Quantum CORDIC: Computing Arcsine on a Budget

Iain Burge
iainburge@cmail.carleton.ca
Ottawa, Ontario, Canada
ORCID: 0009-0008-5360-9430

***Abstract*—This work introduces a quantum-compatible algorithm for computing $\arcsin$ to n bits of accuracy. The method has a spacial complexity of $\mathcal{O}(n)$ auxiliary qubits, a layer count of roughly $\mathcal{O}(n \cdot \log(n))$ and a CNOT count of roughly $\mathcal{O}(n^2)$.**

I. Introduction

A. An annoying problem

Recently, I've been annoyed, because I've had a problem that's been annoying. Let's say you have an $(n+1)$-qubit quantum state like this:

$$|\psi\rangle = \sum_{k=0}^{2^n-1} \alpha_k |x_k\rangle_{\text{in}} |0\rangle_{\text{out}},$$

where $N = 2^n$, the in register has size n, the out register is of size one, and $x_k \in \{0,1\}^n$ is an n-bit binary string. You may someday want to find a U which trans $|\psi\rangle$ to state:

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

In other words, you may want to encode the binary values x_k into the amplitudes of the output registers.

B. Why we care about the annoying problem

A fast solution to this problem is valuable in multiple algorithms. For one, it would be a useful tool for a particular state preparation problem in my research [3]. However, it's also important in work that actually matters!

For example, the quantum speedup of Monte Carlo methods [8] requires the transformation W,

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

Where Φ is a function from binary strings to the interval from zero to one. Take register in to have m-qubits, and out to have 1-qubit. Now, suppose you have a unitary W' which takes binary strings to an n-bit binary approximation of $N \cdot \tilde{\Phi}(x)$.

$$|x\rangle_{\text{in}} |0\rangle_{\text{aux}} \xrightarrow{W'} |x\rangle |\Phi(x)\rangle$$

Then, W can be constructed with n auxiliary qubits by composing U (Equation 1) and W'. In summary, we can solve the problem with a reversible classical algorithm implemented on a quantum computer, and then shove the result into the amplitude of the output bit.

Another practical use case is as a step of the HHL algorithm [6]. In this case, some eigenvalues are encoded as binary strings in superposition. What they mean in this context isn't really necessary though, the important thing is we need to encode the reciprocals of the eigenvalues in the amplitude of an output bit. So, we simply need to perform a reversible algorithm for a number's reciprocal, then we can apply U from Equation (1).

To conclude, an efficient solution to the annoying problem has some immediate benefits to some foundational problems. Additionally, I suspect there are some other problems I'm not aware of where this comes up.

C. Why is the problem annoying

There are a few naive approaches to this problem. One is simply using a lookup table, where each possible input $|x_k\rangle$ is separately implemented. That would be exponential work though. It would also be possible to use qRAM [4], but the exponential space required for that seems very excessive for our current problem.

So, with some naive methods out of the way, here's a slightly less naive direction that will illustrate the main challenge of the paper. Consider the following state,

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

Where in and aux are n-qubit registers, and out is a one-qubit register. We will simply consider x_k to be an unsigned integer for now. Suppose we apply the function $\arcsin$ where in is the input register and aux is the output register. Applying this unitary, which we will call F, yields,

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

Next, suppose we apply the quantum circuit R from Figure 1. This would give us state,

$$\begin{aligned} R|\psi_1\rangle &= |\psi_2\rangle \\ &= \sum_{k=0}^{N-1} \alpha_k |x_k\rangle_{\text{in}} |\arcsin \frac{x_k}{N}\rangle_{\text{aux}} \\ &\quad \left(\cos\left(\arcsin \frac{x_k}{N}\right) |0\rangle + \sin\left(\arcsin \frac{x_k}{N}\right) |1\rangle \right)_{\text{out}}. \end{aligned}$$

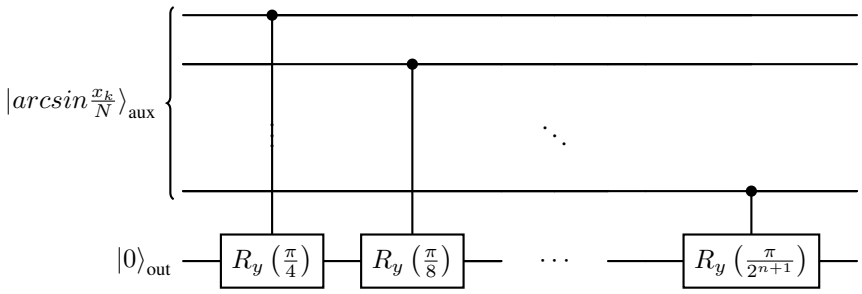


Fig. 1: Unitary transformation R which encodes the binary value of the aux register into the output register. Define $R_y(\theta) = (\cos\theta, -\sin\theta; \sin\theta, \cos\theta)$.

With a bit of trigonometry, we can see,

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

Uncomputing $\arcsin x_k/N$ by performing inverse F, gives us our desired state (Equation (1)).

So, to conclude, why is this so annoying that I insist on overusing the word annoying? Because it's very hard to compute F efficiently!

D. A Brief Outline to Tackle the Annoying Problem

In the next section, Section II, we will discuss a couple of previous attempts at this problem. In Section III, I will introduce a classical approach, known as CORDIC, which can approximate arcsine on very limited hardware. In Section IV, I will translate the approach to a quantum setting. Section V will give a rough idea of complexity. Section VI will discuss some classical simulations of my proposed methods. Finally, Section VII will conclude the work, discussing the spatial and time complexity of my new method.

II. Previous Attempts

There are two relevant attempts I know of towards solving this problem. Häner et al.'s approach involved piecewise polynomial approximations [5]. It's super cool, but far from a short-term solution. Not only does it require a fair few multiplications, but it also requires square rooting for the extreme portions of $\arcsin x$ (i.e. $x \in (-1, -0.5] \cup [0.5, 1]$). To conclude, it's a promising solution but might be a bit too heavy-duty.

There also exists a neat CORDIC flavoured method [10], which does a kind of binary search to approximate $\arcsin x$. Unfortunately, the suggested solution requires a whole lot of squaring operations, which isn't easy to implement in the near term.

III. Classical CORDIC for Arcsine

So, as a solution to our annoying problem, how about we borrow some ideas from the lovely world of low-performance computing? We will try to leverage the CORDIC algorithm to calculate arcsine. For a more comprehensive introduction to traditional CORDIC, check out the great forum post *CORDIC for Dummies* [1].

Let's first give a brief idea of how CORDIC works. We first construct a goal vector that encodes our problem. Then we rotate a vector towards our goal in increasingly small discrete steps, until it converges with satisfactory error. The beautiful thing is, with a few sneaky math tricks, it is possible to avoid performing any multiplications during the rotations! There are variations of CORDIC that can be used to efficiently calculate various mathematical functions, including $1/x$, $\sqrt{x}$, e^x, $\arcsin$, hsin, and more [7].

I will be basing my algorithm on Mazenc et al.'s clever method [7]. Note, that Step 2 is slightly modified since there was a bug in the original paper's algorithm. The procedure is described as an algorithm (Algorithm 1), and as a series of steps:

Algorithm for $\arcsin(t) = \theta$, $t \in [-1, 1]$:

1) Initialize: $\theta_0 = 0$, $x_0 = 1$, $y_0 = 0$, $t_0 = t$.
2) $d_i = \text{isneg}(x_i)$ xor $\text{isneg}(t_i - \text{isneg}(x_i) \cdot y_i)$. Where $\text{isneg}(z) = 1$ if $(z < 0)$ else 0.
3) If $d_i = 1$, then swap x and y registers.
4) Perform "rotation":
$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} 1 & -2^{-i} \\ 2^{-i} & 1 \end{pmatrix}^2 \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$
5) If $d_i = 1$, then unswap x and y registers.
6) Compensate for "rotation":
$$t_{i+1} = (1 + 2^{-2i}) t_i$$
7) Update theta approximation:
$$\theta_{i+1} = \theta_i + (-1)^{d_i} 2 \arctan 2^{-i}.$$
8) Repeat steps 2-7 until the desired accuracy is reached.

Algorithm 1 CORDIC Arcsine

procedure ARCSIN(t, `#iterations`) ▷ $t \in [-1, 1]$
  $\theta \leftarrow 0$; $x \leftarrow 1$, $y \leftarrow 0$, $t \leftarrow t, d_i \leftarrow 0$.
  for $i <$ `#iterations`, `++`i **do**
    $d_i \leftarrow \text{isneg}(x)$ xor $\text{isneg}(t - \text{isneg}(x) \cdot y)$
    if d_i **then** $x, y \leftarrow y, x$
    for `_ in range(2)` **do**
      $x, y \leftarrow x - 2^{-i}y, y + 2^{-i}x$
    if d_i **then** $x, y \leftarrow y, x$
    $t \leftarrow (1 + 2^{-2i})t$
    $\theta \leftarrow \theta + (-1)^{d_i} 2 \arctan 2^{-i}$
  return θ

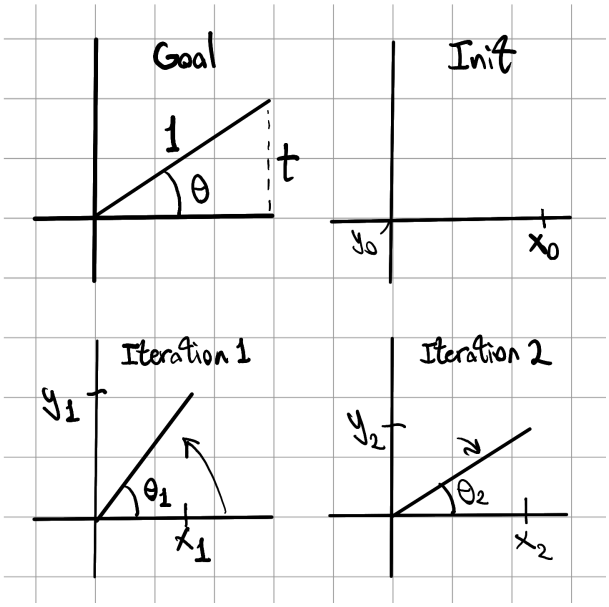


Fig. 2: Example of CORDIC iterations to approximate arcsine.

As $i \to \infty$, $\theta_i \to \theta$. To achieve n bits of accuracy it takes $n + 2$ iterations [7]. These steps likely seem cryptic, so let's walk through them.

Step 1 is initializing our problem, 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 (Figure 2 - Goal). This goal vector will have an angle of θ. So if we keep track of all the small rotations we make, and we point in the correct direction, we will have an approximation for θ.

Step 2 is a convoluted but 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 θ. Otherwise, if $d_i = 1$, we need to rotate clockwise. Steps 3 and 5 conjugate Step 4 to achieve the correct rotation direction.

Step 4 performs a rotation, and unfortunately slightly stretches our vector (x_i, y_i) by a factor of $1 + 2^{-2i}$. It can be done simply by making the following substitutions twice: $x' = x - 2^{-i}y$, $y' = y + 2^{-i}x$. As a result of the operation, we're pointed in a better direction, but we need to compensate for the stretch.

Step 6 deals with the stretching of the (x_i, y_i) vector by stretching our goal vector by the same amount!

Finally, Step 7 records the change to the current angle of (x_i, y_i). Note that $2 \arctan 2^{-i}$ can be precomputed, or even directly encoded into hardware.

The big takeaway is that each iteration uses only about 6 bitshifts and 7 additions, regardless of the number of bits!

IV. 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'll go through it step by step! Before going through the steps we have a few important facts to note. We represent each number $(\theta_i, x_i, y_i, t_i)$ using fixed point two's complement in the range $(-2, 2)$. This avoids any issues with overflow, since $\theta \in (-\pi/2, \pi/2]$, and

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

You can verify the inequality by taking $i \to \infty$, taking the $\ln$ of the infinite product, using the inequality $\ln(1 + x) < x$, and finally using the geometric series. The product represents the stretch caused to each value throughout 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, if it is 0 then the value is positive. Also note that, when right bit-shifting the leftmost bit is copied into the new leftmost bit, for example:

$$11010001 >> 3 = 11111010.$$

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

Now, let's go through the steps of the quantum implementation, assuming n bits to represent each number. Step 1 is trivial, we use t as the input, and we can even keep using the same register. Then we need to carve out registers of equal size to hold the x_i, y_i, and θ_i values. Note that we do *not* need to use new registers for each iteration, one per integer will suffice.

Step 2 is where things start to get a bit more interesting. We will need a register of size `#Number_of_iterations` to store each result d_i as a bit array. We can perform this step with the following sequence of operations, where the leftmost bit is most significant. We define the binary strings as $x_i = x_i^{n-1}x_i^{n-2}\ldots x_i^0$, $y_i = y_i^{n-1}y_i^{n-2}\ldots y_i^0$, $t_i = t_i^{n-1}t_i^{n-2}\ldots t_i^0$. We begin with the state:

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

We perform a controlled subtraction (implemented with an inverse addition gate) from y_i to t_i controlled by the sign bit of x_i, x_i^{n-1}, yielding $T_i = t_i - x_i^{n-1} \cdot y_i$,

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

Next, we do a controlled not from the sign bits of x_i and T_i onto the d_i bit.

$$|x_i^{n-1} \oplus T_i^{n-1}\rangle_{d_i} |x_i\rangle_x |y_i\rangle_y |T_i\rangle_t$$

Finally, we can undo the controlled subtraction with a controlled addition.

Steps 3 and 5 are trivial to implement, just use a bunch of swap gates controlled by d_i, this can be done in $\mathcal{O}(1)$ layers.

Step 4 is probably the trickiest step. I needed to develop a sneaky new algorithm (Algorithm 2 - `Mult`) to solve this problem! Note, that each step of Algorithm 2 is reversible. The explanation for `Mult` will be fleshed out in future work. So, our initial state is,

$$|x_i\rangle |y_i\rangle .$$

We first subtract $y_i >> i$ from y_i, giving us, (recall, we are working in two's complement so the leftmost-bit is copied during right-shifts)

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

Next, we need to take two clean-ish (values near 0) auxiliary registers. We apply $\texttt{Mult}(y_i, \text{aux}_1, \text{aux}_2, 2i)$ to the y register,

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

Finally, we add the x register shifted, $(x_i - 2^{-i} y_i) >> i$, to the y register which contains $y_i + 2^{-2i} y$,

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

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

With our new `Mult` tool, Step 6 is trivial. We simply apply $\texttt{Mult}(t_i, \text{aux}_1, \text{aux}_2, 2i)$ to the t register.

Finally, *Step* 7 has a fun trick, apply controlled negation of the θ register using d_i as the control. Then add the precomputed $2 \arctan 2^{-i}$ to the θ register. Then, once again, perform a controlled negation of the θ register using d_i as the control. This will give,

$$\begin{aligned}
|\theta_i\rangle &\to |(-1)^{d_i}\theta_i\rangle \\
&\to |(-1)^{d_i}\theta_i + 2\arctan 2^{-i}\rangle \\
&\to |(-1)^{2d_i}\theta_i + (-1)^{d_i} 2\arctan 2^{-i}\rangle \\
&= |\theta_i + (-1)^{d_i} 2\arctan 2^{-i}\rangle \\
&= |\theta_{i+1}\rangle
\end{aligned}$$

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

V. Complexity

This section will give a rough idea of complexity and will be fairly informal. My quantum implementation of Algorithm 1 applies `Mult` 3 times per iteration, and addition 6 times per iteration. Mult takes $c \cdot \log_\phi(n/i) + 4$ iterations on the i^th step, where c is a constant which will need a bit more work

Algorithm 2 $\text{Mult}(z, \text{aux}_1, \text{aux}_2, m) \approx (1 + 2^{-m})z$

procedure MULT(z, aux_1, aux_2, m) ▷ z is a register, aux_1 is an auxiliary register with 0 value, aux_2 is a auxiliary register with near 0 value, m is the right shift
  Fib $\leftarrow [1, 1, 2, 3, 5, 8, \dots]$
  `#iterations` $\in \left(\mathcal{O}(\log_\phi(n/m)\right)$
  $\text{aux}_2 \leftarrow \text{aux}_2 + z$
  $z \leftarrow z + \text{aux}_2 >> m$
  $\text{aux}_1 \leftarrow \text{aux}_1 + z$
  for $i = 0$, $i <$ `#iterations`, $++i$ **do**
    if i Even **then**
      $z \leftarrow z + (-1)^{Fib[i]}(\text{aux}_1 >> (m \cdot Fib[i]))$
    else
      $\text{aux}_1 \leftarrow \text{aux}_1 + (-1)^{Fib[i]}(z >> (m \cdot Fib[i]))$
  $\text{aux}_2 \leftarrow \text{aux}_2 - \text{aux}_1$
  for $i =$ `#iterations`, $i \geq 0$, $--i$ **do**
    if i even **then**
      $z \leftarrow z - (-1)^{Fib[i]}(\text{aux}_1 >> (m \cdot Fib[i]))$
    else
      $\text{aux}_1 \leftarrow \text{aux}_1 - (-1)^{Fib[i]}(z >> (m \cdot Fib[i]))$
  $\text{aux}_1 \leftarrow \text{aux}_1 - z$
return z

to find, and ϕ is the golden ratio. Thus, assuming n iterations for $n-2$ bits of accuracy (note this ignoring the error added by `Mult` which is small in simulations), we must perform,

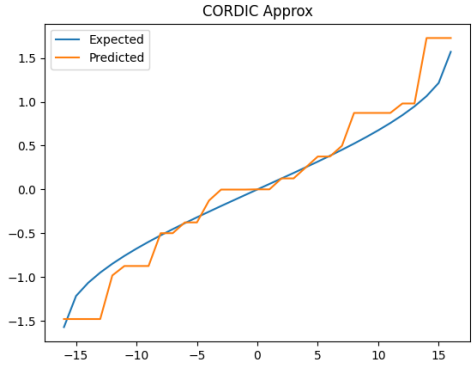
$$\begin{aligned}
\sum_{i=1}^{n}(c \cdot \log_\phi(n/i) + 4 + 6) &= 10n + \frac{c}{\ln\phi}\sum_{i=1}^{n}\ln(n/i) \\
&\approx 10n + \frac{c}{\ln\phi}\int_1^n \ln(n/x)dx \\
&< \left(10 + \frac{c}{\ln\phi}\right) n,
\end{aligned}$$

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.

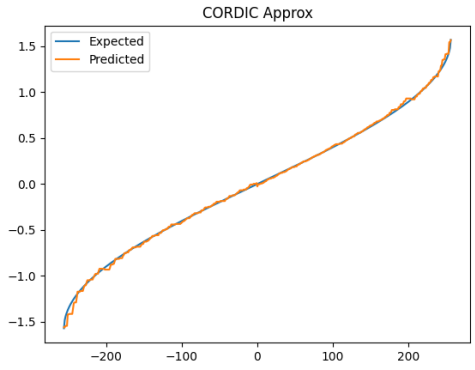
As for space complexity, we need $\mathcal{O}(n)$ bits of space. We need three auxiliary registers of size n, two for `Mult`, and one for addition. We need a register of n bits for θ_i, x_0, y_0 and t_0. Finally, we need $n+2$ bits to store the binary array d. Thus, in total, we need about $8n + 2$ bits. Note though, that this could be improved quite easily, this is a first pass at the problem.

VI. Simulaitons

In my simulations, I found that the max error for any given input can be made arbitrarily small by making n larger. See Figure 3 to get an idea of the error. In this submission, there is a file with a complete classical implementation of the quantum-compatible algorithm. The full codebase can be found on GitHub [2]. Additionally, there is a demo version of `Mult` implemented as a quantum circuit in Qiskit.



(a) $n = 6$



(b) $n = 10$

Fig. 3: Results of Classical simulations of Quantum compatible Algorithm for CORDIC Arcsine.

VII. Conclusion

To conclude, I have proposed a Quantum algorithm for calculating Arcsine with fairly low space and time complexity, though a more in-depth assessment of complexity is necessary in the future. I took advantage of a few sneaky tricks, and, based on the simulation results, it looks like it works well! This work has applications in HHL and quantum Monte Carlo methods.

References

[1] Bean: Cordic for dummies (Nov 2010), https://forums.parallax.com/discussion/127241/cordic-for-dummies

[2] Burge, I.: Quantum cordic algorithm (Feb 2024), https://github.com/iain-burge/QHack2024

[3] Burge, I., Barbeau, M., Garcia-Alfaro, J.: Quantum algorithms for shapley value calculation. In: 2023 IEEE International Conference on Quantum Computing and Engineering (QCE). vol. 1, pp. 1–9. IEEE (2023)

[4] Giovannetti, V., Lloyd, S., Maccone, L.: Quantum random access memory. Physical review letters **100**(16), 160501 (2008)

[5] Häner, T., Roetteler, M., Svore, K.M.: Optimizing quantum circuits for arithmetic. arXiv preprint arXiv:1805.12445 (2018)

[6] Harrow, A.W., Hassidim, A., Lloyd, S.: Quantum algorithm for linear systems of equations. Physical review letters **103**(15), 150502 (2009)

[7] Mazenc, C., Merrheim, X., Muller, J.M.: Computing functions cos/sup-1/and sin/sup-1/using cordic. IEEE Transactions on Computers **42**(1), 118–122 (1993)

[8] Montanaro, A.: Quantum speedup of monte carlo methods. Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences **471**(2181), 20150301 (2015)

[9] Takahashi, Y., Tani, S., Kunihiro, N.: Quantum addition circuits and unbounded fan-out. arXiv preprint arXiv:0910.2530 (2009)

[10] Wang, S., Wang, Z., Li, W., Fan, L., Cui, G., Wei, Z., Gu, Y.: Quantum circuits design for evaluating transcendental functions based on a function-value binary expansion method. Quantum Information Processing **19**, 1–31 (2020)