Difficult Math Done by Computers

Devin Droddy

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.

Square root with Heron's method

The math

Heron's method is a simple algorithm for finding a square root discovered in ancient Greece. It actually happens to be a special case of the Newton–Raphson method for finding the roots of a function. Starting with an initial estimate x_0 (the value of which will be discussed later), you iterate using $x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$. This can be derived as follows.

Where x is the estimate of the desired square root $\sqrt{S}$ and ε is the error in the estimate, $S = (x + \varepsilon)^2$, which expands to $x^2 + 2x\varepsilon + \varepsilon^2$. This can be rearranged to

$$\varepsilon = \frac{S - x^2}{2x + \varepsilon}$$

Assuming that ε is much smaller than x, the denominator can be approximated as just $2x$. Thus, we can compensate for the error by adding the fraction $\frac{\frac{S}{x}+x}{2}$ to x.

The software

Since we're working with floating-point numbers, there isn't much optimization we can do to the algorithm itself.

```
def sqrt(s):
  if s == 0:
    return 0
  if s < 0:
    raise ValueError("Complex math not yet implemented")
  x = initial_estimate(s)
  # 3 or 4 iterations is usually more than enough.
  for _ in range(ITERATIONS):
    x = 0.5 * (x + (s / x))

  return x
```

The initial estimate

There's a neat trick we can do, leveraging floating-point numbers, to find a strong initial estimate. Floating-point numbers store decimal numbers in a sort of scientific notation, with a number of bits defining an exponent and a number of bits defining the number. Since $\sqrt{x} = x^{\frac{1}{2}}$, we can get a very strong initial estimate by halving the exponent of the floating-point number.

This is a bit easier to do in Rust than in Python. Either way, it's pretty technical. You have to extract the raw binary representation of the floating-point number, do some bitwise logic to extract the exponent and mantissa (the name for the numeric value), and some bitshifting. Here's the code:

```python
import struct

def initial_estimate(s):
  # Extracts the binary representation of the floating-point number
  bits = struct.unpack('>Q', struct.pack('>d', n))[0]

  exponent = (bits >> 52) & 0x7FF0000000000000
  mantissa = bits & 0x000FFFFFFFFFFFFF

  # The exponent is shifted by 1023 in the binary representation, so to halve it we
  # have to adjust
  # The division by 2 could be optimized as a right bit shift, but the compiler will
  # probably do that for you and division is easier to read
  new_exponent = ((exponent - 1023) / 2 + 1023)
  estimate_bits = (new_exponent << 52) | mantissa

  # Back to a floating-point number
  return struct.unpack('>d', struct.pack('>Q', estimate_bits))[0]
```

Output

This one gets to 14 decimal places of precision! Additionally, using it to calculate K in the CORDIC algorithm doesn't reduce its precision at all. Nice!

Complex numbers

Representing complex numbers requires structuring data and adding special considerations to calculations in sofware. CPUs do not provide methods for complex math operations. For the square root, simply multiplying by i when $S < 0$ does the trick.

Natural logarithm

Despite extensive research, I was unable to find sufficient information on computer algorithms for natural logarithms in time to submit this assignment. I was referred to some textbooks, and a website documenting a particular C library, but nothing easily accessible with an in-depth explanation of the math. Some people claimed computers use Taylor series, sometimes with some kind of lookup table.