Approximating Trigonometric Functions for Posits Using the CORDIC Method

Jay P. Lim
Department of Computer Science
Rutgers University
Piscataway, USA
jpl169@cs.rutgers.edu

Matan Shachnai
Department of Computer Science
Rutgers University
Piscataway, USA
mys35@scarletmail.rutgers.edu

Santosh Nagarakatte
Department of Computer Science
Rutgers University
Piscataway, USA
santosh.nagarakatte@cs.rutgers.edu

ABSTRACT

Posit is a recently proposed representation for approximating real numbers using a finite number of bits. In contrast to the floating point (FP) representation, posit provides variable precision with a fixed number of total bits (*i.e.*, tapered accuracy). Posit can represent a set of numbers with higher precision than FP and has garnered significant interest in various domains. The posit ecosystem currently does not have a native general-purpose math library.

This paper presents our results in developing a math library for posits using the CORDIC method. CORDIC is an iterative algorithm to approximate trigonometric functions by rotating a vector with different angles in each iteration. This paper proposes two extensions to the CORDIC algorithm to account for tapered accuracy with posits that improves precision: (1) fast-forwarding of iterations to start the CORDIC algorithm at a later iteration and (2) the use of a wide accumulator (*i.e.*, the quire data type) to minimize precision loss with accumulation. Our results show that a 32-bit posit implementation of trigonometric functions with our extensions is more accurate than a 32-bit FP implementation.

CCS CONCEPTS

• **Computing methodologies** → **Representation of mathematical functions**; **Algebraic algorithms**.

KEYWORDS

CORDIC, trigonometric functions, posit, math libraries

ACM Reference Format:
Jay P. Lim, Matan Shachnai, and Santosh Nagarakatte. 2020. Approximating Trigonometric Functions for Posits Using the CORDIC Method. In *17th ACM International Conference on Computing Frontiers (CF '20), May 11–13, 2020, Catania, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3387902.3392632

1 INTRODUCTION

Approximating real numbers is essential for almost all domains in computing. Floating point (FP) [2, 18] is a widely used approximation of real numbers with a finite number of bits. An FP number is abstractly represented as $F \times 2^e$ where $F \in [1, 2)$ and it uses a fixed number of bits to represent the exponent and the significand. Due to its wide usage, there are numerous tools and math libraries for FP [9–11, 14, 17]. Given the need for more performance, there are numerous efforts to explore alternatives to FP [3, 19, 23, 24, 33].

Posit [20, 21], which was recently proposed by John Gustafson, is one such alternative to FP that provides variable precision. A $\langle n, es \rangle$-posit is an n-bit representation consisting of one sign bit, a flexible number of regime bits, a regime guard bit, up to es bits for the exponent, and the remaining bits for the fraction (Section 2 provides more details on posits). Depending on the value, some of the regime bits can be re-purposed for fraction, which provides tapered accuracy. In the posit representation, values near 1 have the most precision. In contrast to FP of the same bit-width, posit can provide more precision for a range of values. For example, a $\langle 32, 2 \rangle$-posit provides four additional precision bits than 32-bit float for values between $[\frac{1}{16}, 16]$. Posit can also represent more distinct values than FP because it has one bit-pattern for zero and one bit-pattern for representing exceptions (32-bit float has two zeros and $2^{24} - 2$ NaNs). Given this tapered accuracy, there is a growing interest in posits [4, 5, 13, 24, 25, 29, 37].

The posit ecosystem does not have a general-purpose math library, yet. The only available math library is limited to 16-bit posit [30]. Math libraries are used in many real-world applications to approximate commonly used elementary functions. In the absence of posit math libraries, developers use existing FP libraries by casting a posit value to an FP number and subsequently casting the FP result back to a posit value. Although a good stop-gap measure, there are two reasons why using FP math libraries for approximating elementary functions for posits is undesirable. First, depending on the choice of the FP type, not all posit values may be representable. Any general-purpose representation should have a default math library. Second, FP math libraries are developed for approximating FP values accurately. Because FP has fixed precision regardless of the magnitude of the value, the goal of its math library is to provide the same amount of precision for all values. On the contrary, posit values have the most precision near 1 and less precision as the value increases (or decreases). Therefore, an accurate posit math library must approximate values near 1 precisely compared to FP. However, very small and large values can be approximated less precisely compared to FP.

CORDIC method for posits. This paper makes a case for exploring the CORDIC method with posits. The **Co**ordinate **R**otation **Di**gital **C**omputer (CORDIC) algorithm [44] is an iterative algorithm to compute trigonometric functions. Intuitively, CORDIC rotates a vector $[x, y]^T$ by a decreasing amount of angles $\theta_i = atan(2^{-i})$ for $i \geq 0$ to compute trigonometric functions of the desired angle (Section 2.2 provides a detailed background). The CORDIC algorithm is known for its simplicity. It has been used in the Lunar Rover's vehicle navigation system [1] and calculators [8, 43]. Extensions to the CORDIC algorithm are available to approximate hyperbolic functions and a wide range of other elementary functions [45].

This paper. This paper describes our effort in building a math library for posits using the CORDIC method. We address the issue of cancellation of bits and loss of precision when using the CORDIC algorithm with posits. To mitigate these issues, we propose two extensions to the CORDIC algorithm. First, instead of starting the CORDIC method from the first iteration, we propose to fast-forward the iterations to a later iteration depending on the magnitude of the input value. We provide a closed form solution to identify the latest iteration at which one can accurately compute the final result. Second, we propose to perform certain computations using a high-precision accumulator, which is called a quire in the posit ecosystem. We have implemented these techniques and compared them against a CORDIC algorithm implemented using floats. The CORDIC method with posits produces results that are more accurate than the corresponding FP versions. Our CORDIC library for posits is 5× faster and produces reasonably accurate results when compared to a high-precision library (*i.e.*, with GNU MPFR) that uses 1024 bits of precision.

Contributions. In summary, this paper makes the following contributions.

- Makes a case for using the CORDIC method with posits for developing a general purpose math library.
- Proposes two extensions to the CORDIC method to account for tapered accuracy with posits: fast-forwarding of iterations and use of a high-precision accumulator.
- Demonstrates that CORDIC for posits can produce results that are more accurate than the FP implementation.

2 BACKGROUND

We provide a background on the posit representation and the CORDIC method for approximating trigonometric functions.

2.1 Posits

Posit is a representation for approximating real numbers with a finite number of bits, which is intended to be a stand-in-replacement for FP. Given the same number of bits, posit can provide better precision than FP for a range of values via tapered accuracy.

A posit bit-pattern. A $\langle n, es\rangle$-posit environment has n bits in total and at most es bits are used to represent the exponent. The first bit represents the sign bit (s), where $s = 0$ represents a positive value and $s = 1$ represents a negative value. If $s = 1$, then the two's complement of the rest of the bit-pattern is computed before decoding other bits. The next consecutive bits of 0's or 1's are called regime bits (r) and represents the super exponent. The number of regime bits are bounded by $1 \le |r| \le n-1$. If $|r| < n-1$, then the regime bits are terminated by the opposite bit (1 or 0, respectively) known as the regime guard bit, $\overline{r}$. The regime is a super exponent. When the regime bits are not needed to represent a number, these can be re-purposed to provide precision, which provides tapered accuracy.

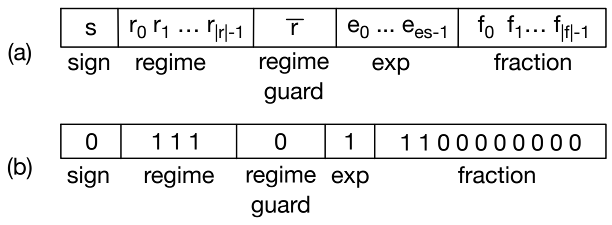


Figure 1: (a) The posit representation. A posit bit-pattern contains a sign bit, regime bits consisting of consecutive 0's or 1's, followed by an opposite bit for the regime guard, exponent bits, and fraction bits. (b) In $\langle 16, 1\rangle$-posits, the bit-string `0x7700` represents the value $4^2 \times 2^1 \times 1.75 = 56$.

If $|r|+2 < n$, the next $min(es, n-r-2)$ bits represent the exponent bits e, where $0 \le |e| \le es$. When $|e| < es$, then the exponent bits are padded with $es - |e|$ number of 0 bits to the right. Finally, if $|r| + 2 + es < n$, the remaining bits represent the fraction bits, f.

Interpreting a posit bit-pattern. First, the regime bits are used to compute the magnitude of the super exponent,

$$useed^m \;\; where \;\; m = \begin{cases} |r| - 1 & if \;\; r = 11\dots1 \\ -|r| & if \;\; r = 00\dots0 \end{cases}$$

where $useed = 2^{2^{es}}$. The exponent bits are treated as an unsigned integer to encode the exponent component, 2^e. The maximum value representable by the exponent component is $2^{2^{es}-1} = \frac{useed}{2}$. Thus, regime bits extend the range of the exponent component while providing tapered accuracy. The fraction bits are interpreted similar to the normalized values in FP,

$$F = 1 + \frac{f}{2^{|f|}}$$

where f is interpreted as an unsigned integer and $|f|$ represents the number of bits used for the fractional part. Finally, the value represented by posit bit-pattern is:

$$(-1)^s \times useed^m \times 2^e \times (1 + \frac{f}{2^{|f|}})$$

As an example, consider the bit-pattern `0x7700` in the posit environment $\langle 16, 1\rangle$ (see Figure 1(b)). Here, $useed = 2^{2^1} = 4$. The value represented by this bit-pattern is $(-1)^0 \times 4^{(3-1)} \times 2^1 \times 1.75 = 56$.

Special values. There are two special values: the posit bit-pattern of all 0's represents zero and the posit bit-pattern of a 1 followed by all zeros represents *Not a Real* (NaR), which is an exception.

Rounding mode. When a value cannot be exactly represented, the posit standard requires the value to be rounded to the nearest representable value with ties going to even.

Additionally, there are no overflows or underflows with posits. All real values p greater than maximum representable value, $p \ge maxpos = (2^{2^{es}})^{n-2}$, are rounded to $maxpos$. Likewise, all values less than minimum representable value excluding 0, $0 < p \le minpos = (2^{2^{es}})^{-(n-2)}$, are rounded to $minpos$.

The quire data type. The posit standard also specifies a quire data type, which is a high precision accumulator that supports accurate computation of a series of multiply-add operations [26, 27]. The quire is required to provide enough bits to express both $maxpos^2$ and $minpos^2$.

Advantages of posits. First, posit can represent more unique values than FP for a given bit-width because posit has a single bit-pattern representing NaR whereas FP has multiple bit-patterns for NaNs. Second, posit can provide more precision than FP for a certain range of values due to tapered accuracy. For example, in a $\langle 32, 3\rangle$-posit configuration, a posit value $p \in [2^{-8}, 2^{8})$ has 26 fraction bits, where as a 32-bit float has 23 fraction bits. All posit values $p \in [2^{-32}, 2^{32})$ have equal or more fraction bits compared to FP32, yet the dynamic range of $\langle 32, 3\rangle$ spans $[2^{-240}, 2^{240}]$ compared to $[2^{-149}, 2^{127})$ for a 32-bit float. The posit standard defines $\langle 32, 2\rangle$ environment as the default representation for 32 bit. It provides up to 27 fraction bits for values $x \in [2^{-4}, 2^{4})$, equal or more fraction bits compared to a 32-bit float for values $x \in [2^{-20}, 2^{20})$ (known as the golden zone) and has a dynamic range of $[2^{-120}, 2^{120}]$.

2.2 The CORDIC Method

Coordinate **R**otation **Di**gital **C**omputer (CORDIC) [44] is a variant of iterative add-shift algorithms for approximating trigonometric functions. While not the most efficient, CORDIC is popular for its simplicity (in hardware) and can be easily extended to compute other elementary functions.

The idea behind CORDIC. The key idea behind CORDIC is to compute the value of a trigonometric function through a series of rotations of a 2-dimensional vector. When we want to compute the trigonometric function for an angle θ, the angle θ is decomposed into a series of small decreasing angles that satisfies certain properties (described below). Subsequently, a suitably initialized vector is rotated through these angles in each iteration. At the end of this process, the components of this vector or the accumulated θ's represent the value of trigonometric functions.

Vector rotation. When a vector $[x, y]^T$ is rotated by θ, the resulting vector $[x', y']^T$ is given by:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

If $[1, 0]^T$ is rotated by angle θ, the resulting vector is equivalent to $[cos(\theta), sin(\theta)]^T$. After simplification, we can express the above formula for vector rotation as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \frac{1}{\sqrt{1 + tan^2(\theta)}} \begin{bmatrix} 1 & -tan(\theta) \\ tan(\theta) & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Now if we choose angle θ_i such that $tan(\theta_i) = 2^{-i}$, where $i \geq 0$ is a non-negative integer, the above matrix multiplication operations can be calculated using only addition, multiplication by 2^{-i} (shifts), and a square root operation.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \frac{1}{\sqrt{1 + 2^{-2i}}} \begin{bmatrix} 1 & -2^{-i} \\ 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

The next step is to decompose the angle θ into a series of decreasing angles θ_i such that $tan(\theta_i) = 2^{-i}$ using the non-restoring decomposition property.

Non-restoring decomposition property. The non-restoring decomposition property states that if an ordered sequence of values $(\omega_0, \omega_1, \omega_2, ...)$ are strictly decreasing, $\omega_i > 0$ for all $i \geq 0$, $\sum_{i=0}^{\infty} \omega_i$ converges, and

$$\forall_i \; \omega_i \leq \sum_{k=i+1}^{\infty} \omega_k$$

then any value $t \in [-\sum_{i=0}^{\infty} \omega_i, \sum_{i=0}^{\infty} \omega_i]$ can be computed as $t = \sum_{i=0}^{\infty} d_i \omega_i$ where $d_i \in \{1, -1\}$.

The sequence $(\theta_0, \theta_1, \theta_2, ...)$ where $\theta_i = atan(2^{-i})$ satisfies the above property. Any value of $\theta \in [-\sum_{i=0}^{\infty} \theta_i, \sum_{i=0}^{\infty} \theta_i]$ can be computed as $\theta = \sum_{i=0}^{\infty} d_i \theta_i$. In the context of CORDIC, this property asserts that it is possible to compute $sin(\theta)$ and $cos(\theta)$ for $\theta \in [-1.74328..., 1.74328...]$ (superset of $[-\frac{\pi}{2}, \frac{\pi}{2}]$) by iteratively rotating the vector $[1, 0]^T$ with θ_i.

Summary. The general CORDIC method, which is also known as the rotation mode, iteratively rotates $[1, 0]^T$ by θ_i,

$$x_0 = 1, \;\; y_0 = 0, \;\; z_0 = \theta$$
$$x_{i+1} = K_i(x_i - d_i y_i 2^{-i})$$
$$y_{i+1} = K_i(y_i + d_i x_i 2^{-i})$$
$$z_{i+1} = z_i - d_i atan(2^{-i})$$
$$K_i = \frac{1}{\sqrt{1 + 2^{-2i}}}, \quad d_i = \begin{cases} 1 & if \;\; z_i \geq 0 \\ -1 & otherwise \end{cases}$$

The value K_i is known as the scaling factor, as it normalizes the length of the vector $[x_i - d_i y_i 2^{-i}, y_i + d_i x_i 2^{-i}]^T$. In each iteration, $x_i = cos(\theta - z_i)$ and $y_i = cos(\theta - z_i)$, and $z_i = \theta - \sum_{k=0}^{i-1} d_i \theta_i$. Intuitively, the goal of CORDIC algorithm is to reduce z_i, the difference between θ and the accumulated angle of rotation, to 0. As such, $x_\infty = cos(\theta)$, $y_\infty = sin(\theta)$, and $z_\infty = 0$.

Optimized CORDIC algorithm. The above algorithm still requires the computation of K_i and θ_i. It can be further optimized by factoring out K_i such that $K = \prod_{i=0}^{n-1} K_i$, and directly multiplied to the initial x_0 and y_0 if the number of iterations, n, is known ahead of time. Hence, K and θ_i can be precomputed in a table. The optimized algorithm is as follows:

$$x_0 = K, \;\; y_0 = 0, \;\; z_0 = \theta$$
$$x_{i+1} = (x_i - d_i y_i 2^{-i})$$
$$y_{i+1} = (y_i + d_i x_i 2^{-i})$$
$$z_{i+1} = z_i - d_i \theta_i$$
$$d_i = \begin{cases} 1 & if \;\; z_i \geq 0 \\ -1 & otherwise \end{cases}$$

Computing atan. The CORDIC method used to compute $atan(\frac{y}{x})$ for $x > 0$, which is also known as vectoring mode, rotates the initial vector $[x, y]^T$ towards $[\sqrt{x^2 + y^2}, 0]^T$ and accumulates the angle in the process. Optimized vectoring mode algorithm is as follows:

$$x_0 = x, \;\; y_0 = y, \;\; z_0 = 0$$
$$x_{i+1} = x_i + d_i y_i 2^{-i}$$
$$y_{i+1} = y_i - d_i x_i 2^{-i}$$
$$z_{i+1} = z_i + d_i \theta_i$$
$$d_i = \begin{cases} 1 & if \;\; y_i \geq 0 \\ -1 & otherwise \end{cases}$$

Consequently, $x_\infty = K\sqrt{x^2 + y^2}$, $y_\infty = 0$, and $z_\infty = atan(\frac{y}{x})$.

3 NAIVE CORDIC USING POSITS

The goal of our approach is to use the CORDIC method to generate approximations of trigonometric functions using posits. The availability of additional precision bits as a result of tapered accuracy with posits makes it an attractive candidate for the CORDIC algorithm. However, tapered accuracy with posits also makes it lose precision bits with small and large numbers. A naive straightforward implementation with posits using the method described in Section 2.2 can cause errors for the following reasons: (1) rounding errors with the computed K and θ_i values, (2) cancellation error that occurs during subtraction operations while computing x_i, y_i, and z_i, and (3) loss of precision bits in the computation of $x_i 2^{-i}$ and $y_i 2^{-i}$ due to tapered accuracy.

Rounding errors with K and θ_i. A large number of precision bits may be needed to accurately compute the values of K. They have to be rounded when maintained in the posit representation, which can lead to imprecise intermediate and final results. In our CORDIC implementation, we precompute these values for K using higher precision (*i.e.*, MPFR [14]) and then round them to the nearest value, which still introduces some rounding errors.

Similar to K, θ_i may also need a large number of precision bits. Moreover, θ_i becomes progressively smaller as i increases. Thus, the rounded posit value of precomputed θ_i can have small number of precision bits, which magnifies the rounding error.

Cancellation errors. Cancellation error occurs when two inexact (*i.e.*, rounded) values of similar magnitude are subtracted. When two values are similar, the significant bits of the fractional part are identical and get canceled. If the remaining bits are all influenced by rounding error, then the subtraction has catastrophic cancellation. Here, all bits in the result are influenced by rounding error.

A major contributing factor to error with the posit version of CORDIC is in the computation of x_i, y_i, and z_i. The subtraction that produces these values incurs cancellation. The cancellation error of z_i can even evaluate $z_i \geq 0$ incorrectly compared to a CORDIC version with real numbers (*e.g.*, MPFR), which leads to divergence of the program execution path compared to the ideal execution.

Example. We show an illustration of cancellation with the CORDIC method using $\langle 32, 2\rangle$-posit when we are computing $sin(0)$ and $cos(0)$ (*i.e.*, $\theta = 0$). The computation of z_{i+1} and y_{i+1} has a large number of subtraction operations. While this does not cause cancellation error in every iteration, it increases the likelihood. We observe that with posits, there are enough occurrences of cancellation error such that the result of y_i and z_i has no correct fraction bits after a number of iterations. Table 1 highlights the intermediate result of x_i, y_i, z_i, and d_i for some iterations i. The inexact digits are in bold font. With cancellation, some values have fraction bits to be inexact and others can have even the exponent and the sign to be inexact. Table 1 shows that by the 27^{th} iteration, both y_{27} and z_{27} experience cancellation error such that none of the fraction digits are computed correctly. By the 31^{st} iteration, the sign of z_{31} is computed incorrectly and causes wrong evaluation of $z_{31} \geq 0$. This causes the implementation to incorrectly compute d_{31}, which eventually leads to inaccurate results.

Precision loss from multiplication. Multiplication by 2^{-i} can increase the relative error of any $\langle 32, 2\rangle$-posit value if the magnitude of the value reduces enough to require more regime bits to represent the value. In the CORDIC method, $y_i 2^{-i}$ (resp. $x_i 2^{-i}$) can have less precision than y_i (resp. x_i) for $i \geq 1$. This loss of precision can contribute to increased error in $y_i 2^{-i}$. In contrast to posits, this loss of precision with multiplication does not occur with FP because every normalized value has fixed precision bits.

Table 1: Intermediate values of x_i, y_i, z_i, and d_i from the CORDIC algorithm with $\langle 32, 2\rangle$-posits for $\theta = 0$. Each column show the iteration number, x_i, y_i, z_i, and d_i for a number of iterations. The sign, digit, and the exponent for each value in bold font indicates that it is inexact. In the 27^{th} iteration, both y_{28} and z_{28} experience catastrophic cancellation. In the 31^{st} iteration, the incorrect result of z_{31} causes divergence of d_{31} from the ideal value, causing the program execution to diverge from the ideal execution.

i	x_i	y_i	z_i	d_i
0	0.6072529...	0	0	1
1	0.6072529...	0.6072529...	-0.7853981...	-1
...	...	...	...	...
26	1.000000...	1.8**99752...**E-8	-1.**706939...**E-8	-1
27	1.000000...	**4.096364...**E-9	**-2.168235...**E-9	-1
28	1.000000...	**-3.354216...**E-9	**5.282345...**E-9	1
...	...	...	...	...
31	1.000000...	**1.302396...E-9**	**+6.257323...E-10**	**1**
...	...	...	...	...

With respect to the final result, the amount of error arising from precision loss with multiplication may not be as significant as the cancellation error. Consider the operation $x_{i+1} = x_i - y_i 2^{-i}$. If x_i (resp. y_i) is considerably larger than $y_i 2^{-i}$ (resp. $y_i 2^{-i}$), then the error of $y_i 2^{-i}$ does not contribute much to the result of x_{i+1}. If $y_i 2^{-i}$ is considerably larger than x_i, then x_{i+1} has as many precision bits as $y_i 2^{-i}$. Otherwise, cancellation error contributes more to the error of the final result.

4 OUR CORDIC METHOD FOR POSITS

To address the challenges described in the previous section, we propose two novel modifications to the CORDIC algorithm with posits. These techniques reduce cancellation error with subtraction operations. To minimize the number of subtraction operations that can experience cancellation, we propose to start the CORDIC algorithm at the last feasible iteration (l) that can accurately compute the result rather than with the first iteration. Even when θ is small, the naive CORDIC algorithm starts with the 0^{th} iteration (rotation by $\frac{\pi}{4}$), which requires a lot of subtractions. Our technique reduces the number of subtraction operations. The first executing iteration with our technique approximates θ more closely than the first iteration of the naive CORDIC method. To improve the precision of z_i, we propose to compute z_i using the quire data type. The quire datatype with posits provides enough precision bits to avoid rounding error in the intermediate result of z_i.

4.1 Fast-Forwarded Iterations

The CORDIC algorithm uses a number of subtraction operations when approximating $sin(\theta)$ and $cos(\theta)$ for small θ, which can cause cancellation error. Consider an example where we use the CORDIC algorithm to approximate $sin(\theta)$ when $\theta = 0.12$ radian. CORDIC rotates the initial vector with a sequence of $(\theta_0, -\theta_1, -\theta_2, \theta_3, -\theta_4, ...)$. The sequence of d_i in the CORDIC algorithm is $(1, -1, -1, 1, -1, ...)$. Consequently, there are a number of subtractions while computing x_i, y_i, and z_i, where cancellation errors can occur. The computation of y_i is more susceptible to cancellation error because $y_0 = 0$ and results in subtractions with values of similar magnitude. Since $x_0 = K$ and $z_0 = \theta$, x_i and z_i are less susceptible to cancellation error.

Our key idea is to fast-forward the iterations to approximate the angle under consideration more quickly. For example, it is possible to approximate 0.12 radian with θ_i's if we start at iteration 4: $\theta_4 + \theta_5 + \theta_6 + \theta_7 + \theta_8 \ldots$. This eliminates the subtraction operations in the first few iterations. The sequence of d_i's after starting at iteration 4 is $(1, 1, 1, 1, 1, ...)$. In this case, the computation of y_5, y_6, ... y_9 all involve additions. Any subsequent y_i computation can involve subtraction, but $x_i 2^{-i}$ will be a value substantially smaller than y_i to cause cancellation error. Similarly, computation of x_i is less likely to experience cancellation error since $y_i 2^{-i}$ for $i \geq 4$ is much smaller than the value of x_i for $i \geq 4$.

The key task in fast-forwarding iterations is to identify an iteration where we can still approximate the angle of interest. In the above example, we need to be careful because if we start at iteration 5, then we cannot approximate 0.12 radian. More specifically, $0.12 \notin [-\sum_{i=5}^{\infty} \theta_i, \sum_{i=5}^{\infty} \theta_i]$. Therefore, it is not possible to correctly approximate 0.12 radian if we start at iteration 5.

Our goal is to start at the latest possible iteration l such that we minimize the number of subtraction operations while also ensuring that we can correctly compute θ by starting at l. More specifically, we have to ensure that $\theta \in [-\sum_{i=l}^{\infty} \theta_i, \sum_{i=l}^{\infty} \theta_i]$

Revisiting Non-restoring decomposition property. The non-restoring decomposition property of θ_i's guarantee that any $|\theta| \leq \sum_{i=0}^{\infty} \theta_i$ can be computed using $\theta = \sum_{i=0}^{\infty} d_i \theta_i$. This property also applies to all consecutive infinite sequences, $(\theta_l, \theta_{l+1}, \theta_{l+2}, \ldots)$, $l > 0$. Any $|\theta| \leq \sum_{i=l}^{\infty} \theta_i$ can be computed using $t = \sum_{i=l}^{\infty} d_i \theta_i$ because the sequence satisfies the necessary conditions for using the non-restoring decomposition property. Hence, CORDIC can correctly compute θ at iteration l as long as $|\theta| \leq \sum_{i=l}^{\infty} \theta_i$.

Modified CORDIC algorithm. We modify the CORDIC algorithm to start at a later iteration depending on the value of θ. More specifically, we start the algorithm at iteration l, where $l = max\{k \mid k \geq 0,\ |\theta| \leq \sum_{j=k}^{\infty} \theta_j\}$. The initial values x_l, y_l, and z_l also change accordingly. Our modified algorithm is:

$$x_l = K' = \prod_{k=l}^{l+n-1} K_k,\ \ y_l = 0,\ \ z_l = \theta$$

$$x_{i+1} = (x_i - d_i y_i 2^{-i})$$

$$y_{i+1} = (y_i + d_i x_i 2^{-i})$$

$$z_{i+1} = z_i - d_i \theta_i$$

$$d_i = \begin{cases} 1 & if\ \ z_i \geq 0 \\ -1 & otherwise \end{cases}$$

$$l = max\{k \mid k \geq 0,\ |\theta| \leq \sum_{j=k}^{\infty} \theta_j\}$$

where $n > 0$ is the total number of iterations executed.

The value K' depends on both the starting iteration l and the number of iterations n. We precompute a table for K'. In a $\langle 32, 2\rangle$-posit, K' rounds to 1.0 for $l > 14$ and K' rounds to a constant value for all $0 \leq l \leq 14$ as long as $n \geq 17$. Using this observation, the size of the precomputed table for K' is just 16 entries as long as the number of iterations n is greater than or equal to 17.

Identifying the starting iteration. Identifying the correct l is compute intensive even if we store $\sum_{j=k}^{\infty} \theta_j$ for $k \geq 0$ in a precomputed table. Instead, we approximate l using the formula $l = max\{0, -e_\theta - 1\}$, where e_θ is the exponent of θ in the posit representation, *i.e.* $|\theta| = F_\theta \times 2^{e_\theta}$ and $F_\theta \in [1, 2)$. This formula guarantees that the CORDIC algorithm starting at iteration l can compute the result for θ correctly.

Sketch of the proof for identifying the iteration. To prove that $l = max\{0, -e_\theta - 1\}$ is sufficient to identify the correct iteration for fast-forwarding, we must identify l in terms of e_θ such that,

$$|\theta| = F_\theta \times 2^{e_\theta} < 2^{e_\theta + 1} \leq \sum_{i=l}^{\infty} \theta_i$$

Observe that $atan(0) = 0$ and $\frac{\mathrm{d}atan(x)}{\mathrm{d}x} \geq \frac{1}{2}$ for $0 \leq x \leq 1$. This implies that $\theta_i = atan(2^{-i}) \geq 2^{-i-1}$ for $i \geq 0$. Then,

$$\sum_{i=l}^{\infty} \theta_i = \sum_{i=l}^{\infty} atan(2^{-i}) \geq \sum_{i=l}^{\infty} 2^{-i-1} = 2^{-l}$$

Consequently, any l that satisfies $2^{e_\theta + 1} \leq 2^{-l}$ also satisfies $|\theta| < \sum_{i=l}^{\infty} \theta_i$. It follows that $l \leq -e_\theta - 1$.

Computing atan with vectoring mode. The vectoring mode, which computes $atan$, can also start at a later iteration l if $|atan(\frac{y}{x})| \leq \sum_{i=l}^{\infty} \theta_i$. To efficiently compute l, observe that

$$|atan(\frac{y}{x})| = |atan(\frac{F_y \times 2^{e_y}}{F_x \times 2^{e_x}})| < |atan(2^{e_y - e_x + 1})|$$

where $y = F_y \times 2^{e_y}$, $x = F_x \times 2^{e_x}$, and $F_y, F_x \in [1, 2)$. It follows that

$$|atan(2^{-(e_x - e_y - 1)})| \leq \sum_{i=e_x - e_y}^{\infty} \theta_i$$

from the non-restoring decomposition property of θ_i. Therefore, it is sufficient to start at iteration $l = max\{0, e_x - e_y\}$.

Advantages of fast-forwarding iterations. There are several advantages that can be gained by starting at a later iteration, l, instead of always starting at iteration 0. First, it reduces the number of subtraction operations in the first few iterations that are executed. Second, when θ is a small value, the θ_l is a closer approximation to θ than θ_0. In the above example, 0.12 radian is closer to $\theta_4 \approx 0.0624$ radian than $\theta_0 \approx 0.7854$ radian. Because the first angle of rotation θ_l is a closer approximation to θ, the intermediate result of the first iteration is a closer approximation to the correct final result. This technique also prevents addition and subtraction of θ_0 ... θ_{l-1}, which have rounding error.

4.2 Accumulation using Quires

The precision of z_i is also important in the CORDIC algorithm. It keeps track of the difference between θ and the accumulated angle, $\sum_{j=0}^{i-1} di_j \times \theta_j$. The value of z_i determines the orientation of the rotation for the next iteration. Since the goal of CORDIC algorithm is to minimize $|z_i|$, ideally to 0, $|z_i|$ decreases each iteration and gradually loses available number of fraction bits with posits to represent the value. This loss of precision increases the chance of z_i experiencing catastrophic cancellation and resulting in wrong d_i. To provide sufficient number of fraction bits even for small z_i, we propose to compute z_i using the quire datatype.

Accuracy of θ_i values. One of the sources of imprecision for z_i stems from the rounding error of θ_i. Since we use quire to compute z_i, it is also necessary to increase the precision for θ_i. The table stored value for θ_i can have as low as 24 fraction bits.

To provide the maximum precision for θ_i during z_i computation, we store the value of θ_i as a pair of two posit values, B_{θ_i} and S_{θ_i}, such that $\theta_i = B_{\theta_i} * S_{\theta_i}$. Intuitively, B_{θ_i} stores the fractional part of θ_i, and S_{θ_i} stores the exponent part of θ_i. Hence, $S_{\theta_i} = RN(2^{-i})$ and $B_{\theta_i} = RN(\frac{\theta_i}{S_{\theta_i}})$ where $RN(x)$ correctly rounds x to a $\langle 32, 2\rangle$-posit value. Now, B_{θ_i} has 27 fraction bits for $i \geq 0$ since $B_{\theta_i} \in [2^{-4}, 2^4)$.

This representation has several advantages. First, we can provide up to 27 fraction bits for θ_i when computing $z_{i+1} = z_i + B_{\theta_i} \times S_{\theta_i}$ with the quire operation. Second, the table stored values for $S_{\theta_i} = 2^{-i}$ can be used for the computation of x_{i+1} and y_{i+1} and eliminate the need to explicitly compute 2^{-i}.

Reason for not computing x_i and y_i using quire. Unlike z_i, we do not use quires for x_i and y_i as the computation of x_i and y_i does not benefit from quire. Specifically, the computation of x_i (resp. y_i) requires an addition of $\pm y_{i-1}2^{-i+1}$ (resp. $\pm x_{i-1}2^{-i+1}$). Quires support only the multiply-add operations with two **posit values**. Hence, $\pm y_{i-1}$ (resp. $\pm x_{i-1}$) must be rounded to the nearest posit value before it is used in a quire operation, losing the benefit of the quire representation.

5 EXPERIMENTAL EVALUATION

This section describes our prototype, our methodology, and the experimental evaluation to understand the accuracy and performance of our prototype of the CORDIC method.

Prototype. We built a prototype math library for posits using the CORDIC method. The library is implemented using our CORDIC method in C++. Because commercial hardware support for posits is limited, we used the SoftPosit [29] library, a software library that implements posit operations and supports quires. We used the 32-bit posit prescribed in the standard, *i.e.*, $\langle 32, 2\rangle$-posit. We precomputed the table values (*i.e.*, K, K', and θ_i) using GNU Multiple Precision Floating-Point (MPFR) library [14] with 3000 precision bits and correctly rounded the result to posit. Our prototype is publicly available [31].

Methodology. To assess the accuracy of our CORDIC implementation with posits, we computed $sin(\theta)$ and $cos(\theta)$ for $\theta \in [0, \frac{\pi}{2}]$ using the rotation mode and $atan(\frac{y}{1})$ for $y \in [0, maxpos]$ using the vectoring mode. We measured the absolute error (maximum, minimum, and average), unit in the last place (ULP) error (maximum, minimum, and average), and the number of results with 0 ULP error for all inputs. To measure the absolute error, we computed the correct (*i.e.*, real number) result using the MPFR math library with 128 precision bits, rounded both the posit and the MPFR result back to double, and computed the absolute difference. As double has 52 faction bits, it can represent all values that are representable with $\langle 32, 2\rangle$-posits, which only has 27 precision bits. The ULP error is defined as the number of distinct posit (or floating point) values between the correctly rounded real value and the computed posit (or floating point) value. If a value has 0 ULP error, then the value is equal to the correctly rounded real value.

Table 2: Details about the accuracy of the results with (a) our CORDIC method implemented with $\langle 32, 2\rangle$-posit and (b) the naive CORDIC method implemented with 32-bit float. Each sub-divided columns provides details for $sin(\theta)$, $cos(\theta)$, and $atan(y)$. The rows show the maximum, minimum, and average absolute error, maximum, minimum, and average ULP error, the number of results with 0 ULP error, and the total number of input values. All decimal values are rounded to 2 decimal places.

	(a) our CORDIC (posit)			(b) naive CORDIC (float)		
	sin	*cos*	*atan*	*sin*	*cos*	*atan*
max abs	3.04E-8	2.96E-8	3.26E-1	4.14E-7	4.12E-7	3.26E-1
min abs	0	0	0	6.62E-24	0	0
avg abs	1.56E-9	3.74E-9	5.13E-9	2.44E-8	5.80E-9	8.84E-4
max ulp	10	1.02E6	4.38E7	1.71E9	2.87E6	8.52E8
min ulp	0	0	0	0	0	0
avg ulp	1.18	1.10	7.20E-1	1.02E9	1.25E-1	1.70E8
# 0 ulp	3.20E8	3.87E8	1.68E9	8.39E6	9.98E8	9.18E7
# input	1.15E9	1.15E9	2.15E9	1.07E9	1.07E9	2.14E9

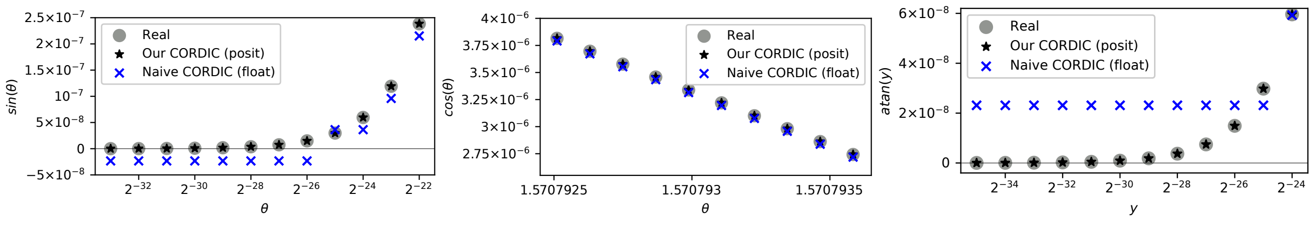
Evaluation Objective. There are two questions that we want to answer in our evaluation. First, how accurate is our CORDIC implementation with respect to the real answer (*i.e.* computed using MPFR math library) and how does it compare to the floating point implementation of CORDIC? Second, how much does each of our techniques improve the accuracy of the result compared to the naive CORDIC algorithm? Next, we describe the evaluation of our prototype to answer the above questions.

5.1 Accuracy of Our CORDIC Method for Posits

Table 2 provides details on the accuracy of our CORDIC method for posits and the naive CORDIC method implemented with a 32-bit float. In summary, our CORDIC method for posits outperforms naive CORDIC with floats for $sin(\theta)$ and $atan(\theta)$ in every dimension: (1) it has a larger number of values with zero ULP error, (2) more than 10× lower average absolute error, (3) more than 10^8× lower average ULP error, and (4) overall, more accurate results.

Our evaluation shows that even the naive $\langle 32, 2\rangle$-posit implementation of CORDIC (see Table 3(b)) is more accurate compared to the float implementation. This shows that posit is more suitable for the CORDIC method than FP.

In the domain of scientific computing where accuracy is important, our CORDIC method for posits has 1.18, 1.10, and 0.72 ULP error on average for $sin(\theta)$, $cos(\theta)$, and $atan(y)$, respectively. Our results show that our method can approximate trigonometric functions for all inputs very accurately. More importantly, our CORDIC can approximate $sin(\theta)$ accurately to within 10 ULP error for all inputs of interest. On the contrary, we observed high ULP error for $cos(\theta)$ for θ near $\frac{\pi}{2}$ and $atan(y)$ for y near $maxpos$. We conjecture that the reason for high ULP error for $cos(\theta)$ is due to the cancellation error of x_i computation because the fast-forwarded iteration



(a) $sin(\theta)$ (b) $cos(\theta)$ (c) $atan(y)$

Figure 2: Graphs showing the result of (a) $sin(\theta)$, (b) $cos(\theta)$, and (c) $atan(y)$ approximation for our CORDIC methods with $\langle 32, 2\rangle$-posit (black star) and the naive CORDIC with 32-bit floats (blue ×) for a range of inputs that output small values. The gray circle represents the correct real result computed with the MPFR library.

technique still started at iteration 0. Although the absolute error in this case is less than 3×10^{-8}, the error is relatively large compared to the real value of the result, which is near 0. The cause of high ULP error for $atan(y)$ is due to the small number of available precision bits for large values.

On average, our CORDIC implementation has an absolute error of 1.56×10^{-9}, 3.74×10^{-9}, and 5.13×10^{-9} for $sin(\theta)$, $cos(\theta)$, and $atan(y)$, respectively. It can also be observed that for $sin(\theta)$ and $cos(\theta)$, the maximum absolute error is within a factor of 10 compared to the average absolute error.

Compared to the float implementation of the CORDIC method, the absolute error of our method is lower by 10× on average for $sin(\theta)$ and $atan(y)$ while the average ULP error is 10^8× lower on average. In contrast to sin and $atan$, the float implementation of the CORDIC method for $cos(\theta)$ produces results that have comparable average absolute error and roughly 10× lower average ULP error. This discrepancy is due to the different distribution of values in the FP and the posit representation. It does not signify that the float implementation of the CORDIC method is more accurate.

In a 32-bit float, $cos(\theta) = 1.0$ for $0 \le \theta \le 10^{-4}$ and the CORDIC method produces 1.0 for these inputs as well. However, roughly 89.8% of the float values in $[0, \frac{\pi}{2}]$ are less than 10^{-4}. This means that 89.8% of the float inputs in our experiment produced $cos(\theta) = 1.0$ with 0 ULP error and minimal amount of absolute error, substantially lowering the overall error. Comparatively, only 9.7% of the posit values in $[0, \frac{\pi}{2}]$ are less than 10^{-4}.

To more accurately analyze the accuracy of our CORDIC method compared to the float implementation, we restricted the input range to $[10^{-4}, \frac{\pi}{2}]$. On average, our method has an absolute error of 4.07×10^{-9} and 1.21 ULP error while the float implementation has an absolute error of 5.29×10^{-8} and 1.14 ULP error. Thus, our CORDIC method with posits can approximate $cos(\theta)$ more accurately than the float implementation.

Figure 2 provides a comparison between our CORDIC method and the naive CORDIC with floats for a range of inputs that have small outputs. It can be observed that for all three trigonometric functions, our implementation produces a value that is much more accurate. In some cases, naive CORDIC for floats even flips the sign. The $sin(\theta)$ approximation of the float implementation outputs a negative result when the correct result should be small positive values for positive θ near zero.

Table 3: Table that shows the ULP error of $sin(\theta)$, $cos(\theta)$, and $atan(y)$ when computed with $\langle 32, 2\rangle$-posit implementation of (a) our CORDIC method with both techniques, (b) naive CORDIC method, (c) CORDIC method that only starts at a later iteration, and (d) CORDIC method implemented with quire for computing z_i. In each table, we show the maximum and average ULP error as well as the number of outputs with 0 ULP error.

	(a) Our CORDIC			(b) Naive CORDIC		
	sin	cos	$atan$	sin	cos	$atan$
max abs	3.04E-8	2.96E-8	3.26E-1	3.04E-8	2.96E-8	3.26E-1
avg abs	1.56E-9	3.74E-9	5.13E-9	2.92E-9	3.73E-9	7.68E-9
max ulp	10	1.02E6	4.38E7	7.38E6	1.02E6	4.38E7
avg ulp	1.18	1.10	7.20E-1	2.94E4	1.10	3.75E4
# 0 ulp	3.20E8	3.87E8	1.68E9	8.34E7	3.84E8	5.09E8
	(c) Fast-Forwarded Iter.			(d) Compute z_i With Quire		
	sin	cos	$atan$	sin	cos	$atan$
max abs	3.04E-8	2.96E-8	3.26E-1	3.04E-8	2.96E-8	3.26E-1
avg abs	1.56E-9	3.74E-9	7.22E-9	2.90E-9	3.73E-9	5.69E-9
max ulp	10	1.02E6	4.38E7	7.38E6	1.02E6	4.38E7
avg ulp	1.19	1.10	1.28	3.06E4	1.10	3.75E4
# 0 ulp	3.12E8	3.87E8	7.91E8	6.62E7	3.84E8	1.05E9

5.2 Sensitivity Experiments

To analyze the contribution of each of our techniques in improving the accuracy of various functions, we measured the ULP error and the absolute error produced by the $\langle 32, 2\rangle$-posit implementation of (1) our CORDIC method with both enhancements, (2) the naive CORDIC, (3) CORDIC with fast-forwarded iteration, and (4) CORDIC with quire computation. Table 3 provides details on this experiment.

The fast-forwarded iteration technique improves the average ULP error significantly for $sin(\theta)$ and $atan(y)$. Specifically, our experiments show that this technique improves the accuracy of the CORDIC implementation for small input values. The naive CORDIC implementation produced results with high ULP error when approximating $sin(\theta)$ for θ near 0 and $atan(y)$ for y near 0. The fast-forwarded iteration technique produces accurate results for $sin(\theta)$ such that there are at most 10 ULP error for all $\theta \in [0, \frac{\pi}{2}]$. In the case of $atan(y)$, fast-forwarded iteration technique produces values with at most 5 ULP error for $0 \le y \le 8.38 \times 10^6$. The fast-forwarded iteration technique contributed the most in reducing the ULP error of our CORDIC method.

The use of quires produces better results for $atan(y)$ but does not significantly improve the result for $sin(\theta)$ and $cos(\theta)$. In the case of $atan(y)$, the inaccuracy of the intermediate z_i values was attributed to the rounding error in the computation of $z_i + d_i\theta_i$. The quire prevented this rounding error and improved $atan(y)$. In contrast, the values of $sin(\theta)$ and $cos(\theta)$ are determined by x_i and y_i. The improvement in the accuracy of z_i did not affect the accuracy of x_i and y_i significantly.

Performance experiments. To test the efficiency of our CORDIC method, we developed a test harness that runs various functions with different inputs a large number of times. We measured the total time it took in cycles using hardware performance counters. Our CORDIC method is on average 5.06× faster than the MPFR version of the CORDIC method with 1024 bits of precision. The naive CORDIC method that produces inaccurate results is 6.47× faster than the MPFR version of the CORDIC method. Our use of the quire data type to improve precision is the reason for this small speedup loss. Overall, our CORDIC method provides reasonably accurate results with speedup over an MPFR execution.

6 RELATED WORK

Posits. Posit [20, 21] is a recently proposed representation as a replacement for FP. Many software implementations of posit arithmetic are available [29, 37]. The tapered accuracy with the posit representation has generated interest and many applications such as neural networks [4, 5, 13] and weather simulation [25] are using it. We have developed PositDebug [7], a debugger to detect numerical errors with posit applications that helped design the proposed math library.

CORDIC Algorithm Since the first proposal of CORDIC [44], a wide range of improvements to CORDIC have been proposed. A detailed overview of prior research on the CORDIC method is available in the survey [36]. Here, we list some notable enhancements to CORDIC.

Walther [45] proposed a generalized CORDIC algorithm that can approximate hyperbolic functions. The generalized algorithm combined with mathematical properties can be used to approximate a wide range of other elementary functions including logarithm and exponential functions. Hsiao et al. [40] further generalized CORDIC by proposing a multi-dimensional CORDIC algorithm.

Prior research has also improved various aspects of the CORDIC algorithm using various techniques including parallelization [16, 39, 42], pipelining [12] for increased throughput of the hardware implementation, angle recoding [6, 22] for increased efficiency and accuracy, and coarse-fine rotation [46] for decreasing number of computations. Among these techniques, angle recoding is the most related to our work. The goal of angle recoding is to reduce the number of matrix multiplications and z_i accumulation by not rotating for some of the θ_i. In other words, the CORDIC method is extended with $d_i \in \{-1, 0, 1\}$. Our fast-forwarded iteration technique can be interpreted similarly as not rotating the vector for the first l iterations. We specifically chose to not rotate the vector with $\theta_0, ..., \theta_{l-1}$ to be able to precompute the values of K'. If an arbitrary angle θ_i is not rotated in the CORDIC algorithm, then K cannot be precomputed and each K_i has to be multiplied in each iteration. We also provide an efficient algorithm to identify the starting iteration l such that the CORDIC algorithm is mathematically guaranteed to compute the desired θ.

Redundant CORDIC algorithm [41] presents a technique that allows an iteration to not rotate the vector and does not require individual multiplication of K_i in each iteration, thus keeping the value of K constant. Abstractly, this technique modifies the rotation matrix such that the intermediate vector accounts for the scaling factor even when the vector is not to be rotated. Thus the number of matrix multiplication operations stays constant but the number of z_i accumulation operations is reduced. Maharatna et al. propose scaling free technique [34, 35] for sufficiently small rotation angle, *i.e.*, $sin(\theta_i) \approx 2^{-i}$. This technique modifies the rotation matrix to not require the multiplication of the scaling factor. CORDIC II [15] combines both redundant CORDIC algorithm and scaling free technique to compute a set of angles that can be used to calculate trigonometric functions to within $\approx 4.88 \times 10^{-4} rad$.

Math Libraries As the posit representation is relatively recent, there is no general-purpose math library for posits, yet. There is only one posit math library available that supports a limited number of 16-bit posit elementary functions [30]. In contrast, there are a number of FP math libraries. GLIBC [17] is the most widely used math library, which supports a wide range of elementary functions. GLIBC provides implementations that balance efficiency and low computation error. Crlibm [9, 11] provides implementations that produce the correctly rounded double precision result for many elementary functions. MPFR [14], a multi-precision FP library, also provides a math library of elementary functions that produce correctly rounded multi-precision results. Metalibm [28] aims to generate an efficient implementation of elementary functions given an acceptable error bound provided by the user. In contrast to prior work, this paper explores a set of techniques to design a math library for posits using the CORDIC method.

7 CONCLUSION

Posit is a new representation for approximating real numbers. It is intended to be a drop-in replacement for the floating point representation. Posits have gained interest in many domains such as neural networks because they provide tapered accuracy. It currently lacks math libraries. We propose the use of the CORDIC method for designing math libraries for posits. We propose two extensions, fast-forwarded iterations and use of quires, to the CORDIC method that improve the accuracy of the results with posits. Our experiments indicate that posits provide better results than FP with the CORDIC method. Our CORDIC method for posits provides reasonably accurate results and can serve as a baseline for future research on math libraries for posits.

ACKNOWLEDGMENTS

We thank John Gustafson for numerous clarifications and inputs on the posit representation. This paper is based on work supported in part by NSF CAREER Award CCF–1453086, NSF Award CCF-1908798, and NSF Award CCF-1917897.

REFERENCES

[1] 1973. Lunar Roving Vehicle Navigation System Performance Review. National Aeronautics and Space Administration.

[2] 2008. *IEEE Standard for Floating-Point Arithmetic.* IEEE 754-2008. IEEE Computer Society. 1–70 pages. https://doi.org/10.1109/IEEESTD.2008.4610935

[3] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. 2017. Deep Learning with Low Precision by Half-Wave Gaussian Quantization. In *2017 IEEE Conference on Computer Vision and Pattern Recognition.* 5406–5414.

[4] 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).*

[5] Zachariah Carmichael, Hamed F. Langroudi, Char Khazanov, Jeffrey Lillie, John L. Gustafson, and Dhireesha Kudithipudi. 2019. Performance-Efficiency Trade-off of Low-Precision Numerical Formats in Deep Neural Networks. In *Proceedings of the Conference for Next Generation Arithmetic 2019 (CoNGA'19).*

[6] 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).

[7] 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).*

[8] David S. Cochran. 1972. Algorithms and Accuracy in the HP-35. *Heweltt-Packard Journal* (1972).

[9] Catherine Daramy, David Defour, Florent Dinechin, and Jean-Michel Muller. 2003. CR-LIBM: A correctly rounded elementary function library. *Proceedings of SPIE - The International Society for Optical Engineering* 5205 (12 2003). https://doi.org/10.1117/12.505591

[10] Beman Dawes and David Abrahams. 2004. *Boost C++ Libraries.* https://www.boost.org

[11] Florente de Dinechin, Alexey V. Ershov, and Nicolas Cast. 2005. Towards the post-ultimate libm. In *17th IEEE Symposium on Computer Arithmetic (ARITH '05).*

[12] E. Deprettere, P. 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.*

[13] 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).*

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

[15] M. Garrido, P. Källström, M. Kumm, and O. Gustafsson. 2016. CORDIC II: A New Improved CORDIC Algorithm. *IEEE Transactions on Circuits and Systems II: Express Briefs* (2016).

[16] Bimal Gisuthan and Thambipillai Srikanthan. 2002. Pipelining flat CORDIC based trigonometric function generators. *Microelectronics Journal* (2002).

[17] GNU. 2019. *The GNU C Library (glibc).* https://www.gnu.org/software/libc/

[18] David Goldberg. 1991. What Every Computer Scientist Should Know About Floating-point Arithmetic. *Comput. Surveys* 23, 1 (March 1991), 5–48. https://doi.org/10.1145/103162.103163

[19] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning (ICML'15).*

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

[21] 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.

[22] Yu Hen Hu and Homer H. M. Chern. 1996. A Novel Implementation of CORDIC Algorithm Using Backward Angle Recoding (BAR). *IEEE Trans. Comput.* (1996).

[23] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

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

[25] 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 (CoNGA'19).*

[26] Ulrich W. Kulisch. 2002. *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units.* Springer-Verlag, Berlin, Heidelberg.

[27] Ulrich W. Kulisch. 2013. *Computer Arithmetic and Validity: Theory, Implementation, and Applications.* De Gruyter. https://books.google.com/books?id=kFR0yMDS6d4C

[28] Olga Kupriianova and Christoph Lauter. 2014. Metalibm: A Mathematical Functions Code Generator. In *Mathematical Software – International Congress on Mathematical Software.*

[29] Cerlane Leong. 2018. *SoftPosit.* https://gitlab.com/cerlane/SoftPosit

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

[31] Jay Lim, Matan Shachnai, and Santosh NagarakatteRutgers-apl. 2020. *CORDIC for Posits.* https://github.com/rutgers-apl/CordicWithPosits

[32] Jay P. Lim. 2020. *CordicWithPosit.* https://github.com/jpl169/CordicWithPosit

[33] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. 2016. Fixed Point Quantization of Deep Convolutional Networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16).*

[34] K. Maharatna, S. Banerjee, E. Grass, M. Krstic, and A. Troya. 2005. Modified virtually scaling-free adaptive CORDIC rotator algorithm and architecture. *IEEE Transactions on Circuits and Systems for Video Technology* (2005).

[35] K. Maharatna, A. Troya, S. Banerjee, and E. Grass. 2004. Virtually scaling-free adaptive CORDIC rotator. *IEE Proceedings - Computers and Digital Techniques* (2004).

[36] P. K. Meher, J. Valls, T. Juang, K. Sridharan, and K. Maharatna. 2009. 50 Years of CORDIC: Algorithms, Architectures, and Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2009).

[37] Theodore Omtzigt. 2018. *Universal Number Arithmetic.* https://github.com/stillwater-sc/universal

[38] Rutgers-APL. 2020. *CordicWithPosits.* https://github.com/rutgers-apl/CordicWithPosits

[39] Shaoyun Wang and E. E. Swartzlander. 1995. Merged CORDIC algorithm. In *Proceedings of ISCAS'95 - International Symposium on Circuits and Systems.*

[40] Shen-Fu Hsiao and J. . Delosme. 1995. Householder CORDIC algorithms. *IEEE Trans. Comput.* (1995).

[41] N. Takagi, T. Asada, and S. Yajima. 1991. Redundant CORDIC methods with a constant scale factor for sine and cosine computation. *IEEE Trans. Comput.* (1991).

[42] 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).

[43] Jeremy M. Underwood and Bruce H. Edwards. 2019. *How Do Calculators Calculate Trigonometric Functions?* https://people.clas.ufl.edu/bruceedwards/files/paper.pdf

[44] 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)).*

[45] J. S. Walther. 1971. A Unified Algorithm for Elementary Functions. In *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference (AFIPS '71 (Spring)).*

[46] S. Wang, V. Piuri, and E. E. Wartzlander. 1997. Hybrid CORDIC algorithms. *IEEE Trans. Comput.* (1997).

A ARTIFACT APPENDIX

A.1 Abstract

Our Cordic implementation using posit [38] and the artifact [32] is open source and publicly available. We also provide our implementation and the artifact in an archival link. The artifact contains the source code and scripts to automatically run experiments and reproduce our results. To ease installation effort, a prebuilt docker image containing the required software and the artifact is also available.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Cordic Algorithm.
- **Program:** C++, Python3, and SoftPosit
- **Compilation:** g++
- **Run-time environment:** Experiments were performed on ubuntu 18.04 and confirmed to work on macOS Catalina.
- **Hardware:** Modern machines with at least 2.3GHz processors and 8GB memory should be sufficient.
- **Metrics:** The artifact contains the expected results.
- **Experiments:** Download and run docker image, run the test scripts, and observe the results.
- **How much time is needed to prepare workflow (approximately)?:** Preparation should take less than 30 minutes
- **How much time is needed to complete experiments (approximately)?:** All experiments may take more than 6 days depending on parallelization option. Therefore, we provide parallelization option as well as a shorter version of the experiments which will take roughly 1.5 hours at most.
- **Publicly available?:** Yes.

A.3 Description

A.3.1 How to access. The artifact can be downloaded from the archive at http://doi.org/10.5281/zenodo.3774064 or use the prebuilt docker image.

A.3.2 Software dependencies. Our implementation is written in C++ and uses SoftPosit library. The experiment scripts are written in Python3 and uses numpy and matplotlib. All softwares are installed in the docker image.

A.4 Installation

A.4.1 Using docker image. Install Docker by going to https://docs.docker.com/get-docker/ and selecting the installation file for the corresponding OS and follow the instructions. Then, pull the docker image and run it.

```
$ docker run −it jpl169/cordicwithposit
```

A.4.2 Manual installation with Ubuntu 18.04. To evaluate the artifact without using Docker, install required packages:

```
$ sudo apt−get update
$ sudo apt−get install −yq −−no−install−recommends apt−utils
$ sudo apt−get install −yq build−essential python3 python3−pip \
libgmp3−dev libmpfr−dev git
$ python3 −m pip install numpy matplotlib
```

Next, download and build the SoftPosit library:

```
$ git clone https://gitlab.com/cerlane/SoftPosit.git
$ cd SoftPosit/build/Linux−x86_64−GCC/
$ make
$ cd ../../..
```

Finally, untar the artifact and build the code:

```
$ export SOFTPOSITPATH=<path to SoftPosit directory>
$ tar −xvf CordicWithPosit.tar.gz
$ cd CordicWithPosit && make
```

A.5 Experiment workflow

This artifact provides scripts to automatically conduct experiments described in Section 5.

Graph Generation. This experiment creates three graphs presented in Section 5. To run the experiment, use the command:

```
$ python3 runGraphGeneration.py
```

This script generates three graphs, `sin.pdf`, `cos.pdf`, and `atan.pdf` in `graph` directory, which can be compared against the reference graph in `expected` directory. To copy pdf files (for example sin.pdf) from docker container to the host machine, use the command

```
$ exit
$ docker cp <container id>:/home/CordicWithPosit/graph/sin.pdf .
$ docker start <container id> && docker attach <container id>
```

Accuracy Experiment (Full). The full accuracy evaluation can be run using the command:

```
$ python3 runAccAnalysis.py <optional: # of parallelization>
$ python3 GenerateTableForAcc.py
```

By default, the first python script without parallelization argument runs 4 experiments in parallel. You can use the argument to specify how many experiments to run in parallel. The script runs a total of 10 individual experiments which can take up to 20 hours each. Without parallelization, these experiments can take up to 6 days. This experiment automatically compares the output (`table/table2table3.txt`) to the reference result (`expected/table2table3.txt`) and checks whether they are the same.

Accuracy Experiment (Fast). Instead of the full accuracy evaluation script above, you can run a simplified evaluation script that uses $0.01\times$ of the total input space. To run this experiment, use the command:

```
$ python3 runSimplifiedAccAnalysis.py <optional: # of parallelization>
$ python3 GenerateTableForSimplifiedAcc.py
```

This experiment should take 1 to 1.5 hours even without parallization. The script automatically compares the output (`table/simpleTable2table3.txt`) to the expected result (`expected/simpleTable2table3.txt`) and checks whether they are exactly the same or not in the terminal.

A.6 Evaluation and expected result

The generated graphs should be compared against the the graphs found in `expected` directory with the same file name. The accuracy experiment automatically compares the result against the expected result.

A.7 Experiment customization

Our Cordic implementation is built as a static library which can be found in `lib/lib_cordic.a` and the header file can be found in `include/cordic.h`. We have provided an example program in the `example` directory. Use the following command to test the example:

```
$ cd example && make && ./example
```