fastInvSqRtAlgDynProg

March 9, 2021

Fast Inverse Square Root Algorithm

$$f(x) = \frac{1}{\sqrt{x}}$$

To calculate the inverse square root in C, some would write:

// float y = 1 / sqrt(x);

(sqrt) has was written as a calculation in the math.h file. This works, but is inefficient. Others have found ways through bit manipulation to approximate the inverse square root for z-space applications and renders:

// Fast Inverse Square Root Algorithm

```
  float Q_rsqrt( float number ) )

{

    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;                       // bit shift
    i  = 0x5f3759df - ( i >> 1 );               // logarithm of f
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
    y  = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iteration, can be removed

    return y;
}
```

The length of a three dimensional vector is

$$\sqrt{x^2 + y^2 + z^2}$$

In two dimensions, Pythagorean Theorem

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

If we divide the length of the vector by itself

$$\frac{\sqrt{x^2+y^2+z^2}}{\sqrt{x^2+y^2+z^2}}$$

We would end up with:

0.1 1

Dividing x, y, or z by the length

$$\frac{x}{\sqrt{x^2+y^2+z^2}}$$

$$\frac{y}{\sqrt{x^2+y^2+z^2}}$$

$$\frac{z}{\sqrt{x^2+y^2+z^2}}$$

Or multiply x, y, or z with 1 divided by the length

$$x * \frac{1}{\sqrt{x^2+y^2+z^2}}$$

$$y * \frac{1}{\sqrt{x^2+y^2+z^2}}$$

$$z * \frac{1}{\sqrt{x^2+y^2+z^2}}$$

Calculating x squared + y squared + z squared is efficient

$$x^2+y^2+z^2$$

$$x*x+y*y+z*z$$

Calculating Square Roots and Division are not efficient

$$\frac{1}{\sqrt{x^2+y^2+z^2}}$$

With 1% error, the Fast Inverse Square Root Algorithm is on the order of correct, as well as able to keep up with thousands of processes at once… by way of taking the log of the function and bit-shifting.

Binary Numbers // Scientific notation is used instead for the bit shifting

// i = * (long *) &y; // float manipulation

// i = 0x5f3759df - (i >> 1); // bit shift

1 IEEE 754

[255 000] 0 0000100 00000000 00000000 00000000

The first bit is the signed bit, used to denote positive or negative intergers

The rest of the remaining first eight bits tell us the Nth power of integer from a range of 0 to 255.

$$+x * 2^4$$

Negative exponents are also needed, so the range is shifted to 128 to -127

$$+x * 2^{-127}$$

1.1 &

$$+x * 2^{128}$$

By shifting the decimal point one digit to the left before the 23 bit Mantissa, giving us a representation between 1 and 2 to work with the binary language.

Standard IEEE 754 includes normalised numbers, denormalised numbers, NaN, infinities, 0 and -0.

$$(1 + \frac{M}{2^{23}}) * 2^E - 127$$

Take the logarithm of this expression, since we are using modulo base - 2

$$\log_2((1 + \frac{M}{2^{23}}) * 2^E - 127)$$

$$\log_2(1 + x) \approx x$$

Adding Mu to approximate.

$$\log_2(1 + x) \approx x + \mu$$

$$\mu = 0.0430$$

$$\log_2((1 + \frac{M}{2^{23}}) * 2^E - 127)$$

Reorder

$$(\frac{M}{2^{23}}) + \mu + E - 127$$

Reordered, again

$$\frac{1}{2^{23}}(M + 2^{23} * E) + \mu - 127$$

Convert the memory address using i = * (long *) &y; y is the memory address converted from a float to a long.C reads this address while retaining decimal point correctness, calling an address instead of a number. Bit Shifting a number to the left doubles it, bit shifting to the right divides it in half. The same can be done to an exponent. One direction squares, the other gives us the square root. Negative exponential 1/2 is equal to the equation needed.

$$\log(\frac{1}{\sqrt{y}}) = \log(y^{-\frac{1}{2}}) = -\frac{1}{2}\log(y)$$

Instead of "dividing by two" we bit shift to the right direction.

$$\Gamma = \frac{1}{\sqrt{y}}$$

$$\log(\Gamma) = log(\frac{1}{\sqrt{y}}) = -\frac{1}{2}\log(y)$$

$$\frac{1}{2^{23}}(M_\Gamma + 2^{23} * E_\Gamma) + \mu - 127 = -\frac{1}{2}(\frac{1}{2^{23}}(M_y + 2^{23} * E_y) + \mu - 127)$$

= 0x5f3759df - (i >> 1);

2 Newtonian Iteration

Find a root for given function

$$f(x) = 0$$

Finds an approximation and returns an approximation. One iteration is enough to get an error within 1%. Needs a function and derivative of that function. Takes x value and approximates how far it is from root by calculating f of x and f prime of x, its derivative.

$$f(x)$$

$$f'(x)$$

$$\Delta y$$

$$\frac{\Delta y}{\Delta x}$$

$$\frac{\Delta y}{\Delta x} * \Delta x = \Delta y$$

$$x_{new} = x - \frac{f(x)}{f'(x)}$$

y = y * (threehalfs - (x2 * y * y)); // Newtonian Iteration

$$f(y) = \frac{1}{y^2} - x$$

3 Adapted from John Cramack and YouTube Render from Nemean