NOVEL POLYNOMIAL APPROXIMATION METHODS FOR GENERATING CORRECTLY ROUNDED ELEMENTARY FUNCTIONS

By

JAY PHIL LIM

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Santosh Nagarakatte

And approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2021

ABSTRACT OF THE DISSERTATION

Novel Polynomial Approximation Methods for Generating Correctly Rounded Elementary Functions

by **JAY PHIL LIM**

Dissertation Director:

Prof. Santosh Nagarakatte

All endeavors in science use math libraries to approximate elementary functions (*e.g.* $ln(x)$ or e^x). Unfortunately, mainstream math libraries for floating point (FP) representations do not produce correctly rounded results for all inputs. In addition, given the importance of FP performance in numerous domains, several new variants of FP and its alternatives have been proposed (*e.g.*, bfloat16, tensorfloat32, posits). These representations do not have correctly rounded math libraries. This dissertation proposes the RLIBM approach, a collection of novel techniques for generating polynomial approximations that produce correctly rounded results of an elementary function $f(x)$.

Existing approaches use polynomial approximations that approximate the real value of $f(x)$. The minimax approach generates polynomials that minimize the maximum approximation error compared to the real value of $f(x)$ across all inputs. However, minimax polynomials produce wrong results due to approximation errors and rounding errors in the implementation. In contrast, the RLIBM approach makes a case for generating polynomials that approximate the correctly rounded result of $f(x)$ (*i.e.*, the exact value of $f(x)$ computed in reals and rounded to the target representations). This approach provides more freedom in generating efficient polynomials that produce correctly rounded results for all

inputs.

Additionally, this dissertation makes the following contributions. First, we show that the problem of generating polynomials that produce correctly rounded results can be structured as a linear programming problem. Second, the RLIBM approach accounts for numerical errors that occur from range reduction by automatically adjusting the amount of freedom available to generate polynomials. The generated polynomial with RLIBM produces the correctly rounded results for all inputs. Third, we propose a set of techniques to develop correctly rounded elementary functions for 32-bit types. Specifically, we generate efficient piecewise polynomials using counterexample guided polynomial generation and bit-pattern based domain splitting. Finally, we extend the RLIBM approach to generate a single polynomial approximation that produces the correctly rounded results for multiple rounding modes and multiple precision configurations. To generate correctly rounded elementary functions for n-bit type, our key idea is to generate a polynomial approximation for a $(n + 2)$-bit representation using the *round-to-odd* mode. We provide formal proof that the resulting polynomial will produce the correctly rounded results for all standard rounding modes and for multiple representations with k bits where $k \leq n$.

Using the RLIBM approach, we have developed several implementations of elementary functions for various representations and rounding modes. These elementary functions produce the correctly rounded results for all inputs in the target representations. Additionally, the functions are faster than mainstream math libraries which have been optimized for decades. Our RLIBM-ALL prototype, a collection of elementary functions that produce the correctly rounded results for multiple floating point representations with all standard rounding modes, has $1.1\times$, $1.3\times$, and $1.9\times$ speedup over glibc, Intel, and CR-LIBM math libraries, respectively. Our prototypes also provide the first correctly rounded elementary functions for 32-bit posit.

ACKNOWLEDGMENTS

First and foremost, this dissertation would have not been possible without my advisor Santosh Nagarakatte, who has been my mentor, collaborator, and role model. His support and guidance has made a tremendous impact in my journey to become a researcher. Santosh has spent countless number of hours with me reading and discussing papers to teach me how to think rigorously and critically. Our time together brainstorming ideas have helped me develop my problem solving skills: simpler is better! He taught me to be compassionate and caring for students. In times of trouble, Santosh has supported me financially and psychologically to ensure that I can focus on growing as a researcher. Without Santosh's support, I would not have been able to begin, let alone complete, this dissertation.

I would like to thank my dissertation committee members, Ulrich Kremer, Richard Martin, and Zachary Tatlock. Their insight and feedback have been extremely helpful in improving this dissertation.

Many insights and ideas in this dissertation would have not been realized without my collaborators, John Gustafson and Mridul Aanjaneya. Our discussions with John Gustafson regarding the Minefield technique and the round-to-odd rounding mode have been crucial in developing the key insights of the RLIBM approach. The realization from Mridul Aanjaneya that generating polynomials is a linear programming problems have been instrumental in automating the RLIBM approach. Although not direct collaborators, I have learned tremendously from the members of the FPBench community on topics related to floating point. My gratitude goes especially towards Zachary Tatlock, Pavel Panchekha, and Bill Zorn for organizing and leading the monthly meetings and yearly workshops. I would also like to extend my thanks to my mentor David Tarditi during my internship in MSR Redmond, who has been instrumental in my development in the foundation of programming languages.

I have been shaped by many great people in Rutgers University throughout my PhD

career. I thank Vinod Ganapathy for noticing my potential and mentoring me in the early years, along with Santosh. I learned the importance of simplifying complex ideas through my interactions with Ulrich Kremer and Badri Nath. It was always fun to talk to Richard Martin and learn about the history of computer science.

I would like to thank my lab-mates, David Menendez, Adarsh Yoga, Nader Boushehrinejad, Mohammedreza Solyaniyeh, Sangeeta Chowdharay, Harishankar Vishwanathan, and Matan Shachnai, at the RAPL group for their support. I would like to give special thanks to Adarsh who has been like a big brother since the early years of my PhD as well as Sangeeta and Matan, my collaborators in different projects. I also met great peers outside of RAPL group including Georgiana, Daehan, Hai, Daeyoung, Liu, Hong-yu, Jaewoo, Sunghyun, and David. In stressful times of PhD, they brought fun and gave me motivation to continue.

I am deeply thankful for my family, my wife, daughter, parents, and parents-in-law for their moral support throughout the years. My wife Inyoung has always been my greatest rival and supporter who has constantly pushed me to be at my best. My daughter Sammie has been the light of my life which motivates me to continue. I thank my parents, Tae-Seop and Myung, for the educational and emotional support they provided throughout my life. I would also like to thank my friends, Dexter, Carolyn, Shinjae, Daehoon, Jisoo, Bong, Grace, James, Inah, Nick, Joonsang, Paul, Kimun, Kay, Hongjin, and Ben for all the fun activities that provided refreshing breaks from work.

To the loves of my life, Inyoung and Sammie

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Every programming language has primitive data types to approximate real numbers. The two important attributes for such datatypes are the ability to represent a wide range of values (dynamic range) and represent a particular real number precisely (precision). The floating point (FP) representation, standardized by the IEEE-754 standard in 1985 [27], is a widely used primitive datatype to represent real numbers for its large dynamic range and reasonable precision. For example, every number in JavaScript used to be represented with double (a 64-bit FP representation), until in 2020 when JavaScript introduced the `BigInt` datatype to represent large integer values [68].

In recent years, the performance of FP arithmetic operations is becoming important, especially in machine learning and high performance computing domains. Hence, modern accelerators, processors, and systems have explored new variants of FP representations that tweak the bit-length, dynamic range, and precision [44, 55, 58, 74, 78, 100, 108, 112, 116, 130, 131]. Some of these variants are precision configurations of the standard FP representation (*i.e.* bfloat16 [131] and TensorFloat32 [108]). Several new representations use the same strategy as FP to encode the values but store the bit-pattern in different ways (*i.e.*, MSFP12 [116] and FlexPoint [78]). There are also completely new representations (*i.e.*, log number systems [44, 112, 130] or posit [55, 58]). In particular, posits can provide a wider dynamic range than FP for a given number of bits. Additionally, posits provide more precision when representing values near 1, known as tapered precision. The new variants of FP also aim towards minimizing the bit-length for efficient hardware implementation of primitive operations [66, 69, 70, 107, 136, 142] while maintaining sufficient amount of dynamic range and precision.

Any number system that approximates real numbers needs a math library that provides

implementations of elementary functions [103] (*e.g.*, $sin(x)$, $ln(x)$, e^x). Elementary functions are widely used in scientific domains ranging from machine learning to scientific simulation. For example, the exponential function e^x is used widely to model the sigmoid function (*i.e.*, $\frac{e^x}{e^x+1}$) and the fourier transformation in digital signal processing. As elementary functions are essential building blocks for scientific applications, efficiently and accurately computing the result of an elementary function $f(x)$ is paramount. The IEEE-754 standard [27] defines a list of elementary functions and recommends math libraries implement them to produce the correctly rounded result.

Given a representation $\mathbb{T}$ with finite precision (*e.g.*, float), the correctly rounded result of an elementary function $f(x)$ for an input $x \in \mathbb{T}$ is defined as the value of $f(x)$ computed with real numbers and rounded to a value in the representation $\mathbb{T}$. Developing correctly rounded math libraries is a challenging problem. Hence, there have been seminal work on generating approximations of elementary functions [4, 11, 17, 20, 21, 22, 71, 80, 99], verifying the correctness of math libraries [8, 35, 38, 39, 59, 60, 62, 83, 119], and repairing math libraries [145]. Further, there are correctly rounded math libraries for the float and double types [29, 65, 101, 146] for some rounding modes. Widely used math libraries [51, 67] do not produce correctly rounded results for all inputs.

The new FP representations currently do not have math libraries. A quick and naive way to address this problem is to repurpose an existing math library. For example, to approximate e^x for bfloat16, we can use an implementation of e^x designed for 32-bit float. We can convert a bfloat16 input to a float value, use an existing implementation of e^x to produce the float result, and round the output back to bfloat16. This approach is appealing if a correctly rounded math library for a representation with significantly higher precision compared to the target representation is available. However, it can produce wrong results due to the double rounding error. A Double rounding error can occur when one rounds a real value to an intermediate representation and subsequently rounds the intermediate result to the target representation, known as double rounding. Depending on the rounding

modes used in each rounding operation, double rounding may result in a different value compared to directly rounding the real value to the target representation, which is called the double rounding error. As a correctly rounded math library effectively rounds the real value of $f(x)$ to its target representation, rounding this result a smaller representation can cause double rounding errors and produce wrong results. Chapter 2 provides more details on the double rounding error and why repurposing an existing library fails to produce the correctly rounded result. The repurposed math library is intended for a larger representation and computes the result with significantly more precision than necessary. Hence, its performance may not be ideal.

The most common method to approximate an elementary function $f(x)$ is with a polynomial approximation. Prior approaches generate polynomials that minimizes the maximum error among all inputs compared to the real value of $f(x)$, which is known as the minimax approach. This approach is based on the Weierstrass approximation theorem [144] which states that if $f(x)$ is a continuous real function over a domain $[a, b]$, then there exists a polynomial $P(x)$ where the error with respect to $f(x)$ is bounded, *i.e.*, $|f(x) - P(x)| < \epsilon$ for all real numbers $x \in [a, b]$ where $\epsilon > 0$. The Chebyshev alternating theorem [137] provides the condition for such a polynomial. A minimax polynomial of degree d has exactly $d+2$ inputs with the absolute maximum error $|f(x) - P(x)|$ and the error for each of these inputs alternate in sign. Finally, Remez algorithm [114] provides the procedure to identify the minimax polynomial of degree d with real numbers.

Polynomial approximations are efficient and accurate when approximating $f(x)$ in a small domain $[a, b]$. To approximate elementary functions for the entire input domain in the target representation $\mathbb{T}$, polynomials are often used with range reduction that reduces the input domain to a smaller domain. A typical range reduction strategy has three components. First, the range reduction function reduces the original input $x \in \mathbb{T}$ to a reduced input x' in a small domain $[a, b]$. The polynomial then approximates the elementary function $f(x')$ using the reduced input x'. Finally, the result of the polynomial approximation,

which approximates for the reduced input, is adjusted with the output compensation function to produce the result for the original input x. Some strategies (*e.g.*, Cody and Waite's range reduction for logarithm functions [25]) transforms the original $f(x)$ into a different function $g(x)$ where the polynomial approximation is more efficient (*i.e.*, even or odd polynomial). The most commonly used strategy in mainstream math libraries uses a combination of mathematical identities and look-up tables to efficiently evaluate the output compensation function [133, 134, 135]. To produce accurate results for $\mathbb{T}$, the range reduction, polynomial approximation, and output compensation functions are evaluated in a representation $\mathbb{H}$ that has higher precision compared to $\mathbb{T}$.

There are several challenges in generating efficient polynomial approximations that produce correctly rounded results for all inputs using the minimax approach. These challenges arise due to the fundamental difference between the goal of correctly rounded math libraries and the minimax approach. Correctly rounded math libraries must produce the correctly rounded result of $f(x)$. In comparison, the goal of the minimax approach is to approximate the real value of $f(x)$ as closely as possible. Although the correctly rounded result of $f(x)$ is computed by rounding the real value of $f(x)$ to the target representation, rounding a value close to (but not exactly equal to) $f(x)$ does not guarantee to produce the same result. Hence, the error bound of the polynomial $P(x)$ generated with the minimax approach compared to $f(x)$ may need to be arbitrarily small. Consider an input x_i where the exact result of $f(x_i)$ with real numbers is close to the rounding boundary of two values in $\mathbb{T}$ (*i.e.*, $f(x_i)$ rounds to a value v_1 but $f(x_i) + \epsilon$ rounds to a different value v_2 for an input x_i and a small value ϵ). To produce the correctly rounded result of $f(x_i)$, the error of the polynomial approximation $P(x_i)$ must be smaller than ϵ. This likely requires a large degree polynomial with many terms, resulting in less than ideal performance. Also, there is no known general method to analyze and predict the error bound of $P(x)$ that guarantees to produce the correctly rounded result for all inputs in $\mathbb{T}$. This problem is widely known as *table-maker's dilemma* [75]. The table maker's dilemma states that there is no general

method to predict how accurate an approximation of $f(x)$ must be to produce the correctly rounded result for an arbitrary input. The only way to determine the error bound is by iteratively computing the real value of $f(x)$ for each input x [85].

Additionally, there is a disparity between the settings assumed by the minimax approach and correctly rounded math libraries. The error bound guaranteed by the Remez algorithm only holds if the polynomial is evaluated with real numbers. However, math library functions are evaluated with finite precision representations, which causes numerical errors when evaluating the polynomial, range reduction, and output compensation functions. To produce correctly rounded results for all inputs, this necessitates the polynomial approximation to account for numerical errors and the internal computations may need to be evaluated in a significantly high precision representation. For example, several parts of the elementary functions in CR-LIBM [30], a correctly rounded math library for the 64-bit double type (with 53 precision bits), are implemented with the *triple-double* representation where each intermediate value is represented with three double values (a total of 159 precision bits). In addition, CR-LIBM formally proves that the polynomial they generate will produce correctly rounded results for all inputs even with numerical errors.

1.1 Dissertation Statement

To generate efficient and correctly rounded polynomial approximations of an elementary function $f(x)$, this dissertation makes a case for generating polynomials that approximate the correctly rounded result of $f(x)$. In comparison, mainstream math libraries generate polynomials that approximate the real value of $f(x)$ using the minimax approach. Approximating the correctly rounded results provide more freedom in generating polynomials that produce correctly rounded results for all inputs, resulting in efficient polynomials.

1.2 Contributions of the Dissertation

This dissertation proposes the RLIBM approach, a set of techniques to generate efficient polynomial approximations that produce correctly rounded results of an elementary function $f(x)$ for various representations. The contributions of this dissertation can be summarized as follows:

1. We make a case for approximating the correctly rounded result of $f(x)$ rather than the real value of $f(x)$. We demonstrate that the task of generating polynomials can be framed as a linear programming (LP) problem. Our approach accounts for the numerical errors when evaluating range reduction and output compensation functions. Our approach guarantees that the resulting polynomials produce the correctly rounded results for all inputs.

2. To scale our approach to 32-bit types, we present counterexample guided polynomial generating with sampling. Additionally, we generate efficient piecewise polynomials by splitting the input domain using few bits in the bit-pattern of the input.

3. We propose a novel approach to generate a single polynomial approximation that produces the correctly rounded results for multiple representations and rounding modes by using the *round to odd* mode.

In the remainder of this section, we provide a brief description of each contribution listed above and highlight the key ideas. We discuss these techniques in greater detail in the subsequent chapters.

1.2.1 The RLIBM Approach for Correctly Rounded Polynomial Approximations

Our RLIBM *approach makes a case for generating polynomials that directly approximate the correctly rounded result of $f(x)$ itself.* Figure 1.1(a) illustrates our key intuition. A finite precision representation $\mathbb{T}$ can only represent finitely many values (*i.e.*, v_1, v_2, and

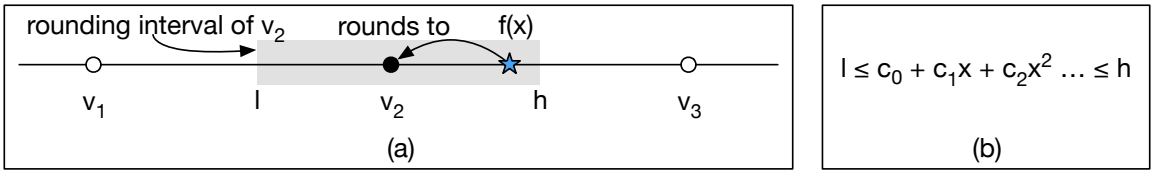


Figure 1.1: (a) The values v_1, v_2, and v_3 are representable values in a representation $\mathbb{T}$. The real value of $f(x)$ for a given input x cannot be exactly represented in $\mathbb{T}$ and is rounded to v_2. Our approach identifies the rounding interval of v_2 (shown in gray box). (b) Then, we generate a polynomial approximation using the rounding interval (*i.e.*, $[l, h]$) for each input x with an LP formulation.

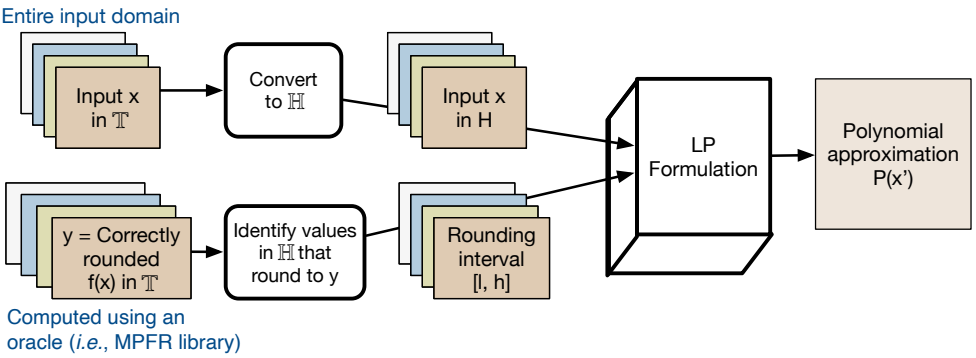


Figure 1.2: The RLIBM's approach to generate a polynomial approximation that produces the correctly rounded result of $f(x)$.

v_3). Given an input x, if the real value of $f(x)$ (highlighted with star) cannot be exactly represented in $\mathbb{T}$, then it is rounded to a value in $\mathbb{T}$ (*i.e.*, v_2). There is a range of values $[l, h]$ in the vicinity of v_2 (highlighted in gray box) where all real values in the interval rounds to v_2. We call this interval the rounding interval. If we generate a polynomial that produces a value in the rounding interval, then the result of the polynomial will always round to the correctly rounded result (*i.e.*, v_2).

The basic RLIBM approach consists of three steps. Figure 1.2 pictorially shows each step. As the goal of the RLIBM approach is to approximate the correctly rounded result of $f(x)$, the first step is to compute the correctly rounded result of $f(x)$ for each input x in $\mathbb{T}$. We use an oracle (an arbitrary precision math library, *i.e.*, MPFR math library) to compute the real value of $f(x)$ and round the value to $\mathbb{T}$. Then, we identify the rounding interval $[l, h]$ for each input, where any value in the rounding interval rounds to the correctly rounded

result of $f(x)$. Because the polynomial is evaluated in the higher precision representation $\mathbb{H}$, the rounding interval is also in $\mathbb{H}$.

Each input x and its corresponding rounding interval $[l, h]$ defines a constraint on the output of the polynomial. The result of the polynomial $P(x) = c_0 + c_1 x + c_2 x^2 + \ldots$ must be a value between l and h for the given input x to produce the correctly rounded result of $f(x)$:

$$l \leq c_0 + c_1 x + c_2 x^2 + \cdots \leq h$$

The variables l, h, and x are constants in this inequality formula. The unknown variables, $c_0, c_1, \ldots$, are the coefficients of the polynomial that we wish to generate. Using the inequality formula for each input x_i, we can create a system of linear inequalities that encodes the problem of identifying the coefficients of a d-degree polynomial that produces the correctly rounded results for all inputs:

$$\begin{bmatrix} l_1 \\ l_2 \\ \vdots \end{bmatrix} \leq \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^d \\ 1 & x_2 & x_2^2 & \ldots & x_2^d \\ \vdots & \vdots & & \ddots & \vdots \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_d \end{bmatrix} \leq \begin{bmatrix} h_1 \\ h_2 \\ \vdots \end{bmatrix}$$

This system of linear inequalities is precisely a linear programming (LP) query without an objective function. Hence, we use an LP solver to solve this formulation. The polynomial constructed using the solution of the LP query produces the correctly rounded result of $f(x)$ for all inputs.

The rounding interval $[l, h]$ for each input x that we identify is larger than $[f(x) - \epsilon, f(x) + \epsilon]$ where ϵ is the error bound of the correctly rounded polynomial approximation generated using prior approaches. Hence, RLIBM provides larger freedom to generate polynomials that produce correctly rounded results. This freedom results in better performance.

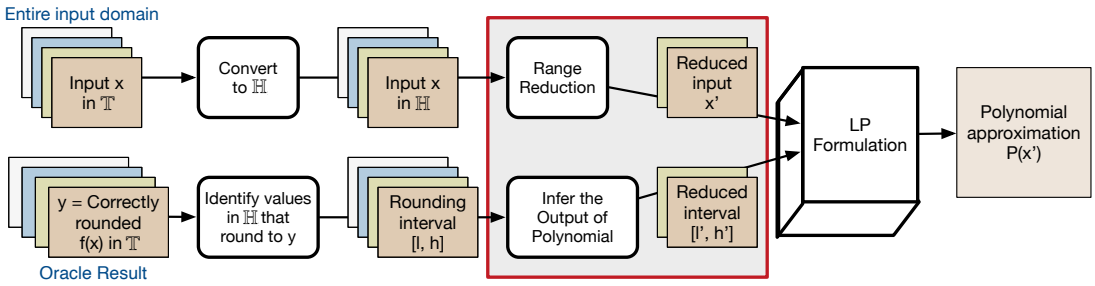


Figure 1.3: The RLIBM's approach to generate a polynomial approximation that produces the correctly rounded result of $f(x)$ when used with a range reduction strategy. We highlight the extension from Figure 1.2 with the gray box.

1.2.2 The RLIBM Approach With Range Reduction

In general, generating polynomials for a small input domain is easier than the entire input domain. Hence, elementary functions are often approximated with a combination of range reduction and polynomial approximation. The original input x is reduced to a smaller domain with the range reduction function. Subsequently, the polynomial approximation function is used with the reduced input. The resulting value is adjusted with the output compensation function to produce the final output. The entire approximation function $A(x)$ of $f(x)$ is a composition of these three functions. For example, the input domain for $log_2(x)$ is $(0, \infty)$. Polynomials can approximate $log_2(x)$ for the smaller domain $[1, 2)$ much more accurately than the entire input domain. Hence, we reduce the original input x into the reduced input x' using $x = x' \times 2^k$ where $x' \in [1, 2)$ and k is an integer. We approximate $y' = log_2(x')$ using a polynomial for the reduced domain $[1, 2)$. Then, $log_2(x)$ for the original input x is approximated by compensating the output y' with $y = y' + e$.

Range reduction, polynomial approximation, and output compensation functions are evaluated in a finite precision representation. Hence, they can experience numerical errors. To use polynomials with range reduction, the RLIBM approach adjusts the output intervals for the polynomials to account for the numerical errors. Figure 1.3 pictorially shows the modifications to handle range reductions. The additional steps that adjust the intervals are highlighted with the gray box. In this setting, each original input and rounding interval con-

straint defines the output of the entire approximation of $f(x)$ (*i.e.*, $A(x)$) should produce. We use the original inputs and the range reduction function to identify the list of inputs for the polynomial. We call these inputs the reduced inputs. Similarly, we use the rounding intervals and the output compensation function to infer the range of values that the polynomial should produce. We call this range of values the reduced interval. If a polynomial approximation produces a value in the reduced interval for each reduced input, then the result when used with the output compensation function produces a value in the rounding interval. With the reduced inputs and reduced intervals, we generate polynomial approximations using LP formulations. Further, we develop modified range reduction techniques for some elementary functions to minimize cancellation errors in output compensation (see Chapter 4 for more detail).

1.2.3 Scaling RLIBM To 32-bit Representations

The 32-bit representations have four billion inputs. Naively using the RLIBM approach for 32-bit types can generate millions of constraints, which is beyond the capabilities of LP solvers. Additionally, it may not be feasible to generate a single polynomial of a reasonable degree that satisfies all constraints. It will most likely require a high-degree polynomial to produce the correctly rounded result, which is not ideal for performance. Hence, we extend the RLIBM approach with two techniques to generate efficient polynomial approximations for 32-bit types: We propose to (1) use the counterexample guided polynomial generation technique that samples inputs to handle a large number of constraints and (2) generate piecewise polynomials for efficient implementations of elementary functions. Figure 1.4 illustrates the steps to generate polynomial approximations for 32-bit types.

Counterexample guided polynomial generation. To generate polynomial approximations that produce the correctly rounded results for millions of inputs, it is not necessary to consider every single input and its corresponding interval. We only need to reason about inputs

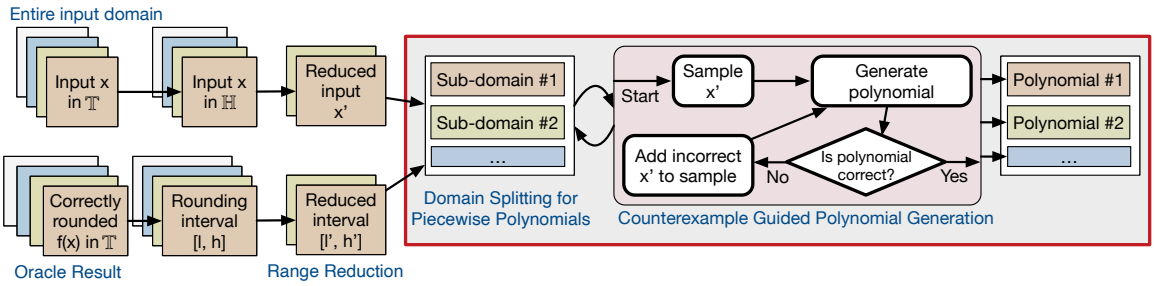


Figure 1.4: The RLIBM's approach for generating a polynomial approximation that produces the correctly rounded result of $f(x)$ for 32-bit types.

with highly constrained intervals to ensure that we generate polynomials that satisfy these constraints. The resulting polynomial will likely satisfy other constraints. Hence, we sample a few inputs and compute the rounding interval for the sampled inputs. If we intend to use range reduction, we also identify the reduced inputs and intervals. We generate a candidate polynomial that produces a value in the interval for each sampled input using LP formulation. Next, we check if the candidate polynomial produces a value in the rounding interval (or reduced interval) for all inputs (or reduced inputs). We add any counterexample inputs to the sample and repeat the process until the generated polynomial produces the correctly rounded results for all inputs. This counterexample guided polynomial generation process is inspired by counterexample guided inductive synthesis (CEGIS) [72, 123] used in program synthesis.

Piecewise polynomials. When the number of inputs in the sample exceeds the capability of our LP solver, the LP solver cannot generate a polynomial, or the generated polynomial does not satisfy our performance constraint, we split the input domain $[a, b]$ into sub-domains $[a, b')$ and $[b', b]$. Then, we generate a polynomial for each sub-domain using counterexample guided polynomial generation. The domain is split in such a way that we can use a few bits in the bit-pattern of the input to identify which polynomial to evaluate. If the RLIBM approach cannot generate a polynomial for every sub-domain, then the input domain is further split into smaller sub-domains. The final number of sub-domains depends

on the chosen elementary function and representation. Among the elementary functions we tested for the 32-bit float representation, the RLIBM approach splits the original input domain into 2^6 sub-domains on average. This strategy allows the RLIBM approach to generate piecewise polynomials with a low degree resulting in implementations that are faster than mainstream math libraries.

1.2.4 A Single Approximation for Multiple Representations and Rounding Modes

The FP representation officially has five rounding modes defined by the IEEE-754 standard. The correctly rounded result of $f(x)$ for a given input x may be different depending on the chosen rounding mode. One possible solution to produce the correctly rounded result for each rounding mode is to adopt the strategy from CR-LIBM [29] or the above RLIBM approach. They generate a correctly rounded polynomial approximation of $f(x)$ for each rounding mode. While this approach may be feasible for a single type, it can quickly become overwhelming when we generate correctly rounded approximations for various representations. It would be ideal if we can generate a single polynomial approximation that produces the correctly rounded results for multiple rounding modes and precision configurations.

We propose a novel approach to create such polynomial approximations. The key idea is to create polynomials that approximate the correctly rounded result of $f(x)$ with the *round-to-odd* (rno) mode. The rno mode is a non-standard rounding mode that has been used to avoid double rounding issues when performing primitive operations with extended precision and subsequently rounding the result to a lower precision representation [9]. The rno mode can be described as follows: If the real value of $f(x)$ is exactly representable in our target representation, then it is rounded to that value. Otherwise, it is rounded to a value whose bit-string is odd when interpreted as an unsigned integer. Our contribution is in recognizing that rno mode can be used to generate correctly rounded polynomial approximations for multiple rounding modes. Chapter 6 provides more detail regarding the

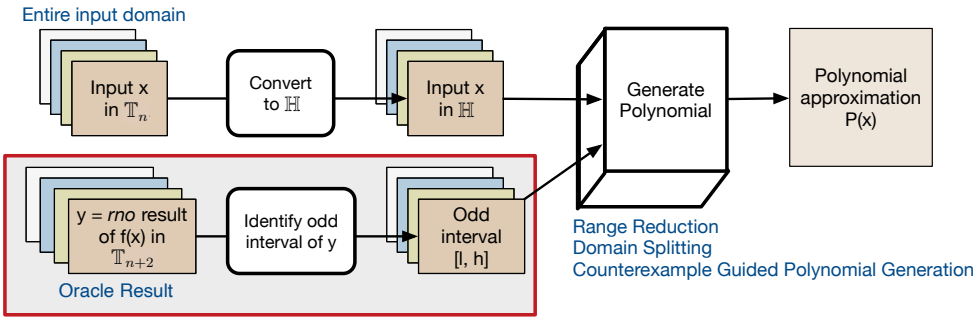


Figure 1.5: The RLIBM's approach to generate polynomial approximations that produces the correctly rounded result for $\mathbb{T}_{n+2}$ with rno mode.

rno mode.

Our goal is to generate correctly rounded polynomial approximation of $f(x)$ for all k bit representations $\mathbb{T}_k$ with any rounding mode where k is bounded (*i.e.*, $k \leq n$). The largest $\mathbb{T}_k$ representation is $\mathbb{T}_n$. We propose to generate a polynomial that produces the correctly rounded result of $f(x)$ in the $(n+2)$-bit representation $\mathbb{T}_{n+2}$ with the rno mode. We prove that this polynomial produces correct results for $\mathbb{T}_k$ for any standard rounding modes as long as $\mathbb{T}_k$ and $\mathbb{T}_{n+2}$ have the same number of exponent bits. We present the proof in Chapter 6.

Figure 1.5 presents the steps to generate a polynomial approximation that produces the rno result in $\mathbb{T}_{n+2}$. For each input in $\mathbb{T}_n$, we compute the correctly rounded result of $f(x)$ in $\mathbb{T}_{n+2}$ with the rno mode. Next, we identify a range of values that rounds to the rno result. We call this range of values the odd interval. Given a list of inputs and odd intervals, we use the RLIBM approach to generate a polynomial approximation that produces the correctly rounded result of $f(x)$ in $\mathbb{T}_{n+2}$ with the rno mode. To generate efficient approximations for large representations, we combine the rno approach with range reduction and generate piecewise polynomials using counterexample guided polynomial generation.

1.2.5 Prototype

Using the RLIBM approach, we generated multiple prototypes of correctly rounded elementary functions targeting various representations and rounding modes. RLIBM-16 con-

tains ten elementary functions for the 16-bit bfloat16 and posit16 types that produce correctly rounded results with the default, *round-to-the-nearest-tie-goes-to-even* (rne), rounding mode. RLIBM-32 contains ten correctly rounded elementary functions for the 32-bit float and posit32 types with rne mode. Finally, RLIBM-ALL contains ten functions that produce the correctly rounded rno results for the 34-bit FP representation. These functions produce correctly rounded results for all FP representations ranging from 10-bits to 32-bits including bfloat16, TensorFloat32, and float types. RLIBM-ALL also contains ten functions for posit representations. These functions produce correctly rounded results for 10-bit to 32-bit posit types. The functions in RLIBM-ALL support all standard rounding modes. The resulting elementary functions are faster than mainstream math libraries while producing correctly rounded results for all inputs in the target representations and rounding modes. In particular, the floating point functions in RLIBM-ALL have $1.1\times$, $1.3\times$, and $1.9\times$ speedup over glibc, Intel, and CR-LIBM math libraries, respectively.

1.3 Contributions to This Dissertation

The ideas and approaches in this dissertation are drawn from previously published papers written with my advisor Santosh Nagarakatte and collaborators Mridul Aanjaneya and John Gustafson. They are:

1. "An Approach to Generate Correctly Rounded Math Libraries for New Floating Point Variants," [90] and its corresponding technical report [89] which introduces the initial RLIBM approach to generate polynomials that approximate the correctly rounded result of $f(x)$ and how to handle numerical errors in range reduction.

2. "High Performance Correctly Rounded Math Libraries for 32-bit Floating Point Representations," [93] and its corresponding technical report [96] which scales the RLIBM approach for 32-bit types with domain splitting and counterexample guided polynomial generation.

3. "A Novel Polynomial Approximation Method to Produce Correctly Rounded Results for Multiple Representations and Rounding Modes," [95] which shows the RLIBM approach to generate polynomials that produces correctly rounded results for multiple representations and rounding modes.

1.4 Organization of This Dissertation

Chapter 2 presents details on the FP representation and the posit representation, the two target representations of the elementary functions that we generate. Additionally, it provides background on prior approaches in approximating elementary functions and challenges in producing correctly rounded results. Chapter 3 presents the RLIBM approach to generate correctly rounded polynomial approximations of $f(x)$ using LP formulation. Chapter 4 extends RLIBM to handle numerical errors in range reduction and output compensation. Chapter 5 scales the RLIBM approach to generate piecewise polynomials for 32-bit types. Chapter 6 presents our approach to generate a single polynomial approximation that produces correctly rounded results for multiple precision configurations and rounding modes using the rno mode. Chapter 7 evaluates the correctness and performance of the elementary functions generated with RLIBM. Chapter 8 discusses the prior related work in creating efficient and accurate approximations of elementary functions. Chapter 9 concludes by presenting future directions.

CHAPTER 2
BACKGROUND

In this chapter, we provide background on the floating point (FP) representation [27] and the posit [55, 58] representation. The FP and posit representations are used to approximate real numbers. We focus on these two representations because FP is the most widely used representation. While posit was proposed in 2014, there is already excitement and interest in exploring the posit representation in different domains [18, 45, 77, 102]. We describe how to interpret an FP or posit bit-string to a real number, round an arbitrary real number to a chosen representation, and the numerical errors that occur when using these representations. Next, we describe the state-of-the-art methodology in approximating elementary functions, performing range reduction, and generating the polynomial approximation. Finally, we conclude by discussing challenges in generating correctly rounded approximations.

2.1 The Floating Point Representation

There are two important attributes in any representation that approximates real numbers: dynamic range and precision. The dynamic range determines the range of values that a representation can express, both values close to zero and values with large magnitude. The precision of a representation measures how accurately a real value can be represented. Given a limited number of bits n to represent real values, number system designers must make a trade-off between the dynamic range and the precision.

The floating point (FP) representation, specified in the IEEE-754 standard [27], uses a fixed number of bits to determine the dynamic and the precision of the representation. They are identified using two parameters, the total number of bits in the representation n and the number of exponent bits $|E|$. We denote a particular FP configuration with $F_{n,|E|}$.

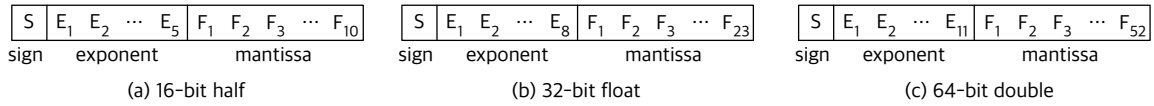


Figure 2.1: Bit-string for the IEEE-754 FP representations. (a) 16-bit half. (b) 32-bit float. (c) 64-bit double.

There are three components in an FP bit-string: a sign bit S to represent whether an FP value is positive or negative, $|E|$-bits to represent the exponent, and $|F| = n - 1 - |E|$ bits to represent the fractional value, known as the mantissa. The exponent bits determine the dynamic range while the mantissa bits determine the precision of the FP representation. Because FP has a fixed amount of precision (*i.e.* mantissa bits) for a given representation, FP is known as a fixed-precision representation. There are three default FP configurations specified by IEEE-754: 16-bit half ($\mathbb{F}_{16,5}$), 32-bit float ($\mathbb{F}_{32,8}$), and a 64-bit double ($\mathbb{F}_{64,11}$). Figure 2.1 shows the bit-string representations for them.

2.1.1 Interpreting a Floating Point Bit-String

There are three categories of floating point values depending on the value of the exponent bits when interpreted as an unsigned integer: normal values, denormal values, and special case values. In all three categories, if the sign bit is 0 then the value is positive. If the sign bit is 1, then the value is negative.

Normal values. The value represented by an FP bit-string is a normal value if the exponent bits E are neither all ones nor all zeros, (*i.e.*, $0 < x < 2^{|E|} - 1$). In such a case, the value represented by the bit-string is equal to,

$$(-1)^S \times (1 + \frac{F}{2^{|F|}}) \times 2^{E-bias}$$

where $bias = 2^{|E|-1} - 1$. E and F represent the exponent and mantissa bits interpreted as an unsigned integer. The bit-string of normal values x in FP encodes the value represented in a binary scientific notation $x = (-1)^s \times m \times 2^e$ where e is the exponent value and

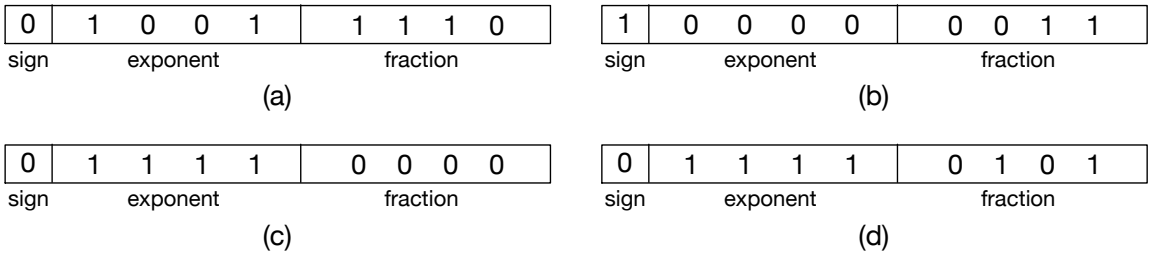


Figure 2.2: Examples of bit-patterns representing a (a) normal value, (b) denormal value, (c) infinity, and (d) NaN in the 9-bit FP representation with 4 exponent bits (*i.e.*, $\mathbb{F}_{9,4}$).

$m \in [1, 2)$ is known as the significand. The mantissa bits F encodes the fractional part of the significand without the leading one to the left of the radix point (*i.e.*, $1.f_1f_2f_3\ldots$). Additionally, the exponent bits E encode exponent value e as an unsigned integer. To represent both negative and positive exponent, E is scaled with $-bias$. By encoding the exponent as an unsigned integer that is scaled with $bias$, comparison of two FP values with the same sign can be efficiently performed by comparing the bit-string of the values using unsigned integer comparisons.

Example. Figure 2.2(a) presents the bit-string of a normal value in a 9-bit FP representation with four exponent bits (*i.e.*, $\mathbb{F}_{9,4}$). In $\mathbb{F}_{9,4}$, $bias = 2^{4-1} - 1 = 7$. This bit-string represents the value,

$$(-1)^0 \times (1 + \frac{14}{2^4}) \times 2^{9-7} = (1 + \frac{14}{16}) \times 2^2 = 1.875 \times 2^2 = 7.5$$

Denormal values. If all exponent bits in E are zeros, then the FP value is denormal. In such a case, the value represented by the bit-string is equal to,

$$(-1)^S \times \frac{F}{2^{|F|}} \times 2^{1-bias}$$

Denormal values are used to represent values close to zero. There is a gap between the smallest representable normal value 2^{1-bias} and 0. By using denormal values, FP representations can represent values significantly smaller in magnitude compared to 2^{1-bias}. The

smallest denormal value representable in FP representation is $2^{1-bias-|F|}$.

Example. Figure 2.2(b) presents the bit-string of a denormal value in $\mathbb{F}_{9,4}$. Since the exponent bits are all zeros, it represents a denormalized value. The value encoded by the bit-string in a binary scientific notation is,

$$(-1)^1 \times \frac{3}{2^4} \times 2^{1-7} = -1 \times \frac{3}{16} \times 2^{-6} = -0.1875 \times 2^{-6} = -1.5 \times 2^{-9}$$

This value is significantly closer to 0 compared to the smallest normal value in $\mathbb{F}_{9,4}$, which is $2^{1-bias} = 2^{-6}$.

Special values. Finally, if all exponent bits are ones, then the FP value is a special value. If all mantissa bits are zeros, then it represents $\pm\infty$ depending on the sign bit S. Figure 2.2(c) shows the bit-string of ∞ in $\mathbb{F}_{9,4}$. In all other cases, it represents *not-a-number* (NaN). NaNs are used to represent exceptional conditions such as the result of $\frac{0.0}{0.0}$ or $\infty - \infty$. Figure 2.2(d) shows a NaN value.

Due to the way FP representations are designed, an n-bit FP does not encode 2^n unique values. First, two bit-patterns represent 0: One where all bits are zero and another with a 1 bit followed by $n-1$ zeros. While these two values are sometimes referred to as $+0$ and -0, they are semantically equivalent. Second, there are $2^{|F|+1} - 2$ bit-patterns that represent NaN in a given $\mathbb{F}_{n,|E|}$ configuration. In the 32-bit float, roughly 16 million bit-patterns represent NaN.

2.1.2 Rounding a Real Number to the FP Representation

The total number of bits in an FP representation is finite. Many real values cannot be exactly represented in a FP representation. According to the IEEE-54 standard, if a real value $v_{\mathbb{R}}$ cannot be exactly represented, then it is rounded to either the largest FP value

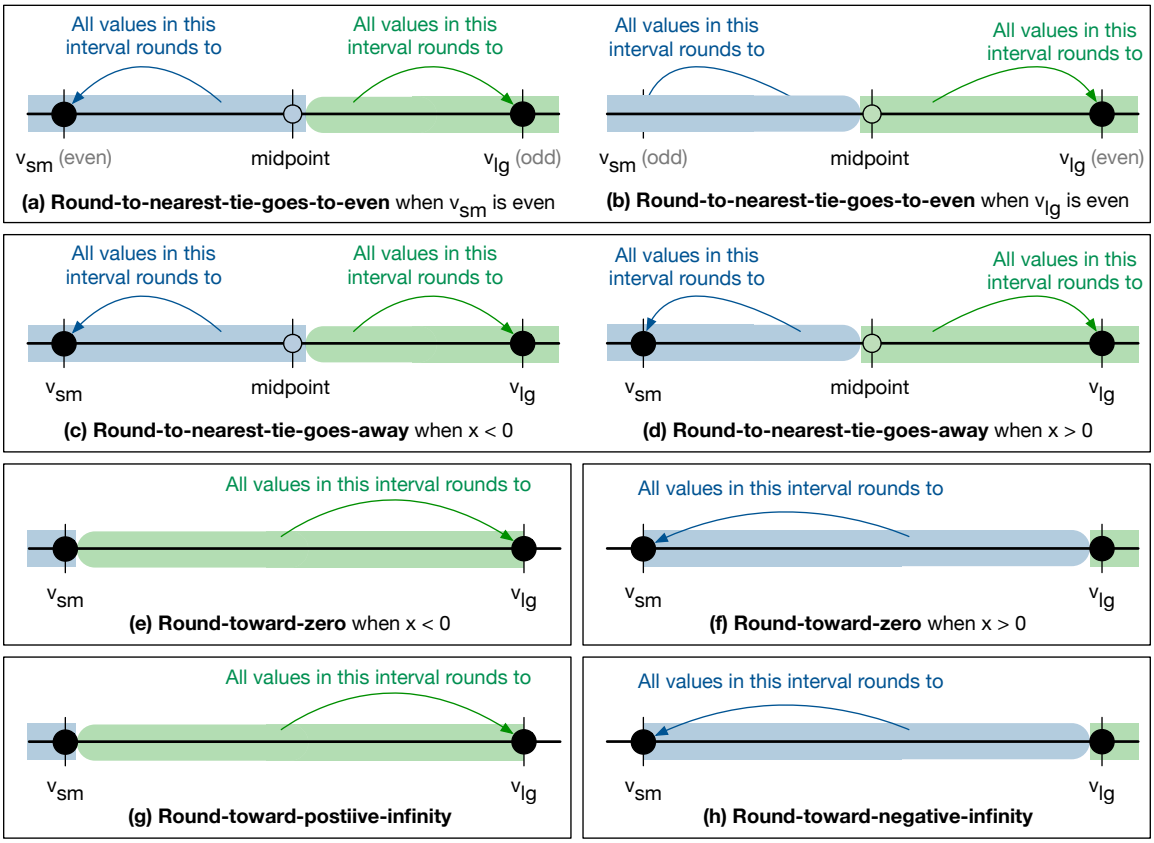


Figure 2.3: When a real value $v_{\mathbb{R}}$ is not exactly representable in $\mathbb{T}$, then $v_{\mathbb{R}}$ is rounded to one of the two adjacent values $v_{sm}, v_{lg} \in \mathbb{T}$ depending on the rounding mode used. We show the range of $v_{\mathbb{R}}$ that rounds to v_{sm} (blue box) and v_{lg} (green box) if we use (a) rne mode when the bit-string of v_{sm} is even, (b) rne mode when the bit-string of v_{lg} is even, (c) rna mode when $x < 0$, (d) rna mode when $x > 0$, (e) rnz mode when $x < 0$, (f) rnz mode when $x > 0$, (g) rnp mode, and (h) rnn mode.

smaller than $v_{\mathbb{R}}$ (v_{sm}) or the smallest FP value larger than $v_{\mathbb{R}}$ (v_{lg}).

$$v_{sm} = max\{v \in \mathbb{F} \mid v \leq v_{\mathbb{R}}\}$$

$$v_{lg} = min\{v \in \mathbb{F} \mid v \geq v_{\mathbb{R}}\}$$

Note that if $v_{\mathbb{R}}$ is exactly representable in $\mathbb{F}$, then $v_{\mathbb{R}} = v_{sm} = v_{lg}$. Otherwise, $v_{\mathbb{R}}$ rounds to v_{sm} or v_{lg} depending on the choice of the rounding mode, rm. We denote the operation of rounding $v_{\mathbb{R}}$ to an FP representation $\mathbb{F}$ using a rounding mode rm with the function $RN_{\mathbb{F},rm}(v_{\mathbb{R}})$.

Standard Rounding Modes for FP Representation The IEEE-754 standard specifies five different rounding modes: round to the nearest, tie goes to even (rne), round to the nearest, tie goes away (rna), round towards zero (rnz), round towards positive infinity (rnp), and round towards negative infinity (rnn). The standard mandates the result of the basic arithmetic operations ($+$, $-$, $\times$, $\div$) and square root operation ($\sqrt{x}$) to be correctly rounded.

The rne ***mode:*** The round to the nearest, tie goes to even rounding mode rounds $v_{\mathbb{R}}$ to either v_{sm} or v_{lg}, depending on which value is closer to $v_{\mathbb{R}}$. If $|v_{sm} - v_{\mathbb{R}}| < |v_{lg} - v_{\mathbb{R}}|$, then $v_{\mathbb{R}}$ rounds to v_{sm}. If $|v_{sm} - v_{\mathbb{R}}| > |v_{lg} - v_{\mathbb{R}}|$, then $v_{\mathbb{R}}$ rounds to v_{lg}. In the case that $v_{\mathbb{R}}$ is exactly in the middle of v_{sm} and v_{lg} (*i.e.*, $v_{\mathbb{R}} = \frac{v_{sm}+v_{lg}}{2}$), $v_{\mathbb{R}}$ is rounded to a value where the bit-string representation is even when interpreted as an unsigned integer. The rne rounding mode is the default and the most commonly used rounding mode. Figure 2.3(a) shows the range of real values between v_{sm} and v_{lg} that rounds to v_{sm} (blue region) and v_{lg} (green region) if the bit-string of v_{sm} is even when interpreted as an unsigned integer. Similarly, Figure 2.3(b) shows the range of real values that rounds to v_{sm} and v_{lg} if the bit-string of v_{lg} is even when interpreted as an unsigned integer.

The rna ***mode:*** Similar to the rne mode, the round to the nearest, tie goes away rounding mode rounds $v_{\mathbb{R}}$ to the closest FP value (*i.e.*, either v_{sm} or v_{lg}). If $|v_{sm} - v_{\mathbb{R}}| < |v_{lg} - v_{\mathbb{R}}|$, then $v_{\mathbb{R}}$ rounds to v_{sm}. If $|v_{sm} - v_{\mathbb{R}}| > |v_{lg} - v_{\mathbb{R}}|$, then $v_{\mathbb{R}}$ rounds to v_{lg}. In the case that $v_{\mathbb{R}}$ is exactly in the middle of v_{sm} and v_{lg} ($v_{\mathbb{R}} = \frac{v_{sm}+v_{lg}}{2}$), $v_{\mathbb{R}}$ is rounded to a value that is farther away from 0. Specifically, $v_{\mathbb{R}}$ rounds to v_{lg} if $v_{\mathbb{R}} > 0$ because

$$0 \leq v_{sm} < v_{\mathbb{R}} < v_{lg} \leq \infty$$

Similarly, $v_{\mathbb{R}}$ rounds to v_{sm} if $v_{\mathbb{R}} < 0$ because

$$-\infty \leq v_{sm} < v_{\mathbb{R}} < v_{lg} \leq 0$$

The rna rounding mode is also known as the human rounding since it resembles how we round decimal numbers, *i.e.* 1.5 rounds up to 2 when rounded to the nearest integer while

1.4 rounds to 1. Figure 2.3(c) and (d) illustrates rounding with rna mode depending on whether $v_{\mathbb{R}} < 0$ or $v_{\mathbb{R}} > 0$.

The rnz ***mode:*** In the round to zero mode, $v_{\mathbb{R}}$ is rounded to the value that is closer to 0. $v_{\mathbb{R}}$ is rounded to v_{sm} if $x > 0$ and $v_{\mathbb{R}}$ is rounded to v_{lg} if $x < 0$. The rnz mode is equivalent to truncating the significand bits of $v_{\mathbb{R}}$ that cannot fit into the mantissa bits, similar to how real numbers are rounded to a machine integer. Figure 2.3(e) and (f) shows the range of real values between v_{sm} and v_{lg} that rounds to v_{sm} or v_{lg} when $v_{\mathbb{R}} < 0$ and $v_{\mathbb{R}} > 0$, respectively.

The rnp ***mode:*** The round towards positive infinity mode always rounds $v_{\mathbb{R}}$ to the larger value v_{lg}, the value that is closer to $+\infty$. The rnp mode is also known as rounding up. Figure 2.3(g) shows the range of $v_{\mathbb{R}}$'s that rounds to v_{sm} or v_{lg} using rnp mode.

The rnn ***mode:*** The round towards negative infinity mode always rounds $v_{\mathbb{R}}$ to the smaller value v_{sm}, the value that is closer to $-\infty$. The rnn mode is also known as rounding down. Figure 2.3(h) shows the range of $v_{\mathbb{R}}$'s that rounds to v_{sm} or v_{lg} using rnn mode.

2.1.3 A Systematic Methodology for Rounding To the FP Representation

We describe a methodology to systematically round a real number $v_{\mathbb{R}}$. Special values $\pm\infty$ or NaN can be directly translated into $\mathbb{F}$. As described above, we need to identify the two values v_{sm} and v_{lg} in $\mathbb{F}_{n,|E|}$ representations that are adjacent to $v_{\mathbb{R}}$ and decide which value $v_{\mathbb{R}}$ rounds to. We identify the four pieces of information (s, v^{-}, rb, $sticky$) necessary to correctly decide whether $v_{\mathbb{R}}$ rounds to v_{sm} or v_{lg}. We call these pieces of information the rounding components. The first component s represents the sign (1 or -1) that identifies whether $v_{\mathbb{R}}$ is positive or negative. The value v^{-}, which we call the truncated value, represents the smaller magnitude value between v_{sm} or v_{lg}. The last two components rb and $sticky$ determines whether $v_{\mathbb{R}}$ is equal to v_{sm}, closer to v_{sm}, in the middle of v_{sm} and v_{lg}, closer to v_{lg}, or equal to v_{lg}.

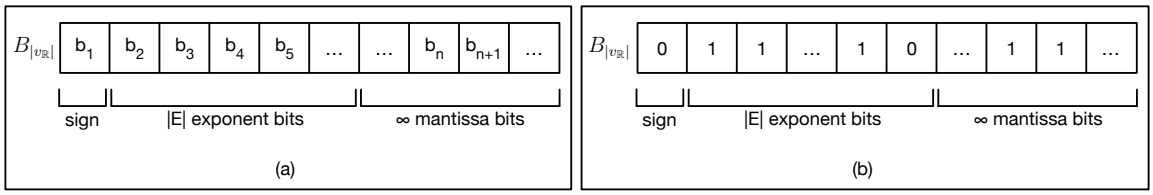


Figure 2.4: (a) Bit-string of $|v_\mathbb{R}|$ when represented in the infinite extended precision $\mathbb{F}_\infty$, where there are infinite number of bits for mantissa. (b) Bit-string of $|v_\mathbb{R}|$ represented in $\mathbb{F}_\infty$ when $v_\mathbb{R}$ is outside the dynamic range of $\mathbb{F}_{n,|E|}$.

To identify the rounding components, we first decompose $v_\mathbb{R}$ into $v_\mathbb{R} = s \times |v_\mathbb{R}|$ where s is the first rounding component that represents the sign of $v_\mathbb{R}$. The value $|v_\mathbb{R}|$ represents the magnitude of $v_\mathbb{R}$. Next, we represent $|v_\mathbb{R}|$ in the FP representation with an infinite number of mantissa bits while having the same number of exponents. We call this representation the infinite extended precision representation $\mathbb{F}_\infty$. Intuitively, $\mathbb{F}_\infty$ is similar to $\mathbb{F}_{n,|E|}$ but has infinite amount of precision to exactly represent $|v_\mathbb{R}|$. The bit-string of $|v_\mathbb{R}|$ in $\mathbb{F}_\infty$ is pictorially illustrated in Figure 2.4(a). The first bit b_1 represents the sign bit. Since $|v_\mathbb{R}| \geq 0$, $b_1 = 0$. The bits b_2 to $b_{|E|+1}$ represent the exponent bits. The remaining bits are mantissa bits. If the value of $|v_\mathbb{R}|$ is larger than the dynamic range, then we represent it with the largest representable value in $\mathbb{F}_\infty$, where the exponent bits are equivalent to the largest normal value in $\mathbb{F}_{n,|E|}$ and all mantissa bits are ones. The bit-string is illustrated in Figure 2.4(b). Note that we cannot use ∞ to represent $|v_\mathbb{R}|$ because we need to make a clear distinction between a real number and ∞. The rnz mode never rounds a real value to ∞, rnp mode does not round negative real values to $-\infty$, and rnn mode does not round positive real values to ∞.

Identifying the truncated value. To identify v_{sm} and v_{lg}, we identify two values v^- and v^+ adjacent to $|v_\mathbb{R}|$ in $\mathbb{F}$. Intuitively, v^- represents the largest value that is smaller than or equal to $|v_\mathbb{R}|$ and v^+ represents the smallest value larger than $|v_\mathbb{R}|$. To identify v^-, we truncate $B_{|v_\mathbb{R}|}$ to n bits,

$$B_{v^-} = 0 b_2 b_3 b_4 \dots b_{n-1} b_n$$

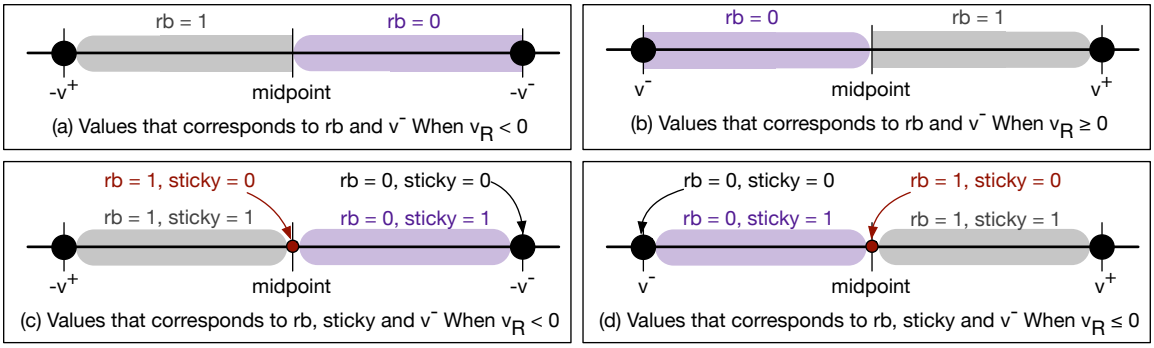


Figure 2.5: We show the range of real values that corresponds to the different values in the rounding bit rb when (a) $v_{\mathbb{R}} < 0$ and (b) $v_{\mathbb{R}} \geq 0$. Additionally, we show the range of values that corresponds to the different values of rb and sticky bit $sticky$ when (c) $v_{\mathbb{R}} < 0$ and (d) $v_{\mathbb{R}} \geq 0$.

The value encoded by B_{v^-} in $\mathbb{F}$ is v^-, the largest value smaller than or equal to $|v_{\mathbb{R}}|$. We call v^- the truncated value, which is the second rounding component. Then, the succeeding value of v^- in $\mathbb{F}_{n,|E|}$ is v^+, which can be obtained by adding 1 to the bit-string of v^-. In the context of rounding $v_{\mathbb{R}}$ to $\mathbb{F}_{n,|E|}$, v^- and v^+ have the following relationship,

$$\begin{cases} -v^+ < v_{\mathbb{R}} \leq -v^- & \text{if } v_{\mathbb{R}} < 0 \ (s = -1) \\ v^- \leq v_{\mathbb{R}} < v^+ & \text{if } v_{\mathbb{R}} \geq 0 \ (s = 1) \end{cases}$$

We can compute the two possible values that $v_{\mathbb{R}}$ rounds to, v_{sm} and v_{lg}, using s, v^-, and v^+. If $v_{\mathbb{R}}$ is exactly representable in $\mathbb{F}_{n,|E|}$, then $|v_{\mathbb{R}}| = v^-$. Thus, $v_{sm} = v_{lg} = s \times v^-$. If $v_{\mathbb{R}}$ is not exactly representable in $\mathbb{F}$, then $v^- < |v_{\mathbb{R}}| < v^+$. Hence, $v_{sm} = v^-$ and $v_{lg} = v^+$ if $s = 1$ (when $v_{\mathbb{R}} \geq 0$). Otherwise, $v_{\mathbb{R}}$ is negative and $v_{sm} = -v^+$ and $v_{lg} = -v^-$.

Identifying the rounding bit. The two components s and v^- alone only informs that $v^- \leq v_{\mathbb{R}} < v^+$ if $s = 1$ and $-v^+ < v_{\mathbb{R}} \leq -v^-$ if $s = -1$. For the rounding mode rnz, these two components are sufficient to identify the correctly rounded result, since $\pm v^-$ is always closer to zero compared to $\pm v^+$. However, to determine the rounding decision for rne and rna, we must find whether $v_{\mathbb{R}}$ is closer to $s \times v^-$, closer to $s \times v^+$, or exactly in the middle of the two values. To determine the rounding decision for rnp and rnn, we need to identify whether $v_{\mathbb{R}}$ is exactly representable with $s \times v^-$. Our next two steps are to identify the rounding bit and sticky bit that will help us identify the relationship between $v_{\mathbb{R}}$, v^-,

and v^+.

We identify the rounding bit (rb) by extracting the $(n+1)^{st}$ bit from $B_{|v_\mathbb{R}|}$. Intuitively, the rounding bit describes whether $|v_\mathbb{R}|$ is closer to v^- than v^+. If the rounding bit is 0, then $|v_\mathbb{R}|$ is closer to v^- (*i.e.*, $v^- \leq |v_\mathbb{R}| < \frac{v^-+v^+}{2}$). If the rounding bit is 1, then $|v_\mathbb{R}|$ is either in the middle or closer to v^- (*i.e.*, $\frac{v^-+v^+}{2} \leq |v_\mathbb{R}| < v^+$). Figure 2.5(a) and (b) shows the range of real values that corresponds to different values of v^- and rb.

Identifying the sticky bit. While the rounding bit tells us whether $|v_\mathbb{R}|$ is closer to v^-, it does not tell us whether $|v_\mathbb{R}|$ is exactly equal to v^- or is exactly in the middle of v^- and v^+ (*i.e.*, $|v_\mathbb{R}| = \frac{v^-+v^+}{2}$). When looking at the bit-string $B_{|v_\mathbb{R}|}$, $|v_\mathbb{R}|$ is equal to v^- when the $(n+1)^{st}$ bit (rb) is zero and the remaining bits from $(n+2)^{nd}$ bits are all zeros. If $rb = 0$ and any bit afterwards is a one, then $|v_\mathbb{R}|$ is not equal to v^-. Likewise, $|v_\mathbb{R}|$ is exactly in the middle of v^- and v^+ when the $(n+1)^{st}$ bit is a one and the remaining bits from $(n+2)^{nd}$ bits are all zeros in $B_{|v_\mathbb{R}|}$. If $rb = 1$ and any bit afterwards is a one, then $|v_\mathbb{R}|$ is closer to v^+. In both cases, we need to determine whether all bits in $B_{|v_\mathbb{R}|}$ starting from the $(n+2)^{nd}$ bits are zeros or not. We define the sticky bit as the result of bitwise `or` operation on all bits in $B_{|v_\mathbb{R}|}$ starting from the $(n+2)^{nd}$ bit:

$$sticky = b_{n+2} \mid b_{n+3} \mid b_{n+3} \mid \ \ldots$$

where $\mid$ is the bitwise `or` operation.

Using the four rounding components (s, v^-, rb, and $sticky$), we can now determine the relationship between $|v_\mathbb{R}|$ and the two nearest FP values v^- and v^+ where v^+ can be computed from v^-:

$$\begin{cases} |v_\mathbb{R}| = v^- & \text{if } rb = 0 \wedge sticky = 0 \\ v^- < |v_\mathbb{R}| < \frac{v^-+v^+}{2} & \text{if } rb = 0 \wedge sticky = 1 \\ |v_\mathbb{R}| = \frac{v^-+v^+}{2} & \text{if } rb = 1 \wedge sticky = 0 \\ \frac{v^-+v^+}{2} < |v_\mathbb{R}| < v^+ & \text{if } rb = 1 \wedge sticky = 1 \end{cases}$$

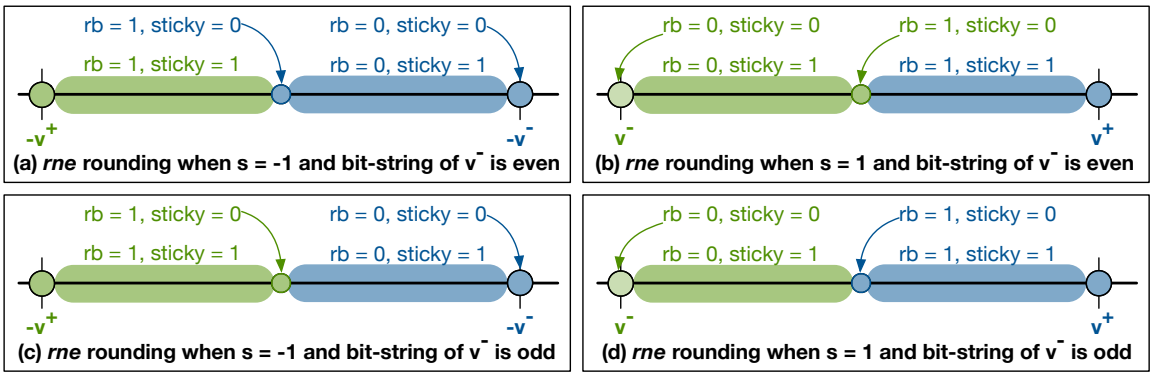


Figure 2.6: The rne mode using the rounding components. We illustrate the rounding decision when $v_{\mathbb{R}}$ is positive or negative and when v^- is even or odd. The green interval represents all real values that round to the FP value with green color (*i.e.*, $-v^+$ when $v_{\mathbb{R}} < 0$ and v^- when $v_{\mathbb{R}} \geq 0$). Similarly, the blue interval represents all real values that round to the FP value with blue color (*i.e.*, $-v^-$ when $v_{\mathbb{R}} < 0$ and v^+ when $v_{\mathbb{R}} \geq 0$) with blue.

Figure 2.5(e) and (f) pictorially show the range of values for $v_{\mathbb{R}}$ that corresponds to the different values of the rounding components.

Determining the correctly rounded result of $v_{\mathbb{R}}$**.** Finally, based on the four components we identified, we identify the correctly rounded value of $v_{\mathbb{R}}$. In the rne mode, $v_{\mathbb{R}}$ rounds to the closest value. If $rb = 0$, then it indicates that $|v_{\mathbb{R}}|$ is closer to v^-, which means $v_{\mathbb{R}}$ is closer to $s \times v^-$. Thus, $v_{\mathbb{R}}$ rounds to $s \times v^-$. If the rounding bit $rb = 1$ and the sticky bit $sticky = 1$, then it indicates that $|v_{\mathbb{R}}|$ is closer to v^+ and $v_{\mathbb{R}}$ rounds to $s \times v^+$. If $|v_{\mathbb{R}}|$ is exactly in the middle of v^- and v^+, (which also indicates that $v_{\mathbb{R}}$ is exactly in the middle of $s \times v^-$ and $s \times v^+$), then the rounding bit $rb = 1$ and the sticky bit $sticky = 0$. $v_{\mathbb{R}}$ rounds to the value where the bit-string is even when interpreted as an unsigned integer. Finally, if $lb = 1$, then the bit-string of v^- is odd and $v_{\mathbb{R}}$ rounds to $s \times v^+$. The rounding decision for the rne mode can be formalized as follows,

$$RN_{\mathbb{F},rne}(v_{\mathbb{R}}) = \begin{cases} s \times v^- & \text{if } rb = 0 \\ s \times v^- & \text{if } rb = 1 \wedge sticky = 0 \wedge IsEven(v^-) \\ s \times v^+ & \text{if } rb = 1 \wedge sticky = 0 \wedge \neg IsEven(v^-) \\ s \times v^+ & \text{if } rb = 1 \wedge sticky = 1 \end{cases}$$

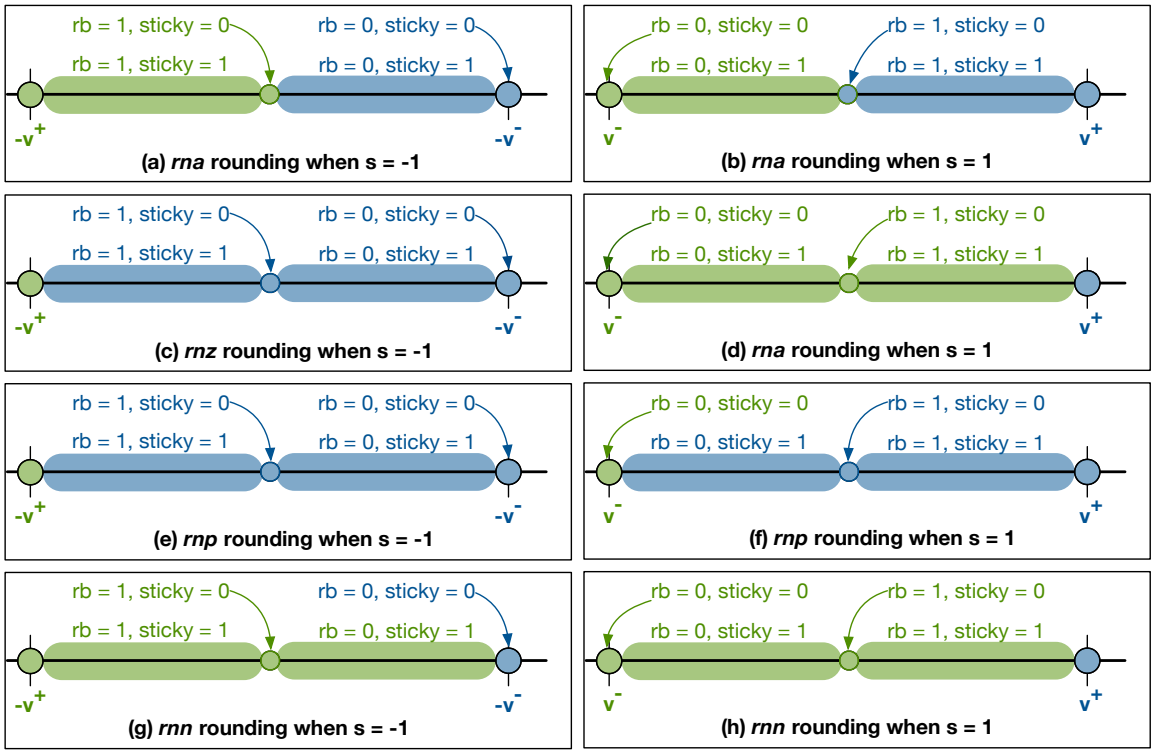


Figure 2.7: Rounding decisions for various rounding modes based on the rounding components. The interval of real values colored with green round to the FP value colored green. Similarly, the interval of real values colored with blue round to the FP value colored blue.

Figure 2.6 shows rounding decision of $v_{\mathbb{R}}$ in rne based on different values of rounding components. We highlight the range of values for $v_{\mathbb{R}}$ that round to the smaller value (*i.e.*, $-v^+$ when $s = -1$ and v^- when $s = 1$) with the green color and the range of values for $v_{\mathbb{R}}$ that round to the larger value (*i.e.*, $-v^-$ when $s = -1$ and v^+ when $s = 1$) with the blue color.

The rna mode is similar to rne mode. If the rounding bit $rb = 0$, then it indicates that $v_{\mathbb{R}}$ is closer to $s \times v^-$. If the rounding bit $rb = 1$ and the sticky bit $sticky = 1$, then it indicates that $v_{\mathbb{R}}$ is closer to $s \times v^+$. If $v_{\mathbb{R}}$ is exactly in the middle of $s \times v^-$ and $s \times v^+$, then the rounding bit $rb = 1$ and the sticky bit $sticky = 0$. We must round to the value that is farther away from 0, which is always $s \times v^+$ because $|v_{\mathbb{R}}| < v^+$. The rounding decision

for the rna mode can be summarized as follows,

$$RN_{\mathbb{F},rna}(v_{\mathbb{R}}) = \begin{cases} s \times v^{-} & \text{if } rb = 0 \\ s \times v^{+} & \text{if } rb = 1 \wedge sticky = 0 \\ s \times v^{+} & \text{if } rb = 1 \wedge sticky = 1 \end{cases}$$

Figure 2.7(a) and (b) shows the rounding decision of $v_{\mathbb{R}}$ in rna based on different values of rounding components.

In the rnz mode, $v_{\mathbb{R}}$ always rounds to the value closer to zero. Since v^{-} is always smaller than or equal to $|v_{\mathbb{R}}|$ (*i.e.*, $v^{-} \leq |v_{\mathbb{R}}|$), $v_{\mathbb{R}}$ always rounds to $s \times v^{-}$:

$$RN_{\mathbb{F},rnz}(v_{\mathbb{R}}) = s \times v^{-}$$

Figure 2.6(c) and (d) shows rounding decision of $v_{\mathbb{R}}$ in rnz based on different values of rounding components.

In the rnp rounding mode, $v_{\mathbb{R}}$ always rounds to the larger value. If $v_{\mathbb{R}}$ is exactly representable with $\mathbb{F}$, which is indicated by the rounding bit with $rb = 0$ and the sticky bit with $sticky = 0$, then $v_{\mathbb{R}} = s \times v^{-}$. If $v_{\mathbb{R}}$ is not exactly representable and is greater than or equal to zero (*i.e.*, $s = 1$) then $v_{\mathbb{R}}$ rounds to v^{+} because $v_{\mathbb{R}} < v^{+}$. Otherwise, $v_{\mathbb{R}}$ rounds to $-v^{-}$ since $v_{\mathbb{R}} < 0$ and $v_{\mathbb{R}} < -v^{-}$. Formally,

$$RN_{\mathbb{F},rnp}(v_{\mathbb{R}}) = \begin{cases} s \times v^{-} & \text{if } rb = 0 \wedge sticky = 0 \\ v^{+} & \text{if } \neg(rb = 0 \wedge sticky = 0) \wedge s = 1 \\ -v^{-} & \text{if } \neg(rb = 0 \wedge sticky = 0) \wedge s = -1 \end{cases}$$

Figure 2.6(e) and (f) shows rounding decision of $v_{\mathbb{R}}$ in rnp based on different values of rounding components.

Finally in the rnn rounding mode, $v_{\mathbb{R}}$ always rounds to the smaller value if it cannot be exactly represented. If $v_{\mathbb{R}}$ is exactly representable with $\mathbb{F}$ (*i.e.*, $rb = 0$ and $sticky = 0$), then $v_{\mathbb{R}}$ is equal to $s \times v^{-}$ (*i.e.*, $v_{\mathbb{R}} = s \times v^{-}$). If $v_{\mathbb{R}}$ is not exactly representable and is greater than or equal to zero, then $v_{\mathbb{R}}$ rounds to v^{-} because $v^{-} < v_{\mathbb{R}}$. Otherwise, $v_{\mathbb{R}}$ rounds

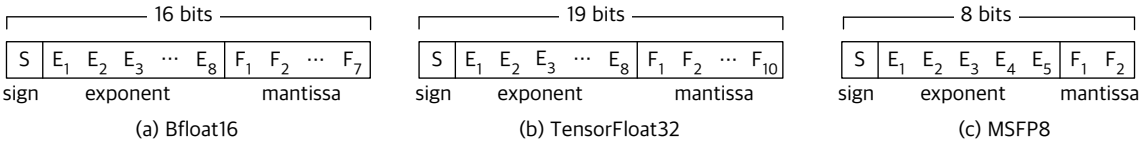


Figure 2.8: Layout of (a) bfloat16, (b) TensorFloat32, and (c) MSFP8.

to $-v^+$ since $v_{\mathbb{R}} < 0$ and in that case $-v^+ < v_{\mathbb{R}}$. The rounding decision for rnn mode can be formalized with,

$$RN_{\mathbb{F},rnn}(v_{\mathbb{R}}) = \begin{cases} s \times v^- & \text{if } rb = 0 \wedge sticky = 0 \\ v^- & \text{if } \neg(rb = 0 \wedge sticky = 0) \wedge s = 1 \\ -v^+ & \text{if } \neg(rb = 0 \wedge sticky = 0) \wedge s = -1 \end{cases}$$

Figure 2.6(g) and (h) shows rounding decision of $v_{\mathbb{R}}$ in rnn based on different values of rounding components.

2.1.4 Different FP Configurations and Fixed-Precision Variants

The IEEE-754 standard specifies a set of configurations for different lengths of n to be used in general including half ($\mathbb{F}_{16,5}$), float ($\mathbb{F}_{32,8}$), and double ($\mathbb{F}_{64,11}$) datatype. These formats have been adopted widely since the introduction of the standard and most modern commercial hardware architectures and programming languages support basic arithmetic operations with these types. The float and double datatype have been used in scientific applications for decades.

Other FP configurations. In the past few years, several new configurations of FP have been proposed and used. Especially in machine learning and high performance computing domains, the performance of computations, cost of data storage, and the data transfer rate is paramount. Additionally, a wide dynamic range is more desirable compared to high precision to produce accurate results. Thus, FP configurations with small bit-length and wide dynamic ranges have been proposed for these domains. Google proposed Bfloat16 [131] ($\mathbb{F}_{16,8}$), which is a 16-bit representation with 8 bits of exponent. Similarly, NVidia proposed

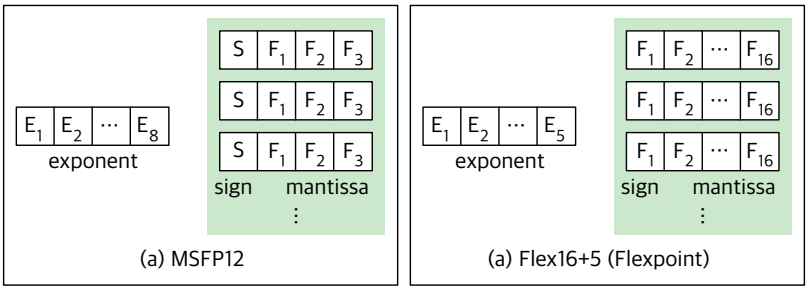


Figure 2.9: Layout of (a) MSFP12 and (b) Flex16+5, a Flexpoint configuration. Values in these representations share the same exponent field. Each MSFP12 value stores its sign bit and mantissa bits separately from the shared exponent field. Each Flexpoint value stores its mantissa bits separately, where the mantissa bits encode both the sign and significand at the same time using a signed integer format (*i.e.*, the mantissa bits of a value v and its inverse $-v$ are two's complement of each other).

tensorfloat32 [108] ($\mathbb{F}_{19,8}$), a 19-bit representation with 8 bits for exponent. Both bfloat16 and tensorfloat32 have the same number of exponent bits as the 32-bit float type. Microsoft proposed MSFP8-11 [100], which are 8 to 11-bit FP configuration with 5 bits of exponent ($\mathbb{F}_{8,5}$ to $\mathbb{F}_{11,5}$). MSFP8-11 has the same number of exponent bits as the 16-bit half type. Figure 2.8 illustrates the bit-pattern of bfloat16, TensorFloat32, and MPSF8.

Non-standard FP representations. Microsoft proposed MSFP12-16 [116], a 12 to 16-bit FP-like representation that slightly deviates from the FP semantics. Figure 2.9(a) shows the layout of MSFP12 bit-strings. Abstractly, MSFP12-16 uses 8 bits of exponent and 3 to 7 bits of mantissa bits. In MSFP12-16, the sign bit and the exponent bits are encoded in the same way that FP representation does. The mantissa bits encode the entire significand of the value, similar to how denormal values are encoded in FP representations. The novelty of MSFP12-16 is that it stores the bit-string of values with similar magnitude together. Because these values have the same exponent bits, MSFP12-16 stores only one instance of the exponent field. The sign bit and mantissa bits are stored separately for each value. Additionally, MSFP12-16 can store values with different magnitudes together by adjusting the mantissa bits (similar to denormal values in the FP representation).

Similarly, Flexpoint [78] proposed by Intel also stores the bit-string of multiple values

together. Figure 2.9(b) shows the layout of Flexpoint bit-strings. Flexpoint only maintains exponent bits and mantissa bits. To represent negative values, the mantissa bits encode the sign of the value using two's complement. More specifically, the mantissa bits of a Flexpoint value v and its additive inverse $-v$ are two's complement of each other. Flexpoint groups the bit-string of multiple values and stores only one instance of the exponent bits. The mantissa bits are adjusted to represent values of different magnitude. Both MSFP12-16 and Flexpoint representations are proposed to minimize the amount of space required to store data and create efficient hardware implementations for primitive operations.

Log number systems. The bit-string of a value x in log number systems (LNS) [44, 112, 130] encodes the logarithm of $|x|$, *i.e.*, $log_2(|x|) = e + f$ where e is an integer and f is a fractional value $f \in [0, 1)$. Alternatively, we can interpret the LNS bit-string as representing the value $x = (-1)^S \times 2^{e+f} = (-1)^S \times 2^e \times 2^f$, where S is encoded by the sign bit, e is encoded by the exponent bits, and f is encoded by the mantissa bits. Depending on the specific LNS representation, the negative exponent value $e < 0$ is encoded with exponent bits using offset (similar to bias in FP representation) or two's complement form (similar to signed integer).

For example, consider the bit-string of a value in $\mathbb{F}_{9,4}$ illustrated in Figure 2.2(a). This bit-string represents the value 1.875×2^2 in $\mathbb{F}_{9,4}$. Now, let us interpret the same bit-string in a 9-bit LNS representation with four exponent bits and four mantissa bits. For the ease of exposition, we interpret the exponent bits with the same strategy as FP representations (*i.e.*, using bias). In our 9-bit LNS representation, the exponent bits represent the value $e = 2$. The mantissa bits are interpreted as $f = 0.875$, where the leading bit is zero. Hence, the value represented by the bit-string in our 9-bit LNS representation is,

$$2^{2+0.875} = 2^{2.875} \approx 7.336\ldots$$

The main advantage of LNS is that multiplication and division operations are more efficient compared to FP representations. Without the loss of generality, let us suppose

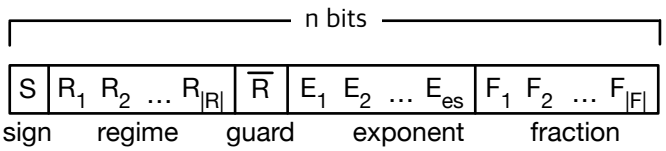


Figure 2.10: Layout of an n-bit posit representation bit-pattern.

$x_1 = 2^{e_1+f_1}$ and $x_2 = 2^{e_2+f_2}$ are two positive values in a LNS representation. Then, $x_1 \times x_2$ can be computed by adding the values e_1, e_2, f_1, and f_2,

$$x_1 \times x_2 = 2^{e_1+f_1} \times 2^{e_2+f_2} = 2^{(e_1+f_1+e_2+f_2)} = 2^{(e_1+e_2)+(f_1+f_2)}$$

Thus, multiplying two values in LNS is as simple as adding the exponent bits and the fraction bits separately. Division operation follows similarly. LNS is often used in signal processing where multiplication is one of the most commonly used primitive operations for Fourier transformations.

2.2 The Posit Representation

Posit [55, 58] is intended to be a stand-in replacement for FP. There are two notable differences in posit compared to FP. First, the posit representation provides tapered precision where different values have different amounts of precision. Specifically, posits represent values near one with higher precision while providing a large dynamic range. Second, an n-bit posit representation can represent more distinct values compared to FP. There is excitement in exploring posits in several domains [18, 45, 77, 102]. Although commercial hardware architectures do not support posit yet, there are hardware proposals for posit arithmetic [69, 70, 136].

A posit representation $\mathbb{P}_{n,es}$ is identified by two parameters, the total number of bits n and the maximum number of bits to represent the exponent es. There are five components to a posit bit-string: a sign bit S, regime bits R, a regime guard bit $\overline{R}$, up to es bits of exponent bits E, and fraction bits F. Figure 2.10(a) shows each of the five components in a posit bit-string. The number of regime bits, exponent bits, and fraction bits varies

depending on the value being represented. The posit representation uses a minimal number of regime bits and exponent bits to express the magnitude of the value. The remaining bits are used for the fraction to provide as much precision as possible. The posit standard [55] specifies four standard configurations of posit, 8-bit posit8 ($\mathbb{P}_{8,0}$), 16-bit posit16 ($\mathbb{P}_{16,1}$), 32-bit posit32 ($\mathbb{P}_{32,2}$), and 64-bit posit64 ($\mathbb{P}_{64,3}$).

2.2.1 Decoding a Posit Bit-String

The first bit in a posit bit-string is the sign bit. If $S = 0$ then the represented value is positive. If $S = 1$, then the value is negative. If the value is negative, then the bit-string is decoded after performing two's complement of the bit-string, similar to how a signed integer is decoded. The next three components, R, $\overline{R}$, and E are used to represent the exponent of the value. Abstract, the regime R is the super exponent. It extends the dynamic range of the posit representation encoded by the exponent bits E. After the sign bit, the next consecutive 1's (or 0's) represent the regime bits. The regime bits are only terminated when it encounters an opposite bit 0 (or 1), known as the regime guard bit ($\overline{R}$), or the bit-string itself terminates. The size of R can be anywhere between $1 \leq |R| \leq n - 1$.

If there are any remaining bits after the regime and regime guard bits, up to es of the next bits (*i.e.*, $min\{n - 1 - |R|, es\}$ bits) represent the exponent bits E. In the case that $|E| < es$ (*i.e.*, there were less than es remaining bits), then the exponent bits E is padded with 0's to the right until $|E| = es$. Finally, the remaining bits (*i.e.* ($max\{0, n - 1 - |R| - |E|\}$) bits) represent the fraction bits. If there is no remaining bit, then we assign a single 0 bit to F ($F = 0$).

2.2.2 Interpreting a Posit Bit-String

The sign of a posit value is determined using the sign bit, $(-1)^S$. The regime and exponent bits together contribute to the magnitude of the represented value. Let $useed = 2^{2^{es}}$ and k

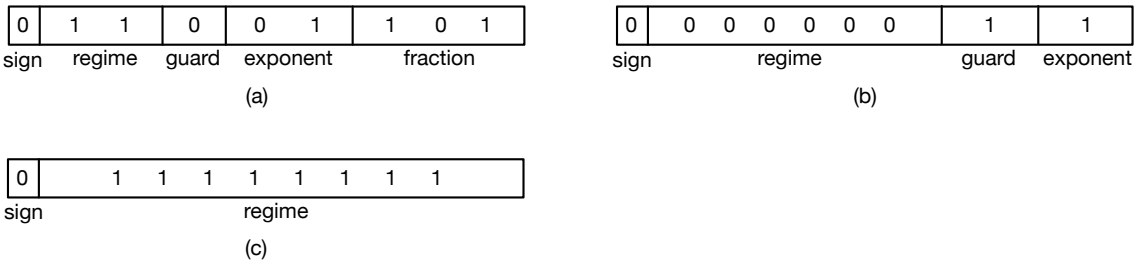


Figure 2.11: (a) Bit-string representing the value 1.625×2^5 in $\langle 9, 2\rangle$-posit. (b) Bit-string representing the value 2^{-22} in $\langle 9, 2\rangle$-posit. (c) Bit-string representing the value 2^{28} in $\langle 9, 2\rangle$-posit.

be,

$$k = \begin{cases} -r & \text{the regime bits are } 0's \\ r - 1 & \text{the regime bits are } 1's \end{cases}$$

Then, the magnitude of the represented value is,

$$useed^k \times 2^E = (2^{2^{es}})^k \times 2^E = 2^{2^{es} \times k + E}$$

where E is interpreted as an unsigned integer. The largest magnitude that the exponent bits can encode is $2^{2^{es}-1}$. This is exactly a half of useed (*i.e.*, $\frac{useed}{2}$). Thus, regime bits extend this magnitude to extends the dynamic range of representable values.

The significand represented by the posit bit-string is calculated similarly to a normal value in FP using the fraction bits F, which contributes $1 + \frac{F}{2^{|F|}}$ to the final value. Finally, the final value that a posit bit-string represents is:

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

There are two special bit-strings in each posit representation that do not follow the above rule. The bit-string of all 0's represents the value 0. The bit-string of a 1 followed by all 0's represents *not-a-real* (NaR). NaR represents all exceptional values.

Example 1 Consider the bit-string of the $\mathbb{P}_{9,2}$ configuration illustrated in Figure 2.11 (a), with a total of 9 bits with up to 2 exponent bits. The $useed$ in $\mathbb{P}_{9,2}$ is $useed = 2^{2^2} = 2^4 = 16$. The bit-string is decomposed into $S = 0$, $R = 11$, $\overline{R} = 0$, $E = 01$, and $F = 101$. Because the regime bits (R) consist of consecutive 1's, $k = |R| - 1 = 2 - 1 = 1$. Finally, the value represented by the bit-string is,

$$(-1)^0 \times useed^1 \times 2^1 \times \left(1 + \frac{5}{8}\right) = (2^{2^2})^1 \times 2^1 \times 1.625 = 1.625 \times 2^5$$

When a small number of regime bits are used, the represented value is closer to 1. More bits are used to represent the fraction bits, leading to four precision bits (one implicit 1 bit to the left of the radix point and three fraction bits in the significand).

Example 2 In some circumstances, posit bit-string may not have fraction bits or even some exponent bits. Consider the bit-string oof a $\mathbb{P}_{9,2}$ value illustrated in Figure 2.11 (b). The bit-string contains one exponent bit and no fraction bits. The bit-string decomposes into $S = 0$, $R = 000000$, $\overline{R} = 1$, $E = 10$, and $F = 0$. Because there is only one exponent bit (`1`), we pad a `0` bit to the right of the exponent bit. Regime bits (R) consist of consecutive 0's, so $k = -|R| = -6$. The value represented by this bit-string is,

$$(-1)^0 \times useed^{-6} \times 2^2 \times (1 + \frac{0}{2}) = (2^{2^2})^{-6} \times 2^2 \times 1 = 2^{-24} \times 2^2 = 2^{-22}$$

When a larger number of regime bits are used to represent the magnitude of the value, a smaller number of bits are allocated to fraction bits (in the case, 0 bits), leading to only one precision bit.

Example 3 In extreme cases, the posit bit-string may consist of only a sign bit and regime bits. Consider the bit-string of a $\mathbb{P}_{9,2}$ value in Figure 2.11 (c). There are only two components in the bit-string, $S = 0$ and $R = 11111111$. In this case, both exponent bits and

fraction bits become zeros (*i.e.*, $E = 00$ $F = 0$) and $k = R - 1 = 7$. The value represented by the bit-string is.

$$(-1)^0 \times useed^7 \times 2^0 \times (1 + \frac{0}{2}) = (2^{2^2})^7 \times 1 \times 1 = 2^{28}.$$

Comparison with FP representation There are two key differences between posit and FP. First, there is exactly one bit-pattern representing 0 and one bit-pattern representing NaR. Hence, every bit-pattern in a given posit represents unique values. Second, given the same number of bits, posit values have more precision than FP in some range of values while providing a similar amount or more dynamic range. For example, let us compare the standard 32-bit FP and posit configurations, which are float and posit32, respectively. Float values have 24 bits of precision with dynamic range of $[2^{-149}, 2^{128})$. In comparison, posit values have up to 27 bits of precision near 1 with the dynamic range of $[2^{-120}, 2^{120}]$. More specifically, posit values in the range $[2^{-20}, 2^{20}]$ have the same or more precision bits compared to float values in the range. This range is known as the golden zone [37]. A slightly different 32-bit posit configuration $\mathbb{P}_{32,3}$ provides up to 26 bits of precision, which is still more than the precision of float, with a dynamic range of $[2^{-240}, 2^{240}]$. However, as posit value approaches 0 or increases to a large value, the number of precision bits can decrease to as little as 1 precision bit, as shown in Figure 2.11(c).

2.2.3 Rounding a Real Number to the Posit Representation

The posit standard supports only one rounding mode, *round to the nearest, tie goes to even* (rne). If a real value $v_{\mathbb{R}}$ is exactly representable in a posit representation $\mathbb{P}_{n,es}$, then we round $v_{\mathbb{R}}$ to that value. If $v_{\mathbb{R}}$ is not exactly representable, then it is rounded to either v_{sm}, the largest posit value smaller than or equal to $v_{\mathbb{R}}$, or v_{lg}, the smallest value larger than or equal to $v_{\mathbb{R}}$. The high-level idea of posit rounding is similar to rne rounding mode in FP representations: $v_{\mathbb{R}}$ rounds to the closer value between v_{sm} or v_{lg}. If $v_{\mathbb{R}}$ is a midpoint

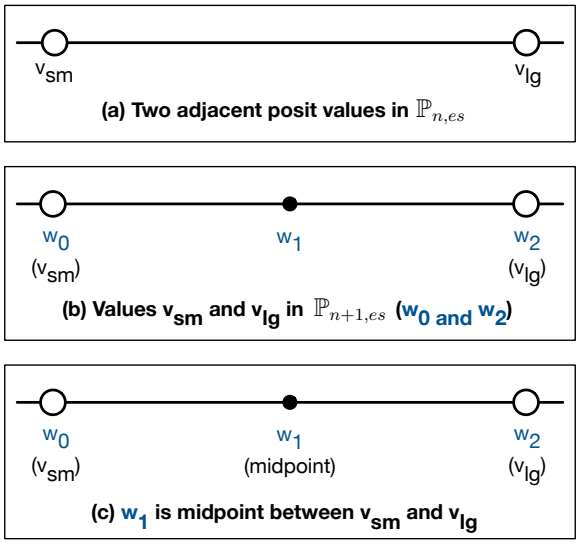


Figure 2.12: Illustration of identifying the midpoint between two posit values v_{sm} and v_{lg}. (a) Suppose there are two adjacent posit values v_{sm} and v_{lg} in $\mathbb{P}_{n,es}$. (b) Then, the two values are exactly representable in $\mathbb{P}_{n+1,es}$ (w_0 and w_2, respectively) and there is a value $w_1 \in \mathbb{P}_{n+1,es}$ between w_0 and w_2. (c) The value w_1 is the midpoint between v_{sm} and v_{lg} when rounding a real value $v_{\mathbb{R}}$ to $\mathbb{P}_{n,es}$.

between v_{sm} or v_{lg}, then $v_{\mathbb{R}}$ rounds to the value where the bit-string is even when interpreted as an unsigned integer.

Definition of midpoint for posit representations. However, the definition of *midpoint* for posit representations is slightly different compared to the *midpoint* in the rne rounding mode for FP representations. In posit representations, the midpoint of two adjacent values v_{sm} and v_{lg} in $\mathbb{P}_{n,es}$ is defined as the value in $\mathbb{P}_{n+1,es}$ between v_{sm} and v_{lg}. We illustrate the midpoint for posit representations with Figure 2.12. Given two adjacent posit values v_{sm} and v_{lg} in the posit representation $\mathbb{P}_{n,es}$ (shown in Figure 2.12(a)), both v_{sm} and v_{lg} are exactly representable in $\mathbb{P}_{n+1,es}$ (the values w_0 and w_2, respectively in Figure 2.12(b)). Additionally, there is another value $w_1 \in \mathbb{P}_{n+1,es}$ between v_{sm} and v_{lg}. The value w_1 is the midpoint between v_{sm} and v_{lg}.

Once the midpoint between v_{sm} and v_{lg} are identified, then rounding a real value $v_{\mathbb{R}}$ between v_{sm} and v_{lg} follows the rule similar to FP's rne rounding mode. If $v_{\mathbb{R}}$ is smaller than the midpoint, then $v_{\mathbb{R}}$ rounds to v_{sm}. If $v_{\mathbb{R}}$ is larger than the midpoint, then $v_{\mathbb{R}}$ rounds to

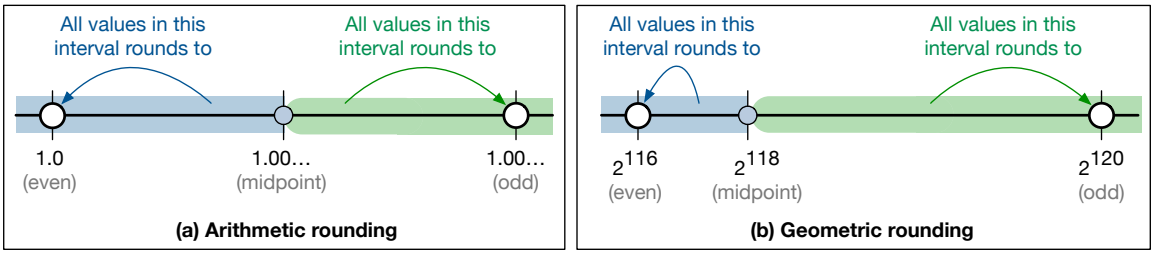


Figure 2.13: Illustration of posit rounding behavior when rounding a real value $v_{\mathbb{R}}$ to $\mathbb{P}_{32,2}$. These behaviors are present for all posit configurations. (a) In most cases, posit exhibits arithmetic rounding behavior. (b) However, in some special cases, posit exhibits geometric rounding behavior.

v_{lg}. If $v_{\mathbb{R}}$ is equal to the midpoint, then we round to the value where the bit-string representation in $\mathbb{P}_{n,es}$ is even when interpreted as an unsigned integer. This method of identifying midpoint allows systematically rounding $v_{\mathbb{R}}$ to a posit value to follow a similar strategy as rounding $v_{\mathbb{R}}$ to an FP value, while still maintaining the tapered-precision property.

Different behaviors in posit rounding. In the FP representation's rne mode, the midpoint of two adjacent values v_{sm} and v_{lg} is the arithmetic mean between the two values (*i.e.*, $\frac{v_{sm}+v_{lg}}{2}$). Rounding a real value $v_{\mathbb{R}}$ using the arithmetic mean of two adjacent values as the midpoint is called arithmetic rounding. Instead, rounding $v_{\mathbb{R}}$ using the geometric mean of two adjacent values (*i.e.*, $\sqrt{v_{sm}v_{lg}}$) as the midpoint is called geometric rounding. Interestingly, posit rounding has both arithmetic rounding and geometric rounding depending on the magnitude of $v_{\mathbb{R}}$. In most cases, the midpoint is the arithmetic mean between v_{sm} and v_{lg}, *i.e.* $\frac{v_{sm}+v_{lg}}{2}$ as illustrated in Figure 2.13(a). Specifically, this occurs when the last bit of the midpoint in $\mathbb{P}_{n+1,es}$ represents a fraction bit. The values v_{sm}, the midpoint, and v_{lg} increase arithmetically and rounding $v_{\mathbb{R}}$ between v_{sm} and v_{lg} exhibit arithmetic rounding.

However, when $v_{\mathbb{R}}$ is an extremal value, posit rounding can exhibit geometric rounding. Figure 2.13(b) illustrates geometric rounding by rounding $v_{\mathbb{R}}$ to $\mathbb{P}_{32,2}$ where $v_{\mathbb{R}}$ is between two largest representable posit values $v_{sm} = 2^{116}$ and $v_{lg} = 2^{120}$. The midpoint w_1 between v_{sm} and v_{lg} is 2^{118} where w_1 is a factor of 4 larger than v_{sm} and v_{lg} is a factor of 4 larger than w_1. Hence, w_1 is the geometric mean of v_{sm} and v_{lg} (*i.e.* $\sqrt{2^{116}2^{120}} = 2^{118}$) and $v_{\mathbb{R}}$ is geometrically rounded to the nearest value. This occurs specifically when the last bit of the

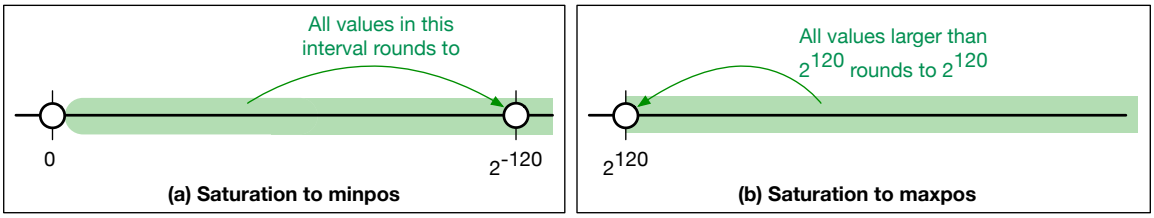


Figure 2.14: When rounding a real value $v_{\mathbb{R}}$ to a posit representation (*i.e.*, $\mathbb{P}_{32,2}$), there are two types of special cases. Without the loss of generality, assume that $v_{\mathbb{R}} \geq 0$. (a) All $v_{\mathbb{R}}$ greater than 0 and smaller than the smallest positive representable value $minpox$ (*i.e.*, $minpos = 2^{-120}$) rounds to $minpos$. (b) All $v_{\mathbb{R}}$ greater than the largest positive representable value $maxpos$ (*i.e.*, $maxpos = 2^{120}$ rounds to $maxpos$.)

midpoint in the $\mathbb{P}_{n+1,es}$ representation is either a regime bit, regime guard bit, or exponent bits.

Special cases in posit rounding. Additionally, there are two classes of special cases when rounding $v_{\mathbb{R}}$ to a posit value. Without the loss of generality, let us assume that $v_{\mathbb{R}} \geq 0$. The special cases apply to negative $v_{\mathbb{R}}$ similarly. In posit rounding, the only real value that rounds to 0 is $v_{\mathbb{R}} = 0$ itself. Any value greater than 0 and smaller than the smallest representable positive value ($minpos$) rounds to $minpos$. Figure 2.14(a) illustrates this special case for the 32-bit standard posit representation $\mathbb{P}_{32,2}$. This special case ensures that the rounded value preserves the sign of the original real value $v_{\mathbb{R}}$. In contrast, FP rounding can underflow non-zero values to 0, losing the sign of $v_{\mathbb{R}}$.

Second, posit does not have a representation for $\pm\infty$. Any positive real value outside of the dynamic range of posit representation is rounded to the largest representable positive value, $maxpos$. Figure 2.14(b) illustrates this case for the 32-bit standard posit representation $\mathbb{P}_{32,2}$. The largest positive value in $\mathbb{P}_{32,2}$ is 2^{120}. All values larger than 2^{120} rounds to 2^{120}. Thus, posit rounding always rounds real values $v_{\mathbb{R}}$ into a real posit value. In contrast, the rne rounding mode in FP representations can overflow real values to ∞, making no distinction between values that are ∞ and values that are just too large to represent in the chosen FP representation.

2.2.4 A Systematic Methodology for Rounding To Posit Representation

Rounding a real value $v_{\mathbb{R}}$ to a posit value in $\mathbb{P}_{n,es}$ follows a similar strategy as rounding to FP representations. Intuitively, we identify the four rounding components, s, v^-, rb, and $sticky$ to identify the two values v_{sm} and v_{lg} in $\mathbb{P}_{n,es}$ that are adjacent to $v_{\mathbb{R}}$ and decide which value $v_{\mathbb{R}}$ rounds to. To provide a brief of how to identify the rounding components, we first decompose $v_{\mathbb{R}}$ into $v_{\mathbb{R}} = s \times |v_{\mathbb{R}}|$ where s is the first rounding component that represents the sign of $v_{\mathbb{R}}$. Next, we represent $|v_{\mathbb{R}}|$ in the infinite extended precision representation $\mathbb{P}_{\infty}$, where $\mathbb{P}_{\infty}$ has the same limit in the es bits but can have infinitely many regime bits and fraction bits. Because the number of available regime and fraction bits is unbounded, any real value can be exactly represented in $\mathbb{P}_{\infty}$. Third, we truncate $B_{|v_{\mathbb{R}}|}$, the bit-string of $|v_{\mathbb{R}}|$ in $\mathbb{P}_{\infty}$, to n bits to obtain the truncated value v^-, which is the second rounding component. We identify the rounding bit by extracting the $(n+1)^{st}$ bit in $B_{|v_{\mathbb{R}}|}$ to identify the third rounding component rb, the rounding bit. Finally, we perform a bit-wise `or` operation on all bits in $B_{|v_{\mathbb{R}}|}$ starting from the $(n+2)^{nd}$ bit to identify the last rounding component, $sticky$.

The truncated value v^- is the largest value smaller than or equal to $|v_{\mathbb{R}}|$. Then v^+, the succeeding value of v^- is the smallest value larger than $|v_{\mathbb{R}}|$. Additionally, the value that we obtain by concatenating v^- and the rounding bit rb, then decoding the resulting $(n+1)$-bit bit-string in $\mathbb{P}_{n+1,es}$ is the midpoint of v^- and v^+. Thus, the four rounding components can be used to determine the relationship between $|v_{\mathbb{R}}|$ and the two nearest posit values:

$$\begin{cases} |v_{\mathbb{R}}| = v^- & \text{if } rb = 0 \wedge sticky = 0 \\ v^- < |v_{\mathbb{R}}| < v_{mid} & \text{if } rb = 0 \wedge sticky = 1 \\ |v_{\mathbb{R}}| = v_{mid} & \text{if } rb = 1 \wedge sticky = 0 \\ v_{mid} < |v_{\mathbb{R}}| < v^+ & \text{if } rb = 1 \wedge sticky = 1 \end{cases}$$

where v_{mid} is the midpoint between v^- and v^+.

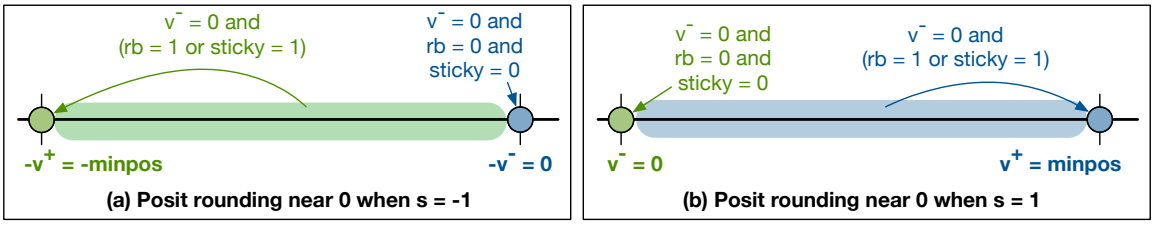


Figure 2.15: (a) Illustration of rounding a real value $v_{\mathbb{R}}$ to a posit value when $-minpos \leq v_{\mathbb{R}} < 0$ where $-minpos$ is the largest negative representable value. (b) Illustration of rounding a real value $v_{\mathbb{R}}$ to a posit value when $0 < v_{\mathbb{R}} \leq minpos$ where $minpos$ is the smallest positive representable value. All real values in the green interval rounds to the posit value with the green color. All real values in the blue interval rounds to the posit value with the blue color.

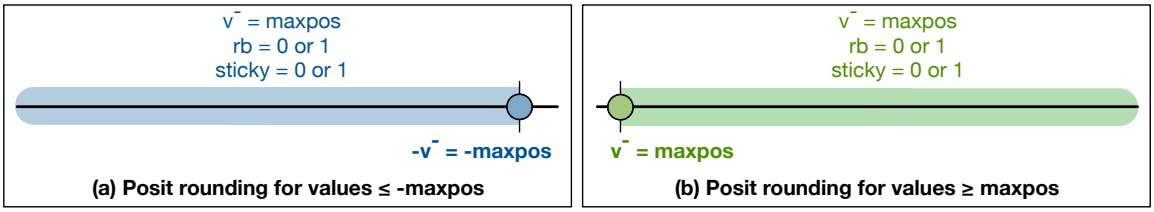


Figure 2.16: (a) When $v_{\mathbb{R}}$ is smaller than the smallest representable negative value in a posit representation, *i.e.*, $v_{\mathbb{R}} \leq -maxpos$, then $v_{\mathbb{R}}$ rounds to $-maxpos$. This case can be identified with $s = -1$ and $v^- = maxpos$. (b) When $v_{\mathbb{R}}$ is larger than the largest representable positive value in a posit representation, *i.e.*, $v_{\mathbb{R}} \geq maxpos$, then $v_{\mathbb{R}}$ rounds to $maxpos$. This case can be identified with $s = 1$ and $v^- = maxpos$.

Determining the correctly rounded result of $v_{\mathbb{R}}$**.** We now explain the rounding decision and the final rounded value of $v_{\mathbb{R}}$ in the posit representation using the four rounding components. First, we determine whether $v_{\mathbb{R}}$ is a special case value in posit rounding. If $v^- = 0$, $rb = 0$, and $sticky = 0$, then $v_{\mathbb{R}} = 0$. In this case, $v_{\mathbb{R}}$ rounds to 0. If $v^- = 0$ and either $rb = 1$ or $sticky = 1$, then it indicates that $|v_{\mathbb{R}}|$ is larger than 0 but smaller than v^+, the smallest representable posit value $minpos$. Then, $v_{\mathbb{R}}$ rounds to $s \times v^+$. Figure 2.15(a) and (b) pictorially shows the rounding decision of $v_{\mathbb{R}}$ to posit value based on different values of rounding components when $v^- = 0$.

Similarly, if v^- is equal to the largest representable posit value $maxpos$, then it indicates that $|v_{\mathbb{R}}|$ is a value outside of the dynamic range of $\mathbb{P}_{n,es}$. In posit rounding, values outside of the dynamic range rounds to $\pm maxpos$. Thus, $v_{\mathbb{R}}$ rounds to $s \times v^-$. Figure 2.16(a) and (b) illustrates the rounding decision of $v_{\mathbb{R}}$ based on different values of rounding components

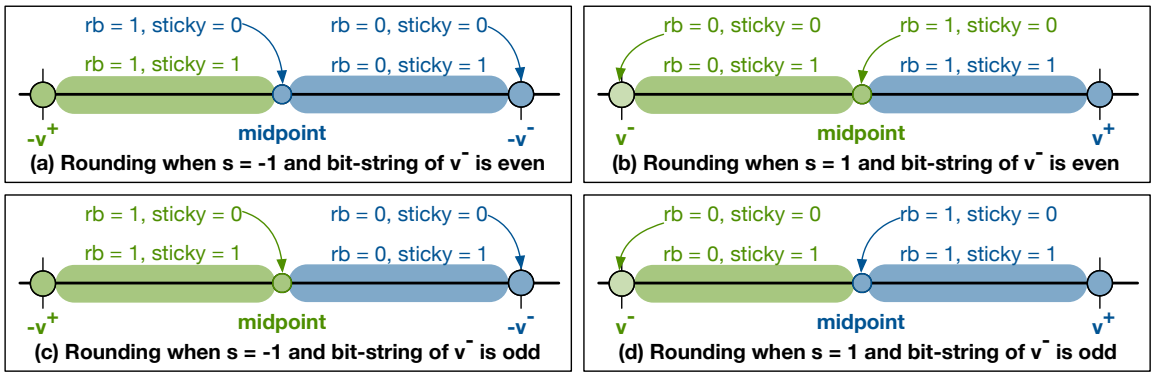


Figure 2.17: The posit rounding rule using the rounding components for all $v_{\mathbb{R}}$ within the dynamic range of the posit representation. When an interval is colored green, all real values in the interval rounds to the smaller value (*i.e.*, $-v^{+}$ when $v_{\mathbb{R}} < 0$ and v^{-} when $v_{\mathbb{R}} \geq 0$). Similarly, all real values in the interval colored blue will round to the larger posit value (*i.e.*, $-v^{-}$ when $v_{\mathbb{R}} < 0$ and v^{+} when $v_{\mathbb{R}} \geq 0$).

when $v_{\mathbb{R}}$ is outside of the dynamic range of the posit representation.

If $v_{\mathbb{R}}$ is not a special case value, then $v_{\mathbb{R}}$ is rounded using the same rule as the rne rule in FP representations. If the rounding bit $rb = 0$, then it indicates that $|v_{\mathbb{R}}|$ is smaller than the midpoint v_{mid}. In this case, $v_{\mathbb{R}}$ rounds to $s \times v^{-}$. If the rounding bit $rb = 1$ and the sticky bit $sticky = 1$, then it indicates that $|v_{\mathbb{R}}|$ is greater than v_{mid} and $v_{\mathbb{R}}$ rounds to $s \times v^{+}$. If $|v_{\mathbb{R}}|$ is equal to the midpoint, then the rounding bit $rb = 1$ and the sticky bit $sticky = 0$. In this case, we round to the value whose bit-string is even when interpreted as an unsigned integer. In the posit representation, the bit-string of a posit value v and its negative counterpart $-v$ are two's complement of each other. The two's complement operation preserves the parity of the bit-string. Thus, the bit-string of v^{-} is even if and only if the bit-string of $-v^{-}$ is even. It suffices to check the parity of v^{-} to determine the correct rounding decision. If $rb = 1$, $sticky = 0$, and v^{-} is even, then $v_{\mathbb{R}}$ rounds to $s \times v^{-}$. Otherwise, $v_{\mathbb{R}}$ rounds to $s \times v^{+}$. Figure 2.17 pictorially shows the rounding decision of $v_{\mathbb{R}}$ to a posit representation based on different values of rounding components. The posit

rounding decision including the special cases can be summarized as follows,

$$RN_{\mathbb{P},rne}(v_{\mathbb{R}}) = \begin{cases} 0 & \text{if } v^- = 0 \wedge rb = 0 \wedge sticky = 0 \\ s \times v^+ & \text{if } v^- = 0 \wedge (rb = 1 \vee sticky = 1) \\ s \times v^- & \text{if } v^- = maxpos \\ s \times v^- & \text{if } v_{\mathbb{R}} \text{ is not special case} \wedge rb = 0 \\ s \times v^- & \text{if } v_{\mathbb{R}} \text{ is not special case} \wedge rb = 1 \wedge sticky = 0 \wedge IsEven(v^-) \\ s \times v^+ & \text{if } v_{\mathbb{R}} \text{ is not special case} \wedge rb = 1 \wedge sticky = 0 \wedge \neg IsEven(v^-) \\ s \times v^+ & \text{if } v_{\mathbb{R}} \text{ is not special case} \wedge rb = 1 \wedge sticky = 1 \end{cases}$$

2.2.5 Posit Configurations and Posit Variants

The posit standard [55] suggests four standard configurations: The 8-bit posit8 ($\mathbb{P}_{8,0}$), 16-bit posit16 ($\mathbb{P}_{16,1}$), 32-bit posit32 ($\mathbb{P}_{32,2}$), and 64-bit posit64 ($\mathbb{P}_{64,3}$). These configurations are specifically chosen to provide both a wide dynamic range as well as high precision. Apart from the standard posit representation, a tapered-precision log number system by combining posit representations and LNS has been explored and shown to be effective in the machine learning domain [74].

2.3 Numerical Errors in Finite Precision Representation

Because any FP representation can only represent a finite number of real values, the results of primitive operations (*i.e.*, rounding, addition, subtraction, multiplication, and division) can experience rounding error. Modern hardware and libraries produce correctly rounded results for these primitive operations. However, rounding errors can get amplified with a certain combination of primitive operations because the intermediate results need to be rounded [37, 52, 104]. We highlight three important classes of numerical errors that are highly relevant to generating correctly rounded math libraries.

2.3.1 Rounding Error with Extremal Values.

Any representation $\mathbb{T}$ has a limit on the dynamic range. When we round a real value $v_{\mathbb{R}}$ outside of the dynamic range, then a significant amount of error between $v_{\mathbb{R}}$ and the rounded results can occur. There are three classes of such rounding errors depending on the rounding mode: overflow, underflow, and saturation error.

Overflow error occurs when a value larger than the largest representable value in $\mathbb{T}$ is rounded to ∞ (*i.e.* 2^{130} rounds to ∞ in the 32-bit float type using the rne mode). When evaluating an expression in $\mathbb{T}$, if an intermediate result encounters an overflow error, then the final value of the expression will likely evaluate to ∞, 0, or NaN. It can occur even if the expression evaluates to a value representable by $\mathbb{T}$ if evaluated in reals. Underflow error occurs when values close to 0 rounds to 0 (*i.e.* 2^{-150} rounds to 0 in the 32-bit float using the rne mode). Although the absolute error of $v_{\mathbb{R}}$ and 0 may be small, underflow error loses the information on the sign of $v_{\mathbb{R}}$, which can be detrimental in some applications. Finally, saturation error occurs when any value outside of the dynamic range rounds to either the smallest or the largest representable value. Rounding posit values experience saturation error.

All three types of error (overflow, underflow, and saturation error) can produce results with significant error, which can make accurate approximation and error analysis difficult. Thus, math libraries typically classify inputs that cause these errors into special cases and directly return the correct result.

2.3.2 Double Rounding

Double rounding occurs when a value $v_{\mathbb{R}}$ is first rounded to a representation $\mathbb{T}_1$ using a rounding mode rm_1, then rounded again to a smaller representation $\mathbb{T}_2$ using a rounding mode rm_2. In some cases, double rounding is harmless and produces the same result as if rounding $v_{\mathbb{R}}$ directly to $\mathbb{T}_2$ using rm_2 rounding mode. However, with a certain combination of rm_1 and rm_2, it is not guaranteed that double rounding produces the correctly rounded

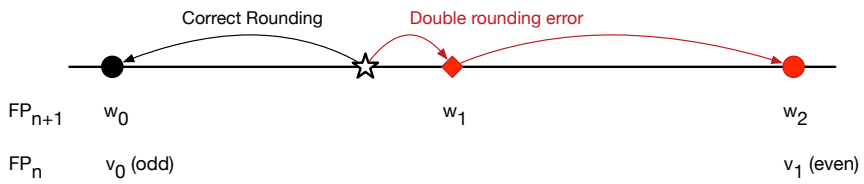


Figure 2.18: An illustration of the double rounding error. Star represents a real value $v_{\mathbb{R}}$. Circles represent the two values v_1 and v_2 adjacent to $v_{\mathbb{R}}$ in an n-bit representation $\mathbb{T}_n$. An $(n+1)$-bit representation $\mathbb{T}_n$ can exactly represent both v_1 and v_2 (w_0 and w_2, respectively). In addition, there is another value w_1 between w_0 and w_1, highlighted with the rhombus. Rounding $v_{\mathbb{R}}$ to $\mathbb{T}_{n+1}$ and then subsequently rounding the result $\mathbb{T}_n$ produces a value different from directly rounding $v_{\mathbb{R}}$ to $\mathbb{T}_n$.

result, known as the double rounding error. Intuitively, double rounding error occurs because the error in rounding $v_{\mathbb{R}}$ to $\mathbb{T}_1$ is sometimes significant enough to affect the rounding decision of rounding the value in $\mathbb{T}_1$ to $\mathbb{T}_2$. We illustrate double rounding error with Figure 2.18. Consider a case of rounding a real value $v_{\mathbb{R}}$ (star) to an n-bit representation $\mathbb{T}_n$. There are two values v_0 and v_1 in $\mathbb{T}_n$ adjacent to $v_{\mathbb{R}}$. When we directly round $v_{\mathbb{R}}$ to $\mathbb{T}_n$ using the rne rounding mode, $v_{\mathbb{R}}$ rounds to v_0. Instead, let us perform double rounding by first rounding $v_{\mathbb{R}}$ to $\mathbb{T}_{n+1}$, an $(n+1)$-bit representation with 1 additional precision bit compared to $\mathbb{T}_n$. $\mathbb{T}_{n+1}$ can exactly represent v_0 and v_1. Additionally, $\mathbb{T}_{n+1}$ can exactly represent another value w_1 between v_0 and v_1. Using the rne rounding mode, $v_{\mathbb{R}}$ rounds to w_1 in $\mathbb{T}_{n+1}$. If we then round the result w_1 to $\mathbb{T}_n$ using rne rounding mode, it rounds to v_1 which is not the correctly rounded result of $v_{\mathbb{R}}$ in $\mathbb{T}_n$.

2.3.3 Cancellation Error

When two similar but inexact values are subtracted from each other, cancellation errors can occur. These two similar values have the same exponent bits and several significant bits of the fraction bits are the same. Subtracting the values result in the significant fraction bits becoming zero. The result then depends on the least significant bits, which are inexact, significantly increasing the error of the result. Cancellation error where all bits are wrong with respect to the correct answer obtained by evaluating the expression in real numbers is

known as the catastrophic cancellation. Typically, catastrophic cancellation results in the subtraction operation to produce 0 (when the correct answer should not be 0), leading to difficulties in accurately evaluating mathematical expressions.

Illustration of catastrophic cancellation. Consider an example of evaluating the expression $b^2 - 4ac$ using $\mathbb{F}_{9,4}$ representation with rne mode, where $a = 1.125$, $b = 2.0$, and $c = 0.875$. All three values are exactly representable. Mathematically, the result of the expression is $b^2 - 4ac = 0.0625$, a value that can also be exactly represented with $\mathbb{F}_{9,4}$.

If we evaluate $b^2 - 4ac$ with a 9-bit FP representation $\mathbb{F}_{9,4}$, then $b^2 = 4.0$ rounds to 4.0, which is an exact result. However, $4ac = 3.9375$ rounds to 4.0 because 3.9375 cannot be exactly represented. This rounding error combined with the subtraction operation causes catastrophic cancellation, producing an incorrect result 0 (compared to the correct result 0.0625). If this result is used to determine the number of roots in $ax^2 + bx + c$, then it can have catastrophic consequences.

2.4 Prior Work on Approximating Elementary Functions

The state-of-the-art techniques in approximating an elementary function $f(x)$ for a target representation $\mathbb{T}$ involves two steps in general. First, mathematical reasoning and approximation theory is used to derive a function $A_{\mathbb{R}}(x)$ that approximates $f(x)$ in real numbers. Second, the approximation function $A_{\mathbb{R}}(x)$ is implemented as $A_{\mathbb{H}}(x)$ using a finite precision $\mathbb{H}$ that has higher precision than $\mathbb{T}$.

2.4.1 Approximating $A_{\mathbb{R}}(x)$

Deriving $A_{\mathbb{R}}(x)$ further involves three steps. First, we identify inputs that exhibit special behavior (*i.e.*, $\pm\infty$). Second, we use range reduction to reduce the entire input domain $[a, b]$ to a smaller domain $[a', b'] \in [a, b]$ and perform any other function transformations. Depending on the range reduction and transformations used, the function that needs to be

approximated may be transformed into a different function $g(x)$. Third, we generate a polynomial $P(x)$ that closely approximates $g(x)$ in the domain $[a', b']$.

Identifying special cases. There are three types of special cases. The first type includes inputs that produce undefined values (*i.e.*, NaN or NaR) or $\pm\infty$. For example, in the case of $f(x) = e^x$, $f(x) = \infty$ if $x = \infty$ and $f(x) = 0$ if $x = -\infty$. The second type consists of hard to compute results. For example, the only real number that rounds to 0 in the posit representation is 0 itself. In the case of approximating $f(x) = ln(x)$ for posit representations, it is simpler to classify $x = 1$ as a special case because $ln(1) = 0$ and the slightest amount of error in approximation will round to a non-zero result.

The third type consists of inputs that produce interesting outputs when evaluating $f(x)$. These cases include inputs that produce results with underflow, overflow, saturation error, or ranges of inputs that produce the same rounded result $RN_{\mathbb{T},rm}(f(x))$. For example, when approximating $f(x) = e^x$ for the 32-bit float with the rne mode, all inputs $x \in (-\infty, -103.9\ldots]$ produce the result $RN_{\mathbb{F},rne}(e^x) = 0$ (underflow), all inputs $x \in [88.72\ldots, \infty)$ produces results that round to ∞ (overflow), and the results for inputs between $[-2.98\cdots \times 10^{-8}, 5.96\cdots \times 10^{-8}]$ rounds to 1.0. These inputs can be explicitly filtered out by directly returning the result, significantly reducing the input domain where we have to approximate $f(x)$.

Range reduction. It is mathematically simpler to approximate $f(x)$ for a small domain of inputs. Therefore, most math libraries use range reduction to reduce the entire input domain $[a, b]$ into a smaller domain $[a', b']$ before generating the polynomial approximation. The basic intuition of range reduction is to reduce an input $x \in [a, b]$ to a different value $x' \in [a', b']$, where $[a', b'] \subseteq [a, b]$. Then, the polynomial $P(x)$ approximates the output y' using the reduced input x', $y' = P(x')$. Finally, the output y' is compensated to produce the result for the original input x. The process of reducing the input is called the range reduction ($x' = RR(x)$) and compensating the output y' to produce y is called output

compensation ($y = OC(y')$).

Example of range reduction. Consider the function $f(x) = log_2(x)$ where the input domain is defined over $(0, \infty)$. One way to reduce the range of inputs is to use the mathematical identity $log_2(a \times 2^b) = log_2(a) + b$. We decompose the input x into $x = (1+x') \times 2^E$ by representing x using the scientific notation. Here, $x' \in [0, 1)$ is the fractional part of the significand and the exponent E is an integer. Using the identity,

$$log_2(x) = log_2((1 + x') \times 2^E) = log_2(1 + x') + log_2(2^E) = log_2(1 + x') + E$$

The result of $log_2(x)$ can be approximated by computing $log_2(1 + x') + E$. Thus, we can approximate $log_2(x)$ by first range reducing the input x into the reduced input x'. The value E can be identified by computing the exponent of x. Then, we approximate $g(x') = log_2(1 + x')$ using a polynomial approximation $P(x')$. Finally, we compensate the output of $P(x')$ by computing $P(x') + E$, where the value of E is dependent on the original input x. The range reduction function, output compensation function, and the function we must approximate can be summarized as follows,

$$RR(x) = \frac{x}{2^E} - 1 \qquad OC(y') = y' + E \qquad g(x') = log_2(1 + x')$$

This range reduction reduces the domain of inputs that $P(x')$ has to approximate from $(0, \infty)$ to $[0, 1)$. Note that even though our goal was to approximate $log_2(x)$, the function that we have to approximate with $P(x')$ is $g(x') = log_2(1 + x')$. Math libraries often exploit this property of range reduction to transform $f(x)$ to another function $g(x)$ that may be easier to approximate.

Suitable range of reduced input domain. Reducing the size of the input domain means that $P(x')$ needs to be accurate only for the reduced domain $[a', b']$. Compared to the entire input domain, we may be able to generate lower degree polynomial approximation while maintaining the same error threshold. In addition, if the range reduction reduces the domain

to within $(-1, 1)$ (*i.e.*, $[a', b'] \subseteq (-1, 1)$) as shown in the above example, then the accuracy of the polynomial approximation can be increased significantly higher. To understand the intuition, consider the Taylor series expansion of $log_2(1 + x')$,

$$log_2(1 + x') = \frac{x'}{ln(2)} - \frac{x'^2}{ln(4)} + \frac{x'^3}{ln(8)} - \frac{x'^4}{ln(16)} + \dots$$

While this infinite polynomial expansion is equivalent to $log_2(1 + x')$, a finite-degree polynomial (truncated after a number of terms) is an approximation of $log_2(1 + x')$. The longer the polynomial, the more accurate the approximation is. However, the convergence rate of the polynomial expansion can vary significantly depending on the value of x'. If x' is significantly larger than 1, then the magnitude of each subsequent term keeps becoming larger (*i.e.*, $\frac{x'^{i+1}}{ln(2)^{i+1}} > \frac{x'^{i}}{ln(2)^{i}}$) until i becomes large enough. Thus, the polynomial approximation may even diverge from the result for several terms before converging to the result. It will require a large degree polynomial to accurately approximate $log_2(1 + x')$.

Comparatively, if $x' \in (-1, 1)$, then the magnitude of each subsequent term will be smaller (*i.e.*, $\frac{x'^{i}}{ln(2)^{i}} > \frac{x'^{i+1}}{ln(2)^{i+1}}$). The lower degree terms contribute the most to the final result. In such a case, the polynomial expansion converges much faster and we can approximate $log_2(1 + x')$ accurately with a polynomial of a small degree. Additionally, because the magnitude of each subsequent term is much smaller, the subtraction will not experience cancellation errors. Hence, it is ideal to design range reduction such that the reduced input domain is a subset of $(-1, 1)$ and ideally as close to 0 as possible.

Types of range reductions. There are two categories of range reductions: Multiplicative reduction and additive reduction. A multiplicative range reduction occurs when the reduced input x' is computed using either a multiplication or a division, *i.e.* $x' = x \times C$ for a constant C. An additive range reduction occurs when the reduced input x' is computed using either an addition or a subtraction, *i.e.*, $x' = x + C$.

Like any other computations, range reduction can also experience numerical error. Ad-

ditive reduction must be designed with extra care due to catastrophic cancellation. Consider the elementary function $sin(x)$ where the function is defined over the input domain $(-\infty, \infty)$. The $sin(x)$ function is a periodic function with the mathematical identify $sin(x + 2\pi) = sin(x)$. Thus, we can decompose the input x into $x = x' + 2\pi k$ where k is an input and $x' \in [0, 2\pi)$. Then, $sin(x)$ can be computed with $sin(x')$. This range reduction allows us to approximate $sin(x)$ only for the input domain $[0, 2\pi)$. In this range reduction strategy, the reduced input x' can be calculated with,

$$x' = x - 2\pi k$$

However, because π is an irrational value, finite precision representations like FP or posit cannot exactly represent π or $2\pi k$. If both x and $2\pi k$ are significantly large values, then $x' - 2\pi k$ can experience catastrophic cancellation resulting in $x' = 0$. The result of $sin(x')$ will then be significantly different from $sin(x)$.

Unlike the additive range reduction illustrated for $sin(x)$, the range reduction for $log_2(x)$ described above (*i.e.*, $x' = \frac{x}{2^E} - 1$) has been carefully reasoned and determined that it does not experience numerical error when evaluated in FP representation. The expression $\frac{x}{2^E}$ can be computed exactly. Additionally, the result of the expression is a value between 1 and 2, *i.e.*, $\frac{x}{2^E} \in [1, 2)$. Sterbenz Lemma [126] states that the subtraction of any two FP values x and y can be computed exactly as long as $\frac{y}{2} \leq x \leq 2y$. Thus, $\frac{x}{2^E} - 1$ can be computed exactly.

Polynomial approximation. A common approach to approximate an elementary function $f(x)$ is with a polynomial function $P(x)$. Compared to other iterative approximation methods, polynomial approximations can be implemented efficiently with addition, subtraction, and multiplication operations only. Polynomial approximations are known to be one of the most efficient approximation techniques.

There are two widely used approaches in generating polynomial approximations. The least-squares approximation aims to minimize the L_2-norm of the polynomial compared to

the real result of $f(x)$,

$$\min ||P(x) - f(x)||_2 = \min \sqrt{\int_a^b (P(x) - f(x))^2}$$

The least-square approximation generates polynomials that minimize the average error. Thus it is a useful technique to generate a polynomial that approximates unstable data with outliers. However, because there is no bound on the error for a specific input (*i.e.*, $|P(x_i) - f(x_i)|$ for an arbitrary input x_i in the input domain), it is not commonly used to approximate elementary functions. In the context of math libraries, the goal lies in minimizing the error for all inputs, rather than most inputs.

The more common approach to generate $P(x)$ is the minimax approximation, which aims to minimize the maximum error or L_∞-norm,

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

where sup is the supremum function that identifies the maximum value in a set. Minimizing the L_∞-norm provides a guarantee on the bound of the error for all inputs, making it a more appealing approach for approximating mathematical functions. Thus, most prior approaches in generating polynomial approximation use the minimax approach, based on the Weierstrass approximation theorem and the Chebyshev alternating theorem [137]. The Weierstrass approximation theorem proves the existence of a minimax polynomial: If $f(x)$ is a continuous real function in the input domain $[a, b]$, there exists a polynomial $P(x)$ such that the maximum error is bounded, *i.e.*, $|f(x) - P(x)| < \epsilon$ where $\epsilon > 0$ for all $x \in [a, b]$. The Chebyshev alternating theorem extends the Weierstrass approximation theorem and states that the polynomial of degree d that minimizes the maximum error is guaranteed to have exactly $d + 2$ points where the error of $P(x)$ at these points is the maximum and it alternates in sign. Remez algorithm [114] is the state-of-the-art method in generating such a polynomial. Remez algorithm also provides the maximum error of $P(x)$, which can be

used to analyze the error bound of the approximation function.

2.4.2 Implementation in Finite Precision

Finally, the mathematically derived approximation $A_{\mathbb{R}}(x)$ of an elementary function $f(x)$ is implemented in finite precision representation. Typically, the implementation uses a higher precision representation than the intended target representation ($\mathbb{T}$) to minimize the error of the intermediate result when evaluating range reduction, output compensation, and polynomial approximation. We use $A_{\mathbb{H}}(x)$ to represent the implementation of $A_{\mathbb{R}}(x)$ using the representation $\mathbb{H}$, which has higher precision than $\mathbb{T}$ (*i.e.*, $\mathbb{T} \subseteq \mathbb{H}$). The result of $A_{\mathbb{H}}(x)$ is then rounded to $\mathbb{T}$ and produces the final result.

2.5 Challenges in Generating Correctly Rounded Functions by Approximating $f(x)$

An approximation function of $f(x)$ is defined to be a correctly rounded function for $\mathbb{T}$ representation with rm rounding mode if it evaluates to $RN_{\mathbb{T},rm}(f(x))$ for all inputs x in the input domain. There are two key challenges in generating correctly rounded approximation. First, because $P(x')$ is an approximation of $f(x')$, $P(x')$ incurs approximation error $\epsilon_{approx} = |P(x') - f(x')| > 0$. Second, evaluating $A_{\mathbb{H}}(x)$ in a finite precision representation incurs rounding error compared to $A_{\mathbb{R}}(x)$: $\epsilon_{round} = |A_{\mathbb{R}}(x) - A_{\mathbb{H}}(x)| > 0$.

Because $P(x')$ is an approximation of $f(x')$ and $A_{\mathbb{H}}(x)$ is evaluated in finite precision, it is impossible to reduce both ϵ_{approx} and ϵ_{round} to zero. The total error ϵ,

$$\epsilon = |A_{\mathbb{H}}(x) - f(x)| \leq \epsilon_{approx} + \epsilon_{rounding}$$

cannot be 0 for all inputs x. The approximation error can be reduced by increasing the degree of polynomial approximation and the rounding error can be reduced by increasing the precision of $\mathbb{H}$. Because increasing the degree of the polynomial or the precision of $\mathbb{H}$ results in performance slowdown, the math library developer has to make trade-offs

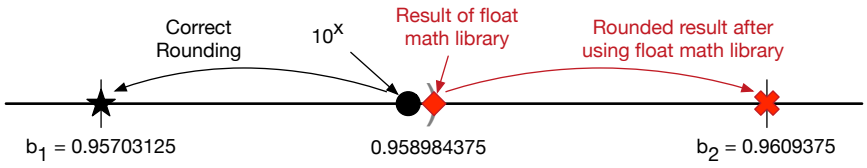


Figure 2.19: Horizontal axis represents a real number line. Given an input $x = -0.0181884765625$ that is exactly representable in `Bfloat16`, b_1 and b_2 represent the two closest `Bfloat16` values to the real value of $10^x = 0.958984357\ldots$. The correctly rounded `Bfloat16` value is b_1 (black star). If an approximation of 10^x has more than $\epsilon = 1.70\cdots \times 10^{-8}$ error, then the result may round to b_2, which is an incorrect result. Using a correctly rounded 32-bit FP math library to approximate 10^x for `Bfloat16` results in wrong results. When we use the 32-bit FP library to compute 10^x, it produces the value shown with red diamond, which then rounds to b_2 producing an incorrect result.

between reducing error or increasing the performance.

Identifying the error bound for correctly rounded approximation Unfortunately, there is no known general method to predict the error bound for ϵ such that the rounded result of the approximation (*i.e.*, $RN_{\mathbb{T},rm}(A_{\mathbb{H}}(x))$) is equal to the rounded result of $f(x)$ (*i.e.*, $RN_{\mathbb{T},rm}(f(x))$) for all inputs x. Since $f(x)$ can be arbitrarily close to the rounding boundary of two representable values, the error ϵ may need to be arbitrarily small to produce the correctly rounded result. To illustrate this problem, consider approximating $f(x) = 10^x$ for bfloat16 ($\mathbb{B}$) using rne mode with the input $x = -0.0181884765625$, which is exactly representable in $\mathbb{B}$. Figure 2.19 illustrates our example. The two closest representable $\mathbb{B} = \mathbb{F}_{16,8}$ values to the real value of $10^x = 0.958984357\ldots$ is b_1 and b_2 in Figure 2.19. Using the rne rounding mode, 10^x rounds to b_1 (highlighted with black start). Consider the rounding boundary between b_1 and b_2. For any value v between b_1 and b_2, if $v < 0.958984375$, then v rounds to b_1. Otherwise, v rounds to b_2. This infers that the approximation of 10^x must have an error less than

$$\epsilon < |0.958984357\cdots - 0.958984375| = 1.70239\cdots \times 10^{-8}$$

Otherwise, $10^x + \epsilon$ will round to b_2, producing an incorrectly rounded result. Thus, the approximation function $A_{\mathbb{H}}(x)$ must produce a value that has a relative error less than $1.77\cdots \times 10^{-8}$, which requires roughly $\lceil -log_2(1.77\cdots \times 10^{-8}) \rceil = 26$ precision bits to determine the correctly rounded result. 26 precision bits are more than $3\times$ the precision of bfloat16 (which has 8 precision bits).

This problem is widely known as the *table-maker's dilemma* [75], which states that there does not exist a general method to predict the amount of precision (*i.e.*, $\mathbb{H}$) required to compute the correctly rounded result in $\mathbb{T}$ for all inputs. The only way of determining the necessary precision is to iteratively compute the real result of $f(x)$ for all inputs and manually analyzing the error bound [85].

Using existing math library. An alternative method to create a math library for a representation $\mathbb{T}$ is to use an existing math library for $\mathbb{T}'$ where $\mathbb{T} \subseteq \mathbb{T}'$. We can convert the input x to $x' \in \mathbb{T}'$, use the math library for $\mathbb{T}'$, and round the result back to the target representation $\mathbb{T}$. This strategy is especially appealing if a correct math library for $\mathbb{T}'$ exists and $\mathbb{T}'$ has significantly more precision than $\mathbb{T}$. If this strategy is viable, we can even employ the same strategy to create math libraries for different representations $\mathbb{T}_1, \mathbb{T}_2, \mathbb{T}_3, \cdots \subseteq \mathbb{T}'$, thus opening the possibility to use a math library for $\mathbb{T}'$ as a generic math library.

However, using a correctly rounded math library designed for $\mathbb{T}'$ does not guarantee a correctly rounded result for $\mathbb{T}$ because of double rounding error. Our example with Figure 2.19 illustrates this problem. If we use a math library for 32-bit float ($\mathbb{F}$) that produces correctly rounded results with rne mode to approximate 10^x, then it produces the value $y' = 0.958984375$ (represented with red diamond in 2.19). In the context of the float math library, y' is the correctly rounded result, *i.e.*, $y' = RN_{\mathbb{F},rne}(10^x)$. However, when we round y' to bfloat16 because y' cannot be exactly represented in bfloat16, then y' rounds to $RN_{B,rne}(y') = b_2$, which is not the correctly rounded result. Therefore,

$$(RN_{\mathbb{B},rne}(RN_{\mathbb{F}}, rne)(10^x)) \neq (b_1 = RN_{\mathbb{B},rne}(10^x))$$

Due to double rounding error, using an existing high precision math library to create a lower precision math library does not guarantee to produce the correctly rounded result for all inputs, unless the math library is specifically created to be a generic math library.

Summary. The FP representation is a fixed-precision representation that is the most widely used to approximate real numbers. The Posit representation is a tapered-precision representation that has been gaining excitement as a possible alternative of FP for its wide dynamic range and high precision near 1. Real values that cannot be exactly represented in the FP or the posit representation can be rounded to the FP or posit value with one of the five rounding modes using the rounding components (s, v^{-}, rb, $sticky$). Any computation in finite precision can experience numerical errors caused by overflow, underflow, saturation, double rounding, and cancellation error.

In the past, math libraries were generated using the minimax approach that generates polynomials that approximate the real value of $f(x)$. To generate efficient polynomials, range reductions are used to reduce the entire input domain into a small domain. Generating a correctly rounded math library by mathematically approximating $f(x)$ is challenging because the error of the approximation may have to be arbitrarily small (*i.e.* $\epsilon = |A_{\mathbb{R}}(x) - f(x)|$). Evaluating the approximation function with finite precision arithmetic ($A_{\mathbb{H}}(x)$) incurs evaluation error, also makes the problem harder. Additionally, approximating an elementary function for $\mathbb{T}$ using a higher precision math library does not guarantee a correctly rounded result due to double rounding error. Thus, either we have to create a separate math library for each desired representation $\mathbb{T}_i$, or create a generic math library designed specifically to generate correctly rounded results for various representations.

CHAPTER 3
THE RLIBM APPROACH FOR CORRECTLY ROUNDED POLYNOMIAL APPROXIMATIONS

In this chapter, we describe our approach to generate polynomial approximations that produce the correctly rounded results of elementary functions $f(x)$. Existing methods generate polynomials that approximate the real value of the elementary function $f(x)$ and produce wrong results due to approximation and rounding error in the implementation. In contrast, we advocate for generating polynomials by approximating the correctly rounded value of $f(x)$ (*i.e.*, the real value $f(x)$ directly rounded to the target representation). The novelty of our approach is that it provides more freedom in generating efficient polynomials that produce correctly rounded results. Then, we frame the problem of generating the polynomial that produces the correctly rounded result as a linear programming problem. The approach and intuition presented in this chapter is the foundation for the remaining chapters when generating polynomials with range reduction (Chapter 4), generating piecewise polynomials (Chapter 5), and generating polynomial approximations for multiple representations and rounding modes (Chapter 6).

3.1 Approximating The Correctly Rounded Result

A correctly rounded result of an elementary function $f(x)$ is defined as the value produced by computing the real value of $f(x)$ and rounding it to the target representation. While there has been a large body of work to create accurate approximations of elementary functions [8, 15, 17, 20, 22, 30, 46, 56, 60, 71, 80, 97, 103, 119, 128, 135, 135], developing elementary functions that produce correctly rounded results for all inputs is still a challenging problem. There are a few correctly rounded math library for the commonly used float and double datatype [29, 65, 101, 146], but widely used math libraries [51, 67] do not produce correctly

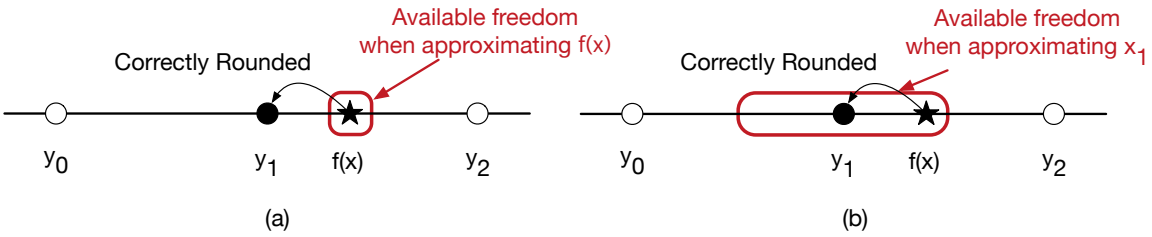


Figure 3.1: Illustration of the amount of freedom available (red box) in generating polynomials that produce correctly rounded results when (a) approximating the real value of $f(x)$ and (b) approximating the correctly rounded result itself. We show the real value of $f(x)$ with a star. The correctly rounded result of $f(x)$ is shown with a filled circle.

rounded results for all inputs.

Prior approaches generate polynomials that minimize the maximum error among all input points compared to the real value of the elementary function $f(x)$, known as the minimax approach. Remez algorithm [114] is a procedure to identify such minimax polynomials. The primary challenge in generating a correctly rounded polynomial approximation using the minimax approach is that the error of the polynomial approximation (*i.e.*, $|P(x) - f(x)|$) may need to be arbitrarily small. When the real value of $f(x)$ is arbitrarily close to the rounding boundary of two adjacent values in the target representation, even a small amount of error can produce the wrong result. We illustrate this problem with Figure 3.1(a), where we show the task of approximating $f(x)$ (star) and producing the correctly rounded result in our target representation using the rne rounding mode, where the correctly rounded result is y_1 (filled circle). Because $f(x)$ lies close to the midpoint between y_1 and y_2, the two adjacent values in our target representation, there is only a small amount of freedom in generating a polynomial that produces the correctly rounded result (red box). This challenge stems from the fundamental disconnect between the goal of math libraries and the purpose of the minimax approach. The goal of math libraries is to produce the correctly rounded result while the goal of the minimax approach is to generate a polynomial that approximates the exact value of $f(x)$ in real number.

Key idea #1: Approximate the correctly rounded result. Our key idea is to generate a polynomial that *approximates the correctly rounded result of* $f(x)$. We illustrate the advantage of our approach with Figure 3.1(b). By approximating the correctly rounded result y_1, our polynomial approximation can produce any value that will round to y_1 when rounded to our target representation (red box). There is a range of values $[l, h]$ in the vicinity of y_1 where all values in the interval rounds to the correctly rounded result y_1. We call this range of values the rounding interval. As long as we generate a polynomial that produces a value in $[l, h]$, the result of the polynomial rounds to the correctly rounded result y_1. This insight provides much more freedom to generate a correctly rounded polynomial approximation. Our key idea is inspired by the Minefield method [56]. Our contribution lies in developing techniques to systematically and automatically generate a polynomial that produces the correctly rounded results, which we describe below.

Key idea #2: Generating the polynomial as a linear programming (LP) problem. Each input x and the corresponding rounding interval $[l, h]$ defines a constraint on the coefficients of the polynomial $P(x)$ that we wish to generate. The result of the polynomial $P(x)$ must be a value between l and h to produce the correctly rounded results of $f(x)$ (*i.e.*, y_1) for a given input x. Specifically, the constraint pair $(x, [l, h])$ is a linear constraint on the coefficients of the polynomial $P(x)$:

$$l \leq c_0 + c_1 x + c_2 x^2 + \cdots \leq h$$

Abstractly, a polynomial with the coefficients that satisfy the above constraint will produce the correctly rounded result of $f(x)$ for the input x. Now our problem is to generate a polynomial that satisfies all linear constraints specified by the constraint pair $(x, [l, h])$ for each input x in the target representation. Hence, *we frame the problem of generating a polynomial that produces the correctly rounded result as a linear programming (LP) problem.* We construct a system of linear inequalities that solves for the coefficients of the polynomial that satisfy all linear constraints $(x, [l, h])$ and use an LP solver to solve for the

coefficients. The polynomial constructed with the resulting coefficients, when evaluated in real numbers, is guaranteed to produce the correctly rounded result of $f(x)$.

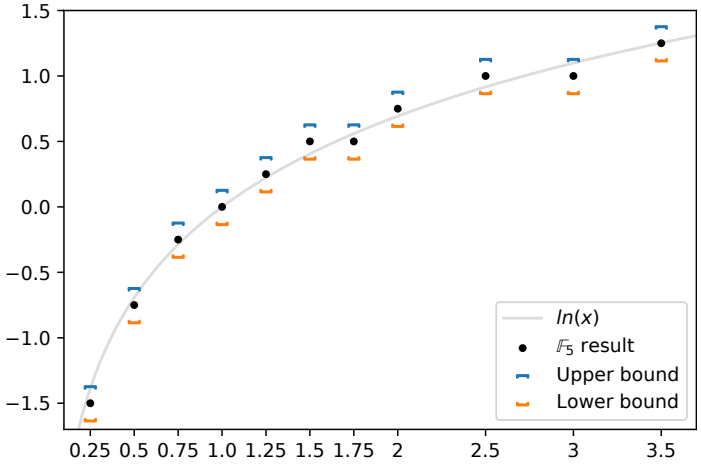
Key idea #3: Search and refine technique. Evaluating the resulting polynomial in a finite precision representation, however, does not guarantee to produce the correctly rounded result. Thus, we propose a *search-and-refine* technique. Initially, we generate a candidate polynomial that satisfies all constraints $(x, [l, h])$ when evaluated in real numbers. If evaluating the polynomial in a chosen finite precision representation does not satisfy any constraint, then we iteratively restrict the interval $[l, h]$ to a smaller interval and solve for the polynomial. We repeat the process until we identify a polynomial that produces the correctly rounded result when evaluated in the finite precision representation, or the LP solver determines that it is infeasible to find such a polynomial.

3.2 Illustration Of Our Approach

We provide an end-to-end example of generating a polynomial approximation of $ln(x)$ that produces the correctly rounded results for all inputs in a 5-bit FP representation with 2 exponent bits (*i.e.*, FP5 $= \mathbb{F}_{5,2}$) using the rne mode. We show this example only to illustrate our approach and insight. In practice, creating a look-up table with pre-computed results is more beneficial for a 5-bit FP. The steps of our approach are shown in Figure 3.2 and the resulting polynomial is shown in Figure 3.4.

The $ln(x)$ function is defined over the input domain $(0, \infty)$. There are 11 values ranging from 0.25 to 3.5 in FP5 within the input domain. The remaining $2^5 - 11 = 21$ values are special case inputs where the result is not a real number:

$$\textit{The correctly rounded result of } ln(x) = \begin{cases} -\infty & \text{if } x = 0 \\ \infty & \text{if } x = \infty \\ NaN & \text{if } x < 0 \text{ or } x = NaN \end{cases}$$



(a)

$$
\begin{array}{lll}
-1.625 & \leq P(0.25) & \leq -1.375 \\
-0.875 & < P(0.5) & < -0.625 \\
-0.375 & < P(0.75) & < -0.125 \\
-0.125 & \leq P(1.0) & \leq 0.125 \\
0.125 & < P(1.25) & < 0.375 \\
0.375 & \leq P(1.5) & \leq 0.625 \\
0.375 & \leq P(1.75) & \leq 0.625 \\
0.625 & < P(2.0) & < 0.875 \\
0.875 & \leq P(2.5) & \leq 1.125 \\
0.875 & \leq P(3.0) & \leq 1.125 \\
1.125 & < P(3.5) & < 1.375
\end{array}
$$

(b)

$$
\begin{bmatrix} -1.625 \\ -0.874\ldots \\ -0.374\ldots \\ -0.125 \\ 0.125\ldots \\ 0.375 \\ 0.375 \\ 0.625\ldots \\ 0.875 \\ 0.875 \\ 1.125\ldots \end{bmatrix}
\leq
\begin{bmatrix} 1 & 0.25 & 0.25^2 & 0.25^3 \\ 1 & 0.5 & 0.5^2 & 0.5^3 \\ 1 & 0.75 & 0.75^2 & 0.75^3 \\ 1 & 1.0 & 1.0^2 & 1.0^3 \\ 1 & 1.25 & 1.25^2 & 1.25^3 \\ 1 & 1.5 & 1.5^2 & 1.5^3 \\ 1 & 1.75 & 1.75^2 & 1.75^3 \\ 1 & 2.0 & 2.0^2 & 2.0^3 \\ 1 & 2.5 & 2.5^2 & 2.5^3 \\ 1 & 3.0 & 3.0^2 & 3.0^3 \\ 1 & 3.5 & 3.5^2 & 3.5^2 \end{bmatrix}
\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}
\leq
\begin{bmatrix} -1.375 \\ -0.625\ldots \\ -0.125\ldots \\ 0.125 \\ 0.374\ldots \\ 0.625 \\ 0.625 \\ 0.874\ldots \\ 1.125 \\ 1.125 \\ 1.374\ldots \end{bmatrix}
$$

(c)

Figure 3.2: Our approach for $ln(x)$ with `FP5`. (a) For each input x in `FP5`, we accurately compute the correctly rounded result (black circle) and identify intervals around the result so that all values round to it. (b) The set of constraints that must be satisfied by the polynomial for the reduced input. (c) LP formulation for generating a polynomial of degree three that satisfies the constraints in (b). The resulting polynomial is shown in Figure 3.4.

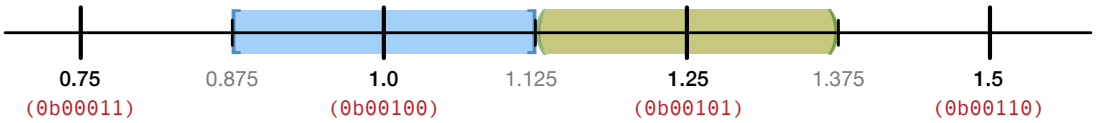


Figure 3.3: This figure shows the real number line and several adjacent FP5 values, 0.75, 1.0, 1.25, and 1.5. Any real value in the blue interval $[0.875, 1.125]$, rounds to 1.0 in FP5 with RNE mode. Similarly, any value in the green interval $(1.125, 1.375)$ rounds to 1.25.

These special cases can be implemented with a simple check followed by returning the correct result.

3.2.1 Computing the Correctly Rounded Result.

Once the special case inputs are filtered, there are 11 FP5 inputs in the input domain $[0.25, 3.5]$. The inputs are shown on the x-axis in Figure 3.2(a). Our first step is to identify the correctly rounded result of $ln(x)$ for these 11 inputs. We use an oracle (*i.e.*, MPFR math library [46]) to compute the real value of $ln(x)$ and round the result to FP5. Figure 3.2(a) shows the correctly rounded result for each of the inputs with black dots.

3.2.2 Identifying the Rounding Interval

The polynomial approximation of $ln(x)$ is evaluated in a higher precision representation compared to FP5. In our example, we will use the double representation to evaluate the polynomial. Our next step is to identify a range of values that rounds to the correctly rounded result when rounded to FP5 with rne mode. We call this range of values the rounding interval. If our polynomial produces a value within the rounding interval, then the result will round to the correctly rounded value, $RN_{FP5,rne}(ln(x))$. Figure 3.2(a) shows the rounding interval for each result using the blue (upper bound) and orange (lower bound) square bracket.

Suppose that we want to compute the rounding interval for 1.0, which is the correctly rounded result of $ln(2.5)$. To identify the lower bound l of the rounding interval, we first identify the preceding value of 1.0 in FP5, which is 0.75. Then, we identify l between 0.75

and 1.0 such that all values $v \geq l$ rounds to 1.0. More formally,

$$l = \min\{v \mid 0.75 \leq v \leq 1.0 \wedge RN_{FP5,RNE}(v) = 1.0\}$$

In our case, $l = 0.875$. Similarly, to identify the upper bound h in the rounding interval, we identify the succeeding value of 1.0 in FP5, which is 1.25. Then, we identify the upper bound h between 1.0 and 1.25 such that all values smaller than or equal to h rounds to 1.0:

$$u = \max\{v \mid 1.0 \leq v1.25 \wedge RN_{FP5,RNE}(v) = 1.0\}$$

In this case, $h = 1.125$. Hence, the rounding interval for 1.0 is $[l, h] = [0.875, 1.125]$. Figure 3.3 shows the rounding interval of 1.0 with a blue box and the rounding interval of 1.25 with a yellow box for FP5. The bit-string of 1.25 in FP5 is odd when interpreted as an unsigned integer. From the definition of rne mode, 1.125 rounds to 1.0 and 1.375 rounds to 1.5. All real values larger than 1.125 and smaller than 1.375 rounds to 1.25. Because all of our computations are evaluated with double, we identify the rounding interval of 1.25 in double. Hence, the rounding interval of 1.25 in double is $[l^+, h^-]$, where l^+ is the smallest double value larger than 1.125 and h^- is the largest double value smaller than 1.375.

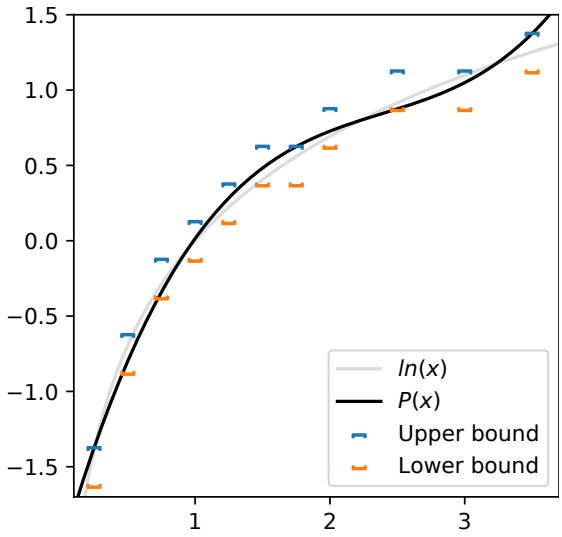
3.2.3 Generating a Polynomial Approximation

The rounding intervals specify constraints on the output of the polynomial $P(x)$ for each input x, where $P(x)$ is our approximation function. Based on the intervals we computed, if $P(x)$ produces a value within the rounding interval, the result is guaranteed to round to the correctly rounded result in FP5. Figure 3.2(b) shows the constraints for $P(x)$ for each input. Thus, our goal is to generate $P(x)$ that satisfies all of the constraints.

To generate a polynomial $P(x)$ of degree 3, we encode the problem as an LP problem that solves for the coefficients of $P(x)$. We create a system of linear inequalities that specify the constraints of $P(x)$ (Figure 3.2(c)) and use an LP solver to solve for the coefficients of $P(x)$ that satisfies all the constraints. If the LP solver produces a solution, the resulting polynomial $P(x)$ created using the coefficients (Figure 3.4(a)) satisfies all the constraints

$$P(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3$$
$$c_0 = -2.084401709401708657765\ldots$$
$$c_1 = 3.1251526251526207111908\ldots$$
$$c_2 = -1.191697191697188351611\ldots$$
$$c_3 = 0.66056166056165438460\ldots$$

(a)



(b)

Figure 3.4: Continuation from Figure 3.2, where we approximate $ln(x)$ for $\mathbb{F}_5$. (a) The coefficients generated by the LP solver for the polynomial. (b) Generated polynomial satisfies the constraints.

(shown with a black line in Figure 3.4(b)). Finally, we verify that $P(x)$ indeed produces the correctly rounded result by evaluating the polynomial and comparing the result against the correctly rounded result for all inputs.

3.3 The RLIBM Approach To Generate Correctly Rounded Polynomial Approximation

We formally define the correctly rounded result of $f(x)$. Given an elementary function $f(x)$, a target representation $\mathbb{T}$, a rounding mode rm, and an input domain $X = [a, b] \subseteq \mathbb{T}$, our goal is to generate an approximation function of $f(x)$ that produces the correctly rounded result for all inputs in X.

Definition 3.1. A value v is the correctly rounded result of an elementary function $f(x)$ for the representation $\mathbb{T}$ and rounding mode rm, if $v = RN_{\mathbb{T},rm}(f(x))$.

Based on Definition 3.1, the correctly rounded result of $f(x)$ is unique for the chosen representation and rounding mode for each input x. More formally, there does not exist two distinct values $v_1 \neq v_2$ where both v_1 and v_2 are correctly rounded results of $f(x)$ for

Algorithm 3.1: Our approach to generate a polynomial approximation $P_{\mathbb{H}}(x)$ of degree d that produces the correctly rounded result of f in the target representation $\mathbb{T}$, for all inputs $X \subseteq \mathbb{T}$. On successfully finding a polynomial, it returns (true, $P_{\mathbb{H}}$). Otherwise, it returns (false, DNE) where DNE means that the polynomial Does-Not-Exist. `CalcRndIntervals` and `GeneratePoly` are shown in Algorithm 3.2 and Algorithm 3.4, respectively.

```
1 Function CorrectlyRoundedPoly(f, T, H, X, d):
2     L ← CalcRndIntervals(f, T, H, X)
3     if L = ∅ then return (false, DNE)
4     S, P_H ← GeneratePoly(L, d)
5     if S = true then return (true, P_H)
6     else return (false, DNE)
```

an input x.

We propose an approach to produce a polynomial approximation that produces the correctly rounded result of $f(x)$ when rounded to the target representation $\mathbb{T}$. The polynomial is evaluated in a higher precision representation $\mathbb{H}$ compared to $\mathbb{T}$ to evaluate the polynomial accurately. We use $P(x)$ to represent the polynomial generated with our approach that is evaluated in real numbers. We use $P_{\mathbb{H}}(x)$ to represent $P(x)$ evaluated in $\mathbb{H}$. The result of $P_{\mathbb{H}}(x)$ is rounded to $\mathbb{T}$ to produce the final result. Our goal is to ensure that $P_{\mathbb{H}}(x)$ produces the correctly rounded result of $f(x)$ in $\mathbb{T}$.

3.3.1 High-Level Overview of The RLIBM Approach

Our approach to generate $P_{\mathbb{H}}(x)$ is illustrated in Algorithm 3.1. Our approach assumes the existence of an oracle that generates the real result of $f(x)$. The oracle is used to compute the correctly rounded result of $f(x)$ in $\mathbb{T}$. We also assume that the special case inputs are already filtered out from the input domain X. The special cases are identified using mathematical identities of $f(x)$. The target representation $\mathbb{T}$, the rounding mode rm to round the result to $\mathbb{T}$, the higher precision representation $\mathbb{H}$ to evaluate the approximation function, and the degree d of the polynomial are inputs provided by the math library developer. Since our approach applies for any rounding mode rm, the remainder of the chapter assumes that $rm = rne$ and omits the discussion of specific rounding mode. Hence, we

Algorithm 3.2: For each input $x \in X$, `CalcRndIntervals` identifies the interval $I = [l, h]$ where all values in I round to the correctly rounded result. The `GetRndInterval` function takes the correctly rounded result y and returns the interval $I \subseteq \mathbb{H}$ where all values in I round to y. `GetPrecValue`(y, $\mathbb{T}$) returns the value preceeding y in $\mathbb{T}$. `GetSuccValue`(y, $\mathbb{T}$) returns the value succeeding y in $\mathbb{T}$.

```
1  Function CalcRndIntervals(f, 𝕋, ℍ, X):
2      L ← ∅
3      foreach x ∈ X do
4          y ← RN_{𝕋,rne}(f(x))
5          I ← GetRndInterval(y, 𝕋, ℍ)
6          L ← L ∪ {(x, I)}
7      end
8      return L
9  Function GetRndInterval(y, 𝕋, ℍ):
10     y⁻ ← GetPrecVal(y, 𝕋)
11     l ← min{v ∈ ℍ | v ∈ (y⁻, y] and RN_𝕋(v) = y}
12     y⁺ ← GetSuccVal(y, 𝕋)
13     h ← max{v ∈ ℍ | v ∈ [y, y⁺) and RN_𝕋(v) = y}
14     return [l, h]
```

use the term $RN_{\mathbb{T}}(x)$ to describe rounding a real value x to the representation $\mathbb{T}$ using a preferred rounding mode (in our case, rne).

Our algorithm consists of two main steps. First, we compute the correctly rounded result of $f(x)$, $y = RN_{\mathbb{T}}(f(x))$, for each input x using the oracle. Then, we compute an interval $I = [l, h] \subseteq \mathbb{H}$ where all values $v \in I$ rounds to y. The interval I describes all the values that that our polynomial $P(x)$ can produce such that $P(x)$ rounds to the correctly rounded result. Thus, the pair (x, I) specifies the input-output constraint of $P(x)$. The function `CalcRndIntervals` in Algorithm 3.1 returns a list L that contains the constraints (x, I) for all inputs $x \in X$. Second, we generate the polynomial $P(x)$ of degree d that satisfies all constraints $(x, I) \in L$ using linear programming. We make sure that evaluating the polynomial in $\mathbb{H}$ (*i.e.*, $P_{\mathbb{H}}(x)$) also satisfies all the constraints in L. The function `GeneratePoly` returns the polynomial $P_{\mathbb{H}}$, if our approach identifies a polynomial of degree d that satisfies all constraints. Otherwise, it returns a message indicating that it was not successful in identifying such a polynomial. We now describe these two steps in more detail.

3.3.2 Computing the Rounding Intervals

Given an elementary function $f(x)$, the RLIBM approach identifies the values that $P_{\mathbb{H}}(x)$ must produce so that the rounded value of $P_{\mathbb{H}}(x)$ is equivalent to the correctly rounded result of $f(x)$ for each input. We call this interval the rounding interval. Because our polynomial is evaluated in $\mathbb{H}$, we look for the interval I in $\mathbb{H}$. For each input x, if our polynomial evaluated in $\mathbb{H}$ produces a value in the corresponding rounding interval I, then the result is guaranteed to round to the correctly rounded result y. Thus, the pair (x, I) for each input x defines constraints on the output of $P_{\mathbb{H}}(x)$ such that rounding the result produces the correctly rounded result.

Algorithm 3.2 presents our approach to compute the constraints (x, I). For each input x in the input domain X, we compute $f(x)$ with real numbers using an oracle and round the result to the target precision $\mathbb{T}$ to produce the correctly rounded result y (line 4). Next, we compute the rounding interval of y. The rounding interval can be computed as follows. First, we identify y^-, the preceding value of y in $\mathbb{T}$ (line 9). Next, we find the minimum value $l \in \mathbb{H}$ between y^- and y that rounds to y (line 10). The value l defines the lower bound of the rounding interval. Similarly for the upper bound, we identify y^+, the succeeding value of y in $\mathbb{T}$ (line 11). Then, we find the maximum value $h \in \mathbb{H}$ between y and y^+ that rounds to y (line 12). The value h is the upper bound of the rounding interval. Thus, the interval $[l, h]$ is the rounding interval of y (line 3) and the pair $(x, [l, h])$ specifies the constraint on the output of $A_{\mathbb{H}}(x)$ to produce the correctly rounded result for the input x. We generate such constraints for all inputs in the input domain X and produce a list of these constraints L (line 6).

Efficiently computing the rounding interval. The function `GetRndInterval` in Algorithm 3.2 presents a naive method of computing the rounding interval. Abstract, we identify y^-, the value preceding y in $\mathbb{H}$. Then, we find the smallest value l between y^- and y that rounds to y in $\mathbb{T}$. Similarly, we identify y^+, the value succeeding y in $\mathbb{H}$. Next, we

Algorithm 3.3: Given a 32-bit float ($\mathbb{F}$) value v, `GetRndIntervalFP32` computes the rounding interval of v with RNE rounding mode. The final rounding interval $[l, h]$ is in the context of the 64-bit double type ($\mathbb{D}$).

```
1  Function GetRndIntervalFP32(v):
2      v⁻ ← GetPrecVal(v, 𝔽)
3      if v⁻ = −∞ then v⁻ = −2^128
4      l ← (v⁻ + v)/2
5      v⁺ ← GetSuccVal(v, 𝔽)
6      if v⁺ = ∞ then v⁺ = 2^128
7      h ← (v + v⁺)/2
8      if bit-string of v is odd when interpreted as unsigned integer then
9          l ← GetSuccVal(l, 𝔻)
10         h ← GetPrecVal(h, 𝔻)
11     end
12     return [l, h]
```

find the largest value h between y and y^+ that rounds to y in $\mathbb{T}$. The interval $[l, h]$ is the rounding interval of y. This naive approach may be an expensive process if $\mathbb{H}$ has substantially larger precision compared to $\mathbb{T}$. For example, if $\mathbb{T}$ is the IEEE-754 standard 32-bit float type $\mathbb{F} = \mathbb{F}_{32,8}$ and $\mathbb{H}$ is the IEEE-754 standard 64-bit double type $\mathbb{D} = \mathbb{F}_{64,11}$, then there are at least 2^{29} double values between two adjacent float values.

Identifying the rounding intervals can be performed efficiently by identifying the rounding boundaries of values in $\mathbb{T}$ using the properties of $\mathbb{H}$, $\mathbb{T}$, and the rounding mode rm. To illustrate this idea, we present an algorithm to efficiently compute the rounding interval in double $[l, h] \in \mathbb{D}$ for the float value $v \in \mathbb{F}$ with the rounding mode rne (Algorithm 3.3). This algorithm can be adapted to compute the rounding interval of various $\mathbb{H}$, $\mathbb{T}$, and rm.

The efficient technique identifies v^-, the preceding float value of v and computes the rounding boundary l between v^- and v (lines 2-4). Among the values between v^- and v, all values smaller than l should round to v^- and all values larger than l should round to v. The rounding boundary l can be computed efficiently with $l = \frac{v^- + v}{2}$, because the rne mode in FP rounds to the nearest float value. The rounding boundary is the arithmetic mean between v^- and v. The value l is exactly representable in double. Similarly, we identify v^+, the succeeding float value of v and round boundary h between v and v^+ using

the formula $h = \frac{v+v^+}{2}$ (lines 5-7). The conditional statements in lines 3 and 6 make sure that we compute the correct rounding boundary when v is the smallest negative value (line 3) or the largest positive value (line 6) where the adjacent value of v is $\pm\infty$.

Next, based on the rounding boundary, we determine whether the rounding boundary l and h themselves should be included in the rounding interval. Because l and h is the exact mid-point between v and its adjacent float values (*i.e.* v^- and v^+, respectively), we have to decide on the *tie goes to even* part of rne, based on the bit-string of v. If the bit-string of v is even when interpreted as an unsigned integer, the rounding interval of v is $[l, h]$ (line 11). Otherwise, the rounding interval is (l, h). In this case, we identify the succeeding value of l in double and the preceding value of h in double (lines 8-10) and return the interval between these two values.

3.3.3 Generating the Polynomial With an LP Formulation

Each constraint $(x, [l, h]) \in L$ specifies that $P_{\mathbb{H}}(x)$ must satisfy the property $l \leq P_{\mathbb{H}}(x) \leq h$. This constraint ensures that $P_{\mathbb{H}}(x)$ produces the correctly rounded result when rounded to $\mathbb{T}$. Each constraint $(x, [l, h])$ on $P(x)$ can be expressed in the form:

$$l \leq c_0 + c_1 x + c_2 x^2 + \cdots + c_d x^d \leq h$$

In the above expression, the values x, l, and h are constants and we must identify the coefficients $c_0, c_1, \ldots, c_d$ that satisfy the constraints. Thus, in the perspective of finding the coefficients, the constraint $(x, [l, h])$ is a linear constraint. We can express all constraints in $(x_i, [l_i, h_i]) \in L$ with a single system of linear inequalities as shown below,

$$\begin{bmatrix} l_1 \\ l_2 \\ \vdots \\ l_{|L|} \end{bmatrix} \leq \begin{bmatrix} 1 & x_1 & x_1^2 \ldots & x_1^d \\ 1 & x_2 & x_2^2 \ldots & x_2^d \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{|L|} & \ldots & x_{|L|}^d \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_d \end{bmatrix} \leq \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_{|L|} \end{bmatrix}$$

This system of linear inequalities is a linear programming (LP) problem. We can use an LP solver to solve for the coefficients that satisfy all constraints. Since LP solvers produce solutions in real numbers, the polynomial $P(x)$ created with the coefficients when evaluated with real numbers satisfies all constraints in L. Thus, the result of $P(x)$ is guaranteed to round to the correctly rounded result for all inputs.

Accounting for numerical error when evaluating $P(x)$ ***in*** $\mathbb{H}$**.** The numerical errors that occur when evaluating the polynomial $P(x)$, generated with our approach, in $\mathbb{H}$ can cause the result of $P_{\mathbb{H}}(x)$ to not satisfy some constraints in L. Suppose that we generate a polynomial $P(x)$ that satisfies a constraint $(x, [l, h]) \in L$, such that evaluating the polynomial in real numbers produces the value h (*i.e.*, $P_{\mathbb{R}}(x) = h$). However, if we evaluate the polynomial in $\mathbb{H}$, it can be the case that $P_{\mathbb{H}}(x)$ to lie outside of the rounding interval (*i.e.*, $P_{\mathbb{H}}(x) > h$), due to the numerical error in finite precision arithmetic. Hence, when we generate $P(x)$ that satisfies all constraints $(x, [l, h])$, we must account for the numerical errors to ensure that $P_{\mathbb{H}}(x)$ evaluates to a value in the rounding interval.

To address this challenge, we propose a *search-and-refine* technique. We first use the LP solver to solve for the coefficients of $P(x)$ by encoding the constraints $(x, [l, h])$ and generate a candidate polynomial that produces a value in the rounding interval when evaluated with real numbers. Next, we evaluate the polynomial in $\mathbb{H}$ for each input x and check whether $P_{\mathbb{H}}(x)$ produces a value in the rounding interval. If $P_{\mathbb{H}}(x)$ does not satisfy a constraint $(x, [l, h]) \in L$, then we refine the rounding interval $[l, h]$ to a smaller interval $[l', h'] \subseteq [l, h]$. If $P_{\mathbb{H}}(x) < l$, then we increase the lower bound l and if $P_{\mathbb{H}}(x) > h$, we decrease the upper bound h. We use the LP solver again to generate the coefficients of $P_{\mathbb{H}}(x)$ that satisfies the refined constraint $(x, [l', h'])$ instead of $(x, [l, h])$. This process is repeated until either $P_{\mathbb{H}}(x)$ satisfies all constraints in L or the LP solver returns with no solution.

Algorithm 3.4 provides the algorithm for generating the coefficients of the polynomial

Algorithm 3.4: `GeneratePoly` generates a polynomial $P_{\mathbb{H}}(x)$ of degree d that satisfies all constraints in L when evaluated in $\mathbb{H}$. If it cannot generate such a polynomial, then it returns *false*. `LPSolve` solves for the real number coefficients of a polynomial $P_{\mathbb{R}}(x)$ using an LP solver where $P_{\mathbb{R}}(x)$ satisfies all constraints in L when evaluated in real number. `CreatePolynomial` creates $P_{\mathbb{H}}(x)$ that evaluates the polynomial $P_{\mathbb{R}}(x)$ in $\mathbb{H}$. The `Verify` function checks whether the generated polynomial $P_{\mathbb{H}}(x)$ satisfies all constraints in L when evaluated in $\mathbb{H}$ and refines the interval for each constraint that $P_{\mathbb{H}}(x)$ does not satisfy.

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

using the LP solver. L' keeps track of the refined constraints that we compute during the search-and-refine process. Initially, we copy the constraints L into L' (line 2). We use the list L' to generate the polynomial and L to verify that the generated polynomial satisfies the input-and-rounding-interval constraints. If the polynomial generated using the LP solver does not satisfy L, then we restrict the intervals in L'.

We encode the constraints in L' into a system of linear inequalities and use an LP solver

to solve for the coefficients of the polynomial of degree d that satisfies the constraints in L' when evaluated with real numbers (line 4). If the LP solver determines that there is no solution (*i.e.*, there exists no polynomial that satisfies the constraints in L'), then our algorithm concludes that it is not possible to generate a polynomial that produces correctly rounded results and terminates (line 5). In this case, the math library developers can increase the degree of the polynomial d or the precision of $\mathbb{H}$ and restart the whole process. Otherwise, we use the solved coefficients and create $P_{\mathbb{H}}(x)$ that evaluates $P(x)$ in $\mathbb{H}$ by rounding all coefficients to $\mathbb{H}$ and performing all operations in $\mathbb{H}$ (line 6).

The polynomial $P_{\mathbb{H}}(x)$ is a candidate solution for $A_{\mathbb{H}}(x)$. We verify that $P_{\mathbb{H}}(x)$ indeed satisfies all constraints in L (line 7) by evaluating $P_{\mathbb{H}}(x)$ and checking the result is within the rounding interval $[l, h]$ for each input x (line 11 - 21). If $P_{\mathbb{H}}(x)$ satisfies all constraints in L, then our algorithm concludes with success and returns $P_{\mathbb{H}}(x)$ (line 8). However, if there exists a constraint $(x, [l, h])$ in L, where $P_{\mathbb{H}}(x)$ evaluates to a value outside of the rounding interval $[l, h]$, then we restrict the refined constraint $(x, [l, h'])$ in L' that corresponds to the same input x. If $P_{\mathbb{H}}(x)$ is smaller than the lower bound l of the rounding interval, then we restrict the lower bound l' of the refined interval in L' to the value succeeding l' in $\mathbb{H}$ (lines 12-16). This process forces the $P(x)$ that we generate in the next iteration to produce a value larger than l', increasing the likelihood of $P_{\mathbb{H}}(x)$ to produce a value larger than or equal to l. Likewise, if $P_{\mathbb{H}}(x)$ is larger than the upper bound l of the rounding interval in L, then we restrict the upper bound h' of the refined interval in L' to the value preceding h' in $\mathbb{H}$ (lines 17-21).

We repeatedly generate a new candidate polynomial with the refined constraints L' until the new polynomial satisfies all constraints in L or the LP solver determines that it is infeasible. If a refined constraint $(x, [l', h']) \in L'$ is restricted to the point where $l' > h'$ (or $[l', h'] = \emptyset$), then the LP solver will automatically determine that there is no feasible solution. When our algorithm successfully returns a polynomial, then $P_{\mathbb{H}}(x)$ is guaranteed to produce a value that rounds to the correctly rounded result of $f(x)$ in $\mathbb{T}$.

3.4 Summary

We propose a novel approach to generate polynomial approximations of elementary functions that produces the correctly rounded result for all inputs. To produce the correctly rounded result of $f(x)$, we make a case for approximating the correctly rounded result itself, not the real value of $f(x)$. With this insight, our approach generates polynomial approximations by identifying the maximum amount of freedom available to generate the correctly rounded result for each input. We identify the rounding interval for each input x where all values in the interval rounds to the correctly rounded result of $f(x)$. This interval defines the amount of freedom to generate the polynomial. The rounding interval for each input defines a linear constraint for the polynomial approximation to produce the correctly rounded results. Hence, we frame the problem of generating the polynomial as an LP problem and use the LP solver to generate the polynomial. When our approach successfully generates a polynomial, then the polynomial is guaranteed to produce the correctly rounded result of $f(x)$ for all inputs. Our approach can automatically produce such polynomials.

This key idea is the foundation for generating efficient and correct polynomial approximations, which we present in the subsequent chapters. Combining with range reduction and domain splitting can generate lower degree polynomials while still producing the correctly rounded results, thus increasing the performance (described in Section 4 and Section 5). An extension of the approach (described in Section 6) can generate a single polynomial approximation that produces the correctly rounded result for multiple representations and rounding modes. The math library created using our approach combined with the techniques presented in subsequent chapters produce correctly rounded results for all inputs and are faster than mainstream libraries.

CHAPTER 4
THE RLIBM APPROACH WITH RANGE REDUCTION

As discussed in the previous chapter, our insight is to approximate the correctly rounded result rather than the real value of $f(x)$. If the degree of the polynomial is sufficiently large, then a single polynomial can produce the correctly rounded result of $f(x)$ for all inputs. In practice, however, polynomials are often used with range reduction strategies to create efficient approximations. Range reductions significantly reduce the input domain of $f(x)$ into a much smaller domain. Polynomial approximations are efficient at approximating elementary functions in a smaller domain. Hence, a much lower degree polynomial approximation can be used to produce the correctly rounded result of $f(x)$ for all inputs.

In this chapter, we present our approach to generate polynomial approximations that produce the correctly rounded results of $f(x)$ when used with range reduction strategies. The main challenge lies in the fact that we now have to account for the numerical errors that arise in the range reduction and the output compensation functions. Our strategy is to infer the range of values that the polynomial should produce such that, when used with the range reduction strategy, it produces the correctly rounded results for all inputs. Additionally, we present the range reduction strategies we use to approximate several elementary functions. Some of these strategies are developed specifically with the goal of minimizing numerical errors.

4.1 Generating Polynomial Approximations With Range Reduction

Our approach that we discussed in Chapter 3 can generate polynomial approximations for any target representation as long as there is no limit on the degree of the polynomial and the representation $\mathbb{H}$ we use to evaluate the polynomial has sufficiently high precision. However, evaluating a high degree polynomial is not efficient especially if $\mathbb{H}$ is a representation

that is not natively supported in the hardware, thus having to use software simulation for all primitive operations. Our goal is not only to generate correctly rounded approximations of $f(x)$, but also an efficient approximation.

A widely adopted strategy to increase the performance of an approximation of elementary functions is to use range reduction along with polynomial approximation, as described in Chapter 2. In essence, range reduction reduces the domain of the function we must approximate to a smaller domain. As polynomials are much more efficient in approximating elementary functions in a small domain, this allows us to generate a lower degree polynomial, which can be evaluated efficiently. With a well designed range reduction, the performance benefit of using a lower degree polynomial far outweighs the additional overhead incurred from using range reduction. Hence, developing efficient range reduction strategies have been researched for many years [12, 25, 32, 40, 42, 48, 103, 117, 133, 134, 135, 141] and all mainstream math libraries use range reduction whenever possible.

Using a range reduction strategy, an approximation $A(x)$ of an elementary function $f(x)$ has three components. (1) A range reduction function that reduces an original input x from the entire input domain to a reduced input x' in a smaller domain, (2) a polynomial approximation function that uses the reduced input x' to approximate a function $g(x')$, and (3) an output compensation function that compensates the approximation of $g(x')$ using the reduced input x' to produce the approximation of $f(x)$ with the original input x. Depending on the range reduction strategy, the function $g(x')$ approximated by the polynomial approximation may not be the same as $f(x)$.

Challenges in generating polynomial approximations that work with range reductions. The primary challenge in generating a polynomial approximation that produces the correctly rounded result of $f(x)$ in the target representation $\mathbb{T}$, when used with range reduction strategy, is the fact that the polynomial must account for the numerical error that occur when evaluating the range reduction and the output compensation function. As range re-

duction, polynomial approximation, and output compensation functions are evaluated in a finite precision representation $\mathbb{H}$, it is impossible to eliminate numerical errors for an arbitrary range reduction strategy. Thus, naively generating polynomial approximations for $g(x')$ will not guarantee to produce correctly rounded results for all inputs. Prior work mathematically analyzes the error of approximation functions and verifies that elementary functions produce correctly rounded results [21, 29, 34, 38, 59, 62, 82, 83, 145].

Our approach. We propose a novel approach to generate polynomial approximations with range reduction strategies. Our key insight is to infer the list of the inputs and the range of values that our polynomial should produce. Our approach identifies these inputs and range of values using the following strategy. First, for each input x in $\mathbb{T}$, we compute the correctly rounded result of $f(x)$ in $\mathbb{T}$ and identify the rounding interval $[l, h]$. The list of inputs x and its corresponding rounding interval $[l, h]$ now constrains the output of the entire approximation $A(x)$ such that it produces the correctly rounded result of $f(x)$ in $\mathbb{T}$. Next, we infer the inputs x' and the range of values $[l', h']$ that our polynomial approximation should produce using the range reduction and the output compensation function based on x and $[l, h]$. We call x' the reduced input and $[l', h']$ the reduced interval. During this step, we also account for the numerical errors that occur with the range reduction and the output compensation function. The list of reduced input and interval pairs $(x', [l', h'])$ constrains the polynomial such that it produces the correctly rounded of $f(x)$ in $\mathbb{T}$ when used with range reduction strategy and evaluated in $\mathbb{H}$. With the list of these constraints, we frame the problem of generating the polynomial as LP problem and use LP solver to create the polynomial. Our approach is compatible with a wide variety of range reduction strategies, handling both univariate and multivariate output compensation functions. Additionally, we designed several range reduction strategies that minimize the amount of numerical error by eliminating cancellation errors.

Using our approach and range reduction strategies that we designed, we created a

prototype, RLIBM-16, containing several elementary functions for bfloat16 and posit16 representations. Our automatic approach in generating polynomial approximations has been pivotal in generating efficient polynomials. Our elementary functions are faster than mainstream math libraries repurposed for bfloat16 and posit16 while producing correctly rounded results for all inputs as detailed in Chapter 7. In contrast, mainstream math libraries do not produce correctly rounded bfloat16 results for all inputs. Our prototype is the first correctly rounded math library for bfloat16.

4.2 Illustration

We illustrate an end-to-end example of our approach for creating a correctly rounded approximation function for $ln(x)$ for the FP5 representation with RNE rounding mode using range reduction. Note that our goal is the same as the example from Chapter 3, but now with range reduction and polynomial approximation. There are two purposes in our example: (1) to show that range reduction allows us to generate a lower degree polynomial that produces correctly rounded results when used with the output compensation function and (2) to illustrate our approach in generating a polynomial that accounts for the numerical error while evaluating the output compensation function in a finite precision representation.

4.2.1 Range Reduction for $ln(x)$

The $ln(x)$ function is defined over the input domain $(0, \infty)$. There are 11 values ranging from 0.25 to 3.5 in $\mathbb{F}_{5,2}$ within $(0, \infty)$. The remaining 21 values are special case inputs. These special cases can be implemented with a simple check followed by returning the correct result. Our strategy for generating a polynomial approximation for $ln(x)$ is by approximating $ln(x)$ using $log_2(x)$. We perform range reduction and output compensation using two mathematical identities,

$$ln(x) = \frac{log_2(x)}{log_2(e)}, \qquad log_2(x \times y^z) = log_2(x) + z log_2(y)$$

We decompose the input x into $x = x' \times 2^m$ where x' is the significand of the input ranging from $x' \in [1, 2)$ and $m \in \mathbb{Z}$ is the integer exponent of x. Then, $ln(x)$ can be computed using,

$$ln(x' \times 2^m) = \frac{log_2(x' \times 2^m)}{log_2(e)} = \frac{log_2(x') + mlog_2(2)}{log_2(e)} = \frac{log_2(x') + m}{log_2(e)}$$

Based on the above equation, we construct the range reduction function $RR(x)$, the output compensation function $OC(y')$, and the function that we need to approximate $g(x')$ as follows,

$$RR(x) = x' \qquad OC(y') = \frac{y' + m}{log_2(e)} \qquad g(x') = log_2(x')$$

Using this range reduction strategy, we approximate $ln(x)$ by first using $RR(x)$ to reduce the original input to the reduced input x'. Next, we use a polynomial approximation $y' = P(x')$ that approximates $g(x')$ for $x' \in [1, 2)$. Since $P(x')$ approximates $g(x')$ with the reduced input x', we use the output compensation function $OC(y')$ to generate an approximation of $ln(x)$ for the original input x. More formally,

$$ln(x) \approx OC(P(RR(x)))$$

Our goal now is to generate a polynomial approximation $P(x')$ that produces the correctly rounded results when used with the range reduction and output compensation function. Mathematically, $P(x')$ approximates $log_2(x')$ for the reduced inputs x' in the domain $[1, 2)$.

4.2.2 Identifying Correctly Rounded Results and Rounding Intervals

There are a total of 11 inputs in the input domain $(0, \infty)$. Our first step is to identify a range of values that our entire approximation function should produce for each input, such that the result rounds to the correctly rounded results. For each of the 11 inputs, we use an oracle (*i.e.*, MPFR math library) to compute y, the correctly rounded result of $ln(x)$ in FP5. Figure 4.1(a) shows the 11 inputs in the x-axis and the correctly rounded result for each input with a black dot.

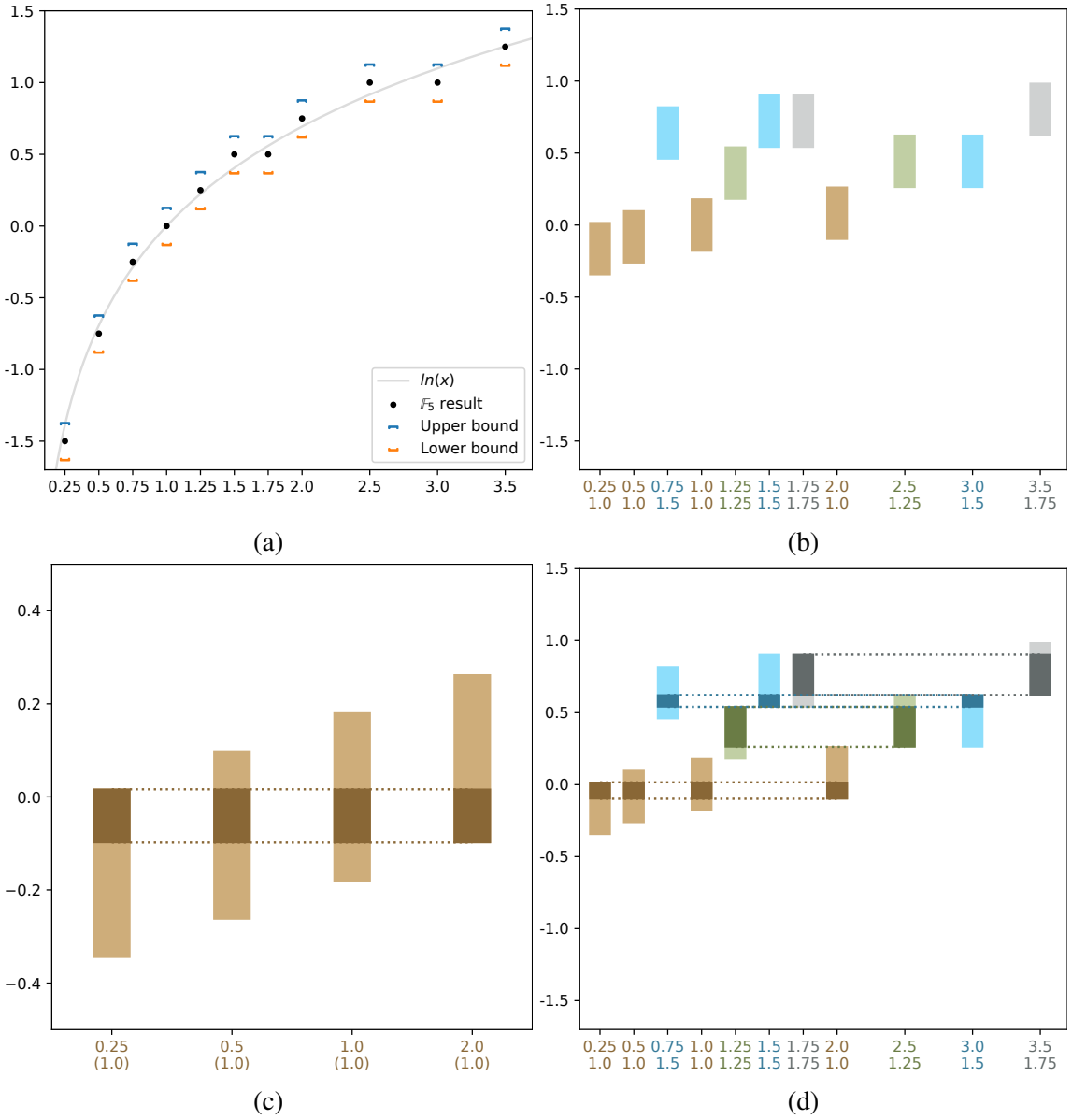


Figure 4.1: Our approach for $ln(x)$ for $\mathbb{F}_{5,2}$ with range reduction. (a) For each input x in FP5, we compute the correctly rounded results (black circle) and the rounding interval. (b) For each input and the corresponding rounding interval, we perform range reduction to obtain the reduced input (The number below each value on the x-axis). We also show the reduced interval for each input. Multiple inputs can map to the same reduced input after range reduction (intervals with the same color). In such scenarios, we combine the reduced intervals by computing the common region in the intervals. (c) We highlight the common region in intervals that correspond to the reduced input 1.0. (d) We highlight the common region in intervals that correspond to each reduced input with the darker color. The combined intervals are shown in Figure 4.2.

We perform the range reduction, output compensation, and polynomial evaluation using double precision. The double result of the approximation function is rounded to FP5. Thus, we identify the rounding interval $[l, h]$, a range of double values that rounds to the correctly rounded result, for each input. Figure 4.1(a) shows the rounding interval for each input with a bracket. The orange bracket represents the lower bound and the blue bracket represents the upper bound of the rounding interval.

4.2.3 Computing the Reduced Input and the Reduced Interval

Our approximation function takes an input x and uses the range reduction $RR(x)$ to produce the reduced input x'. The reduced input x' is used with a polynomial approximation $P(x')$ to produce the output y'. Finally, the output y' is compensated with the output compensation function $OC(y')$ to produce the final output y. The pair, $(x, [l, h])$ computed in the previous step constrains the output of the output compensation function such that our approximation for the elementary function as a whole produces the correctly rounded results. The next step is to use each $(x, [l, h])$ and infer all the inputs for $P(x')$ and the range of values that $P(x')$ should produce such that it produces the correctly rounded results when used with the output compensation function. The inputs to $P(x')$ are the reduced inputs. To compute the reduced input, we perform range reduction on each input x in the original input domain. Figure 4.1(b) shows the reduced input on the x-axis, below the original input.

Next, we identify a range of values that $P(x')$ should produce for each input, such that $P(x')$ produces the correctly rounded results when used with the output compensation function. We call this range of values the reduced interval. We infer the reduced intervals using the rounding interval $[l, h]$ for each input and the output compensation function. In our example where $OC(y')$ is a continuous and bijective function over the real numbers, we can use the inverse of the output compensation function to identify the reduced intervals.

For the input $x = 3.5 = 1.75 \times 2^1$, the output compensation function is,

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

and the inverse of the output compensation function is,

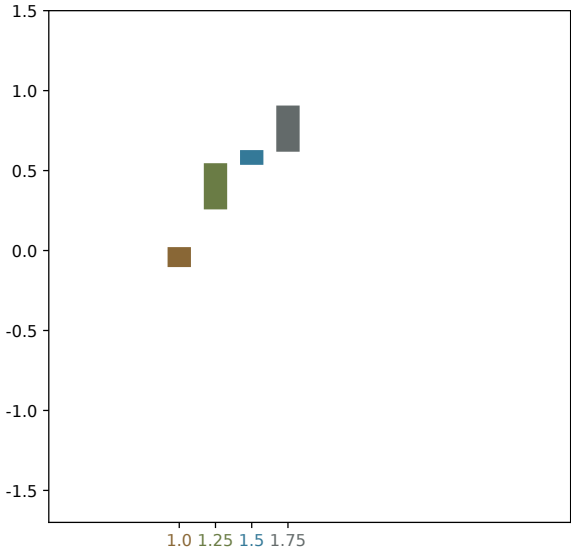
$$OC^{-1}(y) = y \times log_2(e) - 1$$

We use the inverse function and the rounding interval $[l, h]$ to compute the candidate reduced interval, $[l', h']$ by computing $l' = OC^{-1}(l)$ and $h' = OC^{-1}(h)$. Next, we restrict the reduced interval $[l', h']$ to account for the numerical error that occurs when evaluating the output compensation function with the double type. We verify whether the result of the output compensation function with l' (*i.e.*, $OC(l')$) and with h' (*i.e.*, $OC(h')$) lies in $[l, h]$ when evaluated in double. If it does not, then we iteratively restrict the reduced interval $[l', h']$ to a smaller interval until both $OC(l')$ and $OC(h')$ results in $[l, h]$ when evaluated in double. The vertical bars in Figure 4.1(b) show the reduced input and reduced interval for each original input x.

4.2.4 Combining the Reduced Intervals

Multiple inputs from the original input domain can map to the same reduced input after range reduction. In Figure 4.1(b), the inputs, reduced inputs, and reduced intervals corresponding the same reduced input are colored with the same color. Consider Figure 4.1(c), which depicts the reduced intervals for just four inputs $x_1 = 0.25$, $x_2 = 0.5$, $x_3 = 1.0$, and $x_4 = 2.0$. All four inputs map to the same reduced input $x' = 1.0$. However, the reduced intervals that we computed for x_1, x_2, x_3, and x_4 are $[l'_1, h'_1]$, $[l'_2, h'_2]$, $[l'_3, h'_3]$, and $[l'_4, h'_4]$, respectively. The reduced intervals are not identical because the intervals are computed to account for the output compensation function and any numerical error that may occur when evaluating $OC(y')$ in double.

The reduced interval for x_1 indicate that $P(1.0)$ must produce a value in $[l'_1, h'_1]$ such that the final result, after evaluating the output compensation function in double, will round



(a)

$$
\begin{aligned}
-0.098\ldots \leq P(1.00) \leq 0.016\ldots \\
0.262\ldots \leq P(1.25) \leq 0.541\ldots \\
0.541\ldots \leq P(1.50) \leq 0.623\ldots \\
0.623\ldots \leq P(1.75) \leq 0.901\ldots
\end{aligned}
$$

(b)

Figure 4.2: Continuation from Figure 4.1. (a) The combined reduced intervals by computing the common regions in the reduced intervals that correspond to the same reduced inputs from Figure 4.1. (b) The set of constraints that must be satisfied by the polynomial to produce the correctly rounded results when used with the output compensation function. The LP formulation and the resulting polynomial are shown in Figure 4.3.

to the correctly rounded result of $ln(0.25)$. Similarly, the reduced interval for x_2 indicate that $P(1.0)$ must produce a value in $[l'_2, h'_2]$ such that the final result rounds to the correctly rounded result of $ln(0.5)$. The reduced interval for each of the four inputs x_i, for $i \in \{1, 2, 3, 4\}$, indicate that $P(1.0)$ must produce a value in $[l'_i, h'_i]$ such that the final result rounds to the correct result of $ln(x_i)$. Thus, $P(1.0)$ must satisfy all four reduced intervals $[l'_i, h'_i]$:

$$P(1.0) \in [l'_1, h'_1] \cap [l'_2, h'_2] \cap [l'_3, h'_3] \cap [l'_4, h'_4]$$

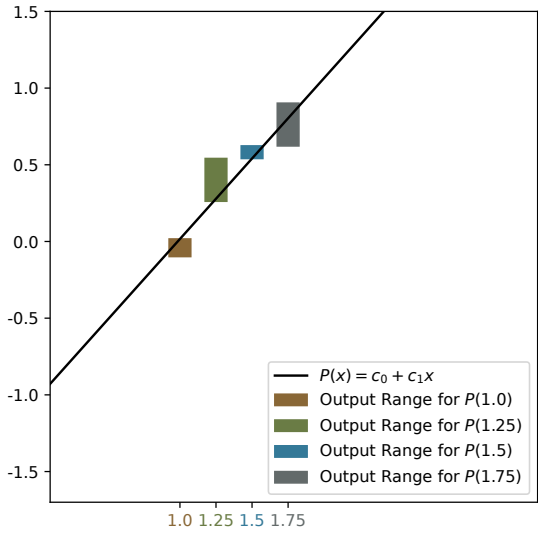
Hence, we combine all reduced intervals that correspond to the same reduced input by computing the common interval. Figure 4.1(c) shows the common interval for the reduced input $x' = 1.0$ using the darker brown color. Figure 4.1(d) shows the common interval for each reduced input using a darker shade of the color. Once the common intervals are combined, we are left with one combined reduced interval for each unique reduced input x'. The combined intervals for each reduced input are shown in Figure 4.2(a).

$$\begin{bmatrix} -0.098\ldots \\ 0.262\ldots \\ 0.541\ldots \\ 0.623\ldots \end{bmatrix} \leq \begin{bmatrix} 1 & 1 \\ 1 & 1.25 \\ 1 & 1.5 \\ 1 & 1.75 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \leq \begin{bmatrix} 0.016\ldots \\ 0.541\ldots \\ 0.623\ldots \\ 0.901\ldots \end{bmatrix}$$

(a)

$$\begin{aligned} P(x) &= c_0 + c_1 x \\ c_0 &= -1.0331383243\ldots \\ c_1 &= 1.04943264311\ldots \end{aligned}$$

(b)



(c)

Figure 4.3: Continuation from Figure 4.2. (a) The LP formulation to generate a monomial that satisfies all the constraints. (b) The coefficients generated by the LP solver. (c) Generated monomial satisfies all constraints.

4.2.5 Generating a Polynomial Approximation

The combined intervals specify the constraints on the output of the polynomial $P(x')$ for each reduced input x', such that the result when used with output compensation function in double results in the correctly rounded results for all original inputs. Figure 4.2(b) numerically shows the constraints for $P(x')$ for each reduced input.

To synthesize a polynomial $P(x')$ for the degree d, we encode the problem as an LP problem, similar to our approach in Chapter 3. We look for a polynomial that satisfies the constraints for each reduced input. Figure 4.3(a) shows the LP formulation for generating a monomial (polynomial of degree 1) that satisfies all constraints in Figure 4.2(b). The generated monomial (in Figure 4.3(b)) satisfies all constraints as shown in Figure 4.3(c). The use of range reduction allows us to generate lower degree polynomial (*i.e.* degree 1 in this example) compared to generating a polynomial for the entire input domain (*i.e.*, polynomial of degree 3 in Chapter 3).

4.3 Range Reduction Strategies for Various Elementary Functions

We now describe range reduction strategies that we used to generate approximations for various elementary functions. We group families of functions with similar range reduction strategies (*e.g.*, $log_a(x)$ or a^x). For each family of functions, we describe two range reductions. We first describe a simple range reduction strategy suitable for representations with a small number of bits (*i.e.* bfloat16). Second, we describe a more sophisticated range reduction strategy that is attractive for a larger bitwidth. The sophisticated strategies typically require pre-computed look-up tables or approximations of multiple functions to significantly reduce the input domain to generate low degree polynomial while also producing the correctly rounded result for all inputs.

4.3.1 Logarithm functions ($log_a(x)$)

The logarithm functions $log_a(x)$ is defined over the input domain $(0, \infty)$. In general, all range reductions for $log_a(x)$ use the mathematical identity,

$$log_a(t \times 2^m) = log_a(t) + m log_a(2)$$

to initially reduce the input domain by decomposing the input x into $x = t \times 2^m$ where $t \in [1, 2)$ is the significand of x and $m \in \mathbb{Z}$ is the integer exponent of x. Then, $log_a(x)$ can be computed with,

$$log_a(x) = log_a(t) + m log_a(2)$$

Using this range reduction, we must approximate $log_a(t)$ for the inputs $t \in [1, 2)$.

Basic Range Reduction of $log_a(x)$

To further reduce the input domain and, we use a slightly modified version of Cody and Waite's range reduction [25]. It uses the mathematical property of logarithms, $log_a(x) = \frac{log_2(x)}{log_2(a)}$ to approximate logarithm functions using the approximation of $log_2(x)$:

$$log_a(x) = \frac{log_2(t)}{log_2(a)} + m\frac{log_2(2)}{log_2(a)} = \frac{log_2(t) + m}{log_2(a)}$$

As a second step, we introduce a new variable $x' = \frac{t-1}{t+1}$. The variable t can be expressed as a function of x' using the following mathematical reasoning:

$$x' = \frac{t-1}{t+1}$$
$$tx' + x' = t - 1$$
$$t - tx' = 1 + x'$$
$$t(1 - x') = 1 + x'$$
$$t = \frac{1+x'}{1-x'}$$

We substitute t in $log_2(t)$ with $\frac{1+x'}{1-x'}$:

$$g(x') = log_2(t) = log_2\left(\frac{1+x'}{1-x'}\right)$$

The polynomial expansion of $g(x')$ is an odd polynomial, *i.e.* $P(x) = c_1x + c_3x^3 + c_5x^5 \ldots$ and converges rapidly as the number of terms increases. Combining all steps, we decompose $log_a(x)$ to,

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

The range reduction function $x' = RR(x)$, the output compensation function $y = OC(y')$, and the function that we need to approximate, $y' = g(x')$ can be summarized as follows,

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

where t and m is identified by decomposing $x = t \times 2^m$. With this range reduction technique, we need to approximate $g(x')$ for the reduced input domain $x' \in [0, \frac{1}{3})$.

Sophisticated Range Reduction for $log_a(x)$

To significantly reduce the input domain for large representations, we use a table-based approach [134] to perform range reduction for $log_a(x)$ functions. We reduce the range of t

Algorithm 4.1: `ComplexLogG1` uses the sophisticated range reduction strategy to approximate the elementary function $log_a(x)$ for inputs $x \geq 1$. `GetSignificand` and `GetExp` returns the significand and the exponent value of x, respectiively. Algorithm 4.2 presents the algorithm to approximate $log_a(x)$ for inputs $0 < x < 1$.

```
1 Function ComplexLogG1(x):
2     t ← GetSignificand(x); m ← GetExp(x)
3     J ← ⌊128(t − 1)⌋
4     F ← 1 + J/128
5     f ← t − F
6     x' ← f/F                          // Reduced input x' is a value in [0, 1/128)
7     y' ← g(x')                        // g(x') approximates log_a(1 + x')
8     y ← y' + log_a(F) + m log_a(2)    // Output compensation
9     return y
```

by transforming t into $t = F + f$ where F is a discrete variable with $F = 1 + \frac{J}{128}$, J is a value in the set $\{0, 1, 2, \ldots, 127\}$ and f is a continuous variable in the range of $f \in [0, \frac{1}{128})$. Intuitively, F is the value represented by the first 8 bits of the significand m and f is the value represented by the rest of the bits. Then, $log_a(t)$ can be computed with,

$$log_a(t) = log_a(F + f) = log_a\left(F\left(1 + \frac{f}{F}\right)\right) = log_a(F) + log_a\left(1 + \frac{f}{F}\right)$$

If we denote the reduced input $x' = \frac{f}{F}$, then $log_a(t)$ can be computed with,

$$log_a(t) = log_a(F) + log_a(1 + x')$$

The reduced input x' is in the range of $[0, \frac{1}{128})$. The computation $x' = \frac{f}{F}$ can be efficiently performed by computing $f \times \frac{1}{F}$ if the value $\frac{1}{F}$ is computed ahead of time and the values are stored in a look-up table (128 values). Additionally, since there are 128 possible values for F and $log_a(F)$ is a constant for a given base a, we pre-compute the values for $log_a(F)$ (128 values for each base a) and store them in a look-up tables. We approximate $log_a(1 + x')$ for the reduced input domain $x' \in [0, \frac{1}{128})$ with a polynomial.

Algorithm 4.1 summarizes the sophisticated range reduction strategy for $log_a(x)$, showing the range reduction (lines 2-6), polynomial approximations (line 7), and output compensation function (line 8).

Eliminating Cancellation Error

The values of $log_a(2)$ cannot be exactly represented with finite precision representations for all bases $a \in \mathbb{Z}$ except for $a = 2$. The approximation error for $log_a(2)$ can be amplified by m if the magnitude of m is large. Additionally, if $m = -1$, the output compensation function may experience cancellation error because $y' + log_a(F) \geq 0$ and $mlog_a(2) < 0$. The magnitude of $y' + log_a(F)$ and $mlog_a(2)$ may be similar and have different signs, leading to cancellation error and significant amount of numerical error when evaluating the output compensation function with finite precision.

We use a different range reduction strategy when $m < 0$ to avoid cancellation error. Once the original input x is decomposed to $x = t \times 2^m$ where $t \in [1, 2)$ is the significand and m is the exponent of x, we check whether $m < 0$. If it is, we modify the decomposition to $x = \frac{t}{2} \times 2^{m+1}$. If we denote $t' = \frac{t}{2}$ and $m' = m + 1$, then $x = t' \times 2^{m'}$. We further reduce the range of t' by transforming t' into $t' = F' - f'$ where F' is a discrete variable $1 - \frac{J'}{256}$, J' is a value in the set $\{0, 1, 2, \ldots, 127\}$, and f' is a continuous variable in the range of $f' \in [0, \frac{1}{256}]$. Then, $log_a(t')$ can be computed with,

$$log_a(t') = log_a(F' - f') = log_a\left(F'\left(1 - \frac{f'}{F'}\right)\right) = log_a(F') + log_a\left(1 - \frac{f'}{F'}\right)$$

If we denote the reduced input $x' = -\frac{f'}{F'}$, then x' is in the range of $[-\frac{1}{128}, 0]$. The function $log_a(x)$ can be computed with,

$$log_a(t') = log_a(F') + log_a(1 + x') + m'log_a(2)$$

All three terms $log_a(F')$, $log_a(1+x')$, and $m'log_a(2)$ are non-positive values and adding these three values do not experience cancellation error. The computation $x' = -\frac{f'}{F'}$ can be efficiently performed by computing $-f' \times \frac{1}{F'}$ if the value $\frac{1}{F'}$ is computed ahead of time and stored in a look-up table (128 values). Additionally, since there are 128 possible values for F' and $log_a(F')$ is constant for a given base a, we also pre-compute the values for $log_a(F')$ and store them in a look-up table (128 values for each base a). We approximate $log_a(1+x')$ for the reduced input domain $x' \in [-\frac{1}{128}, 0]$ with a polynomial.

Algorithm 4.2: `ComplexLogL1` uses the sophisticated range reduction strategy to approximate the elementary function $log_a(x)$ for inputs $0 < x < 1$.

1 **Function** `ComplexLogL1(`x`)`**:**
2 $t' \leftarrow$ `GetSignificand(`x`)` $\div 2$; $m' \leftarrow$ `GetExp(`x`)` $+ 1$ // $t' \in [0.5, 1)$
3 $J' \leftarrow 255 - \lfloor 256t' \rfloor$ // $J' \in \{0, 1, \ldots, 127\}$
4 $F' \leftarrow 1 - \frac{J'}{256}$
5 $f' \leftarrow F' - t'$
6 $x' \leftarrow -\frac{f'}{F'}$ // Reduced input x' is a value in $[-\frac{1}{128}, 0]$
7 $y' \leftarrow g(x')$ // $g(x')$ approximates $log_a(1 + x')$
8 $y \leftarrow y' + log_a(F') + m' log_a(2)$ // Output compensation
9 **return** y

Algorithm 4.2 summarizes the sophisticated range reduction strategy of $log_a(x)$ for the input $0 < x < 1$, showing the range reduction (lines 2-6), polynomial approximations (line 7), and output compensation function (line 8).

4.3.2 Exponential functions (a^x)

The exponential function a^x is mathematically defined for the input domain $x \in (-\infty, \infty)$. For small target representations, we use range reduction that reduces the input domain to a reduced input domain $x' \in [0, 1)$. For large target representations, we use table-based range reduction to reduce the input domain further.

Basic Range Reduction for a^x

For small representations, we approximate all exponential functions with 2^x. As a first step, we use the mathematical property $a^x = 2^{xlog_2(a)}$ to decompose any exponential function to a function of 2^x. Second, we decompose the value $xlog_2(a)$ into the integral part i and the remaining fractional part $x' \in [0, 1)$, *i.e.* $xlog_2(a) = i + x'$. More formally,

$$i = \lfloor xlog_2(a) \rfloor, \qquad x' = xlog_2(a) - i$$

where $\lfloor v \rfloor$ is a floor function that rounds v down to an integer. Finally, using the property $2^{x+y} = 2^x 2^y$, a^x decomposes to

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

The above decomposition allows us to approximate any exponential function by approximating 2^x for $x \in [0, 1)$. Multiplication by 2^i can be computed efficiently using bit-wise operations by increasing the exponent value of $2^{x'}$ in the bit-string of the higher precision representation used to evaluate the approximation function. The range reduction function $x' = RR(x)$, the output compensation function $y = OC(y', x)$, and the function we need to approximate $y' = g(x')$ can be summarized as follows:

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

With this range reduction technique, we need to approximate $2^{x'}$ for the reduced input domain $x' \in [0, 1)$.

Sophisticated Range Reduction for a^x

The table-based range reduction for exponential functions a^x [133] is applicable for all bases a. Our goal in the table-based range reduction is to use the mathematical identities $a^{x+y} = a^x a^y$ and $a^{xlog_a(2)} = 2^x$ to efficiently compute a^x. Suppose that we somehow decompose the input x into $x = tlog_a(2) + r$ where t is an integer and r is the remainder of the value, $r = x - tlog_a(2)$. Then, a^x can be computed with,

$$a^x = a^{tlog_a(2)+r} = a^{tlog_a(2)}a^r = 2^t a^r$$

In such a case, we need to approximate a^r and multiplication by 2^t can be efficiently computed using bit-wise operations. We use this decomposition twice in the range reduction.

To reduce the input x we transform x into $x = tlog_a(2) + \frac{j}{64}log_a(2) + x'$ where t is an integer, j is a value in a set $\{0, 1, 2, \ldots, 63\}$ and $|t'| \leq \frac{log_a(2)}{64}$. Intuitively, t can be computed by identifying the integral part of the value $\frac{x}{log_a(2)}$. The value $\frac{j}{64}$ can be computed by identifying the first 6 fractional bits of $\frac{x}{log_a(2)}$. Lastly, x' can be computed by identifying the remaining value, $x' = x - tlog_a(2) - \frac{j}{64}log_a(2)$.

The scaling by $log_a(2)$ allows us to create efficient output compensation formula. The

Algorithm 4.3: `ComplexExp` uses the sophisticated range reduction strategy to approximate the elementary function a^x for all inputs x.

1 **Function** `ComplexExp(`x`)`**:**
2 **if** $x < 0$ **then** $t \leftarrow \lceil \frac{x}{log_a(2)} \rceil$ **else** $t \leftarrow \lfloor \frac{x}{log_a(2)} \rfloor$ `// Integer cast (rnz mode)`
3 $t1 \leftarrow x - tlog_a(2)$
4 **if** $t1 < 0$ **then** $j \leftarrow \lceil t1 \times \frac{64}{log_a(2)} \rceil$ **else** $j \leftarrow \lfloor t1 \times \frac{64}{log_a(2)} \rfloor$
5 $x' \leftarrow t1 - \frac{j}{64} log_a(2)$ `// Reduced input` x' `is a value in` $[-\frac{log_a(2)}{64}, \frac{log_a(2)}{64}]$
6 $y' \leftarrow g(x')$ `//` $g(x')$ `approximates` $a^{x'}$
7 $y \leftarrow 2^t \times 2^{\frac{j}{64}} \times y'$ `// Output compensation`
8 **return** y

value of a^x can be computed with,

$$a^x = a^{tlog_a(2)+\frac{j}{64}log_a(2)+x'} = a^{tlog_a(2)} \times a^{\frac{j}{64}log_a(2)} \times a^{x'} = 2^t \times 2^{\frac{j}{64}} \times a^{x'}$$

Multiplication by 2^n can be computed efficiently using bit-wise operations. We pre-compute and store the values of $2^{\frac{j}{64}}$ in a look-up table (*i.e.* 64 values in total) and approximate $a^{x'}$ for the input domain of $x' \in [-\frac{log_a(2)}{64}, \frac{log_a(2)}{64}]$. Algorithm 4.3 summarizes the sophisticated range reduction strategy for a^x, showing the range reduction (lines 2-5), polynomial approximations (line 6), and output compensation function (line 7).

4.3.3 Hyperbolic Sine Function ($sinh(x)$)

Mathematically, the $sinh(x)$ is defined for the input domain $x \in (-\infty, \infty)$. We use a table-based range reduction inspired from CR-LIBM [30] for $sinh(x)$ using the hyperbolic identities $sinh(a+b) = sinh(a)cosh(b) + cosh(a)sinh(b)$ and $cosh(a+b) = cosh(a)cosh(b) + sinh(a)sinh(b)$.

Basic Range Reduction for $sinh(x)$

Using the property $sinh(-x) = -sinh(x)$, the result of $sinh(x)$ can be computed with $s \times sinh(|x|)$ where $s = 1$ if $x >= 0$ and $s = -1$ if $x < 0$. We first transform the input x into $|x|$ with $x = s \times |x|$. Next, we decompose $|x|$ into two parts:

$$|x| = kln(2) + x'$$

where $k \geq 0$ is an integer and $x' \in [0, ln(2))$. The intuition for scaling the values of k with $ln(2)$ will be explained below. If we denote $K = kln(2)$, then $sinh(|x|)$ can be computed with,

$$sinh(|x|) = sinh(K + x') = sinh(K)cosh(x') + cosh(K)sinh(x')$$

Because the magnitude of $sinh(x)$ increases exponentially, the result of $sinh(x)$ lies outside of the dynamic range for most values of the original input x. Hence, the values of k are limited even for relatively large precision representations (*i.e.* 32-bit float). We identify all possible values for k, pre-compute $sinh(K)$ and $cosh(K)$ for each k, and store the values into look-up tables.

We approximate two functions $g_1(x') = sinh(x')$ and $g_2(x') = cosh(x')$ for the reduced inputs $x' \in [0, ln(2))$. The function $sinh(x)$ can be computed with,

$$sinh(x) = s \times (sinh(K)cosh(x') + cosh(K)sinh(x'))$$

Finally, the range reduction $x' = RR(x)$, output compensation $y = OC(y'_1, y'_2)$, and the two functions that we must approximate ($y'_1 = g_1(x')$ and $y'_2 = g_2(x')$) can be summarized as follows:

$$\begin{aligned} RR(x) &= x' = |x| - kln(2), \\ g_1(x') &= y'_1 = sinh(x'), \quad g_2(x') = y'_2 = cosh(x') \\ OC(y'_1, y'_2) &= s \times (sinh(kln(2)) \times y'_2 + cosh(kln(2)) \times y'_1), \end{aligned}$$

where s and k are identified from the decomposition $x = s \times kln(2) + x'$ as described above. Note that the output compensation function requires approximation of two functions, $g_1(x')$ and $g_2(x')$.

Sophisticated Range Reduction for $sinh(x)$

Using the property $sinh(-x) = -sinh(x)$, we decompose the input x into $|x|$ with $x = s \times |x|$, where s is the sign of the original input x. Next, we decompose $|x|$ into three parts:

$$|x| = kln(2) + \frac{j}{64}ln(2) + x'$$

where both k and j are integers, $k \geq 0$, $0 \leq j < 64$, and x' is a value in $[0, \frac{ln(2)}{64})$. Intuitively, k can be computed by identifying the integral part of $\frac{|x|}{ln(2)}$, j can be computed by identifying the first 6 bits in the fractional part of $\frac{|x|}{ln(2)}$, and x' can be computed with the remainder, *i.e.* $x' = |x| - kln(2) - \frac{j}{64}ln(2)$. If we denote $K = kln(2)$ and $J = \frac{j}{64}ln(2)$, then $sinh(|x|)$ can be computed with,

$$sinh(|x|) = SH \times cosh(x') + CH \times sinh(x')$$
$$SH = sinh(K) \times cosh(J) + cosh(K) \times sinh(J)$$
$$CH = cosh(K) \times cosh(J) + sinh(K) \times sinh(J)$$

We pre-compute the values of $sinh(J)$, $cosh(J)$, $sinh(K)$, and $cosh(K)$ and store the values in look-up tables. There are only 64 possible values for J. The values of K are also limited, if we filter all inputs that cause the value of $sinh(x)$ to go outside of the dynamic range of the target finite-precision representation as special case inputs. For example, in the case of 32-bit float, the real value of $sinh(x)$ goes outside of the dynamic range of 32-bit float for all inputs x where $|x| > 130ln(2)$. Thus, the values of k do not exceed 129 and the size of the look-up tables for $sinh(K)$ and $cosh(K)$ is 130 entries each.

Alternatively, we can choose to compute the values of $sinh(K)$ and $cosh(K)$ manually using the following properties,

$$sinh(x) = \frac{e^x - e^{-x}}{2} \qquad cosh(x) = \frac{e^x + e^{-x}}{2}$$

Algorithm 4.4: `ComplexSinh` uses the sophisticated range reduction strategy to approximate the elementary function $sinh(x)$ for all inputs x.

1 **Function** `ComplexSinh(`x`)`**:**
2 **if** $x < 0$ **then** $s \leftarrow 1$ **else** $s \leftarrow 0$
3 $k \leftarrow \lfloor \frac{|x|}{ln(2)} \rfloor$
4 $t1 \leftarrow |x| - kln(2)$
5 $j \leftarrow \lfloor t1 \times \frac{64}{ln(2)} \rfloor$
6 $x' \leftarrow t1 - \frac{j}{64} ln(2)$ `// Reduced input` x' `is a value in` $[0, \frac{ln(2)}{64}]$
7 $y'_1 \leftarrow g_1(x')$ `//` $g_1(x')$ `approximates` $sinh(x')$
8 $y'_2 \leftarrow g_2(x')$ `//` $g_2(x')$ `approximates` $cosh(x')$
9 $SH \leftarrow sinh(kln(2))cosh(\frac{j}{64}ln(2)) + cosh(kln(2))sinh(\frac{j}{64}ln(2))$
10 $CH \leftarrow sinh(kln(2))sinh(\frac{j}{64}ln(2)) + cosh(kln(2))cosh(\frac{j}{64}ln(2))$
11 $y \leftarrow (-1)^s (SH \times y'_2 + CH \times y'_1)$ `// Output compensation`
12 **return** y

Then, $sinh(K)$ and $cosh(K)$ can be computed as follows,

$$sinh(K) = sinh(kln(2)) = 2^{k-1} - 2^{-k-1}$$

$$cosh(K) = cosh(kln(2)) = 2^{k-1} + 2^{-k-1}$$

Although $sinh(K)$ and $cosh(K)$ cannot be exactly represented with finite-precision representations for all K, the correctly rounded value of $sinh(K)$ and $cosh(K)$ for a higher-precision representations can be computed efficiently using bit-wise operations, additions, or subtractions. This strategy can be effective for representations with a large dynamic range such as the double type.

We approximate two functions $g_1(x') = sinh(x')$ and $g_2(x') = cosh(x')$ for the reduced inputs $x' \in [0, \frac{ln(2)}{64})$. Algorithm 4.4 summarizes the sophisticated range reduction strategy for $sinh(x)$, showing the range reduction (lines 2-6), polynomial approximations (line 7-8), and output compensation function (lines 9-11).

4.3.4 Hyperbolic Cosine Function ($cosh(x)$)

Mathematically, the $cosh(x)$ is defined for the input domain $x \in (-\infty, \infty)$. The range reduction for $cosh(x)$ uses a similar strategy as $sinh(x)$ (described in Section 4.3.3) to reduce the input x. We use a table-based range reduction inspired from CR-LIBM for $cosh(x)$ using the hyperbolic identities $sinh(a+b) = sinh(a)cosh(b) + cosh(a)sinh(b)$ and $cosh(a+b) = cosh(a)cosh(b) + sinh(a)sinh(b)$.

Basic Range Reduction for $cosh(x)$

The $cosh(x)$ function has a property, $cosh(-x) = cosh(x)$. The result of cosh(x) can be computed with $cosh(|x|)$. Using this property, we decompose the original input x into $x = s \times |x|$ where s is the sign of the input. Next, we decompose $|x|$ into two parts, $|x| = kln(2) + x'$ where $k \geq 0$ is an integer and x' is a value in $x' \in [0, ln(2))$. If we denote $K = kln(2)$, then cosh(x) can be computed with,

$$cosh(x) = cosh(K + x') = sinh(K)sinh(x') + cosh(K)cosh(x')$$

We pre-compute and store the values of $sinh(K)$ and $cosh(K)$ in a look-up table and approximate two functions $g_1(x') = sinh(x')$ and $g_2(x') = cosh(x')$ for the reduced inputs $x' \in [0, ln(2))$. Finally, the range reduction $x' = RR(x)$, output compensation $y = OC(y'_1, y'_2)$, and the two functions that we must approximate ($y'_1 = g_1(x')$ and $y'_2 = g_2(x')$) for $cosh(x)$ can be summarized as follows:

$$\begin{aligned} RR(x) &= x' = |x| - kln(2) \\ g_1(x') &= y'_1 = sinh(x'), \quad g_2(x') = y'_2 = cosh(x'), \\ OC(y'_1, y'_2) &= sinh(kln(2)) \times y'_1 + cosh(kln(2)) \times y'_2 \end{aligned}$$

where k is identified from the decomposition $|x| = kln(2) + x'$ as described above.

Algorithm 4.5: `ComplexCosh` uses the sophisticated range reduction strategy to approximate the elementary function $cosh(x)$ for all inputs x.

1 **Function** `ComplexCosh(`x`)`**:**
2 $k \leftarrow \lfloor \frac{|x|}{ln(2)} \rfloor$
3 $t1 \leftarrow |x| - kln(2)$
4 $j \leftarrow \lfloor t1 \times \frac{64}{ln(2)} \rfloor$
5 $x' \leftarrow t1 - \frac{j}{64}ln(2)$ // Reduced input x' is a value in $[0, \frac{ln(2)}{64}]$
6 $y'_1 \leftarrow g_1(x')$ // $g_1(x')$ approximates $sinh(x')$
7 $y'_2 \leftarrow g_2(x')$ // $g_2(x')$ approximates $cosh(x')$
8 $SH \leftarrow sinh(kln(2))cosh(\frac{j}{64}ln(2)) + cosh(kln(2))sinh(\frac{j}{64}ln(2))$
9 $CH \leftarrow sinh(kln(2))sinh(\frac{j}{64}ln(2)) + cosh(kln(2))cosh(\frac{j}{64}ln(2))$
10 $y \leftarrow SH \times y'_1 + CH \times y'_2$ // Output compensation
11 **return** y

Sophisticated Range Reduction for $cosh(x)$

We first decompose the input x into $x = s \times |x|$ where s is the sign of the original input x. The result of $cosh(x)$ can be computed with $cosh(|x|)$. Next, we decompose $|x|$ into three parts similar to how we decompose the input $|x|$ for $sinh(x)$:

$$|x| = kln(2) + \frac{j}{64}ln(2) + x'$$

Both k and j are integers, $k \geq 0$, $0 \leq j < 64$, and x' is a real number value in $[0, \frac{ln(2)}{64})$. If we denote $K = kln(2)$ and $J = \frac{j}{64}ln(2)$, then $cosh(|x|)$ can be computed with,

$$cosh(x) = SH \times sinh(x') + CH \times cosh(x')$$

$$SH = sinh(K) \times cosh(J) + cosh(K) \times sinh(J)$$

$$CH = cosh(K) \times cosh(J) + sinh(K) \times sinh(J)$$

We pre-compute and store the values of $sinh(K)$, $cosh(K)$, $sinh(J)$, and $cosh(J)$ in lookup tables. Alternatively, $sinh(K)$ and $cosh(K)$ can be computed manually using the methodology described in Section 4.3.3. We approximate both $g_1(x') = sinh(x')$ and $g_2(x') = cosh(x')$ for the reduced inputs $x' \in [0, \frac{ln(2)}{64})$. Algorithm 4.5 summarizes the sophisticated range reduction strategy for $sinh(x)$, showing the range reduction (lines 2-

Algorithm 4.6: `BasicSinpi` uses the basic range reduction strategy to approximate the elementary function $sinpi(x)$ for all inputs x.

```
1 Function BasicSinpi(x):
2     j ← x − 2⌊x/2⌋
3     k ← ⌊j⌋
4     l ← j − k
5     if l ≤ 0.5 then l′ ← l else l′ ← 1.0 − l
6     x′ ← l′                          // The reduced input x′ is a value in [0, 0.5]
7     y′ ← g(x′)    // Polynomial approximation g(x′) approximates sinpi(x′)
8     y ← (−1)^k y′                              // Output compensation function
9     return y
```

5), polynomial approximations (line 6-7), and output compensation function (lines 8-10).

4.3.5 Trigonometric Sinpi Function ($sinpi(x)$)

The $sinpi(x)$ function is equivalent to $sin(\pi x)$. Compared to the $sin(x)$ function, performing range reduction accurately for $sinpi(x)$ is considered straight-forward, because $sinpi(x)$ has a period of length 2. Decomposing the original input x into a value v within a period of oscillation (*i.e.*, $[0, 2]$) can be computed exactly with finite precision arithmetic since v can be computed with $v = x - 2i$ for an integer i. Comparatively, $sin(x)$ has a period of size 2π. Decomposing an arbitrary original input x into a value v within a period (*i.e.*, $[0, 2\pi]$) exactly is impossible using finite precision arithmetic since $v = x - 2\pi \times i$ for an integer i, but π cannot be represented exactly. Thus, many math libraries provide approximations of $sinpi(x)$ along with $sin(x)$.

Basic Range Reduction for $sinpi(x)$

The range reduction of $sinpi(x)$ for small representations leverages the periodicity of $sinpi(x)$ to reduce the input. First, we transform the input x into $x = 2i + j$ where i is an integer and $j \in [0, 2)$. Then, $sinpi(x) = sinpi(j)$ due to periodicity. Next, we decompose j into $j = k + l$ where $k \in \{0, 1\}$ is the integral part of j and $l \in [0, 1)$ is the

Algorithm 4.7: `ComplexSinpi` uses the sophisticated range reduction strategy to approximate the elementary function $sinpi(x)$ for all inputs x.

1 **Function** `ComplexSinpi(`x`)`**:**
2 $j \leftarrow x - 2\lfloor \frac{x}{2} \rfloor$
3 $k \leftarrow \lfloor j \rfloor$
4 $l \leftarrow j - k$
5 **if** $l \leq 0.5$ **then** $l' \leftarrow l$ **else** $l' \leftarrow 1.0 - l$ // l' is a value in $[0, 0.5]$
6 $n \leftarrow \lfloor 512l' \rfloor$
7 $x' \leftarrow l' - \frac{n}{512}$ // Reduced input x' is a value in $[0, \frac{1}{512}]$
8 $y'_1 \leftarrow g_1(x')$ // $g_1(x')$ approximates $sinpi(x')$
9 $y'_2 \leftarrow g_2(x')$ // $g_2(x')$ approximates $cospi(x')$
10 $y \leftarrow (-1)^k \times \left(sinpi\left(\frac{n}{512}\right) y'_2 + cospi\left(\frac{n}{512}\right) y'_1\right)$ // Output compensation
11 **return** y

fractional part. The value $sinpi(j)$ can be computed with,

$$sinpi(j) = (-1)^k \times sinpi(l)$$

Third, we use the fact that $sinpi(x)$ between $x = [0.5, 1)$ is a mirror of $sinpi(x)$ between $x = [0, 0.5]$ and decompose l into,

$$l' = \begin{cases} l & \text{if } l \leq 0.5 \\ 1.0 - l & \text{if } l > 0.5 \end{cases}$$

With this decomposition, $sinpi(l)$ can be computed with $sinpi(l) = sinpi(l')$. For small representations, we use the value l' as the reduced input (*i.e.* $x' = l'$). Then, we can approximate $sinpi(x)$ using the formula,

$$sinpi(x) = (-1)^k \times sinpi(x')$$

Algorithm 4.6 summarizes the basic range reduction strategy for $sinpi(x)$, showing the range reduction (lines 2-6), polynomial approximation (line 7), and output compensation function (line 8).

Sophisticated Range Reduction for $sinpi(x)$

The range reduction of $sinpi(x)$ for large representations is based on the basic range reduction strategy and further reduces the transformed input l' in the domain $l' \in [0, 0.5]$ using

table-based range reduction [135]. We split l' into $l' = \frac{n}{512} + x'$ where n is a discrete variable in the set $\{0, 1, \ldots, 256\}$ and x' is a continuous variable in $[0, \frac{1}{512}]$. The value of $sinpi(l')$ can be computed using the trigonometric identity $sinpi(a + b) = sinpi(a)cospi(b) + cospi(a) + sinpi(b)$:

$$sinpi(l') = sinpi\left(\frac{n}{512}\right) cospi(x') + cospi\left(\frac{n}{512}\right) sinpi(x')$$

We pre-compute and store the values of $sinpi\left(\frac{n}{512}\right)$ and $cospi\left(\frac{n}{512}\right)$ in lookup tables (*i.e.* total of 512 values). We approximate $g_1(x') = sinpi(x')$ and $g_2(x') = cospi(x')$ for the reduced input domain $x' \in [0, \frac{1}{512}]$.

Algorithm 4.7 summarizes the sophisticated range reduction strategy for $sinpi(x)$, showing the range reduction (lines 2-7), polynomial approximations (line 8-9), and output compensation function (line 10).

4.3.6 Trigonometric Cospi Function ($cospi(x)$)

The $cospi(x)$ function is equivalent to $cos(\pi x)$. Many math libraries provide approximations for $cospi(x)$ function along with $cos(x)$. It is a periodic function where the period of oscillation has a length of 2. Thus, $cospi(x)$ can be approximated accurately even for large inputs because the input x can be exactly reduced to a value v within a period of oscillation (*i.e.* $[0, 2]$) using the formula $v = x - 2i$ for an integer i. Similar to the $sinpi(x)$ function, the range reduction of $cospi(x)$ for small representations leverage the periodicity of $cospi(x)$ and the range reduction for large representations use table-based technique.

Basic Range Reduction for $cospi(x)$

First, we transform the input x into $x = 2i + j$ where i is an integer and $j \in [0, 2)$. Due to the periodicity, $cospi(x) = cospi(j)$. Second, we decompose j into $j = k + l$ where $k \in \{0, 1\}$ is the integral part of j and $l \in [0, 1)$ is the fractional part. Using the property $cospi(1 + x) = -cospi(x)$, the value $cospi(j)$ can be computed with,

$$cospi(j) = (-1)^k \times cospi(l)$$

Algorithm 4.8: `BasicCospi` uses the basic range reduction strategy to approximate the elementary function $cospi(x)$ for all inputs x.

```
1  Function BasicCospi(x):
2      j ← x − 2⌊x/2⌋
3      k ← ⌊j⌋
4      l ← j − k
5      if l ≤ 0.5 then l' ← l else l' ← 1.0 − l
6      if l ≤ 0.5 then m ← 0 else m ← 1
7      x' ← l'                          // The reduced input x' is a value in [0,0.5]
8      y' ← g(x')     // Polynomial approximation g(x') approximates cospi(x')
9      y ← (−1)^k (−1)^m y'                          // Output compensation function
10     return y
```

Third, we reduce l using the fact that $cospi(x)$ between $[0.5, 1)$ is a mirror image of $cospi(x)$ between $[0, 0.5]$ with the opposite sign. We decompose l into m and l' where:

$$m = \begin{cases} 0 & \text{if } l \leq 0.5 \\ 1 & \text{if } l > 0.5 \end{cases} \qquad l' = \begin{cases} l & \text{if } l \leq 0.5 \\ 1.0 - l & \text{if } l > 0.5 \end{cases}$$

With this decomposition, $cospi(l)$ can be computed with $cospi(l) = (-1)^m \times cospi(l')$. For small representations, we use the value l' as the reduced input (*i.e.*, $x' = l'$). Then, we can approximate $cospi(x)$ using the formula,

$$cospi(x) = (-1)^k \times (-1)^m \times cospi(x')$$

We approximate $g(x') = cospi(x')$ for the reduced input domain $x' \in [0, 0.5]$.

Algorithm 4.8 summarizes the basic range reduction strategy for $cospi(x)$, showing the range reduction (lines 2-7), polynomial approximation (line 8), and output compensation function (line 9).

Sophisticated Range Reduction for $cospi(x)$

The range reduction of $cospi(x)$ for large representations is based on the basic range reduction strategy. The transformed input $l' \in [0, 0.5]$ is further reduced to a value in smaller domain using table-based range reduction. One way of reducing l' into smaller value is to decompose it into $l' = a + b$, where $a >= 0$ and $b >= 0$, and use the trigonometric identity

$cospi(a+b) = cospi(a)cospi(b) - sinpi(a)sinpi(b)$. However, this strategy can cause cancellation error especially when the values of $cospi(a)cospi(b)$ and $sinpi(a)sinpi(b)$ are similar in magnitude and has the same sign.

Instead, we transform l' into x' and n such that,

$$l' = \begin{cases} x' & \text{if } l' < \frac{1}{512} \text{ (in this case, } n = 0) \\ \frac{n}{512} - x' & otherwise \end{cases}$$

where n is an integer value in the set $\{0, 1, 2, \ldots, 256\}$ and x' is a fractional value in $[0, \frac{1}{512}]$. Then, $cospi(l')$ can be computed with the trigonometric identity $cospi(a-b) = cospi(a)copsi(b) + sinpi(a)sinpi(b)$,

$$cospi(l') = \begin{cases} cospi(x') \text{ (because } n = 0) & \text{if } l' < \frac{1}{512} \\ cospi(\frac{n}{512}) \times cospi(x') + sinpi(\frac{n}{512}) \times sinpi(x') & otherwise \end{cases}$$

This formula is monotonically increasing for all inputs x' and does not experience cancellation error because $cospi(\frac{n}{512})$, $sinpi(\frac{n}{512})$, $cospi(x')$, and $sinpi(x')$ are non-negative for all possible combination of n and x'. We pre-compute and store the values of $sinpi\left(\frac{n}{512}\right)$ and $cospi\left(\frac{n}{512}\right)$ in lookup tables for a total of 514 values. We approximate $g_1(x') = sinpi(x')$ and $g_2(x') = cospi(x')$ for the reduced input domain $x' \in [0, \frac{1}{512}]$.

Algorithm 4.9 summarizes the sophisticated range reduction strategy for $cospi(x)$, showing the range reduction (lines 2-8), polynomial approximations (line 9-10), and output compensation function (lines 11-12).

4.4 Our Approach For Generating Polynomials With Range Reduction

Our goal in this chapter is to generate a polynomial approximation $P(x)$ that produces the correctly rounded result of $f(x)$ in the target representation $\mathbb{T}$ using a particular rounding mode rm when used with range reduction strategies. Depending on the range reduction

Algorithm 4.9: `ComplexCospi` uses the sophisticated range reduction strategy to approximate the elementary function $cospi(x)$ for all inputs x.

Function `ComplexCospi(`x`)`**:**
- $j \leftarrow x - 2\lfloor \frac{x}{2} \rfloor$
- $k \leftarrow \lfloor j \rfloor$
- $l \leftarrow j - k$
- **if** $l \leq 0.5$ **then** $l' \leftarrow l$ **else** $l' \leftarrow 1.0 - l$ `// l' is a value in [0,0.5]`
- **if** $l \leq 0.5$ **then** $m \leftarrow 0$ **else** $m \leftarrow 1$
- $n \leftarrow \lceil 512l' \rceil$
- **if** $l' < \frac{1}{512}$ **then** $x' \leftarrow l'$ **else** $x' \leftarrow \frac{n}{512} - l'$ `// x' is a value in [0, 1/512]`
- $y'_1 \leftarrow g_1(x')$ `// g1(x') approximates sinpi(x')`
- $y'_2 \leftarrow g_2(x')$ `// g2(x') approximates cospi(x')`
- **if** $l' < \frac{1}{512}$ **then** $y \leftarrow (-1)^k(-1)^m y'_2$ `// Output compensation`
- **else** $y \leftarrow (-1)^k(-1)^m \left(cospi\left(\frac{n}{512}\right) y'_2 + sinpi\left(\frac{n}{512}\right) y'_1\right)$
- **return** y

strategy, the output compensation function may be univariate (requires us to approximate a single function, *i.e.*, range reduction used for $log_a(x)$) or multivariate (requires us to approximate multiple functions, *i.e.*, range reduction used for $sinh(x)$). In this section, we describe how to generate polynomial approximations of $f(x)$ with univariate output compensation functions to explain our key insights and approach. In Section 4.5, we present a more general technique to handle multivariate output compensation functions.

We use the notation $A_{\mathbb{H}}(x)$ to represent the approximation of the elementary function $f(x)$ produced with our approach while evaluating all internal computation with the higher precision representation $\mathbb{H}$. The result of $A_{\mathbb{H}}(x)$ is then rounded to the target representation $\mathbb{T}$ to produce the final result. $A_{\mathbb{H}}(x)$ is composed of three components: (1) the range reduction function $x' = RR_{\mathbb{H}}(x)$ that reduces the original input into the reduced input in a smaller domain, (2) the polynomial approximation $y' = P_{\mathbb{H}}(x')$ using the reduced input x', and (3) the output compensation function $y = OC_{\mathbb{H}}(y')$ that compensates the output y' to produce the approximation of $f(x)$. More formally,

$$A_{\mathbb{H}}(x) = OC_{\mathbb{H}}(P_{\mathbb{H}}(RR_{\mathbb{H}}(x)))$$

The result of $A_{\mathbb{H}}(x)$ (*i.e.*, y) is rounded to $\mathbb{T}$ to produce the final result. Given a range reduction strategy, the task of creating $A_{\mathbb{H}}(x)$ that produces the correctly rounded result of

$f(x)$ for all inputs involves generating an appropriate polynomial $P_{\mathbb{H}}(x')$.

There are two challenges in generating $P_{\mathbb{H}}(x')$ such that the final result of $A_{\mathbb{H}}(x)$ produces the correctly rounded results. First, the polynomial must account for the range reduction and the output compensation function. Specifically, $P_{\mathbb{H}}(x')$ must produce a value that rounds to the correctly rounded result of $f(x)$ when used with the output compensation function $OC_{\mathbb{H}}(y')$. If we were to approximate $f(x)$ without range reduction (*i.e.*, Chapter 3), the rounding interval $[l, h]$ of each input x directly defined the values that our desired polynomial approximation should produce. However, with range reduction, the rounding interval for each input defines the constraint on the output of the $A_{\mathbb{H}}(x)$ as a whole (*i.e.*, $A_{\mathbb{H}}(x) \in [l, h]$ for each x). The original input x is transformed via the range reduction function and the output of the $P_{\mathbb{H}}(x)$ is altered by the output compensation function. We can no longer directly use the constraints $(x, [l, h])$ to generate polynomial approximations. Second, the polynomial also needs to account for the numerical error in evaluating the range reduction and output compensation function in $\mathbb{H}(x)$.

Hence, we infer the inputs x' for $P_{\mathbb{H}}(x')$ and the range of values $[l', h']$ that $P_{\mathbb{H}}(x')$ should produce based on the original input x, the rounding interval $[l, h]$, the range reduction function, and the output compensation function. We call the input x' the reduced input and the interval $[l', h']$ the reduced interval. The reduced inputs and their corresponding reduced intervals define input-and-output constraints for $P_{\mathbb{H}}(x')$ such that the result of $P_{\mathbb{H}}(x')$ used with the output compensation function produces correctly rounded results. We call the constraint pair $(x', [l', h'])$ the reduced constraint. Once the reduced constraints are identified for all inputs, we can then encode the problem of generating a polynomial that satisfies all reduced constraints into a system of linear inequalities and use an LP solver to generate $P_{\mathbb{H}}(x')$.

Algorithm 4.10: Our approach to generate a polynomial approximation $P_{\mathbb{H}}(x)$ that produces the correctly rounded result for all inputs when used with univariate output compensation function. On successfully finding a polynomial, it returns (true, $P_{\mathbb{H}}$). Otherwise, it returns (false, DNE) where DNE means that the polynomial Does-Not-Exist. Functions, `CalcReducIntervals` and `CombineReducIntervals` are shown in Algorithm 4.11 and Algorithm 4.12, respectively.

1 **Function** `CorrectlyRoundedPoly(`f, $\mathbb{T}$, $\mathbb{H}$, X, d, $RR_{\mathbb{H}}$, $OC_{\mathbb{H}}$`)`**:**
2 $L \leftarrow$ `CalcRndIntervals(`f, $\mathbb{T}$, $\mathbb{H}$, X`)`
3 **if** $L = \emptyset$ **then return** *(false, DNE)*
4 $L' \leftarrow$ `CalcReducIntervals(`L, $\mathbb{H}$, $RR_{\mathbb{H}}$, $OC_{\mathbb{H}}$`)`
5 **if** $L' = \emptyset$ **then return** *(false, DNE)*
6 $L^* \leftarrow$ `CombineReducIntervals(`L'`)`
7 **if** $L^* = \emptyset$ **then return** *(false, DNE)*
8 S, $P_{\mathbb{H}} \leftarrow$ `GeneratePoly(`L^*, d`)`
9 **if** S = *true* **then return** *(true,* $P_{\mathbb{H}}$*)*
10 **else return** *(false, DNE)*

4.4.1 High-level Overview of the RLIBM Approach For Univariate Output Compensation Functions

Our approach for generating the the polynomial $P_{\mathbb{H}}(x')$ is shown in Algorithm 4.10. Our approach assumes the existence of an oracle, which produces the real value of $f(x)$. The polynomial approximation is closely related to the specific range reduction strategy. Hence, we also require the range reduction $RR_{\mathbb{H}}(x')$ and the output compensation function $OC_{\mathbb{H}}(y')$ from the math library developer. Additionally, our approach requires the output compensation function $OC(y')$ to be invertible (*i.e.*, continuous and bijective) to automatically identify the reduced intervals. In our experience, we found that all univariate output compensation functions that we have used are invertible. Finally, the degree of the polynomial is an input provided by the math library designer.

Our approach extends from the approach described in Chapter 3. First, we compute the correctly rounded result of $f(x)$ (*i.e.* $RN_{\mathbb{T},rm}(f(x))$) for each input x using our oracle. Then, we identify the rounding interval $[l, h] \in \mathbb{H}$ where all values in the interval rounds to the correctly rounded result using the steps described in Chapter 3 (line 2). The pair $(x, [l, h])$ specifies that the final result of $A_{\mathbb{H}}(x)$ must produce a value in $[l, h]$ (*i.e.*, $A_{\mathbb{H}}(x) \in$

$[l, h]$) for the result to round to the correctly rounded result.

Second, for each input x, we compute the reduced input x' using range reduction. Then, we infer the reduced interval $[l', h']$ for each pair $(x, [l, h])$ in L using the output compensation function (line 4). Abstractly, the reduced intervals $[l', h']$ constrains the output of the polynomial $P_{\mathbb{H}}(x')$ such that the result, when used with the output compensation function, will produce a value in the rounding interval $[l, h]$. The final result of the output compensation function will then round to the correctly rounded result of $f(x)$ in $\mathbb{T}$. `CalcReducIntervals` in Algorithm 4.11 returns a list L' containing the the pair of reduced input and interval $(x', [l'.h'])$ for each pair $(x, [l, h])$ in L.

Third, multiple inputs from the original input domain x can map to the same reduced input after range reduction. Hence, there can be multiple reduced intervals corresponding to the same reduced input x'. Thus, we combine all reduced intervals that correspond to the same reduced input x' by identifying the common region and produce a pair $(x', [l^*, h^*])$ for each reduced input x' (line 6). `CombineReducIntervals` in Algorithm 4.12 returns a list L^* containing the constraint pair $(x', [l^*, h^*])$. Finally, we generate a polynomial of degree d using LP formulation so that all constraints in L^* are satisfied using the approach described in Section 3.3.3 (line 8).

Note that our approach for generating $P_{\mathbb{H}}(x)$ with range reduction involves two additional steps compared to the steps in Chapter 3. The first step that identifies the rounding interval for $A_{\mathbb{H}}(x)$ (`CalcRndIntervals`) and the last step that generates a polynomial that satisfies certain constraints (`GeneratePoly`) are identical. The two additional steps involve identifying the reduced inputs, reduced intervals, and combining reduced intervals. In the remainder of the section, we focus our efforts in presenting these steps.

4.4.2 Identifying Reduced Inputs and Reduced Intervals

Once we compute the rounding intervals $[l, h]$ for each original input x, the intervals constrain the values that our entire approximation $A_{\mathbb{H}}(x)$ should produce such that it produces

the correctly rounded result of $f(x)$ (*i.e.* $RN_{\mathbb{T}}(A_{\mathbb{H}}(x)) = RN_{\mathbb{T}}(f(x))$), for each input $x \in X$. Our next step is to identify the inputs x' to the polynomial $P_{\mathbb{H}}(x')$ and the range of values $[l', h']$ that the polynomial should produce such that $A_{\mathbb{H}}(x)$ produces the correctly rounded results. We call x' the reduced input and the interval $[l', h']$ the reduced interval.

The original input x is reduced to a reduced input x' using the range reduction function ($x' = RR(x)$). The reduced input x' is used as the input to the polynomial $P_{\mathbb{H}}(x')$ which produces the value y'. Hence, the input to the polynomial can be computed by applying the range reduction $x' = RR(x)$ to each original input x.

Next, we need to identify the reduced interval for each reduced input x'. The output of the polynomial y' is used as the input to the output compensation function $y = OC(y')$. The resulting value y is the final output of $A_{\mathbb{H}}(x)$. More specifically, $A_{\mathbb{H}}(x) = OC_{\mathbb{H}}(P_{\mathbb{H}}(x'))$. To produce the correctly rounded result, y has to be a value in the rounding interval $[l, h]$. Hence, we use the inverse of the output compensation function ($y' = OC^{-1}(y)$) along with the rounding interval to infer the reduced interval. This strategy is feasible if the output compensation function is continuous and bijective.

With real numbers, it is straightforward to compute the reduced interval by using the inverse output compensation function (*i.e.*, $[l', h'] = [OC^{-1}(l), OC^{-1}(h)]$). The result of the output compensation evaluated in real numbers using any value in this reduced interval will round to the correctly rounded results. However, the output compensation function is evaluated in a finite precision representation, $\mathbb{H}$, which may cause numerical errors. Thus, we take numerical errors into account by restricting the reduced interval and ensure that the result of the output compensation function, when evaluated in $\mathbb{H}$, produces the correctly rounded results for all values in the reduced interval.

Algorithm 4.11 presents our approach in computing the reduced constraints $(x', [l', h'])$ for each $(x, [l, h]) \in L$. For each original input x, we use the range reduction function to compute the reduced input x' (line 4). To compute the reduced interval $[l', h']$, we first evaluate the values $v_1 = OC^{-1}_{\mathbb{H}}(l)$ and $v_2 = OC^{-1}_{\mathbb{H}}(h)$, where $OC^{-1}_{\mathbb{H}}(y)$ evaluates the

Algorithm 4.11: `CalcReducIntervals` computes the reduced input x' and the reduced interval $[l', h']$ for each constraint pair $(x, [l, h])$ in L. The reduced constraint pair $(x', [l', h'])$ specifies the bound on the output of $P_{\mathbb{H}}(x')$ such that it produces the correct value for the input x when used with the output compensation function.

```
1  Function CalcReducIntervals(L, ℍ, RR_ℍ, OC_ℍ):
2      L' ← ∅
3      foreach (x, [l, h]) ∈ L do
4          x' ← RR_ℍ(x)
           // Set initial reduced interval
5          if OC_ℍ is an increasing function then
6              [α, β] ← [OC_ℍ^-1(l, x), OC_ℍ^-1(h, x)]
7          else [α, β] ← [OC_ℍ^-1(h, x), OC_ℍ^-1(l, x)]
           // Increase the lower bound if necessary
8          while OC_ℍ(α, x) ∉ [l, h] do
9              α ← GetSuccVal(α, ℍ)
10             if α > β then return ∅
11         end
           // Decrease the upper bound if necessary
12         while OC_ℍ(β, x) ∉ [l, h] do
13             β ← GetPrecVal(β, ℍ)
14             if α > β then return ∅
15         end
16         L' ← L' ∪ {(x', [α, β])}
17     end
18     return L'
```

inverse of the output compensation function with $\mathbb{H}$. We set a candidate reduced interval $[\alpha, \beta] = [v_1, v_2]$ if the output compensation function is an increasing function (lines 5-6) or $[\alpha, \beta] = [v_2, v_1]$ if the output compensation function is a decreasing function (line 7). Next, we restrict the interval $[\alpha, \beta]$ to account for the numerical error that occurs when evaluating $OC_{\mathbb{H}}(y')$. We verify whether the output compensated value of the lower bound (*i.e.*, $OC_{\mathbb{H}}(\alpha)$) results in [l, h]. If it does not, we replace α with the succeeding value in $\mathbb{H}$. We repeat the process until $OC_{\mathbb{H}}(\alpha)$ results in the rounding interval (lines 8-10). Similarly, we verify whether the output compensated value of the upper bound (*i.e.*, $OC_{\mathbb{H}}(\beta)$) results in [l, h]. If it does not, we replace β with the preceding value in $\mathbb{H}$. We repeat the process until $OC_{\mathbb{H}}(\beta)$ results in [l, h] (lines 11-13). If $\alpha > \beta$ at any point in our process, then

it indicates that there does not exist a polynomial that will produce the correctly rounded result when used with the output compensation function. In such a case, the math library developer must design a different range reduction strategy or increase the precision of $\mathbb{H}$.

If the resulting interval $[\alpha, \beta] \neq \emptyset$, then $[\alpha, \beta]$ is the reduced interval (*i.e.*, $[l', h'] = [\alpha, \beta]$). The reduced constraint pair $(x', [l', h'])$ for each $(x, [l, h])$ specifies the constraint on the values that $P_{\mathbb{H}}(x')$ should produce, such that $A_{\mathbb{H}}(x) \in [l, h]$. We create a list L' containing all reduced constraints (line 16).

4.4.3 Combining the Reduced Constraints

Each reduced constraint $(x'_i, [l'_i, h'_i]) \in L'$ corresponds to a constraint $(x_i, [l_i, h_i]) \in L$ and specifies a bound on the output of $P_{\mathbb{H}}(x'_i) \in [l'_i, h'_i]$ to ensure that $A_{\mathbb{H}}(x_i) \in [l_i, h_i]$. The range reduction function reduces the original input x_i from the entire input domain of $f(x)$ into a reduced input x'_i in a smaller reduced domain. Hence, it is possible for multiple original inputs to be range reduced to the same reduced input. More formally, there can exist multiple constraints $(x_1, [l_1, h_1]), (x_2, [l_2, h_2]), \cdots \in L$ where $x^* = RR(x_1) = RR(x_2) = \ldots$. Consequently, L' can contain multiple reduced constraints with the same reduced input $(x^*, [l'_1, h'_1]), (x^*, [l'_2, h'_2]), \cdots \in L'$. These reduced constraints specify that the polynomial $P_{\mathbb{H}}(x^*)$ must produce a value in $[l'_1, h'_1]$ to guarantee that $A_{\mathbb{H}}(x_1) \in [l_1, h_1]$ and it should also produce a value in $[l'_2, h'_2]$ to guarantee that $A_{\mathbb{H}}(x_2) \in [l_2, h_2]$. Hence, for each unique reduced input x^*, the polynomial must satisfy all reduced constraints corresponding to x^*, *i.e.*, $P_{\mathbb{H}}(x^*) \in [l'_1, h'_1] \cap [l'_2, h'_2] \cap \ldots$. If there are multiple reduced constraints with the same reduced input, we combine the reduced intervals corresponding to the same reduced inputs by computing the common interval.

The function `CombineReducIntervals` in Algorithm 4.12 describes our technique that combines all reduced constraints with the same reduced input. For each unique reduced input x^* in the set of reduced constraints L' (line 2-4), we identify all reduced intervals that correspond to x^* (line 5). Then, we identify the combined interval $[l^*, h^*]$ by computing

Algorithm 4.12: `CombineReducIntervals` combines any reduced constraints with the same reduced input, *i.e.* $(x'_1, [l'_1, h'_1])$ and $(x'_2, [l'_2, h'_2])$ where $x'_1 = x'_2$ into a single combined constraint $(x'_1, [l^*, h^*])$ by computing the common interval range in $[l'_1, h'_1]$ and $[l'_2, h'_2]$.

```
1  Function CombineReducIntervals(L'):
2      X* ← {x' | (x', I') ∈ L'}
3      L* ← ∅
4      foreach x* ∈ X* do
5          Ω ← {[l', h'] | (x*, [l', h']) ∈ L'}
6          [l*, h*] ← ⋂_{[l',h']∈Ω} [l', h']
7          if [l*, h*] = ∅ then return ∅
8          L* ← L* ∪ {(x*, [l*, h*])}
9      end
10     return L*
```

the intersection between the reduced intervals (line 6). If the combined interval is empty, then it indicates that there is no value that $P_{\mathbb{H}}(x')$ can produce, such that it produces the correctly rounded results of $f(x)$ for all inputs when used with output compensation function. Otherwise, we create a combined constraint pair $(x^*, [l^*, h^*])$ for each unique reduced input x^* and produce a list of constraints L^* (line 8).

Each combined constraint $(x', [l', h']) \in L^*$ specifies that $P_{\mathbb{H}}(x')$ should satisfy $l' \leq P_{\mathbb{H}}(x') \leq h'$. This constraint ensures that the result of the output compensation function using $P_{\mathbb{H}}(x')$ produces the correctly rounded result for all inputs,

$$RN_{\mathbb{T},rm}(OC_{\mathbb{H}}(P_{\mathbb{H}}(x'))) = RN_{\mathbb{T},rm}(f(x))$$

Our final step encodes the problem of solving for a polynomial that satisfies all combined constraints in L^* into a system of linear inequalities and uses an LP solver to generate a polynomial that satisfies all constraints in L^*. We use the same methodology described in Chapter 3 (`GenreatePoly` in Algorithm 3.4) to generate such polynomial.

4.5 The RLIBM Approach For Multivariate Output Compensation Functions

We now describe our approach to generate a polynomial approximation $P_{i\mathbb{H}}(x')$ that produce the correctly rounded results of an elementary function $f(x)$ in our target representa-

tion $\mathbb{T}$ when used with multivariate output compensation functions. Some range reduction strategy (*i.e.*, $sinh(x)$) uses multivariate output compensation function, where the output compensation functions require multiple functions to be approximated,

$$y' = OC_{\mathbb{H}}(g_1(x'), g_2(x'), \dots)$$

where $g_i(x')$ are the functions that we have to approximate with polynomial approximations. Our goal is to synthesize the polynomial approximations $P_{i\mathbb{H}}(x')$ for each $g_i(x')$. With multiple polynomial approximations and multivariate output compensation function, the entire approximation of $f(x)$ can be formally defined as,

$$A_{\mathbb{H}}(x) = OC_{\mathbb{H}}(P_{1\mathbb{H}}(x'), P_{2\mathbb{H}}(x'), \dots)$$

where x' is the reduced input computed with $x' = RR_{\mathbb{H}}(x)$. Similar to the approach we used in generating polynomial approximation for univariate output compensation function, our strategy involves identifying the reduced intervals $[l'_i, h'_i]$ for each $P_{i\mathbb{H}}(x)$ and for each reduced input x'. The reduced intervals for each $P_{i\mathbb{H}}(x')$ define the values that $P_{i\mathbb{H}}(x')$ should produce such that the result of the output compensation function produces the correctly rounded results. Then, the reduced inputs and the reduced intervals for each $P_{i\mathbb{H}}(x')$ can be used to frame the problem of generating $P_{i\mathbb{H}}(x')$ that satisfies the reduced constraint pair $(x', [l'_i, h'_i])$ as a linear programming problem and use an LP solver to generate each polynomial.

Challenges in identifying reduced intervals with multivariate output compensation function. There are two main challenges in generating reduced intervals for each $P_i(x')$ in the context of multivariate output compensation function. First, the output compensation function is a multivariate function with one output (*i.e.*, $y = OC(y'_1, y'_2, \dots)$). Hence, the inverse function does not exist, based on the Inverse Function Theorem [124]. We cannot use the same approach described in Section 4.4 to infer the reduced intervals with inverse output compensation function. Second, we must ensure that the size of the reduced in-

Algorithm 4.13: Our approach to generate polynomials $P_{i\mathbb{H}}(x')$ that approximate $g_i(x')$ in the multivariate output compensation function. On successfully finding polynomials, it returns a polynomial for each $g_i(x')$, where the output of the polynomial used with the output compensation function produces the correctly rounded results. Otherwise, it returns (false, DNE) where DNE means that the polynomial Does-Not-Exist. `CalcReducIntervalsMulti` is shown in Algorithm 4.14.

```
1  Function CorrectlyRoundedPolyMulti(f, 𝕋, ℍ, X, d, RR_ℍ, OC_ℍ):
2      L ← CalcRndIntervals(f, 𝕋, ℍ, X)
3      if L = ∅ then return (false, DNE)
4      𝕃' ← CalcReducIntervalsMulti(L, ℍ, RR_ℍ, OC_ℍ)
5      Result ← ∅
6      foreach (g_i, L'_i) ∈ 𝕃' do
7          if L'_i = ∅ then return (false, DNE)
8          L*_i ← CombineReducIntervals(L'_i)
9          if L*_i = ∅ then return (false, DNE)
10         S, P_iℍ ← GeneratePoly(L*_i, d)
11         if S = false then return (false, DNE)
12         Result ← Result ∪ (g_i, P_i)
13     end
14     return (true, Result)
```

terval $[l'_i, h'_i]$ for each $P_i(x')$ is large enough to provide sufficient freedom for generating polynomial approximations.

To address these challenges, we initially set the reduced interval for each $P_{i\mathbb{H}}(x')$ with a singleton value $[v_i, v_i]$. The intuition is to identify at least a single value where the output compensation function produces the correctly rounded results. Each value v_i for $P_{i\mathbb{H}}(x')$ guarantees that the result of the output compensation function using the singleton values will produce the correctly rounded result of $f(x)$. Next, we incrementally widen the reduced intervals $[l'_i, h'_i]$ for each $P_{i\mathbb{H}}(x')$ at the same time to ensure that the relative size of the reduced intervals is equal to each other. If the output compensation function is monotonically increasing and our methodology returns reduced intervals, then evaluating the output compensation function using any values in the reduced intervals is guaranteed to produce the correctly rounded result.

4.5.1 High-level Overview of the RLIBM Approach For Multivariate Output Compensation Function

The top-level algorithm for generating polynomials that produce the correctly rounded results when used with multivariate output compensation function is shown in Algorithm 4.13. Our algorithm generates a polynomial $P_{i\mathbb{H}}(x')$ for each $g_i(x')$ that must be approximated to use the output compensation function (line 12-13). If we are unable to find a polynomial for a particular $g_i(x')$ (line 11), then the developer of the math library should explore a different range reduction strategy, use a higher precision representation for $\mathbb{H}$, or generate a higher degree polynomial.

The approach to generate polynomials for multivariate output compensation function has four main steps. First, we compute the correctly rounded result of $f(x)$ for each input x using an oracle (line 2). Then, we identify the rounding interval $[l, h] \in \mathbb{H}$ where all values in the interval rounds to the correctly rounded result. The first step returns a list L containing the constraint pair $(x, [l, h])$ for each input x. Each constraint $(x, [l, h])$ constrains the output of our entire approximation $A_{\mathbb{H}}(x)$ such that it produces the correctly rounded result of $f(x)$.

Second, for each constraint $(x, [l, h])$, we identify the reduced input x' and the reduced intervals $[l'_i, h'_i]$ for each polynomial $P_{i\mathbb{H}}(x')$ at the same time (line 4). The reduced constraint pairs $(x', [l'_i, h'_i])$ for each $P_{i\mathbb{H}}(x')$ specify that if $P_{i\mathbb{H}}(x')$ produces a value in the reduced interval, then the polynomials used with the output compensation function will produce the correctly rounded results. `CalcReducIntervalsMulti` in Algorithm 4.14 returns a list $\mathbb{L}'$ containing the list of reduced constraints L'_i for each function $g_i(x')$.

Third, for each list L'_i, we combine all reduced intervals that correspond to the same reduced input x' using the strategy described in Section 4.4.3 to make sure that there is one combined interval $[l^*_i, h^*_i]$ for each unique x' (line 7). Finally, we generate the polynomials $P_{i\mathbb{H}}(x')$ that approximates $g_i(x')$ using the list of combined constraints L^*_i. In the remainder of this section, we present our algorithm to identify reduced intervals for each polynomial

Algorithm 4.14: `CalcRedIntervalsMulti` computes the reduced interval $[l_i', h_i']$ and the reduced input x' corresponding to input x for each function $g_i(x')$ used in the output compensation. If our polynomial approximation for $g_i(x')$ produces a value in $[l_i', h_i']$, then we can generate the correctly rounded result for x. The function returns a list with $(x', [l_i', h_i'])$ for each g_i.

1 **Function** `CalcReducIntervalsMulti(`L, $RR_{\mathbb{H}}$, $OC_{\mathbb{H}}$`)`**:**
2 **if** $OC_{\mathbb{H}}$ *is not monotonic function* **then return** $\emptyset$
3 $G \leftarrow$ {list of functions used in $OC_{\mathbb{H}}$}
4 **foreach** $g_i \in G$ **do** $\mathcal{L}_i \leftarrow \emptyset$
5 **foreach** $(x, [l, h]) \in L$ **do**
6 $x' \leftarrow RR_{\mathbb{H}}(x)$
7 $V \leftarrow \{RN_{\mathbb{H}}(g_i(x')) \mid g_i \in G\}$
8 **if** $OC_{\mathbb{H}}(V, x) \notin [l, h]$ **then return** $\emptyset$
9 $I' \leftarrow \{[v, v] \mid v \in V\}$ `// Singleton reduced interval for each` $g_i(x')$
10 **while** *true* **do**
`// Decrease the lower bounds` l_i' `simulataneously`
11 $A \leftarrow \{$`GetPrecVal(`l_i', $\mathbb{H}$`)` $\mid [l_i', h_i'] \in I'\}$
12 **if** $OC_{\mathbb{H}}(A, x) \notin [l, h]$ **then break**
13 $I' \leftarrow \{[$`GetPrecVal(`I_i', $\mathbb{H}$`)`$, h_i'] \mid [l_i', h_i'] \in I'\}$
14 **end**
15 **while** *true* **do**
`// Increase the upper bounds` h_i' `simulataneously`
16 $B \leftarrow \{$ `GetSuccVal(`h_i', $\mathbb{H}$`)` $\mid [l_i', h_i'] \in I'\}$
17 **if** $OC_{\mathbb{H}}(B, x) \notin [l, h]$ **then break**
18 $I' \leftarrow \{[I_i',$ `GetSuccVal(`h_i', $\mathbb{H}$`)`$] \mid [l_i', h_i'] \in I'\}$
19 **end**
20 **foreach** $[l_i', h_i'] \in I'$ **do**
21 $\mathcal{L}_i \leftarrow \mathcal{L}_i \cup (x', [l_i', h_i'])$
22 **end**
23 **end**
24 **return** $\{(g_i, \mathcal{L}_i) \mid g_i \in G\}$

$P_{i\mathbb{H}}(x')$.

4.5.2 Identifying Reduced Inputs and Intervals For Each Polynomial

After computing the rounding intervals from the first step (line 2 in Algorithm 4.13), we have a list of constraints $(x, [l, h])$ that our entire approximation function $A_{\mathbb{H}}(x)$ must satisfy for each input x to produce the correctly rounded results. The multivariate output compensation function requires us to approximate multiple functions $g_i(x')$. The polynomial approximations of each $g_i(x')$ must produce values such that evaluating the output

compensation function using these values produces the correctly rounded result of $f(x)$. The goal of our second step is to identify the reduced inputs and the reduced intervals for each $P_{i\mathbb{H}}(x')$.

Algorithm 4.14 presents the process of identifying the reduced input and intervals. The reduced inputs can be computed using the range reduction function on each input x (line 6). To compute the reduced intervals, we use a two-step approach. First, for each reduced input x', we identify a single value v_i for each $g_i(x')$ using an oracle. Using v_i's with the output compensation function produces the correctly rounded result (*i.e.*, $OC_{\mathbb{H}}(v_1, v_2, \dots) \in [l, h]$). The purpose of v_i's is to be the starting point in identifying the reduced intervals. If the representation $\mathbb{H}$ has high enough precision, then using the correctly rounded result of $g_i(x')$ in $\mathbb{H}$ (*i.e.*, $v_i = RN_{\mathbb{H}}(g_i(x'))$) with the output compensation function will produce the correctly rounded result (line 7). If the result of output compensation using $RN_{\mathbb{H}}(g_i(x'))$ does not produce the correctly rounded result, then either the precision of $\mathbb{H}$ should be increased or the range reduction strategy should be redesigned. An alternative approach that we have found useful in some cases is to search for a value in the vicinity of $g_i(x')$ in $\mathbb{H}$ that can be used for v_i. In our experience, the values (*i.e.*, v_i) that produce the correctly rounded result when used with output compensation were at most one or two $\mathbb{H}$ values away from $g_i(x')$. Using v_i, we create a candidate reduced interval $[l'_i.h'_i] = [v_i, v_i]$ for each $g_i(x')$ (line 9). These singleton intervals satisfy the most important invariant of a reduced interval: Any value within the reduced intervals (*i.e.*, v_i's) must guarantee to produce the correctly rounded result when used with the output compensation function.

Based on the candidate reduced interval $[l'_i, h'_i]$ for each $P_{i\mathbb{H}}(x')$, our next step is to identify the maximum amount of freedom available to generate $P_{i\mathbb{H}}(x')$. We check if we can decrease the lower bound of the intervals for each $P_{i\mathbb{H}}(x')$ at the same time. We iteratively check if using the preceding values of l'_i in $\mathbb{H}$ with the output compensation function produces a value in the rounding interval $[l, h]$. If it does, then we widen the reduced in-

terval by replacing l_i' with the preceding value of l_i. We repeat the process until using the preceding values of l_i' with the output compensation function does not produce a value in $[l, h]$ (lines 10-13).

Similarly, we check if we can increase the upper bound of the intervals for each $P_{i\mathbb{H}}(x')$. We compute the succeeding value of h_i' and check if the output compensation function using these values will produce a value in $[l, h]$. If it does, then we widen the reduced interval by replacing each h_i' with the succeeding value of h_i'. We repeat this process until the output compensation function using the succeeding value of h_i' does not produce a value in $[l, h]$ (lines 14-17). This process identifies the reduced intervals $[l_i', h_i']$ for each $P_{i\mathbb{H}}(x')$ where using any value within the interval with the output compensation function produces the correctly rounded results. Additionally, it ensures that the relative size of $[l_i', h_i']$ for each $P_{i\mathbb{H}}(x')$ for a given input x is the same, providing a similar amount of freedom between each polynomial.

Identifying the lower bound and the upper bound of the reduced inputs can be computed more efficiently. Because the output compensation function is monotonically increasing, the final lower bounds can be identified by performing a binary search between v_i and the minimum representable value of $\mathbb{H}$. Similarly, the final upper bounds can be identified by performing a binary search between v_i and the maximum representable value of $\mathbb{H}$. Thus, widening the reduced intervals requires at most n iterations where n is the number of bits in $\mathbb{H}$.

Finally, we store the reduced constraints $(x', [l_i', h_i'])$ for each function $g_i(x')$ in a list L_i' (line 19) and return L_i' for all $g_i(x')$ (line 20). Our approach then combines the reduced intervals that map to the same reduced inputs and produces a polynomial that satisfies all constraints using LP formulation.

4.6 Summary

A common strategy to generate an efficient polynomial approximation of an elementary function $f(x)$ is to use range reduction strategies, which reduce the input domain of the function into a smaller domain. Thus, a low degree polynomial can be used to produce correctly rounded results. Generating polynomial approximations that produce correctly rounded results when used with range reduction is a difficult task. Both the range reduction function and the output compensation function are evaluated in finite precision representations and can experience numerical errors.

In this chapter, we propose a novel technique to generate efficient and correctly rounded approximation functions in the presence of range reduction. Our key insight is to infer the inputs and the range of values that our polynomial approximation should produce using the rounding interval of the correctly rounded results of $f(x)$, the range reduction function, and the output compensation functions. Once such a list of inputs and the range of output values for the desired polynomial is identified, then we can use an LP formulation to specify the constraints for our polynomial and use an LP solver to generate the polynomial, as described in Chapter 3. When our approach successfully generates polynomials, then the polynomials used with the output compensation function are guaranteed to produce the correctly rounded results for all inputs. Our approach supports various complex range reduction strategies where the output compensation functions require approximations of one or more functions. We also propose improved range reduction strategies for several elementary functions to eliminate cancellation errors in the output compensation functions, increasing the numerical stability. Our methodology automatically generates a polynomial given an elementary function $f(x)$, the range reduction function, and the output compensation function, thus requiring little assistance from math library developers.

CHAPTER 5
THE RLIBM APPROACH FOR 32-BIT REPRESENTATIONS

In this chapter, we scale the RLIBM approach described in the previous chapters to produce correctly rounded results for 32-bit representations. The primary challenge in generating polynomial approximations for 32-bit representations is that there are four billion inputs and the result must be significantly more accurate compared to 16-bit representations. To address these challenges, we propose two enhancements. First, we use counterexample guided polynomial generation to handle millions of constraints. Second, we generate efficient piecewise polynomials by splitting the input domain using the bit-pattern of the inputs.

5.1 Scaling Our Approach to 32-Bit Representations

The IEEE-754 standard 32-bit float type is one of the most commonly used datatypes for three reasons. (1) Most commercial hardware supports efficient float arithmetic. (2) It provides a reasonable amount of dynamic range (approximately $[10^{-45}, 10^{38})$) and precision (roughly 6 decimal digits) for many scientific applications. (3) It requires half the storage space compared to the 64-bit double, another configuration supported in common hardware, providing faster data transfer. Unfortunately, widely used math libraries (*i.e.*, glibc's and Intel's libm) do not produce correctly rounded results of elementary functions for all inputs in the 32-bit float.

There are two primary challenges in scaling the RLIBM approach to 32-bit representations. First, 32-bit representations have four billion inputs. Even after range reduction, there may be millions of reduced inputs. LP formulations with millions of constraints are beyond the capabilities of the state-of-the-art LP solvers. Second, it may not be possible to generate a single polynomial with a reasonable degree that satisfies all the constraints.

Although a high-degree polynomial can produce correct results for all 32-bit inputs, such a polynomial may not be ideal from a performance viewpoint. It is desirable to use lower-degree polynomial considering the performance of the math library. Hence, we extend the RLIBM approach with two techniques. First, we propose counterexample guided polynomial generation technique that samples inputs to handle a large number of constraints. Second, we generate piecewise polynomials to create efficient polynomial approximations.

Piecewise polynomials. Given a list of inputs in the domain $[a, b]$, we first try to generate a single polynomial that produces the correctly rounded result for all inputs using the counterexample guided polynomial generation. If we are unable to generate a polynomial or the polynomial does not satisfy our performance constraint, then we split the input domain $[a, b]$ into sub-domains $[a, b')$ and $[b', b]$ and generate a polynomial for each sub-domain. To identify the inputs that belong to each sub-domain, we use bits in the bit-string of the input. If we still cannot generate polynomials with a reasonable degree for each sub-domain, then we iteratively split the original input domain into smaller sub-domains until we can generate a piecewise polynomial that produces the correctly rounded results for all inputs. By using bit-patterns to identify sub-domains and generating low degree polynomial for each sub-domain, our strategy produces efficient piecewise polynomials.

Counterexample guided polynomial generation. Even after splitting the input domain, there may still be millions of inputs in each sub-domain. Thus, we sample a small portion of inputs in the sub-domain to generate a polynomial. Our intuition is that it is not necessary to reason about all constraints to generate a polynomial that produces correctly rounded results for all inputs. It is sufficient to reason about inputs with highly constrained intervals. Our counterexample guided polynomial generation technique uses the following process. First, we sample some inputs in the sub-domain and generate a candidate polynomial that satisfies the constraints in the sample. Next, we check whether the candidate polynomial satisfies all constraints in the sub-domain. We add any constraints not satisfied

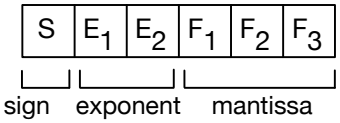


Figure 5.1: Bit-string representation of a 6-bit FP (FP6) with 2 exponent bits and 3 mantissa bits.

by the polynomial to the sample. We repeat the process until the generated polynomial satisfies all constraints in the sub-domain or the LP solver determines that it cannot generate a polynomial that satisfies all constraints in the sample. If the LP solver cannot generate a polynomial or there are too many inputs in the sample, we split the input domain into smaller sub-domains and repeat the process. The counterexample guided polynomial generation is inspired by the counterexample guided inductive synthesis (CEGIS) [72, 123] used in program synthesis.

5.2 Illustration

We show an end-to-end example of creating piecewise polynomials that produce the correctly rounded results of $ln(x)$ in a 6-bit FP representation (FP6) that has two exponent bits and three mantissa bits (Figure 5.1). In our illustration, we use the double representation to perform all internal computation. This example is illustrated for pedagogical reasons to highlight our approach in splitting sub-domains and performing counterexample guided polynomial generation. In practice, it is more beneficial to create a look-up table containing the results of $ln(x)$ in FP6 as there are only $2^6 = 64$ values.

The elementary function $ln(x)$ is defined over the input domain $(-\infty, \infty)$. There are 23 FP6 values ranging from 0.125 to 3.75 in the input domain. The remaining FP6 values are considered as special cases. If $x = 0$, then we return $ln(x) = -\infty$. When $x = \infty$, then we return $ln(x) = \infty$. If $x < 0$ or $x = NaN$, then the function $ln(x)$ is not defined and we return NaN.

Our approach to generating piecewise polynomials for FP6 involves three steps. First, we identify the correctly rounded result of $ln(x)$ in FP6 and its rounding interval for each

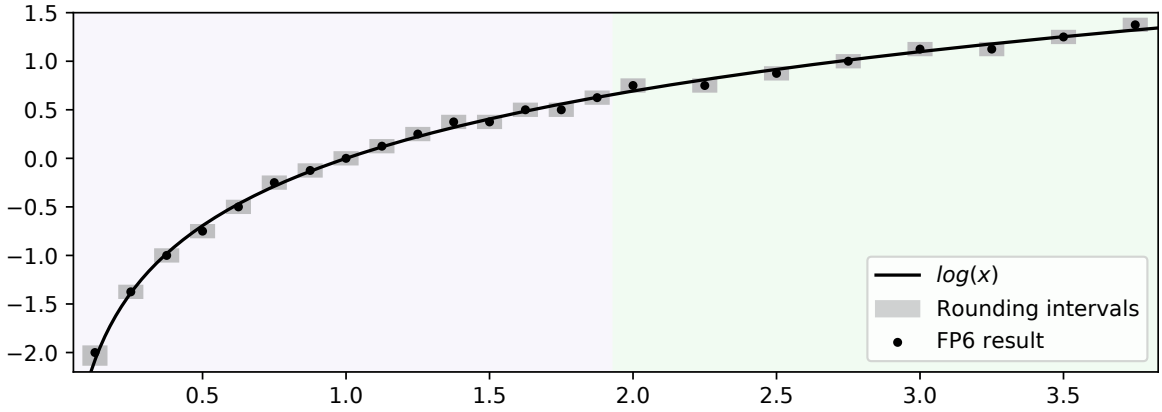


Figure 5.2: We show the function $ln(x)$ with black line. The correctly rounded result of $ln(x)$ for each input in FP6 is shown with a black circle. The gray boxes represent the rounding intervals of each correctly rounded result. We split the entire input domain into two sub-domains: the first sub-domain including inputs from 0.125 to 1.875 (purple background) and the second sub-domain including inputs from 2.0 to 3.5 (green background). We then generate a polynomial that produces correct results for each sub-domain.

input. Next, we need to generate polynomial approximations for $ln(x)$. While it is possible to generate a single polynomial of degree 5 for the entire input domain, our goal is to generate efficient polynomials. Instead of a single polynomial, we generate a piecewise polynomial where each polynomial has a lower degree, thus increasing performance. Hence, our second step splits the input domain into smaller sub-domains. Third, we perform counterexample guided polynomial generation to obtain a polynomial for each sub-domain. At the end of this process, each polynomial produces the correctly rounded result for all inputs in its corresponding sub-domain. Since there are only 23 inputs in FP6 for $ln(x)$, we do not use range reduction for exposition.

Identifying the correctly rounded result and rounding intervals. Similar to the approaches described in previous chapters, we compute the correctly rounded result of $ln(x)$ using an oracle for each input x in FP6. Then, we identify the rounding interval $[l, h]$ in double representation where all values in the interval rounds to the correctly rounded result. Our goal is to generate piecewise polynomials that produce a value in the rounding interval for each input x. Figure 5.2 shows the correctly rounded result of $ln(x)$ in FP6 (black circle) and

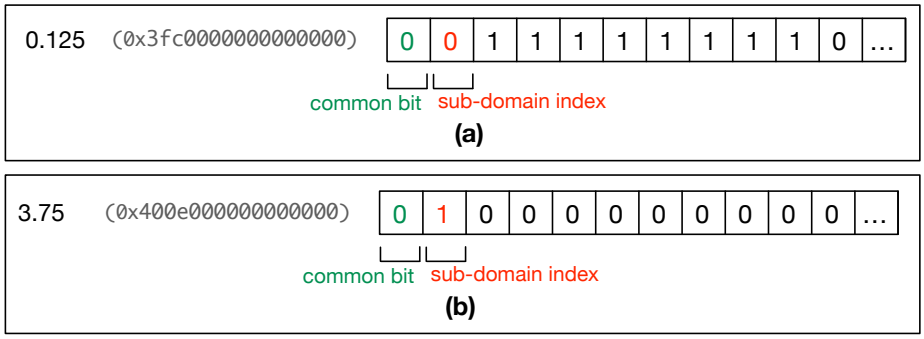


Figure 5.3: To approximate $ln(x)$ for FP6, we create piecewise polynomials with 2 sub-domains. We use a bit in the double representation of the input to identify the sub-domain for each polynomial. (a) shows the smallest input for $ln(x)$ in FP6 which corresponds to the first sub-domain. (b) shows the largest input which corresponds to the second sub-domain.

the corresponding rounding interval (gray area) for each input in FP6. Each pair $(x, [l, h])$ defines a constraint on the piecewise polynomials that we want to generate.

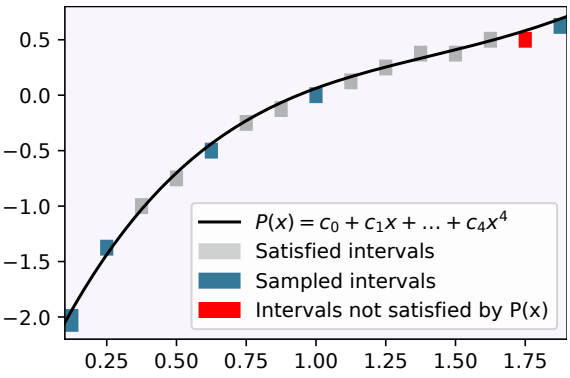
5.2.1 Domain Splitting

Once we have a list of constraints $(x, [l, h])$, the next step is to generate polynomial approximations for $ln(x)$. Initially, we try to generate a single polynomial for the entire input domain using the counterexample guided polynomial generation. If we cannot generate a single polynomial or the polynomial does not meet the performance threshold, then we split the input domain into sub-domains and generate piecewise polynomials. We iteratively split the domain into smaller sub-domains until we can generate polynomials that produce the correctly rounded results for all inputs and satisfies the performance requirement.

Let us suppose that we are going to split the input domain into 2 sub-domains and generate a polynomial approximation for each sub-domain. We use the bit-pattern of the input in the double representation to identify the sub-domain. All inputs for $ln(x)$ are positive values. In FP representations, the first bit of the bit-string represents the sign bit. Thus, the most significant bit (the first bit) among all inputs is identical, as shown in Figure 5.3(a) and (b). We use the next bit (the second bit), to identify the inputs that belong to each of the two sub-domains. All inputs such as $x = 0.125$ (Figure 5.3(a)) where the

$$
\begin{aligned}
-2.125 &\leq P(0.125) \leq -1.9375 \\
-1.4375 &< P(0.25) < -1.3125 \\
-0.5625 &\leq P(0.625) \leq -0.4375 \\
-0.0625 &\leq P(1.0) \leq 0.0625 \\
0.5625 &< P(1.875) < 0.6875
\end{aligned}
$$

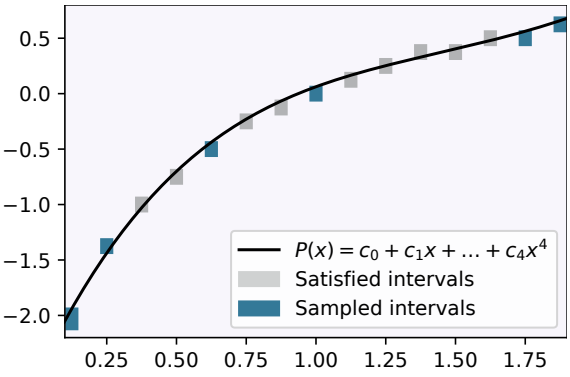
(a)



(b)

$$
\begin{aligned}
-2.125 &\leq P(0.125) \leq -1.9375 \\
-1.4375 &< P(0.25) < -1.3125 \\
-0.5625 &\leq P(0.625) \leq -0.4375 \\
-0.0625 &\leq P(1.0) \leq 0.0625 \\
0.4375 &\leq P(1.75) \leq 0.5625 \\
0.5625 &< P(1.875) < 0.6875
\end{aligned}
$$

(c)



(d)

Figure 5.4: The procedure to generate a polynomial for the first sub-domain. (a) We initially sample five constraints and generate a candidate 4^{th} degree polynomial that satisfies these constraints. (b) The generated polynomial satisfies all constraints except for the input $x = 1.75$. (c) We add the constraint corresponding to the input $x = 1.75$ to the sample and generate another polynomial that satisfies the six constraints. (d) The second polynomial satisfies all constraints in the first sub-domain.

second bit is 0 is grouped into the first sub-domain. There are 15 inputs ranging from 0.125 to 1.875 in the first sub-domain. Similarly, all inputs similar to $x = 3.75$ (Figure 5.3(b)) where the second bit is 1 is grouped into the second sub-domain. There are 8 inputs ranging from 2.0 to 3.75 in the second sub-domain. The graph in Figure 5.2 shows the two sub-domains with different background colors, purple for the first sub-domain and green for the second sub-domain.

5.2.2 Polynomial Generation for Each Sub-Domain.

The final step is to generate a polynomial for each sub-domain. This polynomial must produce a value within the rounding interval $[l, h]$ for each input x in the sub-domain. In the first sub-domain, there are 15 such inputs and the corresponding rounding intervals. Initially, we sample a portion of the inputs in the first sub-domain. Figure 5.4(a) shows the five inputs that we sample. We encode the inputs and the rounding intervals as linear constraints, create an LP query, and use an LP solver to generate a candidate polynomial that satisfies the constraints in the sample. Figure 5.4(b) shows a candidate 4^{th} degree polynomial. Next, we check whether the candidate polynomial produces a value in the rounding interval for all inputs in the sub-domain. There is an input where the polynomial does not produce a value within the reduced interval, highlighted with the red box Figure 5.4(b). We add this counterexample input to the sample (Figure 5.4(c)). The polynomial created using the new sample satisfies all the constraints in the first sub-domain, as shown in Figure 5.4(d).

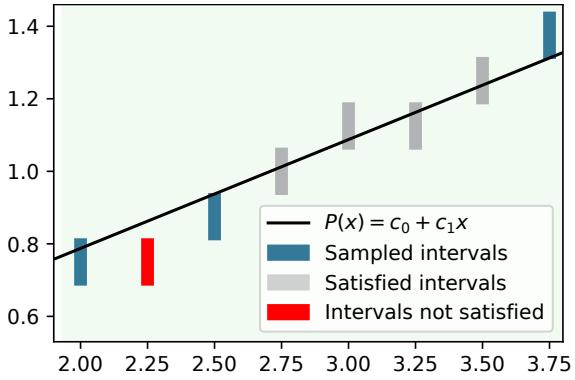
Using the same approach, we generate a polynomial for the second sub-domain. Figure 5.5(a) shows the initial sample of three inputs and the constraints. The 1^{st} degree polynomial generated using the sample of inputs does not produce a value in the rounding interval for an input in the second sub-domain. (Figure 5.5(b)). We add this counterexample to the sample (Figure 5.5(c)) and generate another candidate polynomial that satisfies the four constraints (Figure 5.5(d)). This polynomial does produce a value in the rounding interval for all inputs in the second sub-domain. The final piecewise polynomial that produces the correctly rounded results of $f(x)$ for all 23 inputs is shown in Figure 5.6

5.3 Our Approach to Generate Piecewise Polynomials

Algorithm 5.1 provides a high-level overview of our approach to generate piecewise polynomials that produce correctly rounded results for all inputs in a given 32-bit representation

$$0.6875 \leq P(2) \leq 0.8125$$
$$8.125 < P(2.5) < 0.9375$$
$$1.3125 < P(3.75) < 1.4375$$

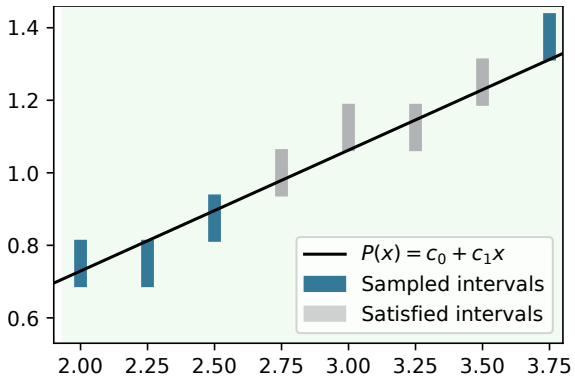
(a)



(b)

$$0.6875 \leq P(2) \leq 0.8125$$
$$0.6875 \leq P(2.25) \leq 0.8125$$
$$8.125 < P(2.5) < 0.9375$$
$$1.3125 < P(3.75) < 1.4375$$

(c)



(d)

Figure 5.5: The procedure to generate a polynomial for the second sub-domain. (a) We initially sample three constraints and generate a candidate 1^{st} degree polynomial that satisfies these constraints. (b) The generated polynomial satisfies all constraints except for the input $x = 2.25$. (c) We add the constraint corresponding to the input $x = 2.25$ to the sample and generate another polynomial that satisfies the four constraints. (d) The second polynomial satisfies all constraints in the second sub-domain.

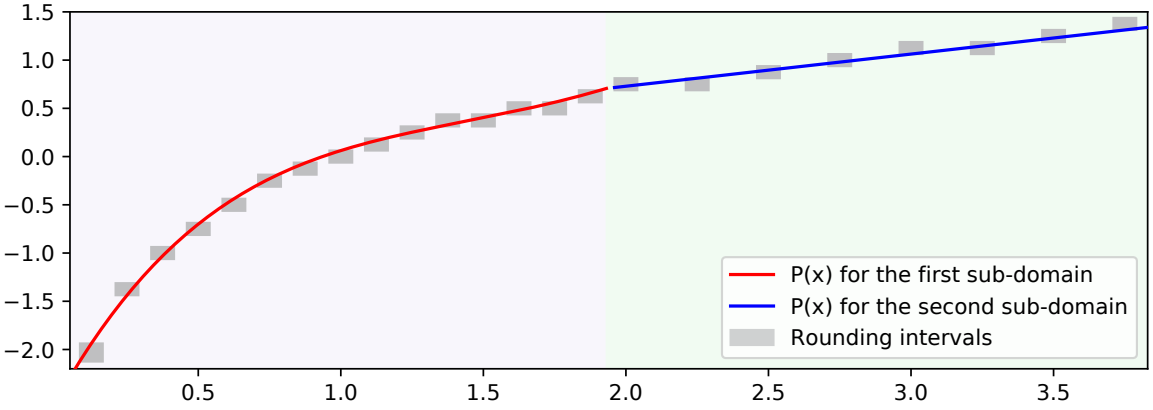


Figure 5.6: Illustration of the piecewise polynomials produced with our approach. Each polynomial produces the correctly rounded results of $ln(x)$ for all inputs in the corresponding sub-domain.

Algorithm 5.1: `CorrectPiecewise` computes piecewise polynomials of degree d. The generated polynomials produce the correctly rounded result of $f(x)$ for each input in the corresponding sub-domain when used with output compensation function. `GenPiecewise` is shown in Algorithm 5.2.

```
1  Function CorrectPiecewise(f, 𝕋, ℍ, X, d, RR_ℍ, OC_ℍ):
2      L ← CalcRndIntervals(f, 𝕋, ℍ, X)
3      if L = ∅ then return (false, DNE)
4      L' ← CalcReducIntervals(L, ℍ, RR_ℍ, OC_ℍ)
5      if L' = ∅ then return (false, DNE)
6      L* ← CombineReducIntervals(L')
7      if L* = ∅ then return (false, DNE)
       // Change from Chapter 4: Generates piecewise polynomial
8      Ψ ← GenPiecewise(L*, d)
9      return (true, Ψ)
```

$\mathbb{T}$. Algorithm 5.1 extends the RLIBM approach from Chapter 4. Given an elementary function $f(x)$ and a list of inputs X, we compute the correctly rounded result of $f(x)$ and identify the rounding interval in $\mathbb{H}$ for each result (line 2). Then, using the range reduction function and output compensation function, we identify the reduced inputs x' and reduced intervals $[l', h']$ (lines 4-7). The pair $(x', [l', h'])$ for each x' specifies a constraint on the output of the polynomial approximation that we want to generate. If the polynomial approximation produces a value in the reduced interval, then it will produce the correctly rounded result for all inputs when used with the output compensation function. Instead of generating a single polynomial, the extended RLIBM approach generates a piecewise polynomial using domain splitting and counterexample guided polynomial generation (line 8). We first iteratively split the domain of the reduced input into multiple sub-domains (Algorithm 5.2). Then, we use counterexample guided polynomial generation which uses a small portion of constraints to generate a polynomial that produces a value in the reduced interval for all inputs in the sub-domain (Algorithm 5.3). In the remainder of the section, we focus on describing our domain splitting and counterexample guided polynomial generation in more detail. We assume that we have already identified the list of reduced inputs and intervals.

5.3.1 Domain Splitting for Efficient Piecewise Polynomials

Once we compute the list of constraints L^* containing reduced inputs x' and its corresponding reduced intervals $[l', h']$, our goal is to generate polynomials that satisfy the constraints in L^*. Even after range reduction, there can be millions of constraints in L^*. Given a sufficiently high degree d, the counterexample guided polynomial generation technique which we describe in Section 5.3.2 can generate a single polynomial. However, a high degree polynomial is not efficient. Hence, we generate piecewise polynomials with lower degrees. We split the reduced input domain into sub-domains and generate a polynomial for each sub-domain. Each polynomial only needs to satisfy the constraints in the corresponding sub-domain, leading to a lower degree and better performance when evaluating the polynomial. For best performance, effectively splitting the input domain is essential. We must avoid the situation where the performance gain in evaluating lower degree polynomial is outweighed by the overhead of identifying which polynomial to evaluate for a given input. Hence, we group the reduced inputs into sub-domains based on the bit-pattern of the reduced inputs in $\mathbb{H}$.

Algorithm 5.2 illustrates our approach to generate piecewise polynomials. Abstractly, our strategy is to split the entire input domain into 2^n sub-domains and use n bits of the bit-pattern of x' to group the reduced inputs. Considering the bit-string of the reduced input x' in $\mathbb{H}$, the first bit is the sign bit and the next several bits represent the exponent bits. Additionally, range reduction reduces the input in $\mathbb{T}$ to a value in a small domain $[a, b]$. Thus, it is likely that the first several bits of the bit-string are identical for all reduced inputs. We use the next n bits to group the reduced inputs into each sub-domain.

Some range reduction techniques can create both positive and negative reduced inputs. The bit-string of positive and negative reduced inputs in $\mathbb{H}$ will not have any common bits because the first bit distinguishes between negative and positive values. Hence, we separate the reduced inputs and their corresponding reduced interval into two groups: L^- that contains negative reduced inputs, and L^+ that contains positive reduced inputs (lines 2

Algorithm 5.2: `GenPiecewise` generates piecewise polynomials that produce a value in the reduced interval for all reduced inputs in L. It initially attempts to produce a single polynomial for the entire reduced input domain. If unsuccessful, then it splits the domain into multiple sub-domains. `SplitDomain` splits the reduced input domain into sub-domains based on the bit-pattern of the reduced inputs in $\mathbb{H}$. `CeGPolyGen` generates a polynomial for each sub-domain, which is shown in Algorithm 5.3.

```
1  Function GenApproxFunc(L, d):
2      L⁻ ← {(x', [l', h']) ∈ L | x' < 0}
3      L⁺ ← {(x', [l', h']) ∈ L | x' ≥ 0}
4      Ψ⁻ ← GenApproxHelper(L⁻, d)
5      Ψ⁺ ← GenApproxHelper(L⁺, d)
6      return {Ψ⁻, Ψ⁺}
7  Function GenApproxHelper(L, d):
8      n ← 0
9      while true do
10         Δ = SplitDomain(L, n)
11         (status, Ψ) = GenPiecewise(Δ, d)
12         if status = true then return Ψ
13         n ← n + 1
14     end
15 Function GenPiecewise(Δ, d):
16     Ψ ← ∅
17     foreach Δ_i ∈ Δ do
18         (status, Ψ_i) ← CeGPolyGen(Δ_i, d)
19         if status = false then return (false, ∅)
20         Ψ ← Ψ ∪ Ψ_i
21     end
22     return (true, Ψ)
```

and 3). Then create polynomial approximations for each L^- and L^+ (lines 4 and 5).

When the list L has either positive reduced inputs (*i.e.*, $L = L^+$) or negative reduced inputs (*i.e.*, $L = L^-$), we try to generate a single polynomial of degree d that produces a value in the reduced interval for each reduced input x' in L (line 11). If we cannot generate such a polynomial, then we split the reduced input domain into sub-domains (lines 9-13).

The reduced input domain is split into 2^n sub-domains (line 10). To split the domain, we identify the smallest reduced input x'_{min} and the largest reduced input x'_{max} in L, excluding the reduced input 0. The range reduction function is evaluated in the higher precision representation $\mathbb{H}$ and reduces the entire original input domain into a smaller domain. This

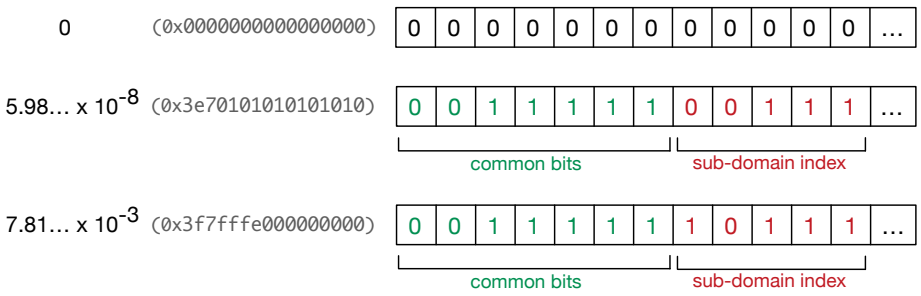


Figure 5.7: Bit-pattern of the reduced inputs 0, x'_{min}, and x'_{max} in the double type ($\mathbb{H}$) when splitting the reduced input domain of $ln(x)$ for 32-bit float ($\mathbb{T}$) after using the range reduction strategy described in Chapter 4. The first 7 bits in the bit-pattern of x'_{min} and x'_{max} are identical. To split the input domain into 2^5 sub-domains, we use the next 5 bits of the bit-pattern to group reduced inputs into each sub-domain.

may lead to a significant gap between 0 and the nearest reduced input in L. If we do not exclude 0 when splitting the domain, several sub-domains will be assigned for the reduced inputs between 0 and its nearest reduced input, resulting in empty sub-domains. Hence, when we determine the bit-pattern to split the input domain, we exclude 0 from consideration to evenly place the reduced inputs into each sub-domain.

The reduced inputs (excluding 0) in L are likely to have common bits in the sign bit and the exponent bits since the inputs are in a small domain. We identify the consecutive bits that are identical in the bit-string representation of x'_{min} and x'_{max} in the higher precision representation $\mathbb{H}$, starting from the most significant bit. Then, we use the next n bits to identify the sub-domain. We group all reduced inputs and reduced intervals that share the same bit-pattern of the n bits into the same sub-domain Δ_i, indexed by the n-bit used for sub-domain identification. The reduced input 0 is grouped into Δ_0, as the bit-pattern of 0 is composed of all 0 bits.

Illustration Consider an example where our goal is to generate polynomials approximations of $ln(x)$ for 32-bit float. Mathematically, the range reduction for $ln(x)$ described in Chapter 4 reduces the input domain into $[0, \frac{1}{128})$. When we reduce the 32-bit float inputs into reduced inputs, there is a significant gap between 0 and the nearest reduced input. Specifically, the reduced input domain is $\{0\} \cup [5.98\cdots \times 10^{-8}, 7.81\cdots \times 10^{-3}]$. Hence,

$x'_{min} = 5.98\cdots \times 10^{-8}$ and $x'_{max} = 7.81\cdots \times 10^{-3}$.

Figure 5.7 shows the bit-pattern of 0, x'_{min}, and x'_{max} in the higher precision representation, 64-bit double. There is a significant gap between 0 and x'_{min} as there are roughly 4.5×10^{18} bit-patterns between the two values. The first 7 bits in the bit-patterns of x'_{min} and x'_{max} are identical. To create 2^5 sub-domains, we use the next 5 bits in the bit-string. The reduced input 0 is then assigned to the 0^{th} sub-domain.

Once the sub-domains and the reduced inputs corresponding to each sub-domain are identified, we try to generate a polynomial of degree d that satisfies the constraints in each sub-domain Δ_i (line 18 in Algorithm 5.2). If we cannot generate a polynomial for at least one sub-domain, then we split the reduced input into 2^{n+1} sub-domains and repeat the process. Once we generate a polynomial for each sub-domain, the coefficients of the polynomials are stored in a table, indexed by the bit-pattern used to identify the sub-domain. Using bit-patterns of the reduced input in $\mathbb{H}$ allows us to efficiently identify the sub-domain given a reduced input. It requires a single masking operation and a shift operation.

5.3.2 Counterexample Guided Polynomial Generation

Even after range reduction and domain splitting, there can be millions of reduced inputs and reduced intervals in each sub-domain. With the current hardware technology, even the state-of-the-art LP solvers can only handle thousands of constraints. To address this challenge, we propose counterexample guided polynomial generation with sampling. Our insight is that it is not necessary to reason about every single constraint in the sub-domain, as illustrated in Section 5.2. As we are approximating continuous elementary functions, the reduced intervals of adjacent reduced inputs are gradually increasing or decreasing. If a polynomial satisfies the constraint for a given reduced input, then it is likely that the same polynomial satisfies the constraint for nearby inputs, except when the reduced intervals of the nearby inputs are significantly constrained. Thus, as long as we use highly constrained intervals, the generated polynomial will produce a value in the interval for all inputs in the

Algorithm 5.3: `GenPolynomial` attempts to find a polynomial of degree d that satisfies the reduced input and interval constraints in Δ_i using our counterexample guided sampling approach. If it is infeasible to find a polynomial of degree d or the size of the sample exceeds a threshold, then it returns (false, $\emptyset$). `GeneratePoly` generates the coefficients of a polynomial that satisfies all constraints in $\mathcal{S}$ using the LP solver (described in Section 3.3). `Check` validates that the polynomial generated using the sample satisfies all reduced input and interval constraints.

```
1  Function GenPolynomial(Δ_i, d):
2      S ← Sample(Δ_i)
3      while true do
4          (status, Ψ_i) ← GeneratePoly(S, ℍ, d)
5          if status = false then return (false, ∅)
6          (Done, S) ← Check(Ψ_i, Δ_i, S)
7          if Done = true then return (true, Ψ_i)
8          if |S| > threshold then return (false, ∅)
9      end
10 Function Check(Ψ_i, Δ_i, S):
11     Done ← true
12     foreach (x', [l', h']) ∈ Δ_i do
13         if not l' ≤ Ψ_i(x') ≤ h' then
14             S ← {(x', [l', h'])} ∪ S
15             Done ← false
16         end
17     end
18     return (Done, S)
```

sub-domain.

Algorithm 5.3 shows the counterexample guided polynomial generation technique. It accepts a list of reduced constraints $[x', [l', h']]$ in the sub-domain Δ_i and the degree d of the polynomial that we wish to generate. The list of constraints in Δ_i is stored in increasing order of x'. Our goal is to generate a polynomial of degree d that produces a value in the interval $[l', h']$ for each reduced input in Δ_i. We choose a small portion of constraints in Δ_i (line 2) by uniformly sampling the constraints based on the distribution of the reduced inputs. If there are a significant number of reduced inputs in a small region of the sub-domain, our process samples more inputs from that region. We also add constraints where the size of the reduced interval is smaller than a certain bound ϵ, which can be specified by the math library developer.

Next, we generate a candidate polynomial that satisfies all the constraints in the sample using the LP formulation (line 4). If it is not possible to generate a polynomial of degree d that satisfies all constraints in the sample, we split the reduced input domain into smaller sub-domains and repeat the entire process. Otherwise, we check whether the candidate polynomial satisfies all constraints in the sub-domain Δ_i (lines 12-17). If there is any constraint not satisfied by the polynomial, then the constraints are added to the sample (lines 13-15). We repeat the process of generating candidate polynomials until the polynomial satisfies all constraints in Δ_i (line 7). At any point, if the number of constraints in the sample exceeds a certain threshold, we determine that we cannot generate a polynomial of degree d (line 8).

5.3.3 Storing the Coefficients of Piecewise Polynomials

At the end of our process, we obtain two piecewise polynomials, Ψ^- that produces the correctly rounded results of all inputs that reduce to negative reduced inputs and Ψ^+ that produces the correctly rounded results of all inputs that reduce to positive reduced inputs. The two piecewise polynomials are stored in a separate table.

Each polynomial Ψ_i in a piecewise polynomial Ψ (*i.e.*, $\Psi = \Psi^-$ or $\Psi = \Psi^+$) is indexed using the bit-pattern, i, used to group reduced inputs to the corresponding sub-domain. The polynomial has a degree of d or smaller,

$$\Psi_i = c_{i,0} + c_{i,1}x + c_{i,2}x^2 + \cdots + c_{i,d}x^d$$

If Ψ_i has a degree $d' < d$, then all coefficients after the d'^{th} term are zeros. If there are no reduced inputs in a particular sub-domain with index i, then all coefficients in Ψ_i are zeros. If there is a term $c_{i,j}x^j$ where $c_{i,j} = 0$ across all polynomials in Ψ, *i.e.*, when polynomial approximations are even or odd, then we remove the term to avoid unnecessary overhead when evaluating the polynomial. Finally, the coefficients of each Ψ_i in Ψ are stored in a two-dimensional look-up table. The coefficients corresponding to each $\Psi_i$in Ψ are indexed with i. Each coefficient $c_{i,j}$ in Ψ_i is indexed with j. Using this strategy, we can efficiently

retrieve the coefficients of a polynomial Ψ_i and use the same implementation to evaluate different Ψ_i's.

5.3.4 Implementing the Elementary Function

We now describe how to implement the elementary function of $f(x)$ that produces the correctly rounded result for all inputs in $\mathbb{T}$ using the generated piecewise polynomials Ψ^- and Ψ^+, the range reduction function $RR(x)$, and the output compensation function $OC(y')$. All internal computations are performed in the higher precision representation $\mathbb{H}$. Given an input $x \in \mathbb{T}$, we determine whether x is a special case input, which can be implemented using a series of branch statements. If x is a special case input, then we return the corresponding result. Next, we convert the input x to $\mathbb{H}$ and perform range reduction $x' = RR(x)$ to obtain the reduced input. We identify the sign of x' to determine which piecewise polynomial to use. If $x' < 0$, then we retrieve the look-up table for Ψ^-. Otherwise, we retrieve the look-up table for Ψ^+.

Once we retrieve the look-up table for the correct piecewise polynomial Ψ for x', our next goal is to retrieve the polynomial that corresponds to x' by identifying the index of the sub-domain that x' belongs to. At implementation time, we already know the size (*i.e.*, 2^n) of the piecewise polynomial Ψ. We use the strategy described in Section 5.3.1 to identify the n-bit bit-pattern of x' in $\mathbb{H}$. This process can be efficiently implemented using a single bitwise `and` operation and a `shift right` operation. We use this bit-pattern to index into the look-up table and retrieve the coefficients of the polynomial Ψ_i. We evaluate $y' = \Psi_i(x')$ and then use y' to evaluate the output compensation function $y = OC(y')$. The value y is in the higher precision representation $\mathbb{H}$. Thus we round y to $\mathbb{T}$, which is the correctly rounded result of $f(x)$ in the representation $\mathbb{T}$.

5.4 Summary

In this chapter, we present our approach for generating piecewise polynomials to create efficient and correctly rounded implementations. This approach is aimed towards generating correctly rounded polynomial approximations for 32-bit representations. We propose two extensions to the RLIBM approach. First, we split the input domain into multiple sub-domains using bit-patterns in the input and generate a polynomial for each sub-domain. Second, we use counterexample guided polynomial generation to handle millions of inputs. Our approach has been instrumental in generating elementary functions for 32-bit float and posit32. Our functions produce correctly rounded results for all inputs in 32-bit representations and are faster than mainstream math libraries.

CHAPTER 6
A SINGLE POLYNOMIAL THAT PRODUCES CORRECT RESULTS FOR MULTIPLE REPRESENTATIONS AND ROUNDING MODES

The previous chapters describe our approach to generate correctly rounded polynomial approximations for a specific FP representation and a single rounding mode. The generated polynomials do not guarantee to produce the correctly rounded results for a different representation or even a different rounding mode, due to double rounding error. In this chapter, we propose a novel approach to generate polynomial approximations that produce correctly rounded results for multiple representations and rounding modes. Our key idea is to approximate the correctly rounded result for an $(n+2)$-bit representation using the *round to odd* (rno) rounding mode. The rno rounding mode has properties that avoid double rounding issues. We formally prove that the resulting polynomial approximation will produce correctly rounded results for multiple representations and all standard rounding modes.

6.1 Case For Generic Math Libraries

Using our approaches described in the previous chapters, we have been successful in generating polynomial approximations of $f(x)$ that produce correctly rounded results for a given representation and a rounding mode. Since the posit standard specifies one rounding mode, round to the nearest tie goes to even (*i.e.*, rne), one polynomial approximation suffices for each given representation. In the IEEE-754 standard for FP representations, there are five standard rounding modes, rne, rna, rnz, rnp, and rnn mode. To produce correctly rounded results of $f(x)$ for different rounding modes, we can generate one polynomial approximation for each rounding mode. Five approximation functions for a given elementary function and an FP representation are reasonable to generate, especially if we consider only a few widely used representations.

Recently, new variations of FP with smaller bit-length have been proposed with varying amount of dynamic range and precision, including bfloat16 [76], FlexPoint [78], tensorfloat32 [108], MSFP [100, 116], log number systems [44, 74, 112, 130], and the Posit representation [55, 58]. Some of these variants are different configurations of FP while some others (*i.e.*, posits) are completely new number systems. All of these representations require math libraries to approximate elementary functions if we are to use them in scientific applications. Generating math libraries for multiple variants of representations requires significant effort and time, especially for different rounding modes.

Our goal is to generate a single approximation of an elementary function that produces correctly rounded results for multiple representations and rounding modes. The primary challenge is in identifying the values that the polynomial approximations should produce. For example, consider the task of creating a math library for the float representation that produces correctly rounded results with all rounding modes. A naive attempt at accomplishing this task may use a correctly rounded double math library such as CR-LIBM which produces the correctly rounded result in the double type with rne rounding mode. The double result can be rounded to the float type using the chosen rounding mode. Due to the double rounding error, however, the final result will not be equal to the correctly rounded result of $f(x)$ for float for all inputs. If the real value of $f(x)$ is extremely close to the rounding boundary of two adjacent float values, then the error caused by rounding $f(x)$ to double using rne rounding mode can be significant enough to produce the wrong result for float. We empirically show that this approach of using a correctly rounded math library for double indeed produces wrong float results in Chapter 7.

Our key idea. Instead of generating a correctly rounded elementary function for each representation and each rounding mode, it would be ideal to generate a single polynomial approximation that produces the correctly rounded result of an elementary function for multiple representations and rounding modes. We propose a novel approach to generate

such polynomials! Our key insight is to approximate the correctly rounded result of an elementary function $f(x)$ in the *round to odd* (rno) rounding mode. The rno rounding mode has previously been explored to avoid the double rounding error when performing primitive operations in extended precision and then subsequently rounding the result to the smaller precision representation [9]. The rno mode can be described as follows. If a real value is exactly representable in the target representation, it is rounded to the value in the representation. Otherwise, rno rounds the real value to the nearest value in the target representation where the bit-string is odd when interpreted as an unsigned integer. Abstractly, the rno mode avoids double rounding error because it retains all the information necessary to identify the correctly rounded value of the original real value. Our contribution lies in recognizing that the rno mode has the properties which can be used to generate correctly rounded elementary functions for multiple representations and rounding modes. We provide more information on the rno mode in Section 6.3.1.

Our approach generates polynomial approximations that produce the correctly rounded result of $f(x)$ in the $(n+2)$-bit representation $\mathbb{T}_{n+2}$ with the rno mode. This polynomial approximation produces the correctly rounded result for all k bit representation $\mathbb{T}_k$ with any standard rounding modes, as long as $k \leq n$ and the number of exponent bits of $\mathbb{T}_k$ is the same as $\mathbb{T}_{n+2}$. We formally define the $\mathbb{T}_{n+2}$ and $\mathbb{T}_k$ for FP and posit representations in Section 6.3 and prove that the polynomial that we generate produces the correctly rounded results for $\mathbb{T}_k$ with all standard rounding modes in Section 6.4.

High-level overview of our approach. To generate the polynomial approximation that produces the correctly rounded result of $f(x)$ in $\mathbb{T}_{n+2}$ with the rno mode, we extend the approach described in the previous chapters. We first compute the correctly rounded result of $f(x)$ in $\mathbb{T}_{n+2}$ representation with the rno mode using an oracle to produce y_{rno}. Using the correctly rounded result, we identify a range of values that round to y_{rno} when rounded to $\mathbb{T}_{n+2}$ with the rno mode. We call this interval the odd interval. When a polynomial

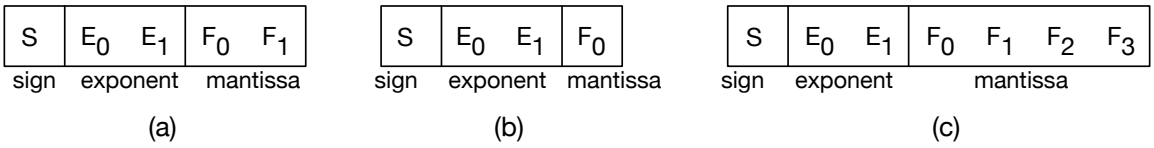


Figure 6.1: (a) Layout of the bit-pattern of the (a) 5-bit FP5, (b) 4-bit FP4, and (c) 7-bit FP7 used in our illustration. All three representations have two exponent bits.

generates a value in the odd interval, then the polynomial is guaranteed to produce the correctly rounded result of $f(x)$ for all $\mathbb{T}_k$ using any standard rounding modes. Next, our goal is to generate a polynomial that produces a value in the odd interval for each input. One of the challenges in generating the polynomial is the existence of singleton odd intervals. This can occur when the real value of $f(x)$ for a particular input x is exactly representable in $\mathbb{T}_{n+2}$ and the bit-string of the value in $\mathbb{T}_{n+2}$ is even when interpreted as an unsigned integer. Because generating a polynomial that produces a singleton value for a particular input is challenging, we use mathematical properties to identify the set of inputs resulting in a singleton interval and develop an efficient implementation to compute the result. We generate a polynomial that produces a value in the odd interval for the remaining inputs and non-singleton odd intervals.

Using this approach, we created a math library with ten elementary functions for FP and ten elementary functions for posit, which we call RLIBM-ALL. The FP functions in RLIBM-ALL produce the correctly rounded result in the 34-bit FP representation, $\mathbb{F}_{34,8}$, with the rno mode. They produce the correct results for all FP representations ranging from the 10-bit $\mathbb{F}_{10,8}$ to the 32-bit $\mathbb{F}_{32,8}$ for all standard IEEE-754 rounding modes. These representations include bfloat16, TensorFloat32, and float types. The posit functions in RLIBM-ALL produce the correctly rounded result in the 34-bit posit representation, $\mathbb{P}_{34,2}$, with the rno mode. They produce the correct results for all posit representations ranging from the 2-bit $\mathbb{P}_{2,2}$ to the 32-bit $\mathbb{P}_{32,2}$, the standard 32-bit posit configuration.

6.2 Illustration

We provide an end-to-end example of creating a single polynomial that produces the correctly rounded results of $ln(x)$ for a 5-bit FP with 2 exponent bits (FP5) and a 4-bit FP with 2 exponent bits (FP4) using the standard rounding modes (*i.e.*, rne, rna, rnz, rnp, and rnn). Figure 6.1(a) and (b) shows the layout of the bit-string of FP5 values and FP4 values, respectively. We present our example with FP5 and FP4 to illustrate our approach and describe the insights. In practice, creating a look-up table with pre-computed results is more beneficial for FP5 and FP4 as there are only 32 and 16 bit-patterns, respectively.

The $ln(x)$ function is defined over the input domain $(0, \infty)$. There are 11 values ranging from 0.25 to 3.5 in FP5 within the input domain. The remaining $2^5 - 11 = 21$ value are special case inputs. Similarly in FP4, there are 5 values ranging from 0.5 to 3.0 within the input domain. The remaining $2^4 - 5 = 11$ values are special case inputs. The special cases can be filtered with a simple check followed by returning the correct result.

6.2.1 A Strawman Approach

Before presenting our approach, we present a strawman approach to generate a polynomial for FP4 and FP5. The strawman approach highlights the challenges of generating polynomial approximations for multiple representations and rounding modes. Let us pick an input $x = 1.5$ that can be represented by both FP5 and FP4. Figure 6.2 shows the real value of $ln(1.5)$ (gray star) and the correctly rounded result for FP5 and FP4 with different rounding modes. Consider the first row in Figure 6.2, which shows the correctly rounded result of FP5 with the rne mode (black circle). To produce this value, our polynomial approximation has to produce a value in the rounding interval of $ln(1.5)$, for the rne result. The second row in Figure 6.2 shows the correctly rounded result of FP5 with the rna mode and the rounding interval to produce the correctly rounded rna result. The subsequent rows show the correctly rounded result and the rounding interval for other rounding modes of

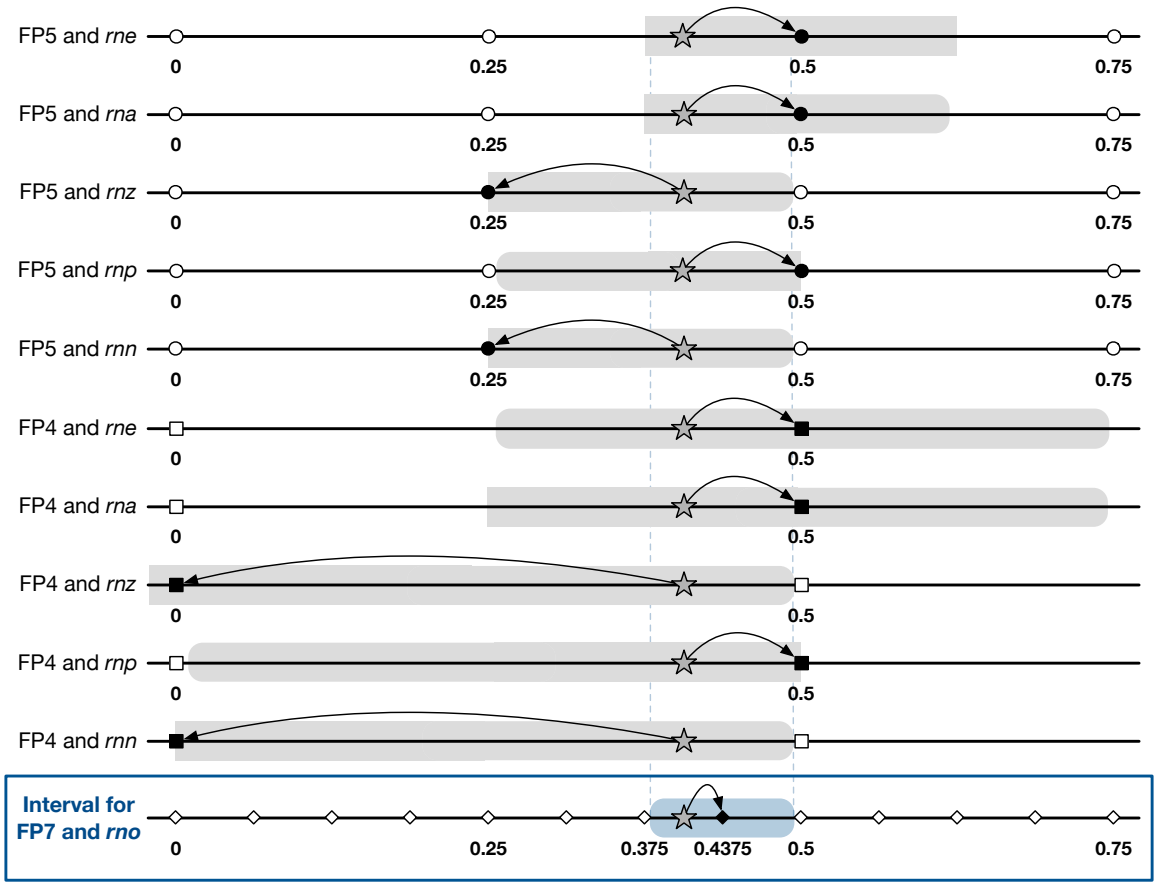


Figure 6.2: We show the correctly rounded result of $ln(1.5)$ in FP5 and FP4 with all five rounding modes. The gray star represents the real value of $ln(1.5)$. Values that are representable in FP5, FP4, and FP7 are shown with circle, square, and rhombus, respectively. The black solid color represents the correctly rounded result and the gray interval shows the rounding interval. The last row shows the odd interval where all values in the interval rounds to the correctly rounded results of $ln(1.5)$ in FP7 with the rno rounding. The odd interval is a subset of all rounding intervals for FP5 and FP4 with five standard rounding modes.

FP5 and FP4. It is possible for a single polynomial to produce the correctly rounded result of $ln(1.5)$ for both FP5 and FP4 with all five rounding modes if it produces a value within all ten rounding intervals. A strawman approach for generating a generic polynomial is to compute the rounding intervals for all possible combinations of target representations and rounding modes and identify the common region in all rounding intervals. However, this requires a significant amount of computation especially as the number of target representations increase.

6.2.2 Generating A Polynomial Approximation With the rno Mode.

To create a single polynomial, we generate a polynomial that produces the correctly rounded results in FP7 with the *round to odd* (rno) mode, where FP7 is a 7-bit FP representation with two exponent bits (Figure 6.1(c)). The rno rounding mode can be summarized as follows. If a real value is exactly representable in FP7, then we represent it with the FP7 value. Otherwise, the real value rounds to an adjacent FP7 value where the bit-string is odd when interpreted as an unsigned integer (*i.e.*, the last bit is 1). The rno mode is a non-standard rounding mode proposed to address double rounding issues with primitive arithmetic operations when using rne rounding mode [9]. We show that rno can be used to generate correctly rounded results with all standard rounding modes. In Section 6.4, we prove that a polynomial that produces the correctly rounded results for a $(n+2)$-bit representation with the rno mode also produces the correctly rounded results for all k bit representations with any standard rounding modes as long as the representation has the same number of exponent bits $|E|$ and $|E|+1 < k \leq n$. In our illustration, all three representations (*i.e.*, FP4, FP5, and FP7) have two exponent bits. When we generate a polynomial that produces correctly rounded results of $ln(x)$ in FP7 with the rno mode, the result of the polynomial will round to the correctly rounded result of FP5 and FP4 for all rounding modes.

Additionally, computing the correctly rounded result of $ln(x)$ in FP7 with the rno mode and identifying the range of values that round to this result is an efficient way to compute the common interval described above. The last row in Figure 6.2 shows the correctly rounded result of $ln(1.5)$ in FP7 with the rno mode (black rhombus) and the interval of values that round to the correctly rounded result (blue region). We call this interval the odd interval. The odd interval for $ln(1.5)$ is common in all ten rounding intervals shown above. Thus, as long as we generate a polynomial that produces a value in the odd interval for each input, the polynomial will produce the correctly rounded result of $ln(x)$ for both FP5 and FP4 with any five rounding modes.

Our approach works because the FP7 representation can represent three additional val-

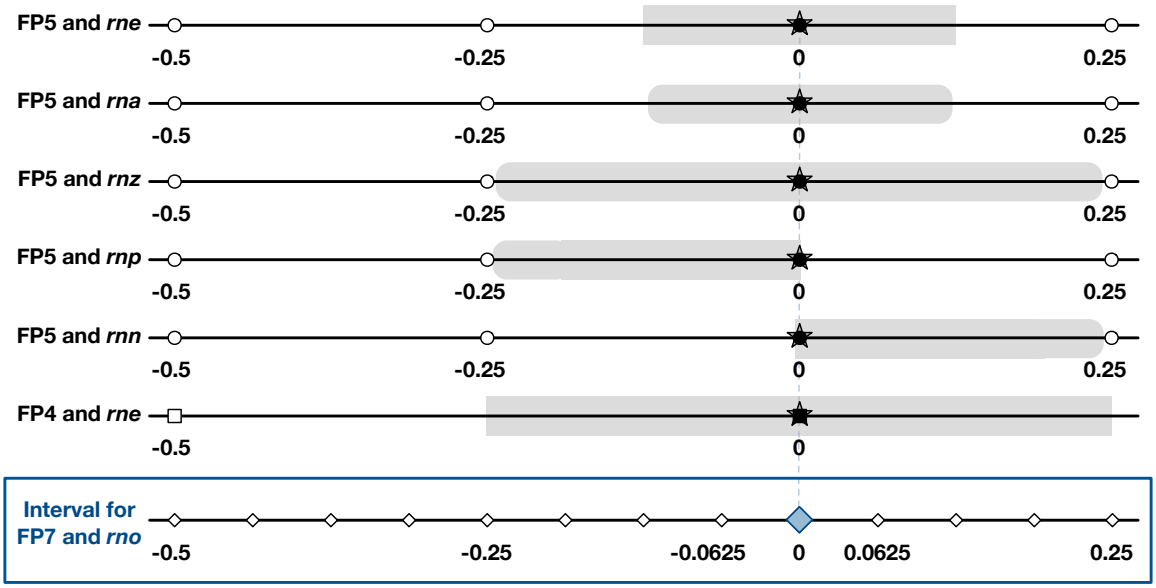


Figure 6.3: We show the correctly rounded result of $ln(1.0)$ in FP5 and FP4 with all five rounding modes. The gray star represents the real value of $ln(1.0)$. Values that are representable in FP5, FP4, and FP7 are shown with circle, square, and rhombus, respectively. The black solid color represents the correctly rounded result and the gray interval shows the rounding interval. The last row shows the odd interval (a singleton value) where all values in the interval rounds to the correctly rounded results of $ln(1.0)$ in FP7 with the rno mode. The odd interval is a subset of all rounding intervals for FP5 and FP4 with five standard rounding modes.

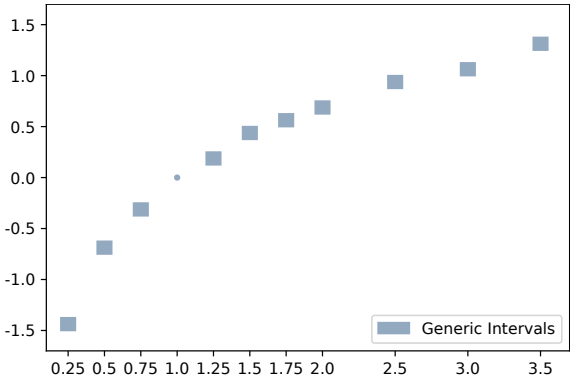
ues between two adjacent FP5 values and the rno result in FP7 holds enough information to produce the correctly rounded result of various representations (including FP4 and FP5) and rounding modes (including the five standard rounding modes). Consequently, the odd interval is typically smaller than the common intervals among FP4 and FP5 with all rounding modes. For instance, the common intervals for $ln(1.5)$ illustrated in Figure 6.2 is $[0.375, 0.5)$ while the odd interval for $ln(1.5)$ is $(0.375, 0.5)$. We provide more information regarding the rno mode in Section 6.3.1. The proof in Section 6.4 shows that our approach with the rno mode applies in general for any representation with n bits.

6.2.3 Generating Generic Polynomial Approximations

To generate a polynomial approximation for FP4 and FP5, we compute the correctly rounded result of $ln(x)$ in FP7 with the rno mode for each input in FP5 and identify the odd intervals. Since all values in FP4 are representable in FP5, it is sufficient to compute the odd

intervals for the inputs in FP5. In our illustration, we evaluate the polynomials with double precision. Hence, we identify the odd interval in the double precision. If the bit-string of the correctly rounded result of $ln(x)$ in FP7 with the rno mode is even when interpreted as an unsigned integer, then the odd interval is a singleton consisting of the correctly rounded result itself. For example, the odd interval of $ln(1.0) = 0$ is a singleton value because 0 is exactly representable in FP7 and the bit-string is even. We pictorially show the rounding intervals of $ln(1.0)$ for FP4 and FP5 with different rounding modes as well as the odd interval for FP7 in Figure 6.3. If the correctly rounded result of $ln(x)$ in FP7 with the rno mode is odd, then we can identify the odd interval as follows. We identify the preceding value l and the succeeding value h of the correctly rounded result in FP7. Then, the open interval (l, h) is the odd interval in real numbers. Since we evaluate the polynomial with double precision, we identify the range of values in double within the odd interval. We compute the succeeding value of l in double, which we denote as l^+, and the preceding value of h in double, which we denote as h^-. The closed interval $[l^+, h^-]$ is the odd interval in double for the input x. Figure 6.4(a) shows the odd intervals for each input in FP5.

In some instances, such as $ln(1.0) = 0$, the odd interval may be composed of a single value. Generating a polynomial that produces a single value, while also producing a value in the odd interval for all other inputs, is a challenging problem. This is especially true considering that the polynomial is evaluated in double, a finite precision representation. Hence, we treat these inputs similarly to special case inputs and exclude them from the list of inputs that we must approximate. For the $ln(x)$ function in FP7, there is only one input $x = 1.0$ with singleton odd interval. Other elementary functions for higher precision representations may have multiple inputs with singleton odd intervals. In Section 6.3.7, we present our approach to identify such inputs and efficiently compute the correctly rounded results using mathematical properties of the elementary functions. Figure 6.4(b) shows the final list of ten inputs and the corresponding odd interval, excluding the input with singleton interval, that our polynomial must approximate.



(a)

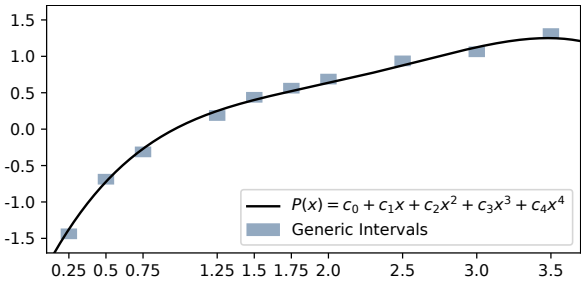
$$
\begin{aligned}
-1.5 &< P(0.25) < -1.375 \\
-0.75 &< P(0.5) < -0.625 \\
-0.375 &< P(0.75) < -0.25 \\
0.125 &< P(1.25) < 0.25 \\
0.375 &< P(1.5) < 0.5 \\
0.5 &< P(1.75) \leq 0.625 \\
0.625 &< P(2.0) < 0.75 \\
0.875 &< P(2.5) < 1.0 \\
1.0 &< P(3.0) \leq 1.125 \\
1.25 &< P(3.5) < 1.375
\end{aligned}
$$

(b)

Figure 6.4: (a) shows the odd interval for each input in FP7. The odd interval for the input $x = 1.0$ is a singleton interval, $[0, 0]$. (b) The set of constraints that must be satisfied by our polynomial to produce the correctly rounded results of $ln(x)$ for FP7 with rno mode. Note that the case for input $x = 1.0$ is missing because generating a polynomial approximation that produces $P(1.0) = 0.0$ is challenging. Thus, we classify the input 1.0 as a special case.

$$
\begin{aligned}
P(x) &= c_0 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4 \\
c_0 &= -2.2907925\ldots \\
c_1 &= 4.2825369\ldots \\
c_2 &= -2.6750194\ldots \\
c_3 &= 8.1274281\ldots \\
c_4 &= -9.0132090\ldots
\end{aligned}
$$

(a)



(b)

Figure 6.5: (a) The coefficients generated by the LP solver for the polynomial that produces correctly rounded results of $ln(x)$ for all inputs in FP7 with rno mode. (b) Generated polynomial produces a value in the odd interval for all inputs.

Our next step is to generate a polynomial that produces a value in the odd interval for all inputs. Since there are only 10 non-special case inputs, we can use the approach described in Chapter 3 and frame the problem of identifying a polynomial that produces a value in the odd interval as an LP problem and use an LP solver to find a 4^{th} degree polynomial. Figure 6.5(a) shows the resulting polynomial and the coefficients generated by the LP solver. Since the polynomial produces a value in the odd interval (illustrated in

Figure 6.5(b)), the result will round to the correctly rounded result of $ln(x)$ for FP4 and FP5 with all five standard rounding modes. To generate polynomial approximations for larger representations, we can combine this approach with range reduction, domain splitting, and counterexample guided polynomial generation described in Chapter 4 and Chapter 5.

6.3 The RLIBM Approach to Generate Generic Polynomials

We now formally define the problem. Let $\mathbb{T}_n$ be either an n-bit FP ($\mathbb{T}_n = \mathbb{F}_{n,|E|}$) or posit ($\mathbb{T}_n = \mathbb{P}_{n,es}$) representation. Let $\mathbb{T}_k$ be a representation where $\mathbb{T}_k$ has no more precision bits compared to $\mathbb{T}_n$ with the same number of exponent bits. Specifically, $\mathbb{T}_k = \mathbb{F}_{k,|E|}$ where $|E|+1 < k \leq n$ if $\mathbb{T}_n$ is a FP representation and $\mathbb{T}_k = \mathbb{P}_{k,|E|}$ where $2 \leq k \leq n$ if $\mathbb{T}_n$ is a posit representation. Note that all values exactly representable in $\mathbb{T}_k$ are also exactly representable in $\mathbb{T}_n$ (*i.e.*, $\mathbb{T}_k \subseteq \mathbb{T}_n$) for both FP and posit representation. Finally, we define rm to be a standard rounding mode $rm \in \{rne, rna, rnz, rnp, rnn\}$. For exposition, the posit rounding mode can be considered as the rne mode. Our goal is to generate a polynomial approximation $P(x)$ of an elementary function $f(x)$ that produces correctly rounded results for all inputs in any representation $\mathbb{T}_k$ and any rounding mode rm. Specifically, rounding the result of $P(x)$ to any representation $\mathbb{T}_k$ with rm rounding mode must result in the same value as computing $f(x)$ in real numbers and rounding the result to $\mathbb{T}_k$ with rm rounding mode, for all inputs in $\mathbb{T}_k$,

$$RN_{\mathbb{T}_k,rm}(P(x)) = RN_{\mathbb{T}_k,rm}(f(x))$$

To produce the correctly rounded results for $\mathbb{T}_k$ with rm rounding mode, our approach approximates the correctly rounded result of $f(x)$ in the $\mathbb{T}_{n+2}$ representation with the *round to odd* (rno) rounding mode. Depending on the target representation, $\mathbb{T}_{n+2}$ is defined as follows. If $\mathbb{T}_k$ is an FP representation, then $\mathbb{T}_{n+2} = \mathbb{F}_{n+2,|E|}$ is an $(n+2)$-bit FP representation with $|E|$ bits for the exponent. If $\mathbb{T}_k$ is a posit representation, then $\mathbb{T}_{n+2} = \mathbb{P}_{n+2,es}$ is an $(n+2)$-bit posit representation with at most es exponent bits. The key intuition

is that the result in $\mathbb{T}_{n+2}$ with the rno mode maintains a sufficient amount of information to identify the correctly rounded result of $f(x)$ in $\mathbb{T}_k$ with any rounding mode. Section 6.4 formally proves that a polynomial that produces the correctly rounded result of $f(x)$ in $\mathbb{T}_{n+2}$ with the rno mode will also produce the correctly rounded result for all $\mathbb{T}_k$ using any rm.

To generate such a polynomial, we extend the approach described in the previous chapters to work for the rno mode. We compute the correctly rounded result of $f(x)$ in $\mathbb{T}_{n+2}$ with the rno mode for each input in $\mathbb{T}_n$. Next, we identify a range of values that round to the correctly rounded result when they are rounded to $\mathbb{T}_{n+2}$ using the rno mode. We call this range of values the odd interval. Each input in $\mathbb{T}_n$ and its corresponding odd interval defines the constraint on the output of the polynomial we wish to generate. Hence, we can use the approach described in Chapter 3 to generate polynomials that satisfy these constraints by framing the problem as an LP problem and using an LP solver to identify the coefficients of the polynomial.

6.3.1 The Round to Odd (rno) Rounding Mode

As we make a case for approximating $f(x)$ for $\mathbb{T}_{n+2}$ with the rno mode, we now define the rno mode and describe how to systematically round a real value with the rno mode using the rounding components described in Chapter 2. We also provide a more detailed intuition on why rounding the rno result with $\mathbb{T}_{n+2}$ produces the correctly rounded result in $\mathbb{T}_k$ with any standard rounding modes.

The *round to odd* rounding mode is a non-standard rounding mode proposed to address double rounding errors in primitive operations [9]. Previously, it was explored specifically for the rne mode for computation with the extended precision and then rounding the result to the target representation. Given a real value $v_{\mathbb{R}}$, the rno mode rounds $v_{\mathbb{R}}$ to the target representation $\mathbb{T}$ using the following strategy. If $v_{\mathbb{R}}$ is exactly representable with a value in $\mathbb{T}$, *i.e.*, $v \in \mathbb{T}$, then $v_{\mathbb{R}}$ rounds to v. Otherwise, $v_{\mathbb{R}}$ rounds to the nearest value in $\mathbb{T}$

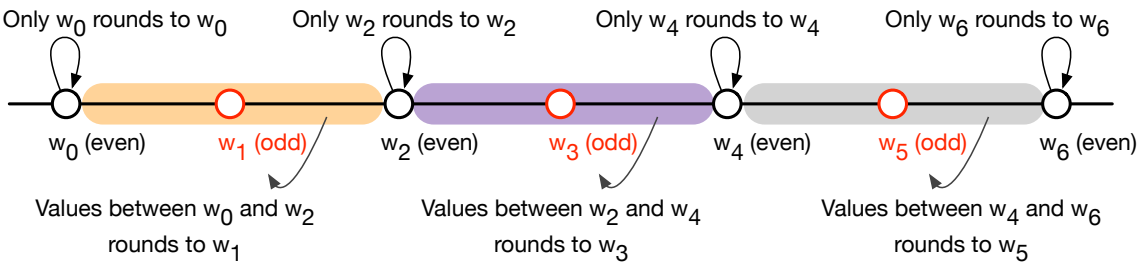


Figure 6.6: We show the rounding of real values $v_{\mathbb{R}}$ to a representation $\mathbb{T}$ with the rno mode. The values $w_0, w_1, \ldots, w_6$ are adjacent values in $\mathbb{T}$. The values highlighted with red color (*i.e.*, w_1, w_3, and w_5) represent values where the bit-string representation is odd when interpreted as unsigned integer. The orange, purple, and gray boxes represent ranges of real values that round to w_1, w_3, and w_5, respectively.

where the bit-string of the value is odd when interpreted as an unsigned integer. Figure 6.6 pictorially describes the rno mode. The values $w_0, w_1, \ldots, w_6$ are adjacent values in $\mathbb{T}$. The values highlighted with red color (*i.e.*, w_1, w_3, and w_5) represent values with odd bit-string. Similarly, the values highlighted with black color (*i.e.*, w_0, w_2, w_4, and w_6) represent values with the even bit-string. The only real values that round to w_0, w_2, w_4, or w_6 are the values w_0, w_2, w_4, and w_6 themselves, respectively. Otherwise, the real value rounds to the nearest value in $\mathbb{T}$ where the bit-string is odd (*i.e.*, w_1, w_3, and w_5). Based on the definition of rno, we can systematically round $v_{\mathbb{R}}$ to $\mathbb{T}$ with the rno mode using the rounding components $(s, v^-, rb, sticky)$,

$$v_{rno} = RN_{\mathbb{T},rno}(v_{\mathbb{R}}) = \begin{cases} s \times v^- & \text{if } IsOdd(v^-) \vee (rb = 0 \wedge sticky = 0) \\ s \times v^+ & otherwise \end{cases}$$

where v^+ is the succeeding value of v^- in $\mathbb{T}$. The definition of the rno mode is the same for both FP and posit representations.

Our contribution. Our contribution is to use the rno mode to generate correctly rounded elementary functions for multiple representations and rounding modes. Specifically, a polynomial that produces the correctly rounded result of $f(x)$ in $\mathbb{T}_{n+2}$ with the rno mode will also produce the correctly rounded result of $f(x)$ in $\mathbb{T}_k$ with any standard rounding modes.

Theorem 6.1. *Let $f(x)$ be the real value of an elementary function given an input x and $v_{rno} = RN_{\mathbb{T}_{n+2},rno}(f(x))$. Let v be a value in the odd interval of v_{rno} such that for all v, $RN_{\mathbb{T}_{n+2},rno}(v) = v_{rno}$. Choose a rounding mode $rm \in \{rne, rna, rnz, rnp, rnn\}$. Then,*

$$RN_{\mathbb{T}_k,rm}(v) = RN_{\mathbb{T}_k,rm}(f(x))$$

We also provide an efficient procedure to create a polynomial approximation that produces a value in the odd interval for each input x. By Theorem 6.1, rounding any value v in the odd interval to $\mathbb{T}_k$ using a rounding mode rm is guaranteed to produce the correctly rounded result of $f(x)$ in $\mathbb{T}_k$ with the same rounding mode rm.

6.3.2 Why Does The rno Result Avoid Double Rounding Error?

Let us provide an intuition of why rounding with the rno mode avoids double rounding error in Figure 6.7. Any values v_0 and v_1 that is representable in $\mathbb{T}_n$ is exactly representable in $\mathbb{T}_{n+2}$ (*i.e.*, w_0 and w_4). Further, there are three additional values (*i.e.*, w_1, w_2, and w_3) in $\mathbb{T}_{n+2}$ between w_0 and w_4. The value w_2 is the midpoint between w_0 and w_4. Similarly, the value w_1 is the midpoint between w_0 and w_2 while w_3 is the midpoint between w_2 and w_4. In the rno mode, all values between w_0 and w_2 rounds to w_1. Similarly, any value between w_2 and w_4 rounds to w_3. Figure 6.7(a) illustrates the result of rounding a real value (red star) directly to $\mathbb{T}_n$ with the rne mode (solid black arrow) and the result of double rounding by first rounding the real value to $\mathbb{T}_{n+2}$ with the rno mode and subsequently rounding the result to $\mathbb{T}_n$ using the rne mode (blue dotted arrow). It can be seen, double rounding with the rno mode produces the same value as if directly rounding the real value to $\mathbb{T}_n$ using the rne mode. Similarly, Figure 6.7(b) through (e) illustrates that double rounding with rno produces the same value as rounding the real value directly to $\mathbb{T}_n$ using other four standard rounding modes.

To round a real value $v_{\mathbb{R}}$ to $\mathbb{T}_n$ using one of the five standard rounding modes, we must identify the relationship between $v_{\mathbb{R}}$ and the two adjacent values v_0 and v_1 in $\mathbb{T}_n$, as explained in Chapter 2. There are five categories of such relationships:

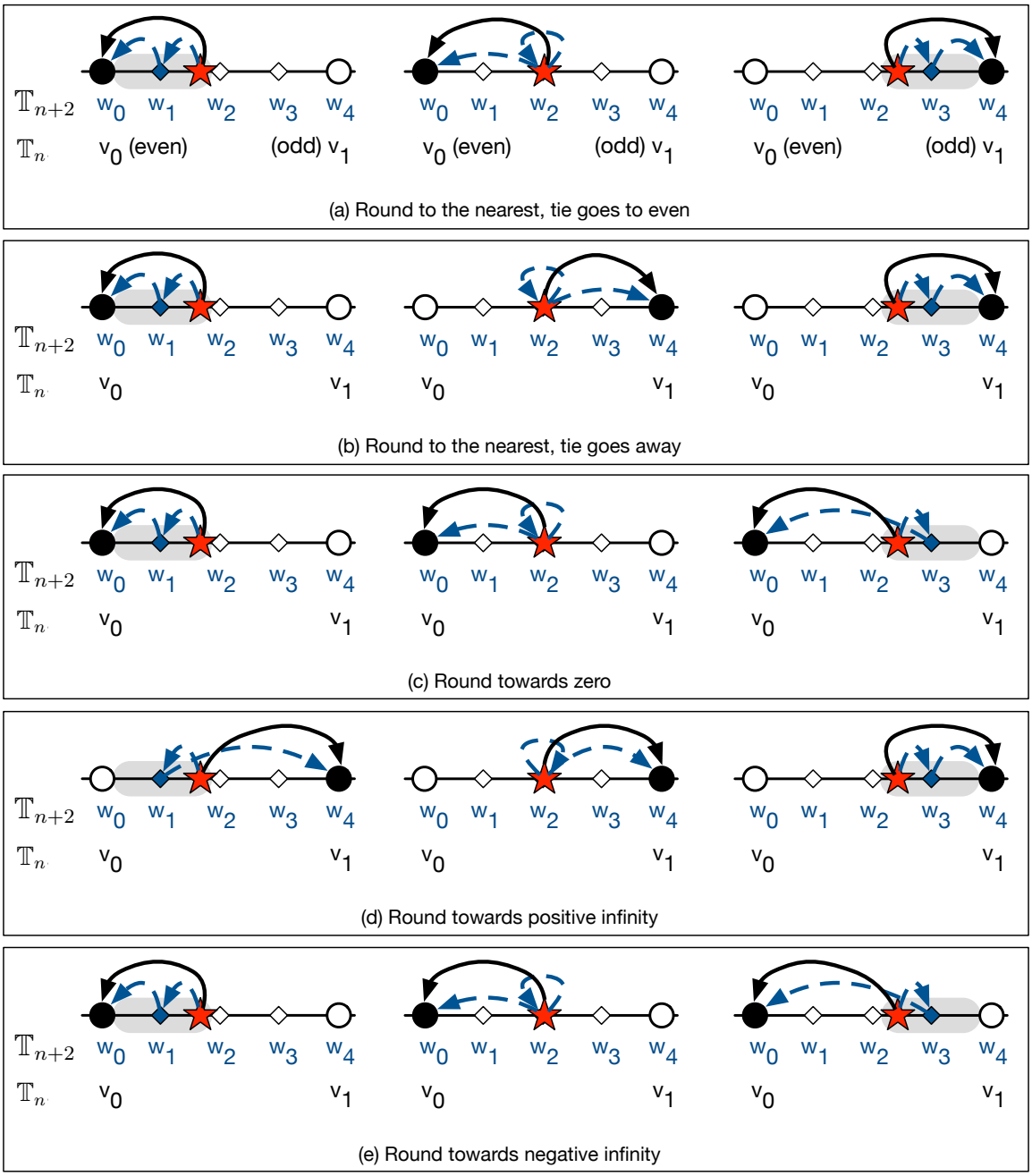


Figure 6.7: Illustration to show that rno result in $\mathbb{T}_{n+2}$ maintains sufficient information to produce the correctly rounded result in $\mathbb{T}_n$ with the (a) rne, (b) rna, (c) rnz, (d) rnp, and (e) rnn mode. The real value is represented with a red star. Values v_0 and v_1 are two adjacent values in $\mathbb{T}_n$. $\mathbb{T}_{n+2}$ can exactly represent v_0 and v_1 (*i.e.*, w_0 and w_4, respectively). Additionally, $\mathbb{T}_{n+2}$ can represent three more values between v_0 and v_1 (*i.e.*, w_1, w_2, and w_3). The dotted blue arrows show the double rounding from a real value to $\mathbb{T}_{n+2}$ with rno mode and then subsequently rounding the result to $\mathbb{T}_n$. Sold black arrow shows the result of directly rounded the real value to $\mathbb{T}_n$.

- $v_{\mathbb{R}} = v_0$.
- $v_0 < v_{\mathbb{R}} < w_2$.
- $v_{\mathbb{R}} = w_2$.
- $w_2 < v_{\mathbb{R}} < v_1$.
- $v_{\mathbb{R}} = v_1$.

where w_2 represents the midpoint between v_0 and v_1. Double rounding error occurs when the intermediate rounded result loses the information regarding the relationships between $v_{\mathbb{R}}$, v_0, and v_1. However, rounding $v_{\mathbb{R}}$ to $\mathbb{T}_{n+2}$ using the rno mode preserves this relationship. All values between v_0 and w_2 rounds to w_1, which is still a value between v_0 and w_2. A real value that is exactly equal to w_2 is rounded to w_2. All values between w_2 and v_1 rounds to w_3, which is still a value between w_2 and v_1. Hence, double rounding with the rno mode produces the same value as if rounding $v_{\mathbb{R}}$ directly to $\mathbb{T}_n$. Moreover, by our definition of odd interval, any value v in the odd interval of the correctly rounded result of $v_{\mathbb{R}}$ in $\mathbb{T}_{n+2}$ with the rno mode will round to the same value in $\mathbb{T}_k$ using rm mode as if rounding $v_{\mathbb{R}}$ directly to $\mathbb{T}_k$ using rm mode. Figure 6.7 shows the odd intervals using the gray area. It can be seen that if we round any value within the gray area to $\mathbb{T}_n$ using the standard rounding modes, then the result will be the same as if rounding the real value in red star directly to $\mathbb{T}_n$.

6.3.3 Generating Polynomials for $\mathbb{T}_{n+2}$ with the rno Mode

Our strategy to create a polynomial approximation that produces correctly rounded results of $f(x)$ in $\mathbb{T}_k$ with any standard rounding modes is to generate a polynomial that produces the correctly rounded results for $\mathbb{T}_{n+2}$ with the rno mode. Algorithm 6.1 provides a high-level sketch of our approach to generate such a polynomial. Given an elementary function $f(x)$ and a list of inputs X in the $\mathbb{T}_n$ representation (*i.e.*, $X \subseteq \mathbb{T}_n$), our first step is to

Algorithm 6.1: A sketch of our approach to generate piecewise polynomial of degree d for the elementary function $f(x)$ in $\mathbb{T}_{n+2}$ using rno mode. The resulting polynomial when used with range reduction strategy ($RR_{\mathbb{H}}$ and $OC_{\mathbb{H}}$) produces the correctly rounded results for all inputs with all $\mathbb{T}_k$ and standard rounding modes. `CalcResultsInRNO` computes the rno result of $f(x)$ in $\mathbb{T}_{n+2}$ using oracle (shown in Algorithm 6.2). `CalcOddIntervals` computes the odd interval for each input x and the set S containing inputs with singleton odd interval. (shown in Algorithm 6.3). Using the list of inputs and odd intervals, we use the approach described in Chapter 5 to generate the piecewise polynomial (lines 5 - 9).

```
1  Function GenericPolynomial(f, T_{n+2}, H, X, d, RR_H, OC_H):
2      O ← CalcResultsInRNO(f, T_{n+2}, X)
3      (L, S) ← CalcOddIntervals(O, T_{n+2}, H)
4      if L = ∅ then return (false, ∅, DNE)
       // Generate piecewise polynomial using the approach from
          Chapter 5
5      L' ← CalcReducIntervals(L, H, RR_H, OC_H)
6      if L' = ∅ then return (false, DNE)
7      L* ← CombineReducIntervals(L')
8      if L* = ∅ then return (false, DNE)
9      Ψ ← GenPiecewise(L*, d)
10     return (true, S, Ψ)
```

compute the correctly rounded result of $f(x)$ in $\mathbb{T}_{n+2}$ with the rno mode to produce y_{rno} for each input $x \in X$ (line 2). Algorithm 6.2 shows our algorithm to compute y_{rno} for each input.

Next, we compute the odd interval of each result y_{rno} (line 3). Any value in the odd interval rounds to y_{rno}. The algorithm to compute the odd interval is shown in Algorithm 6.3. There can be inputs where the corresponding odd interval is a singleton. By the definition of the rno mode and the odd interval, this occurs when the real value of $f(x)$ is exactly representable in $\mathbb{T}_{n+2}$ and the bit-string of $f(x)$ in $\mathbb{T}_{n+2}$ is even. Generating polynomial approximations that produce the exact value of $f(x)$ is a difficult problem. Hence, we identify inputs with singleton odd intervals using mathematical properties of elementary functions and handle them as special cases (Section 6.3.7). Once we compute the odd intervals for each input, then we can use the approach described in Chapter 5 to generate piecewise polynomials with counterexample guided polynomial generation that produces the correctly rounded result of $f(x)$ when used with range reduction strategies (lines 5-9).

Algorithm 6.2: `CalcResultsInRNO` computes the correctly rounded result of $f(x)$ in $\mathbb{T}_{n+2}$ using the rno rounding mode for each input $x \in X$. `FGetRComp` returns the rounding components for rounding y to $\mathbb{T}_{n+2}$.

```
1  Function CalcResultsInRNO(f, T_{n+2}, X):
2      O ← ∅;
3      foreach x ∈ X do
4          y = f(x);
5          (s, v⁻, rb, sticky) ← RComp(y, T_{n+2});
6          if IsOdd(v⁻) ∨ (rb = 0 ∧ sticky = 0) then
7              y_rno ← s × v⁻
8          end
9          else
10             v⁺ ← GetSuccVal(v⁻, T_{n+2});
11             y_rno ← s × v⁺;
12         end
13         O ← O ∪ (x, y_rno);
14     end
15     return O
```

At the end of this process, we have two components to produce correctly rounded results for $f(x)$. First, our approach generates a set S containing the inputs x whose odd interval is a singleton. We implement a check to these inputs and return the pre-computed results for them. A naive approach is to store the list of these inputs x and the pre-computed result of $f(x)$ into a look-up table and implement the check using branch conditions or switch statements. To create a faster implementation, we use mathematical properties of the elementary functions to implement these checks (Section 6.3.7). Second, our approach produces piecewise polynomials that produce the correctly rounded results for all inputs when rounded to any $\mathbb{T}_k$ with all standard rounding modes. We now describe our approach in computing the rno result of $f(x)$ and their odd intervals in $\mathbb{T}_{n+2}$.

6.3.4 Computing the rno result of $f(x)$ in $\mathbb{T}_{n+2}$

The first step in our approach is to identify the correctly rounded result y_{rno} for each input $x \in X$. Algorithm 6.2 illustrates our steps to compute the rno result of $f(x)$ in $\mathbb{T}_{n+2}$. We compute the real value $y = f(x)$ for each input x using an oracle (line 4). Then, we obtain

Algorithm 6.3: `CalcOddIntervals` computes the odd intervals for each input x based on the correctly rounded result y_{rno} in $\mathbb{T}_{n+2}$. The list S is a set of inputs with singleton odd interval. The list L contains inputs and the corresponding odd intervals where the interval is not a singleton. `GetPrecVal`(a, $\mathbb{T}$) returns the value preceding a in the representation $\mathbb{T}$. `GetSuccVal`(a, $\mathbb{T}$) returns the value succeeding a in the representation $\mathbb{T}$.

```
1  Function CalcOddIntervals(O, T_{n+2}, H):
2      foreach (x, y_rno) ∈ O do
3          L ← ∅
4          S ← ∅
5          if IsEven(y_rno) then
6              S ← S ∪ (x, y_rno)
7          end
8          else
9              y⁻ ← GetPrecVal(y_rno, T_{n+2})
10             l ← GetSuccVal(y⁻, H)
11             y⁺ ← GetSuccVal(y_rno, T_{n+2})
12             h ← GetPrecVal(y⁺, H)
13             L ← L ∪ (x, [l, h])
14         end
15         return (L, S)
16     end
```

the rounding components $(s, v^-, rb, sticky)$ as described in Section 2.1.3 (line 5). If the real value y is exactly representable in $\mathbb{T}_{n+2}$ (*i.e.*, $rb = 0$ and $sticky = 0$) or the truncated value v^- is odd, then the rno result is $s \times v^-$ (line 7). Otherwise, the rno result is $s \times v^+$, where v^+ is the succeeding value of v^- in $\mathbb{T}_{n+2}$.

6.3.5 Computing the Odd Intervals

Once the correctly rounded result y_{rno} of $f(x)$ in $\mathbb{T}_{n+2}$ using the rno mode is computed for each x, the next step is to identify a range of values such that producing any value in the interval rounds to y_{rno}. We call this interval the odd interval. Because the range reduction, polynomial evaluation, and output compensation are evaluated in a higher precision representation $\mathbb{H}$, we compute the odd interval in $\mathbb{H}$. Algorithm 6.3 illustrates our steps to compute the odd interval.

If the bit-string of y_{rno} in $\mathbb{T}_{n+2}$ is even when interpreted as an unsigned integer, then the odd interval is a singleton. The only value that rounds to y_{rno} with the rno mode is y_{rno}

itself. Generating a polynomial approximation that produces the exact singleton value y_{rno} is a challenging problem. These singletons restrict the amount of freedom available for polynomial generation approaches described in previous chapters. Hence, we filter these inputs and store the input x and the exact result y_{rno} separately in the set S (line 7). In Section 6.3.7, we describe our approach to identify the inputs with singleton odd intervals and efficiently compute the result using mathematical properties of the elementary function.

If the bit-string of y_{rno} is odd when interpreted as an unsigned integer, then all values in $\mathbb{H}$ that are strictly greater than the preceding value of y_{rno} in $\mathbb{T}_{n+2}$ (lines 9-10) and strictly less than the succeeding value of y_{rno} in $\mathbb{T}_{n+2}$ (lines 11-12) forms the odd interval. Any value in the odd interval rounds to y_{rno} when rounded to $\mathbb{T}_{n+2}$ using the rno mode. The odd interval in this case is not a single interval. We compute the odd intervals for each input and store them in the list L (line 14).

Generating piecewise polynomial using the odd interval. The final step is to generate piecewise polynomials that produce a value in the odd interval for each input. Each input and its odd interval, $(x, [l, h]) \in L$ specifies the constraint on the polynomial that we want to generate. We can use the approach described in Chapter 5 to generate the piecewise polynomial that satisfies the constraints in L. Our approach also supports generating polynomials with range reduction strategies described in Chapter 4 as well.

6.3.6 Implementing the Elementary Function

At the end of this process, our approach produces two components: (1) a set S containing inputs whose odd interval is a singleton and (2) a polynomial approximation of $f(x)$ that produces the correctly rounded result of $f(x)$ in $\mathbb{T}_{n+2}$ with the rno mode for all inputs (excluding the inputs in S). We implement the elementary function $f(x)$ as follows. Given an input x, we first check whether x is an input with a singleton odd interval. If so, we can use the set S to obtain the result. Section 6.3.7 describes our approach to perform

this check and compute the result efficiently. Otherwise, we use the generated polynomial approximation to produce the result. We use range reduction $RR_{\mathbb{H}}$ to compute the reduced input, use the reduced input the evaluate the polynomial, and use output compensation $OC_{\mathbb{H}}$ to obtain the result. We round the result of output compensation to $\mathbb{T}_k$ using a user specified rounding mode to produce the final result. By Theorem 6.1, we guarantee that our implementation produces the correctly rounded result of $f(x)$ for any $\mathbb{T}_k$ with any standard rounding modes for all inputs $x \in \mathbb{T}_k$.

6.3.7 Efficiently Identifying Inputs With Singleton Generic Interval

One of the challenges in generating polynomials that produce a value in the odd interval is that there are inputs with singleton odd intervals containing only y_{rno}. To remedy this problem, our approach treats all inputs x where the odd interval is a singleton as special case inputs. Our approach identifies these inputs and the exact result of $f(x)$ in a special case set S. If the number of special case inputs in S is small, then we can check whether an input is a special case using a series of branch conditions or switch statements with minimal performance overhead. However, if there are hundreds of special case inputs, then this strategy will incur overhead, significantly slowing down the performance of the approximation function. Thus, our goal is to efficiently identify these inputs.

Mathematically identifying inputs with singleton odd interval. For many elementary functions, it is possible to mathematically identify whether an input may result in a singleton odd interval and identify the exact result of $f(x)$. Observe that both $\mathbb{T}_n$ and $\mathbb{T}_{n+2}$ are finite precision representations. Hence, all values in $\mathbb{T}_n$ and $\mathbb{T}_{n+2}$ are rational values. The only case where the real value of an elementary function $f(x)$ is exactly representable in $\mathbb{T}_{n+2}$ for an input in $\mathbb{T}_n$ is if the input is a rational value and the result of $f(x)$ is a rational value. For many elementary functions, there are only a few rational inputs that produce rational results. For example, e^x is a transcendental function where the only rational input

that produces a rational result is the input $x = 0$ resulting in $e^0 = 1.0$. The problem of identifying rational inputs that produce rational outputs for various elementary functions is well studied [1, 5, 26, 106].

To identify inputs in $\mathbb{T}_n$ with a single odd interval, we first use mathematical properties of elementary functions $f(x)$ to identify all rational inputs such that $f(x)$ is a rational value. Then, we check if these rational inputs and the corresponding rational results are exactly representable in $\mathbb{T}_n$ and $\mathbb{T}_{n+2}$, respectively. If it is, the odd interval of y_{rno} corresponding to the rational input may be a singleton interval. We classify these rational inputs as special cases. Finally, we develop a programmatic approach to efficiently identify the inputs without using a series of branch statements and compute the exact result of $f(x)$. We now describe the mathematical properties that we use to identify inputs with singleton odd intervals and programmatic approaches to efficiently identify these inputs and the corresponding results.

Exponential function e^x**.** From Lindemann-Weierstrass theorem [5], if the input x is rational and non-zero, then e^x cannot be a rational value. Thus, $x = 0$ is the only input where e^x is exactly representable in $\mathbb{T}_{n+2}$ with $e^0 = 1$. We use a single branch statement to check whether an input $x = 0$ and return the value $e^x = 1$.

Exponential function 2^x**.** To determine inputs $x \in \mathbb{T}_n$ that produces rational result of 2^x, we first break the input x down into $x = f + i$ where $f \in [0, 1.0)$ represents the fractional part of x and the integer i represents the integral part of x. Then, the function 2^x can be broken down into the following formula,

$$2^x = 2^{f+i} = 2^f 2^i$$

Our first goal is to find the rational inputs x where $2^x = 2^f 2^i$ is a rational value. The value 2^i is guaranteed to be a non-zero rational value since i is an integer. For the product $2^f 2^i$ to be a rational value, the value 2^f must be a rational value. Since f is also a rational value,

f can be broken down into $f = \frac{p}{q}$ where p and q are both positive integers. Additionally, because $f \in [0, 1)$, it must be the case that $p < q$. Then,

$$2^f = 2^{\frac{p}{q}} = \sqrt[q]{2^p}$$

The value 2^p is an integer. The q^{th} root of an integer is either an integer or an irrational value [26]. For the value $\sqrt[q]{2^p}$ to be an integer, it must be the case that 2^p is an integer and p is a multiple of q. Since $p < q$, it must be the case that $p = 0$ and $f = \frac{0}{q} = 0$. Thus, the only rational inputs where 2^x is a rational value is when x is an integer i and the fractional value $f = 0$.

Next, to identify all inputs $x \in \mathbb{T}_n$ where 2^x is exactly representable in $\mathbb{T}_{n+2}$, we identify all integer inputs $x \in \mathbb{T}_n$ and check whether 2^x is exactly representable in $\mathbb{T}_{n+2}$. If 2^x is exactly representable in $\mathbb{T}_{n+2}$, then x is an input that can have a singleton generic interval and x is classified as a special case input. In the case when $\mathbb{T}_n$ is a 32-bit float, $\mathbb{T}_{n+2}$ can represent all values of 2^x for x between $-151 \leq x \leq 127$. Thus, the special case inputs $x \in \mathbb{T}_n$ are all integers between -151 and 127 (279 inputs in total),

$$x \in \mathbb{Z} \wedge -151 \leq x \leq 127$$

Checking whether an input $x \in \mathbb{T}_n$ is an integer can be performed efficiently using bit-wise operations. Similarly, computing the final result 2^x can be performed with bit-wise operations.

Exponential function 10^x**.** Similar to the 2^x function, the only rational inputs x that result in a rational result of 10^x is when x is an integer. The reasoning can be followed similarly to the case for 2^x. Additionally, the result of 10^x for negative inputs x (*i.e.*, $10^{-1} = 0.1$) cannot be exactly represented in $\mathbb{T}_{n+2}$. Thus, the only inputs $x \in \mathbb{T}_n$ where 10^x can be exactly represented in $\mathbb{T}_{n+2}$ are positive integers. In contrast to 2^x, 10^x grows much faster and there are only a few inputs in $\mathbb{T}_n$ for which 10^x is exactly representable in $\mathbb{T}_{n+2}$. When

$\mathbb{T}_n$ is a 32-bit float, there are exactly 12 integer inputs ranging from 0 to 11 where 10^x is exactly representable in $\mathbb{T}_{n+2}$. Although there is no efficient methodology in computing 10^x even for integer inputs, we can use a switch statement to check whether an input is one of the twelve special case inputs and return the correct result.

Natural log function $ln(x)$**.** The result of the function $ln(x)$ is always irrational if an input x is rational and not equal to zero. This property can be proven using the Lindemann-Weierstrass theorem [5]. Let us suppose that we have a rational value $x \neq 1$ where $y = ln(x) \neq 0$ is also a rational value. Since y is a non-zero rational value, e^y must be an irrational value from Lindemann-Weierstrass theorem [5]. However,

$$e^y = e^{ln(x)} = x$$

This creates a contraction because we assumed that x is a rational value. Hence, it must be the case that y is an irrational value.

Thus, the only special case input $x \in \mathbb{T}_n$ is the input $x = 1$ where $ln(x)$ is exactly representable in $\mathbb{T}_{n+2}$ with $ln(1) = 0$. We use a single branch statement to check whether an input $x = 1$ and returning the value $ln(1) = 0$.

Log base two function $log_2(x)$**.** To determine rational inputs x that produces rational result of $log_2(x)$, let us suppose that $log_2(x) = \frac{p}{q}$ for positive integers p and q. Then, it follows that

$$x = 2^{\frac{p}{q}} = \sqrt[q]{2^p}$$

The value 2^p is an integer. The q^{th} root of an integer is either an integer or an irrational value [26]. For x to be rational, x has to be an integer. For x to be an integer, 2^p has to be a perfect q^{th} power of x. Hence, x must be an integer of the form 2^i where i is an integer. We use bitwise operations to check if the input x is a power of two, *i.e.*, 2^i. If it is, we return the value i, which can be efficiently computed using bitwise operations.

Log base 10 function $log_{10}(x)$**.** Similar to $log_2(x)$, $log_{10}(x)$ produces a rational result when x is a power of 10 (*i.e.*, $x = 10^i$ where i is an integer). Additionally, $\mathbb{T}_n$ cannot exactly represent negative power of 10 (*i.e.*, $10^{-1} = 0.1$). While there is no efficient way of identifying whether an arbitrary value x is a power of 10, there are a limited number of inputs in $\mathbb{T}_n$ that are power of 10. In 32-bit float (*i.e.*, $\mathbb{F}_{32,8}$), there are only 11 such values ranging from 10^0 to 10^{10}. For each inputs 10^i where $0 \leq i \leq 10$, the result $log_{10}(10^i) = i$ is exactly representable in $\mathbb{T}_{n+2}$ (*i.e.*, $\mathbb{F}_{34,8}$). We use a look-up table to store the inputs and its corresponding result in $\mathbb{T}_{n+2}$ and implement the check in a switch statement.

The hyperbolic sine function $sinh(x)$**.** As an extension of the Lindemann-Weierstrass theorem, if the input x is rational and non-zero, then $y = sinh(x)$ cannot be a rational value [106]. The only input $x \in \mathbb{T}_n$ where $sinh(x)$ is exactly representable in $\mathbb{T}_{n+2}$ is the input $x = 0$ where $sinh(0) = 0$. We use a single branch statement to check whether an input $x = 0$ and returning the value $sinh(0) = 0$.

The hyperbolic cosine function $cosh(x)$**.** Similar to the hyperbolic sine function, if the input x is rational and non-zero, then $y = cosh(x)$ cannot be a rational value. The only input $x \in \mathbb{T}_n$ where $cosh(x)$ is exactly representable in $\mathbb{T}_{n+2}$ is the input $x = 0$ with $cosh(0) = 1$. We use a single branch statement to check whether an input $x = 0$ and returning the value $cosh(0) = 1$.

Trigonometric sinpi function $sinpi(x) = sin(\pi x)$**.** The function $sinpi(x)$ is equal to $sin(\pi x)$. By Niven's theorem [106], the only rational values of x between $0 \leq x \leq \frac{1}{2}$ where $sinpi(x)$ is also rational number are one of three values:

$$sinpi(x) = \begin{cases} 0 & \text{if } x = 0 \\ \frac{1}{2} & \text{if } x = \frac{1}{6} \\ 1 & \text{if } x = \frac{1}{2} \end{cases}$$

Among these inputs $\frac{1}{6}$ is not exactly represented in $\mathbb{T}_n$. When we extend the domain of x to the set of all inputs, there are only three categories of inputs in $x \in \mathbb{T}_n$ where the result of $sinpi(x)$ is representable in $\mathbb{T}_{n+2}$:

$$sinpi(x) = \begin{cases} 0 & \text{if } x \text{ is an integer} \\ 1 & \text{if } x \equiv \frac{1}{2} \mod 2.0 \\ -1 & \text{if } x \equiv \frac{3}{2} \mod 2.0 \end{cases}$$

We implement the modulo operation efficiently using integer operations. Consider the case when $\mathbb{T}_n$ is a 32-bit float. Without the loss of generality, let us only consider positive inputs in $\mathbb{T}_n$. First, because float has 23 precision bits, when x is broken down into the binary scientific notation $x = m \times 2^e$, m can have up to 23 fraction bits. Thus, all inputs $x \in \mathbb{T}_n$ greater than or equal to 2^{23} are guaranteed to be integer values and the result of $sinpi(x)$ for these values are always 0. Next, if $x < 2^{23}$, then we compute $2x$ in float and round the result to a 32-bit integer to obtain the value t. In essence, this operation truncates the value $2x$ to the integral part of $2x$ and removes the fractional part. Because $2x < 2^{24}$, the rounded result is exactly representable in 32-bit integer. If t and $2x$ are identical, then x is guaranteed to be either an integer or a multiple of 0.5, indicating that x is a special case input. We compute the result of $sinpi(x)$ based on t:

$$sinpi(x) = \begin{cases} 0 & \text{if } t \equiv 0 \mod 2 \\ 1 & \text{if } t \equiv 1 \mod 4 \\ -1 & \text{if } t \equiv 3 \mod 4 \end{cases}$$

Trigonometric cospi function $cospi(x) = cos(\pi x)$**.** Similar to the sinpi function, the only input x in $\mathbb{T}_n$ where $cospi(x) = cos(\pi x)$ is also rational can be classified as one of the three

categories:

$$cospi(x) = \begin{cases} 1 & \text{if } x \text{ is an even integer} \\ -1 & \text{if } x \text{ is an odd integer} \\ 0 & \text{if } fraction(x) = 0.5 \end{cases}$$

These checks can be performed efficiently using a similar strategy illustrated for the $sinpi(x)$ function.

6.4 Proof of $\mathbb{T}_{n+2}$ Result with rno Producing Correctly Rounded Result for $\mathbb{T}_k$

We now provide a proof of Theorem 6.1. Through this proof, we show that the result produced by our polynomial approximation, which approximates the correctly rounded result of $f(x)$ in $\mathbb{T}_{n+2}$ with rno rounding mode, will round to the correctly rounded result for $\mathbb{T}_k$ using any standard rounding modes. Our proof for Theorem 6.1 applies to both FP and posit representations. We first prove properties of the rno rounding mode and the rounding components we use to round real values to $\mathbb{T}_k$. Then, we use these properties to prove Theorem 6.1. When we refer to the bit-string of $v_{\mathbb{R}}$, we refer to it in the infinite extended precision representation, $\mathbb{T}_\infty$. When we refer to the bit-string of v_{rno}, the rno result of $v_{\mathbb{R}}$ in $\mathbb{T}_{n+2}$, we refer to it in the $\mathbb{T}_{n+2}$ representation.

Lemma 1. *Let $v_{rno} = RN_{\mathbb{T},rno}(v_{\mathbb{R}})$. Then, v_{rno} preserves the sign of $v_{\mathbb{R}}$.*

The value zero is exactly representable in $\mathbb{T}_{n+2}$. In both FP and posit representations, the bit-string of zero is even when interpreted as an unsigned integer. Thus, the only real value that rounds to zero when using rno is zero itself. All positive real values round to a positive value in $\mathbb{T}_{n+2}$ and all negative real values round to a positive value in $\mathbb{T}_{n+2}$ when using rno. Hence, v_{rno} preserves the sign of $v_{\mathbb{R}}$. □

Lemma 2. *Let $v_{rno} = RN_{\mathbb{T},rno}(v_{\mathbb{R}})$. Then, the first $n+1$ bits in $|v_{rno}|$ and $|v_{\mathbb{R}}|$ are identical.*

We prove the lemma with the rounding components (s, v^-, rb, $sticky$) used for rounding $v_{\mathbb{R}}$ to v_{rno} in $\mathbb{T}_{n+2}$ using rno. The value v^- is the truncated value of $|v_{\mathbb{R}}|$. Hence, all

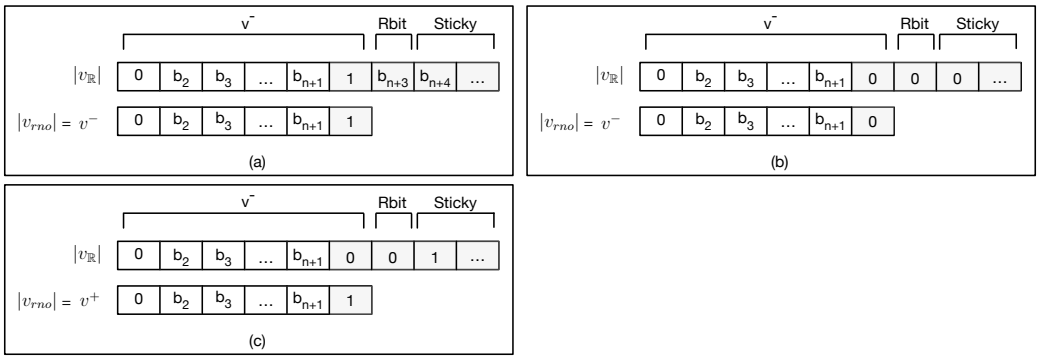


Figure 6.8: Illustration of Lemma 2 and Lemma 3 for different cases of $|v_{\mathbb{R}}|$ and $|v_{rno}|$ where $|v_{rno}|$ is the rno result of $|v_{\mathbb{R}}|$ in $\mathbb{T}_{n+2}$. We show the bit-string of $|v_{\mathbb{R}}|$ and $|v_{rno}|$ for three cases. (a) v^- is odd. (b) v^- is even, $rb = 0$, and $sticky = 0$. (c) v^- is even and either $rb = 1$ or $sticky = 1$.

$n + 2$ bits in v^- and the first $n + 2$ bits in $|v_{\mathbb{R}}|$ are identical. The value $|v_{\mathbb{R}}|$ rounds to either v^- or the succeeding value v^+ in $\mathbb{T}_{n+2}$. We split our proof into two cases: When the bit-string of v^- is odd and when it is even. Figure 6.8 pictorially illustrates our proof.

If the bit-string of v^- is odd, then $|v_{rno}| = v^-$ by the definition of rno (Figure 6.8(a)). Hence, all $n + 2$ bits of $|v_{rno}|$ and $|v_{\mathbb{R}}|$ are identical. If the bit-string of v^- is even, then the last bit of v^- is even. Depending on the value of rb and $sticky$, either $|v_{rno}| = v^-$ or $|v_{rno}| = v^+$. If $rb = 0$ and $sticky = 0$, then $|v_{rno}| = v^-$ (Figure 6.8(b)). Hence, all $n + 2$ bits of $|v_{rno}|$ and $|v_{\mathbb{R}}|$ are identical. If either $rb = 1$ and $sticky = 1$, then $|v_{rno}| = v^+$ (Figure 6.8(c)). Because the last bit of v^- is a 0, the only bit that is different between v^- and v^+ is the last bit, where the last bit of v^+ is a 1. Hence, the first $n + 1$ bits of $|v_{rno}|$ and $|v_{\mathbb{R}}|$ are identical. □

Lemma 3. *Let $v_{rno} = RN_{\mathbb{T},rno}(v_{\mathbb{R}})$. Then, the last bit in $|v_{rno}|$ is equal to the bitwise* `OR` *of all the bits in $|v_{\mathbb{R}}|$ starting from the $(n + 2)^{th}$ bit.*

Intuitively, Lemma 3 states that the last bit of $|v_{rno}|$ is zero if and only if all bits starting from the $(n + 2)^{nd}$ bit in $|v_{\mathbb{R}}|$ are zeros. We prove this lemma using a similar strategy as Lemma 2. Let (s, v^-, rb, and $sticky$) be the rounding components for rounding $v_{\mathbb{R}}$ to $\mathbb{T}_{n+2}$ using rno. The value $|v_{\mathbb{R}}|$ rounds to either v^- or the succeeding value v^+ in $\mathbb{T}_{n+2}$.

If the bit-string of v^- is odd, then $|v_{rno}| = v^-$. The $(n + 2)^{nd}$ bit of v^- and $|v_{rno}|$ in

$\mathbb{T}_{n+2}$ is 1. Since v^- is a truncated value of $|v_\mathbb{R}|$, the $(n+2)^{nd}$ bit of $|v_\mathbb{R}|$ is 1. Hence, the bitwise OR of all bits of $|Real|$ starting from the $(n+2)^{nd}$ bit is 1.

If the bit-string of v^- is even, then our proof is subdivided into two additional cases depending on the values of rb and $sticky$. If $rb = 0$ and $sticky = 0$, then $|v_{rno}| = v^-$. The $(n+2)^{nd}$ bit of $|v_{rno}|$ is 0. Similarly, the $(n+2)^{nd}$ bit of $|v_\mathbb{R}|$ is even. Additionally, from the definition of rounding components, rb represents the the $(n+3)^{rd}$ bit in $|v_\mathbb{R}|$ and $sticky$ represents the bitwise OR operation of all bits starting from $(n+4)^{th}$ bit in $|v_\mathbb{R}|$. Since both rb and $sticky$ are 0, all bits in $|v_\mathbb{R}|$ starting from the $(n+2)^{nd}$ bit are 0.

If either $rb = 1$ or $sticky = 1$, then $|v_{rno}|$ is equal to v^+, where the bit-string of v^+ is odd. Hence, the $(n+2)^{nd}$ bit of $|v_{rno}|$ is 1. Because either rb or $sticky$ is not zero, at least one bit starting from $(n+2)^{nd}$ bit in $|v_\mathbb{R}|$ is 1. The bitwise OR operation of all bits starting from $(n+2)^{nd}$ bit in $|v_\mathbb{R}|$ is 1, which matches the $(n+2)^{nd}$ bit in $|v_{rno}|$. □

Lemma 4. *Define a representation $\mathbb{T}_k$, a rounding mode rm, and two real values v_1 and v_2. Let (s_1, v_1^-, rb_1, $sticky_1$) be the round components for rounding v_1 to $\mathbb{T}_k$ and (s_2, v_2^-, rb_2, $sticky_2$) be the rounding components for rounding v_2 to $\mathbb{T}_k$. If $s_1 = s_2$, $v_1^- = v_2^-$, $rb_1 = rb_2$, and $sticky_1 = sticky_2$, then $RN_{\mathbb{T}_k,rm}(v_1) = RN_{\mathbb{T}_k,rm}(v_2)$.*

Lemma 4 follows directly from the definition of rounding components we described in Section 2.1.3. Identifying the correctly rounded result of the real value in $\mathbb{T}_k$ for any rounding mode only depends on the rounding components. Chapter 2 describes our deterministic approach to round $v_\mathbb{R}$ to FP and posit representation, respectively, using the rounding components. □

Proof Of Double Rounding With rno Producing Correctly Rounded Result We now present the proof of Theorem 6.1, which we reiterate for clarity:

Theorem 6.1. *Let $f(x)$ be the real value of an elementary function given an input x and $v_{rno} = RN_{\mathbb{T}_{n+2},rno}(f(x))$. Let v be a value in the odd interval of v_{rno} such that for all v,*

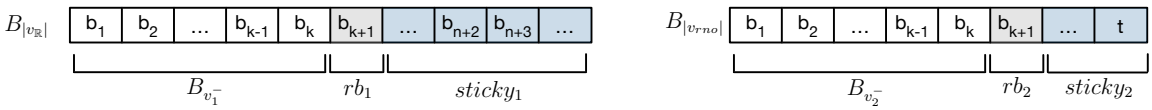


Figure 6.9: Rounding components when rounding $v_\mathbb{R}$ and v_{rno} to $\mathbb{T}_k$, where $1 + |E| < k < n$. We show the bit-string of $v_\mathbb{R}$ in the infinite extended precision and the bit-string of v_{rno} in $\mathbb{T}_{n+2}$ representation.

$RN_{\mathbb{T}_{n+2},rno}(v) = v_{rno}$. *Choose a rounding mode* $rm \in \{rne, rna, rnz, rnp, rnn\}$. *Then,*

$$RN_{\mathbb{T}_k,rm}(v) = RN_{\mathbb{T}_k,rm}(f(x)))$$

Note that both $f(x)$ and v are real values that round to v_{rno} when rounded to $\mathbb{T}_{n+2}$ with the rno mode. Hence, we show that rounding any real value $v_\mathbb{R}$ to $\mathbb{T}_{n+2}$ using rno to produce the value v_{rno} and then subsequently rounding v_{rno} to $\mathbb{T}_k$ using any standard rounding mode rm produces the same value as directly rounding $v_\mathbb{R}$ to $\mathbb{T}_k$ using the same rounding mode. More formally, we prove that,

$$RN_{\mathbb{T}_k,rm}(RN_{\mathbb{T}_{n+2},rno}(v_\mathbb{R})) = RN_{\mathbb{T}_k,rm}(v_\mathbb{R})$$

Since both $f(x)$ and v round to v_{rno} when rounded to $\mathbb{T}_{n+2}$ with the rno, we have the relationship

$$RN_{\mathbb{T}_k,rm}(RN_{\mathbb{T}_{n+2},rno}(v)) = RN_{\mathbb{T}_k,rm}(v_\mathbb{R}) = RN_{\mathbb{T}_k,rm}(RN_{\mathbb{T}_{n+2},rno}(f(x)))$$

thus proving Theorem 6.1. Our proof holds for FP representations when $\mathbb{T}_{n+2} = \mathbb{F}_{n+2,|E|}$ and $\mathbb{T}_k = \mathbb{F}_{k,|E|}$ as long as $1 + |E| < k \leq n$. For posit representations, our proof holds when $\mathbb{T}_{n+2} = \mathbb{P}_{n+2,es}$ and $\mathbb{T}_k = \mathbb{P}_{k,es}$ as long as $2 \leq k \leq n$.

In high-level, our strategy is to prove that the rounding components for rounding $v_\mathbb{R}$ to $\mathbb{T}_k$ are the same as the rounding components for rounding v_{rno} to $\mathbb{T}_k$. By Lemma 4, the equivalence of rounding components proves that $RN_{\mathbb{T}_k,rm}(v_\mathbb{R}) = RN_{\mathbb{T}_k,rm}(v_{rno})$ for all rounding modes rm. We now show the rounding components for rounding $v_\mathbb{R}$ and v_{rno} to $\mathbb{T}_k$.

We first identify the rounding components (s_1, v_1^-, rb_1, $sticky_1$) for rounding $v_\mathbb{R}$ to $\mathbb{T}_k$. We decompose $v_\mathbb{R}$ into $v_\mathbb{R} = s_1 \times |v_\mathbb{R}|$. The value s_1 is the first rounding compo-

nent representing the sign of $v_{\mathbb{R}}$. The bit-string of $|v_{\mathbb{R}}|$ in the infinite extended precision representation, $B_{|v_{\mathbb{R}}|}$, is pictorially shown in Figure 6.9.

$$B_{|v_{\mathbb{R}}|} = b_1 b_2 \ldots b_{k-1} b_k b_{k+1} \ldots b_{n+2} b_{n+3} \ldots$$

We identify the remaining three rounding components from $B_{|v_{\mathbb{R}}|}$. The truncated value v_1^- in $\mathbb{T}_k$ and its bit-string $B_{v_1^-}$ is identified by truncating the first k bits in $B_{|v_{\mathbb{R}}|}$. The rounding bit rb_1 is the $(k+1)^{st}$ bit in $B_{|v_{\mathbb{R}}|}$ and the sticky bit $sticky_1$ is the bit-wise OR of all bits starting from the $(k+2)^{nd}$ bit in $B_{|v_{\mathbb{R}}|}$:

$$B_{v_1^-} = b_1 b_2 b_3 \ldots b_{k-1} b_k, \qquad rb_1 = b_{k+1}, \qquad sticky_1 = b_{k+2} \mid b_{k+3} \mid \ldots$$

Figure 6.9 pictorially shows the rounding components $B_{v_1^-}$, rb_1, and $sticky_1$.

Next, we identify the rounding components (s_2, v_2^-, rb_2, $sticky_2$) for rounding v_{rno} to $\mathbb{T}_k$. The sign of v_{rno} is the first component s_2. The bit-string of $|v_{rno}|$ in $\mathbb{T}_{n+2}$ is pictorially shown in Figure 6.9 with $B_{|v_{rno}|}$.

Note that from Lemma 2, the first $n+1$ bits of $|v_{rno}|$ and $|v_{\mathbb{R}}|$ are identical. Additionally from Lemma 3, the $(n+2)^{nd}$ bit (the last bit) of $|v_{rno}|$ is equal to the bitwise OR operation of all bits in $|v_{\mathbb{R}}|$ starting from the $(n+2)^{nd}$ bit. Hence,

$$B_{|v_{rno}|} = b_1 b_2 b_3 \ldots b_{k-1} b_k \ldots b_n b_{n+1} t \qquad t = b_{n+2} \mid b_{n+3} \mid \ldots$$

Because $k \leq n$, there is at least 1 bit (*i.e.*, b_{n+1}) between b_k and t, where t is the $(n+2)^{nd}$ bit in $|v_{rno}|$.

We identify the remaining three rounding components for v_{rno} from $B_{|v_{rno}|}$. The bit-string of v_2^-, rb_2, and $sticky_2$ are as follows:

$$b_{v_2^-} = b_1 b_2 b_3 \ldots b_{k-1} b_k \qquad rb_2 = b_{k+1} \qquad sticky_2 = b_{k+2} \mid \ldots \mid b_{n+1} \mid t$$

Figure 6.9 also pictorially shows $B_{v_2^-}$, rb_2, and $sticky_2$.

Now, let us compare the rounding components for rounding $v_{\mathbb{R}}$ and v_{rno} to $\mathbb{T}_k$. The sign, s_1 and s_2, are identical because rno rounding mode preserves the sign of $v_{\mathbb{R}}$. The

truncated values, v_1^- and v_2^-, are equal to each other because the bit-strings are the same. The rounding bits, rb_1 and rb_2, are both equal to b_{k+1}, which is guaranteed to be a bit in the first $n+1$ bits in both $|v_{\mathbb{R}}|$ and $|v_{rno}|$ since $k \leq n$. Finally, the sticky bit $sticky_2$ is equal to,

$$sticky_2 = b_{k+2} \mid \ldots \mid b_{n+1} \mid t = b_{k+2} \mid \ldots \mid b_{n+1} \mid b_{n+2} \mid b_{n+3} \mid \cdots = sticky_1$$

Hence, all rounding components for rounding $v_{\mathbb{R}}$ and v_{rno} to $\mathbb{T}_k$ are identical. From Lemma 4, it follows that $RN_{\mathbb{T}_k,rm}(RN_{\mathbb{T}_{n+2},rno}(v_{\mathbb{R}})) = RN_{\mathbb{T}_k,rm}(v_{\mathbb{R}})$. $\square$

6.5 Odd Intervals for Extremal Values in Posit Representations

As stated multiple times throughout the chapter, our approach and Theorem 6.1 applies directly to posit representations and posit rounding mode. Polynomial approximations that produce a value in the odd interval of the rno result in $\mathbb{T}_{n+2}$ representation (*i.e.*, $(n+2)$-bit posit) will also produce the correctly rounded result for $\mathbb{T}_k$ (a k-bit posit) using the posit rounding mode, as long as $\mathbb{T}_{n+2}$ and $\mathbb{T}_k$ has the same number of es bits and $2 \leq k \leq n$.

However, when the real value of $f(x)$ for a given input x is outside the dynamic range of the posit representation $\mathbb{T}_n$, the corresponding odd interval may be unnecessarily restrictive in the context of producing the correctly rounded result of $f(x)$ in $\mathbb{T}_k$. This is due to the special behavior of posit rounding for extremal values. Without the loss of generality, let us assume that a real value $v_{\mathbb{R}} > 0$. The posit rounding rule states that when $v_{\mathbb{R}}$ is larger than the largest representable posit value $maxval$ in a posit representation, then $v_{\mathbb{R}}$ rounds to $maxval$. Similarly, when a value $v_{\mathbb{R}}$ is smaller than the smallest representable posit value $minval$, then $v_{\mathbb{R}}$ rounds to $minval$. The odd intervals do not take these special rounding rules into account. This results in odd intervals that are far smaller than necessary. As the size of the odd interval defines the amount of freedom we have to generate polynomials that produce the correctly rounded results, we need to generate the largest possible interval that still produces correct results for $\mathbb{T}_k$.

Figure 6.10 illustrates the difference between posit rounding and rno rounding for extremal values. Suppose that our goal is to generate an approximation function of a function

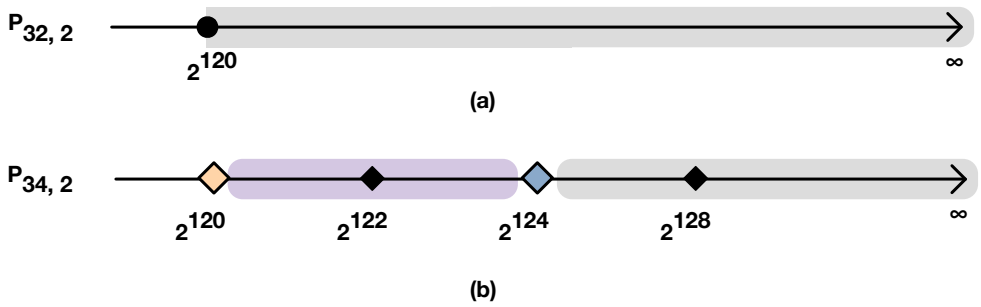


Figure 6.10: (a) The posit rounding mode specifies that all real values $v_{\mathbb{R}}$ larger than the largest representable value (*i.e.*, 2^{120} for $\mathbb{P}_{32,2}$) rounds to the largest representable value. (b) There are three values representable in $\mathbb{P}_{34,2}$ larger than 2^{120}. Separating odd intervals for each of these three values are unnecessary for the posit rounding mode, since all values in the odd intervals of 2^{120}, 2^{122}, 2^{124}, and 2^{128} always rounds to 2^{120} when rounded to $\mathbb{P}_{32,2}$.

$f(x)$ that produces the correctly rounded results for the posit representation $\mathbb{T}_n = \mathbb{P}_{32,2}$ and all smaller representations $\mathbb{T}_k = \mathbb{P}_{k,2}$ where $2 \leq k \leq 32$. The largest positive value that $\mathbb{P}_{32,2}$ can represent is $maxval_n = 2^{120}$ (the circle in Figure 6.10(a)). Based on the posit rounding rule, all values $v_{\mathbb{R}}$ larger than 2^{120} rounds to 2^{120} when rounded to $\mathbb{P}_{32,2}$. Similarly, any $v_{\mathbb{R}}$ larger than 2^{120} rounds to the corresponding maximum positive representable value ($maxval_k$) when rounded to any $\mathbb{P}_{k,2}$. Hence, if the real value of $f(x)$ given an x is larger than $maxval_n$, it is sufficient for the polynomial approximation to produce any value larger than $maxval_n$ (*i.e.*, $[maxval_n, \infty)$) for the input x. All values in this interval will round to $maxval_k$, the correctly rounded result in $\mathbb{T}_k$.

In contrast, the odd interval corresponding to the rno result of $f(x)$ is far smaller than $[maxval_n, \infty)$ when the real value of $f(x) \geq maxval_n$. The $\mathbb{P}_{34,2}$ representation can represent three values larger than 2^{120}: 2^{122}, 2^{124}, and 2^{128} (rhombuses in Figure 6.10(a). The odd interval of 2^{120}, 2^{122}, 2^{124}, and 2^{128} are shown with different colors. Depending on the rno result of $f(x)$, the odd interval can be as small as a singleton value (*e.g.*, $[2^{120}, 2^{120}]$ if rno result is 2^{120})! Yet all values in all four odd intervals round to the same value when rounded to any $\mathbb{T}_k$ representation using the posit rounding mode. There is no need to reason about the odd intervals of 2^{120}, 2^{122}, 2^{124}, and 2^{128} separately. In fact, it is better to constrain the polynomial approximation to produce a value in $[2^{120}, \infty)$ to provide the

maximum amount of freedom to generate the polynomial.

Hence, if the real value of $f(x)$ is larger than $maxpos_n$ for a given input x, we set the odd interval of the input x to be $[maxpos_n, \infty)$. Similarly, if the real value of $f(x)$ is smaller than the minimum positive representable value in $\mathbb{T}_n$ ($minpos_n$) but larger than 0, we set the odd interval to be $(0, minpos_n]$. The odd interval for the inputs, where the real value of $f(x) < 0$ is outside of the dynamic range of $\mathbb{T}_n$, is computed similarly.

6.6 Summary

This chapter presents a novel approach to generate polynomial approximations that produce correct results for multiple representations $\mathbb{T}_k$ and standard rounding modes where the total number of bits is bounded by $k \leq n$. Our key insight is to create polynomial approximations that produce the correctly rounded result of $\mathbb{T}_{n+2}$ using the rno mode. We identify a range of values that round to the correctly rounded result of $\mathbb{T}_{n+2}$ with the rno mode, which we call the odd interval. We formally showed that polynomials that produce a value in the odd interval are guaranteed to produce correctly rounded results for all representations $\mathbb{T}_k$ with standard rounding modes. Although our proof only shows for $\mathbb{T}_k$ with the same number of exponent bits as $\mathbb{T}_{n+2}$, we believe that our approach can produce correctly rounded results for any $\mathbb{T}_k$ as long as all values in $\mathbb{T}_k$ are representable in $\mathbb{T}_n$ representation. We empirically support our claim in Section 7.3. We handle singleton odd intervals by mathematically reasoning when elementary functions $f(x)$ will produce exact rational values.

Using this approach, we generated RLIBM-ALL, a generic math library that produces the correctly rounded results for multiple FP and posit representations with all standard rounding modes. Our generic math library is the first math library that produces the correctly rounded results for bfloat16, TensorFloat32, and float at the same time. This allows developers to design and use new configurations of FP or posit representations and still be able to accurately approximate elementary functions with the new representations.

CHAPTER 7
EXPERIMENTAL EVALUATION

We present the RLIBM prototype, a correctly rounded elementary function generator. Using RLIBM, we created several correctly rounded elementary functions for various representations and rounding modes. We evaluate our elementary functions on their ability to produce correctly rounded results for all inputs in their target representations and the performance. In this chapter, we provide details on the experimental methodology, the experimental setup, and the result of our experimental evaluation.

7.1 Experimental Methodology And Setup

The RLIBM prototype is a correctly rounded elementary function generator. Currently, RLIBM supports the FP and the posit representation. RLIBM uses the MPFR library with up to 1000 precision bits to compute the oracle value of $f(x)$ and round the result to the target representation. Although the Table-maker's dilemma states that it is not feasible to mathematically determine the number of precision necessary to identify the correctly rounded results, prior work has empirically shown that roughly 160 precision bits in the worst case are sufficient to compute the correctly rounded result for the double representation [85]. RLIBM uses SoPlex [50], an exact rational LP solver, to generate the coefficients of the polynomials with a time limit of five minutes. We limit the size of the LP formulation to contain up to fifty thousand reduced input and interval constraints. If the number of intervals in the sample pool exceeds fifty thousand, then we increase the number of subdomains. To generate elementary functions with good performance, the user can provide custom range reduction functions, the degree, and the structure of the polynomial (*i.e.*, odd or even polynomial).

Using RLIBM, we generated several correctly rounded elementary functions for vari-

ous representations. For evaluation purposes, we classify these functions into three math library prototypes, RLIBM-16 [91, 92], RLIBM-32 [94], and RLIBM-ALL, depending on the bit-width of the target representation. RLIBM-16 contains ten elementary functions for bfloat16 with the rne mode and ten functions for posit16. RLIBM-32 contains ten functions for the 32-bit float type with the rne mode and ten functions for posit32. Finally, RLIBM-ALL contains ten functions that produce the correctly rounded result of $f(x)$ for the 34-bit FP representation (*i.e.*, $\mathbb{T}_{n+2}$) with 8 bits of exponent (FP34) in the rno mode. FP34 representation is not supported in hardware. Hence, RLIBM-ALL stores the FP34 result in the double type. RLIBM-ALL's FP functions are designed to produce correct results for all k-bit FP representations with 8 bits of exponent using any IEEE-754 standard rounding modes as long as $9 < k \leq 32$. This includes bfloat16, tensorfloat32, and the 32-bit float. RLIBM-ALL also contains ten posit functions that produce the correctly rounded result of $f(x)$ for the 34-bit posit representation with $es = 2$ with the rno mode. These functions produce correctly rounded results for all k-bit posit representations with $es = 2$ as long as $2 \leq k \leq 32$.

All three prototypes perform range reduction, polynomial evaluation, and output compensation using the double type. Polynomials are evaluated with Horner's method [10] for efficiency and accuracy of the results. The functions in RLIBM-16 use the simple range reduction strategies. RLIBM-32 and RLIBM-ALL use the more sophisticated range reductions described in Chapter 4 to significantly reduce the input domain.

Experiment Methodology. We evaluate the functions in RLIBM-16, RLIBM-32, and RLIBM-ALL on two criteria: (1) the correctness of the output and (2) the performance compared to the state-of-the-art libraries. To measure the performance, we compare our FP functions against glibc's libm [51], Intel's libm [67], CR-LIBM [30], and MetaLibm [80]. Intel's and glibc's libm have elementary functions for float and double types. They are the most widely used math libraries. CR-LIBM has correctly rounded elementary functions for

the double type. CR-LIBM provides multiple implementations for each elementary function to support four IEE-754 standard rounding modes, rne, rnp, rnn, and rnz. MetaLibm provides elementary functions for both the float and the double types which are optimized with AVX2 vector instructions. To produce the results in a target representation $\mathbb{T}$ that is not natively supported by these libraries, we first convert the input in $\mathbb{T}$ to the representation supported by the library, use the elementary function, and round the result back to $\mathbb{T}$. We compare our posit16 functions in RLIBM-16 against SoftPosit-Math [88], a correctly rounded math library for posit16. There are no math libraries for other configurations of the posit representation (*e.g.*, 32-bit posit32). We compare our posit32 functions in RLIBM-32 and the posit functions in RLIBM-ALL against glibc's and Intel's double library as well as CR-LIBM. The double type can exactly represent all posit32 values. We do not compare against existing float libraries because float cannot exactly represent all posit32 values.

Experimental setup. We perform all of our experiments on a 2.10GHz Intel Xeon Gold 6230R machine with 192GB of RAM running Ubuntu 18.04. We disabled Intel turbo boost and hyper-threading to minimize noise. All of our libraries are compiled at the `O3` optimization level. We use Intel's libm from the oneAPI Toolkit and glibc's libm from glibc-2.33. The MetaLibm functions are generated using the optimizations for AVX2 extensions enabled. The test harness for comparing glibc's libm, CR-LIBM, and MetaLibm is built using the `gcc-10` compiler with `-O3 -static -frounding-math -fsignaling-nans` flags. Because Intel's libm is only supported in Intel's compiler, we built the test harness that compares Intel's libm against our libraries using the `icc` compiler. We use the flags `-O3 -static -no-ftz -fp-model strict` to obtain as many accurate results as possible. To measure performance, we measure the number of cycles taken to compute the result for each input using hardware performance counter `rdtscp`. We then measured the total time taken to compute the elementary function as the sum of the time taken by all inputs (*i.e.*, 2^{32} inputs for a 32-bit representation).

7.2 Experimental Evaluation of RLIBM-16

Table 7.1 provides details on the list of elementary functions available in RLIBM-16 and the size of the polynomials generated for each function. Because there are only 2^{16} inputs in 16-bit representations, we aimed to generate polynomial approximations that minimize the memory usage of RLIBM-16 functions. Otherwise, it may be better to simply store the correctly rounded results of $f(x)$ for all 2^{16} inputs in a table. We used range reductions that use a minimal amount of look-up tables. We tried to create the smallest piecewise polynomials instead of the domain-splitting technique described in Chapter 5. The polynomials used RLIBM-16 functions are generated within a few minutes.

7.2.1 Correctness Evaluation of RLIBM-16

Table 7.2(a) reports the result of our experiment to check the correctness of bfloat16 functions in RLIBM-16 and other math libraries. All bfloat16 functions in RLIBM-16 produce correctly rounded bfloat16 results with rne for all inputs. In contrast, we discovered that re-purposing glibc's or Intel's float library did not produce the correctly rounded result for the input $x = -0.0181884765625$ in 10^x function. This case is especially interesting because both glibc and Intel's float library produce the correctly rounded result of 10^x for the float type. However, the float result rounded to bfloat16 is not the correctly rounded result for bfloat16 due to the double rounding error. Thus, repurposing a correctly rounded function for a representation $\mathbb{T}'$ does not necessarily produce the correctly rounded result for another representation $\mathbb{T}$, even if $\mathbb{T}'$ has more precision compared to $\mathbb{T}$. This observation has been one of the motivations for us to create RLIBM-ALL, a math library that guarantees to produce correctly rounded results for multiple precisions and rounding modes. Our experiment showed that repurposing glibc's and Intel's double library produces the correctly rounded results of all inputs for bfloat16. Table 7.2(b) reports that all posit16 functions in RLIBM-16 produce correctly rounded results for all inputs. SoftPosit-Math functions also

Table 7.1: Details on generated polynomials for bfloat16 and posit16 functions. For each elementary function, we report the total time taken to generate the polynomials, the total number of reduced intervals, the number of polynomials generated, the degree of the generated polynomial, and the number of terms in the polynomial.

$f(x)$	Total Time (Seconds)	Reduced Inputs	# of Polynomials	Degree	# of Terms
bfloat16 functions					
$\mathbf{ln(x)}$	0.84	128	1	7	4
$\mathbf{log2(x)}$	8.65	128	1	5	3
$\mathbf{log10(x)}$	1.63	128	1	5	3
$\mathbf{exp(x)}$	2.9	3820	1	4	5
$\mathbf{exp2(x)}$	0.89	1937	1	4	5
$\mathbf{exp10(x)}$	3	3840	1	4	5
$\mathbf{sinh(x)}$	0.27	422	3	5 0 6	3 1 4
$\mathbf{cosh(x)}$	0.27	471	2	5 6	3 4
$\mathbf{sinpi(x)}$	32	16129	2	1 7	1 4
$\mathbf{cospi(x)}$	32.2	16129	3	0 6 0	1 4 1
posit16 functions					
$\mathbf{ln(x)}$	3.32	4096	1	9	5
$\mathbf{log2(x)}$	5.7	4096	1	9	5
$\mathbf{log10(x)}$	6.5	4096	1	9	5
$\mathbf{exp(x)}$	6.2	57371	1	6	7
$\mathbf{exp2(x)}$	5.2	24201	1	6	7
$\mathbf{exp10(x)}$	12	53106	1	6	7
$\mathbf{sinh(x)}$	37.4	13044	2	7 6	4 4
$\mathbf{cosh(x)}$	391.9	14400	4	1 7 6 6	1 4 4 4
$\mathbf{sinpi(x)}$	38	12289	2	1 9	1 5
$\mathbf{cospi(x)}$	85.7	12289	3	0 8 0	1 5 1

Table 7.2: (a) Generation of correctly rounded results for bfloat16 with RLIBM-16, glibc's float libm, and Intel's float libm. Math libraries created by repurposing glibc's double libm and Intel's double libm produce correct bfloat16 results for all inputs. (b) Generation of correctly rounded results for posit16 with RLIBM-16 and SoftPosit-Math. ✓indicates that the library produces the correctly rounded result for all inputs. Otherwise, we use ✗. N/A indicates that the implementation is not available.

Bfloat16 Functions	**Using RLIBM-16**	Using glibc float	Using Intel float
$ln(x)$	✓	✓	✓
$log_2(x)$	✓	✓	✓
$log_{10}(x)$	✓	✓	✓
e^x	✓	✓	✓
2^x	✓	✓	✓
10^x	✓	✗(1)	✗(1)
$sinh(x)$	✓	✓	✓
$cosh(x)$	✓	✓	✓
$sinpi(x)$	✓	N/A	✓
$cospi(x)$	✓	N/A	✓

(a) Correctly rounded results with bfloat16

Posit16 Functions	**Using RLIBM-16**	Using SoftPosit-Math
$ln(x)$	✓	✓
$log_2(x)$	✓	✓
$log_{10}(x)$	✓	N/A
e^x	✓	N/A
2^x	✓	✓
10^x	✓	✓
$sinh(x)$	✓	N/A
$cosh(x)$	✓	N/A
$sinpi(x)$	✓	✓
$cospi(x)$	✓	✓

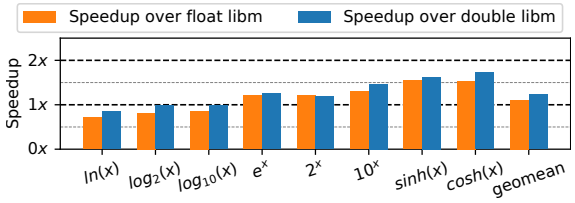
(b) Correctly rounded results with posit16

produce correctly rounded results for the available functions. However, SoftPosit-Math does not contain $log_{10}(x)$, 10^x, $sinh(x)$, and $cosh(x)$.

7.2.2 Performance Evaluation of RLIBM-16

***Performance of bfloat16 functions in* RLIBM-16.** Figure 7.1(a) reports the speedup of bfloat16 functions in RLIBM-16 against glibc's float library (left bar in each cluster) and glibc's double library (right bar in each cluster). On average, RLIBM-16 has $1.1\times$ speedup over glibc's float functions and $1.2\times$ speedup over glibc's double functions. Figure 7.1(b) reports the speedup of bfloat16 functions in RLIBM-16 against Intel's float library (left bar in each cluster) and Intel's double library (right bar in each cluster). On average, RLIBM-16 has $1.4\times$ speedup over Intel's float functions and $1.6\times$ speedup over Intel's double functions. RLIBM-16's bfloat16 functions are faster than all corresponding functions in Intel's double library. RLIBM-16 is faster than glibc's float and double library as well as Intel's float functions except for $ln(x)$, $log_2(x)$, and $log_{10}(x)$. RLIBM-16 aims to generate efficient polynomials that produce correctly rounded results for all inputs while using a

(a) Speedup of RLIBM-16's float functions over glibc libm



(b) Speedup of RLIBM-16's float functions over Intel libm

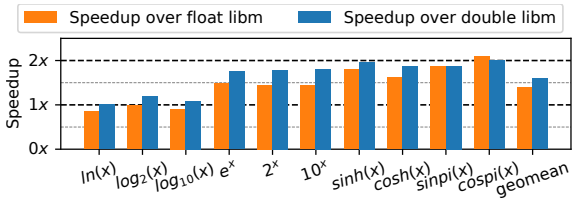


Figure 7.1: (a) Speedup of RLIBM-16's bfloat16 functions compared to glibc's float functions (left) and glibc's double functions (right). (b) Speedup of RLIBM-16's functions compared to Intel's float functions (left) and Intel's double functions (right).

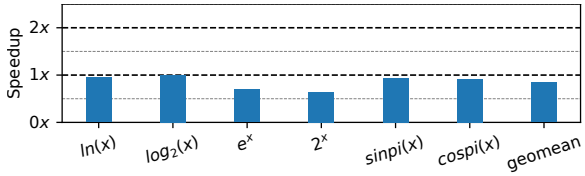


Figure 7.2: Speedup of RLIBM-16's posit16 functions compared to SoftPosit-Math library when the input is available as a double. It avoids the software simulated cast from posit16 to double and vice versa. SoftPosit-Math takes a posit16 input that is internally represented as an integer.

small amount of memory. The $ln(x)$, $log_2(x)$, and $log_{10}(x)$ functions in RLIBM-16 use range reductions that do not require look-up tables to minimize memory usage. These range reductions have higher overhead compared to the range reductions used in glibc and Intel's functions. In all other cases, the lower degree polynomial that RLIBM generates allows our bfloat16 functions to be faster than mainstream math libraries.

***Performance of posit16 functions in* RLIBM-16.** We measure the speedup of RLIBM-16's posit16 functions compared to SoftPosit-Math functions when computing the posit16 result given a posit16 input. We found out that RLIBM-16 can have a significant slowdown compared to SoftPosit-Math. The primary reason for this slowdown is because posit16 inputs are cast to double type before using RLIBM-16. The double result is then rounded to posit16 value. Because common hardware architecture does not support posit representations, the cast operation is done using software simulation. We found out that posit16 functions in RLIBM-16 spend 75% of the time casting values between posit16 and double. In comparison, SoftPosit-Math represents posit16 values internally as an integer and the el-

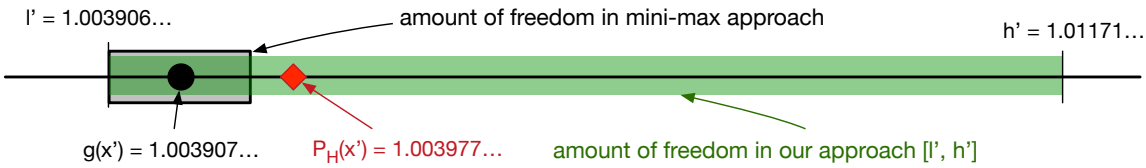


Figure 7.3: An illustration showing that our approach provides more freedom in generating a polynomial that produces correctly rounded results of 10^x for all inputs. The reduced interval $[l', h']$ (shown in green box) corresponds to the reduced input $x' = 0.0056264\ldots$. The real value of $g(x')$ is shown with the black circle. The polynomial generated using our approach produces a value (red diamond) within the rounding interval. If we approximated the real result $g(x')$ instead of the correctly rounded result, the margin of error allowed is much smaller (box with black border).

ementary functions are super optimized with purely integer operations to eliminate the cost of casting. We measured the performance of RLIBM-16 without the cost of this cast, which we report in Figure 7.2. SoftPosit-Math does not have implementations for $log_{10}(x)$, 10^x, $sinh(x)$, and $cosh(x)$. Hence, we report the speedup for the functions available in both SoftPosit-Math and RLIBM-16. On average, RLIBM-16 has 15% slowdown compared to SoftPosit-Math. The $ln(x)$, $log_2(x)$, $sinpi(x)$, and $cospi(x)$ functions in RLIBM-16 have similar performance compared to SoftPosit-Math. However, the super-optimized implementations in SoftPosit-Math show higher performance for e^x and 2^x even though both libraries use similar degree polynomials.

Performance impact of approximating the correctly rounded result. We provide a case study to show that the RLIBM approach in approximating the correctly rounded result provides more freedom in generating efficient polynomial. Consider the elementary function 10^x for bfloat16. The output compensation function we use for 10^x requires us to approximate $g(x') = 2^{x'}$ where the reduced inputs are in the domain $x' \in [0, 1)$. RLIBM generated a 4^{th} degree polynomial that approximates $g(x')$ which produces the correctly rounded result of 10^x for the entire input domain of bfloat16 when used with output compensation functions.

The RLIBM approach provides more freedom to produce low degree polynomials. We illustrate this point with Figure 7.3, which shows the reduced interval $[l', h']$ (green box)

for the reduced input $x' = 0.00562\ldots$. The real value of $g(x')$ is highlighted with the black circle. The real value of $g(x')$ is extremely close to l' with only $\epsilon = |g(x') - l'| \approx 1.31 \times 10^{-6}$ amount of error. If we are to generate a polynomial that approximates the real value of $g(x')$, then the polynomial must have an error less than ϵ, *i.e.* the polynomial must produce a value within $[g(x') - \epsilon, g(x') + \epsilon]$. This tight restriction can force the mini-max approach to generate a higher degree polynomial to produce correctly rounded results. Instead, the polynomial generated with our approach produces a value shown with the red diamond from Figure 7.3. This value has an error of approximately 7.05×10^{-5}, which is much larger than ϵ, but still within the reduced interval. This freedom allows our approach to generate lower degree polynomial, yet produce the correctly rounded results for all bfloat16 inputs when used with output compensation function.

7.3 Experimental Evaluation with RLIBM-32

Table 7.3 provides details on the list of elementary functions available in RLIBM-32 and the size of the polynomials generated for each function. We aimed to generate polynomials with the best performance given a storage budget. Hence, we generated piecewise polynomials where the degree of the polynomial is less than or equal to 7 (at most 8 coefficients) and the number of sub-domains was less than or equal to 2^{14}. Since each coefficient is stored as a double value (8 bytes), the maximum size of the look-up table that stores the coefficients of the polynomial will be 8bytes $\times\ 8 \times 2^{14} = 1$MB. The output compensation function that we use for $sinh(x)$, $cosh(x)$, $sinpi(x)$, and $cospi(x)$ involve two elementary functions (*i.e.*, $g_i(x')$). We generated two piecewise polynomials for these functions. The e^x, 2^x, and 10^x function has both negative and positive reduced inputs. Hence, we created a piecewise polynomial for negative reduced inputs and a piecewise polynomial for positive reduced inputs.

Table 7.3 also reports the amount of time taken to generate float and posit32 functions. The amount of time needed ranges from 19 minutes for $cospi(x)$ for the float type to 25

Table 7.3: Details on the generated polynomials for float and posit32 in RLIBM-32. For each elementary function, we report the time taken to generate the polynomials in minutes, the number of reduced inputs, the size of the piecewise polynomial for approximating $g_i(x')$, the maximum degree of the polynomial, and the number of terms in the polynomial.

$f(x)$	Gen. Time (Minutes)	Reduced Inputs	# of Polynomials	Degree	# of Terms
float functions					
$\mathbf{ln(x)}$	218	7.2E6	2^{10}	3	3
$\mathbf{log_2(x)}$	251	7.2E6	2^{8}	3	3
$\mathbf{log_{10}(x)}$	429	7.2E6	2^{8}	3	3
$\mathbf{e^x}$	117	5.2E8	2^{7} 2^{7}	4 4	5 5
$\mathbf{2^x}$	86	3.0E8	2^{4} 2^{3}	4 4	5 5
$\mathbf{10^x}$	169	5.2E8	2^{6} 2^{7}	4 3	5 4
$\mathbf{sinh(x)}$	28	1.5E8	2^{6}	5	3
$\mathbf{cosh(x)}$	24	1.5E8	2^{6}	4	3
$\mathbf{sinpi(x)}$	30	1.2E8	1	5	3
$\mathbf{cospi(x)}$	19	1.2E8	1	4	3
posit32 functions					
$\mathbf{ln(x)}$	264	1.1E8	2^{11}	4	4
$\mathbf{log_2(x)}$	288	1.1E8	2^{8}	4	4
$\mathbf{log_{10}(x)}$	685	1.1E8	2^{12}	3	3
$\mathbf{e^x}$	1089	3.5E9	2^{12} 2^{12}	3 3	4 4
$\mathbf{2^x}$	814	7.9E8	2^{10} 2^{12}	3 3	4 4
$\mathbf{10^x}$	1528	3.4E9	2^{13} 2^{13}	3 3	4 4
$\mathbf{sinh(x)}$	461	1.6E9	2^{14} 2^{14}	5 4	3 3
$\mathbf{cosh(x)}$	528	1.7E9	2^{14} 2^{12}	3 6	2 4
$\mathbf{sinpi(x)}$	716	1.1E8	2^{12} 2^{13}	3 2	2 2
$\mathbf{cospi(x)}$	342	1.9E8	2^{11} 2^{12}	5 2	3 2

hours for 10^x for the posit type. The majority of the total time is spent in computing the oracle results and the rounding intervals (*i.e.*, 86% and 70% of total time for float and posit32, respectively). It takes significantly longer to generate polynomials for posit32 as a whole. There are fewer special cases in posit32 functions and requires more time to compute the or-

Table 7.4: Generation of correctly rounded results for 32-bit floats with RLIBM-32, Intel's float and double libm, glibc's float and double libm, CR-LIBM, and MetaLibm's float and double libm. ✓indicates that the library produces the correctly rounded result for all inputs. Otherwise, we use ✗. For each ✗, we show the number of inputs with wrong results.

float functions	**Using RLIBM-32**	Using glibc float	Using glibc double	Using Intel float	Using Intel double
$\mathbf{ln(x)}$	✓	✗(4.2E5)	✗(5)	✗(1060)	✗(5)
$\mathbf{log_2(x)}$	✓	✗(3.1E5)	✓	✗(276)	✓
$\mathbf{log_{10}(x)}$	✓	✗(3.0E7)	✗(1)	✗(1.5E5)	✗(1)
$\mathbf{e^x}$	✓	✗(1.7E5)	✓	✗(2.5E5)	✓
$\mathbf{2^x}$	✓	✗(1.7E5)	✗(2)	✗(7.2E5)	✗(2)
$\mathbf{10^x}$	✓	✗(1.7E5)	✓	✗(3.9E5)	✓
$\mathbf{sinh(x)}$	✓	✗(7.1E7)	✗(2)	✗(2.5E5)	✗(2)
$\mathbf{cosh(x)}$	✓	✗(1.8E7)	✓	✗(1.4E5)	✓
$\mathbf{sinpi(x)}$	✓	N/A	N/A	✗(3.4E5)	✓
$\mathbf{cospi(x)}$	✓	N/A	N/A	✗(3.8E5)	✓

float functions	Using CR-LIBM	Using MetaLibm float	Using MetaLibm double
$\mathbf{ln(x)}$	✗(5)	N/A	N/A
$\mathbf{log_2(x)}$	✓	N/A	N/A
$\mathbf{log_{10}(x)}$	✗(1)	N/A	N/A
$\mathbf{e^x}$	✓	✗(5.1E8)	✗(5.1E8)
$\mathbf{2^x}$	N/A	✗(6.5E7)	✗(1026)
$\mathbf{10^x}$	N/A	N/A	N/A
$\mathbf{sinh(x)}$	✗(2)	N/A	N/A
$\mathbf{cosh(x)}$	✓	✗(1.1E7)	✓
$\mathbf{sinpi(x)}$	✓	N/A	N/A
$\mathbf{cospi(x)}$	✓	N/A	N/A

acle results. Additionally, posit32 has higher precision compared to the 32-bit float and has saturating behavior with extremal values. Hence, RLIBM-32 generates larger piecewise polynomials. Nonetheless, our domain splitting and counterexample guided polynomial generation techniques have been vital in generating low degree polynomials that produce the correctly rounded results for all 2^{32} inputs in 32-bit representations.

7.3.1 Correctness Evaluation of RLIBM-32

Correctness of float functions. Table 7.4 shows the result of our experiment to check the correctness of float functions in RLIBM-32 and other math libraries. All ten elementary

Table 7.5: Generation of correctly rounded results with posit32 functions for all inputs by RLIBM-32, Intel and glibc's double libraries, and CR-LIBM. ✓indicates that the library produces the correctly rounded result for all inputs and otherwise, we use ✗.

posit32 functions	**Using RLIBM-32**	Using glibc double	Using Intel double	Using CR-LIBM
$\mathbf{ln(x)}$	✓	✗(22)	✗(22)	✗(22)
$\mathbf{log_2(x)}$	✓	✗(19)	✗(18)	✗(18)
$\mathbf{log_{10}(x)}$	✓	✗(26)	✗(23)	✗(23)
$\mathbf{e^x}$	✓	✗(4.4E8)	✗(4.4E8)	✗(4.4E8)
$\mathbf{2^x}$	✓	✗(4.0E8)	✗(4.0E8)	N/A
$\mathbf{10^x}$	✓	✗(5.2E8)	✗(5.2E8)	N/A
$\mathbf{Sinh(x)}$	✓	✗(4.4E8)	✗(4.4E8)	✗(4.4E8)
$\mathbf{Cosh(x)}$	✓	✗(4.4E8)	✗(4.4E8)	✗(4.4E8)
$\mathbf{Sinpi(x)}$	✓	N/A	✗(70)	✗(70)
$\mathbf{Cospi(x)}$	✓	N/A	✗(90)	✗(90)

functions in RLIBM-32 produce the correctly rounded result in the 32-bit float with rne for all inputs. In contrast, the elementary functions in glibc, Intel, and MetaLibm's float libraries do not produce correct results for all inputs. Several functions in glibc and MetaLibm's float library produce wrong results for millions of inputs while Intel's float library produces wrong results for thousands of inputs. When repurposed for the 32-bit float, the double functions from glibc, Intel, and CR-LIBM do not produce correct float results for $ln(x)$, $log_{10}(x)$, 2^x, and $sinh(x)$ due to double rounding error (*i.e.*, rounding the real value of $f(x)$ to double and subsequently rounding the result to float). Even when $f(x)$ is approximated accurately enough to produce the correctly rounded double value (*i.e.*, CR-LIBM functions), rounding the double result to float may not produce the correctly rounded float result. Finally, the functions in MetaLibm do not produce correct float results even when MetaLibm uses Sollya [21], the same tool used by CR-LIBM to generate polynomials that produce correctly rounded double results.

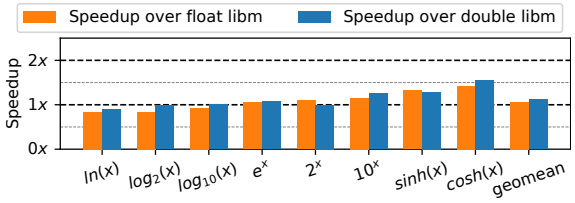
Correctness of posit32 functions. Table 7.5 shows that all ten posit functions in RLIBM-32 produce correctly rounded results in posit32 for all inputs. All posit32 values cannot be exactly represented in float representation but they can be represented in the double

representation. Hence, we created posit32 math libraries by repurposing glibc and intel's double math library as well as CR-LIBM. We tested the ability of the repurposed libraries to produce correct posit32 results for the ten posit functions available in RLIBM-32. These libraries do not produce correct results for posit32 functions. Contrary to float results, they produce wrong results for millions of inputs, especially for exponential and hyperbolic functions. The primary reason for such a high number of wrong results is the different rounding behaviors between the FP representation with rne and the posit representations. Posit representations do not underflow to 0 or overflow to ∞. Instead, extremely large values are rounded to the largest representable posit value. Similarly, values extremely close, but not equal, to 0 are rounded to the smallest non-zero representable posit value. Hence, all repurposed libraries produce wrong results when the real value of $f(x)$ is either extremely large or close to zero.

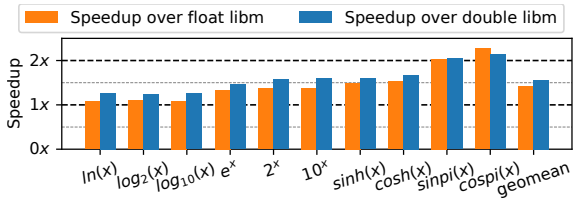
7.3.2 Performance Evaluation of RLIBM-32

Performance of float functions. Figure 7.4(a) reports the speedup of RLIBM-32's float functions compared to glibc's float library (left bar in each cluster) and glibc's double library (right bar in each cluster). On average, RLIBM-32 has $1.1\times$ speedup over glibc's float library and $1.1\times$ speedup over glibc's double library. Figure 7.4(b) reports the speedup of RLIBM-32 compared to Intel's float library (left bar in each cluster) and Intel's double library (right bar in each cluster). On average, RLIBM-32 has $1.4\times$ and $1.6\times$ speedup over Intel's float and double library, respectively. Intel's libraries produce correctly rounded float results for more inputs compared to glibc's libraries. Hence, RLIBM-32 has more speedup compared to Intel's libraries. Figure 7.4(c) reports the speedup of RLIBM-32 compared to CR-LIBM. RLIBM-32 has $1.9\times$ speedup over CR-LIBM on average. CR-LIBM produces correctly rounded double results for all double inputs. Thus, RLIBM-32 has a higher speedup over CR-LIBM compared to glibc or Intel's libraries. Finally, Figure 7.4(d) reports the speedup of RLIBM-32 compared to MetaLibm's float library (left

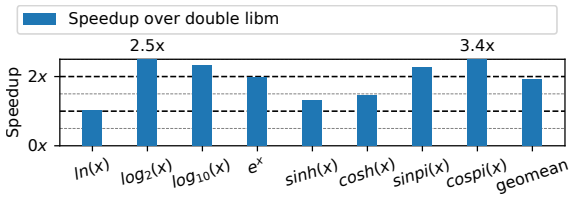
(a) Speedup of RLIBM-32's float functions over glibc libm



(b) Speedup of RLIBM-32's float functions over Intel libm



(c) Speedup of RLIBM-32's float functions over CR-LIBM



(d) Speedup of RLIBM-32's float functions over MetaLibm

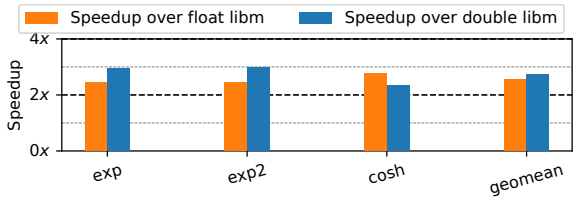
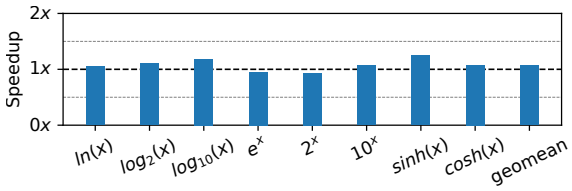


Figure 7.4: (a) Speedup of RLIBM-32's float functions compared to glibc's float functions (left) and glibc's double functions (right). (b) Speedup of RLIBM-32's functions compared to Intel's float functions (left) and Intel's double functions (right). (c) Speedup of RLIBM-32's functions compared to CR-LIBM functions. (d) Speedup of RLIBM-32's functions compared to MetaLibm's float functions (left) and double functions (right) built with AVX2 optimizations.

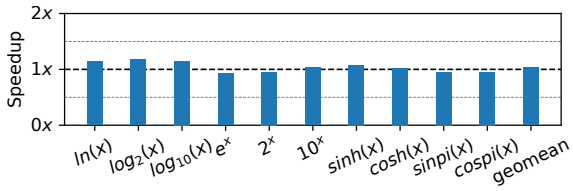
bar in each cluster) and double library (right bar in each cluster). On average, RLIBM-32 has $2.5\times$ and $2.7\times$ speedup compared to MetaLibm. RLIBM-32's float functions are faster than all corresponding functions in Intel's library, CR-LIBM, and MetaLibm. RLIBM-32 is faster than glibc's library except for $ln(x)$, $log_2(x)$ and $log_{10}(x)$ functions. However, glibc's functions do not produce correctly rounded results for all inputs. In all other cases, RLIBM-32 produces correctly rounded float results in rne rounding mode and is more efficient. RLIBM's approach in generating piecewise polynomials for RLIBM-32 with low degree polynomials has been instrumental in creating efficient elementary functions.

Performance of posit functions. The graphs in Figure 7.5 reports the speedup of RLIBM-32's posit functions compared to the posit32 math libraries created by repurposing glibc's double library, Intel's double library, and CR-LIBM. On average, RLIBM-32 has $1.1\times$, $1\times$, and $1.5\times$ speedup over glibc's library, Intel's library, and CR-LIBM. All three repurposed math libraries do not produce correctly rounded posit32 results for all inputs. RLIBM-32

(a) Speedup of RLIBM-32's posit32 functions over glibc libm



(b) Speedup of RLIBM-32's posit32 functions over Intel libm



(c) Speedup of RLIBM-32's posit32 functions over CR-LIBM

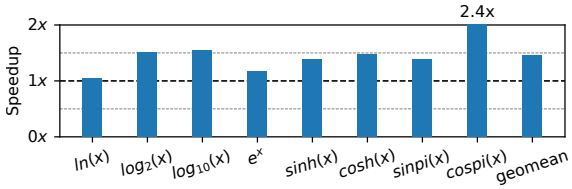


Figure 7.5: (a) Speedup of RLIBM-32's posit32 functions compared to glibc's double functions. (b) Speedup of RLIBM-32's posit32 functions compared to Intel's double functions. (c) Speedup of RLIBM-32's posit32 functions compared to CR-LIBM functions.

is the first correctly rounded math library for posit32 types and performs similar to or faster than the repurposed libraries.

Performance impact of piecewise polynomials. We present a case study to show the performance benefits of using piecewise polynomials. There are four 32-bit float functions $log_2(x)$, $log_{10}(x)$, $sinpi(x)$, and $cospi(x)$ where RLIBM could generate a single polynomial and produce correctly rounded results for all inputs. We created piecewise polynomials for these functions with an increasing number of sub-domains. We validated each piecewise polynomials to ensure that they produce correctly rounded results for all inputs. We measured the change in performance as the number of sub-domains in the piecewise polynomial increases from a single polynomial (*i.e.*, 2^0) to 2^{12}. Figure 7.6 reports the performance of $log_2(x)$ and $log_{10}(x)$ with increasing number of sub-domains compared to the performance of a single polynomial. Figure 7.6 does not show $sinpi(x)$ and $cospi(x)$ because we found that using a single polynomial has the best performance.

Initially, there is a small decrease in the performance (14% overhead) from a single polynomial to a piecewise polynomial because the degree of the polynomial does not de-

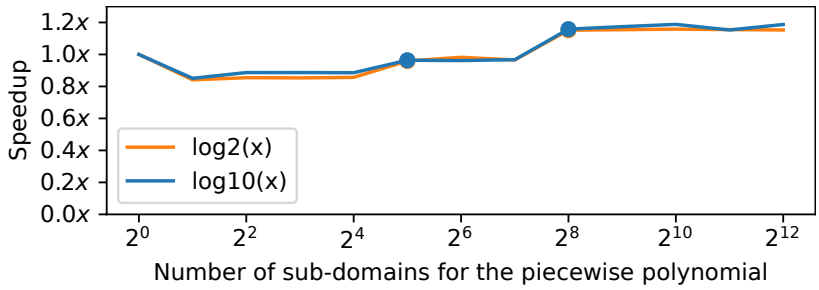


Figure 7.6: Speedup of $log2(x)$ and $log10(x)$ function for the 32-bit float with increasing size of piecewise polynomial approximations compared to a single polynomial generated using our approach. All polynomials produce the correctly rounded result for all inputs when used with the same output compensation function. A circle represents a decrease in the degree of the polynomial.

crease and the implementations experience overhead when using look-up tables. However, when we keep increasing the number of sub-domains, the degree of the polynomials starts to decrease, and the performance increases. When the number of sub-domains increases from 2^4 to 2^5, the degree of the polynomials decreases from 5 to 4 and we only see approximately 5% overhead compared to a single polynomial. At 2^8 sub-domains, the degree of the polynomials decrease to 3 and we achieve $1.15\times$ speedup compared to a single polynomial. A piecewise polynomial containing 2^8 polynomials with degree 3 requires 6KB to store all coefficients.

7.4 Experimental Evaluation With RLIBM-ALL

Table 7.6 provide details on the list of elementary functions available in RLIBM-ALL, the total time taken to generate the polynomial approximations, and the size of the piecewise polynomials generated for each function. For these generic functions in RLIBM-ALL, we restricted the degree of the piecewise polynomials to 8 degrees at most. We also aimed for smaller than or equal to 2^{17} sub-domains. We increased the storage budget for RLIBM-ALL compared to RLIBM-32 to account for the additional precision that RLIBM-ALL requires. As RLIBM-ALL produces piecewise polynomials for 34-bit representations, the number of sub-domains used in the resulting piecewise polynomials are bigger than RLIBM-32. However, the degree of the polynomial in each sub-domain is similar to RLIBM-32.

Table 7.6: Details about the generated polynomials. For each elementary function, we show the time taken to generate the polynomials in minutes, the size of the piecewise polynomial for approximating $g_i(r)$, the maximum degree of the polynomial, and the number of terms in the polynomial.

floating point functions				
$f(x)$	Gen. Time (Min.)	# of Polynomials	Degree	# of Terms
$\mathbf{ln(x)}$	325	2^{10}	3	3
$\mathbf{log_2(x)}$	420	2^8	3	3
$\mathbf{log_{10}(x)}$	546	2^8	3	3
$\mathbf{e^x}$	241	2^7 2^7	4 4	5 5
$\mathbf{2^x}$	151	2^7 2^7	3 3	4 4
$\mathbf{10^x}$	402	2^8 2^8	3 3	4 4
$\mathbf{sinh(x)}$ $\mathbf{cosh(x)}$	143 135	2^6 2^5	5 4	3 3
$\mathbf{sinpi(x)}$ $\mathbf{cospi(x)}$	308 316	2^2 2^2	5 4	3 3

posit functions				
$f(x)$	Gen. Time (Min.)	# of Polynomials	Degree	# of Terms
$\mathbf{ln(x)}$	262	2^{13} 2^{12}	3 3	3 3
$\mathbf{log_2(x)}$	296	2^{11} 2^{11}	3 3	3 3
$\mathbf{log_{10}(x)}$	419	2^{14} 2^{14}	3 3	3 3
$\mathbf{e^x}$	1048	2^{14} 2^{14}	2 3	3 4
$\mathbf{2^x}$	1179	2^{14} 2^{15}	2 2	3 3
$\mathbf{10^x}$	1825	2^{17} 2^{15}	2 2	3 3
$\mathbf{sinh(x)}$	652	2^{15} 2^{15}	3 2	2 2
$\mathbf{cosh(x)}$	603	2^{15} 2^{15}	3 2	2 2
$\mathbf{sinpi(x)}$	410	2^{14} 2^{13}	3 4	2 3
$\mathbf{cospi(x)}$	745	2^{15} 2^{15}	3 2	2 2

Table 7.6 also reports the amount of time taken to generate RLIBM-ALL functions in the second column. The amount of time needed ranges from roughly 2 hours to generate $cosh(x)$ function for the FP representation to 30 hours to generate 10^x function for the posit representation. For FP functions, roughly 79% of the total time is spent in computing the oracle results using the MPFR library. In comparison, computing the intervals and generating the polynomials takes 15% and 5% of the total time on average, respectively. Because the posit functions have fewer special cases and require more precision, RLIBM spends more time computing the intervals (23% of the total time) and generating larger piecewise polynomials (39% of the total time). The remaining 38% of the total time is spent in computing the oracle result using MPFR and correctly rounding the result to 34-bit posit value with rno mode.

Table 7.7: (a) RLIBM-ALL's FP functions produce correctly rounded results in the 34-bit floating point representation (*i.e.*, $\mathbb{F}_{34,8}$) using the rno rounding mode for all inputs. (b) Similarly, RLIBM-ALL's posit functions produce correctly rounded results in the 34-bit posit representation (*i.e.*, $\mathbb{P}_{34,2}$) using the rno rounding mode for all inputs. ✓indicates that the library produces the correctly rounded 34-bit result using rno for all inputs. Otherwise, we use ✗.

$f(x)$	RLIBM-ALL produces correct 34-bit FP results with rno?
$\mathbf{ln(x)}$	✓
$\mathbf{log_2(x)}$	✓
$\mathbf{log_{10}(x)}$	✓
$\mathbf{e^x}$	✓
$\mathbf{2^x}$	✓
$\mathbf{10^x}$	✓
$\mathbf{sinh(x)}$	✓
$\mathbf{cosh(x)}$	✓
$\mathbf{sinpi(x)}$	✓
$\mathbf{cospi(x)}$	✓

(a)

$f(x)$	RLIBM-ALL produces correct 34-bit posit results with rno?
$\mathbf{ln(x)}$	✓
$\mathbf{log_2(x)}$	✓
$\mathbf{log_{10}(x)}$	✓
$\mathbf{e^x}$	✓
$\mathbf{2^x}$	✓
$\mathbf{10^x}$	✓
$\mathbf{sinh(x)}$	✓
$\mathbf{cosh(x)}$	✓
$\mathbf{sinpi(x)}$	✓
$\mathbf{cospi(x)}$	✓

(b)

7.4.1 Correctness Evaluation of RLIBM-ALL

Table 7.7(a) reports that the FP functions in RLIBM-ALL produces the correctly rounded results in FP34 with the rno mode for all inputs. By Theorem 6.1 in Chapter 6, our experiment indicates that the FP functions in RLIBM-ALL produce the correct results for all k-bit FP representations with 8 bits of exponent using any IEEE-754 rounding modes as long as $9 < k \leq 32$. These representations include bfloat16, tensorfloat32, and float representations. Similarly, Table 7.7(b) reports that all posit functions in RLIBM-ALL produces the correctly rounded results in the 34-bit posit representation with $es = 2$ using the rno mode. Hence, the posit functions are guaranteed to produce the correct results for all k-bit posit representation as long as $2 \leq k \leq 34$ and $es = 2$, including the standard posit32 representation.

In addition, we wanted to check if RLIBM-ALL's functions can produce correctly rounded results for other representations. Hence, we built a test harness to check if the FP functions in RLIBM-ALL produce correctly rounded results for all inputs with different FP representations where the number of exponent bits ranges from 2 to 8 bits and the

number of mantissa bits ranges from 1 to 23 bits (*i.e.*, $23 \times 7 = 161$). These configurations include the standard 16-bit half type. Our experiment shows that RLIBM-ALL produces the correct results for all 161 representations with all standard rounding modes. RLIBM-ALL is the first library that provides the correctly rounded results for all inputs in hundreds of FP representations with any standard rounding modes. Similarly, we built a test harness to check and verify that RLIBM-ALL's posit functions produce correctly rounded results for all inputs in the standard 16-bit posit representation (*i.e.*, posit16).

Next, we evaluate the ability of various state-of-the-art math libraries to produce correctly rounded results in float, tensorfloat32, and bfloat16 using the five IEEE-754 standard rounding modes. The glibc's and Intel's float and double libraries only have one implementation per elementary function. Hence, we use the same function for all five rounding modes. In contrast, CR-LIBM has four implementations for each elementary function that produces correctly rounded double results with rne, rnz, rnp, and rnn, respectively. Hence, when computing the correctly rounded result for a given rounding mode rm using CR-LIBM, we use the implementation that corresponds to rm.

Correctly rounded results with all rounding modes for float. Table 7.8 reports the result of our experiment to check whether the existing libraries produce correctly rounded float results. While RLIBM-ALL produces the correct float results will all five rounding modes for all inputs, many elementary functions in glibc and Intel's libm do not produce correctly rounded results for all inputs with all rounding modes. In particular, even glibc and Intel's double functions do not produce correct float results of e^x, 10^x, $sinh(x)$, and $cosh(x)$ for all inputs with rnn, rnp, and rnz modes. These results are especially interesting because the error occurs due to the different behaviors of rnn, rnp, and rnz mode compared to rne. For example, e^x approaches infinity for large positive inputs. Both glibc and Intel's double libraries produce ∞ for these inputs, which is the correct float result of e^x in the rne mode. However, this result is incorrect for rnn or rnz mode because the result of e^x should

Table 7.8: Generation of correctly rounded results for 32-bit float using the five standard IEEE-754 rounding modes rne, rnn, rnp, rnz, and rna. We use the elementary functions from RLIBM-ALL, glibc's libm (float and double), Intel's libm (float and double), RLibm32, and CR-LIBM. If the math library produces a double value, we round the output to a float value. ✓indicates that the library produces the correctly rounded float result using a given rounding mode for all inputs. Otherwise, we use ✗.

	Using RLIBM-ALL					Using glibc float libm					Using glibc double libm				
$f(x)$	rne	rnn	rnp	rnz	rna	rne	rnn	rnp	rnz	rna	rne	rnn	rnp	rnz	rna
$\mathbf{ln(x)}$	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
$\mathbf{log_2(x)}$	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓
$\mathbf{log_{10}(x)}$	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
$\mathbf{e^x}$	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
$\mathbf{2^x}$	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
$\mathbf{10^x}$	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
$\mathbf{sinh(x)}$	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
$\mathbf{cosh(x)}$	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
$\mathbf{sinpi(x)}$	✓	✓	✓	✓	✓	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$\mathbf{cospi(x)}$	✓	✓	✓	✓	✓	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

	Using Intel float libm					Using Intel double libm					Using CRLIBM				
$f(x)$	rne	rnn	rnp	rnz	rna	rne	rnn	rnp	rnz	rna	rne	rnn	rnp	rnz	rna
$\mathbf{ln(x)}$	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	N/A
$\mathbf{log_2(x)}$	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	N/A
$\mathbf{log_{10}(x)}$	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	N/A
$\mathbf{e^x}$	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓	✓	N/A
$\mathbf{2^x}$	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	N/A	N/A	N/A	N/A	N/A
$\mathbf{10^x}$	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	N/A	N/A	N/A	N/A	N/A
$\mathbf{sinh(x)}$	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	N/A
$\mathbf{cosh(x)}$	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓	✓	N/A
$\mathbf{sinpi(x)}$	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓	N/A
$\mathbf{cospi(x)}$	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓	✓	✓	✓	N/A

	Using RLibm32				
$f(x)$	rne	rnn	rnp	rnz	rna
$\mathbf{ln(x)}$	✓	✗	✗	✗	✓
$\mathbf{log_2(x)}$	✓	✗	✗	✗	✓
$\mathbf{log_{10}(x)}$	✓	✗	✗	✗	✓
$\mathbf{e^x}$	✓	✗	✗	✗	✓
$\mathbf{2^x}$	✓	✗	✗	✗	✗
$\mathbf{10^x}$	✓	✗	✗	✗	✓
$\mathbf{sinh(x)}$	✓	✗	✗	✗	✓
$\mathbf{cosh(x)}$	✓	✗	✗	✗	✓
$\mathbf{sinpi(x)}$	✓	✗	✗	✗	✓
$\mathbf{cospi(x)}$	✓	✗	✗	✗	✓

round down and produce the largest representable positive value in float instead. Similarly, e^x approaches zero for small negative numbers. Both glibc and Intel's double libraries produce 0 for these inputs which is the correct float result of e^x in the rne mode. However,

this result is incorrect for rnp because the result of e^x should round up and produce the smallest representable positive value in float.

In contrast, CR-LIBM's functions for rnn, rnp, and rnz mode produce the correctly rounded float results with rnn, rnp, and rnz, respectively. Double rounding with these rounding modes always produces correctly rounded results. For example, rounding a real value $v_{\mathbb{R}}$ to double with rnn and then subsequently rounding the result to float with rnn produces the same result as if rounding $v_{\mathbb{R}}$ directly to float with rnn mode. This is not true for rne mode. CR-Libm's functions for rne do not produce the correctly rounded float results with rne mode for all inputs due to double rounding error. Additionally, CR-LIBM does not provide implementations for the rna mode. RLIBM-ALL's elementary function guarantees to provide correctly rounded results for all five rounding modes.

RLIBM-32, which produces correctly rounded float results in the rne rounding mode also produces correctly rounded results in the rna rounding mode for all functions except 2^x. The rna rounding mode behaves similar to rna except when the value is equal to the midpoint between two adjacent float values. In comparison, RLIBM-32 does not produce correct results for all inputs with rnn, rnp, or rnz modes.

Correctly rounded results with all rounding modes for tensorfloat32 and bfloat16. Table 7.9 reports the result of our experiment to check whether existing libraries produce correctly rounded tensorfloat32 results. As tensorfloat32 has the same number of exponent bits as FP34, RLIBM-ALL produces the correctly rounded results for all inputs and all rounding modes. In contrast, glibc and Intel's library, as well as RLIBM-32 does not produce correctly round results in tensorfloat32 for all inputs and all rounding modes. Even when glibc and Intel's double libraries are designed to produce double results which have significantly higher precision than tensorfloat32, they produce wrong results with extremal values for rnn, rnp, and rnz mode, due to double rounding error. CR-LIBM's functions for each rounding mode produce correctly rounded tensorfloat32 results for all available

Table 7.9: Generation of correctly rounded results for TensorFloat32 using the five standard IEEE-754 rounding modes rne, rnn, rnp, rnz, and rna. We use the elementary functions from RLIBM-ALL, glibc's libm (float and double), Intel's libm (float and double), RLibm32, and CR-LIBM. Then, we convert the output to TensorFloat32 values. ✓indicates that the library produces the correctly rounded TensorFloat32 result using a given rounding mode for all inputs. Otherwise, we use ✗.

	Using RLIBM-ALL					Using glibc float libm					Using glibc double libm				
$f(x)$	rne	rnn	rnp	rnz	rna	rne	rnn	rnp	rnz	rna	rne	rnn	rnp	rnz	rna
$\ln(x)$	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓
$\log_2(x)$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\log_{10}(x)$	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓
e^x	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
2^x	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗	✓
10^x	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✓
$\sinh(x)$	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
$\cosh(x)$	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
$\text{sinpi}(x)$	✓	✓	✓	✓	✓	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$\text{cospi}(x)$	✓	✓	✓	✓	✓	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

	Using Intel float libm					Using Intel double libm					Using CRLIBM				
$f(x)$	rne	rnn	rnp	rnz	rna	rne	rnn	rnp	rnz	rna	rne	rnn	rnp	rnz	rna
$\ln(x)$	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	N/A
$\log_2(x)$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	N/A
$\log_{10}(x)$	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	N/A
e^x	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓	✓	N/A
2^x	✗	✗	✗	✗	✓	✓	✗	✗	✗	✓	N/A	N/A	N/A	N/A	N/A
10^x	✓	✗	✗	✗	✗	✓	✗	✗	✗	✓	N/A	N/A	N/A	N/A	N/A
$\sinh(x)$	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓	✓	N/A
$\cosh(x)$	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓	✓	N/A
$\text{sinpi}(x)$	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	N/A
$\text{cospi}(x)$	✓	✗	✗	✗	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	N/A

	Using RLibm32				
$f(x)$	rne	rnn	rnp	rnz	rna
$\ln(x)$	✗	✗	✗	✗	✗
$\log_2(x)$	✓	✓	✓	✓	✓
$\log_{10}(x)$	✗	✗	✗	✗	✗
e^x	✗	✗	✗	✗	✗
2^x	✗	✗	✗	✗	✓
10^x	✓	✗	✗	✗	✗
$\sinh(x)$	✗	✗	✗	✗	✗
$\cosh(x)$	✗	✗	✗	✗	✗
$\text{sinpi}(x)$	✓	✗	✗	✗	✓
$\text{cospi}(x)$	✓	✗	✗	✗	✓

elementary functions and rounding modes.

Table 7.10 presents the result of our experiment to check the ability of existing libraries to produce correctly rounded bfloat16 results. Similar to tensorfloat32, RLIBM-ALL pro-

Table 7.10: Generation of correctly rounded results for bfloat16 using the five standard IEEE-754 rounding modes rne, rnn, rnp, rnz, and rna. We show the results with the elementary functions from RLIBM-ALL, glibc's libm (float and double), Intel's libm (float and double), RLibm, RLibm32, and CR-LIBM. ✓indicates that the library produces the correctly rounded bfloat16 result using a given rounding mode for all inputs. Otherwise, we use ✗.

	Using RLIBM-ALL					Using glibc float libm					Using glibc double libm				
$f(x)$	rne	rnn	rnp	rnz	rna	rne	rnn	rnp	rnz	rna	rne	rnn	rnp	rnz	rna
$\mathbf{ln(x)}$	✓	✓	✓	✓	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓	✓
$\mathbf{log_2(x)}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\mathbf{log_{10}(x)}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\mathbf{e^x}$	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓
$\mathbf{2^x}$	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓
$\mathbf{10^x}$	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
$\mathbf{sinh(x)}$	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓
$\mathbf{cosh(x)}$	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓
$\mathbf{sinpi(x)}$	✓	✓	✓	✓	✓	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$\mathbf{cospi(x)}$	✓	✓	✓	✓	✓	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

	Using Intel float libm					Using Intel double libm					Using CRLIBM				
$f(x)$	rne	rnn	rnp	rnz	rna	rne	rnn	rnp	rnz	rna	rne	rnn	rnp	rnz	rna
$\mathbf{ln(x)}$	✓	✗	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	N/A
$\mathbf{log_2(x)}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	N/A
$\mathbf{log_{10}(x)}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	N/A
$\mathbf{e^x}$	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	N/A
$\mathbf{2^x}$	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓	N/A	N/A	N/A	N/A	N/A
$\mathbf{10^x}$	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	N/A	N/A	N/A	N/A	N/A
$\mathbf{sinh(x)}$	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	N/A
$\mathbf{cosh(x)}$	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	N/A
$\mathbf{sinpi(x)}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	N/A
$\mathbf{cospi(x)}$	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	N/A

	Using RLibm32					Using RLibm				
$f(x)$	rne	rnn	rnp	rnz	rna	rne	rnn	rnp	rnz	rna
$\mathbf{ln(x)}$	✓	✗	✗	✓	✗	✓	✗	✗	✗	✓
$\mathbf{log_2(x)}$	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓
$\mathbf{log_{10}(x)}$	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓
$\mathbf{2^x}$	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓
$\mathbf{2^x}$	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗
$\mathbf{10^x}$	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
$\mathbf{sinh(x)}$	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓
$\mathbf{cosh(x)}$	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓
$\mathbf{sinpi(x)}$	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓
$\mathbf{cospi(x)}$	✓	✗	✓	✗	✓	✓	✗	✗	✗	✓

duces the correctly rounded bfloat16 results for all rounding modes. Many elementary functions in Glibc's library, Intel's library, RLIBM-32, and RLIBM-16 do not produce correctly rounded bfloat16 results especially with rnn, rnp, and rnz mode. CR-LIBM's functions, on the other hand, produces correctly rounded results for all inputs in bfloat16

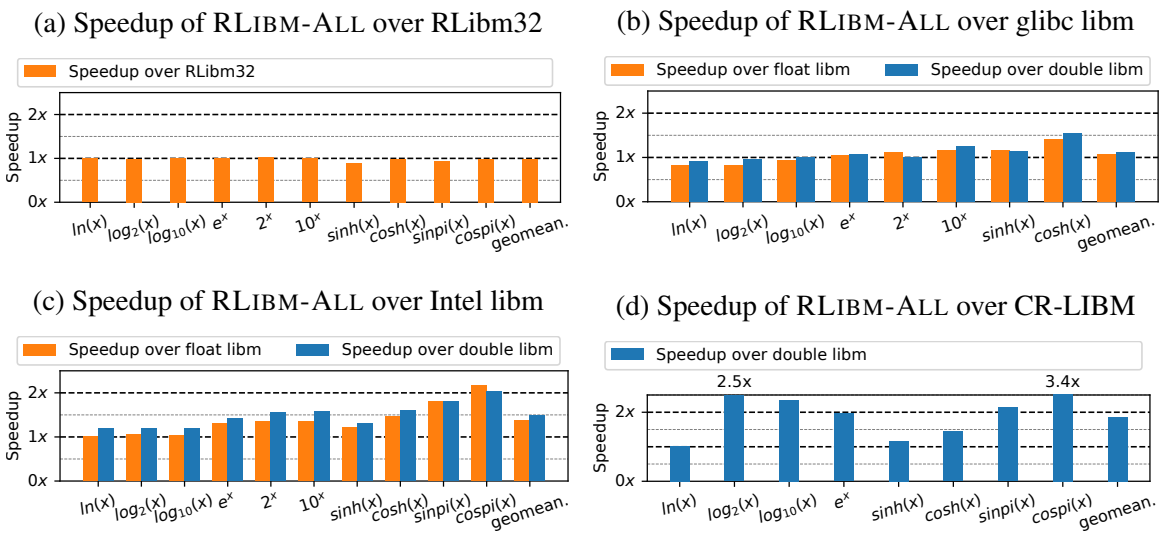


Figure 7.7: (a) Speedup of RLIBM-ALL functions compared to RLibm32 functions when producing 32-bit float results. (b) Speedup of RLIBM-ALL functions compared to Intel's float functions (left) and Intel's double functions (right) when producing 32-bit float results. (c) Speedup of RLIBM-ALL functions compared to CR-LIBM functions when producing 32-bit float results. (d) Speedup of RLIBM-ALL functions compared to glibc's float functions (left) and glibc's double functions (right) when producing 32-bit float results.

with the four available rounding modes.

7.4.2 Performance Evaluation

Performance evaluation of* RLIBM-ALL *when producing float results. Figure 7.7 presents the speedup of RLIBM-ALL's FP functions compared to several math libraries when producing float results with rne mode. Figure 7.7(a) reports the speedup of RLIBM-ALL's FP functions over RLIBM-32's functions. RLIBM-ALL is as fast as RLIBM-32 (2% slower than RLIBM-32) while being able to produce correctly rounded results for multiple representations with all standard rounding modes. RLIBM-ALL's $sinh(x)$ function is 12% slower than to RLIBM-32. When input x is near 0, $sinh(x)$ exhibits a linear behavior and $sinh(x) \approx x$ produces correctly rounded float values with rne rounding mode. RLIBM-32 uses this property by simply returning x for inputs near 0. In comparison, RLIBM-ALL spends significantly more computation to ensure that it produces correctly rounded results for all rounding modes

Figure 7.7(b) presents the speedup of RLIBM-ALL's FP functions over glibc's float functions (left bar in each cluster) and double functions (right bar in each cluster). RLIBM-ALL has $1.1\times$ and $1.1\times$ speedup over glibc's float and double functions, respectively. Figure 7.7(c) presents the speedup of RLIBM-ALL's FP functions over Intel's float functions (left bar in each cluster) and double functions (right bar in each cluster). RLIBM-ALL has $1.3\times$ and $1.5\times$ speedup over Intel's float and double functions, respectively. Figure 7.7(d) presents the speedup of RLIBM-ALL's FP functions over CR-LIBM functions. RLIBM-ALL has $1.9\times$ speedup over CR-LIBM functions. RLIBM-ALL is comparable or faster than the existing libraries while producing correct float results with all rounding modes. In comparison, glibc's libm, Intel's libm, and CR-LIBM do not produce correct results for all inputs when they are used to produce float values.

Performance evaluation of* RLIBM-ALL *when producing bfloat16 values. Figure 7.7 reports the speedup of RLIBM-ALL's FP functions compared to several math libraries when producing bfloat16 results with rne mode. Figure 7.8(a) presents the speedup of RLIBM-ALL's FP functions over RLIBM-32. On average, RLIBM-ALL's FP functions are 8% slower than RLIBM-32 when producing bfloat16. While RLIBM-32 produces float values which can be efficiently rounded to bfloat16 using bit-wise operations, RLIBM-ALL produces double values. Because there is no hardware support for correctly rounding a double value to bfloat16, RLIBM-ALL simulates this rounding with software simulation. Hence, there is an additional 6% slowdown when RLIBM-ALL produces bfloat16 results compared to RLIBM-32.

Figure 7.8(b) reports the speedup of RLIBM-ALL's FP functions over glibc's float functions (left bar in each cluster) and glibc's double functions (right bar in each cluster). RLIBM-ALL has $1\times$ and $1.1\times$ speedup over glibc's float and double library, respectively. Figure 7.8(c) reports the speedup of RLIBM-ALL's FP functions over Intel's float functions (left bar in each cluster) and Intel's double functions (right bar in each cluster).

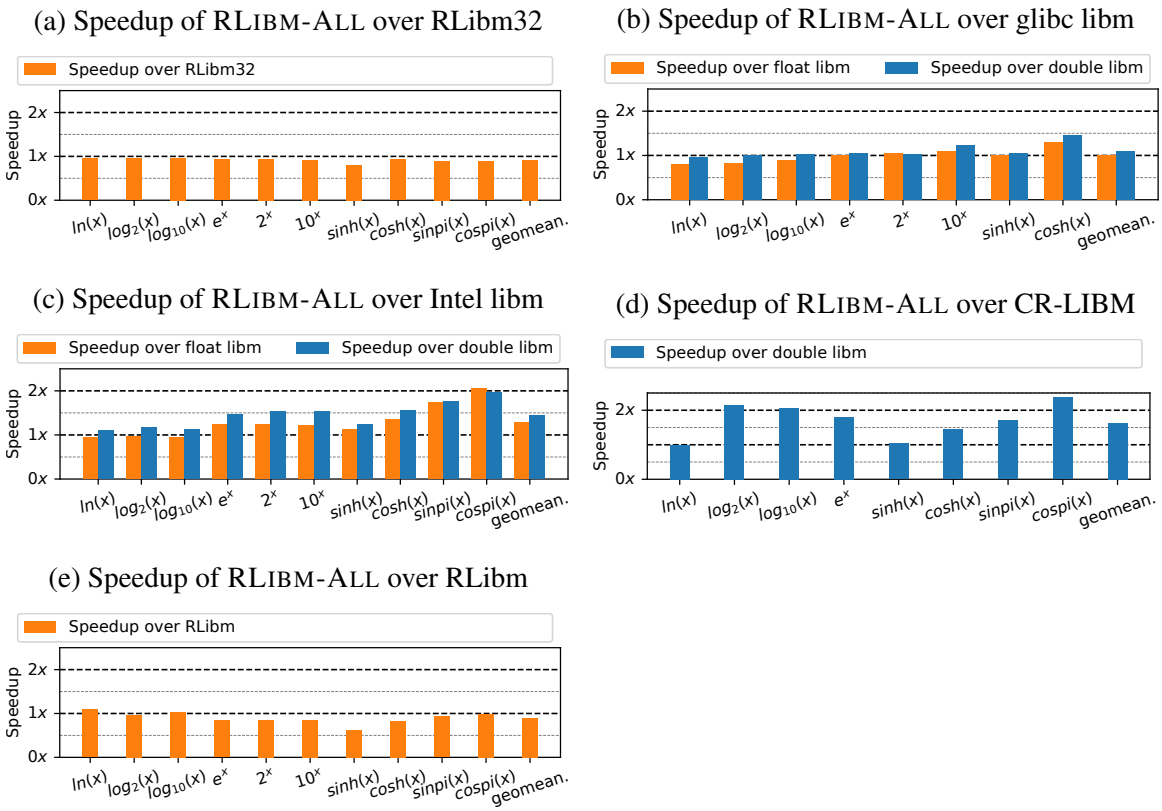


Figure 7.8: (a) Speedup of RLIBM-ALL functions compared to RLibm32 functions when producing bfloat16 results. (b) Speedup of RLIBM-ALL functions compared to Intel's float functions (left) and Intel's double functions (right) when producing bfloat16 results. (c) Speedup of RLIBM-ALL functions compared to CR-LIBM functions when producing bfloat16 results. (d) Speedup of RLIBM-ALL functions compared to glibc's float functions (left) and glibc's double functions (right) when producing bfloat16 results. (e) Speedup of RLIBM-ALL functions compared to RLibm functions when producing bfloat16 results.

RLIBM-ALL has $1.2\times$ and $1.4\times$ speedup over Intel's float and double library, respectively. Figure 7.8(d) reports the speedup of RLIBM-ALL's FP functions over CR-LIBM. RLIBM-ALL has $1.62\times$ speedup over CR-LIBM. CR-LIBM performs better in producing bfloat16 results compared to its performance in producing float results. We conjecture that this is because CR-LIBM has a two-phase approach in approximating elementary functions. For inputs where the correctly rounded result of $f(x)$ is easy to approximate, CR-LIBM produces the results efficiently. For inputs where the exact value of $f(x)$ in reals is close to the rounding boundary of two double values, CR-LIBM uses more accurate but slower approximation methods. CR-LIBM likely computes the correctly rounded result of $f(x)$ efficient

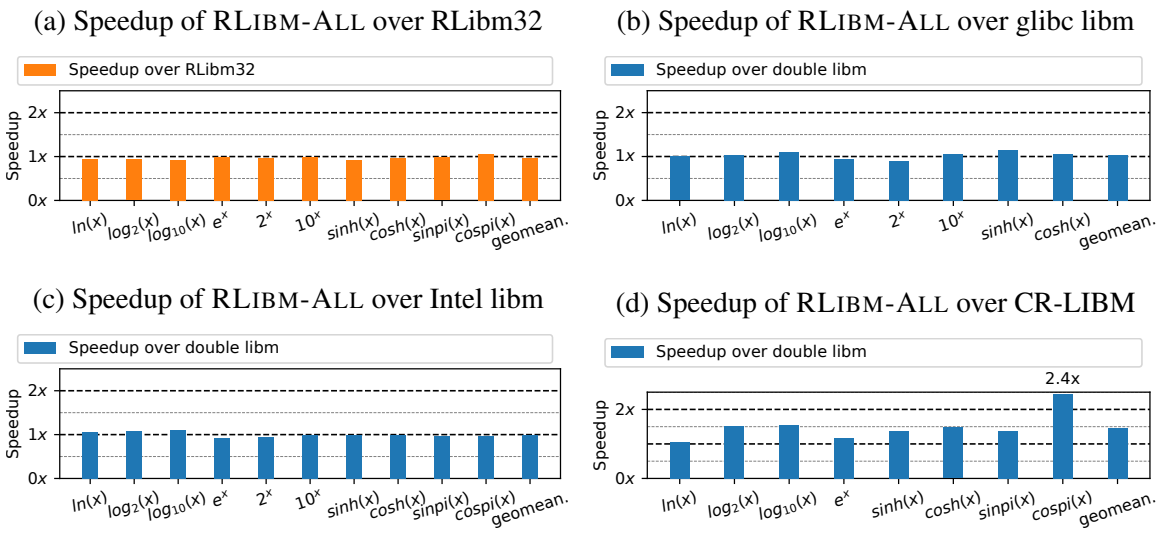


Figure 7.9: (a) Speedup of RLIBM-ALL functions compared to RLibm32 functions when producing posit32 results. (b) Speedup of RLIBM-ALL functions compared to Intel's double functions (right) when producing posit32 results. (c) Speedup of RLIBM-ALL functions compared to CR-LIBM functions when producing posit32 results. (d) Speedup of RLIBM-ALL functions compared to glibc's double functions (right) when producing posit32 results.

with bfloat16 inputs compared to float inputs.

Figure 7.8(e) reports the speedup of RLIBM-ALL's FP functions over RLIBM-16. On average, RLIBM-ALL's FP functions have 9% overhead over RLIBM-16. Since RLIBM-16 is developed and optimized specifically for bfloat16, the majority of RLIBM-16's functions have speedup over RLIBM-ALL except for $ln(x)$, $log_2(x)$, $log_{10}(x)$, and $cospi(x)$. RLIBM-ALL uses a more efficient range reduction strategy compared to RLIBM-16 and results in better performance for these functions.

Performance evaluation of* RLIBM-ALL *when producing posit32 values. Figure 7.9 presents the speedup of RLIBM-ALL's posit functions over various libraries when producing posit32 values. Figure 7.9(a) presents the speedup of RLIBM-ALL's posit functions over RLIBM-32's posit functions. On average, RLIBM-ALL is 3% slower compared to RLIBM-32. This additional overhead is caused by RLIBM-ALL's posit functions producing correctly rounded results for 34-bit posit representation. Figure 7.9(b), (c), and (d) reports the speedup of RLIBM-ALL's posit functions over the repurposed math library using

glibc's double library, Intel's double library, and CR-LIBM. RLIBM-ALL has $1.0\times$, $1.0\times$, and $1.4\times$ speedup over these libraries, respectively. The performance of RLIBM-ALL's posit functions is comparable or faster than the state-of-the-art libraries while guaranteeing to produce correct results for multiple posit configurations. In comparison, Glibc's double library, Intel's double library, and CR-LIBM do not produce correctly rounded posit32 results for all inputs.

CHAPTER 8
RELATED WORK

For small bit-length datatypes (*e.g.*, 8 bits), it is more beneficial to mathematically compute the results of $f(x)$ for each input (*i.e.*, a total of 256 values) and store the results in a table. However, as the size of data types increases (*i.e.*, 32-bits or 64-bits), storing the results of $f(x)$ for each input required exponentially larger storage space. For instance, storing the results of $f(x)$ for the 32-bit float type requires $2^{32} \times 4B = 16GB$ of storage. Hence, multiple decades of research in approximation and range reduction techniques have been developed to accurately approximating elementary functions possible. Further, there are numerous efforts in verifying the error bound of various implementations of elementary functions and repairing math libraries to improve them. We present several seminal work that has influenced and inspired the RLIBM approach.

8.1 Approximation Methods

Polynomial approximation. Polynomials are efficient to evaluate especially when using Horner's method [10] which requires at most $n+1$ additions and n multiplications to evaluate a polynomial of degree n. Further, based on the Weierstrass approximation theorem, any smooth function $f(x)$ can be approximated as closely as needed using a polynomial approximation [144]. Hence, it is one of the most popular approximation techniques. Taylor series [2], also known as the Taylor polynomial expansion, is an infinite degree polynomial used to approximate any smooth function $f(x)$ for inputs near a single pivot point a:

$$P(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \dots$$

Truncating it up to a desirable degree n creates a finite-degree (*i.e.*, n^{th} degree) polynomial approximation. There are two distinct advantages of using the Taylor series when approxi-

mating elementary functions. First, generating the polynomial (especially the coefficients) is straightforward as long as the derivatives can be computed. Second, the error of the Taylor polynomial approximation is bounded by the maximum value of the $(n+1)^{st}$ term among all inputs in the input domain. Hence, Taylor polynomial can be used to produce an approximation of $f(x)$ with arbitrary precision. However, the main drawback of the Taylor polynomial is that the error of the polynomial increases substantially as the input x deviates away from the pivot point a. Comparatively, the Remez algorithm (explained in more detail in Chapter 2) generates polynomials that produce more accurate results for a wider domain of inputs, if the desired error bound is known when generating the polynomial.

Newton's method for square root function. Unlike elementary functions, the $\sqrt{x}$ function is most commonly approximated with the Newton's method or its variants. Abstractly, the square root function $y = \sqrt{x}$ for a given input x (*i.e.*, x is a constant) is converted to $y^2 - x = 0$. Then, the root of this quadratic formula (with the independent variable y) is solved using the iterative Newton's method

$$y_0 = \text{initial guess of } \sqrt{x}$$
$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)} = \frac{y_n}{2} + \frac{x}{2y_n}$$

where $f(y_n) = y^2 - x$. The result y_n is an approximation of $\sqrt{x}$ after n iteration. The Newton's method converges quadratically on average. It is common to store a rough estimate of $\sqrt{x}$ in a small lookup-table and use this value as the y_0 to expedite the algorithm [3]. An advantage of the Newton's ralgorithm is that the numerical error occurring in each iteration does not affect the numerical error of the final result. Rather, the numerical error of an iteration will only slightly affect the convergence rate of the next iteration. With enough iterations, the Newton's method will produce an accurate result.

The hardware implementation of floating point division operation is slower than other primitive operations. A technique to avoid the division operation in the Newton's method

is to approximate the reciprocal square root (*i.e.*, $\frac{1}{\sqrt{x}}$) using the Newton's method (*i.e.*, $f(y) = y^2 - \frac{1}{\sqrt{x}}$). Once the reciprocal of the square root (*i.e.*, $y = \frac{1}{\sqrt{x}}$) is approximated, $\sqrt{x}$ can be found by computing $\sqrt{x} = x \times y$. This strategy completely eliminates the division operations and still converges quadratically. A similar technique known as Halley's method [110] provides a technique to iteratively approximates $\frac{1}{\sqrt{x}}$ with a cubic convergence rate and the Goldschmidt's algorithm [53] approximates both $\sqrt{x}$ and $\frac{1}{\sqrt{x}}$ simultaneously.

CORDIC. The *shift-and-add* algorithms are iterative processes that can approximate various functions using only addition and binary shift operations (*i.e.*, multiplication by powers of 2's). The very first concept of *shift-and-add* algorithm was used to compute the logarithm of various values [115]. Since then, it has been adopted by the CORDIC method [139], an algorithm for approximating trigonometric functions.

In essence, the CORDIC method computes trigonometric functions through a series of 2-dimensional vector rotations. The angle θ is decomposed into a series of small decreasing angles $\theta = \theta_0 \pm \theta_1 \pm \theta_2 \pm \ldots$. An initial vector $[x_0, y_0]^T = [1, 0]^T$ is rotated using these angles (*i.e.*, θ_i) iteratively. By choosing a particular sequence of angles $\pm\theta_i$ such that $tan(\theta_i) = 2^{-i}$, the vector rotation by the angle θ_i in each iteration can be computed with,

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = K \begin{bmatrix} 1 & -2^{-i} \\ 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

where K is a predetermined constant. After a sequence of rotations by the angle θ_i, the resulting vector $[x_i, y_i]^T$ is an approximation of $[cos(\theta), sin(\theta)]^T$. This vector rotation only requires additions and shift operations after some optimization steps.

Mathematically, each iteration of the CORDIC method provides one digit of accuracy in approximating $cos(\theta)$ and $sin(\theta)$. It is well received in the hardware community for its straightforward implementations and ability to produce results efficiently when only a few digits of accuracy is required. A generalized CORDIC algorithm was developed to approximate hyperbolic functions [140] or perform multi-dimensional CORDIC algorithm [122]. Several work have been proposed to improve the performance by parallelizing [49, 138],

pipelining [43], and modularizing [143] the CORDIC method. The angle recoding [19, 64], fast-forwarding [97], and the redundant algorithm [132] reduces the number of iterations to both increase the performance and the accuracy. A detailed overview of prior research on CORDIC is available in the survey [98].

Bipartite table method. Instead of storing the result of $f(x)$ in a lookup table for every input, the bipartite method uses two smaller lookup tables to approximate $f(x)$. First proposed to approximate trigonometric functions [129] and reciprocals (*i.e.*, $\frac{1}{x}$) [31], it has been generalized for all elementary functions using polynomial approximations [120, 127]. Abstractly, the bipartite method divides the input domain into k larger subdomains and then further split each subdomain into another k smaller subdomains (a total of k^2 subdomains). The function $f(x)$ in each smaller subdomain is approximated using a 1^{st} degree polynomial, $P(x) = c_{i,j} + c_i x$, where c_i is unique for each larger subdomain (*i.e.*, a total of k values) and $c_{i,j}$ is specific to each smaller subdomain (*i.e.*, a total of k^2 values). Using this technique, approximating $f(x)$ only requires two lookup tables with k and k^2 entries, respectively.

The coefficients c_i, j and c_j can be identified using Taylor's expansion. The input x is decomposed into three k-bit smaller numbers, $x = x_0 + x_1 + x_2$. Then, the Taylor's expansion of $f(x)$ with the pivot point $x_0 + x_1$ is used to create a monomial approximation,

$$P(x) = f(x_0 + x_1) + (x - x_0 - x_1) \times f'(x_0 + x_1) = f(x_0 + x_1) + x_2 \times f'(x_0 + x_1)$$

By approximating the term $f'(x_0 + x_1)$ with $f'(x_0)$, we get the approximation function,

$$P(x) = c_{i,j} + c_i x_2, \qquad c_{i,j} = f(x_0 + x_1), \qquad c_i = f'(x_0)$$

The bipartite method can be implemented efficiently using two small lookup tables and a few primitive operations. Hence, it is a popular technique along with the CORDIC method to create custom hardware implementation. An extension, known as the multipartite table method [41, 121] uses several smaller lookup tables created with higher degree Taylor expansion to increase the accuracy.

8.2 Correctly Rounded Approximation

Naively using the approximation algorithms discussed in Section 8.1 with finite precision representations does not guarantee to produce the correctly rounded result of $f(x)$. As the table maker's dilemma [75] states, it may require an arbitrary amount of precision to compute the correctly rounded results for an arbitrary input x and elementary function $f(x)$. Hence, several techniques were developed to avoid the table maker's dilemma, produce the correctly rounded results of $f(x)$ efficiently, and create correctly rounded math libraries.

Math libraries using the onion peeling strategy. The first successful attempts in creating correctly rounded math libraries for the float and double types (*i.e.*, LibUltim [65, 146] and LibMCR [101]) use the onion peeling approach (also know as Ziv's strategy). It avoids the table maker's dilemma by iteratively approximating $f(x)$ with progressively higher precision until it can determine the correctly rounded result of $f(x)$. Initially, the onion peeling strategy chooses a target precision p_1. Then, it approximates $f(x)$ with p_1 bits of precision to produce a result result y_1 and its corresponding error bound ϵ_1. The onion peeling strategy typically uses approximation methods that can potentially compute $f(x)$ to arbitrary precision and produce the error bound at the same time (*e.g.*, Taylor series or the CORDIC method). Based on the result y_1 and ϵ_1, the real value of $f(x)$ is within $[y_1 - \epsilon_1, y_1 + \epsilon_1]$. If both $y_1 - \epsilon_1$ and $y_1 + \epsilon_1$ rounds to the same value in the target representation, then the rounded result of y_1 is the correctly rounded result. Otherwise, the current approximation cannot conclude what the correct result is. The onion peeling strategy chooses a higher precision p_2 and repeats the process.

MPFR [46], an arbitrary precision math library, implements elementary functions using the same strategy. In the MPFR math library, the user can define the precision of the target FP representation. Hence, the amount of precision required to compute the correctly rounded result of $f(x)$ for the target representation is truly unbounded, which makes the onion peeling strategy especially suitable for MPFR.

The introduction of the onion peeling strategy changed the perception of math library developers by showing that avoiding the table maker's dilemma and generating a correctly rounded math library is possible. However, the main drawback is its performance. Iteratively approximating $f(x)$ with higher precision is expensive. Theoretically, the onion peeling strategy may never terminate and keep increasing the working precision (*i.e.*, p_i). Regardless, we emphasize the impact of the onion peeling strategy, especially on our RLIBM approach. Although the elementary functions that we create do not use the onion peeling strategy directly, the RLIBM approach computes the correctly rounded results of $f(x)$ using the MPFR library. This allows us to avoid the Table-maker's dilemma and generate efficient and correct polynomial approximations.

Resolving table maker's dilemma for specific representations. In 2001, the table maker's dilemma for the double precision was resolved by identifying the highest amount of precision required to approximate $f(x)$ and produce the correctly rounded results for all inputs in the double precision [85]. This worst-case precision requirement was computed using a filter-and-check strategy. The filtering step initially splits the input domain into extremely small sub-domains where $f(x)$ can be accurately approximated even with a 1^{st} degree polynomial approximation. Then, it approximates $f(x)$ for all inputs in the subdomain and checks whether the result is within a certain threshold of the rounding boundary. Any subdomain where no $f(x)$ is close to the rounding boundary is filtered. Next, the checking step computed $f(x)$ for all inputs in the remaining subdomains and identified the necessary amount of precision to compute the correctly rounded result of $f(x)$ in double. It was found that roughly 160 precision bits were required, in the worst case, to produce correctly rounded results for the double type. There is active research in identifying the precision required to produce correctly rounded results for other representations [14, 125].

Math libraries using the Remez algorithm. Using the worst case precision requirement, CR-LIBM [29, 86] implements several elementary functions for the double type with min-

imax polynomial approximations. The polynomials are generated using Sollya [21], which provides a modified Remez algorithm. The original Remez algorithm produces minimax polynomials $P(x)$ with real number coefficients c_i and guarantees error bound ϵ when evaluated in real number. However, rounding these coefficients and evaluating the polynomial in a finite precision representation may produce an error significantly larger than ϵ. Hence, Sollya generates minimax polynomials with coefficients in a finite precision representation $\mathbb{H}$ [15]. Abstractly, Sollya searches through all polynomials $\hat{P}(x)$ with rational coefficients $\hat{c_i}$ around c_i. All possible ranges of $\hat{c_i}$ for $0 \leq i \leq n$ where n is the degree of $\hat{P}(x)$ form a rational polytope in $\mathbb{Z}^{n+1}$. Then it uses a rational value search algorithm within the polytype to identify candidate polynomials and calculates the infinity norm of the error of the polynomial until it finds a $\hat{P}(x)$ with a sufficiently small maximum error. Sollya also computes the maximum numerical error of polynomial evaluation using interval arithmetic [20, 22] and produces Gappa [99] proofs to ensure that the polynomial produces the correctly rounded results. The polynomial searching algorithm has later been optimized using the lattice basis reduction technique [11].

MetaLibm [16, 80] is an elementary function generator built using Sollya. The main goal of MetaLibm is to generate efficient minimax polynomials with user-defined error bounds. It automatically creates range reduction strategies, performs hardware specific optimizations, and generates piecewise polynomials to approximate $f(x)$ [79]. MetaLibm uses another polynomial approximation to select the appropriate polynomial in the piecewise polynomials for a given input [81]. This allows the elementary functions to be parallelized with vector instructions. Other than Sollya, another modified Remez algorithm has been proposed to generate a minimax polynomial that also accounts for numerical errors when evaluated in a finite precision representation [4]. It has been shown to produce correctly rounded results for small input domains where the range reduction is not necessary.

SoftPosit-Math using minefield method. While identifying the worst case precision requirement resolves the table maker's dilemma, approximating the real value of $f(x)$ does not provide the maximum amount of freedom to generate efficient and correctly rounded polynomial approximation. Instead, the Minefield method [57] identifies the largest interval of values that polynomial approximations should produce to compute the correctly rounded results. It declares all other regions as minefields. Then, it generates a polynomial that avoids all mines. SoftPosit-Math [88] uses the Minefield method to create several correctly rounded posit16 elementary functions. Our RLIBM approach is inspired by the Minefield method. The rounding intervals in the RLIBM approach can be considered as the safe paths in the Minefield method. The RLIBM approach generalizes the Minefield method for numerous representations while considering range reduction and output compensation. We also formalize an automatic process to encode the safe paths as linear constraints and use an LP solver to generate the polynomial. More importantly, all prior approaches including the minefield method generate elementary functions that target a particular representation and rounding mode. The RLIBM approach can generate a single polynomial approximation that handles multiple representations and rounding modes at the same time.

Custom hardware function evaluation. One of the most important aspects of custom hardware design for approximating elementary functions is the hardware cost to implement the function. One way to reduce the hardware cost is to use fixed point representations rather than floating point representations. Fixed point representations directly encode binary fraction values in an n-bit representation. Essentially, it is equivalent to an integer type with an implicit radix point. Hence, all fixed point arithmetic can be performed using integer arithmetic. Another strategy to reduce the hardware cost, especially when using piecewise polynomial approximations, is to reduce the bit-length of the coefficients. Smaller bit-length coefficients will require smaller lookup tables and arithmetic units. Hence, a technique [36] was proposed to generate piecewise polynomials with the

smallest bit-length fixed point coefficients that still produce the correctly rounded results for all inputs. This technique generates an integer linear programming (ILP) formulation that specifies constraints on the outputs of the polynomial they wish to generate, similar to the RLIBM approach. Additionally, they encode the objective function in the ILP formulation to minimize the bit-length of the coefficients and use ILP solver to generate the polynomial. This technique is effective in generating efficient piecewise polynomial for fixed point representations where the dynamic range is limited. In comparison, floating point representations have significantly larger dynamic range and the RLIBM approach presents a technique to produce correctly rounded results when evaluated with range reduction strategy. We hope to combine it with the RLIBM approach to incorporate range reduction techniques and our odd interval techniques to optimize our elementary functions.

8.3 Range Reduction Strategies

To generate efficient implementations of elementary functions, math library designers use polynomial approximations and range reduction strategies together. The simplest form of range reduction strategies directly use mathematical identities of elementary functions. For example, the natural log function $ln(x)$ has the mathematical identify $ln(a \times b^c) = ln(a) + cln(b)$. If the original input x is decomposed to $x = x' \times 2^m$ where $x' \in [1, 2)$ and $m \in \mathbb{Z}$ is the exponent of x, then $ln(x)$ can be computed using $ln(x') + mln(2)$. This strategy allows us to only approximate $ln(x')$ for a small input domain $[1, 2)$. There are also range reduction strategies designed for individual elementary functions [25] that transforms the function we must approximate (*i.e.*, $ln(x)$) into a different function. In particular for $ln(x)$, the reduced input x' can be transformed into a different value $t = \frac{x'-1}{x'+1}$ such that $ln(x')$ can be approximated with $ln(\frac{1+t}{1-t})$. This transformed function can be approximated with an odd polynomial, effectively reducing the cost of evaluating the polynomial to half. More detail on this specific range reduction strategy is provided in Chapter 4. Unfortunately, these strategies must be developed separately for each elementary function and does not

guarantee that a similar approach can be found for other elementary functions.

Tabled-Based Range Reduction Strategy. Instead, all mainstream math libraries and the RLIBM approach adopt a more general strategy known as the table-based range reduction (also known as the Tang's method) [133, 134, 135]. This strategy significantly reduces the input domain which results in a low degree polynomial and uses a look-up tables to accurately and efficiently evaluate the output compensation function. The insight is to decompose the original input x into two components: a continuous variable and a discrete variable. For instance, consider the function $ln(x)$ for the input space $x \in [1, 2)$. The input x can be decomposed into $x = u + v$ where $u \in \{\frac{64}{64}, \frac{65}{64}, \ldots, \frac{127}{64}\}$ is a discrete variable and v is a continuous variable in $[0, \frac{1}{64})$. This decomposition effectively splits the original input domain $[1, 2)$ into 64 equal sub-divisions (*i.e.* $[1, \frac{65}{64}), [\frac{65}{64}, \frac{66}{64}), \ldots$) where u represents the index of each sub-division. Then, $ln(x)$ can be computed with,

$$ln(x) = ln(u + v) = ln(u(1 + v)) = ln(u) + ln(1 + \frac{v}{u})$$

Because u is a discrete variable (*i.e.* exactly 64 possible values), we can pre-compute $ln(u)$ with high precision and store the results into a look-up table. If we denote $\frac{v}{u}$ as x', approximating $ln(x)$ only requires a polynomial approximation of $ln(1 + x')$ for the reduced input domain $x' \in [0, \frac{1}{64})$. The output compensation function requires a single table lookup and an addition of the value $ln(u)$.

There are three distinct advantages of the table-based range reduction that make it attractive. First, the significant reduction in the input domain allows us to generate a lower degree polynomial, beneficial for performance. The output compensation function can also be evaluated efficiently using the lookup table. Second, the table-based range reduction is flexible. It is straightforward to configure the amount of domain reduction by adjusting the input decomposition. The original range reduction for $ln(x)$ [134] recommended to split the original input domain into 64 sub-divisions (*i.e.*, reduced domain of $[0, \frac{1}{64})$). Our prototypes split the domain into 128 sub-divisions to reduce the input domain into $[0, \frac{1}{128})$. Using

table-based method, the size of the reduced input domain (*i.e.*, $[0, \frac{1}{2^i}]$) can be configured by the number of the sub-domains (*i.e.*, 2^i sub-domains). The table-based method can also be nested multiple times to further reduce the input domain [29, 73]. Finally, table-based range reduction can be applied to various elementary functions. The only requirement to apply this strategy is for the elementary function to have an identify $f(a \circ b) = f(a) \diamond f(b)$ where $\circ$ and $\diamond$ are primitive operations (*i.e.*, $ln(a \times b) = ln(a) + ln(b)$). There are range reduction strategies for logarithm functions [134], exponential functions [133], trigonometric functions [135], and hyperbolic functions [29]. All elementary functions in RLIBM-32 and RLIBM-ALL use table-based range reduction for efficient implementations (additional details can be found in Chapter 4).

Accurate Additive Range Reduction Strategies. Additive range reductions can experience significant amount of cancellation error when evaluated in a finite precision representation $\mathbb{H}$. As described in Chapter 2, additive range reduction reduces an input x into the reduced input $x' \in (-C, C)$ by computing $x' = x - kC$ for a constant value C and an integer k (*i.e.*, $x' \equiv x \; mod \; C$). When the magnitude of x is similar to C (*i.e.*, k is a small value), $x - kC$ does not experience significant error. However, when x is considerably larger than C and the value of x and kC are similar, then $x - kC$ can experience catastrophic cancellation. Trigonometric functions use additive range reductions (*i.e.*, $C = \frac{\pi}{2}$) to exploit trigonometric identities (*i.e.*, $sin(a + 2\pi) = sin(a)$) and experience such errors when the input x is large. Hence, the very early implementations of trigonometric functions only supported small inputs (*i.e.*, $x \in [-\frac{\pi}{2}, \frac{\pi}{2}]$) delegating the job of range reduction to the user [6]. Some implementations use their own definition of π [105], usually a value obtained by rounding the real value of π to a finite-representation, or use an arbitrary precision representation to perform range reduction for trigonometric functions [109].

Cody and Waite reduction method [24, 25] provides a strategy to accurately perform additive range reduction for relatively small inputs (*i.e.*, $x < 100$). It stores the value of

C with two values C_1 and C_2 in $\mathbb{H}$. These values are chosen in a particular way such that C_1 is close to C and the sum of C_1 and C_2 approximates C (*i.e.*, $C \approx C_1 + C_2$). Then the range reduction is evaluated with $x' = (x - kC_1) - kC_2$. If the value of kC_1 is exactly representable in $\mathbb{H}$, then $z = x - kC_1$ can be exactly computed by Sterbenz Lemma [126] and $z - kC_2$ will not experience significant amount of numerical error.

The Payne and Hanek reduction [113] is an effective strategy for extremely large inputs (*i.e.*, $x > 2^{64}$). The insight is that when the value of an input x is close to kC, the additive range reduction (*i.e.*, $x - kC$) will cancel the first p significant bits of x and kC (which is the cause of cancellation error). If it is possible to identify the number of bits that will be canceled (*i.e.*, p), then the subtraction can be optimized by not considering the first p bits of both x and kC. This also means the first p bits of kC do not have to be computed. The Payne and Hanek reduction provides a systematic algorithm to approximate p and perform $x - kC$ accurately. Due to the overhead of identifying p, Payne and Hanek reduction method is mostly used for large inputs where it is common to experience catastrophic cancellation unless the additive range reduction is performed with an extremely high precision representation.

For medium sized inputs (*i.e.*, $2^3 < x < 2^{64}$), the modular range reduction [13, 33, 87] can accurately compute additive range reduction. This strategy uses mathematical identities of the modulus operation,

$$(x + y)\ mod\ C = ((x\ mod\ C) + (y\ mod\ C))\ \ mod\ C$$

The modular range reduction breaks down an input with n-bits of precision $x = m \times 2^k$, where m is the significand and k is the exponent of x, into $x = \sum_{i=0}^{n-1} x_i 2^{k-i}$. Here, $x_i \in \{0, 1\}$ is the i^{th} most significant precision bit in m. Abstractly, $x_i 2^{k-i}$ is the value represented by the x_i bit. Next, it computes the value,

$$r = \sum_{i=0}^{n-1} x_i\ (2^{k-1}\ mod\ C)$$

By the properties of modulus operation, $r\ mod\ C$ is equivalent to $x\ mod\ C$. Since r

is a sum of at most n values smaller than C, it is guaranteed that $r < nC$ where r is a value much smaller than the original input x. Then, the value r is reduced to $x' = r - kC$ (where $0 \leq k < n$ is an integer) to produce the reduced input $x' \in (-C, C)$. All possible values of $x_i 2^{k-i} \; mod \; C$ and kC are stored in a lookup table. Because the magnitude of x as well as the possible values of k are limited (*i.e.*, $x < 2^{64}$ and $k < n$), the size of the lookup table is small. For example, computing $x \; mod \; \frac{\pi}{2}$ (*i.e.*, $C = \frac{\pi}{2}$) for an input between $x \in [0, 2^{64}]$ would require roughly 64 values for $x_i 2^{k-i} \; mod \; C$ and 53 possible values for kC. The modular range reduction has been optimized where x is decomposed into groups of bits (rather than individual bits) to reduce the number of accumulation operations [13]. The on-the-fly range reduction strategy [87] integrates the $x' = r - kC$ computation into the accumulation of each term $(x_i 2^{k-1} \; mod \; C)$ to efficiently compute the final result. The trigonometric functions in CR-LIBM use all three additive range reduction strategies to produce correctly rounded double results.

8.4 Verification of Math Libraries

As performance and correctness are both important with math libraries, there is extensive research to formally bound the error of elementary functions. HOL Light [63] is a theorem prover primarily used for verifying FP operations. It formalizes the semantics of FP arithmetic [62] and proves the bound of the error of several implementations of elementary functions [59, 60]. It has also been used to verify the correctness of the floating point arithmetic operations in Intel Itanium chipset [61]. Similarly, CoQ proof assistant has been used to prove the correctness of implementations of Cody and Waite's reduction method with `fused-multiply-add` operations [8]. These verification approaches require a significant amount of manual work. Hence, researchers have developed an automatic verification technique and verified that many elementary functions in Intel's `math.h` have at most 1 ulp error [82, 84].

Sollya verifies that the elementary functions that it generates produce correctly rounded

results with the help of Gappa [35, 38, 39]. Sollya translates the implementations of the elementary functions to Gappa lemmas. Then, Gappa uses integer arithmetic to bound the error of each operation with some hints from the math library developers. Finally, Sollya ensures that the final result of the elementary function will round to the correctly rounded result based on the error bound. It has been used to verify the elementary functions in CR-LIBM.

Instead of using formal proofs, RLIBM validates that the generated polynomial produces the correctly rounded results by evaluating the polynomial for each input. This manual process is possible because our target representations (*i.e.*, 32-bit float) have a reasonable number of inputs (*i.e.*, 2^{32} inputs). To extend RLIBM for the double type where iterating through all inputs is infeasible, we will likely have to rely on prior verification efforts to formally prove the correctness of the generated polynomials.

8.5 Math Library Repair Tools

An alternative for creating correctly rounded elementary functions is to repair existing math libraries that do not produce correctly rounded results. If the numerical error in evaluating the implementation is the cause of an incorrect result, we can use tools that detect numerical errors to diagnose the root cause of the error [7, 23, 47, 54, 118, 145, 147]. Subsequently, we can rewrite parts of the implementation that causes these errors using rewriting tools such as Herbie [111] or Salsa [28]. If the cause of the incorrect result stems from the approximation error, then we can use a math library repair tool [145]. This tool identifies small domains of inputs that result in high error by performing binary searches in the entire input domain. Then, it uses piecewise polynomials containing linear or quadratic equations to repair the elementary function for these domains.

Unfortunately, these tools are primarily designed to reduce errors rather than completely removing them. Hence, math libraries repaired with these tools are not likely to produce correctly rounded results for all inputs. Additionally, the repaired code may intro-

duce overhead, negatively impacting the performance. Hence, it is ideal to use tools like RLIBM or Sollya and create correctly rounded elementary functions.

CHAPTER 9
CONCLUSION AND FUTURE DIRECTIONS

This dissertation presents the RLIBM approach for generating polynomial approximations that produce correctly rounded results of elementary functions $f(x)$ for all inputs. The RLIBM approach makes a case for approximating the correctly rounded result of $f(x)$ rather than the real value of $f(x)$ itself. We first summarize the contributions in this dissertation and then present directions for future work.

9.1 Dissertation Summary

As elementary functions are essential components in scientific applications, there have been seminal research over decades to approximate elementary functions accurately and efficiently. However, efficiently computing the correctly rounded results of elementary functions remains a challenging problem. In this dissertation, we propose the RLIBM approach to generate polynomial approximations that produce correctly rounded results for all inputs. While prior approaches try to approximate the real value of the elementary function, RLIBM advocates for approximating the correctly rounded result. The RLIBM approach identifies the maximum amount of freedom available to generate the correctly rounded results and uses this freedom to generate efficient polynomial approximations using linear programming.

To generate polynomial approximations that produce the correctly rounded results for 32-bit representations, RLIBM uses the counterexample guided polynomial generation technique inspired by program synthesis. This technique samples a small fraction of inputs to generate a polynomial that produces the correctly rounded results for all 32-bit inputs. Additionally, RLIBM generates piecewise polynomials with low degrees by dividing the input domain into sub-domains using the bit-pattern of the inputs. These two techniques allow

RLIBM to generate efficient implementations of elementary functions.

Finally, RLIBM presents a novel approach to generate a single polynomial approximation that produces the correctly rounded results for multiple representations and rounding modes. When the goal is to generate correctly rounded results for $\mathbb{T}_k$ representations where $k \leq n$, the RLIBM approach generates a polynomial approximation that produces the correctly rounded result for $\mathbb{T}_{n+2}$ with the rno mode. RLIBM addresses the issue of inputs with singleton intervals using mathematical identities of elementary functions to efficiently identify these inputs and compute the exact result of $f(x)$. Based on our formal proofs, the resulting polynomial is guaranteed to produce correctly rounded results for all inputs in all $\mathbb{T}_k$ representations with any standard rounding modes.

The resulting elementary functions generated with the RLIBM approach not only produce correctly rounded results for various representations and rounding modes, but are faster than mainstream math libraries. The target representations of our elementary functions include, but are not limited to, bfloat16, tensrofloat32, float, posit16, and posit32. In particular, our RLIBM-ALL prototype provides the first implementations of elementary functions that produce the correctly rounded results for hundreds of representations with all standard rounding modes. We hope that this dissertation motivates existing and new representations to mandate correctly rounded elementary functions.

9.2 Future Directions

The RLIBM approach is our initial effort in creating the correctly rounded math libraries. There are multiple directions to improve it. We present three potential venues that can be explored.

Generating correctly rounded approximations for all elementary functions. The IEEE-754 standard defines 36 commonly used elementary functions and recommends math libraries to provide correctly rounded implementations for them. Currently, our prototypes

provide implementations for ten of these functions. We intend to generate correctly rounded approximations for all 36 elementary functions. We believe the RLIBM approach can generate polynomial approximations for univariate elementary functions (*e.g.*, $sin(x)$). However, it may require us to develop novel range reductions to create efficient implementations. For example, it may be necessary to revisit range reductions for trigonometric functions such as $sin(x)$ and $cos(x)$ to ensure that we compute the reduced inputs accurately. While performing range reduction in higher precision will accomplish this task, the performance of high precision arithmetic operations are not ideal.

Some elementary functions recommended by the IEEE-754 are multivariate functions (*i.e.*, x^y and $atan(\frac{y}{x})$). The RLIBM approach does not handle automatically generating polynomial approximations for these functions. As there are multiple inputs in these elementary functions, the polynomial approximation must be a multivariate polynomial as well. The challenge in using the RLIBM to generate multivariate polynomials is in framing the input-output constraints as linear constraints.

Generating correctly rounded polynomial approximations for 64-bit representations. Another possible avenue to improve the RLIBM approach is in its support for generating correctly rounded polynomial approximations for the double type. The double type is the standard 64-bit FP representation. It has more than twice the precision of the 32-bit float type (*i.e.*, 53 bits for double compared to 24 bits for float). Many scientific applications requiring high precision perform internal computations with double. Hence, an efficient and correctly rounded math library for double is essential. The RLIBM approach can generate polynomials that produce the correctly rounded results for a small sample of inputs in double. However, the challenge lies in verifying that this polynomial produces the correctly rounded results for all inputs in double. Iterating through all 2^{64} inputs each time we generate a polynomial to verify its correctness is not feasible. We will most likely have to extend the RLIBM approach with formal verification techniques to check for the correctness of the

polynomial approximation.

Performance optimizations with integer arithmetics. Both the performance and the correctness of math libraries are important. Hence, we plan to focus on improving the performance of our elementary functions while making sure that they produce correctly rounded results. One possible approach to optimize our implementations is to use integer operations rather than floating point operations. All of our elementary functions perform internal computations with double. While common architectures provide hardware support for arithmetic operations with double, it is still slower than integer operations. Hence, implementing elementary functions with integer operations instead of floating point operations will increase the performance. The challenge in creating elementary functions with integer operations is to ensure that the resulting values are correctly rounded results. We plan to extend the RLIBM approach to generate polynomials with integer coefficients that produce correctly rounded results for all inputs when evaluated with integers. Another direction that we will explore is to develop new range reduction techniques that do not require floating point operations.

BIBLIOGRAPHY

[1] Martin Aigner and Gnter M. Ziegler. 2009. *Proofs from THE BOOK* (4th ed.). Springer Publishing Company, Incorporated.

[2] Tom M Apostol. 1974. *Mathematical analysis; 2nd ed.* Addison-Wesley, Reading, MA.

[3] Inc. Apple Computer. 2002. *sqrt.c*. https://opensource.apple.com/source/Libm/Libm-47.1/ppc.subproj/sqrt.c.auto.html

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

[5] Alan Baker. 1975. *Transcendental Number Theory*. Cambridge University Press.

[6] Nelson H. F. Beebe. 2017. *The Mathematical-Function Computation Handbook: Programming Using the MathCW Portable Software Library* (1st ed.). Springer Publishing Company, Incorporated.

[7] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A Dynamic Program Analysis to Find Floating-point Accuracy Problems. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. ACM, New York, NY, USA, 453–462. https://doi.org/10.1145/2345156.2254118

[8] 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

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

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

[11] 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

[12] N. Brisebarre, D. Defour, P. Kornerup, J.-M. Muller, and N. Revol. 2005. A new range-reduction algorithm. *IEEE Trans. Comput.* 54, 3 (2005), 331–339. https://doi.org/10.1109/TC.2005.36

[13] Nicolas Brisebarre, David Defour, Peter Kornerup, Jean-Michel Muller, and Nathalie Revol. 2005. A new range-reduction algorithm. *IEEE Trans. Comput.* 54, 3 (2005), 331–339. https://doi.org/10.1109/TC.2005.36

[14] Nicolas Brisebarre and Guillaume Hanrot. 2021. Integer points close to a transcendental curve and correctly-rounded evaluation of a function. (May 2021). https://hal.archives-ouvertes.fr/hal-03240179 working paper or preprint.

[15] 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

[16] 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

[17] 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

[18] Zachariah Carmichael, Hamed F. Langroudi, Char Khazanov, Jeffrey Lillie, John L. Gustafson, and Dhireesha Kudithipudi. 2019. Deep Positron: A Deep Neural Network Using the Posit Number System. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1421–1426.

[19] Cheng-Shing Wu, An-Yeu Wu, and Chih-Hsiu Lin. 2003. A high-performance/low-latency vector rotational CORDIC architecture based on extended elementary angle set and trellis-based searching schemes. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* (2003).

[20] 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.

[21] 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

[22] 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

[23] Sangeeta Chowdhary, Jay P. Lim, and Santosh Nagarakatte. 2020. Debugging and Detecting Numerical Errors in Computation with Posits. In *41st ACM SIGPLAN*

Conference on Programming Language Design and Implementation (PLDI'20). https://doi.org/10.1145/3385412.3386004

[24] William J Cody. 1982. Implementation and testing of function software. In *Problems and Methodologies in Mathematical Software Production*, Paul C. Messina and Almerico Murli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 24–47.

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

[26] P. M. (Paul Moritz) Cohn. 1974. *Algebra [by] P. M. Cohn.* Wiley, London.

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

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

[29] 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

[30] 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. LIP,. https://hal-ens-lyon.archives-ouvertes.fr/ensl-01529804

[31] Debjit Das Sarma and David W. Matula. 1995. Faithful bipartite ROM reciprocal tables. In *Proceedings of the 12th Symposium on Computer Arithmetic*. 17–28. https://doi.org/10.1109/ARITH.1995.465381

[32] Marc Daumas, Christophe Mazenc, Xavier Merrheim, and Jean michel Muller. 1995. Modular range reduction: A new algorithm for fast and accurate computation of the elementary functions. *Journal of Universal Computer Science* (1995), 175.

[33] Marc Daumas, Christophe Mazenc, Xavier Merrheim, and Jean-Michel Muller. 1995. Modular Range Reduction: a New Algorithm for Fast and Accurate Computation of the Elementary Functions. (1995), 162–175. https://doi.org/10.1007/978-3-642-80350-5_15

[34] Marc Daumas and Guillaume Melquiond. 2010. Certification of Bounds on Expressions Involving Rounded Operators. *ACM Transactions on Mathematical Softwware* 37, 1, Article 2 (Jan. 2010), 20 pages. https://doi.org/10.1145/1644001.1644003

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

[36] Davide De Caro, Ettore Napoli, Darjn Esposito, Gerardo Castellano, Nicola Petra, and Antonio G. M. Strollo. 2017. Minimizing Coefficients Wordlength for Piecewise-Polynomial Hardware Function Evaluation With Exact or Faithful Rounding. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2017). https://doi.org/10.1109/TCSI.2016.2629850

[37] Florent de Dinechin, Luc Forget, Jean-Michel Muller, and Yohann Uguen. 2019. Posits: the good, the bad and the ugly. In *CoNGA'19 - Conference for Next Generation Arithmetic*. ACM Press, Singapore, Singapore, 1–10.

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

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

[40] Florent de Dinechin and Arnaud Tisserand. 2005. Multipartite table methods. *IEEE Trans. Comput.* 54, 3 (2005), 319–330. https://doi.org/10.1109/TC.2005.54

[41] Florent de Dinechin and Arnaud Tisserand. 2005. Multipartite table methods. *IEEE Trans. Comput.* 54, 3 (2005), 319–330. https://doi.org/10.1109/TC.2005.54

[42] Hugues de Lassus Saint-Geniès, David Defour, and Guillaume Revy. 2015. Range reduction based on Pythagorean triples for trigonometric function evaluation. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 74–81. https://doi.org/10.1109/ASAP.2015.7245712

[43] Ed Deprettere, Patrick Dewilde, and R. Udo. 1984. Pipelined cordic architectures for fast VLSI filtering and array processing. In *ICASSP '84. IEEE International Conference on Acoustics, Speech, and Signal Processing*.

[44] Albert D. Edgar and Samuel C. Lee. 1977. The Focus Number System. *IEEE Trans. Comput.* C-26, 11 (1977), 1167–1170. https://doi.org/10.1109/TC.1977.1674770

[45] Seyed H. Fatemi Langroudi, Tej Pandit, and Dhireesha Kudithipudi. 2018. Deep Learning Inference on Embedded Devices: Fixed-Point vs Posit. In *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. 19–23.

[46] 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

[47] Zhoulai Fu and Zhendong Su. 2019. Effective Floating-point Analysis via Weak-distance Minimization. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 439–452. https://doi.org/10.1145/3314221.3314632

[48] Shmuel Gal. 1986. Computing elementary functions: A new approach for achieving high accuracy and good performance. In *Accurate Scientific Computations*, Willard L. Miranker and Richard A. Toupin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

[49] Bimal Gisuthan and Thambipillai Srikanthan. 2002. Pipelining flat CORDIC based trigonometric function generators. *Microelectronics Journal* (2002).

[50] 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

[51] GNU. 2019. *The GNU C Library (glibc)*. https://www.gnu.org/software/libc/

[52] 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.

[53] Robert Goldschmidt. 2005. Applications of division by convergence. (08 2005).

[54] Eric Goubault. 2001. Static Analyses of the Precision of Floating-Point Operations. In *Proceedings of the 8th International Symposium on Static Analysis (SAS)*. Springer, 234–259. https://doi.org/10.1007/3-540-47764-0_14

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

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

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

[58] John Gustafson and Isaac Yonemoto. 2017. Beating Floating Point at Its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations: an International Journal* 4, 2 (June 2017), 71–86.

[59] 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

[60] 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

[61] John Harrison. 2006. Floating-Point Verification using Theorem Proving. , 211–242 pages.

[62] John Harrison. 2009. HOL Light: An Overview. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009 (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer-Verlag, Munich, Germany, 60–66. https://doi.org/10.1007/978-3-642-03359-9_4

[63] John Harrison. 2017. *The HOL Light theorem prover*. https://www.cl.cam.ac.uk/~jrh13/hol-light/

[64] Yu Hen Hu and Homer H. M. Chern. 1996. A Novel Implementation of CORDIC Algorithm Using Backward Angle Recoding (BAR). *IEEE Trans. Comput.* (1996).

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

[66] Intel. 2020. *Code Sample: Intel® Deep Learning Boost New Deep Learning Instruction bfloat16 - Intrinsic Functions*. https://software.intel.com/content/www/us/en/develop/articles/intel-deep-learning-boost-new-instruction-bfloat16.html

[67] Intel. 2020. *Overview: Intel® Math Library*. https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/intel-math-library/overview-intel-math-library.html

[68] ECMA International. 2020. *ECMA-262, 11th edition, June 2020 ECMAScript 2020 Language Specification*. https://262.ecma-international.org/11.0/

[69] Manish Kumar Jaiswal. 2017. *Universal number Posit HDL Arithmetic Architecture generator*. https://github.com/manish-kj/Posit-HDL-Arithmetic

[70] Manish Kumar Jaiswal and Hayden K.-H So. 2018. Universal number posit arithmetic generator on FPGA. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1159–1162.

[71] 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

[72] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 215–224. https://doi.org/10.1145/1806799.1806833

[73] Fredrik Johansson. 2014. Efficient implementation of elementary functions in the medium-precision range. *CoRR* abs/1410.7176 (2014). arXiv:1410.7176 http://arxiv.org/abs/1410.7176

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

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

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

[77] Milan Klöwer, Peter D. Düben, and Tim N. Palmer. 2019. Posits As an Alternative to Floats for Weather and Climate Models. In *Proceedings of the Conference for Next Generation Arithmetic 2019* (Singapore, Singapore) *(CoNGA'19)*. ACM, New York, NY, USA, Article 2, 8 pages.

[78] Urs Köster, Tristan J. Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William H. Constable, Oğuz H. Elibol, Scott Gray, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, and Naveen Rao. 2017. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. 1740–1750.

[79] Olga Kupriianova and Christoph Lauter. 2014. A domain splitting algorithm for the mathematical functions code generator. In *2014 48th Asilomar Conference on Signals, Systems and Computers*. 1271–1275. https://doi.org/10.1109/ACSSC.2014.7094664

[80] 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

[81] Olga Kupriianova and Christoph Lauter. 2015. Replacing Branches by Polynomials in Vectorizable Elementary Functions. In *Scientific Computing, Computer Arithmetic, and Validated Numerics*, Marco Nehmeier, Jürgen Wolff von Gudenberg, and Warwick Tucker (Eds.). Springer International Publishing, Cham, 14–22. https://doi.org/10.1007/978-3-319-31769-4_2

[82] Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2016. Verifying Bit-Manipulations of Floating-Point. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 70–84. https://doi.org/10.1145/2908080.2908107

[83] Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2017. On Automatically Proving the Correctness of Math.h Implementations. *Proceedings of the ACM Programming Langguages* 2, POPL, Article 47 (Dec. 2017), 32 pages. https://doi.org/10.1145/3158135

[84] Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2017. On Automatically Proving the Correctness of Math.h Implementations. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 47 (Dec. 2017), 32 pages. https://doi.org/10.1145/3158135

[85] Vincent Lefèvre and Jean-Michel Muller. 2001. Worst Cases for Correct Rounding of the Elementary Functions in Double Precision. In *15th IEEE Symposium on Computer Arithmetic (Arith '01)*. 111–118. https://doi.org/10.1109/ARITH.2001.930110

[86] 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

[87] Vincent Lefèvre and Jean-Michel Muller. 2003. On-the-Fly Range Reduction. *Journal of VLSI Signal Processing* 33 (01 2003), 31–35. https://doi.org/10.1023/A:1021137717282

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

[89] 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 [cs.MS]

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

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

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

[93] Jay P. Lim and Santosh Nagarakatte. 2021. High Performance Correctly Rounded Math Libraries for 32-Bit Floating Point Representations. In *Proceedings of the 42nd*

ACM SIGPLAN International Conference on Programming Language Design and Implementation (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 359–374. https://doi.org/10.1145/3453483.3454049

[94] Jay P. Lim and Santosh Nagarakatte. 2021. *RLibm-32*. https://github.com/rutgers-apl/rlibm-32

[95] 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.

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

[97] Jay P. Lim, Matan Shachnai, and Santosh Nagarakatte. 2020. Approximating Trigonometric Functions for Posits Using the CORDIC Method. In *Proceedings of the 17th ACM International Conference on Computing Frontiers* (Catania, Sicily, Italy) *(CF '20)*. Association for Computing Machinery, New York, NY, USA, 19–28. https://doi.org/10.1145/3387902.3392632

[98] Pramod K. Meher, Javier Valls, Tso-Bing Juang, K. Sridharan, and Koushik Maharatna. 2009. 50 Years of CORDIC: Algorithms, Architectures, and Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2009).

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

[100] Microsoft. 2019. *(Cloud) Acceleration at Microsoft*. https://old.hotchips.org/hc31/HC31_T2_Microsoft_CarrieChiouChung.pdf

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

[102] Raúl Murillo Montero, Alberto A. Del Barrio, and Guillermo Botella. 2019. Template-Based Posit Multiplication for Training and Inferring in Neural Networks. arXiv:1907.04091 [cs.CV]

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

[104] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. 2018. *Handbook of Floating-Point Arithmetic* (2nd ed.). Birkhäuser Basel.

[105] Kwok C. Ng. 1992. *Argument reduction for huge arguments: Good to the last bit (can be obtained by sending an e-mail to the author: kwok.ng@eng.sun.com*. Technical Report. SunPro.

[106] Ivan Niven. 1956. *Irrational Numbers*. Mathematical Association of America.

[107] NVIDIA. 2020. *NVIDIA AMPERE ARCHITECTURE*. https://www.nvidia.com/en-us/data-center/nvidia-ampere-gpu-architecture/

[108] 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/

[109] Oracle. 2015. *Oracle® Solaris Studio 12.4: Numerical Computation Guide*. https://docs.oracle.com/cd/E37069_01/html/E39019/z4000ac119729.html

[110] James M. Ortega and Werner C. Rheinboldt. 2000. *Iterative Solution of Nonlinear Equations in Several Variables*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719468

[111] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vol. 50. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2813885.2737959

[112] Behrooz Parhami. 2020. Computing with logarithmic number system arithmetic: Implementation methods and performance benefits. *Computers & Electrical Engineering* 87 (2020), 106800.

[113] Mary H. Payne and Robert N. Hanek. 1983. Radian Reduction for Trigonometric Functions. *ACM SIGNUM Newslett.* 18, 1 (Jan. 1983), 19–24. https://doi.org/10.1145/1057600.1057602

[114] 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.

[115] Denis Roegel. 2010. *A reconstruction of the tables of Briggs' ¡i¿Arithmetica logarithmica¡/i¿ (1624)*. Research Report. https://hal.inria.fr/inria-00543939

[116] Bita Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, Alessandro Forin, Haishan Zhu, Taesik Na, Prerak Patel, Shuai Che, Lok Chand Koppaka, Xia Song, Subhojit Som, Kaustav Das, Saurabh Tiwary, Steve Reinhardt, Sitaram Lanka, Eric Chung, and Doug Burger. 2020. Pushing the Limits of Narrow Precision Inferencing at Cloud Scale with Microsoft Floating Point. In *The Thirty-fourth Annual Conference on Neural Information Processing Systems*. ACM.

[117] Hughes Saint-Genies, David Defour, and Guillaume Revy. 2017. Exact Lookup Tables for the Evaluation of Trigonometric and Hyperbolic Functions. *IEEE Trans. Comput.* 66, 12 (dec 2017), 2058–2071. https://doi.org/10.1109/TC.2017.2703870

[118] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2018. Finding Root Causes of Floating Point Error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 256–269. https://doi.org/10.1145/3296979.3192411

[119] 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*.

[120] Michael J. Schulte and James E. Stine. 1997. Symmetric bipartite tables for accurate function approximation. In *Proceedings 13th IEEE Sympsoium on Computer Arithmetic*. 175–183. https://doi.org/10.1109/ARITH.1997.614893

[121] Michael J. Schulte and James E. Stine. 1999. Approximating elementary functions with symmetric bipartite tables. *IEEE Trans. Comput.* 48, 8 (1999), 842–847. https://doi.org/10.1109/12.795125

[122] Shen-Fu Hsiao and Jean-Marc Delosme. 1995. Householder CORDIC algorithms. *IEEE Trans. Comput.* (1995).

[123] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by Sketching for Bit-Streaming Programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. New York, NY, USA, 281–294. https://doi.org/10.1145/1065010.1065045

[124] Michael Spivak. 1971. *Calculus On Manifolds: A Modern Approach To Classical Theorems Of Advanced Calculus*. Avalon Publishing.

[125] Damien Stehle, Vincent Lefevre, and Paul Zimmermann. 2005. Searching worst cases of a one-variable function using lattice reduction. *IEEE Trans. Comput.* 54, 3 (2005), 340–346. https://doi.org/10.1109/TC.2005.55

[126] Pat H Sterbenz. 1974. *Floating-point computation*. Prentice-Hall, Englewood Cliffs, NJ.

[127] James E. Stine and Michael J. Schulte. 1999. The Symmetric Table Addition Method for Accurate Function Approximation. In *The Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*. https://doi.org/10.1023/A:1008004523235

[128] 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

[129] David A. Sunderland, Roger A. Strauch, Steven S. Wharfield, Henry T. Peterson, and Christopher R. Cole. 1984. CMOS/SOS frequency synthesizer LSI circuit for

spread spectrum communications. *IEEE Journal of Solid-State Circuits* 19, 4 (1984), 497–506. https://doi.org/10.1109/JSSC.1984.1052173

[130] Earl E. Swartzlander and Aristides G. Alexopoulos. 1975. The Sign/Logarithm Number System. *IEEE Trans. Comput.* C-24, 12 (1975), 1238–1242. https://doi.org/10.1109/T-C.1975.224172

[131] 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

[132] Naofumi Takagi, Tohru Asada, and Shuzo Yajima. 1991. Redundant CORDIC methods with a constant scale factor for sine and cosine computation. *IEEE Trans. Comput.* (1991).

[133] 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

[134] 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

[135] 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

[136] Sugandha Tiwari, Neel Gala, Chester Rebeiro, and V. Kamakoti. 2019. PERI: A Posit Enabled RISC-V Core. arXiv:1908.01466 [cs.AR]

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

[138] Tso-Bing Juang, Shen-Fu Hsiao, and Ming-Yu Tsai. 2004. Para-CORDIC: parallel CORDIC rotation algorithm. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2004).

[139] Jack Volder. 1959. The CORDIC Computing Technique. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference (IRE-AIEE-ACM '59 (Western))*.

[140] John S. Walther. 1971. A Unified Algorithm for Elementary Functions. In *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference (AFIPS '71 (Spring))*.

[141] Dong Wang, Jean-Michel Muller, Nicolas Brisebarre, and Miloš D. Ercegovac. 2014. (M, p, k)-Friendly Points: A Table-Based Method to Evaluate Trigonometric Function. *IEEE Transactions on Circuits and Systems II: Express Briefs* 61, 9 (2014), 711–715. https://doi.org/10.1109/TCSII.2014.2331094

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

[143] Shaoyun Wang, Vincenzo Piuri, and Earl E. Wartzlander. 1997. Hybrid CORDIC algorithms. *IEEE Trans. Comput.* (1997).

[144] Karl Weierstrass. 1885. Über die analytische Darstellbarkeit sogenannter willkürlicher Functionen reeller Argumente. In *Sitzungsberichte der Königlich Preussischen Akademie der Wissenschaften zu Berlin*.

[145] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2019. Efficient Automated Repair of High Floating-Point Errors in Numerical Libraries. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 56 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290369

[146] 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

[147] Daming Zou, Muhan Zeng, Yingfei Xiong, Zhoulai Fu, Lu Zhang, and Zhendong Su. 2019. Detecting Floating-Point Errors via Atomic Conditions. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 60 (Dec. 2019), 27 pages. https://doi.org/10.1145/3371128