CSE565M: Acceleration of Algorithms in Reconfigurable Logic

Learn by Doing: CORDIC (Pt. 1)

Anthony Cabrera

FL24::L05

Washington University in St. Louis

Table of contents

Overview

What is CORDIC

CORDIC stands for **C**oordinate **R**otation **DI**gital **C**omputer

- digit-by-digit algorithm that produces one output digit per iteration
- fine-tune the algorithm based on desired **accuracy**
- performs simpule computations using only addition, substraction, bit shifting, and table lookups, which are efficient to implement in hardware. (Multiply ops on *just* the FPGA fabric is expensive)
- most of the computation is performed within a single `for` loop

What is CORDIC

CORDIC stands for **C**oordinate **R**otation **DI**gital **C**omputer

- digit-by-digit algorithm that produces one output digit per iteration
- fine-tune the algorithm based on desired **accuracy**
- performs simpule computations using only addition, substraction, bit shifting, and table lookups, which are efficient to implement in hardware. (Multiply ops on *just* the FPGA fabric is expensive)
- most of the computation is performed within a single `for` loop

What is CORDIC

CORDIC stands for **C**oordinate **R**otation **DI**gital **C**omputer

- digit-by-digit algorithm that produces one output digit per iteration
- fine-tune the algorithm based on desired **accuracy**
- performs simpule computations using only addition, substraction, bit shifting, and table lookups, which are efficient to implement in hardware. (Multiply ops on *just* the FPGA fabric is expensive)
- most of the computation is performed within a single `for` loop

What is CORDIC

CORDIC stands for **C**oordinate **R**otation **DI**gital **C**omputer

- digit-by-digit algorithm that produces one output digit per iteration
- fine-tune the algorithm based on desired **accuracy**
- performs simpule computations using only addition, substraction, bit shifting, and table lookups, which are efficient to implement in hardware. (Multiply ops on *just* the FPGA fabric is expensive)
- most of the computation is performed within a single `for` loop

Learning Goal

Understand the CORDIC algorithm well enough to create an optimized CORDIC compute core using HLS

The major HLS optimization we will highlight with CORDIC is choosing the correct number representation for the variables

- tradeoff between accuracy, performance, and resource utilization of the design

The Tradeoff

Larger numbers provide more precision at the cost of increased resource usage (more FFs and logic blocks).

Background

Core Idea

Efficiently perform a set of vector rotations in a 2D plane

- With these rotations, we can implement a variety of fundamental ops
  - trigonometric
  - hyperbolic
  - logarithmic

Cordic Algo for Sine/Cosine Overview

How to use CORDIC to compute sine and cosine for a given input angle, θ

- perform a series of vector rotations in order to reach the target angle theta

Cordic Algo for Sine/Cosine Overview (cont)

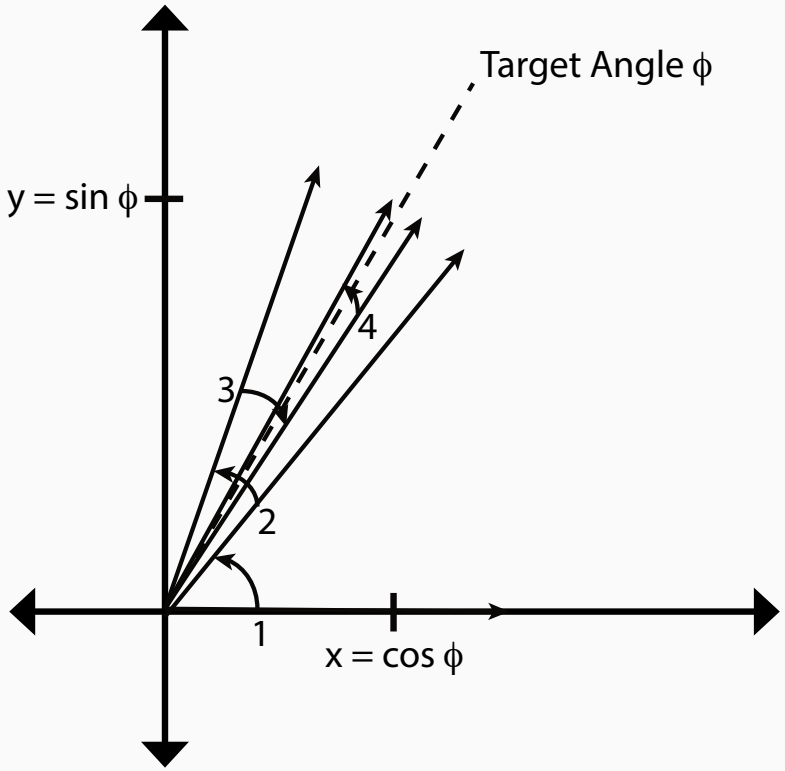

Figure 1: Using the CORDIC to calculate the functions $\sin\phi$ and $\cos\phi$. CORDIC starts at the x-axis with a corresponding 0° angle. Performs four iterative +/- rotations in increasingly smaller rotation angle to reach target angle ϕ. The corresponding x and y values of the final vector correspond to $\cos\phi$ and $\sin\phi$ (respectively).

Cordic Algo for Sine/Cosine Overview (cont)

Algorithm at a High Level:

- Perform an iterative series of rotations
- Each subsequent rotation uses an increasingly smaller angle, so each rotation adds a little bit of precision
  - More iterations yields more accuracy at the expense of latency
- The rotation angles are fixed at compile time
  - can esaily store values in a small memory and keep a running sum of the cumulative rotated angle
- after an iteration, if the cumulative angle is larger than the target angle ϕ, we perform a negative rotation. If it's smaller, than we perform a positive rotation.
- Once you've performed the set number of rotations, you can determine $cos(\phi)$ and $cos(\phi)$ simply by looking at the x and y values from the final rotated vector.

Cordic Algo for Sine/Cosine Overview (cont)

Algorithm at a High Level:

- Perform an iterative series of rotations
- Each subsequent rotation uses an increasingly smaller angle, so each rotation adds a little bit of precision
  - More iterations yields more accuracy at the expense of latency
- The rotation angles are fixed at compile time
  - can esaily store values in a small memory and keep a running sum of the cumulative rotated angle
- after an iteration, if the cumulative angle is larger than the target angle ϕ, we perform a negative rotation. If it's smaller, than we perform a positive rotation.
- Once you've performed the set number of rotations, you can determine $cos(\phi)$ and $cos(\phi)$ simply by looking at the x and y values from the final rotated vector.

Cordic Algo for Sine/Cosine Overview (cont)

Algorithm at a High Level:

- Perform an iterative series of rotations
- Each subsequent rotation uses an increasingly smaller angle, so each rotation adds a little bit of precision
  - More iterations yields more accuracy at the expense of latency
- The rotation angles are fixed at compile time
  - can esaily store values in a small memory and keep a running sum of the cumulative rotated angle
- after an iteration, if the cumulative angle is larger than the target angle ϕ, we perform a negative rotation. If it's smaller, than we perform a positive rotation.
- Once you've performed the set number of rotations, you can determine $cos(\phi)$ and $cos(\phi)$ simply by looking at the x and y values from the final rotated vector.

Cordic Algo for Sine/Cosine Overview (cont)

Algorithm at a High Level:

- Perform an iterative series of rotations
- Each subsequent rotation uses an increasingly smaller angle, so each rotation adds a little bit of precision
  - More iterations yields more accuracy at the expense of latency
- The rotation angles are fixed at compile time
  - can esaily store values in a small memory and keep a running sum of the cumulative rotated angle
- after an iteration, if the cumulative angle is larger than the target angle ϕ, we perform a negative rotation. If it's smaller, than we perform a positive rotation.
- Once you've performed the set number of rotations, you can determine $cos(\phi)$ and $cos(\phi)$ simply by looking at the x and y values from the final rotated vector.

Cordic Algo for Sine/Cosine Overview (cont)

Algorithm at a High Level:

- Perform an iterative series of rotations
- Each subsequent rotation uses an increasingly smaller angle, so each rotation adds a little bit of precision
  - More iterations yields more accuracy at the expense of latency
- The rotation angles are fixed at compile time
  - can esaily store values in a small memory and keep a running sum of the cumulative rotated angle
- after an iteration, if the cumulative angle is larger than the target angle ϕ, we perform a negative rotation. If it's smaller, than we perform a positive rotation.
- Once you've performed the set number of rotations, you can determine $cos(\phi)$ and $cos(\phi)$ simply by looking at the x and y values from the final rotated vector.

Cordic Algo for Sine/Cosine Overview (cont)

Algorithm at a High Level:

- Perform an iterative series of rotations
- Each subsequent rotation uses an increasingly smaller angle, so each rotation adds a little bit of precision
  - More iterations yields more accuracy at the expense of latency
- The rotation angles are fixed at compile time
  - can esaily store values in a small memory and keep a running sum of the cumulative rotated angle
- after an iteration, if the cumulative angle is larger than the target angle ϕ, we perform a negative rotation. If it's smaller, than we perform a positive rotation.
- Once you've performed the set number of rotations, you can determine $cos(\phi)$ and $cos(\phi)$ simply by looking at the x and y values from the final rotated vector.

Cordic Algo for Sine/Cosine Overview (cont)

Algorithm at a High Level:

- Perform an iterative series of rotations
- Each subsequent rotation uses an increasingly smaller angle, so each rotation adds a little bit of precision
  - More iterations yields more accuracy at the expense of latency
- The rotation angles are fixed at compile time
  - can esaily store values in a small memory and keep a running sum of the cumulative rotated angle
- after an iteration, if the cumulative angle is larger than the target angle ϕ, we perform a negative rotation. If it's smaller, than we perform a positive rotation.
- Once you've performed the set number of rotations, you can determine $cos(\phi)$ and $cos(\phi)$ simply by looking at the x and y values from the final rotated vector.

Cordic Algo for Sine/Cosine Overview (cont)

How do we represent the rotation? In general for the 2D case, consider the rotation matrix in the Cartesian plane:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{1}$$

Cordic Algo for Sine/Cosine Overview (cont)

Each rotation iteration takes the form

$$v_i = R_i \cdot v_{i-1} \tag{2}$$

Thus, in CORDIC, we perform the following operations

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{3}$$

Writing out the linear equations, the coordinates of the newly rotated vector are:

$$x_i = x_{i-1}\cos\theta - y_{i-1}\sin\theta \tag{4}$$

and

$$y_i = x_{i-1}\sin\theta + y_{i-1}\cos\theta \tag{5}$$

Cordic Algo for Sine/Cosine Overview (cont)

Consider first a 90° rotation. In this case the rotation matrix is:

$$R(90^\circ) = \begin{bmatrix} \cos 90^\circ & -\sin 90^\circ \\ \sin 90^\circ & \cos 90^\circ \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \tag{6}$$

and thus we only have to perform the operations:

$$\begin{aligned} x_i &= x_{i-1} \cos 90^\circ - y_{i-1} \sin 90^\circ \\ &= x_{i-1} \cdot 0 - y_{i-1} \cdot 1 \\ &= -y_{i-1} \end{aligned} \tag{7}$$

and

$$\begin{aligned} y_i &= x_{i-1} \sin 90^\circ + y_{i-1} \cos 90^\circ \\ &= x_{i-1} \cdot 1 + y_{i-1} \cdot 0 \\ &= x_{i-1} \end{aligned} \tag{8}$$

Cordic Algo for Sine/Cosine Overview (cont)

Putting this altogether we get

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -y \\ x \end{bmatrix} \tag{9}$$

You can see that this is requires a very minimal amount of calculation; the rotated vector simply negates the y value, and then swaps the x and y values. A two's complement negation requires the hardware equivalent to an adder.

CORDIC :: Exercise 1

Rotating instead by $-90°$?

What if you wanted to rotation by $-90°$? What is the rotation matrix $R(-90°)$? What type of calculation is required for this rotation? How would one design the most efficient circuit that could perform a positive and negative rotation by $-90°$, i.e., the direction of rotation is an input to the circuit?

How to Rotate with Smaller Angles?

Let's try $\pm 45^\circ$. Using the rotation matrix from Equation 1

$$R(45^\circ) = \begin{bmatrix} \cos 45^\circ & -\sin 45^\circ \\ \sin 45^\circ & \cos 45^\circ \end{bmatrix} = \begin{bmatrix} \sqrt{2}/2 & -\sqrt{2}/2 \\ \sqrt{2}/2 & \sqrt{2}/2 \end{bmatrix} \tag{10}$$

Calculating out the computation for performing the rotation, we get

$$\begin{aligned} x_i &= x_{i-1} \cos 45^\circ - y_{i-1} \sin 45^\circ \\ &= x_{i-1} \cdot \sqrt{2}/2 - y_{i-1} \cdot \sqrt{2}/2 \end{aligned} \tag{11}$$

and

$$\begin{aligned} y_i &= x_{i-1} \sin 45^\circ + y_{i-1} \cos 45^\circ \\ &= x_{i-1} \cdot \sqrt{2}/2 + y_{i-1} \cdot \sqrt{2}/2 \end{aligned} \tag{12}$$

which when put back into matrix vector notation is

$$\begin{bmatrix} \sqrt{2}/2 & -\sqrt{2}/2 \\ \sqrt{2}/2 & \sqrt{2}/2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \sqrt{2}/2x - \sqrt{2}/2y \\ \sqrt{2}/2x + \sqrt{2}/2y \end{bmatrix} \tag{13}$$

How to Rotate with Smaller Angles?

This certainly is not as efficient of a computation as compared to rotating by $\pm 90^{\circ}$.

What if we "forced" the rotation matrix to be constants that were easy to multiply?

- If we set the constants in the rotation matrix to be powers of two, we could very easily perform rotations without multiplication.
- This is the key idea behind the CORDIC – finding rotations that are very efficient to compute while minimizing any "side effects".

Important!

There is an engineering decision that is being made here. In order to get efficient computationwe have to deal with the fact that the rotation also performs scaling, i.e., it changes the magnitude of the rotated vector – more on this later.

How to Rotate with Smaller Angles?

This certainly is not as efficient of a computation as compared to rotating by $\pm 90^{\circ}$.

What if we "forced" the rotation matrix to be constants that were easy to multiply?

- If we set the constants in the rotation matrix to be powers of two, we could very easily perform rotations without multiplication.
- This is the key idea behind the CORDIC – finding rotations that are very efficient to compute while minimizing any "side effects".

Important!

There is an engineering decision that is being made here. In order to get efficient computationwe have to deal with the fact that the rotation also performs scaling, i.e., it changes the magnitude of the rotated vector – more on this later.

How to Rotate with Smaller Angles?

Consider again the rotation matrix

$$R_i(\theta) = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) \end{bmatrix} \tag{14}$$

By using the following trigonometric identities,

$$\cos(\theta_i) = \frac{1}{\sqrt{1 + \tan^2(\theta_i)}} \tag{15}$$

$$\sin(\theta_i) = \frac{\tan(\theta_i)}{\sqrt{1 + \tan^2(\theta_i)}} \tag{16}$$

we can rewrite the rotation matrix as

$$R_i = \frac{1}{\sqrt{1 + \tan^2(\theta_i)}} \begin{bmatrix} 1 & -\tan(\theta_i) \\ \tan(\theta_i) & 1 \end{bmatrix} \tag{17}$$

How to Rotate with Smaller Angles?

If we restrict the values of $\tan(\theta_i)$ to be a multiplication by a factor of two, the rotation can be performed using **shifts** (for the multiplication) **and additions**. More specifically, we let $\tan(\theta_i) = 2^{-i}$. The rotation then becomes

$$v_i = K_i \begin{bmatrix} 1 & -2^{-i} \\ 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix} \tag{18}$$

where

$$K_i = \frac{1}{\sqrt{1 + 2^{-2i}}} \tag{19}$$

How to Rotate with Smaller Angles?

A few things to note here. The 2^{-i} is equivalent to a right shift by i bits (div by a power of 2). This is essentially just a simple rewiring which does not require any sort of logical resources, i.e., it is essentially "free" to compute in hardware.

How to Rotate with Smaller Angles?

But what are the drawbacks?

- We are limited to rotate by angles θ such that $\tan(\theta_i) = 2^{-i}$.
- We are only showing rotation in one direction; the CORDIC requires the ability to rotation by $\pm\theta$. This is simple to correct by adding in σ which can have a value of 1 or -1, which corresponds to performing a positive or negative rotation. We can have a different σ_i at every iteration/rotation. Thus the rotation operation generalizes to

$$v_i = K_i \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix} \tag{20}$$

How to Rotate with Smaller Angles?

But what are the drawbacks?

- We are limited to rotate by angles θ such that $\tan(\theta_i) = 2^{-i}$.
- We are only showing rotation in one direction; the CORDIC requires the ability to rotation by $\pm\theta$. This is simple to correct by adding in σ which can have a value of 1 or -1, which corresponds to performing a positive or negative rotation. We can have a different σ_i at every iteration/rotation. Thus the rotation operation generalizes to

$$v_i = K_i \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix} \tag{20}$$

How to Rotate with Smaller Angles?

But what are the drawbacks? (cont)

- Finally, the rotation requires a multiplication by K_i. K_i is typically ignored in the iterative process and then adjusted for after the series of rotations is completed. The cumulative scaling factor is

$$K(n) = \prod_{i=0}^{n-1} K_i = \prod_{i=0}^{n-1} \frac{1}{\sqrt{1+2^{-2i}}} \tag{21}$$

and

$$K = \lim_{n\to\infty} K(n) \approx 0.607252935008881256169 \tag{22}$$

The scaling factors for different iterations can be calculated in advance and stored in a table. If we always perform a fixed number of rotations, this is simply one constant. Sometimes it is ok to ignore this scaling, which results in a processing gain

$$A = \frac{1}{K} = \lim_{n\to\infty} \prod_{i=0}^{n-1} \sqrt{1+2^{-2i}} \approx 1.64676025812107 \tag{23}$$

Example CORDIC Gains

Table 1: The rotating angle, scaling factor, and CORDIC gain for the first seven iterations of a CORDIC. Note that the angle decreases by approximately half each time. The scaling factor indicates how much the length the the vector increases during that rotation. The CORDIC gain is the overall increase in the length of the vector which is the product of all of the scaling factors for the current and previous rotations.

i	2^{-i}	Rotating Angle	Scaling Factor	CORDIC Gain
0	1.0	45.000°	1.41421	1.41421
1	0.5	26.565°	1.11803	1.58114
2	0.25	14.036°	1.03078	1.62980
3	0.125	7.125°	1.00778	1.64248
4	0.0625	3.576°	1.00195	1.64569
5	0.03125	1.790°	1.00049	1.64649
6	0.015625	0.895°	1.00012	1.64669

CORDIC :: Exercise 2

On precision

Describe the effect of the ith iteration on the precision of the results? That is, what bits does it change? How does more iterations change the precision of the final result, i.e., how do the values of $\sin\phi$ and $\cos\phi$ change as the CORDIC performs more iterations?

Calculating Sine and Cosine

A Specific Example

Start at 0° and approach the target angle $\phi = 60^\circ$. Thus we are interested in computing $cos(60^\circ)$ and $sin(60^\circ)$.

We'll use the values from Table 1 as the pre-defined angles to rotate by.

i	2^{-i}	Rotating Angle	Scaling Factor	CORDIC Gain
0	1.0	45.000°	1.41421	1.41421
1	0.5	26.565°	1.11803	1.58114
2	0.25	14.036°	1.03078	1.62980
3	0.125	7.125°	1.00778	1.64248
4	0.0625	3.576°	1.00195	1.64569
5	0.03125	1.790°	1.00049	1.64649
6	0.015625	0.895°	1.00012	1.64669

A Specific Example (cont.)

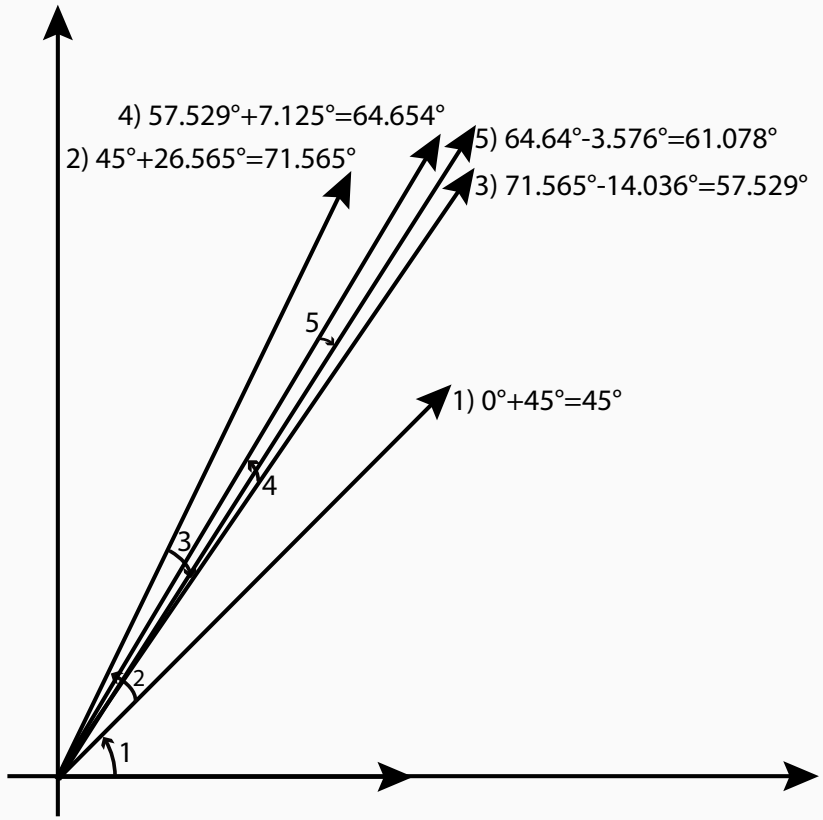

Figure 2: Calculating $\cos 60^\circ$ and $\sin 60^\circ$ using the CORDIC algorithm. Five rotations are performed using incrementally larger i values (0,1,2,3,4). The result is a vector with an angle of 61.078°. The corresponding x and y values of that vector give the approximate desired cosine and sine values.

CORDIC :: Exercise 3

Additonal rotations

How would the answer change if we performed one more rotation? How about two (three, four, etc.) more rotations? What is the accuracy (e.g., compared to a MATLAB implementation) as we perform more rotations?

CORDIC :: Exercise 4

Additonal rotations

Is it possible to get worse accuracy by performing more rotations?

Provide an example when this would occur.

Sequential Version of CORDIC

```
#include "cordic.h"
// The cordic_phase array holds the angle for the current rotation
// cordic_phase[0] =~ 0.785, cordic_phase[1] =~ 0.463, etc.
void cordic(THETA_TYPE theta, COS_SIN_TYPE &s, COS_SIN_TYPE &c) {
  // Set the initial vector that we will rotate
  COS_SIN_TYPE current_cos = 0.60735; // start at x = 1, y = 0, phi = 0
  COS_SIN_TYPE current_sin = 0.0;

  COS_SIN_TYPE factor = 1.0;
  // This loop iteratively rotates the initial vector to find the
  // sine and cosine values corresponding to the input theta angle
  for (int j = 0; j < NUM_ITERATIONS; j++) {
    // Determine if we are rotating by a positive or negative angle
    int sigma = (theta < 0) ? -1 : 1;

    // Multiply previous iteration by 2^(-j)
    COS_SIN_TYPE cos_shift = current_cos * sigma * factor;
    COS_SIN_TYPE sin_shift = current_sin * sigma * factor;

    // Perform the rotation
    current_cos = current_cos - sin_shift;
    current_sin = current_sin + cos_shift;

    // Determine the new theta
    theta = theta - sigma * cordic_phase[j];

    factor = factor / 2;
  }
  // Set the final sine and cosine values
  s = current_sin;
  c = current_cos;
}
```

References i