

# 6.S965

# Digital Systems Laboratory II

## Lecture 8: CORDIC and Iterative Algorithms

# Administrative

- Please don't hesitate to post on piazza if you see a thing that might be a typo. I am an imperfect being.
- Week 4 is out/due this upcoming Friday.
- Week 5's assignments will be a transition week...we'll still have a Pynq board thing to build (based on your week*little* 3 build so hopefully super easy), but will mostly be simulation.
- Weeks 6-8 will be on RFSoc

# Status

- Week 4's assignment is deploying a DMA engine on the Pynq board.
- Forces us into an AXI-state of mind
- You'll use your FIR filter as an accelerator, but there are many other accelerators.

# This Week

- Study the CORDIC and implement it.
- Use it to get magnitudes and angles of vectors.



# There are tons of cool algorithms out there

- Particularly for FPGAs or digital environments in general
- People actively making improvements and things!

## Low-cost, High-speed Parallel FIR Filters for RFSoc Front-Ends Enabled by C $\lambda$ SH

Craig Ramsay  
University of Strathclyde  
Glasgow, Scotland  
craig.ramsay.100@strath.ac.uk

Louise H. Crockett  
University of Strathclyde  
Glasgow, Scotland  
louise.crockett@strath.ac.uk

Robert W. Stewart  
University of Strathclyde  
Glasgow, Scotland  
r.stewart@strath.ac.uk

**Abstract**—We present a new low-cost, high-speed parallel FIR filter generator targeting the Xilinx Radio Frequency System on Chip (RFSoc) and direct RF sampling applications. We compose two existing approaches in a novel hierarchy: efficient parallelism with Fast FIR Algorithm (FFA) structures, and efficient multiplierless FIR implementations with  $H_{cub}$ . The resource usage advantages (in both area and type) are compared with similar output from the traditional architecture, exemplified by vendor tools, as well as the  $H_{cub}$ -based filters without the FFA optimisation. Although these techniques are well studied individually in the literature, they have not enjoyed mainstream use as their structural complexity proves awkward to capture with traditional Hardware Description Languages (HDLs). This work continues a discussion of the use of functional programming techniques in hardware description, highlighting the benefits of having easily composable circuit generators.

### I. INTRODUCTION

We present a new family of low-cost, high-speed, parallel Finite Impulse Response (FIR) filters targeting direct Radio Frequency (RF) sampling applications with the Xilinx Zynq

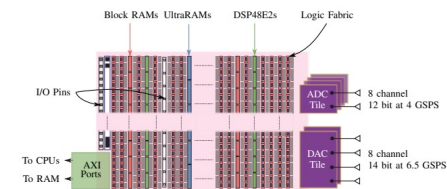


Fig. 1. Overview of RFSoc's FPGA and RF Data Converters

- Custom Digital Up/Down Conversion (DUC/DDC) as a front-end of *any* radio application. Especially useful when the characteristics of the available hardened DUC/DDCs [5] do not meet the application's requirements.

The demand for sample parallelism and the multi-channel nature of the RFSoc device amplifies the effects of filter

# One algorithm we should study...

# CORDIC

- **Coordinate Rotation Digital Computer**
- Super versatile class of iterative algorithms that are used widely in hardware because they are relatively simple to implement (mostly just shifts and adds and compares)...maybe a multiply
- Might not be the fastest, but are a good gateway algorithm for lots of options out there.

# CORDIC

- What can you compute with CORDIC?

## Directly computable functions [\[ edit | edit source \]](#)

$\sin z$	$\cos z$
$\tan^{-1} z$	$\sinh z$
$\cosh z$	$\tanh^{-1} z$
$y/x$	$xz$
$\tan^{-1}(y/x)$	$\sqrt{x^2 + y^2}$
$\sqrt{x^2 - y^2}$	$e^z = \sinh z + \cosh z$

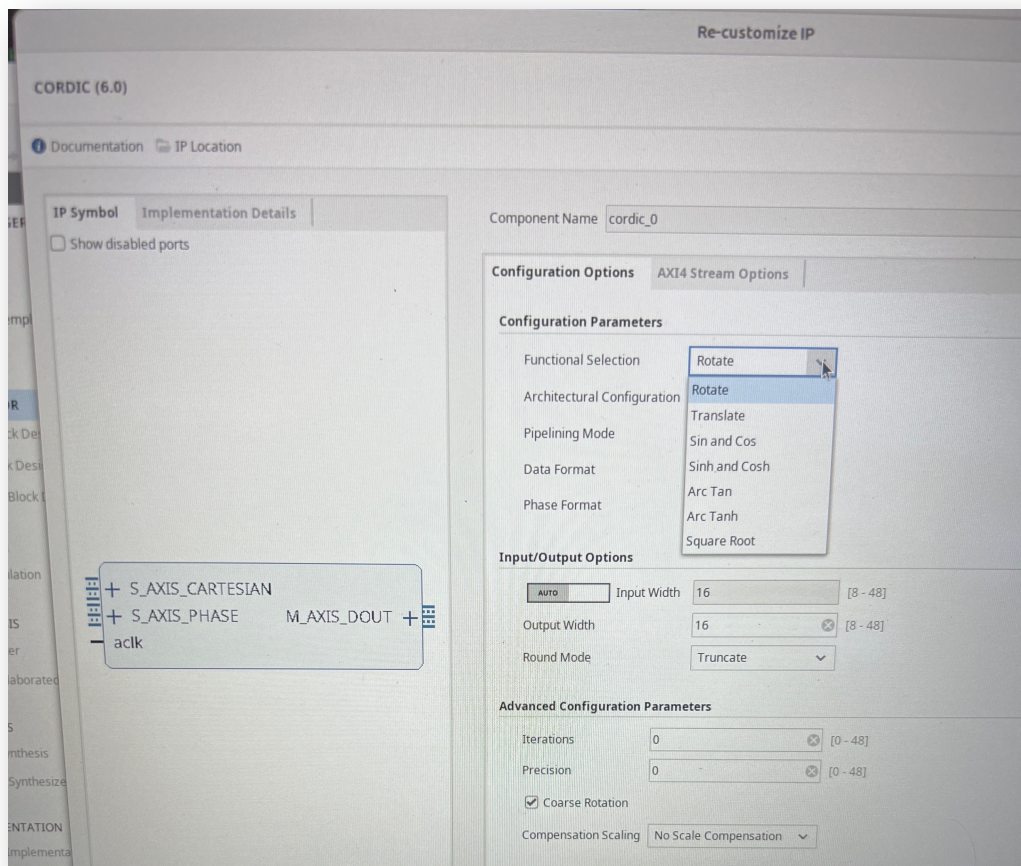
## Indirectly computable functions [\[ edit | edit source \]](#)

In addition to the above functions, a number of other functions can be produced by combining the results of previous computations:

$\tan z = \frac{\sin z}{\cos z}$	$\cos^{-1} w = \tan^{-1} \frac{\sqrt{1 - w^2}}{w}$
$\tanh z = \frac{\sinh z}{\cosh z}$	$\sin^{-1} w = \tan^{-1} \frac{w}{\sqrt{1 - w^2}}$
$\ln w = 2 \tanh^{-1} \frac{w - 1}{w + 1}$	$\log_b w = \frac{\ln w}{\ln b}$
$w^t = e^{t \ln w}$	$\cosh^{-1} = \ln(w + \sqrt{w^2 - 1})$
$\tan^{-1}(y/x)$	$\sinh^{-1} = \ln(w + \sqrt{w^2 + 1})$
$\sqrt{x^2 - y^2}$	$\sqrt{w} = \sqrt{(w + 1/4)^2 - (w - 1/4)^2}$

*What can't you  
compute with  
CORDIC?*

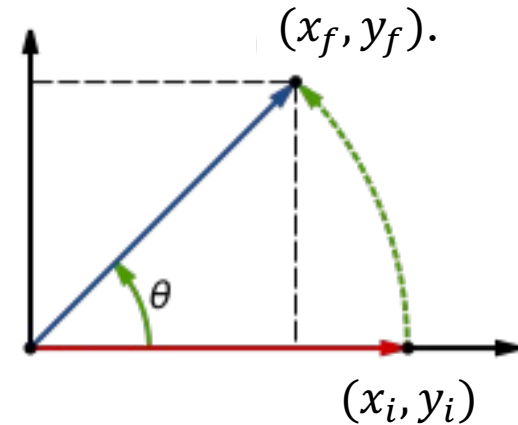
# Vivado..



- Just tell the little elves inside to calculate this for you...
- Or you do it yourself to learn.

# CORDIC

- Built around the idea of rotations



- Rotation Matrix:

$$\begin{bmatrix} x_f \\ y_f \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

- Also break down into two equations:

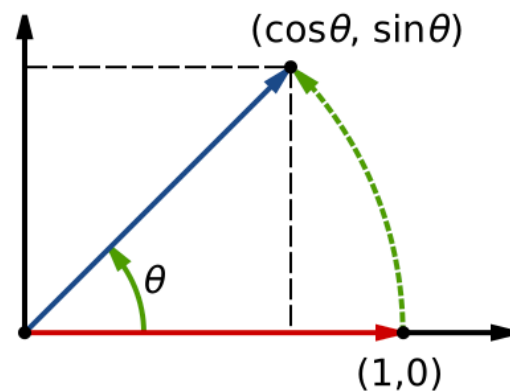
$$x_f = \cos(\theta) x_i - \sin(\theta) y_i$$

$$y_f = \sin(\theta) x_i + \cos(\theta) y_i$$

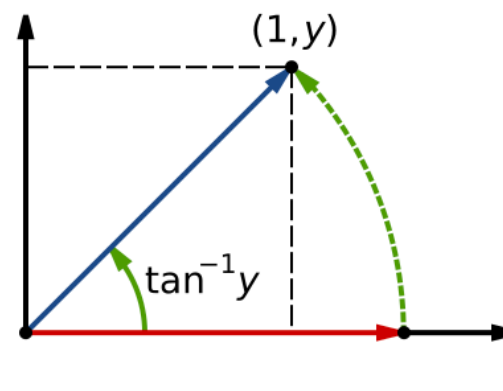
# Why is the ability to rotate useful?

*Motivation to do this...*

- If we could carry out that rotation we could start to answer questions like...



- Start at (1, 0)
- Rotate by  $\theta$
- We get  $(\cos \theta, \sin \theta)$



- Start at (1, y)
- Rotate until  $y = 0$
- The rotation is  $\tan^{-1} y$

# OK so what do we need to do...

- We need to be able to do this...

$$\begin{bmatrix} x_f \\ y_f \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

- But this is a little chicken-and-egg...because in order to do this, we need to be able to do  $\sin(\theta)$  or  $\cos(\theta)$  which are things we don't have as ready-made functions



# Trig Identities

- > Reflections, shifts, and periodicity
- > Angle sum and difference identities
- > Multiple-angle and half-angle formulae
- Power-reduction formulae
- > Product-to-sum and sum-to-product identities
- > Linear combinations
- Lagrange's trigonometric identities
- Certain linear fractional transformations
- Relation to the complex exponential function
- Series expansion
- Infinite product formulae
- Inverse trigonometric functions
- > Identities without variables
- Composition of trigonometric functions
- Further "conditional" identities for the case  $\alpha + \beta + \gamma = 180^\circ$
- Historical shorthands
- > Miscellaneous
- See also



cos  $\theta$

sec  $\theta$

Trigonometric functions and their reciprocals on the unit circle. All of the right-angled triangles are similar, i.e. the ratios between their corresponding sides are the same. For sin, cos and tan the unit-length radius forms the hypotenuse of the triangle that defines them. The reciprocal identities arise as ratios of sides in the triangles where this unit line is no longer the hypotenuse. The triangle shaded blue illustrates the identity  $1 + \cot^2 \theta = \csc^2 \theta$ , and the red triangle shows that  $\tan^2 \theta + 1 = \sec^2 \theta$ .

Each trigonometric function in terms of each of the other five.<sup>[1]</sup>

in terms of	$\sin \theta$	$\csc \theta$	$\cos \theta$	$\sec \theta$	$\tan \theta$	$\cot \theta$
$\sin \theta =$	$\sin \theta$	$\frac{1}{\csc \theta}$	$\pm \sqrt{1 - \cos^2 \theta}$	$\pm \frac{\sqrt{\sec^2 \theta - 1}}{\sec \theta}$	$\pm \frac{\tan \theta}{\sqrt{1 + \tan^2 \theta}}$	$\pm \frac{1}{\sqrt{1 + \cot^2 \theta}}$
$\csc \theta =$	$\frac{1}{\sin \theta}$	$\csc \theta$	$\pm \frac{1}{\sqrt{1 - \cos^2 \theta}}$	$\pm \frac{\sec \theta}{\sqrt{\sec^2 \theta - 1}}$	$\pm \frac{\sqrt{1 + \tan^2 \theta}}{\tan \theta}$	$\pm \sqrt{1 + \cot^2 \theta}$
$\cos \theta =$	$\pm \sqrt{1 - \sin^2 \theta}$	$\pm \frac{\sqrt{\csc^2 \theta - 1}}{\csc \theta}$	$\cos \theta$	$\frac{1}{\sec \theta}$	$\pm \frac{1}{\sqrt{1 + \tan^2 \theta}}$	$\pm \frac{\cot \theta}{\sqrt{1 + \cot^2 \theta}}$
$\sec \theta =$	$\pm \frac{1}{\sqrt{1 - \sin^2 \theta}}$	$\pm \frac{\csc \theta}{\sqrt{\csc^2 \theta - 1}}$	$\frac{1}{\cos \theta}$	$\sec \theta$	$\pm \sqrt{1 + \tan^2 \theta}$	$\pm \frac{\sqrt{1 + \cot^2 \theta}}{\cot \theta}$
$\tan \theta =$	$\pm \frac{\sin \theta}{\sqrt{1 - \sin^2 \theta}}$	$\pm \frac{1}{\sqrt{\csc^2 \theta - 1}}$	$\pm \frac{\sqrt{1 - \cos^2 \theta}}{\cos \theta}$	$\pm \sqrt{\sec^2 \theta - 1}$	$\tan \theta$	$\frac{1}{\cot \theta}$
$\cot \theta =$	$\pm \frac{\sqrt{1 - \sin^2 \theta}}{\sin \theta}$	$\pm \sqrt{\csc^2 \theta - 1}$	$\pm \frac{\cos \theta}{\sqrt{1 - \cos^2 \theta}}$	$\pm \frac{1}{\sqrt{\sec^2 \theta - 1}}$	$\frac{1}{\tan \theta}$	$\cot \theta$

# Trig Identities

- > Reflections, shifts, and periodicity
- > Angle sum and difference identities
- > Multiple-angle and half-angle formulae
- Power-reduction formulae
- > Product-to-sum and sum-to-product identities
- > Linear combinations

Lagrange's trigonometric identities

Certain linear fractional transformations

Relation to the complex exponential function

Series expansion

Infinite product formulae

Inverse trigonometric functions

- > Identities without variables

Composition of trigonometric functions

Further "conditional" identities for the case  $\alpha + \beta + \gamma = 180^\circ$

Historical shorthands

- > Miscellaneous

See also

$\cos \theta$

$\sec \theta$

Trigonometric functions and their reciprocals on the unit circle. All of the right-angled triangles are similar, i.e. the ratios between their corresponding sides are the same. For sin, cos and tan the unit-length radius forms the hypotenuse of the triangle that defines them. The reciprocal identities arise as ratios of sides in the triangles where this unit line is no longer the hypotenuse. The triangle shaded blue illustrates the identity  $1 + \cot^2 \theta = \csc^2 \theta$ , and the red triangle shows that  $\tan^2 \theta + 1 = \sec^2 \theta$ .

Each trigonometric function in terms of each of the other five.<sup>[1]</sup>

in terms of	$\sin \theta$	$\csc \theta$	$\cos \theta$	$\sec \theta$	$\tan \theta$	$\cot \theta$
$\sin \theta =$	$\sin \theta$	$\frac{1}{\csc \theta}$	$\pm \sqrt{1 - \cos^2 \theta}$	$\pm \frac{\sqrt{\sec^2 \theta - 1}}{\sec \theta}$	$\pm \frac{\tan \theta}{\sqrt{1 + \tan^2 \theta}}$	$\pm \frac{1}{\sqrt{1 + \cot^2 \theta}}$
$\csc \theta =$	$\frac{1}{\sin \theta}$	$\csc \theta$	$\pm \frac{1}{\sqrt{1 - \cos^2 \theta}}$	$\pm \frac{\sec \theta}{\sqrt{\sec^2 \theta - 1}}$	$\pm \frac{\sqrt{1 + \tan^2 \theta}}{\tan \theta}$	$\pm \sqrt{1 + \cot^2 \theta}$
$\cos \theta =$	$\pm \sqrt{1 - \sin^2 \theta}$	$\pm \frac{\sqrt{\csc^2 \theta - 1}}{\csc \theta}$	$\cos \theta$	$\frac{1}{\sec \theta}$	$\pm \frac{1}{\sqrt{1 + \tan^2 \theta}}$	$\pm \frac{\cot \theta}{\sqrt{1 + \cot^2 \theta}}$
$\sec \theta =$	$\pm \frac{1}{\sqrt{1 - \sin^2 \theta}}$	$\pm \frac{\csc \theta}{\sqrt{\csc^2 \theta - 1}}$	$\frac{1}{\cos \theta}$	$\sec \theta$	$\pm \sqrt{1 + \tan^2 \theta}$	$\pm \frac{\sqrt{1 + \cot^2 \theta}}{\cot \theta}$
$\tan \theta =$	$\pm \frac{\sin \theta}{\sqrt{1 - \sin^2 \theta}}$	$\pm \frac{1}{\sqrt{\csc^2 \theta - 1}}$	$\pm \frac{\sqrt{1 - \cos^2 \theta}}{\cos \theta}$	$\pm \sqrt{\sec^2 \theta - 1}$	$\tan \theta$	$\frac{1}{\cot \theta}$
$\cot \theta =$	$\pm \frac{\sqrt{1 - \sin^2 \theta}}{\sin \theta}$	$\pm \sqrt{\csc^2 \theta - 1}$	$\pm \frac{\cos \theta}{\sqrt{1 - \cos^2 \theta}}$	$\pm \frac{1}{\sqrt{\sec^2 \theta - 1}}$	$\frac{1}{\tan \theta}$	$\cot \theta$

# Identity

$$\cos(\theta) = \frac{1}{\sqrt{1 + \tan^2(\theta)}} \quad \sin(\theta) = \frac{\tan(\theta)}{\sqrt{1 + \tan^2(\theta)}}$$

- That means these:

$$x_f = \cos(\theta) x_i - \sin(\theta) y_i$$

$$y_f = \sin(\theta) x_i + \cos(\theta) y_i$$

- Can turn into these:

$$x_f = (x_i - \tan(\theta) y_i) \frac{1}{\sqrt{1 + \tan^2(\theta)}}$$

$$y_f = (y_i + \tan(\theta) x_i) \frac{1}{\sqrt{1 + \tan^2(\theta)}}$$

*Let's ignore these*



# So now our task:

- Now we have this:
- Ignoring that factor on the outside does break stuff.
- We're no longer really doing a pure rotation
- ...we have to call it something else...

The ' means value isn't same as before



$$x'_f = (x_i - \tan(\theta) y_i)$$

$$y'_f = (y_i + \tan(\theta) x_i)$$

*We also still don't know how to calculate  $\tan(\theta)$ ...that'll come. Patience, friend.*

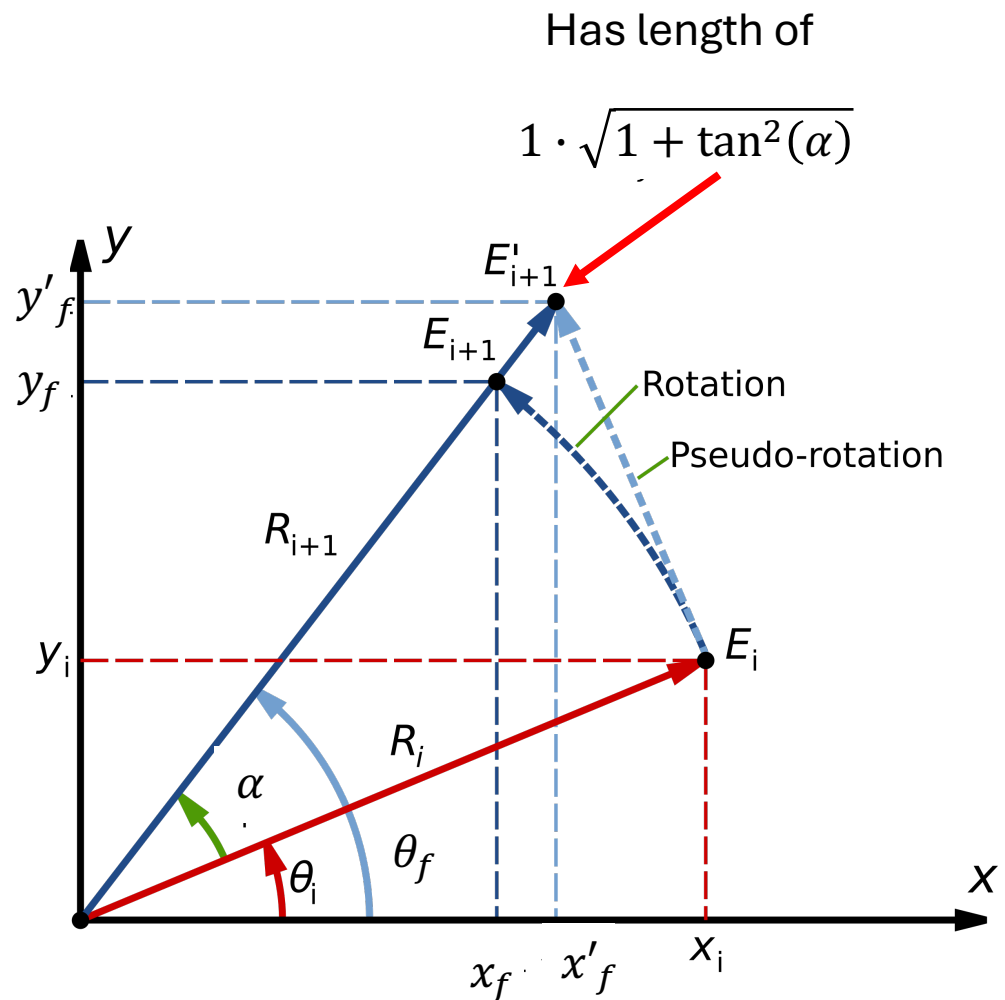
# PseudoRotations

- In a pseudorotation, you still rotate by the same angle, but you depart the unit circle:

What we've got

$$x'_f = (x_i - \tan(\alpha) y_i)$$

$$y'_f = (y_i + \tan(\alpha) x_i)$$



What we wanted

$$x_f = (x_i - \tan(\alpha) y_i) \frac{1}{\sqrt{1 + \tan^2(\alpha)}} \quad y_f = (y_i + \tan(\alpha) x_i) \frac{1}{\sqrt{1 + \tan^2(\alpha)}}$$

# OK still though...

*What we've got*

$$x'_f = (x_i - \tan(\alpha) y_i)$$

$$y'_f = (y_i + \tan(\alpha) x_i)$$

- We still don't know  $\tan(\theta)$
- Now we're using a thing we don't know, to do a thing we don't want....seems dumb if you ask me.

# Iterations

- We don't have to do this move all at one time. We could do it in steps.
- Just like you can apply a matrix...then apply a matrix...you can do the same thing here.
- Do a bunch of smaller pseudo rotations forwards and even backwards (like a binary search)
- Since we know the angle we want, we could keep track and adjust as we go.

step0

$$x_0 = (x_i - \tan(\alpha_0) y_i)$$

$$y_0 = (y_i + \tan(\alpha_0) x_i)$$

$$\theta_0 = 0 + \alpha_0$$

step1

$$x_1 = (x_0 - \tan(-\alpha_1) y_0)$$

$$y_1 = (y_0 + \tan(-\alpha_1) x_0)$$

$$\theta_1 = 0 + \alpha_0 - \alpha_1$$

step2

$$x_2 = (x_1 - \tan(\alpha_2) y_1)$$

$$y_2 = (y_1 + \tan(\alpha_2) x_1)$$

$$\theta_2 = 0 + \alpha_0 - \alpha_1 + \alpha_2$$

stepn...

$$x_n = (x_{n-1} - \tan(\alpha_n) y_{n-1})$$

$$y_n = (y_{n-1} + \tan(\alpha_n) x_{n-1})$$

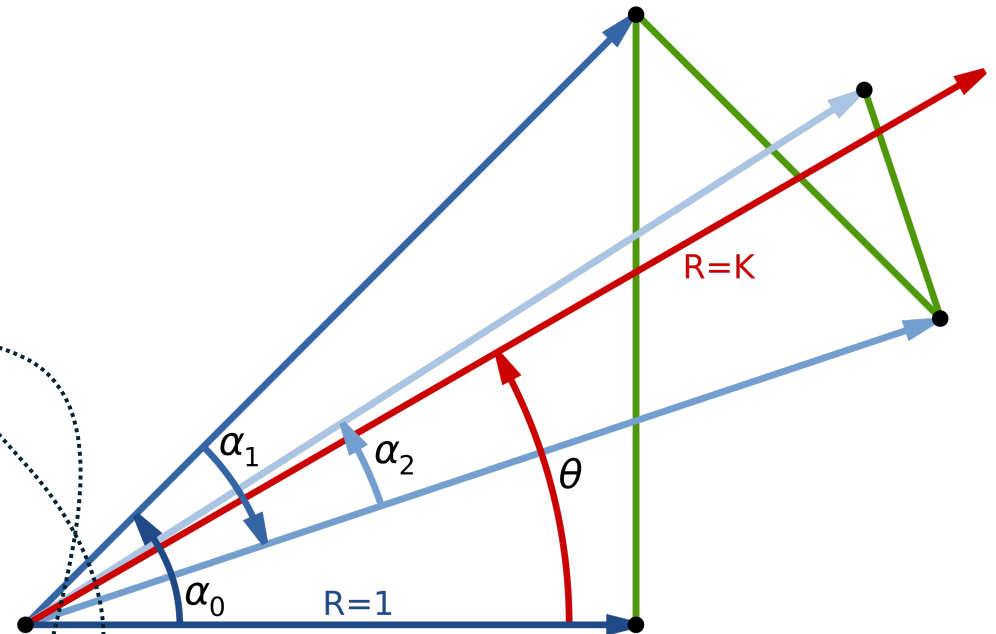
$$\theta_n = 0 + \alpha_0 - \alpha_1 + \alpha_2 + \cdots \alpha_n$$

Or alternatively: stepn...

$$x_n = (x_{n-1} - \tan(\alpha_n) y_{n-1})$$

$$y_n = (y_{n-1} + \tan(\alpha_n) x_{n-1})$$

$$\theta_n = \theta_{n-1} + \alpha_n$$





# OK interesting...

- If we could conceivably arrive at an arbitrary angle using a number of other steps...
- Could we pick a collection of steps that could be used to arrive at most arbitrary angles (within reason?)
- And could we pre-compute those angles?

*If* we have these precomputed angle jumps...

- Then we could potentially iterate towards our target  $\theta$  with a number of pre-calculated  $\alpha$  jumps
- We could keep track if our running tally is  $>$  or  $<$   $\theta$  and add or subtract our  $\alpha$  as needed.

# What do attributes do we want for our precomputed $\alpha$ ?

Or alternatively: stepn...

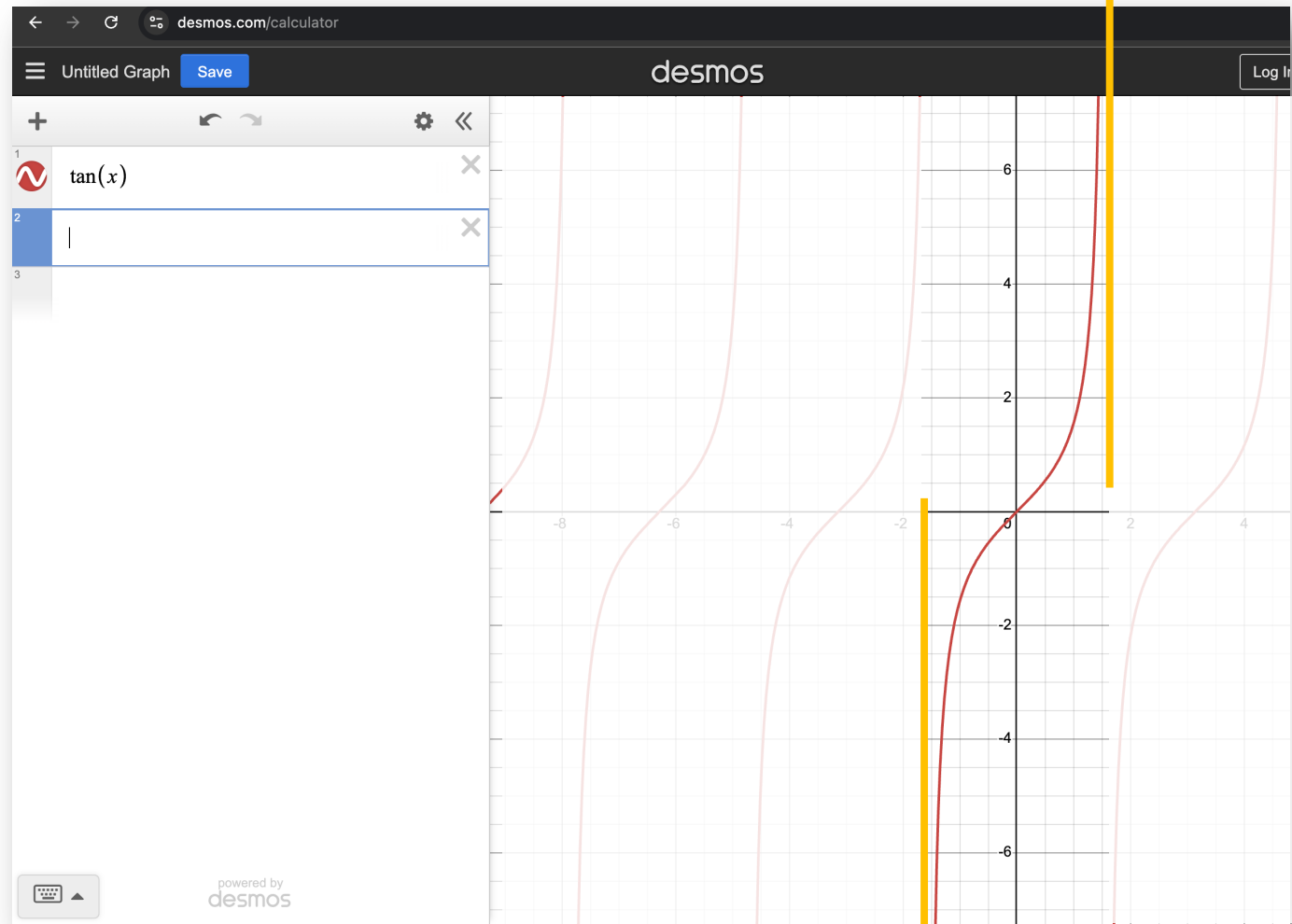
$$x_n = (x_{n-1} - \tan(\alpha_n) y_{n-1})$$

$$y_n = (y_{n-1} + \tan(\alpha_n) x_{n-1})$$

$$\theta_n = \theta_{n-1} + \alpha_n$$

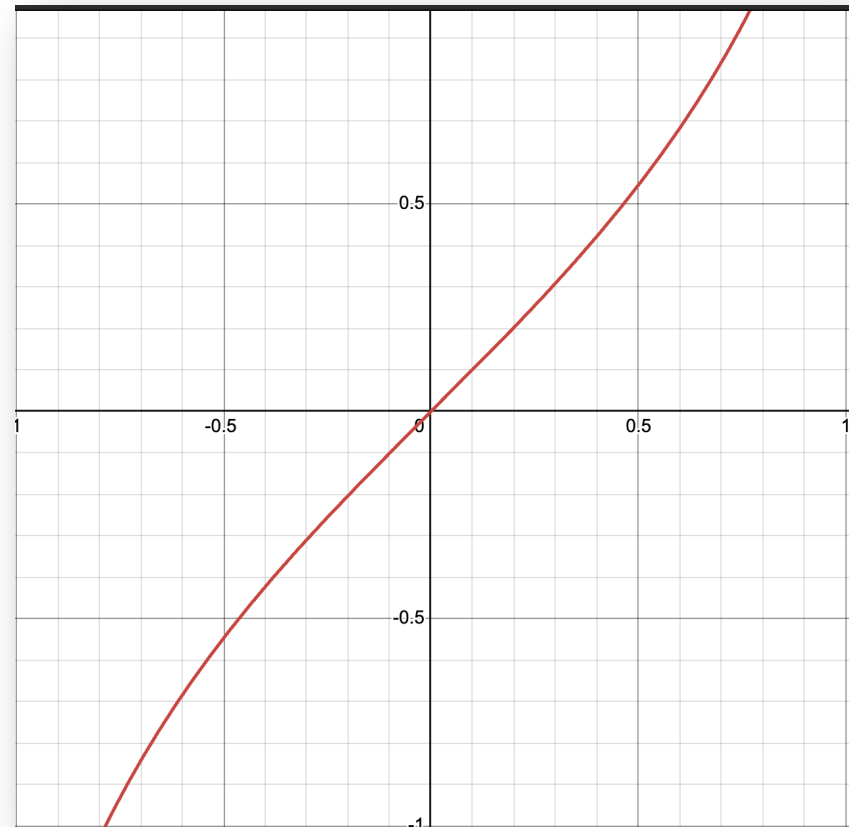
- What we really care about are good, clean, wholesome, easy-to-multiply values of  $\tan(\alpha)$
- And remember we're not in human land, we're in digital land...so what are nice and easy to apply are in base 2!
- So are there any nice base-2 friendly  $\tan(\alpha)$  values?
- And it sure would be nice to have angles that could go forwards or backwards

# Appreciate $\tan(x)$



# $\tan(x)$ has some symmetry

- That's nice...that means we could just store precomputed values of  $\tan(\alpha)$  for  $\alpha > 0$  and just flip signs when needed.



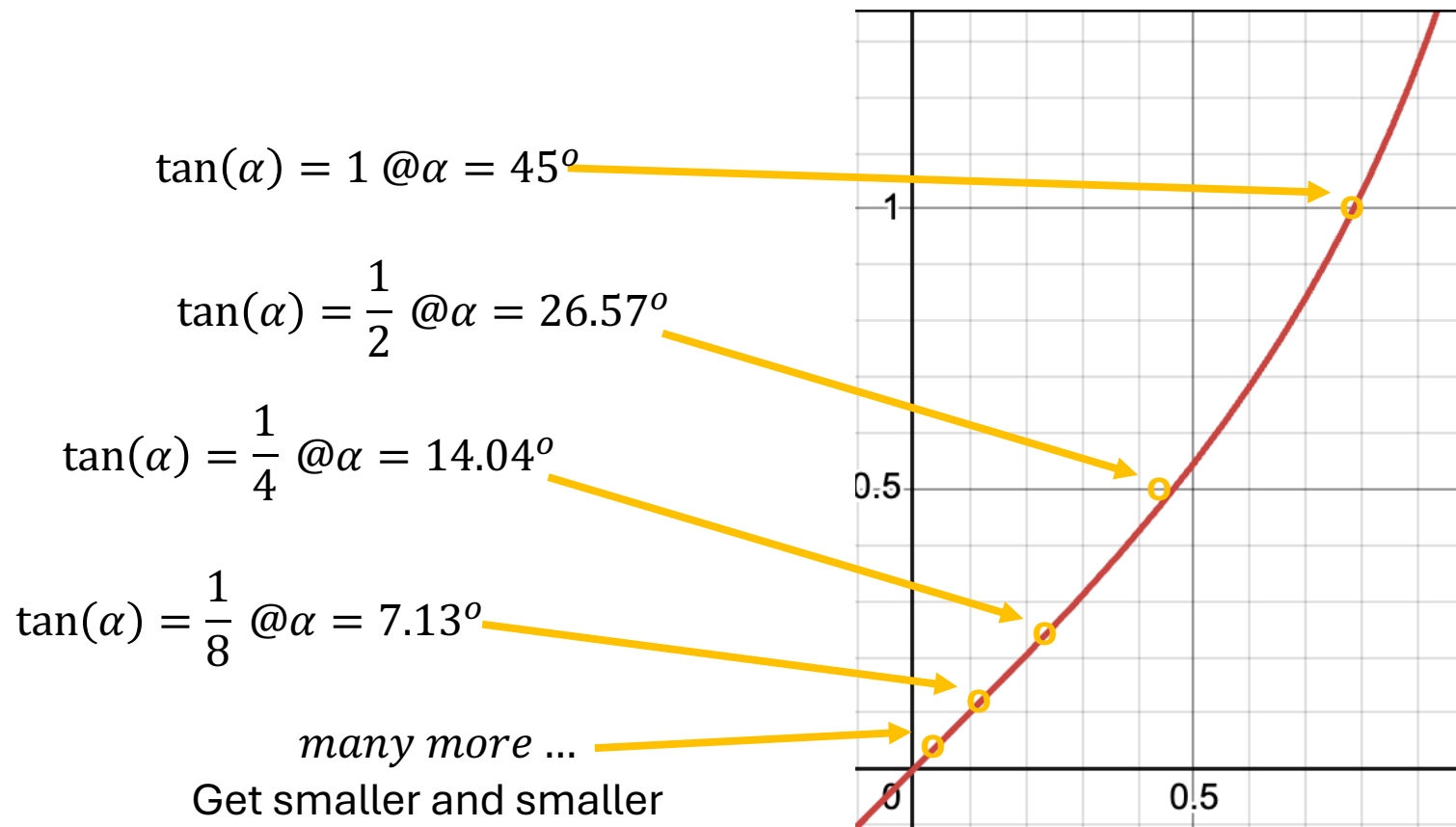
# Are there any “nice” $\tan(\alpha)$ ???

Or alternatively: stepn...

$$x_n = (x_{n-1} - \tan(\alpha_n) y_{n-1})$$

$$y_n = (y_{n-1} + \tan(\alpha_n) x_{n-1})$$

$$\theta_n = \theta_{n-1} + \alpha_n$$



# Do this for a bunch of power-of-2 values

- Can generate a whole table...basically as many as you want
- The only nasty thing you need to store would be these precomputed angles
- But worth it since now all those multiplications by tangents are easy.

i	$\alpha_i = \tan^{-1}(2^{-i})$	
	Degrees	Radians
0	45.00	0.7854
1	26.57	0.4636
2	14.04	0.2450
3	7.13	0.1244
4	3.58	0.0624
5	1.79	0.0312
6	0.90	0.0160
7	0.45	0.0080
8	0.22	0.0040
9	0.11	0.0020

step0

$$x_0 = (x_i - \tan(\alpha_0) y_i)$$

$$y_0 = (y_i + \tan(\alpha_0) x_i)$$

$$\theta_0 = 0 + \alpha_0$$

step1

$$x_1 = (x_0 - \tan(-\alpha_1) y_0)$$

$$y_1 = (y_0 + \tan(-\alpha_1) x_0)$$

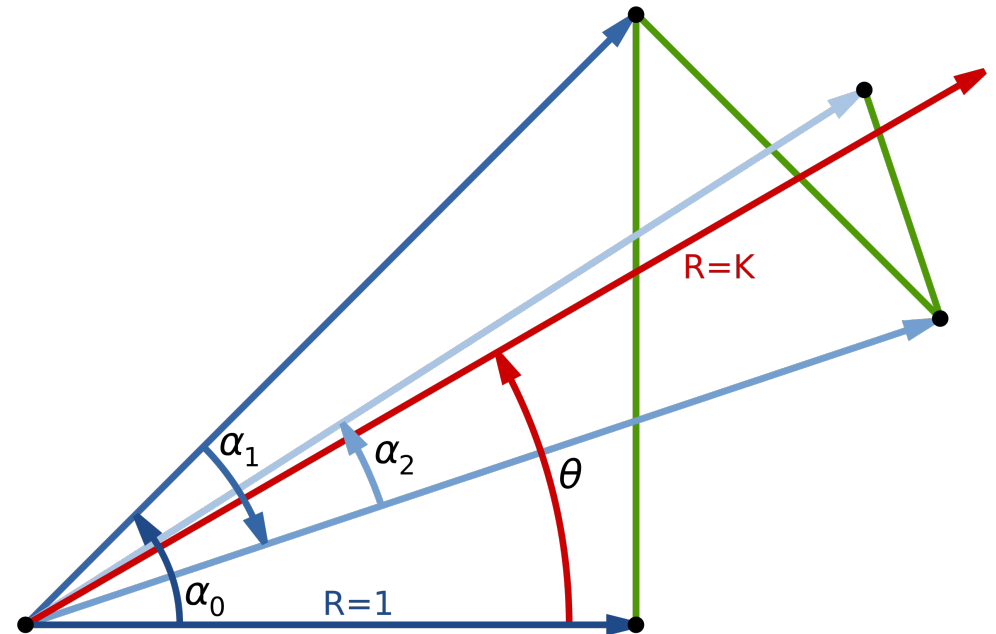
$$\theta_1 = 0 + \alpha_0 - \alpha_1$$

step2

$$x_2 = (x_1 - \tan(\alpha_2) y_1)$$

$$y_2 = (y_1 + \tan(\alpha_2) x_1)$$

$$\theta_2 = 0 + \alpha_0 - \alpha_1 + \alpha_2$$



Or alternatively: stepn...

$$x_n = (x_{n-1} - \tan(\alpha_n) y_{n-1})$$

$$y_n = (y_{n-1} + \tan(\alpha_n) x_{n-1})$$

$$\theta_n = \theta_{n-1} + \alpha_n$$



step0

$$x_0 = (x_i - 1 \cdot y_i)$$

$$y_0 = (y_i + 1 \cdot x_i)$$

$$\theta_0 = 0 + 45$$

step1

$$x_1 = (x_0 - 1/2 y_0)$$

$$y_1 = (y_0 + 1/2 x_0)$$

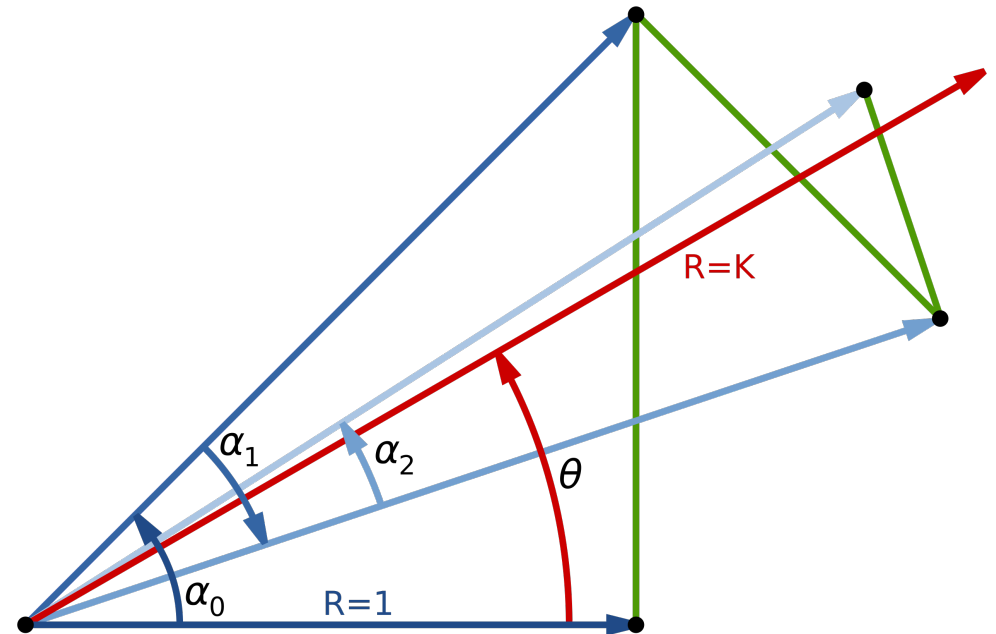
$$\theta_1 = 45 - 26.57$$

step2

$$x_2 = (x_1 - 1/4 y_1)$$

$$y_2 = (y_1 + 1/4 x_1)$$

$$\theta_2 = 18.43 + 14.04$$



stepn...

$$x_n = (x_{n-1} - 1/2^n y_{n-1})$$

$$y_n = (y_{n-1} + 1/2^n x_{n-1})$$

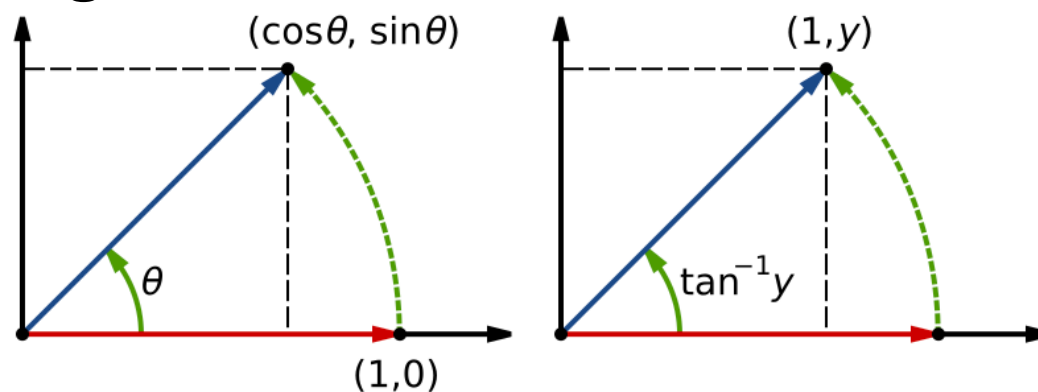
$$\theta_n = 32.47 + \dots \alpha_n$$

# More and More

- The more iterations you do, the closer and closer you'll be able to get your final angle to your desired angle.
- It works out to about 1 bit of precision per iteration.
- But we're *still* not there yet.

# We wanted to do this...

- Rotate things.



- Start at  $(1, 0)$
- Rotate by  $\theta$
- We get  $(\cos\theta, \sin\theta)$
- Start at  $(1, y)$
- Rotate until  $y = 0$
- The rotation is  $\tan^{-1}y$

- But we're not...We're pseudo-rotating :/

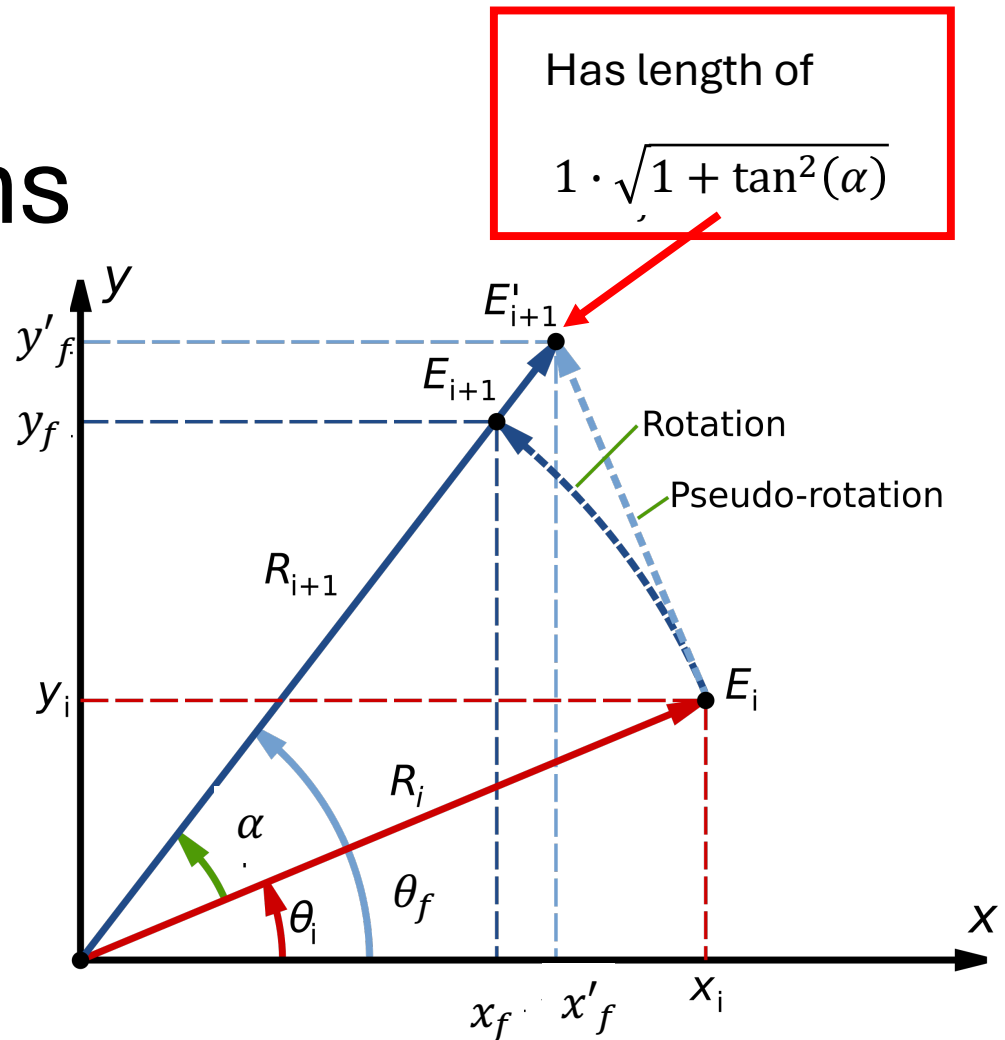
# PseudoRotations

- In a pseudorotation, you still rotate by the same angle, but you depart the unit circle:

What we've got

$$x'_f = (x_i - \tan(\alpha) y_i)$$

$$y'_f = (y_i + \tan(\alpha) x_i)$$



What we wanted

$$x_f = (x_i - \tan(\alpha) y_i) \frac{1}{\sqrt{1 + \tan^2(\alpha)}} \quad y_f = (y_i + \tan(\alpha) x_i) \frac{1}{\sqrt{1 + \tan^2(\alpha)}}$$

# Remember...

$$\cos(\theta) x_i = \frac{1}{\sqrt{1 + \tan^2(\theta)}}$$

$$\sin(\theta) = \frac{\tan(\theta)}{\sqrt{1 + \tan^2(\theta)}}$$

- That means these:

$$x_f = \cos(\theta) x_i - \sin(\theta) y_i$$

$$y_f = \sin(\theta) x_i + \cos(\theta) y_i$$

- Can turn into these:

$$x_f = (x_i - \tan(\theta) y_i) \frac{1}{\sqrt{1 + \tan^2(\theta)}}$$

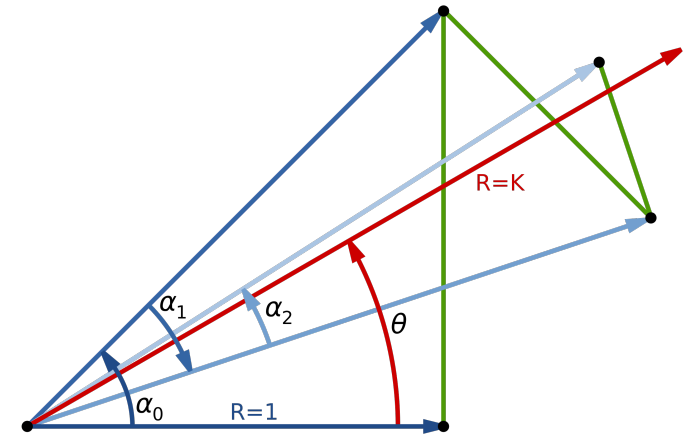
$$y_f = (y_i + \tan(\theta) x_i) \frac{1}{\sqrt{1 + \tan^2(\theta)}}$$

*Let's ignore these*



# We can zero in on our angle...

- But the x,y final locations are still messed up
- On each iteration since we're not multiplying by  $\frac{1}{\sqrt{1+\tan^2(\alpha_i)}}$ ...
- That means we're *actually* multiplying by  $\sqrt{1 + \tan^2(\alpha_i)}$
- You'll hear this called “gain” of a pseudorotation...



# So after n iterations of pseudo-rotation...

- One would expect the vector to be this large...

$$K = \prod_{i=0}^{n-1} \sqrt{1 + \tan^2 \alpha_i}$$

- What will K be?
- It is going to depend on what/how we rotated right? And that is nasty...

# BUTTTT Not really!!!!

- We know ahead of time all those  $\alpha$  values and because of their behavior around 0 and the squaring, it doesn't matter if we + or - with them
- For a given implementation of  $n$  steps...

$$K = \prod_{i=0}^{n-1} \sqrt{1 + \tan^2 \alpha_i}$$

- This *will stay the same*

i	$\alpha_i = \tan^{-1} (2^{-i})$	
	Degrees	Radians
0	45.00	0.7854
1	26.57	0.4636
2	14.04	0.2450
3	7.13	0.1244
4	3.58	0.0624
5	1.79	0.0312
6	0.90	0.0160
7	0.45	0.0080
8	0.22	0.0040
9	0.11	0.0020



# And not only that...

$$K = \prod_{i=0}^{n-1} \sqrt{1 + \tan^2 \alpha_i}$$

- All CORDIC implementations pick the same  $\alpha$  values and these get smaller and smaller
- That means this product actually converges to a fixed value,
- which works out to be:  
**1.646760258121**

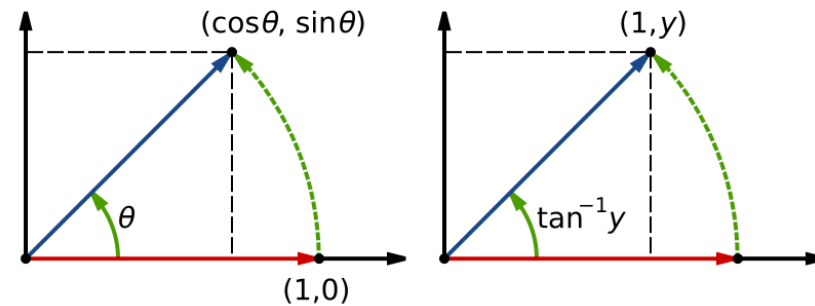
i	$\alpha_i = \tan^{-1}(2^{-i})$	
	Degrees	Radians
0	45.00	0.7854
1	26.57	0.4636
2	14.04	0.2450
3	7.13	0.1244
4	3.58	0.0624
5	1.79	0.0312
6	0.90	0.0160
7	0.45	0.0080
8	0.22	0.0040
9	0.11	0.0020

*Smaller n smaller*

# So once you're done...

- You can take your  $x_f$  and  $y_f$  and multiplying by 0.60725293634
  - Which is the same as multiplying by 39796 and then right shifting by 16
  - OR...which is the same as multiplying by 2608131502 and right shifting by 32.
- 
- You *can* also pre-multiply by this in your starting  $x_i$  and  $y_i$  (then just right shift at end!)

# Generalizing CORDIC



- Start at  $(1, 0)$
- Rotate by  $\theta$
- We get  $(\cos \theta, \sin \theta)$
- Start at  $(1, y)$
- Rotate until  $y = 0$
- The rotation is  $\tan^{-1} y$

- The pre-compute and step-by-step iterations are universal
- Their meaning and the target can be altered:
  - We previously targeted our accumulator to be  $\theta$
  - We could also target to get  $y$  to be 0...
    - The amount the accumulator ends up with is based on inverse tan of starting  $x$  and  $y$
    - The amount  $x$  ends up with is based on the  **$\text{sqrt}(x^{**2}+y^{**2})$**

# Generalized CORDIC

- The three equations we're iterating on can be generalized to this format

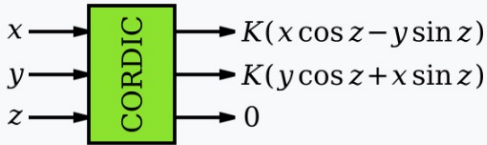
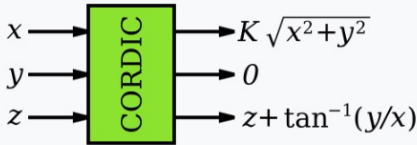
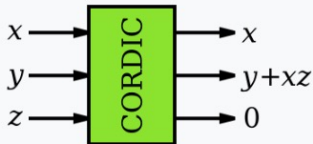
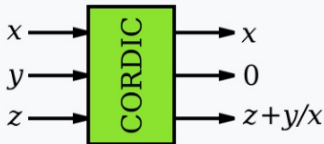
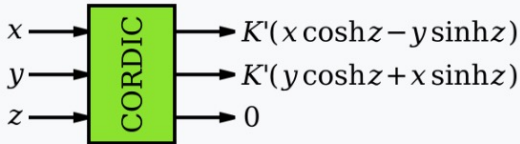
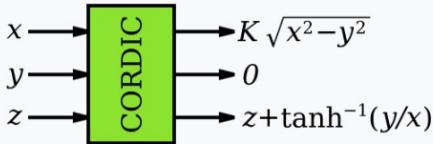
Diagram illustrating the generalized CORDIC equations and their components:

$$\begin{aligned}x_{i+1} &= x_i - \mu d_i y_i 2^{-i} \\y_{i+1} &= y_i + d_i x_i 2^{-i} \\z_{i+1} &= z_i + d_i \alpha_i\end{aligned}$$

Annotations:

- $z$  is our angle accumulator (points to  $z_{i+1}$ )
- $\mu$  is settable constant (points to  $\mu$ )
- $d_i$  is our control/feedback function for locking into a target ...this was  $\text{sgn}(\theta)$  in our walkthrough example so far (points to  $d_i$ )
- $2^{-i}$  are the  $\tan(\alpha_i)$  from our original example (points to  $2^{-i}$ )

# Different Modes

Mode	Rotation	Vectoring
	$d_i = \text{sgn}(z_i), \quad z \rightarrow 0$	$d_i = -\text{sgn}(y_i), \quad y \rightarrow 0$
<b>Circular</b> $\mu = 1$ $\alpha_i = \tan^{-1} 2^{-i}$		
<b>Linear</b> $\mu = 0$ $\alpha_i = 2^{-i}$		
<b>Hyperbolic</b> $\mu = -1$ $\alpha_i = \tanh^{-1} 2^{-i}$		

- In hyperbolic mode, iterations 4, 13, 40, 121, ...,  $j, 3j+1, \dots$  must be repeated. The constant  $K'$  given below accounts for this.
- $K = 1.646760258121\dots$
- $1/K = 0.607252935009\dots$
- $K' = 0.8281593609602\dots$
- $1/K' = 1.207497067763\dots$

# CORDIC

- You can use these outputs to generate all these weird things

## Directly computable functions [\[ edit | edit source \]](#)

$\sin z$	$\cos z$
$\tan^{-1} z$	$\sinh z$
$\cosh z$	$\tanh^{-1} z$
$y/x$	$xz$
$\tan^{-1}(y/x)$	$\sqrt{x^2 + y^2}$
$\sqrt{x^2 - y^2}$	$e^z = \sinh z + \cosh z$

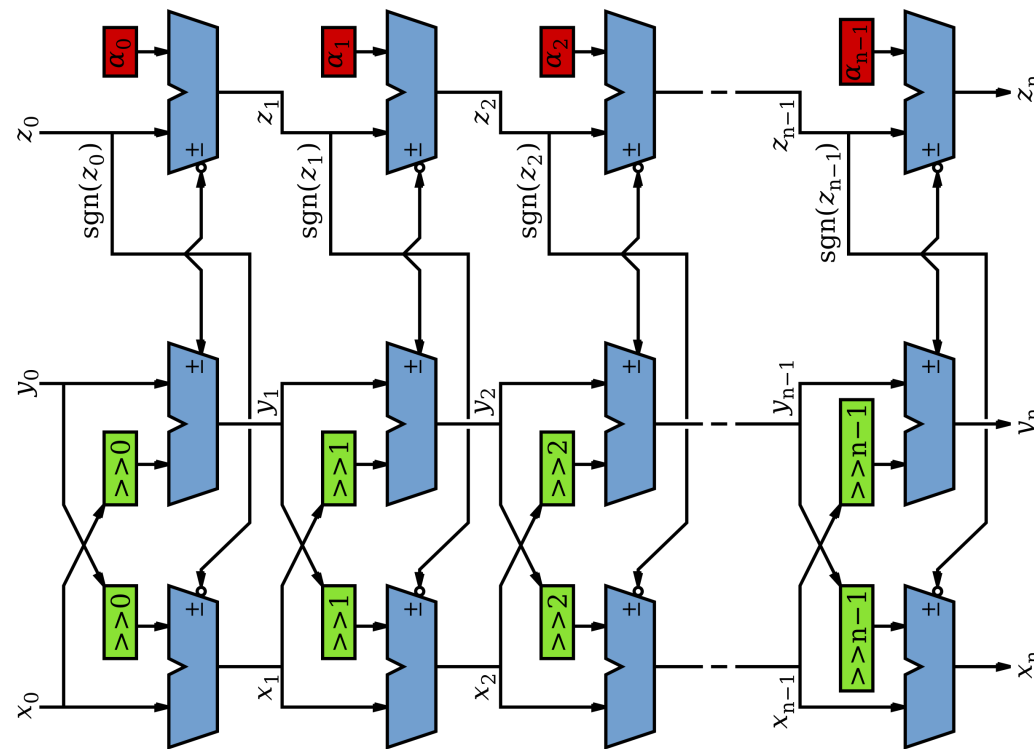
## Indirectly computable functions [\[ edit | edit source \]](#)

In addition to the above functions, a number of other functions can be produced by combining the results of previous computations:

$\tan z = \frac{\sin z}{\cos z}$	$\cos^{-1} w = \tan^{-1} \frac{\sqrt{1 - w^2}}{w}$
$\tanh z = \frac{\sinh z}{\cosh z}$	$\sin^{-1} w = \tan^{-1} \frac{w}{\sqrt{1 - w^2}}$
$\ln w = 2 \tanh^{-1} \frac{w - 1}{w + 1}$	$\log_b w = \frac{\ln w}{\ln b}$
$w^t = e^{t \ln w}$	$\cosh^{-1} = \ln(w + \sqrt{w^2 - 1})$
$\tan^{-1}(y/x)$	$\sinh^{-1} = \ln(w + \sqrt{w^2 + 1})$
$\sqrt{x^2 - y^2}$	$\sqrt{w} = \sqrt{(w + 1/4)^2 - (w - 1/4)^2}$

# There's very few multiplications in this...one at the beginning or end

- And really no divisions.



# People still making improvements/updates

2156

IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 27, NO. 9, SEPTEMBER 2019

## Generalized Hyperbolic CORDIC and Its Logarithmic and Exponential Computation With Arbitrary Fixed Base

Yuanyong Luo<sup>1</sup>, Yuxuan Wang, Yajun Ha, *Senior Member, IEEE*, Zhongfeng Wang<sup>2</sup>, *Fellow, IEEE*, Siyuan Chen, and Hongbing Pan<sup>3</sup>

**Abstract**—This paper proposes a generalized hyperbolic COordinate Rotation Digital Computer (GH CORDIC) to directly compute logarithms and exponentials with an arbitrary fixed base. In a hardware implementation, it is more efficient than the state of the art which requires both a hyperbolic CORDIC and a constant multiplier. More specifically, we develop the theory of GH CORDIC by adding a new parameter called base to the conventional hyperbolic CORDIC. This new parameter can be used to specify the base with respect to the computation of logarithms and exponentials. As a result, the constant multiplier is no longer needed to convert base  $e$  (Euler's number) to other values because the base of GH CORDIC is adjustable. The proposed methodology is first validated using MATLAB with extensive vector matching. Then, example circuits with 16-bit fixed-point data are implemented under the TSMC 40-nm CMOS technology. Hardware experiment shows that at the highest frequency of the state of the art, the proposed methodology saves 27.98% area, 50.69% power consumption, and 6.67% latency when calculating logarithms; it saves 13.09% area, 40.05% power consumption, and 6.67% latency when computing exponentials. Both calculations do not compromise accuracy. Moreover, it can increase 13% maximum frequency and reduce up to 17.65% latency accordingly compared to the state of the art.

**Index Terms**—Architecture, exponential, generalized hyperbolic COordinate Rotation Digital Computer (GH CORDIC),

evaluate logarithms and exponentials: approximation method and iterative method. Although loads of well-related research achievements have been proposed on these methods, there is still plenty of room for improvement. First, current approaches do not support easy porting to other fixed bases while they are needed. Second, current approaches still have room to further reduce the hardware overheads. In this paper, we will propose a promising solution to abovementioned concerns.

The following literature addresses the evaluation of logarithms and exponentials using the approximation method. [1]–[4] evaluate binary logarithms and exponentials via simple piecewise linear approximation. When the output approaches zero, this method encounters notably large relative error. In order to overcome this shortage, Nam *et al.* [5] perform finer subdivisions around the output of zero since the error increases as the output value gets closer to zero. Subsequently, they have designed a processor of the logarithmic number system for 3-D graphics. The main shortcoming of a simple linear approximation method is the high relative error with limited lookup tables. Paul *et al.* [6] use a second-order polynomial approximation method to reduce the relative error. The main contribution of [6] is approximating the multiplication



# For Upcoming Week 5 Assignments

- We'll have to calculate angles and magnitudes of 2D vectors (for future weeks with RFSoc).
- We need to write one and also make it AXIS-compliant



# CORDIC Convergence

- If you sum up all the possible angle they converge to about 99.88 degrees
- Dictates the range over which cordic functions can “converge” through multiple iterations
- Also when doing atan, you’ll likely need to do some quadrant determination to rotate prior to running.

i	$\alpha_i = \tan^{-1}(2^{-i})$	
	Degrees	Radians
0	45.00	0.7854
1	26.57	0.4636
2	14.04	0.2450
3	7.13	0.1244
4	3.58	0.0624
5	1.79	0.0312
6	0.90	0.0160
7	0.45	0.0080
8	0.22	0.0040
9	0.11	0.0020



# Returning to Generalized CORDIC

- The three equations we're iterating on can be generalized to this format

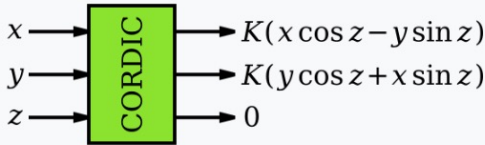
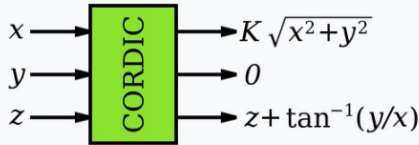
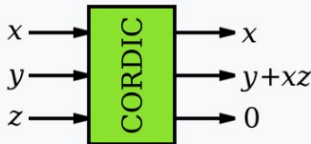
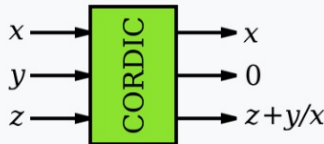
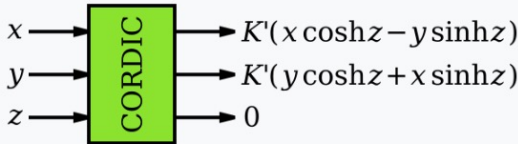
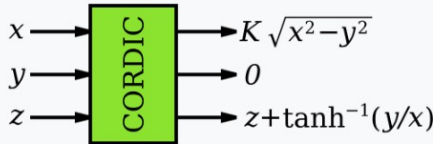
Diagram illustrating the generalized CORDIC equations and their components:

$$\begin{aligned}x_{i+1} &= x_i - \mu d_i y_i 2^{-i} \\y_{i+1} &= y_i + d_i x_i 2^{-i} \\z_{i+1} &= z_i + d_i \alpha_i\end{aligned}$$

Annotations:

- $z$  is our angle accumulator (points to  $z_{i+1}$ )
- $\mu$  is settable constant (points to  $\mu$ )
- $d_i$  is our control/feedback function for locking into a target (points to  $d_i$ )
- $\text{sgn}(\theta)$  in our walkthrough example (points to  $d_i$ )
- $2^{-i}$  are the  $\tan(\alpha_i)$  from our original example (points to  $2^{-i}$ )

# Different Modes

Mode	Rotation	Vectoring
	$d_i = \text{sgn}(z_i), \quad z \rightarrow 0$	$d_i = -\text{sgn}(y_i), \quad y \rightarrow 0$
<b>Circular</b> $\mu = 1$ $\alpha_i = \tan^{-1} 2^{-i}$		
<b>Linear</b> $\mu = 0$ $\alpha_i = 2^{-i}$		
<b>Hyperbolic</b> $\mu = -1$ $\alpha_i = \tanh^{-1} 2^{-i}$		

- In hyperbolic mode, iterations 4, 13, 40, 121, ...,  $j, 3j+1, \dots$  must be repeated. The constant  $K'$  given below accounts for this.
- $K = 1.646760258121\dots$
- $1/K = 0.607252935009\dots$
- $K' = 0.8281593609602\dots$
- $1/K' = 1.207497067763\dots$

# CORDIC

- You can use these outputs to generate all these weird things

## Directly computable functions [\[ edit | edit source \]](#)

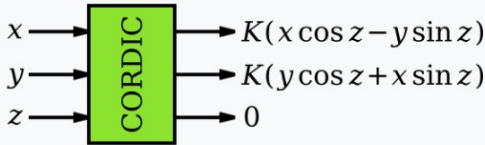
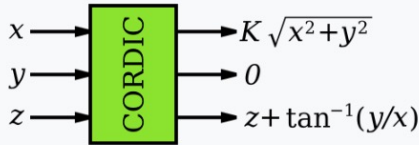
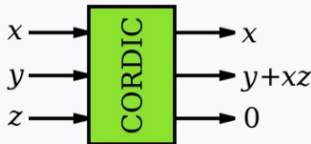
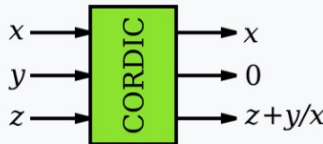
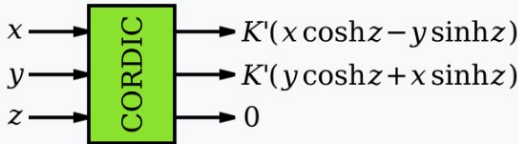
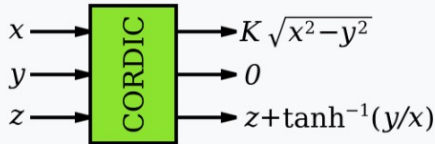
$\sin z$	$\cos z$
$\tan^{-1} z$	$\sinh z$
$\cosh z$	$\tanh^{-1} z$
$y/x$	$xz$
$\tan^{-1}(y/x)$	$\sqrt{x^2 + y^2}$
$\sqrt{x^2 - y^2}$	$e^z = \sinh z + \cosh z$

## Indirectly computable functions [\[ edit | edit source \]](#)

In addition to the above functions, a number of other functions can be produced by combining the results of previous computations:

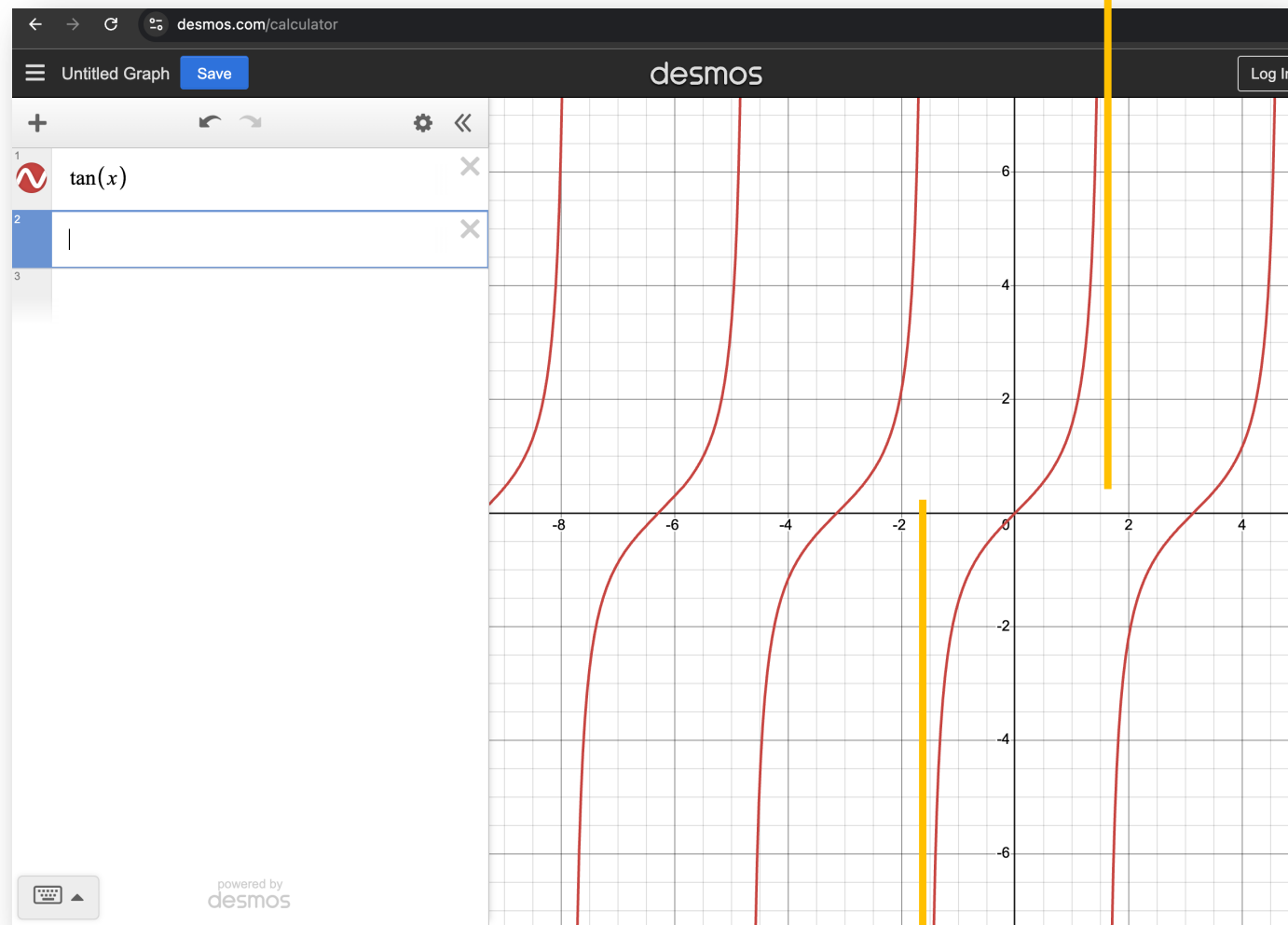
$\tan z = \frac{\sin z}{\cos z}$	$\cos^{-1} w = \tan^{-1} \frac{\sqrt{1 - w^2}}{w}$
$\tanh z = \frac{\sinh z}{\cosh z}$	$\sin^{-1} w = \tan^{-1} \frac{w}{\sqrt{1 - w^2}}$
$\ln w = 2 \tanh^{-1} \frac{w - 1}{w + 1}$	$\log_b w = \frac{\ln w}{\ln b}$
$w^t = e^{t \ln w}$	$\cosh^{-1} = \ln(w + \sqrt{w^2 - 1})$
$\tan^{-1}(y/x)$	$\sinh^{-1} = \ln(w + \sqrt{w^2 + 1})$
$\sqrt{x^2 - y^2}$	$\sqrt{w} = \sqrt{(w + 1/4)^2 - (w - 1/4)^2}$

# How to Actually Get $\sqrt{a}$ ?

Mode	Rotation	Vectoring
	$d_i = \text{sgn}(z_i), \quad z \rightarrow 0$	$d_i = -\text{sgn}(y_i), \quad y \rightarrow 0$
<b>Circular</b> $\mu = 1$ $\alpha_i = \tan^{-1} 2^{-i}$		
<b>Linear</b> $\mu = 0$ $\alpha_i = 2^{-i}$		
<b>Hyperbolic</b> $\mu = -1$ $\alpha_i = \tanh^{-1} 2^{-i}$		

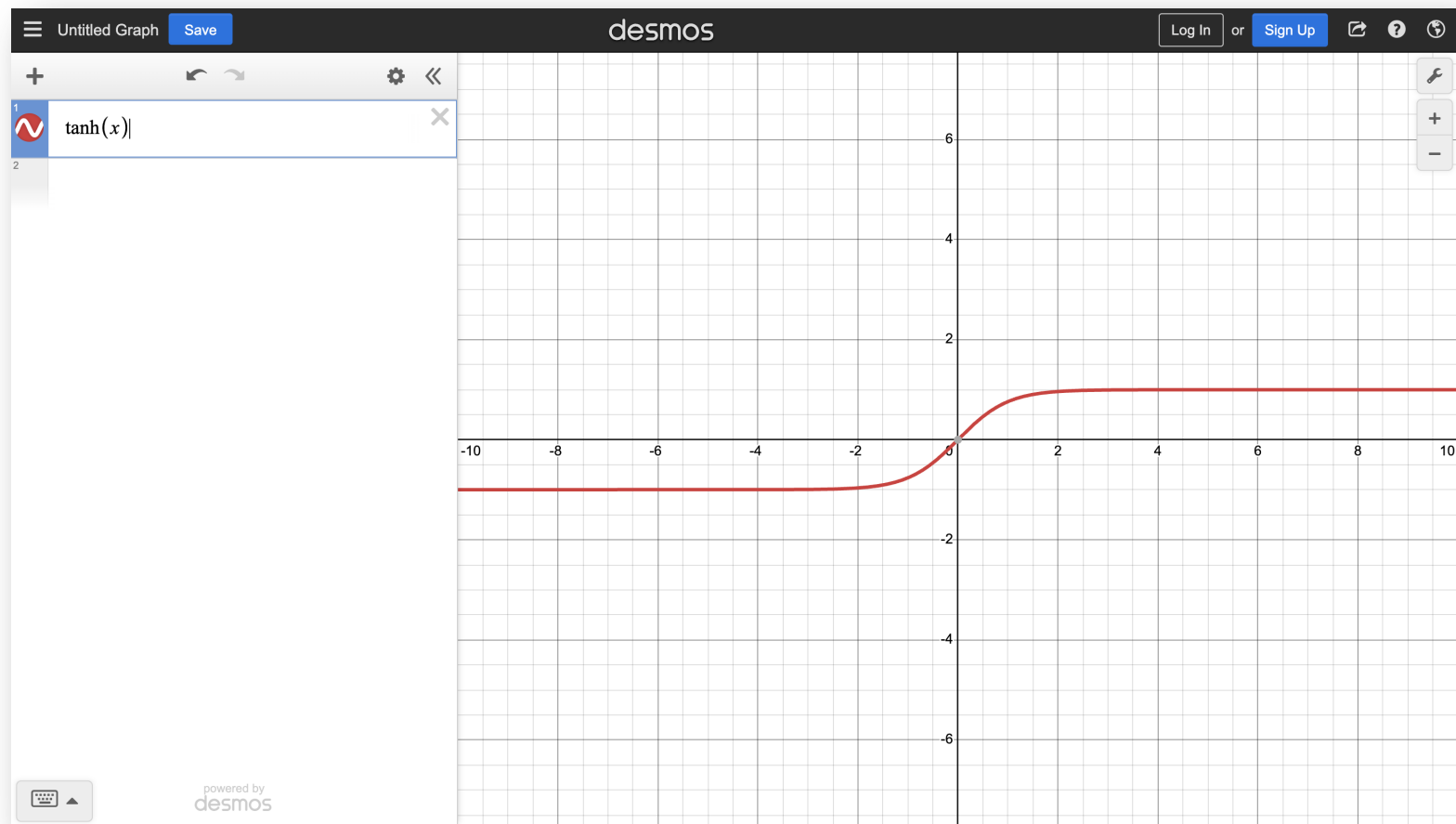
- In hyperbolic mode, iterations 4, 13, 40, 121, ...,  $j, 3j+1, \dots$  must be repeated. The constant  $K'$  given below accounts for this.
- $K = 1.646760258121\dots$
- $1/K = 0.607252935009\dots$
- $K' = 0.8281593609602\dots$
- $1/K' = 1.207497067763\dots$

# Observe $\tan(x)$



# Observe $\tanh(x)$

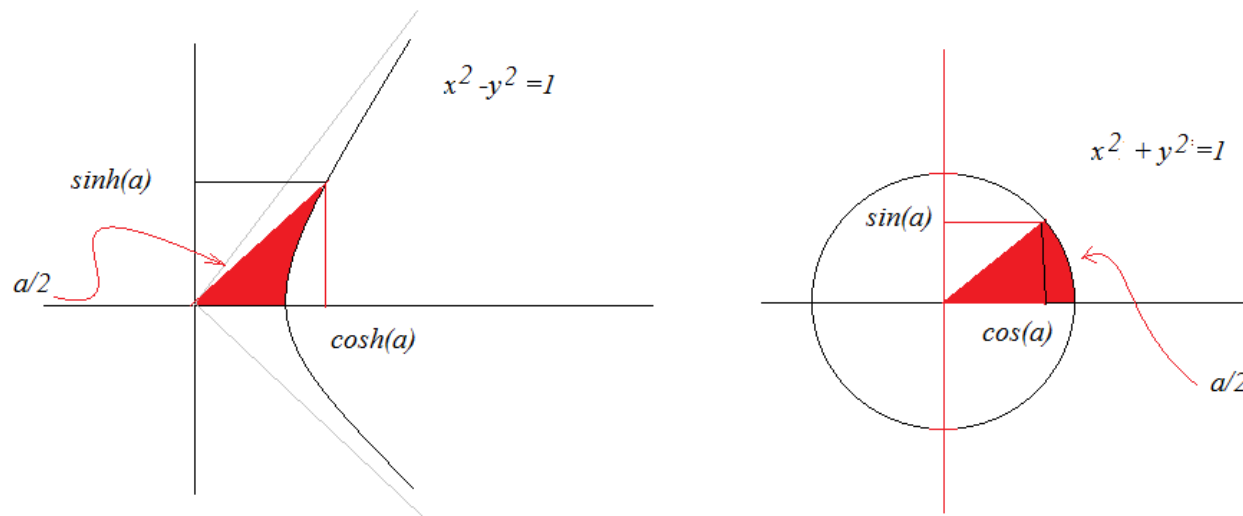
- Just a different function...still get our values for lookup from it...but kinda weird...





# Hyperbolic Functions

- Whereas regular trig functions are following around the unit circle...  $x^2 + y^2 = 1$



- Hyperbolic trig functions are following the unit hyperbola:  $x^2 - y^2 = 1$

Mode	Rotation	Vectoring
	$d_i = \text{sgn}(z_i), \quad z \rightarrow 0$	$d_i = -\text{sgn}(y_i), \quad y \rightarrow 0$
<b>Circular</b> $\mu = 1$ $\alpha_i = \tan^{-1} 2^{-i}$	$x \rightarrow \text{CORDIC} \rightarrow K(x \cos z - y \sin z)$ $y \rightarrow \text{CORDIC} \rightarrow K(y \cos z + x \sin z)$ $z \rightarrow \text{CORDIC} \rightarrow 0$	$x \rightarrow \text{CORDIC} \rightarrow K \sqrt{x^2 + y^2}$ $y \rightarrow \text{CORDIC} \rightarrow 0$ $z \rightarrow \text{CORDIC} \rightarrow z + \tan^{-1}(y/x)$
<b>Linear</b> $\mu = 0$ $\alpha_i = 2^{-i}$	$x \rightarrow \text{CORDIC} \rightarrow x$ $y \rightarrow \text{CORDIC} \rightarrow y + xz$ $z \rightarrow \text{CORDIC} \rightarrow 0$	$x \rightarrow \text{CORDIC} \rightarrow x$ $y \rightarrow \text{CORDIC} \rightarrow 0$ $z \rightarrow \text{CORDIC} \rightarrow z + y/x$
<b>Hyperbolic</b> $\mu = -1$ $\alpha_i = \tanh^{-1} 2^{-i}$	$x \rightarrow \text{CORDIC} \rightarrow K'(x \cosh z - y \sinh z)$ $y \rightarrow \text{CORDIC} \rightarrow K'(y \cosh z + x \sinh z)$ $z \rightarrow \text{CORDIC} \rightarrow 0$	$x \rightarrow \text{CORDIC} \rightarrow K \sqrt{x^2 - y^2}$ $y \rightarrow \text{CORDIC} \rightarrow 0$ $z \rightarrow \text{CORDIC} \rightarrow z + \tanh^{-1}(y/x)$

• In hyperbolic mode, iterations 4, 13, 40, 121, ...,  $j, 3j+1, \dots$  must be repeated. The constant  $K'$  given below accounts for this.  
 •  $K = 1.646760258121\dots$   
 •  $1/K = 0.607252935009\dots$   
 •  $K' = 0.8281593609602\dots$   
 •  $1/K' = 1.207497067763\dots$

$$x_{i+1} = x_i - \mu d_i y_i 2^{-i}$$

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

$$z_{i+1} = z_i + d_i \alpha_i$$

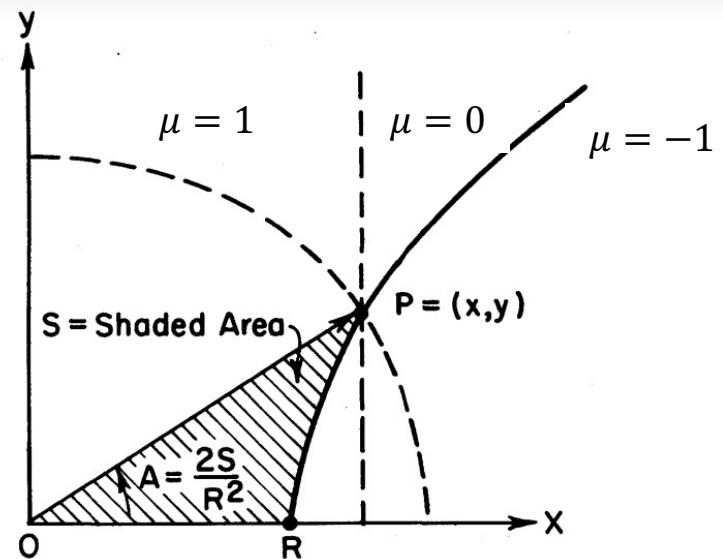


Figure 1—Angle  $A$  and Radius  $R$  of the vector  $P = (x, y)$

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

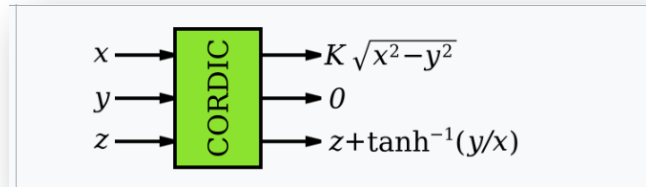
# How to Actually Get $\sqrt{a}$ ?

- You then need to do:

- $x = a + 0.25$

- $y = a - 0.25$

- So that... $\sqrt{(a + 0.25)^2 - (a - 0.25)^2} = \sqrt{a}$



- AMD/Xilinx has a pretty decent writeup of how to do it in a low-level digital form.

# From the Xilinx/AMD Docs...

That is, given input  $x$ , it computes the output  $\text{sqrt}(x)$ . The CORDIC processor is implemented using building blocks from the Xilinx blockset.

- The square root is calculated indirectly by the CORDIC algorithm by applying the identity listed as follows.  $\text{sqrt}(w) = \text{sqrt}\{(w + 0.25)^2 - (w - 0.25)^2\}$
- The CORDIC square root algorithm is implemented in the following 4 steps:
  1. Co-ordinate Rotation: The CORDIC algorithm converges only for positive values of  $x$ . If  $x < \text{zero}$ , the input data is converted to a non-negative number. If  $x = 0$ , a zero detect flag is passed to the co-ordinate correction stage. The square root circuit has been designed to converge for all values of  $x$ , except for the most negative value.
  2. Normalization: The CORDIC algorithm converges only for  $x$  between 0.25 (inclusive) and 1. During normalization, the input  $x$  is shifted to the left till it has a 1 in the most significant non-signed bit. If the left shift results in an odd number of shift values, a right shift is performed resulting in an even number of left shifts. The shift value is divided by 2 and passed on to the co-ordinate correction stage. The square root is derived using the identity  $\text{sqrt}(w) = \text{sqrt}\{(w + 0.25)^2 - (w - 0.25)^2\}$ . Based on this identity the input  $x$  gets mapped to,  $X = x + 0.25$  and  $Y = x - 0.25$ .
  3. Hyperbolic Rotations: For  $\text{sqrt}(X^2 - Y^2)$  calculation, the resulting vector is rotated through progressively smaller angles, such that  $Y$  goes to zero.
  4. Co-ordinate Correction: If the input was negative and a left shift was applied to  $x$ , this step assigns the appropriate sign to the output and multiplies it with  $2^{-\text{shift}}$ . If the input was zero, the zero detect flag is used to set the output to 0.

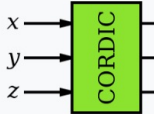
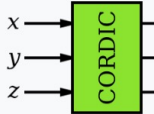
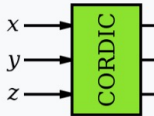
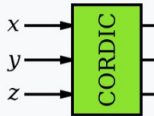
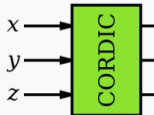
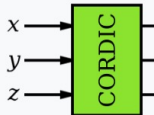
# CORDIC Square Root Convergence

- If you sum up all the possible angle they converge to about 99.88 degrees
- Dictates the range over which cordic functions can “converge” through multiple iterations
- Same thing with hyperbolic...but their possible angle total approaches: 64.74 degrees (significantly lower since can't do for  $i=0$ )

i	$\alpha_i = \tan^{-1}(2^{-i})$	
	Degrees	Radians
0	45.00	0.7854
1	26.57	0.4636
2	14.04	0.2450
3	7.13	0.1244
4	3.58	0.0624
5	1.79	0.0312
6	0.90	0.0160
7	0.45	0.0080
8	0.22	0.0040
9	0.11	0.0020



# Another Interesting Thing...

Mode	Rotation	Vectoring
	$d_i = \text{sgn}(z_i), \quad z \rightarrow 0$	$d_i = -\text{sgn}(y_i), \quad y \rightarrow 0$
<b>Circular</b> $\mu = 1$ $\alpha_i = \tan^{-1} 2^{-i}$		
<b>Linear</b> $\mu = 0$ $\alpha_i = 2^{-i}$		
<b>Hyperbolic</b> $\mu = -1$ $\alpha_i = \tanh^{-1} 2^{-i}$		
<ul style="list-style-type: none"> <li>• In hyperbolic mode, iterations 4, 13, 40, 121, ..., <math>j, 3j+1, \dots</math> must be repeated. The constant <math>K'</math> given below accounts for this.</li> <li>• <math>K = 1.646760258121\dots</math></li> <li>• <math>1/K = 0.607252935009\dots</math></li> <li>• <math>K' = 0.8281593609602\dots</math></li> <li>• <math>1/K' = 1.207497067763\dots</math></li> </ul>		

# Source...

- This 1971 paper is what everyone points to as justification for the repeated sequence for convergence

TABLE II—Shift Sequences for a binary code

radix $\rho$	coordinate system $m$	shift sequence $F_{mi}; i \geq 0$	domain of convergence $\max  A_0 $	radius factor $K$
2	1	0, 1, 2, 3, 4, $i, \dots$	$\sim 1.74$	$\sim 1.65$
2	0	1, 2, 3, 4, 5, $i+1, \dots$	1.0	1.0
2	-1	1, 2, 3, 4, 4, 5, $\dots$ *	$\sim 1.13$	$\sim 0.80$

\* for  $m = -1$  the following integers are repeated:  
{4, 13, 40, 121,  $\dots$ ,  $k$ ,  $3k+1, \dots$ }

Table II shows some  $F$  sequences, convergence domains, and radius factors for a binary code.

The hyperbolic mode ( $m = -1$ ) is somewhat complicated by the fact that for  $\alpha_i = \tanh^{-1}(2^{-i})$  the convergence criterion (23) is not satisfied. However, it can be shown that

$$\alpha_i - \left( \sum_{j=i+1}^{n-1} \alpha_j \right) - \alpha_{3i+1} < \alpha_{n-1} \quad (33)$$

and that therefore if the integers {4, 13, 40, 121,  $\dots$ ,  $k$ ,  $3k+1, \dots$ } in the  $F_i$  sequence are repeated then (23) becomes true.

The magnitude of each element of the sequence may be predetermined, but the direction of rotation must be determined at each step such that

$$|A_{i+1}| = |A_i| - \alpha_i \quad (22)$$

The sum of the remaining rotations must at each step be sufficient to bring the angle to at least within  $\alpha_{n-1}$  of zero, even in the extreme case where  $A_i = 0$ ,  $|A_{i+1}| = \alpha_i$ . Thus,

$$\alpha_i - \sum_{j=i+1}^{n-1} \alpha_j < \alpha_{n-1} \quad (23)$$

The domain of convergence is limited by the sum of the rotations.

$$|A_0| - \sum_{j=0}^{n-1} \alpha_j < \alpha_{n-1} \quad (24)$$

$$1 + \sum_{j=0}^{n-1} \alpha_j \quad (25)$$

es to within  $\alpha_{n-1}$  of zero  
the following theorem.

$$1 + \sum_{j=i}^{n-1} \alpha_j \quad (26)$$

- Some students confirmed this last year...

## Proof that repetition works in CORDIC algorithm

Updated 12 months ago by Andi Qu 🟡

I told one of my 6.2050 friends about the problem and we managed to solve it I think (thanks Steven)!

First, note that  $\tanh^{-1}$  is convex. By Jensen's inequality, we have:

$$\begin{aligned} \sum_{i=n+1}^{\infty} \tanh^{-1}(2^{-i}) &= \lim_{k \rightarrow \infty} \sum_{i=n+1}^k \tanh^{-1}(2^{-i}) \\ &\geq \lim_{k \rightarrow \infty} (k - n) \tanh^{-1} \left( \frac{1}{k - n} \sum_{i=n+1}^k 2^{-i} \right) \\ &= \lim_{k \rightarrow \infty} (k - n) \tanh^{-1} \left( \frac{2^{-n}(1 - 2^{n-k})}{k - n} \right) \\ &= \lim_{k \rightarrow \infty} (k - n) \tanh^{-1} \left( \frac{2^{-n}}{k - n} \right) \end{aligned}$$

Then by L'Hopital's rule:

$$\begin{aligned} \lim_{k \rightarrow \infty} k \tanh^{-1}(x/k) &= \lim_{k \rightarrow \infty} \frac{(\tanh^{-1}(x/k))'}{(1/k)'} \\ &= \lim_{k \rightarrow \infty} \frac{\frac{1}{1-(x/k)^2} \cdot (-\frac{x}{k^2})}{-1/k^2} \\ &= \lim_{k \rightarrow \infty} \frac{x}{1 - (x/k)^2} \\ &= x \end{aligned}$$

So now our inequality becomes:

$$2^{-n} + \tanh^{-1}(2^{-(3n+1)}) > \tanh^{-1}(2^{-n})$$

which is true by the Taylor expansion of  $\tanh^{-1}$ . (Note that  $3n + 1$  is the largest integer that satisfies this inequality.)



# Square root convergence in Practice

- Normalization: **The CORDIC algorithm converges only for  $x$  between 0.25 (inclusive) and 1.** During normalization, the input  $x$  is shifted to the left till it has a 1 in the most significant non-signed bit. If the left shift results in an odd number of shift values, a right shift is performed resulting in an even number of left shifts. The shift value is divided by 2 and passed on to the co-ordinate correction stage. The square root is derived using the identity  $\text{sqrt}(w) = \text{sqrt}\{(w + 0.25)^2 - (w - 0.25)^2\}$ . Based on this identity the input  $x$  gets mapped to,  $X = x + 0.25$  and  $Y = x - 0.25$ .

## Overcoming Algorithm Input Range Limitations

Many square root algorithms normalize the input value,  $v$ , to within the range of  $[0.5, 2)$ . This pre-processing as well as large input value ranges.

<https://www.mathworks.com/help/fixedpoint/ug/compute-square-root-using-cordic.html>

# So these are coming from the assertion

- Keeping in mind:  $\sqrt{(a + 0.25)^2 - (a - 0.25)^2}$   
which is forcing the starting value of the x and y values...

# Wrote some code to test it

```
import sys

a = float(sys.argv[1])

x = a+.25
y = a-.25

for i in range(1,20):
    if y > 0:
        xn = x - 1/(2**i)*y
        yn = y - 1/(2**i)*x
    else:
        xn = x + 1/(2**i)*y
        yn = y + 1/(2**i)*x
    print(f"x:{xn}, y:{yn}")
    x = xn
    y = yn
print(x/0.828)
```

# So I ran it...

```
python3 cordic_test.py 4
x:2.375, y:1.625
x:1.96875, y:1.03125
x:1.83984375, y:0.78515625
x:1.790771484375, y:0.670166015625
x:1.7698287963867188, y:0.6142044067382812
x:1.760231852531433, y:0.5865508317947388
x:1.7556494241580367, y:0.572799020446837
x:1.7534119279844163, y:0.5659410148837196
x:1.7523065744397215, y:0.562516382211875
x:1.7517572420352177, y:0.5608051453227738
x:1.751483411397853, y:0.5599497951069363
x:1.751346704904907, y:0.5595221868522006
x:1.7512784038567073, y:0.559308399412637
x:1.7512442663811572, y:0.5592015098616203
x:1.7512272009053924, y:0.559148066127905
x:1.7512186689829967, y:0.5591213445214459
x:1.7512144032256685, y:0.5591079837833097
x:1.7512122703979716, y:0.5591013034305142
x:1.7512112039968648, y:0.5590979632581845
2.1149893768078076
```

# So I ran it...

```
python3 cordic_test.py 0.81
x:0.78, y:0.0300000000000000027
x:0.7725, y:-0.16499999999999998
x:0.751875, y:-0.06843749999999998
x:0.74759765625, y:-0.021445312499999987
x:0.746927490234375, y:0.001917114257812512
x:0.7468975353240966, y:-0.009753627777099597
x:0.746821335107088, y:-0.003918490782380092
x:0.7468060285024694, y:-0.0010012199421180297
x:0.7468040729947699, y:0.0004573855823008558
x:0.7468036263291622, y:-0.0002719152702330992
x:0.7468034935580341, y:9.273493793543702e-05
x:0.7468034709176684, y:-8.95901337340049e-05
x:0.7468034599813728, y:1.5723993369995485e-06
x:0.7468034598854011, y:-4.400886653100416e-05
x:0.7468034585423571, y:-2.121823359993113e-05
x:0.7468034582185925, y:-9.822917154887839e-06
x:0.7468034581436496, y:-4.125258934836321e-06
x:0.7468034581279129, y:-1.2764298250964468e-06
x:0.7468034581254783, y:1.47984729743475e-07
0.9019365436298048
```

# Conclusions

- Seems to converge for input values of 0 to 2
- Beyond that it doesn't converge, and this is because for hyperbolics,  $\mu = -1$  so:

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

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

- Whereas in original:

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

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

# Anyways

- CORDICs
- Wednesday we'll either start talking about I/Q signaling or do some other stuff on AXIS...probably I/Q