

# Quantum CORDIC — Arcsin on a Budget

Iain Burge

SAMOVAR, Télécom SudParis,  
Palaiseau, France

iain-james.burge@telecom-sudparis.eu

Michel Barbeau

Carleton University, School of Computer  
Science, Ottawa, Canada

barbeau@scs.carleton.ca

Joaquin Garcia-Alfaro

SAMOVAR, Télécom SudParis,  
Palaiseau, France

joaquin.garcia\_alfaro@telecom-sudparis.eu

**Abstract**—This work introduces a quantum algorithm for computing the arcsine function to an arbitrary accuracy. We leverage a technique from embedded computing and field-programmable gate array (FPGA), called COordinate Rotation Digital Computer (CORDIC). CORDIC is a family of iterative algorithms that, in a classical context, can approximate various trigonometric, hyperbolic, and elementary functions using only bit shifts and additions. Adapting CORDIC to the quantum context is non-trivial, as the algorithm traditionally uses several non-reversible operations. We detail a method for CORDIC which avoids such non-reversible operations. We propose multiple approaches to calculate the arcsine function reversibly with CORDIC. For  $n$  bits of precision, our method has space complexity of order  $n$  qubits, a layer count in the order of  $n$  times  $\log n$ , and a CNOT count in the order of  $n$  squared. This primitive function is a required step for the Harrow–Hassidim–Lloyd (HHL) algorithm, is necessary for quantum digital-to-analog conversion, can simplify a quantum speed-up for Monte-Carlo methods, and has direct applications in the quantum estimation of Shapley values.

## I. INTRODUCTION

The problem of quantum digital-to-amplitude conversion [7] is an essential step in various quantum algorithms, including Harrow–Hassidim–Lloyd (HHL) — a quantum algorithm for solving linear equations [5], a quantum speed-up for Monte Carlo methods [8], and a quantum algorithm for Shapley value estimation [1]. However, to efficiently perform this conversion it is necessary to calculate inverse trigonometric functions, which is computationally expensive [7]. As quantum computation is very young, it is valuable to consider techniques used in early classical computing intended for weak hardware. In particular, this work adapts the classical CORDIC algorithm for  $\arcsin$  to the quantum setting.

Section II presents related work. Sections III and IV provide preliminaries to our problem, and introduces the approach of classical CORDIC to approximate  $\arcsin$  on minimal hardware. Section V translates the approach to a quantum setting. Section VI demonstrates the use of quantum CORDIC to perform a quantum digital-to-analogue transformation. Section VII gives a sketch proof of the time and space complexity of quantum CORDIC and provides simulation results. Section VIII concludes the work.

## II. RELATED WORK

There have been a few attempts to design a quantum algorithm for the  $\arcsin$  function. For instance, Häner et al. [4] leverages quantum parallelism to perform piecewise polynomial approximations of a wide range of functions. While the approach is flexible, it is not ideal in the short term, since it requires some memory access, a fair number of multiplications,

and square rooting for the extreme portions of  $\arcsin x$ , i.e.,  $x \in [-1, -0.5] \cup [0.5, 1]$ . It can be seen as a promising solution that scales well in precise situations, e.g., more than 32 bits. There also exists an iterative method [12], which performs a binary search to approximate  $\arcsin x$ , among other elementary functions. The suggested solution is very general. However, it requires many squaring operations that are challenging to implement in the near term.

The above techniques are valuable for various applications when sufficient hardware is available. In the late 1950s, Volder [10] introduced the CORDIC algorithm to solve an important family of trigonometric equations using the under-powered hardware of the time. This was later generalized to a single algorithm that solves various elementary functions, including multiplication, division, the trigonometric functions,  $\arctan$ , the hyperbolic trigonometric functions,  $\ln$ ,  $\exp$ , and square root [11]. CORDIC has been further modified to solve various other functions, including our target function,  $\arcsin$  [6].

## III. PRELIMINARIES

Suppose you have the following  $(n + 1)$ -qubit quantum state:

$$|\phi\rangle = \sum_{k=0}^{L-1} \alpha_k |h_k\rangle_{\text{in}} |0\rangle_{\text{out}},$$

where  $L \leq 2^n$ , the  $\text{in}$  register has size  $n$ , the  $\text{out}$  register is of size one, and  $h_k$  represents a number in range  $[0, 1]$  as an  $n$ -bit binary string. A particularly useful transformation on this state is given by the following equation:

$$U |\phi\rangle = \sum_{k=0}^{L-1} \alpha_k |h_k\rangle_{\text{in}} \left( \sqrt{1 - h_k} |0\rangle + \sqrt{h_k} |1\rangle \right)_{\text{out}}. \quad (1)$$

which encodes the binary values  $h_k$  into the probability amplitudes of the output register. This transformation is an important instance of quantum digital-to-analog conversion [7].

A fast solution to this problem is valuable in multiple algorithms. For instance, a quantum algorithm for approximating Shapley values [1] could leverage this transformation to simplify the state preparation step. Another important example is the quantum speedup of Monte Carlo methods [8] which requires the transformation  $W$ , defined as:

$$|x\rangle_{\text{in}} |0\rangle_{\text{out}} \xrightarrow{W} |x\rangle_{\text{in}} \left( \sqrt{1 - \Phi(x)} |0\rangle + \sqrt{\Phi(x)} |1\rangle \right)_{\text{out}}.$$

where  $\Phi$  is a function from binary strings to the interval from zero to one. This can be translated easily to our transformation.

We split the computation into two steps, where we first implement a reversible algorithm to compute  $\Phi$  in the computational basis, and then apply our transformation. Take register `in` to have  $m$  qubits, the register `aux` to have  $n$  qubits, and the register `out` to have one qubit. We first apply the unitary  $V$  using the `in` register as input and outputting the  $n$ -bit approximate result  $\tilde{\Phi}$  to `aux`, as follows:

$$|x\rangle_{\text{in}} |0\rangle_{\text{aux}} \xrightarrow{V} |x\rangle_{\text{in}} |\tilde{\Phi}(x)\rangle_{\text{aux}}.$$

where  $|x\rangle_{\text{in}}$  is a computational basis vector. Then,  $W$  can be constructed with  $n$  auxiliary qubits by composing  $V$ , and  $U$ , Equation (1), where the `aux` acts as the input. Finally, to restore the `aux` register to its initial state, we apply the  $V^{-1}$  transformation. In summary, we can solve the problem using a reversible classical algorithm implementation of  $\tilde{\Phi}$  followed by encoding the result into the probability amplitude of the output bit.

Another practical use case is as a step of the HHL algorithm [5]. In this case, reciprocal eigenvalues are encoded as binary strings in a superposition. An essential step is to encode the reciprocals of the eigenvalues in the probability amplitudes of an output bit. In particular, we can apply a transformation closely related to  $U$  from Equation (1).

Hence, an efficient algorithm to implement the transformation  $U$ , Equation (1), has immediate benefits to foundational problems. There are a few naive approaches to implementing the transformation. One is to use a lookup table, where each possible input  $|h_k\rangle_{\text{in}}$  is separately implemented, but that would require exponential circuit depth. qRAM could also be leveraged [3], however it would require exponential space.

With some naive methods out of the way, we examine a less trivial direction that illustrates the main challenge solved by this paper. Consider the following state,

$$|\psi_0\rangle = \sum_{k=0}^{L-1} \alpha_k |h_k\rangle_{\text{in}} |0\rangle_{\text{aux}} |0\rangle_{\text{out}}.$$

where `in` and `aux` are  $n$ -qubit registers, each representing two's complement, fixed point integers in the range  $[-2, 2)$  of the form  $x_0x_1x_2 \dots x_{n-1}$  where  $x_0$  is the sign bit, and `out` is a single qubit register. We require  $h_k$  to be in range  $[0, 1]$ . Suppose we apply the function  $\arcsin$  where `in` is the input register, and `aux` is the output register. Applying this unitary, which we call  $F$ , yields,

$$F|\psi_0\rangle = |\psi_1\rangle = \sum_{k=0}^{L-1} \alpha_k |h_k\rangle_{\text{in}} |\arcsin \sqrt{h_k}\rangle_{\text{aux}} |0\rangle_{\text{out}}.$$

Suppose now that we apply the quantum circuit  $R$  from Figure 1. This would give us state,

$$R|\psi_1\rangle = |\psi_2\rangle = \sum_{k=0}^{L-1} \alpha_k |h_k\rangle_{\text{in}} |\arcsin \sqrt{h_k}\rangle_{\text{aux}} \cdot \left( \cos \left( \arcsin \sqrt{h_k} \right) |0\rangle + \sin \left( \arcsin \sqrt{h_k} \right) |1\rangle \right)_{\text{out}}.$$

Using trigonometry identities, we have that,

$$|\psi_2\rangle = \sum_{k=0}^{L-1} \alpha_k |h_k\rangle_{\text{in}} |\arcsin \sqrt{h_k}\rangle_{\text{aux}} \cdot \left( \sqrt{1-h_k} |0\rangle + \sqrt{h_k} |1\rangle \right)_{\text{out}}.$$

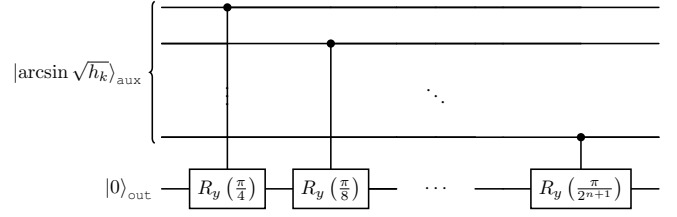


Fig. 1: Unitary transformation  $R$  which encodes the binary value of the `aux` register into the output register. Define  $R_y(\omega) = (\cos \omega, -\sin \omega; \sin \omega, \cos \omega)$ .

Uncomputing  $\arcsin h_k$  by performing  $F^{-1}$ , we get our desired state, Equation (1). This approach transforms our previous problem into a problem of seemingly similar difficulty, computing  $\arcsin$  reversibly. However, this paper shows a method to achieve this task efficiently.

#### IV. CLASSICAL CORDIC FOR ARCSINE

Let us give a brief insight into how CORDIC works. First, there is an implicit two-dimensional goal vector that encodes the problem. The exact vector is unknown, but we have a method to compare it to other vectors. A starting vector rotates towards the goal in increasingly small discrete steps, picking the direction of rotation based on our comparison method, until it converges with satisfactory error. It is possible to avoid performing any multiplications during the rotations by using pseudo-rotations, rotations with a small stretch.

Our proposed method is based on Mazenc et al.'s algorithm [6] (but Step 1 is slightly modified, as there was an error in the original paper). The procedure is fully described in Algorithm 1. Each iteration follows the steps below:

With initial values:  $\theta_0 = 0, x_0 = 1, y_0 = 0, t_0 = t$ . The algorithm iteratively repeats the following steps, let  $i$  be the index of the current iteration,

- 1) Define  $s(z) = 1$  if  $z < 0$  and equal to 0 otherwise,

$$d_i = [s(x_i) \wedge s(t_i - y_i)] \oplus s(x_i) \oplus [s(x_i) \wedge s(y_i)] \oplus s(t_i - y_i).$$

- 2) If  $d_i = 1$ , then swap  $x_i$  and  $y_i$ .
- 3) Perform pseudo-rotation:

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} 1 & -2^{-i} \\ 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$

- 4) If  $d_i = 1$ , then unswap  $x_i$  and  $y_i$ .
- 5) Compensate for pseudo-rotation:

$$t_{i+1} = (1 + 2^{-2i})t_i$$

- 6) Update theta approximation:

$$\theta_{i+1} = \theta_i + (-1)^{d_i} 2 \arctan 2^{-i}.$$

We first initialize the problem, given input  $t \in [-1, 1]$ . Our goal is to get the vector  $(x_i, y_i)$  to point in the same direction as the vector of length 1 with height  $t$ , i.e., vector  $(\sqrt{1-t^2}, t)$ . This goal vector has an angle of  $\theta = \arcsin(t)$ . By keeping track of each rotation we make, assuming  $(x_i, y_i)$  points in roughly the same direction as the goal vector, we can construct an approximation for  $\theta$ . In particular, assuming infinite precision, as  $i$  tends to infinity, approximation  $\theta_i$  tends

**Algorithm 1** CORDIC Arcsine

---

```

1: procedure ARCSIN( $t, n$ )  $\triangleright t \in [-1, 1]$ 
2:    $\text{ang} \leftarrow 0, x \leftarrow 1, y \leftarrow 0, t \leftarrow t, d \leftarrow 0$ 
3:   for  $i = 1, i < n, ++i$  do
4:      $d_i \leftarrow [s(x_i) \wedge s(t_i - y_i)] \oplus s(x_i) \oplus [s(x_i) \wedge s(y_i)] \oplus s(t_i - y_i)$ 
5:     if  $d_i$  then  $x, y \leftarrow y, x$ 
6:     for  $\_$  in range(2) do
7:        $x, y \leftarrow x - 2^{-i}y, y + 2^{-i}x$ 
8:     if  $d_i$  then  $x, y \leftarrow y, x$ 
9:      $t \leftarrow (1 + 2^{-2i})t$ 
10:     $\text{ang} \leftarrow \text{ang} + (-1)^{d_i} 2 \arctan 2^{-i}$ 
11:   return  $\text{ang}$ 

```

---

**Algorithm 2** Mult( $\text{in}, \text{aux}, m$ ):  $\text{in} \leftarrow (1 + 2^{-m})\text{in}$ 


---

```

1: procedure MULT( $\text{in}, \text{aux}, m$ )  $\triangleright \text{in}$  is a register
  of size  $n$ ,  $\text{aux}$  is an auxiliary register of size  $n$  with near
  0 value,  $m$  is the right shift
2:    $F \leftarrow [1, 1, 2, 3, 5, 8, 13, \dots]$   $\triangleright$  Fibonacci Sequence
3:    $\# \text{iter} \leftarrow \lceil \sqrt{5}\varphi^{-m}n \rceil$   $\triangleright \varphi = (1 + \sqrt{5})/2$ 
4:    $\text{aux} \leftarrow \text{aux} + \text{in}$ 
5:   for  $i = \# \text{iter}, i \geq 0, --i$  do
6:     if  $i$  even then
7:        $\text{in} \leftarrow \text{in} - (-1)^{F[i]}(\text{aux} \gg (m \cdot F[i]))$ 
8:     else
9:        $\text{aux} \leftarrow \text{aux} - (-1)^{F[i]}(\text{in} \gg (m \cdot F[i]))$ 
10:   $\text{aux} \leftarrow \text{aux} - \text{in}$ 

```

---

**Algorithm 3** Div( $\text{in}, \text{aux}, m$ ):  $\text{in} \leftarrow (1 + 2^{-m})^{-1}\text{in}$ 


---

```

1: procedure DIV( $\text{in}, \text{aux}, m$ )  $\triangleright \text{in}$  is a register
  of size  $n$ ,  $\text{aux}$  is an auxiliary register of size  $n$  with near
  0 value,  $m$  is the right shift
2:    $F \leftarrow [1, 1, 2, 3, 5, 8, 13, \dots]$   $\triangleright$  Fibonacci Sequence
3:    $\# \text{iter} \leftarrow \lceil \sqrt{5}\varphi^{-m}n \rceil$   $\triangleright \varphi = (1 + \sqrt{5})/2$ 
4:    $\text{aux} \leftarrow \text{aux} + \text{in}$ 
5:   for  $i = 0, i < \# \text{iter}, ++i$  do
6:     if  $i$  even then
7:        $\text{aux} \leftarrow \text{aux} + (-1)^{F[i]}(\text{in} \gg (m \cdot F[i]))$ 
8:     else
9:        $\text{in} \leftarrow \text{in} + (-1)^{F[i]}(\text{aux} \gg (m \cdot F[i]))$ 
10:   $\text{aux} \leftarrow \text{aux} - \text{in}$ 

```

---

to  $\theta$ . According to Ref. [6], to achieve  $n$  bits of accuracy it takes  $n + \mathcal{O}(1)$  iterations.

Step 1 is a computationally convenient way to check if  $\theta_i < \theta$ , which we store in variable  $d_i$ . If  $d_i = 0$ , then we need to rotate counterclockwise to get closer to  $\theta$ . Otherwise, if  $d_i = 1$ , we need to rotate clockwise. Steps 2 and 4 conjugate Step 3 to achieve the correct rotation direction.

Step 3 performs a pseudo-rotation, slightly stretching our vector  $(x_i, y_i)$  by a factor of  $1 + 2^{-2i}$ . It is performed simply by making the following substitutions twice:  $x' = x - 2^{-i}y$  and  $y' = y + 2^{-i}x$ . As a result of the operation, we point closer to the goal direction, but it is necessary to compensate for the stretch.

Step 5 deals with the stretching of the vector  $(x_i, y_i)$  by stretching our goal vector by the same amount,  $(1 + 2^{-2i})$ . Note that multiplication by powers of 2 can be done efficiently through bit-shifting.

Finally, Step 6 records the change to the current angle of vector  $(x_i, y_i)$ . Note that the expression  $2 \arctan 2^{-i}$  can be precomputed and directly encoded into hardware.

The big takeaway is that each iteration uses only about five bitshifts and seven additions, regardless of the number of bits.

## V. QUANTUM CORDIC FOR ARCSINE

There are a few immediate challenges that appear when we try and move CORDIC to a quantum setting, in this section we detail the solutions step-by-step. Before going through the steps, we have a few important facts to note. We represent the values  $\theta_i, x_i, y_i, t_i$  using fixed point two's complement in the range  $[-2, 2)$  with one sign bit, one integer bit, and  $n - 2$  fractional bits. This avoids any issues with overflow, in the case of  $\theta_i$ ,  $\theta$  is in the range  $[-\pi/2, \pi/2)$ , so  $|\theta| \leq \pi/2 < 2$ . In the case of  $|x_i|, |y_i|, |t_i|$ , each represents side-lengths of vectors with a magnitude that increases in size by  $(1 + 4^{-i})$  each step, thus,

$$|x_i|, |y_i|, |t_i| \leq \prod_{i=1}^n (1 + 4^{-i}) < \sqrt[3]{e} < 2.$$

You can verify the inequality by taking  $n \rightarrow \infty$ , taking the  $\ln$  of the infinite product, using the inequality  $\ln(1 + x) < x$  for positive  $x$ , and finally using the geometric series. The product represents the stretch caused by the pseudo-rotations over the iterations. The two's complement also allows us to check if a value is negative easily using the most significant bit, if it is 1 then the value is negative, otherwise, the value is positive. To approximate multiplication by  $2^{-m}$ , we will use the right bit-shift operation,  $x_0x_1 \cdots x_{n-1} \gg m$ , we move each bit to

the right by  $m$  places, deleting the  $m$  rightmost bits. Also note that, when right bit-shifting in two's complement, the previous leftmost bit is copied into the new leftmost bits, for example:

$$x_0x_1x_2x_3x_4 \gg 2 = x_0x_0x_0x_1x_2, \quad x_k \in \{0, 1\}.$$

This ensures right bitshifts work as a division by powers of two for negative numbers.

Now, let us go through the steps of the quantum implementation, assuming  $n$  bits each to represent  $\theta_i, x_i, y_i, t_i$  as fixed point numbers in registers  $\text{ang}, x, y, t$  respectively. Allocate  $n - 1$  bits to represent  $d = d_1d_2 \cdots d_{n-1}$  as a bit array in register  $d$  where  $d_i$  represents the  $i$ th register qubit. Initialization is trivial, we use  $t \leftarrow t$  as the input. Note that we do *not* need to use new registers for each iteration, one register per value is sufficient.

For Step 1, we can perform this step with the following sequence of operations, where the leftmost bit is most significant. We begin with the state:

$$|0\rangle_{d_i} |x_i\rangle_x |y_i\rangle_y |t_i\rangle_t$$

We perform a subtraction (implemented with an inverse addition gate) from  $y$  to  $t$ , yielding,

$$|0\rangle_{d_i} |x_i\rangle_x |y_i\rangle_y |t_i - y_i\rangle_t.$$

Looking directly at the sign bits of the  $x$ ,  $y$ , and  $t$  registers gives  $s(x_i)$ ,  $s(y_i)$ , and  $s(t_i - y_i)$  respectively, where  $s(z) = 1$  if  $z < 0$  else  $s(z) = 0$ . We next perform the following operations,

- Perform a `Toffoli` gate using the sign bits of the  $x$  and  $t$  registers as controls and qubit  $d_i$  as the target.
- Perform a `Toffoli` gate using the sign bits of the  $x$  and  $y$  registers as controls and qubit  $d_i$  as the target.
- Perform a `CNOT` gate using the sign bits of the  $x$  register as control and qubit  $d_i$  as the target.
- Perform a `CNOT` gate using the sign bits of the  $t$  register as control and qubit  $d_i$  as the target.

Finally, we can undo the previous subtraction by adding the  $y$  register back to the  $t$  register. This gives us a correct  $d_i$  bit without side effects.

Steps 2 and 4 are trivial to implement using swap gates between each bit of the  $x$  and  $y$  register controlled by  $d_i$ , this can be done using  $18n$  `CNOTs`.

Step 3 is the most difficult. It leverages Algorithm 2 for multiplying by  $(1 + 2^{-k})$ . Note, that each step of Algorithm 2

is reversible. An explanation for `Mult` is the subject of Appendix A. For notational simplicity, we assume `x` and `y` have not been swapped Our initial state for this step is,

$$|x_i\rangle_x |y_i\rangle_y.$$

We first subtract  $y_i \gg i$  ( $y_i$  right shifted by  $i$  bits) from register `x`, giving us,

$$|x_i - 2^{-i}y_i\rangle_x |y_i\rangle_y.$$

Next, we need to take a mostly clean (value near 0) auxiliary register. We apply `Mult(y, aux, 2i)` to the `y` register,

$$|x_i - 2^{-i}y_i\rangle_x |y_i + 2^{-2i}y_i\rangle_y.$$

Finally, we add the `x` register shifted,  $(x_i - 2^{-i}y_i) \gg i$ , to the `y` register,

$$|x_i - 2^{-i}y_i\rangle_x |y_i + 2^{-2i}y_i + 2^{-i}(x_i - 2^{-i}y_i)\rangle_y.$$

Which equals  $|x_i - 2^{-i}y_i\rangle_x |y_i + 2^{-i}x_i\rangle_y$ . By repeating these operations once more, Step 4 is accomplished.

With our new `Mult` tool, Step 5 is trivial. We simply apply `Mult(t, aux, 2i)` to the `t` register.

Finally, for Step 6 we apply controlled negation of the `ang` register using `di` as the control. Then add the precomputed  $2 \arctan 2^{-i}$  to the  $\theta$  register, note, this operation can be encoded directly on quantum hardware. Then, once again, perform a controlled negation of the `ang` register using `di` as the control. This gives,

$$\begin{aligned} |\theta_i\rangle_{\text{ang}} &\rightarrow |(-1)^{d_i}\theta_i\rangle_{\text{ang}} \rightarrow |(-1)^{d_i}\theta_i + 2 \arctan 2^{-i}\rangle_{\text{ang}} \\ &\rightarrow |(-1)^{2d_i}\theta_i + (-1)^{d_i}2 \arctan 2^{-i}\rangle_{\text{ang}} \\ &= |\theta_i + (-1)^{d_i}2 \arctan 2^{-i}\rangle_{\text{ang}} = |\theta_{i+1}\rangle_{\text{ang}}. \end{aligned}$$

This step can be improved slightly by using bitwise NOT on the `ang` register instead of negation.

Thus, we have a quantum-compatible method for applying each step of the classical CORDIC algorithms. As we see in Section VII, these adapted techniques add minimal error in simulation.

## VI. BINARY TO AMPLITUDE TRANSFORMATION

Suppose you are given a  $5n$ -qubit quantum state of the form,

$$|\kappa_0\rangle = \sum_{k=0}^{L-1} \alpha_k |h_k\rangle_t |0\rangle_d^{\otimes n-1} |0\rangle_x^{\otimes n} |0\rangle_y^{\otimes n} |0\rangle_{\text{mult}}^{\otimes n} |0\rangle_{\text{out}},$$

where  $L \leq N = 2^n$ , the `t` register has size  $n$ , and  $h_k \in \{0, 1\}^n$  is an  $n$ -bit binary string representing a value from the set  $\{2^{-n+1}j : j \in \mathbb{Z}_{2^{n-1}}\}$ . In other words, register `t` stores a fixed point binary number between  $-2$  and  $2$  in two's complement, but we restrict the initial value to be between 0 and 1. The registers `x`, `y`, `mult` are of size  $n$ , register `d` is of size  $n-1$ , and each is initialized to  $|0\rangle$ . Finally, the `out` register is one qubit, and will store a value corresponding to  $h_k$  in its probability amplitudes.

**Remark VI.1.**  $\arcsin(\sqrt{x})$  is an affine transformation of  $\arcsin(x)$ ,

$$\arcsin(\sqrt{x}) = \frac{\arcsin(2x-1)}{2} + \frac{\pi}{4}.$$

We would like to find each  $d_i$ , we can do this by performing the quantum implementation of the CORDIC Arcsine, with a couple modifications. To leverage Remark VI.1, we first multiply register `t` by two and subtract one. Next, we set the `x` register equal to 1. This gives us state,

$$|\kappa_1\rangle = \sum_{k=0}^{L-1} \alpha_k |h'_k\rangle_t |0\rangle_d^{\otimes n} |1\rangle_x |0\rangle_y^{\otimes n} |0\rangle_{\text{mult}}^{\otimes n} |0\rangle_{\text{out}},$$

where  $h'_k = 2h_k - 1 \in [-1, 1]$ .

We then apply Algorithm 1, skipping Line 2, since the registers are already initialized, and skipping Line 10, as we do not require an explicit representation of theta. This yields the state,

$$|\kappa_2\rangle = \sum_{k=0}^{L-1} \alpha_k |h'_k\rangle_t |d^{(h'_k)}\rangle_d |x'\rangle_x |y'\rangle_y |g\rangle_{\text{mult}} |0\rangle_{\text{out}}. \quad (2)$$

The  $j$ th bit of  $d^{(h'_k)}$  represents the rotation direction at iteration  $j$  of the algorithm.  $x'$ ,  $y'$ , and  $g$  are unnecessary garbage for our application.

Performing the inverse of Algorithm 1, where Lines 2, 4, and 10 are skipped, gives the state,

$$|\kappa_3\rangle = \sum_{k=0}^{L-1} \alpha_k |h'_k\rangle_t |d^{(h'_k)}\rangle_d |0\rangle_x^{\otimes n} |0\rangle_y^{\otimes n} |0\rangle_{\text{mult}}^{\otimes n} |0\rangle_{\text{out}}.$$

**Remark VI.2.** By Mazenc [6], we have the following relation,

$$\frac{\arcsin(h'_k)}{2} \approx \sum_{j=0}^n (-1)^{d_j^{(h'_k)}} \arctan(2^{-j}),$$

with error of order  $\mathcal{O}(2^{-n})$ .

Thus, applying the transformation,

$$\begin{aligned} R' |d^{(h'_k)}\rangle_d |0\rangle_{\text{out}} &= |d^{(h'_k)}\rangle_d \left[ \cos\left(\frac{\pi}{4} + \sum_{j=0}^n (-1)^{d_j^{(h'_k)}} \arctan(2^{-j})\right) |0\rangle \right. \\ &\quad \left. + \sin\left(\frac{\pi}{4} + \sum_{j=0}^n (-1)^{d_j^{(h'_k)}} \arctan(2^{-j})\right) |1\rangle \right]_{\text{out}}, \end{aligned}$$

implemented in Figure 2, by Remark VI.2, gives the state,

$$\begin{aligned} |\kappa_4\rangle &\approx \sum_{k=0}^{L-1} \alpha_k |h'_k\rangle_t |d^{(h'_k)}\rangle_d |0\rangle_x^{\otimes n} |0\rangle_y^{\otimes n} |0\rangle_{\text{mult}}^{\otimes n} \\ &\quad \otimes \left[ \cos\left(\frac{\pi}{4} + \frac{\arcsin(2h_k-1)}{2}\right) |0\rangle \right. \\ &\quad \left. + \sin\left(\frac{\pi}{4} + \frac{\arcsin(2h_k-1)}{2}\right) |1\rangle \right]_{\text{out}}. \end{aligned}$$

By Remark VI.1, this is equal to state,

$$\sum_{k=0}^{L-1} \alpha_k |h'_k\rangle_t |d^{(h'_k)}\rangle_d |0\rangle_x^{\otimes n} |0\rangle_y^{\otimes n} |0\rangle_{\text{mult}}^{\otimes n} \left[ \sqrt{1-h_k} |0\rangle + \sqrt{h_k} |1\rangle \right]_{\text{out}}.$$

Finally, if required, the register `d` can be cleaned up by reconstructing  $x', y', g$ . Then running the inverse of Algorithm 1 where Lines 2 and 10 are skipped. After adding one to and dividing register `t` by two and subtracting one from `x`, the binary to amplitude transformation is complete.

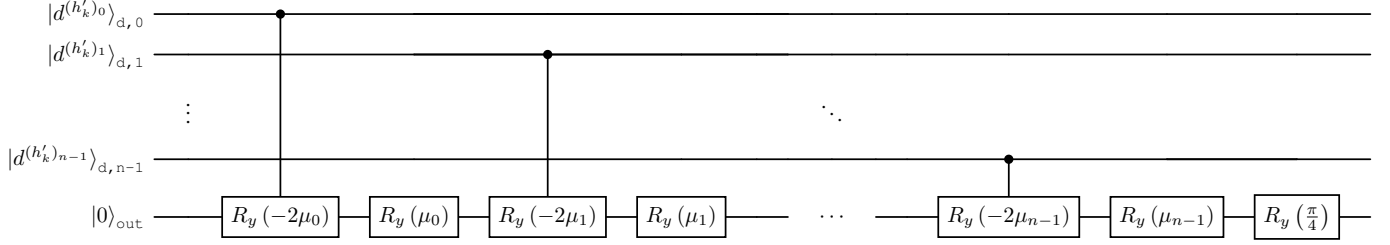


Fig. 2: Unitary transformation  $R'$  which transfers the binary encoding of CORDIC rotation direction stored  $d$  register into the amplitude of the  $out$  register. Define  $R_y(\omega) = (\cos \omega, -\sin \omega; \sin \omega, \cos \omega)$ , and  $\mu_i = \tan^{-1} 2^{-i}$ .

## VII. COMPLEXITY AND SIMULATIONS

In this section, we detail the complexity to construct  $|\kappa_2\rangle$ , Equation (2). The quantum implementation of Algorithm 1, skipping Line 10, applies `Mult` three times per iteration, and addition six times per iteration. `Mult` takes less than  $\sqrt{5}\varphi^{-2i}n + 2$  additions on the  $i$ th step if  $i < n$ , where  $\varphi$  is the golden ratio, if  $i \geq n$ , then `Mult` can be skipped. Thus, assuming  $n$  iterations for  $\mathcal{O}(n)$  bits of accuracy, we perform less than,

$$3 \sum_{i=1}^{n/2} \left( \frac{\sqrt{5}n}{\varphi^{2i}} + 2 \right) + \sum_{i=1}^n 6 = 9n + 3\sqrt{5}n \sum_{i=1}^{n/2} \frac{1}{\varphi^{2i}} < 14n,$$

additions. With  $\mathcal{O}(n)$  auxiliary qubits, it is possible to compute addition in  $\mathcal{O}(\log n)$  layers and  $\mathcal{O}(n)$  CNOTs [9]. Thus, the total layer complexity is  $\mathcal{O}(n \log(n))$  and the total CNOT complexity is  $\mathcal{O}(n^2)$  where  $n$  corresponds to bits of precision. Note that, if  $n$  is not large, it is better to use a non-parallel version of addition to minimize overhead.

As for space complexity, we need  $\mathcal{O}(n)$  bits of space. We require an auxiliary register of size  $n$  for `Mult`, and optionally another register to speed up addition. We need registers of  $n$  bits for  $x$ ,  $y$  and  $t$ , and optionally  $ang$ . Finally, we need  $n - 1$  bits for register  $d$ . Thus, in total, we require a minimum of  $5n - 1$  qubits.

In simulations, it was found that the max error for any given input can be made arbitrarily small by making  $n$  larger (Figure 3). The full codebase can be found on our GitHub companion repository [2], containing a complete implementation using Qiskit as well as an identical classical implementation.

## VIII. CONCLUSION

We have introduced a Quantum algorithm for calculating  $\arcsin$  with low space and time complexity. It also has applications in HHL (Harrow–Hassidim–Lloyd), approximating Shapley values, and quantum Monte Carlo methods. As CORDIC is most appropriate for lower precision applications, future work could apply more budget conscious implementations of CORDIC  $\arcsin$  at the expense of asymptotic behavior. Further work could also develop a full CORDIC module to cheaply perform a verity of elementary functions critical for quantum computation beyond quantum digital-to-analog conversion.

## REFERENCES

- [1] Iain Burge, Michel Barbeau, and Joaquin Garcia-Alfaro. Quantum algorithms for shapley value calculation. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 1, pages 1–9. IEEE, 2023.
- [2] Iain Burge, Michel Barbeau, and Joaquin Garcia-Alfaro. Quantum cordic algorithm implementation, companion github repository, Oct 2024. <https://github.com/iain-burge/QuantumCORDIC>.
- [3] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. Quantum random access memory. *Physical review letters*, 100(16):160501, 2008.
- [4] Thomas Häner, Martin Roetteler, and Krysta M Svore. Optimizing quantum circuits for arithmetic. *arXiv preprint arXiv:1805.12445*, 2018.
- [5] Aram W Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical review letters*, 103(15):150502, 2009.
- [6] Christophe Mazenc, Xavier Merriem, and J-M Muller. Computing functions  $\cos/\sup-1$  and  $\sin/\sup-1$  using cordic. *IEEE Transactions on Computers*, 42(1):118–122, 1993.
- [7] Kosuke Mitarai, Masahiro Kitagawa, and Keisuke Fujii. Quantum analog-digital conversion. *Physical Review A*, 99(1):012301, 2019.
- [8] Ashley Montanaro. Quantum speedup of monte carlo methods. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 471(2181):20150301, 2015.
- [9] Yasuhiro Takahashi, Seiichiro Tani, and Noboru Kunihiro. Quantum addition circuits and unbounded fan-out. *arXiv preprint arXiv:0910.2530*, 2009.
- [10] Jack E Volder. The cordic trigonometric computing technique. *IRE Transactions on electronic computers*, (3):330–334, 1959.
- [11] John Stephen Walther. A unified algorithm for elementary functions. In *Proceedings of the May 18-20, 1971, spring joint computer conference*, pages 379–385, 1971.
- [12] Shengbin Wang, Zhimin Wang, Wendong Li, Lixin Fan, Guolong Cui, Zhiqiang Wei, and Yongjian Gu. Quantum circuits design for evaluating transcendental functions based on a function-value binary expansion method. *Quantum Information Processing*, 19:1–31, 2020.

## APPENDIX

### A. MULT FUNCTION

To understand the `Mult` algorithm, it is most simple to consider its inverse, `Div` (Algorithm 3). To simplify the following proofs, we assume infinite bits of precision in our number representations.

**Lemma A.1.** *Suppose  $i$  is even, at the beginning of the  $i$ th iteration of Algorithm 3’s for loop, the state is,*

$$i_n = z \sum_{k=0}^{F[i+1]-1} r^k, \quad aux = z \sum_{k=0}^{F[i]-1} r^k.$$

where  $z$  is the initial value of  $i_n$ , and  $r = -2^{-m}$ .

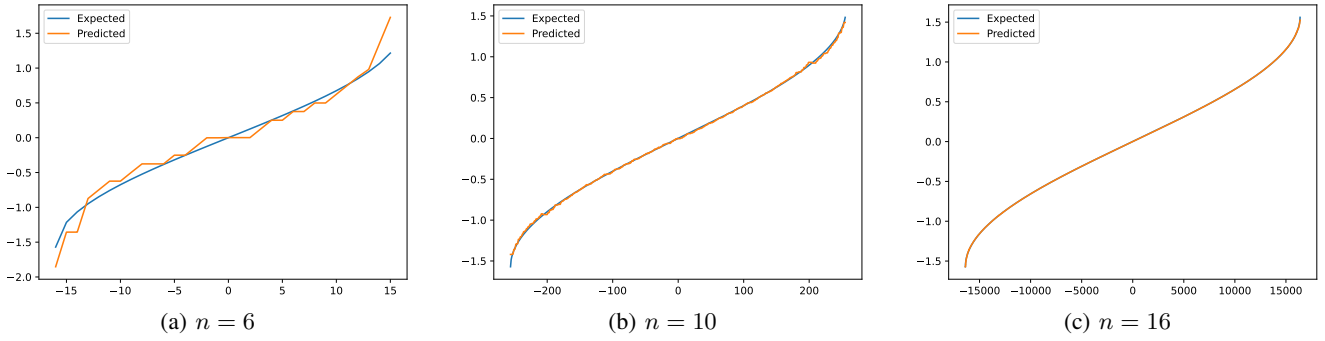


Fig. 3: Results of Classical simulations of Quantum compatible Algorithm for CORDIC Arcsine (see our GitHub repository at <https://github.com/iain-burge/QuantumCORDIC> for further details).

*Proof:* We proceed by induction. Base case: for  $i = 0$ , at the beginning of the loop  $\text{in} = \text{aux} = zr^0$ . Thus, the base case holds.

Inductive step: suppose that for an even  $i < \lceil \sqrt{5}\varphi^{-m}n \rceil$ , the registers hold the values,

$$\text{in} = z \sum_{k=0}^{F[i+1]-1} r^k, \quad \text{aux} = z \sum_{k=0}^{F[i]-1} r^k.$$

$i$  is even so the logical statement in Line 6 is true. Hence,  $\text{aux}$  is updated to,

$$\text{aux} \leftarrow z \sum_{k=0}^{F[i]-1} r^k + (-1)^{F[i]} \cdot 2^{mF[i]} z \sum_{k=0}^{F[i+1]-1} r^k.$$

By definition  $(-1)^{F[i]} \cdot 2^{mF[i]} = r^{F[i]}$ , thus, the new value for  $\text{aux}$  is  $z \sum_{k=0}^{F[i]-1} r^k + z \sum_{k=F[i]}^{F[i+1]+F[i]-1} r^k$ . By definition of the Fibonacci sequence,  $F[i+2] = F[i+1] + F[i]$ , combining the summations yields,

$$\text{aux} = z \sum_{k=0}^{F[i+2]-1} r^k.$$

On the next loop, the index register holding  $i$  incremented by 1 and is now odd, so the if statement on Line 6 gets false. Hence, the next operation updates  $\text{in}$ ,

$$\text{in} \leftarrow z \sum_{k=0}^{F[i+1]-1} r^k + (-1)^{F[i+1]} \cdot 2^{mF[i+1]} z \sum_{k=0}^{F[i+2]-1} r^k$$

Similarly to the previous iteration, we find  $\text{in}$  is equal to  $z \sum_{k=0}^{F[i+1]-1} r^k + z \sum_{k=F[i+1]}^{F[i+2]+F[i+1]-1} r^k$ . Bringing the summations together shows,

$$\text{in} = z \sum_{k=0}^{F[i+3]-1} r^k.$$

The index register is incremented once again, such that it holds  $i+2$ . By induction, the result holds.  $\blacksquare$

**Remark A.1.** Recall, for  $r \in (0, 1)$ , and an integer  $J$ , the geometric sum formula gives,

$$\sum_{k=0}^{J-1} r^k = \frac{1-r^J}{1-r} = \frac{1-2^{-Jm}}{1+2^{-m}}.$$

**Theorem A.2.** Assuming  $J$  iterations, the final state will be,

$$\text{in} \approx \frac{z}{1-2^{-m}}, \quad \text{aux} \approx 0,$$

with an accuracy of  $\mathcal{O}(m\varphi^J)$  bits.

*Proof:* First note the state prior to Line 10 is,

$$\text{in} = z \sum_{k=0}^{F[J+1]-1} r^k, \quad \text{aux} = z \sum_{k=0}^{F[J]-1} r^k,$$

where, for simplicity,  $J$  is even, greater than two, and represents the number of iterations. Then, Line 10 will result in state,

$$\text{aux} \leftarrow z \sum_{k=0}^{F[J]-1} r^k - z \sum_{k=0}^{F[J+1]-1} r^k.$$

This further yields  $\text{aux}$  is equal to  $-z \sum_{k=F[J]}^{F[J+1]-F[J]-1} r^k$ . This can be simplified to,  $-z r^{F[J]} \sum_{k=0}^{F[J]-1} r^k$ . Then, the geometric sum formula gives,

$$\text{aux} = -z r^{F[J]} \frac{1-r^{F[J]}}{1-r}.$$

Note that  $\varphi^{k-1} < F[k]$  for positive  $k$ . Thus,

$$\text{aux} < -z 2^{-\varphi^{J-1}m} \frac{1-2^{-\varphi^{J-2}m}}{1-2^{-m}}.$$

Note that  $z$  is always less than 2 as argued at the beginning of Section V, and since  $m \geq 1$ ,  $J \geq 2$ , we have,  $|\text{aux}| \leq 2^{-\varphi^{J-1}m+2}$ . Thus, the error introduced to  $\text{aux}$  in terms of bit precision is exponentially small.

On the other hand, the  $\text{in}$  register state is,

$$\text{in} = z \frac{1-2^{-mF[J]}}{1+2^{-m}}.$$

Thus, we have an absolute error of less than,  $2 \cdot 2^{-mF[J]} \leq 2^{-m\varphi^{J-1}}$ . Therefore, the absolute error is exponentially small with respect to bits of accuracy.  $\blacksquare$