Contents

Introduction

Often we take our calculators for granted. We carry devices in our pockets that can graph extremely complicated functions and evaluate various functions and expressions that would be unreasonable to do by hand. However, it's easy to forget that there are still methods that must be used for computers to complete difficult calculations. This paper will provide an overview of the algorithms that are used in real life for such math. I'll be explaining the math behind the algorithms, describing how they are implemented in code, providing Python snippets, and writing my own implementations in Rust.

It's important to note that, in the majority of cases, these algorithms are not implemented in code. Because of how fundamental they are, CPUs provide built-in methods for performing the calculations, with the algorithms themselves implemented with transistor logic. These implementations are necessarily incredibly faster than any code I could write. I'm doing this purely for education. As we go along, I'll be comparing the output of my code to these CPU implementations.

Trigonometric functions with CORDIC

The math

The basic idea behind the CORDIC algorithm is to perform a rotation of a unit vector to a particular angle β by performing many increasingly small rotations towards the desired angle, starting at 45° and halving each time.

Each step angle is defined by $\gamma_i = \arctan(2^{-i})$. Starting with $v_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $v_{i+1} = R_i v_i$ where R_i is a rotation matrix $\begin{bmatrix} \cos(\gamma_i) & -\sin(\gamma_i) \\ \sin(\gamma_i) & \cos(\gamma_i) \end{bmatrix}$.

R_i can be simplified to $\cos(\gamma_i) \begin{bmatrix} 1 & -\tan(\gamma_i) \\ \tan(\gamma_i) & 1 \end{bmatrix}$, and since γ_i is defined with arctan, we arrive at

$$R_i = \cos(\arctan(2^{-i})) \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix}$$

where σ_i is either 1 or −1, depending on the direction in which the rotation must go to get closer to the desired final angle.

The identity $\cos(\gamma_i) = \frac{1}{\sqrt{1+\tan(\gamma_i)^2}}$ can be used to simplify the scalar of R_i to $K_i = \frac{1}{\sqrt{1+2^{-2i}}}$. This K can be extracted from the algorithm and applied at the end as a scaling factor

$$K(n) = \prod_{i=0}^{n-1} K_i$$

where n is the number of steps.

Thus, $\begin{bmatrix} \cos(\beta) \\ \sin(\beta) \end{bmatrix}$ can be approximated as $K(n) \begin{bmatrix} x_n \\ y_n \end{bmatrix}$ where

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

for n iterations.

The software

Instead of doing matrix multiplication for each iteration, you can simply keep track of x_i and y_i as separate variables and perform scalar multiplication $x_{i+1} = -\sigma_i 2^{-i} x_i$ (and similar for y). To greatly improve performance and complexity, most implementations decide on a static number of iterations and pre-compute $K(n)$ as a constant scaling factor. In addition, in order to determine σ_i, each step angle is precomputed, and the current angle is compared with the desired angle on each iteration.

Code

```
def cordic(beta):
  theta = 0.0 # stores current angle
  point = (1.0, 0.0)
  p2i = 1.0 # stores 2^(-i)

  # where STEPS is a list of precomputed step angles
  for gamma in STEPS:
    sigma = 1 if theta < beta else -1
    theta += sigma * gamma
    point = (point[0] - sigma * point[1] * p2i, point[1] + sigma * p2i * point[0])
    p2i /= 2.0

  # where K is precomputed
  return (point[0] * K, point[1] * K)
```

Output

Using unit tests, I was able to verify that this algorithm matches the output of built-in CPU insructions within 12 decimal places.