The CORDIC Algorithm

Return to...
Notes Page

Introduction

6.1

- On a current FPGA there are ***many*** multipliers and adders available. However for various communciations techniques, and matrix algorithms, we require trigonometry, square root etc...

  How would you perform these computations on an FPGA?

  Perhaps look-up tables, iterative techniques (or even come up with algorithms to try and circumvent the trigonometry!)

- In this section the CORDIC algorithm is introduced; this is a "shift and add" algorithm that allows calculation of many various trigonometric functions, such as:
  - $\sqrt{x^2 + y^2}$
  - $\cos\theta$, $\tan\theta$, $\sin\theta$
  - and other functions including divide and logarithmic functions.

THIS SLIDE IS BLANK

Notes:

For more detail and background on the CORDIC algorithm the following material may be useful:

[1] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers" : www.andraka.com/cordic.htm

[2] J. Walther, "The Story of Unified CORDIC", Journal of VLSI Signal Processing, June 2000 : http://www.springerlink.com/content/k21104p108676p0k/

[3] J. Valls et al, "The Use of CORDIC in Software Defined Radios: A Tutorial", IEEE Communications Magazine, September 2006 : http://ieeexplore.ieee.org/iel5/35/36012/01705978.pdf

[4] CORDIC FAQs: http://www.dspguru.com/info/faqs/cordic.htm

[5] Y. H. Hu, "The Quantization Effects of the CORDIC Algorithm", in IEEE Trans. On Signal Processing, Vol 40, No 4, April 1992.

This is nothing "new" in the CORDIC technique. In fact it dates back to 1957 in a paper by an author J. Volder. In the 1950s shift and adds on large physical computers was the limit of technology so CORDIC was of real interest. Also, in the 1970s, with the advent of handheld calculators from Hewlett Packard and other companies, many had an internal CORDIC unit to calculate all of the trigonometric functions (those who recall this time, will remember that taking the tangent of an angle, had a delay of sometimes up to a second while the calculator calculated the result!).

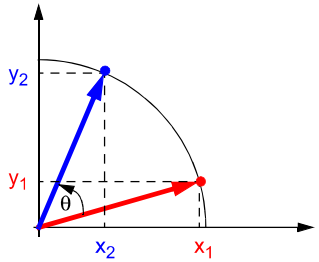
In the 1980s CORDIC was less relevant given the advent of high speed multipliers and general purpose processors with plenty of memory available. However now in the 2000's for FPGAs, CORDIC is definitely a candidiate technique for the calculation of trigonometric functions in DSP applications such as MIMO, beamforming and other adaptive systems.

Cartesian Coordinate Plane Rotations

6.2

- The standard method of rotating a point (x_1, y_1) by θ degrees in the *xy* plane to a point (x_2, y_2) is given by the well known equations:

$$x_2 = x_1\cos\theta - y_1\sin\theta$$
$$y_2 = x_1\sin\theta + y_1\cos\theta$$



- This is variously known as a plane rotation, a vector rotation, or in linear (matrix) algebra, a Givens Rotation.

Pseudo-Rotations

6.3

- By taking out the factor $\cos\theta$ term we can rewrite the equations as:

$$x_2 = x_1\cos\theta - y_1\sin\theta = \cos\theta(x_1 - y_1\tan\theta)$$
$$y_2 = x_1\sin\theta + y_1\cos\theta = \cos\theta(y_1 + x_1\tan\theta)$$

- If we now drop the $\cos\theta$ term then we have a ***pseudo-rotation***:

$$\hat{x}_2 = \cancel{\cos\theta}(x_1 - y_1\tan\theta) = x_1 - y_1\tan\theta$$
$$\hat{y}_2 = \cancel{\cos\theta}(y_1 + x_1\tan\theta) = y_1 + x_1\tan\theta$$

ie. the angle of rotation is correct but the *x* and *y* values are scaled by $\cos^{-1}\theta$ (i.e. both become larger than before as $\cos^{-1}\theta > 1$.

- Note that we have ***NO*** mathematical justification for dropping the $\cos\theta$ term, however later we note it can make the computation of plane rotations more amenable to simple operations.

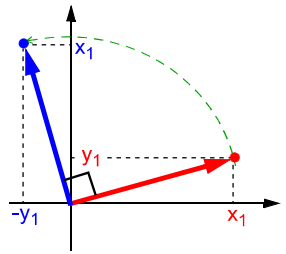
Notes:

This can also be written in a matrix vector form as:

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

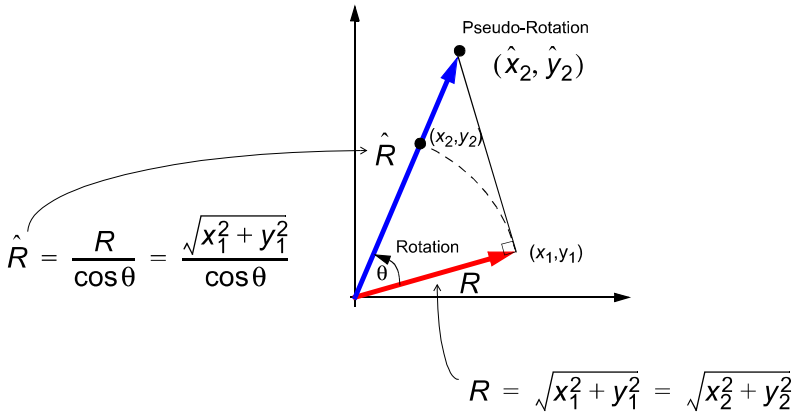
So for example a 90° phase shift would be:

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} -y_1 \\ x_1 \end{bmatrix}$$



Notes:

In the xy plane this is:



Therefore the magnitude of the vector *R* increases by a factor of $1/\cos\theta$ after the pseudo-rotation.

So the resulting point is at the correct angle, but the vector has the wrong magnitude.

The CORDIC Method

6.4

- The key to the CORDIC method is to restrict (pseudo-) rotations to angles of θ, where $\tan\theta^{i} = 2^{-i}$. Therefore in the equation:

$$\hat{x}_2 = x_1 - y_1 \tan\theta = x_1 - y_1 2^{-i}$$

$$\hat{y}_2 = y_1 + x_1 \tan\theta = y_1 + x_1 2^{-i}$$

- The table below shows the rotation angles (to 9 decimal places) that can be used for each iteration (i) of the CORDIC algorithm:

i	θ^{i} (Degrees)	$\tan\theta^{i} = 2^{-i}$
0	45.0	1
1	26.565051177...	0.5
2	14.036243467...	0.25
3	7.125016348...	0.125
4	3.576334374...	0.0625

Rotation Angles and Scaling

6.5

- The first 13 CORDIC rotation angles and cos θ values are:

Iteration (i)	tan(θ)	Angle θ	cos(θ)
0	1	45.0000000000	0.707106781
1	0.5	26.5650511771	0.894427191
2	0.25	14.0362434679	0.970142500
3	0.125	7.1250163489	0.992277877
4	0.0625	3.5763343750	0.998052578
5	0.03125	1.7899106082	0.999512076
6	0.015625	0.8951737102	0.999877952
7	0.0078125	0.4476141709	0.999969484
8	0.00390625	0.2238105004	0.999992371
9	0.001953125	0.1119056771	0.999998093
10	0.000976563	0.0559528919	0.999999523
11	0.000488281	0.0279764526	0.999999881
12	0.000244141	0.0139882271	0.999999997

- A multiply by **1 / [(1/cos(45))*(1/cos(26.56...))*....*(1/cos(0.0139...))]** is required to scale the pseudo-rotated vector back to its true length.

Notes:

At this stage we alter the transform to become an iterative algorithm. We restrict the angles that we are able to rotate by, such that to rotate by an arbitrary θ requires a series of successively smaller rotations at each iteration *i*. As we will later see, the rotation can take place in either direction.

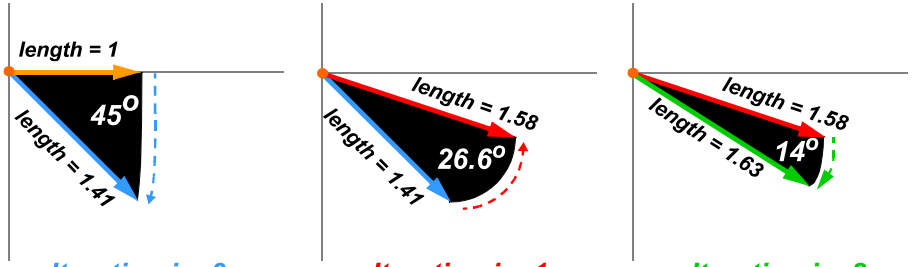
The rotation angles obey the law: $\tan\theta^{(i)} = 2^{-i}$. Obeying this law causes the ***multiplication by the tangent term to become a binary shift***.

The first few iterations take the form:

1st iteration: rotate by 45°; ***2nd iteration: rotate by 26.6°***; ***3rd iteration: rotate by 14°***; etc.

Notice that the length of the vector grows by:

1st iteration:1/cos(45°); ***2nd iteration: 1/cos(26.6°)***; ***3rd iteration: 1/cos(14°)***; etc.



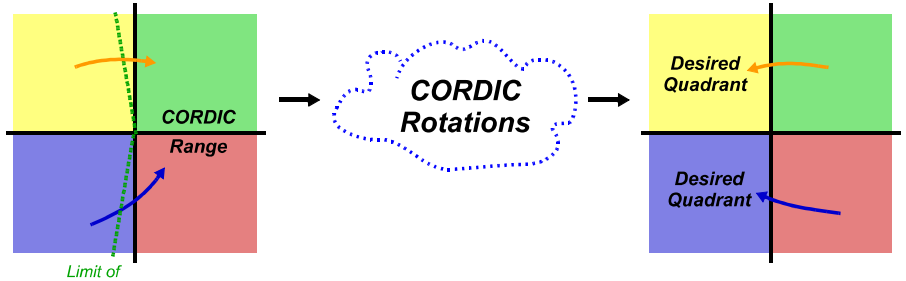
Notes:

In this example, therefore, there are 13 rotations and the vector must be scaled by

$$\frac{1}{K_n} = \frac{1}{(1/\cos(45)) \times (1/\cos(26.565)) \times \ldots \times (1/\cos(0.014))} = \frac{1}{1.6467602} = 0.607252941$$

where K_n is the scaling factor resulting from *n* iterations. The number of bits resolution in the angle is of course relevant to the accuracy of the final result.

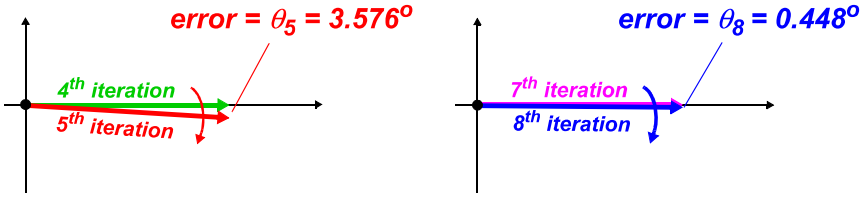
The direction with each rotation takes obviously affects the accumulative angle that is rotated. Arbitrary angles can be rotated in the range $-99.7 \le \theta \le 99.7$, as the sum of all angles obeying the law $\tan\theta^{i} = 2^{-i}$ is 99.7. For angles outside this range, trigonometric identities can be used to convert the desired angle into one within the range. The result is then mapped back into the appropriate quadrant.



Residual Angular Error

6.6

- The maximum error in the final result is equal to the ***final rotation angle***. Therefore, the more iterations performed, the smaller the error.
- Hardware implementation of the CORDIC algorithm requires that ***a fixed number of iterations*** are executed, even if a particular iteration increases the error. ***CORDIC accuracy is therefore limited by the worst case error.***
- The worst case error occurs if the result of the penultimate rotation was exactly correct. The last iteration (which must be performed) rotates away by θ_n.



n = 5 iterations *n = 8 iterations*

Notes:

Of course the numerical precision used to represent the rotations also has an impact. Suppose we stored the rotation values to only 1 decimal place, as shown in the table on the right. Let us compare the effects of rounding to 1 and 10 decimal places, for a desired CORDIC rotation angle of 55°.

10 decimal places	1 decimal place
+45.0000000000	+45.0
+26.5650511771	+26.6
-14.0362434679	-14.0
-7.1250163489	-7.1
+3.5763343750	+3.6
+1.7899106082	+1.8
-0.8951737102	-0.9
+0.4476141709	+0.5
-0.2238105004	-0.2
-0.1119056771	-0.1
+0.0559528919	-0.1
-0.0279764526	-0.0
-0.0139882271	-0.0
Result = 55.000748838900016°	**Result = 55.1°**

i	Angle θ_{10}	θ_1
1	45.0000000000	45.0
2	26.5650511771	26.6
3	14.0362434679	14.0
4	7.1250163489	7.1
5	3.5763343750	3.6
6	1.7899106082	1.8
7	0.8951737102	0.9
8	0.4476141709	0.5
9	0.2238105004	0.2
10	0.1119056771	0.1
11	0.0559528919	0.1
12	0.0279764526	0.0
13	0.0139882271	0.0

iterations 12 and 13: zero rotation!!!

The Scaling Factor

6.7

- The Scaling Factor is a by-product of the pseudo-rotations.
- When simplifying the algorithm to allow pseudo-rotations the cosθ term was omitted.
- Thus outputs $x^{(n)}$, $y^{(n)}$ are scaled by a ***Scaling Factor***, K_n, where:

$$K_n = \prod_n 1/(\cos\theta^{(i)}) = \prod_n (\sqrt{1+2^{(-2i)}})$$

note: the $\cos\theta^{(i)}$ term represents the i^{th} cosθ angle, rather than a power

- If the number of iterations are known, then the scaling factor can be precomputed.
- More importantly, $1/K_n$ can be precomputed and used for post-rotation compensation in order to calculate the true values of $x^{(n)}$ and $y^{(n)}$.

Notes:

To simplify the Givens rotation we removed the cosθ term to allow us to perform pseudo-rotations. However, this simplification has a **side-effect**. The output values $x^{(n)}$ and $y^{(n)}$ are scaled by a factor K_n known as the Scaling Factor where:

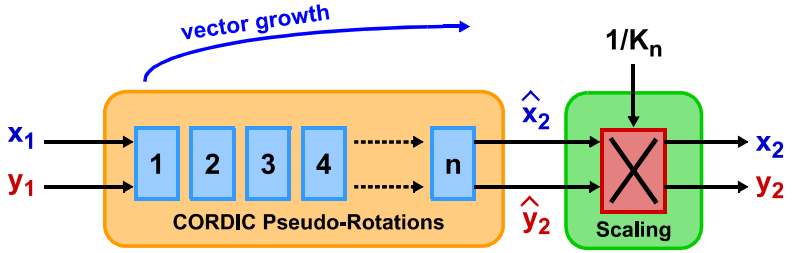
$$K_n = \prod_n 1/(\cos\theta^{(i)}) = \prod_n \left(\sqrt{1+\tan^2\theta^{(i)}}\right) = \prod_n \left(\sqrt{1+2^{(-2i)}}\right)$$

$$K_n \to 1.6476 \text{ as } n \to \infty$$

$$1/K_n \to 0.6073 \text{ as } n \to \infty$$

n = number of iterations

However, if we know the number of iterations that will be performed then we can precompute the value of $1/K_n$ and correct the final values of $x^{(n)}$ and $y^{(n)}$ by multiplying them by this value.



Angle Accumulator

6.8

- The pseudo rotation shown earlier can now be expressed for each iteration as:

$$x^{(i+1)} = x^{(i)} - d_i(2^{-i}y^{(i)})$$
$$y^{(i+1)} = y^{(i)} + d_i(2^{-i}x^{(i)})$$

where d_i = +/- *1*

- At this stage we introduce a third equation called the Angle Accumulator, which is used to keep track of the accumulative angle rotated at each iteration:

$$z^{(i+1)} = z^{(i)} - d_i\theta^{(i)} \qquad \text{(Angle Accumulator)}$$

- The symbol d_i is a decision operator and is used to determine the direction of rotatation.

Shift-Add Algorithm

6.9

- Hence, the original algorithm has now been reduced to an iterative shift-add algorithm for pseudo-rotations of a vector:

$$x^{(i+1)} = x^{(i)} - d_i(2^{-i}y^{(i)})$$
$$y^{(i+1)} = y^{(i)} + d_i(2^{-i}x^{(i)})$$
$$z^{(i+1)} = z^{(i)} - d_i\theta^{(i)}$$

- Thus each iteration requires:
  - 2 shifts
  - 1 table lookup ($\theta^{(i)}$ values)
  - 3 additions

Notes:

At this stage we can now express the equations for each iteration as:

$$x^{(i+1)} = x^{(i)} - d_i(2^{-i}y^{(i)})$$
$$y^{(i+1)} = y^{(i)} + d_i(2^{-i}x^{(i)})$$

where d_i the decision operator is used to give the rotation a clockwise or anticlockwise direction. The conditions of d_i depend on the **mode** of operation which shall be discussed shortly.

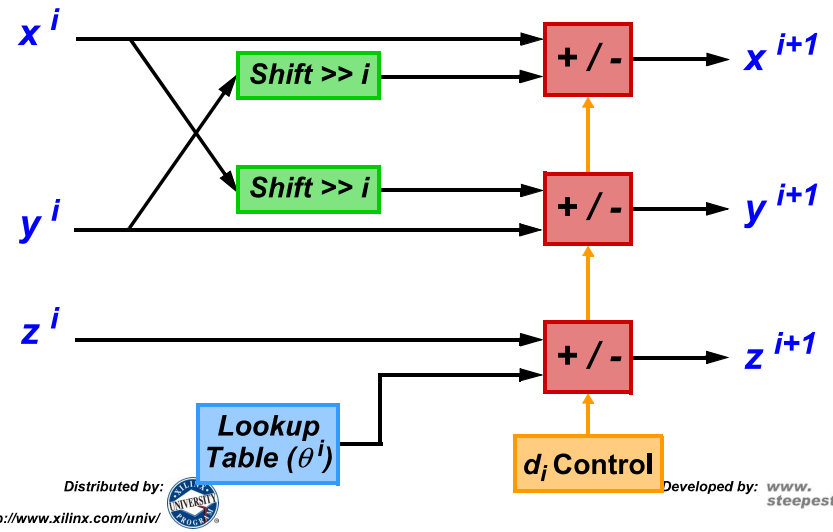
Also, at this stage we introduce a third equation called the **Angle Accumulator** which is used to keep track of the accumulative angle rotated at each iteration:

$$z^{(i+1)} = z^{(i)} - d_i\theta^{(i)}$$

The three equations now represent the CORDIC algorithm for rotations in a Circular Coordinate System. We shall see later that there are other coordinate systems that can be used with the CORDIC method to calculate a greater range of functions.

Notes:

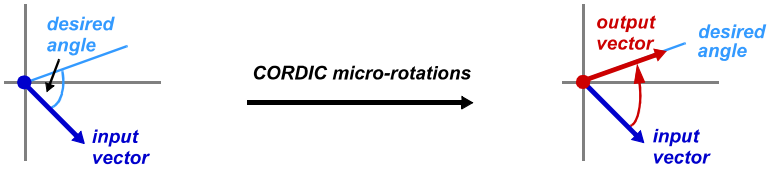
Here, the reason for removing the cos **θ** term earlier becomes clear: with this term removed, the transform is reduced to an iterative shift-add algorithm for pseudo-rotations. We can therefore construct a simple hardware model of a CORDIC rotator for the i^{th} iteration. Strategies for implementing a series of CORDIC rotations will be presented later in the notes. First, we will consider the control of rotation direction, and the two modes of CORDIC operation.



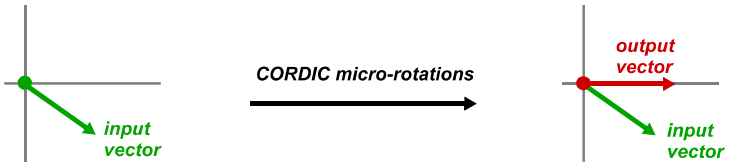
CORDIC Modes

6.10

- The CORDIC algorithm can be operated in two modes:
  - ***Rotation mode*** - the input vector is rotated by a desired angle



  - ***Vectoring mode*** - the input vector is rotated onto the *x* axis

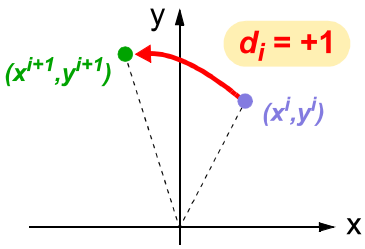


- The mode of operation dictates the condition for the control operator d_i

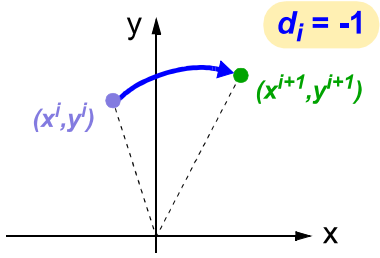
Rotation Mode: Decision Variable

6.11

- In ***Rotation Mode***, the direction of each rotation depends on the sign of z^i, the angle input to the cell:
  - d_i = **+1** for **+ve** z^i ***anti-clockwise*** rotation
  - d_i = **-1** for **-ve** z^i ***clockwise*** rotation



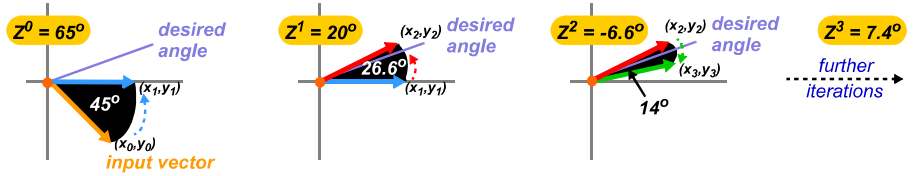
$$x^{i+1} = x^i - d_i(2^{-i}y^i) = x^i - 2^{-i}y^i$$
$$y^{i+1} = y^i + d_i(2^{-i}x^i) = y^i + 2^{-i}x^i$$



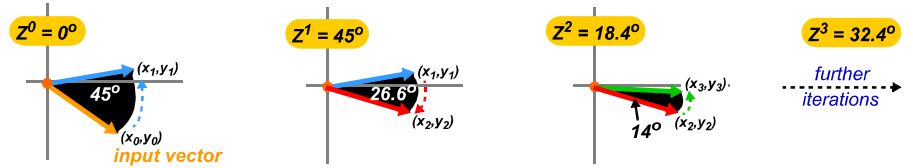
$$x^{i+1} = x^i - d_i(2^{-i}y^i) = x^i + 2^{-i}y^i$$
$$y^{i+1} = y^i + d_i(2^{-i}x^i) = y^i - 2^{-i}x^i$$

Notes:

In ***rotation*** mode, the ***z*** variable is initialised with the desired angle of rotation. CORDIC iterations subsequently drive the vector towards this angle, through a series of progressively smaller micro-rotations as described earlier in the notes. The direction of rotation at the i^{th} iteration is determined by comparing the ***z*** input with zero. The z value converges towards 0 as the rotation converges towards the desired angle. (Note that in these diagrams, we ignore the vector growth.)



In ***vectoring*** mode, the direction of rotation is determined through the y input: if y > 0 then the vector is above the x axis and must be rotated downwards; if y < 0 then the vector is below the x axis and must be rotated upwards. The final value of z is the accumulative angle of rotation (onto the x axis), which tends towards the original angle of the vector as the number of iterations tends towards infinity.



Distributed by: http://www.xilinx.com/univ/

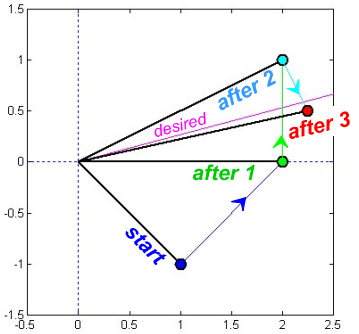
Developed by: www.steepestascent.com

Notes:

Therefore, if an overall anticlockwise rotation is required, then the angle supplied as the Z input to the CORDIC processor should be positive; if a clockwise rotation is required, the angle should be negative. During the series of pseudo-rotations, the z angle will likely flip between positive and negative values as the CORDIC iterates towards the desired angle.

For example, suppose that the initial position of the vector is (1,-1), and the supplied angle z_0 is **+60°**. The angle is positive, so the first iteration (i = 0), is 45° in the ***anti-clockwise*** direction. The z value is now (+60°-45° = **+15°**). As a result of the angle still be positive, a further ***anti-clockwise*** rotation is made during the next iteration (i = 1), this time of 26.6° and resulting in a new z angle of (+15°-26.6° = **-11.6°**). As the angle is now negative, the third iteration (i = 2) is in the ***clockwise*** direction, resulting in a z angle of (-11.6°+14.0° = **+2.4°**) after three iterations. Note from the plot that the vector grows at each stage.

	x	y	z	
start	1	1	+60°	i = 0 rotate +45°
after 1 iteration	2	0	+15°	i = 1 rotate +26.6°
after 2 iterations	2	1	-11.6°	i = 2 rotate -14.0°
after 3 iterations	2.25	0.5	+2.4°	



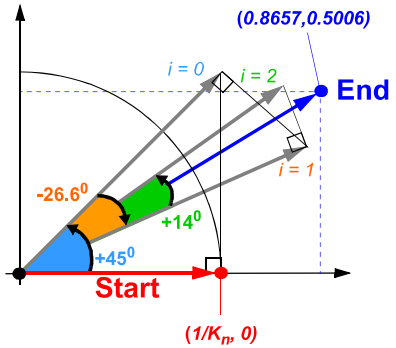
Distributed by: http://www.xilinx.com/univ/

Developed by: www.steepestascent.com

Rotation Mode: Calculating Sine & Cosine

- Compute $\cos z^{(0)}$, $\sin z^{(0)}$ by starting with $x^{(0)} = 1/K_n$ and $y^{(0)} = 0$
- Example: calculate $\sin z^{(0)}$, $\cos z^{(0)}$ where $z^{(0)} = 30^o$...

i	d_i	$\theta^{(i)}$	$z^{(i)}$	$y^{(i)}$	$x^{(i)}$
0	+1	45	+30	0	0.6073
1	-1	26.6	-15	0.6073	0.6073
2	+1	14	+11.6	0.3036	0.9109
3	-1	7.1	-2.4	0.5313	0.8350
4	+1	3.6	+4.7	0.4270	0.9014
5	+1	1.8	+1.1	0.4833	0.8747
6	-1	0.9	-0.7	0.5106	0.8596
7	+1	0.4	+0.2	0.4972	0.8676
8	-1	0.2	-0.2	0.5040	0.8637
9	+1	0.1	+0	0.5006	0.8657



Notes:

In Rotation Mode the decision operator d_i obeys the condition:

$$d_i = sign(z^{(i)})$$

Hence for positive values of the input angle, z, the resulting rotation is in the

After n iterations we have:

$$x^{(n)} = K_n(x^{(0)}\cos z^{(0)} - y^{(0)}\sin z^{(0)})$$

$$y^{(n)} = K_n(y^{(0)}\cos z^{(0)} - y^{(0)}\sin z^{(0)})$$

$$z^{(n)} = 0$$

Thus, to rotate a vector from the x axis to a new desired angle, we input $x^{(0)}$ and $z^{(0)}$ (setting $y^{(0)} = 0$) and then drive $z^{(0)}$ towards 0.

In the example considered here, where we rotated a vector of magnitude $1/K_n$ by 30^0, notice from the table that:

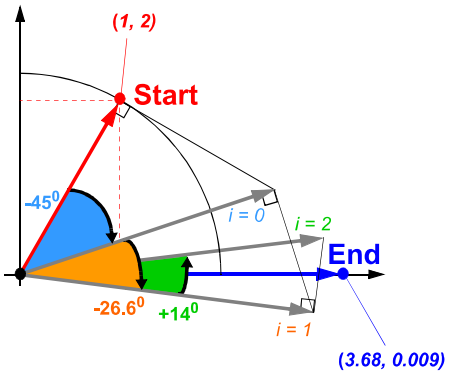
$$x^{(9)} = 0.5006 \approx \sin(30^0)$$

$$y^{(9)} = 0.8657 \approx \cos(30^0)$$

Vectoring Mode

- In Vectoring Mode choose: $d_i = -sign(x^{(i)}y^{(i)}) \Rightarrow y^{(i)} \rightarrow 0$
- Can compute $\tan^{-1} y^{(0)}$ by setting $x^{(0)} = 1$ and $z^{(0)} = 0$...

i	$z^{(i)}$	θ^i	$y^{(i)}$	$x^{(i)}$
0	0	45	2	1
1	45	26.6	1	3.00
2	71.6	14	-0.5	3.50
3	57.6	7.1	0.375	3.63
4	64.7	3.6	-0.078	3.67
5	61.1	1.8	0.151	3.68
6	62.9	0.9	0.039	3.68
7	63.8	0.4	-0.019	3.68
8	63.4	0.2	0.009	3.68



Notes:

In Vectoring Mode the decision operator d_i obeys the condition:

$$d_i = -sign(x^{(i)}y^{(i)})$$

After n iterations we have:

$$x^{(n)} = K_n\left(\sqrt{(x^{(0)})^2 + (y^{(0)})^2}\right)$$

$$y^{(n)} = 0$$

$$z^{(n)} = z^{(0)} + \tan^{-1}\left(\frac{y^{(0)}}{x^{(0)}}\right)$$

Thus, we input $x^{(0)}$ and $y^{(0)}$ (setting $z^{(0)} = 0$) and then drive $y^{(0)}$ towards 0. The final value of z corresponds to the accumulated angle of rotation.

In the given example, we calculated $\tan^{-1}(y^{(0)}/x^{(0)})$ where $y^{(0)} = 2$ and $x^{(0)} = 1$. Notice that after 8 iterations,

$$z^{(8)} = 63.4 \approx \tan^{-1}(2)$$

Circular Coordinate System

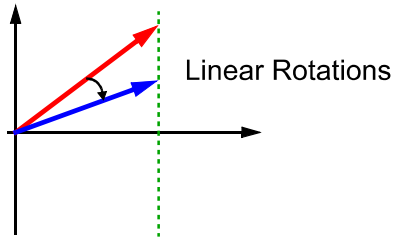
- So far only pseudo-rotations in a Circular Coordinate System have been considered.
- Thus, the following functions can be computed:

Coordinate System	Rotation Mode $z^{(i)} \rightarrow 0$; $d_i = \text{sign}(z^{(i)})$	Vectoring Mode $y^{(i)} \rightarrow 0$; $d_i = -\text{sign}(x^{(i)}y^{(i)})$
Circular	x, y, z → CORDIC → K(x.cos z - y.sin z), K(y.cos z + x.sin z), 0 For cos z & sin z, set x = 1/K, y = 0	x, y, z → CORDIC → $K(x^2+y^2)^{1/2}$, 0, $z + \tan^{-1}(y/x)$ For $\tan^{-1}$ y, set x = 1, z = 0

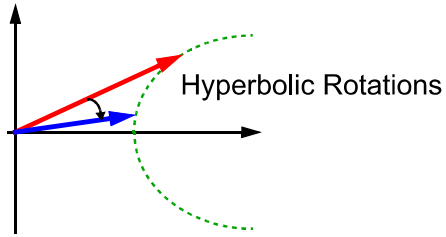
- However, more functions can be computed if we use other coordinate systems.

Other Coordinate Systems

- Linear Coordinate System

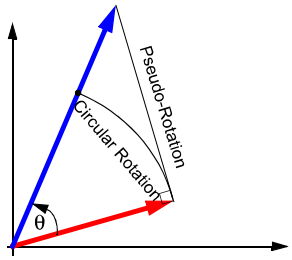


- Hyperbolic Coordinate System



Notes:

With rotations in a Circular Coordinate System we are limited to the number of functions that can be calculated.



However, we shall see that by considering rotations in other coordinate systems we can calculate more functions directly, such as multiplications and divides, which allow us to calculate even more functions *indirectly*.

Notes:

The advantage of using other coordinate systems with the CORDIC algorithm is that it allows more functions to be calculated. The drawback is that the system becomes more complex. The set of rotation angles used for the Circular Coordinate System are no longer valid when using the CORDIC algorithm with a Linear or Hyperbolic system. Hence, two other sets of angles are used for rotations made in these systems.

It is also notable that, when using the CORDIC algorithm for Hyperbolic rotations, the scaling factor K is different from that used for Circular rotations.

The Hyperbolic scaling factor is denoted K^* and is calculated using the equation:

$$K^*_n = \prod_n \left(\sqrt{1 - 2^{(-2i)}}\right)$$

$$K^*_n \rightarrow 0.82816 \text{ as } n \rightarrow \infty$$

$$1/K^*_n \rightarrow 1.20750 \text{ as } n \rightarrow \infty$$

n = number of iterations

We shall see shortly that the CORDIC equations can be generalised for the 3 coordinate systems and that this involves the introduction of two new variables to the equations. One of these new variables ($e^{(i)}$) represents the set of angles used to represent rotations in the appropriate coordinate system.

Generalised CORDIC Equations

6.16

- With the addition of two other Coordinate Systems, the CORDIC equations can now be generalised to accommodate all three systems:

$$x^{(i+1)} = x^{(i)} - \mu d_i (2^{-i} y^{(i)})$$

$$y^{(i+1)} = y^{(i)} + d_i (2^{-i} x^{(i)})$$

$$z^{(i+1)} = z^{(i)} - d_i e^{(i)}$$

- ***Circular Rotations:*** $\mu = 1, e^{(i)} = \tan^{-1} 2^{-i}$
- ***Linear Rotations:*** $\mu = 0, e^{(i)} = 2^{-i}$
- ***Hyperbolic Rotations:*** $\mu = -1, e^{(i)} = \tanh^{-1} 2^{-i}$

Summary of CORDIC Functions

6.17

	Rotation Mode: d_i=sign($z^{(i)}$); $z^{(i)} \rightarrow 0$	***Vectoring Mode:*** d_i=-sign($x^{(i)}y^{(i)}$); $y^{(i)} \rightarrow 0$
Circular $\mu = 1$ $e^{(i)} = \tan^{-1}2^{-i}$	x → C O R D I C → K(x.cos z - y.sin z) y → C O R D I C → K(y.cos z + x.sin z) z → C O R D I C → 0 For cos z & sin z, set x = 1/K, y = 0	x → C O R D I C → $K(x^2 + y^2)^{1/2}$ y → C O R D I C → 0 z → C O R D I C → $z + \tan^{-1}(y/x)$ For $\tan^{-1}$ y, set x = 1, z = 0
Linear $\mu = 0$ $e^{(i)} = 2^{-i}$	x → C O R D I C → x y → C O R D I C → y + (x.z) z → C O R D I C → 0 For multiplication, set y = 0	x → C O R D I C → x y → C O R D I C → 0 z → C O R D I C → z + (y/x) For division, set z = 0
Hyperbolic $\mu = -1$ $e^{(i)} = \tanh^{-1}2^{-i}$	x → C O R D I C → K^*(x.cosh z - y.sinh z) y → C O R D I C → K^*(y.cosh z + x.sinh z) z → C O R D I C → 0 For cosh z & sinh z, set x = 1/K^*, y = 0	x → C O R D I C → $K^*(x^2 - y^2)^{1/2}$ y → C O R D I C → 0 z → C O R D I C → $z + \tanh^{-1}(y/x)$ For $\tanh^{-1}$y, set x = 1, z = 0

Notes:

Convergence is guaranteed for Circular CORDIC using angles in range $-99.7 \leq z \leq 99.7$, and as mentioned previously, angles outside this range can be catered for using standard trigonometric identities.

The linear and hyperbolic co-ordinate systems are different in a number of respects:

- Shift sequence
- Angles of convergence*
- Scaling factor

* In the case of the Hyperbolic co-ordinate system, convergence is only achieved if certain of the elemental rotations are repeated {i = 4, 13, 40....}.

Co-ordinate System	μ	Scale Factor ($n \rightarrow \infty$)	Convergence (max. angle as $n \rightarrow \infty$)	Shift Sequence
Circular	+1	1.64676	99.7°	0,1,2,3,4, ... *n-1*
Linear	0	1.0	57.3°	1,2,3,4,5, ... *n*
Hyperbolic	-1	0.83816	64.7°	1,2,3,4,4,5....

J. S. Walther, "A Unified Algorithm for Elementary Functions" Spring Joint Computer Conference, 1971, pp. 379-385.

Notes:

Although the CORDIC algorithms can only compute a limited number of functions directly, there are many more that can be achieved indirectly:

$$\tan z = \frac{\sin z}{\cos z}$$

$$\tanh z = \frac{\sinh z}{\cosh z}$$

$$\ln w = 2\tanh^{-1}\left|\frac{w-1}{w+1}\right|$$

$$e^z = \sinh z + \cosh z$$

$$w^t = e^{t \ln w}$$

$$\tan^{-1} w = \tan^{-1}\frac{\sqrt{1-w^2}}{w}$$

$$\sin^{-1} w = \tan^{-1}\frac{w}{\sqrt{1-w^2}}$$

$$\cosh^{-1} w = \ln\left(w + \sqrt{1-w^2}\right)$$

$$\sqrt{w} = \sqrt{\left(w+\frac{1}{4}\right)^2 - \left(w-\frac{1}{4}\right)^2}$$

Example: calculate ***tan z***...

First, directly calculate ***cos z*** and ***sin z*** using the CORDIC system in Circular Rotation Mode.

Second, feed these values back into the system to divide the latter by the former using Linear Vectoring Mode, which will yield ***tan z***.

FPGA Implementation

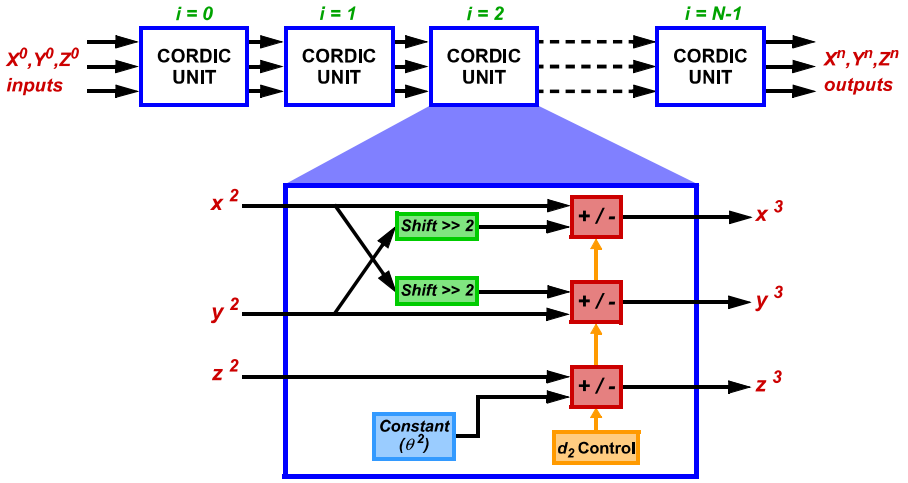
6.18

- The ideal CORDIC architecture depends on speed vs area tradeoffs in the intended application.
- Alternative implementation options will be presented, and the cost and performance trade-offs highlighted.
  - ***Rolled*** - all CORDIC iterations performed using a single cell.
  - ***Unrolled*** - an array of cells, one for each iteration.
  - ***Unrolled, pipelined*** - retimed to improve performance.
- We will consider the key problem of ***shifting by i*** in detail.
  - A direct translation of the CORDIC equations is an iterative bit-parallel design, however...
  - Bit-parallel variable shift shifters do not map well into FPGAs... Resulting in a large and slow implementation.

Unrolled Architecture

6.19

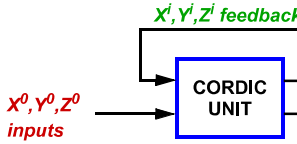
- In the unrolled design, each cell has dedicated shifters with a fixed shift. Recall our earlier mapping of CORDIC equations to hardware...



Notes:

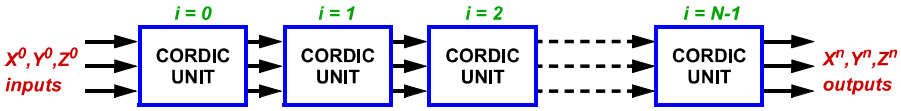
At a high level, these three implementation options can be represented as follows:

1) Rolled

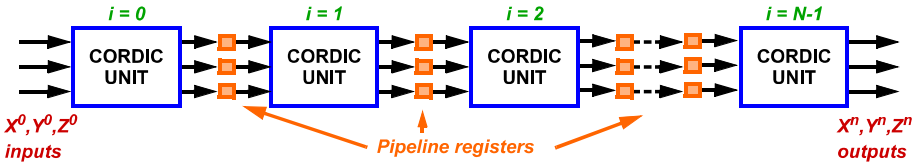


Note that the CORDIC unit in the rolled architecture is different compared to the unrolled designs: it must keep track of the iteration number at each clock cycle and synthesise the correct shift. As the CORDIC unit is shared N times, it runs at 1/N of the rate.

2) Unrolled



3) Unrolled and Pipelined

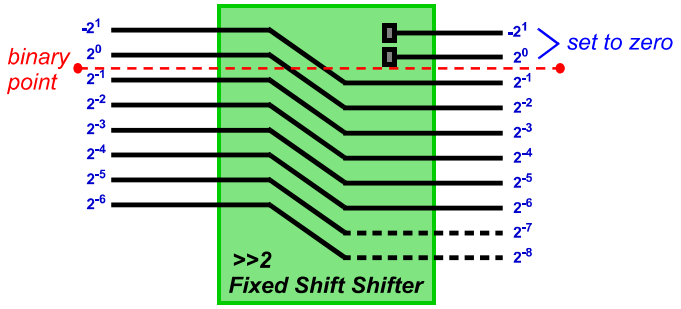


Distributed by: http://www.xilinx.com/univ/

Developed by: www.steepestascent.com

Notes:

The shifts within each cell are always the same, which implies a very simple hardware implementation. In fact, a fixed shift can be realised using only routing resources, as shown below. The least significant bits can be discarded if required.



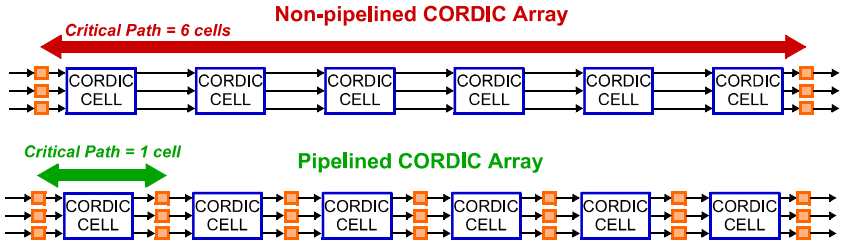
Distributed by: http://www.xilinx.com/univ/

Developed by: www.steepestascent.com

Unrolled and Pipelined Architecture

6.20

- The critical path within the unrolled design is very long...
  - There are no registers within the CORDIC cells
  - The critical path is ***N x 1 cell delay***
- The critical path can be reduced to the delay of one cell, if pipeline registers are inserted between cells. For example with 6 cells...



- This speeds up the design at modest hardware cost (a few flip flops), but also introduces a latency.

Notes:

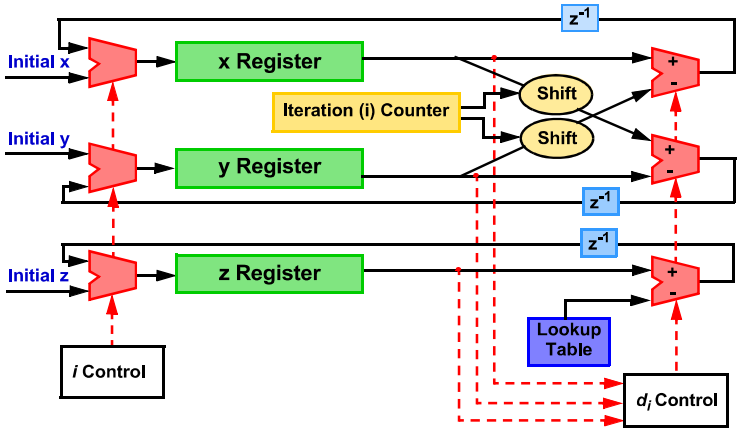
These examples assume that both the inputs and outputs are registered; if not, the critical path is longer than the CORDIC processor, and also includes delays from components preceding or succeeding it.

For an *N*-cell CORDIC array, therefore, it should be straightforward to notice that introducing pipelining adds an additional *N* clock cycles of latency.

Rolled Architecture

6.21

- In the previous examples, a dedicated cell has been used for each iteration. It may be desirable to lower the hardware cost of the CORDIC processor by time-sharing one cell to perform all iterations.
- The functionality of the cell now depends on the iteration number, ***i***.



Notes:

The Rolled CORDIC is clocked for ***N*** iterations in order to produce the final results in the X, Y, and Z registers. Therefore, it throughput is decreased by ***1/N***, and it also features a latency as in the pipelined CORDIC design.

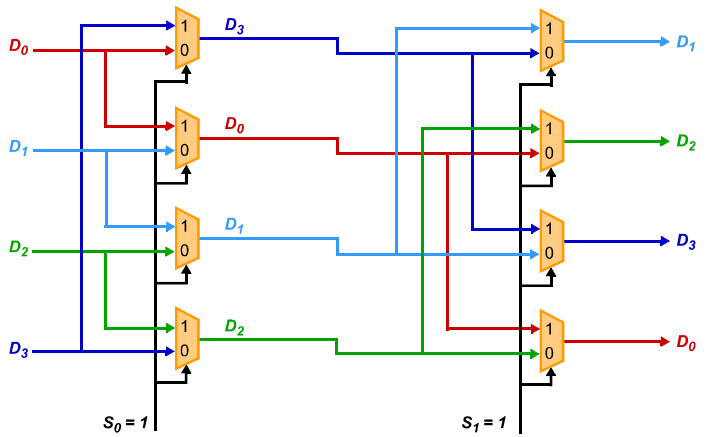
The main increase in complexity is due to the shifters. In the unrolled architecture, a fixed shift is required for each cell, and this can be created entirely using routing resources. However, the rolled design requires a variable shift... recall from the set of CORDIC equations that we must multiply by 2^{-i} within each cell, which is implicitly a shift by ***i*** bits, and that the single cell must cater for all values of ***i***.

This leads to the consideration of a ***barrel shifter*** for incorporation into the rolled architecture. We would like the shift operation to take place with as little latency as possible, to maximise throughput.

Barrel Shifter

6.22

- Barrel shifters are commonly constructed using multiplexers. Consider this simple 4-bit barrel shifter as an example:



- The S_1S_0 word activates the multiplexers to create the desired shift.

Vector Magnitude Calculation

6.23

- CORDIC can be used to compute the magnitude of a vector, as discussed previously, i.e.

$$|v| = \sqrt{x^2 + y^2}$$

- It is important to consider the accuracy of the CORDIC algorithm when computing $|v|$.
- We must choose appropriate parameters to provide a desired accuracy, in particular:
  - Number of iterations (***n***)
  - Number of bits in the datapath (***b***)
- Both of these factors affect hardware cost and performance. In the examples which follow, an unrolled design is assumed.
  - More accurate = more expensive!!

Notes:

In this example the input word is $S_1S_0 = 11$, synthesising a 3-bit shift.

The first column of multiplexers are controlled by the S_0 bit. If $S_0 = 0$, the inputs pass directly through to the outputs, i.e. the shift is 0 bits. If $S_0 = 1$, the output from the bottom multiplexer is D_2 - a shift of 1 bit. The other multiplexers behave similarly, with D_3 appearing at the output of the top multiplexer, i.e. wrapping round.

The second column of multiplexers is controlled by the S_1 bit. If $S_1 = 0$, a shift is not applied. If $S_1 = 1$, a shift of 2 bits is synthesised by this column. For example, the output from the bottom multiplexer is D_0, a further shift of 2 bits, and a total shift of 3 bits. In this case, the D_1 and D_2 inputs wrap round.

This architecture is scalable, so if for example we required an 8-bit barrel shifter, we would need an extra column of multiplexers, and the array would also be extended downwards.

Notice that this design is entirely combinatorial, thus offering a variable shift with zero clock cycles delay.

Notes:

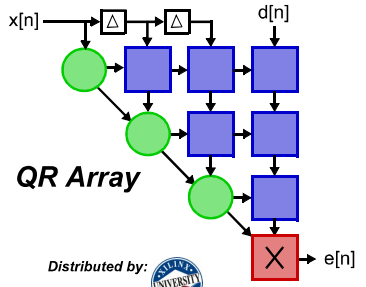
One significant application of CORDIC vector magnitude calculations is the QR-algorithm, which is increasingly used in adaptive applications. The hardware implementation of QR is a triangular array, and requires that an incoming vector is subject to a Given's rotation, as described in Slide 6.2, i.e.

$$x_{new} = x\cos\theta - y\sin\theta$$

$$y_{new} = x\sin\theta + y\cos\theta$$

These rotations are performed by a subset of the cells within a QR array (internal cells or Given's Rotators). First of all, the angle of rotation is computed in terms of cosθ and sinθ, by boundary cells (also known as Given's Generators). The boundary cells use CORDIC processors to generate cosθ and sinθ as follows:

$$\cos\theta = \frac{x}{\sqrt{x^2 + y^2}} \qquad \sin\theta = \frac{y}{\sqrt{x^2 + y^2}}$$



Given's Generators - Compute values of sinθ and cosθ and pass these along the row to the right, allowing the Given's Rotators to rotate the incoming vectors by the same angle (θ).

Given's Rotators - These cells rotate the incoming vectors (vertical signals) by θ, by multiplying by sinθ and cosθ.

NOTE: **x[n]**, **d[n]** and **e[n]** correspond respectively to the input, disturbance and error signals, as commonly defined in adaptive systems.

Computing The Magnitude Of A Vector

6.24

- To compute the magnitude of a vector, a Circular coordinate system must be used with Vectoring mode:

Coordinate System	Rotation Mode $z^{(i)} \rightarrow 0$; $d_i = \text{sign}(z^{(i)})$	Vectoring Mode $y^{(i)} \rightarrow 0$; $d_i = -\text{sign}(x^{(i)}y^{(i)})$
Circular	x → CORDIC → K(x.cos z - y.sin z) y → CORDIC → K(y.cos z + x.sin z) z → CORDIC → 0 For cos z & sin z, set x = 1/K, y = 0	x → CORDIC → $K(x^2+y^2)^{1/2}$ y → CORDIC → 0 z → CORDIC → z + $\tan^{-1}(y/x)$ For $\tan^{-1}$ z, set x = 1, z = 0

- The 'K' value is the scaling factor, which can be removed by multiplying the result by 1/K.

Notes:

To compute the magnitude of a vector using CORDIC, circular rotations must be used with Vectoring mode. The *x* output will then generate the following result:

$$x = K\sqrt{x^2 + y^2}$$

Clearly the scaling factor *K* must be removed. This can be achieved by multiplying the *x* output by *1/K*. The value of *K* is dependent on the number of iterations which is known before hand and thus can be precomputed according to:

$$K(n) = \prod_{i=0}^{n-1} k(i) = \prod_{i=0}^{n-1} \sqrt{1 + 2^{(-2i)}}$$

Simplifying The Equations

6.25

- The Generalised CORDIC equations are:

$$x^{(i+1)} = x^{(i)} - \mu d_i(2^{-i}y^{(i)})$$
$$y^{(i+1)} = y^{(i)} + d_i(2^{-i}x^{(i)})$$
$$z^{(i+1)} = z^{(i)} - d_i e^{(i)}$$

- which, for vector magnitude calculations can be simplified to:

$$x^{(i+1)} = x^{(i)} - d_i(2^{-i}y^{(i)})$$
$$y^{(i+1)} = y^{(i)} + d_i(2^{-i}x^{(i)})$$

- The angle accumulator can be neglected when computing the magnitude of a vector. Also, $\mu = 1$ for a Circular coordinate system.

Notes:

The full set of Generalised CORDIC equations are:

$$x^{(i+1)} = x^{(i)} - \mu d_i(2^{-i}y^{(i)})$$
$$y^{(i+1)} = y^{(i)} + d_i(2^{-i}x^{(i)})$$
$$z^{(i+1)} = z^{(i)} - d_i e^{(i)}$$

where,

- Circular Rotations: $\mu = 1$, $e^{(i)} = \tan^{-1}2^{-i}$
- Linear Rotations: $\mu = 0$, $e^{(i)} = 2^{-i}$
- Hyperbolic Rotations: $\mu = -1$, $e^{(i)} = \tanh^{-1}2^{-i}$

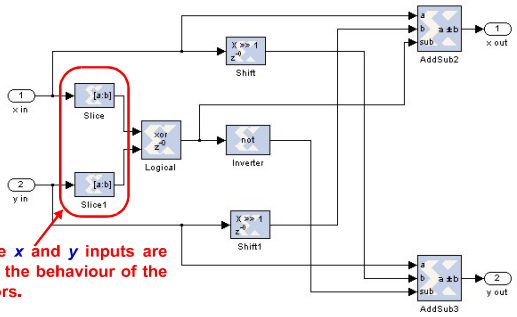
When using Vectoring mode, $d_i = -sign(x^{(i)}y^{(i)})$. The *x* equation is the one that will generate the magnitude of the vector and it is only dependent on the *y* equation. Hence, the *z* equation (angle accumulator) can be ignored. This leaves only:

$$x^{(i+1)} = x^{(i)} - d_i(2^{-i}y^{(i)})$$
$$y^{(i+1)} = y^{(i)} + d_i(2^{-i}x^{(i)})$$

Hardware Requirements

6.26

- The hardware required to implement a ***single iteration*** for a vector magnitude calculation can be seen below:



- In this example, ***i = 3***, therefore...
  - If (***x*** xor ***y***) is ***positive***, $X = x - 2^{-3}y$ and $Y = y + 2^{-3}x$
  - If (***x*** xor ***y***) is ***negative***, $X = x + 2^{-3}y$ and $Y = y - 2^{-3}x$

How Many Iterations?

6.27

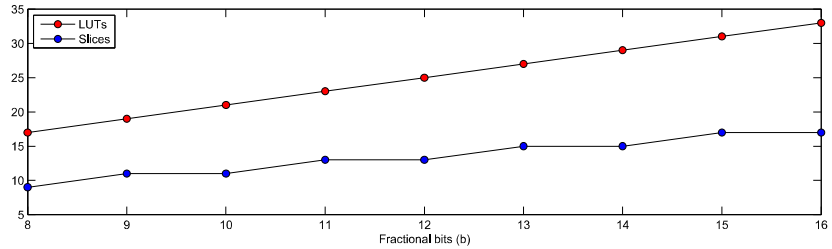
- To achieve a desired accuracy, two questions need to be answered:
  - ***How many iterations are required?***
  - ***How wide do the data paths need to be?***
- Yu Hen Hu (see references under Slide 6.1) developed an algorithm for answering these questions based on the Overall Quantization Error (***OQE***) experienced during the CORDIC iterations.
- Having determined the OQE for a certain datapath width / iterations combination, the number of effective fractional bits can be computed.
- Tables of effective fractional bits can be formed for different combinations of fractional bits (***b***) and iterations (***n***).
- The designer can therefore choose a set of parameters which provides fulfils the computational accuracy requirements with the minimum hardware cost.

Notes:

Within a single cell, therefore, the only factor which affects hardware cost is the width of the data signals. Furthermore, we have established that within an unrolled design, the fixed shift shifters effectively consume no resources. Hence the adders are of prime interest.

With ***b*** bits in the data path, what is the cost of implementing a CORDIC cell? This table shows slice count against data path width. Recall that each bit added requires 1 LUT, and there are 2 LUTs per slice in most FPGA architectures.

	b = 8	b = 9	b = 10	b = 11	b = 12	b = 13	b = 14	b = 15	b = 16
Cost of 1 cell (LUTs)	17	19	21	23	25	27	29	31	33
Cost of 1 cell (Slices)	9	11	11	13	13	15	15	17	17

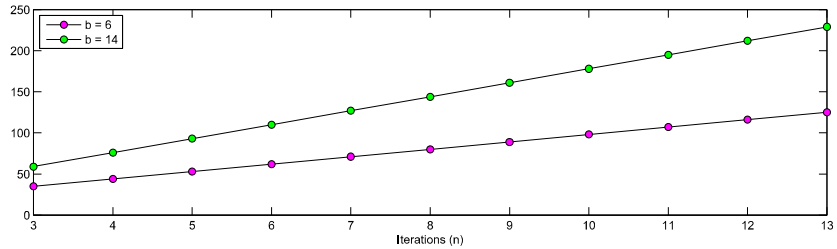


Notes:

To understand the effect of increasing the number of iterations on hardware cost, let us consider the cases where there are 6 fractional bits and 14 fractional bits in the data path. The cost (in slices) of an array of CORDIC cells is summarised by the table below. This demonstrates the effect on hardware cost of increasing the number of iterations (***n***)..

	n = 3	n = 4	n = 5	n = 6	n = 7	n = 8	n = 9	n = 10	n = 11	n = 12	n = 13
Cost of cell array (slices) with *b = 6*	35	44	53	62	71	80	89	98	107	116	125
Cost of cell array (slices) with *b = 14*	59	76	93	110	127	144	161	178	195	212	229

With ***n = 4, b = 6***, the cost is 44 slices. If we increase the number of fractional bits by 8, (***n = 4, b = 14***) the cost becomes 76 slices. Compare this with increasing the number of iterations by 8... if we choose ***n = 12, b = 6***, 116 slices are required.



Determining the OQE

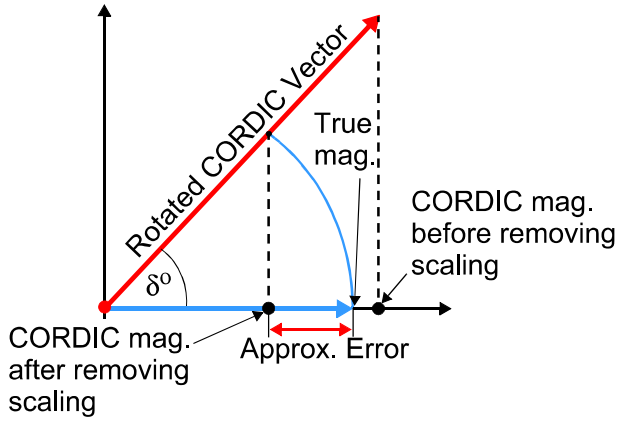
6.28

- Yu Hen Hu determined that the OQE comprises two parts:
  - ***The Approximation Error*** - the error due to the quantised representation of a CORDIC rotation angle by finite numbers of elementary angles.
  - ***The Rounding Error*** - due to the finite precision arithmetic used in a practical implementation.
- Both of these errors can be defined in terms of:
  - The number of iterations (***n***).
  - The number of fractional bits in data paths (***b***).
  - The magnitude of the largest vector ($|v(0)|$).
- The OQE is found by summing the approximation and rounding errors.

The Approximation Error (i)

6.29

- When using Vectoring mode, the aim is to drive the vector onto the x-axis.
- However, due to finite rotations there is often a small angle δ left, and this causes the Approximation Error.



Notes:

As the full set of micro-rotations is ALWAYS performed, errors are based on the very worst case in order to guarantee a certain level of numerical precision.

To illustrate this point, consider the errors encountered after each iteration (post-scaling), when attempting to find the magnitude of three random vectors. Note that errors are calculated by comparing the CORDIC output to a floating point reference magnitude, and that the data path contains 19 fractional bits in each case.

x^0 = 0.164459, y^0 = 0.099121 Floating Point mag. = 0.192021	n = 4	n = 5	n = 6	n = 7	n = 8	n = 9	n = 10	n = 11
Calculated Magnitude	0.190926	0.191750	0.191887	0.191879	0.191887	0.191887	0.191887	0.191887
Error	1.095e-3	271.1e-6	133.7e-6	141.3e-6	133.7e-6	133.7e-6	133.7e-6	133.7e-6

x^0 = 0.068482, y^0 = -0.067859 Floating Point mag. = 0.096408	n = 4	n = 5	n = 6	n = 7	n = 8	n = 9	n = 10	n = 11
Calculated Magnitude	0.095924	0.096283	0.096321	0.096313	0.096321	0.096321	0.096329	0.096329
Error	484.3e-6	125.7e-6	87.56e-6	95.19e-6	87.56e-6	87.56e-6	79.93e-6	79.93e-6

x^0 = -0.030576, y^0 = 0.485885 Floating Point mag. = 0.486845	n = 4	n = 5	n = 6	n = 7	n = 8	n = 9	n = 10	n = 11
Calculated Magnitude	0.483887	0.482823	0.486778	0.486847	0.486832	0.486839	0.486847	0.486847
Error	2.958e-3	563.0e-3	67.04e-6	-1.622e-6	13.64e-6	6.007e-6	-1.622e-6	-1.622e-6

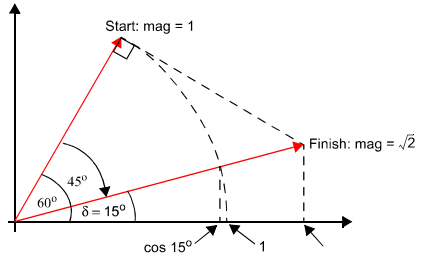
For example, although the average error experienced in these three trials for $n = 6$, $b = 19$ is $96.1e^{-5}$, the worst case error is actually larger than this. As we are forced to use all iterations (even when they make the error bigger than at the previous iteration!), we must choose parameters based on the worst case error in order to provide the necessary precision for all input values. Notice that in the third example, the error was smaller after 7 iterations than after 8. We will now go on to discuss the two components of the OQE.

Notes:

The figure below shows an example where only 1 iteration is performed. The vector, which has a magnitude of 1, is initially at 60°. The first iteration takes the vector through 45° thus leaving an Angle Quantization Error of $\delta = 15^o$. With only 1 iteration the scaling factor K is equal to:

$$K(1) = \prod_{i=0}^{0} \sqrt{1 + 2^{(-2i)}} = \sqrt{2}$$

The magnitude of the rotated vector is now $1 \times \sqrt{2}$, due to vector growth. The value given by the x equation (below) becomes $\sqrt{2}\cos 15^o$. Once divided by the scaling factor, we obtain the "true" quantised magnitude, which is cos 15° in this case. However, cos 15^0 = 0.966, and not 1! Hence an error is introduced.

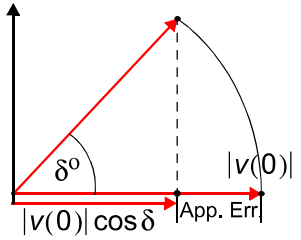


Correspondingly, the x equation is given below:

$$x^{(i+1)} = x^{(i)} - d_i(2^{-i}y^{(i)}) = \cos 60^o + \sin 60^o = \sqrt{2}\cos 15^o$$

The Approximation Error (ii)

- To compute an upper bound for the Approximation Error, an upper bound for δ must be found.
- Yu Hen Hu proposed that: $\delta \le a(n-1) = \text{atan}(2^{-n+1})$.
- Where $a(n-1)$ is the final rotation angle that is made.
- Looking at the diagram below, clearly the Approximation Error is:



$$App.\ Err. = |v(0)| - |v(0)|\cos(\text{atan}(2^{-n+1}))$$

Notes:

For more information on the OQE...

Y. H. Hu, "The Quantization Effects of the CORDIC Algorithm", in IEEE Trans. On Signal Processing, Vol 40, No 4, April 1992

The Rounding Error

- Yu Hen Hu derived the Rounding error to be:

$$Rounding\ Error = 2^{-b-0.5}\left[\frac{G(\mu, n)}{K_\mu(n)} + 1\right]$$

- where,

$$G(\mu, n) = 1 + \sum_{j=1}^{n-1}\prod_{i=j}^{n-1} k_\mu(i)$$

$$K_\mu(n) = \prod_{i=0}^{n-1} k_\mu(i) = \prod_{i=0}^{n-1}\sqrt{1+\mu 2^{(-2i)}}$$

- Again, b = number of fractional bits in data paths, n = number of iterations.

Notes:

Effective Fractional Bits (d_{eff})

6.32

- To compute the number of effective bits (d_{eff}), the OQE must first be computed.
- It has been shown that:

$$OQE = Approximation\ Error + Rounding\ Error$$

- The number of effective bits can then be computed as:

$$d_{eff} = -(\log_2 OQE)$$

- This analysis relies on first choosing ***b*** and ***n,*** and then computing d_{eff}.
- However, from a design perspective, we ideally want to specify d_{eff} and obtain values for ***b*** and ***n,*** rather than the other way round!
- Hence, the approach taken by Yu Hen Hu was to take a set of ***b*** and ***n*** and compile a table showing all values of d_{eff} for this set. By scanning the table to find a desired d_{eff}, ***b*** and ***n*** can be selected appropriately.

Predicted Effective Fractional Bits

6.33

- Using the equation for the OQE, a table can be produced that predicts d_{eff} for a set of ***n*** and ***b***.
- Assume the input is constrained to ± 0.5, therefore $|v(0)| = \sqrt{0.5}$.
- Using this value of $|v(0)|$ and for $3 \le n \le 9$ and $8 \le b \le 10$ the following table was generated.

n \ b	Predicted d_{eff}		
	8	9	10
3	5.09	5.31	5.43
4	6.03	6.59	6.98
5	6.28	7.13	7.88
6	6.21	7.17	8.10
7	6.06	7.05	8.04
8	5.92	6.91	7.91
9	5.78	6.78	7.78

Notes:

The number of effective fractional bits can be found by using the following equation:

$$d_{eff} = -(\log_2 OQE)$$

The table below shows the relationship between the OQE and d_{eff} for the first few values.

OQE	d_{eff}
≤ 0.5	1
≤ 0.25	2
≤ 0.125	3
≤ 0.0625	4
≤ 0.03125	5
≤ 0.015625	6
.	.
.	.
.	.

Not all calculators allow base 2 calculations explicitly. This equivalent calculation may therefore prove useful:

$$\log_2 OQE = \frac{\log_{10} OQE}{\log_{10} 2}$$

Notes:

The d_{eff} table can be used to find values for ***n*** and ***b*** for a specific d_{eff}. Say for example that we wanted to compute the magnitude of a vector with an accuracy of 6 fractional bits. By scanning the table below, the most efficient architecture that appears to give this accuracy is ***n* = 4, *b* = 8** (highlighted in the table below). When these values are used to design an actual CORDIC system, the worst case error that this design will produce, relative to a floating point reference design, is less than 2^{-6}.

Simulated results are presented on the right hand side of the table, and confirm that the experienced errors are never worse than the theoretically predicted values.

n \ b	Predicted			Simulated		
	8	9	10	8	9	10
3	5.09	5.31	5.43	5.32	5.71	5.80
4	**6.03**	6.59	6.98	**6.22**	6.88	7.34
5	6.28	7.13	7.88	6.52	7.56	8.27
6	6.21	7.17	8.10	6.55	7.11	8.12
7	6.06	7.05	8.04	6.42	7.29	8.29
8	5.92	6.91	7.91	6.33	7.29	8.31
9	5.78	6.78	7.78	6.00	7.00	8.00

Vector Magnitude Hardware Comparison

- The hardware implementations of the following designs are compared, with 10 iterations and 16 fractional bits:
  - ***Parallel / Unrolled (without pipelining)***
  - ***Parallel / Unrolled (with pipelining)***
  - ***Serial / Rolled***

	Parallel	Pipelined Parallel	Serial
LUTs	565	550	416
Flip Flops	73	442	136
Slices	333	328	290
Max. Clock Frequency	~21MHz	~121MHz	~44MHz
Max. Throughput	~21Msamples/s	~121Msamples/s	~4.4Msamples/s

- Note that the serial design provides the lowest throughput, because a single CORDIC cell is used for all iterations (i.e. time shared 10x).

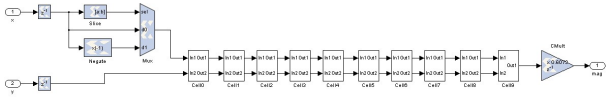
Conclusions

- In this section the theory behind the ***CORDIC*** algorithm has been introduced.
- The ***Givens transform*** is used as the basis for the algorithm.
- Simplifications reduce the transform to an ***iterative*** series of successively smaller ***pseudo-rotations***.
- It was demonstrated that each iteration comprises only ***shifts and adds***.
- An undesirable ***scaling*** is introduced as a result of the pseudo-rotations, but this is easily correctable using a ***constant multiplier***.
- Different operation ***modes*** and ***coordinate systems*** were introduced. Overall, CORDIC offers a great range of computable functions.
- Finally, the FPGA ***implementation*** of a CORDIC core was considered, and features and characteristics of candidate architectures reviewed.

Notes:

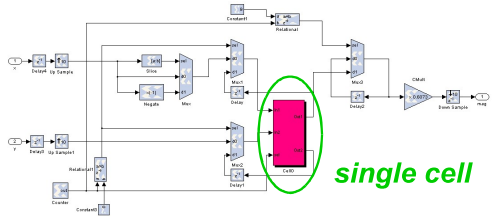
These designs all have 16 fractional bits in the data path, with 10 iterations. Area and speed were generated based on a Virtex-II Pro target device.

Parallel Design



Parallel, Pipelined Design

Serial Design



Notes:

Distributed by: *http://www.xilinx.com/univ/*

Developed by: www.steepestascent.com