounded logf based on RLIBM library*. https://github.com/llvm/llvm-project/commit/d08a801b5f5678af52c89d202e5f22e0d43a38cd

[31] Tue Ly. 2022. *Implement correctly rounded log10f based on RLIBM library*. https://github.com/llvm/llvm-project/commit/e581841e8cf46109acea92e1acb661c404fa62b9

[32] Tue Ly. 2022. *Implement correctly rounded log2f based on RLIBM library*. https://github.com/llvm/llvm-project/commit/63d2df003e9c198bfa70d448f8ad7b361cbb1351

[33] Nimrod Megiddo. 1984. Linear Programming in Linear Time When the Dimension Is Fixed. *J. ACM* 31, 1 (Jan. 1984), 114–127. https://doi.org/10.1145/2422.322418

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

[35] Santosh Nagarakatte, Mridul Aanjaneya, and Jay P. Lim. 2022. *The RLIBM Project.* https://www.cs.rutgers.edu/~santosh.nagarakatte/rlibm/

[36] 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/

[37] 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.

[38] 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.*

[39] 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

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

[41] 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

[42] 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

[43] 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

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

[45] 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