Exploring the Quake III Fast Inverse Square Root Algorithm

Undergraduate Seminar Presentation

Daniel Harrington

University of Toronto

February 1st 2023

Outline

An Introduction

- Source code from a genre-defining 1999 multiplayer FPS
- A clever approximation of $\frac{1}{\sqrt{x}}$
- 4x faster solution with $< 1\%$ error
- Meta-manipulation of the C-language and IEEE Standard for Floating-Point Arithmetic
- Foggy origins

So What Is It?

The 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;                       // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iteration, this can be removed

    return y;
}
```

The Algorithm

```
float Q_rsqrt( float number )
{
















{
```

The Algorithm

```
float Q_rsqrt( float number )
{
     long i;           // Note long and float are both 32 bit
     float x2, y;









{
```

The Algorithm

```
float Q_rsqrt( float number )
{
    long i;          // Note long and float are both 32 bit
    float x2, y;
    const float threehalfs = 1.5F; // also 32 bit







{
```

The Algorithm

```
float Q_rsqrt( float number )
{
    long i;           // Note long and float are both 32 bit
    float x2, y;
    const float threehalfs = 1.5F; // also 32 bit
    x2 = number * 0.5F;
    y  = number;




{
```

The Algorithm

```
float Q_rsqrt( float number )
{
    long i;            // Note long and float are both 32 bit
    float x2, y;
    const float threehalfs = 1.5F; // also 32 bit
    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y; // Step 1




{
```

The Algorithm

```
float Q_rsqrt( float number )
{
    long i;           // Note long and float are both 32 bit
    float x2, y;
    const float threehalfs = 1.5F; // also 32 bit
    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y; // Step 1
    i  = 0x5f3759df - ( i >> 1 ); //Step 2




{
```

The Algorithm

```
float Q_rsqrt( float number )
{
    long i;          // Note long and float are both 32 bit
    float x2, y;
    const float threehalfs = 1.5F; // also 32 bit
    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y; // Step 1
    i  = 0x5f3759df - ( i >> 1 ); //Step 2
    y  = * ( float * ) &i;




{
```

The Algorithm

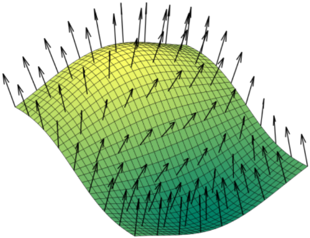
```
float Q_rsqrt( float number )
{
    long i;          // Note long and float are both 32 bit
    float x2, y;
    const float threehalfs = 1.5F; // also 32 bit
    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y; // Step 1
    i  = 0x5f3759df - ( i >> 1 ); //Step 2
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) ); // Step 3
    //  y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
{
```

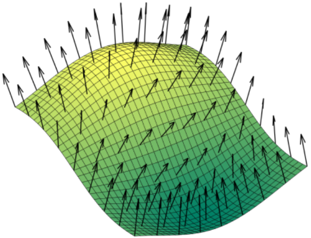
Why Inverse Square Root?

Why Inverse Square Root?

Surface Normals



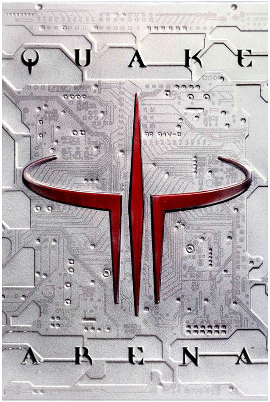
Surface Normals



$$\vec{v} = (v_1, v_2, v_3) \qquad ||\vec{v}|| = \sqrt{v_1^2 + v_2^2 + v_3^2} \qquad \hat{v} = \frac{\vec{v}}{||\vec{v}||}$$

Outline

The Origins



Quake III Cover Art

A screenshot of an in-game reflection

The Origins

- Copies of the code first appeared on Usenet and other forums in 2002/2003.

The Origins

- Copies of the code first appeared on Usenet and other forums in 2002/2003.
- Who was the codes author?

The Origins

- Copies of the code first appeared on Usenet and other forums in 2002/2003.
- Who was the codes author?
- How was the “Magic Number” constant derived?

The Search Begins

Forum Investigation begins in 2004:

Beyond3D

News Reviews Articles Interviews PR 3D Tables Forums About Us MyB3D

Consumer Graphics · Pro Graphics · GPGPU · Consoles · Displays · Games · Software · Processor · Games Development

Latest Reviews

- NVIDIA Fermi GPU and Architecture Analysis
- ATI Cypress Gaming Performance Analysis
- ATI Cypress GPU and Architecture analysis
- ATI RV740 GPU and Architecture Analysis
- NVIDIA GT200 GPU and Architecture Analysis

Latest Articles

- E3 2011: Behind Closed Doors - Witcher 2 Xbox 360... and a Nurse
- Q&A with Visceral's Technical Art Director Doug Brooks on Dead Space 2
- Travelling in Style: Beyond3D's C++ AMP contest - WINNER ANNOUNCED
- AMD's John Bridgman on Radeon, Linux and Open Source 3D
- Ian Buck - NVIDIA Tesla Launch
- Was Harry Potter actually into rendering? Beyond3D's first ever book review
- Andy Keane Interview & Tesla Adoption/Deployment
- Dave Kirk - NVIDIA Tesla Launch
- A speculative look on the Wii U GPU
- Diving into Anti-Aliasing

Origin of Quake3's Fast InvSqrt() - Page 1

Published on 29th Nov 2006, written by Rys for Software - Last updated: 21st Mar 2007

Introduction

Note! This article is a republishing of something I had up on my personal website a year or so ago before I joined Beyond3D, which is itself the culmination of an investigation started in April 2004. So if timeframes appear a little wonky, it's entirely on purpose! One for the geeks, enjoy.

Origin of Quake3's Fast InvSqrt()

To most folks the following bit of C code, found in a few places in the recently released Quake3 source code, won't mean much. To the Beyond3D crowd it might ring a bell or two. It might even make some sense.

```
float InvSqrt (float x){
    float xhalf = 0.5f*x;
    int i = *(int*)&x;
    i = 0x5f3759df - (i>>1);
    x = *(float*)&i;
    x = x*(1.5f - xhalf*x*x);
    return x;
}
```

Finding the inverse square root of a number has many applications in 3D graphics, not least of all the normalisation of 3D vectors. Without something like the *nrm* instruction in a modern fragment processor where you can get normalisation of an fp16 3-channel vector for free on certain NVIDIA hardware if you're (or the compiler is!) careful, or if you need to do it outside of a shader program for whatever reason, inverse square root is your friend. Most of you will know that you can calculate a square root using Newton-Raphson iteration and essentially that's what the code above does, but with a twist.

How the code works

The *magic* of the code, even if you can't follow it, stands out as the `i = 0x5f3759df - (i>>1);` line. Simplified, Newton-Raphson is an approximation that starts off with a guess and refines it with iteration. Taking advantage of the nature of 32-bit x86 processors, `i`, an integer, is initially set to the value of the floating point number you want to take the inverse square of, using an integer cast. `i` is then set to `0x5f3759df`, minus itself shifted one bit to the right. The right shift drops the least significant bit of `i`, essentially halving it.

Using the integer cast of the seeded value, `i` is reused and the initial guess for Newton is calculated using the magic seed value minus a free divide by 2 courtesy of the CPU.

But why that constant to start the guessing game? Chris Lomont wrote a paper analysing it while at Purdue in 2003. He'd seen the code on the *gamedev.net* forums and that's probably also where DemoCoder saw it before commenting in the first NV40 Doom3 thread on B3D. Chris's analysis for his paper explains it for those interested in the base math behind the implementation. Suffice to say the constant used to start the Newton iteration is a very clever one. The paper's summary wonders who wrote it and whether they got there by guessing or derivation.

So who did write it? John Carmack?

While discussing NV40's render path in the Doom3 engine as mentioned previously, the code was brought up and attributed to John Carmack; and he's the obvious choice since it appears in the source for one of his engines. Michael Abrash was mooted as a possible author too. Michael stands up here as x86 assembly optimiser extraordinaire, author of the legendary *Zen of Assembly Language* and *Zen of Graphics Programming* tomes, and employee of id during Quake's development where he worked alongside Carmack on optimising Quake's software renderer for the CPUs around at the time.

Asking John whether it was him or Michael returned a "not quite".

```
-----Original Message-----
```

Following the clues

2004

Figure: John Carmack - Lead Programmer of Quake

Following the clues

2004

(a) Michael Abrash

(b) Terje Mathisen

Figure: Experts in x86 assembly optimization

Following the clues

1997

Figure: James F. Blinn - American Computer graphics expert

Following the clues

1997

Following the clues

1994 - The first real lead!

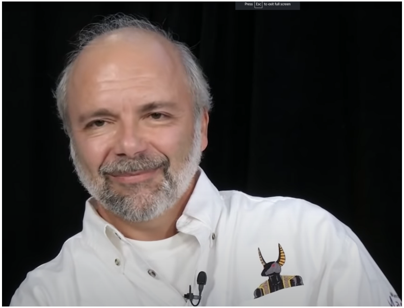


Figure: Gary Tarolli - Founder of 3dfx

Following the clues

1994 - A dead end to the search.

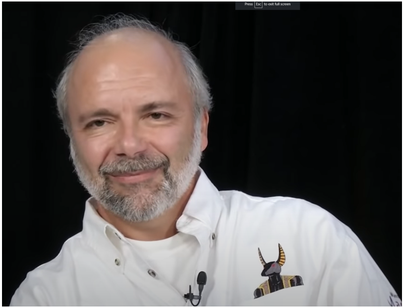


Figure: Gary Tarolli - Founder of 3dfx

The Author Comes Forward

What really happened

1986 William Kahan and K.C. Ng at Berkeley writes an unpublished paper on the technique for sqrt(x)

Figure: William Kahan - U of T Alumni / Turing laureate

The Author Comes Forward

What really happened

1980's Cleve Moler at Ardent Computer learns about the technique and shows Greg Walsh

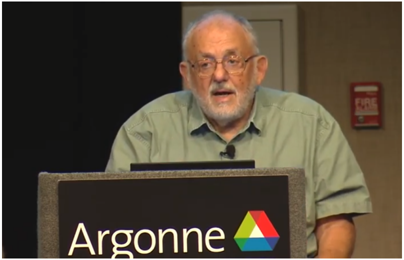


Figure: Cleve Moler - American Mathematician

The Author Comes Forward

Greg Walsh

Gregory Walsh - the man who authored the modern version of the code.

The Magic Number

So What about the magic number?

So What about the magic number?

2010 Still Unknown

1974 The oldest known use of the magic number in a PDP-11 Unix Manual

A PDP-11 System

Outline

Before we dive in

This algorithm relies heavily on the float representation of numbers, so we will go over those and it will be much clearer.

Before we dive in

This algorithm relies heavily on the float representation of numbers, so we will go over those and it will be much clearer.

32-Long

00000000 00000000 00000000 00000000 00000000 = 0
00000000 00000000 00000000 00000000 00000001 = 1
00000000 00000000 00000000 00000000 00000010 = 2
00000000 00000000 00000000 00000000 00000011 = 3

Before we dive in

This algorithm relies heavily on the float representation of numbers, so we will go over those and it will be much clearer.

32-bit Long

$$\underbrace{0}_{\textit{sign bit}} \quad 0000000 \;\; 00000000 \;\; 00000000 \;\; 00000000 \;\; 00000011 = 3$$

Before we dive in

This algorithm relies heavily on the bit representation of numbers, so we will go over those and it will be much clearer.

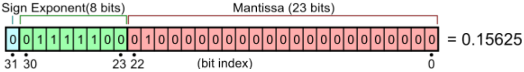
32-bit Long

$$\underbrace{1}_{sign\ bit}\ 0000000\ 00000000\ 00000000\ 00000000\ 00000011 = -3$$

-2147483647$\leq x \leq$ 2147483647

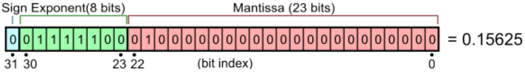
Before we dive in

The IEEE 754 Standard for floating-point arithmetic:



Before we dive in

The IEEE 754 Standard for floating-point arithmetic:



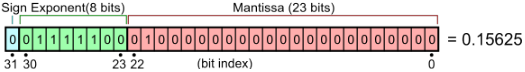
Considering the Mantissa as a fraction, each bit from left to right adds $(\frac{1}{2})^{(23-b_n)}$ where b_n is just the index of the bit.

So $\frac{1}{2} + \frac{1}{4} + \frac{1}{8}$... and so on.

Since this is binary, a bit is saved by assuming the first number before the decimal is one.

Before we dive in

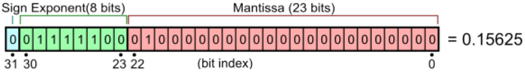
The IEEE 754 Standard for floating-point arithmetic:



Let M represent our Mantissa's actual integer value and note $\frac{M}{2^{23}} \in [0, 1)$, this way we can represent it as a fraction like we wanted.

Before we dive in

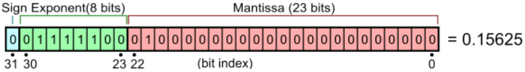
The IEEE 754 Standard for floating-point arithmetic:



Next, E will represent our Exponent before the bias, giving the range of values $0 \leq E \leq 255$

Before we dive in

The IEEE 754 Standard for floating-point arithmetic:



But we this algorithm only handles a special set called normalized numbers, and 11111111 is reserved so our actual range is

$$1 \leq E \leq 254$$

Before we dive in

The IEEE 754 Standard for floating-point arithmetic:

$$x_{bit} = (-1)^{sign}$$

Before we dive in

The IEEE 754 Standard for floating-point arithmetic:

$$x_{bit} = (-1)^{sign} \times 2^{E-127}$$

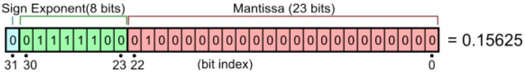
Before we dive in

The IEEE 754 Standard for floating-point arithmetic:

$$x_{bit} = (-1)^{sign} \times 2^{E-127} \times (1 + \frac{M}{2^{23}})$$

Before we dive in

The IEEE 754 Standard for floating-point arithmetic:



$$x_{dec} = (-1)^{sign} \times 2^{E-127} \times (1 + \frac{M}{2^{23}})$$

$$x_{bit} = 2^{31} \times sign + 2^{23} \times E + M$$

Before we dive in

The IEEE 754 Standard for floating-point arithmetic:

$$x_{dec} = 2^{E-127} \times (1 + \frac{M}{2^{23}})$$

$$x_{bit} = 2^{23} \times E + M$$

An interesting result hidden in logarithms

$$\log_2(x_{dec}) = \log_2(2^{E-127} \times (1 + \frac{M}{2^{23}}))$$

An interesting result hidden in logarithms

$$\log_2(x_{dec}) = \log_2(2^{E-127} \times (1 + \frac{M}{2^{23}}))$$

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

An interesting result hidden in logarithms

$$\log_2(x_{dec}) = \log_2(2^{E-127} \times (1 + \frac{M}{2^{23}}))$$

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

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

An interesting result hidden in logarithms

$$\log_2(x_{dec}) = \log_2(2^{E-127} \times (1 + \frac{M}{2^{23}}))$$

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

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

$\log_b(1 + x) \approx x + \sigma$ ¡- Click Me!

An interesting result hidden in logarithms

$$\log_2(x_{dec}) = \log_2(2^{E-127} \times (1 + \frac{M}{2^{23}}))$$

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

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

$$\log_b(1 + x) \approx x + \sigma \text{ ¡- Click Me!}$$

$$\log_2(x_{dec}) \approx E - 127 + \frac{M}{2^{23}} + \sigma$$

An interesting result hidden in logarithms

$$\log_2(x_{dec}) = \log_2(2^{E-127} \times (1 + \frac{M}{2^{23}}))$$

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

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

$\log_b(1 + x) \approx x + \sigma$ ¡- Click Me!

$$\log_2(x_{dec}) \approx E - 127 + \frac{M}{2^{23}} + \sigma$$

$$\log_2(x_{dec}) \approx E + \frac{M}{2^{23}} - 127 + \sigma$$

An interesting result hidden in logarithms

$$\log_2(x_{dec}) = \log_2(2^{E-127} \times (1 + \frac{M}{2^{23}}))$$

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

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

$\log_b(1 + x) \approx x + \sigma$ ¡- Click Me!

$$\log_2(x_{dec}) \approx E - 127 + \frac{M}{2^{23}} + \sigma$$

$$\log_2(x_{dec}) \approx E + \frac{M}{2^{23}} - 127 + \sigma$$

$$\log_2(x_{dec}) \approx \frac{1}{2^{23}}(2^{23} \times E + M) - 127 + \sigma$$

Notice

Our formula for float representation appears in the logarithm of our int value!

$$\log_2(x_{dec}) \approx \frac{1}{2^{23}}(\underbrace{2^{23} \times E + M}_{x_{bit}}) - 127 + \sigma$$

$$\implies x_{bit} \propto \log_2(x_{dec})$$

Notice

Our formula for float representation appears in the logarithm of our int value!

$$\log_2(x_{dec}) \approx \frac{1}{2^{23}}(\underbrace{2^{23} \times E + M}_{x_{bit}}) - 127 + \sigma$$

$$\implies x_{bit} \propto \log_2(x_{dec})$$

Why do we care that about $\log_2(x_{dec})$?

Notice

Our formula for float representation appears in the logarithm of our int value!

$$\log_2(x_{dec}) \approx \frac{1}{2^{23}}(\underbrace{2^{23} \times E + M}_{x_{bit}}) - 127 + \sigma$$

$$\implies x_{bit} \propto \log_2(x_{dec})$$

Why do we care that about $\log_2(x_{dec})$?

$$\log_b(x^a) = a\log_b(x) \implies log_2(x^{-1/2}) = -\frac{1}{2}log_2(x)$$

Combining what we know

Now if we want to get an approximation for $y = \frac{1}{\sqrt{x}}$ we simply plug it into our formula and rearrange a lot.

Combining what we know

Now if we want to get an approximation for $y = \frac{1}{\sqrt{x}}$ we simply plug it into our formula and rearrange a lot.

$$\frac{1}{2^{23}}(\underbrace{2^{23} \times E_y + M_y}_{y_{dec}}) - 127 + \sigma \approx -\frac{1}{2}(\frac{1}{2^{23}}(\underbrace{2^{23} \times E_x + M_x}_{x_{bit}}) - 127 + \sigma)$$

Combining what we know

Now if we want to get an approximation for $y = \frac{1}{\sqrt{x}}$ we simply plug it into our formula and rearrange a lot.

$$\frac{1}{2^{23}}(\underbrace{2^{23} \times E_y + M_y}_{y_{dec}}) - 127 + \sigma \approx -\frac{1}{2}(\frac{1}{2^{23}}(\underbrace{2^{23} \times E_x + M_x}_{x_{bit}}) - 127 + \sigma)$$

$$E_y + \frac{M_y}{L} - 127 + \sigma \approx -\frac{1}{2}((E_x + \frac{M_x}{L}) - 127 + \sigma)$$

Combining what we know

Now if we want to get an approximation for $y = \frac{1}{\sqrt{x}}$ we simply plug it into our formula and rearrange a lot.

$$\frac{1}{2^{23}}(\underbrace{2^{23} \times E_y + M_y}_{y_{dec}}) - 127 + \sigma \approx -\frac{1}{2}(\frac{1}{2^{23}}(\underbrace{2^{23} \times E_x + M_x}_{x_{bit}}) - 127 + \sigma)$$

$$E_y + \frac{M_y}{L} - 127 + \sigma \approx -\frac{1}{2}((E_x + \frac{M_x}{L}) - 127 + \sigma)$$

$$E_y + \frac{M_y}{L} \approx -\frac{1}{2}((E_x + \frac{M_x}{L}) - 127 + \sigma) + 127 - \sigma$$

Combining what we know

Now if we want to get an approximation for $y = \frac{1}{\sqrt{x}}$ we simply plug it into our formula and rearrange a lot.

$$\frac{1}{2^{23}}(\underbrace{2^{23} \times E_y + M_y}_{y_{dec}}) - 127 + \sigma \approx -\frac{1}{2}(\frac{1}{2^{23}}(\underbrace{2^{23} \times E_x + M_x}_{x_{bit}}) - 127 + \sigma)$$

$$E_y + \frac{M_y}{L} - 127 + \sigma \approx -\frac{1}{2}((E_x + \frac{M_x}{L}) - 127 + \sigma)$$

$$E_y + \frac{M_y}{L} \approx -\frac{1}{2}((E_x + \frac{M_x}{L}) - 127 + \sigma) + 127 - \sigma$$

Combining what we know

Now if we want to get an approximation for $y = \frac{1}{\sqrt{x}}$ we simply plug it into our formula and rearrange a lot.

$$\frac{1}{2^{23}}(\underbrace{2^{23} \times E_y + M_y}_{y_{dec}}) - 127 + \sigma \approx -\frac{1}{2}(\frac{1}{2^{23}}(\underbrace{2^{23} \times E_x + M_x}_{x_{bit}}) - 127 + \sigma)$$

$$E_y + \frac{M_y}{L} - 127 + \sigma \approx -\frac{1}{2}((E_x + \frac{M_x}{L}) - 127 + \sigma)$$

$$E_y + \frac{M_y}{L} \approx -\frac{1}{2}((E_x + \frac{M_x}{L}) - 127 + \sigma) + 127 - \sigma$$

$$E_y + \frac{M_y}{L} \approx \frac{3}{2}(127 - \sigma) - \frac{1}{2}((E_x + \frac{M_x}{L}))$$

Combining what we know

Now if we want to get an approximation for $y = \frac{1}{\sqrt{x}}$ we simply plug it into our formula and rearrange a lot.

Finally,

$$L \times E_y + M_y \approx \tfrac{3}{2}L(127 - \sigma) - \tfrac{1}{2}((L \times E_x + M_x))$$

$$y_{bit} \approx \tfrac{3}{2}L(127 - \sigma) - \tfrac{1}{2}(x_{bit})$$

We found something cool.

Interestingly, this is can be generalized for some exponent p other than $-\frac{1}{2}$ as:

$$y_{bit} \approx (1 - p)L(127 - \sigma) + p(x_{bit})$$

How close can this be?

Here is a graph of this approximation using a sub-obtimal value of σ.

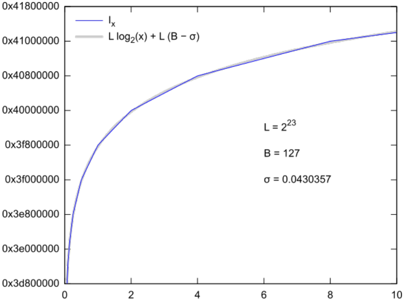


Figure: Scale in hexadecimal as we haven't yet converted back to Float

Back to the code

The 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;                       // Step 1 <--
    i  = 0x5f3759df - ( i >> 1 );               // Step 2
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) ); // Step 3
//  y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
}
```

Step 1 - Accessing the bits

i = * (long *) &y;

Fun fact

Not only was this undefined behaviour in C at the time, it still is! There are now different methods to do this.

The Problem

This would just convert our float into a long, losing any precision after the decimal. For example:

```c
#include <stdio.h>
int main() {
    long i;
    float y = 3.14159265;
    i = (long)y;
    printf("%lu", i);
    return 0;
}
```

This code snippet would return the number 3.

The Solution

Instead, we use pointers which point to memory address and convert that, think of it like putting the wrong tag on something at the store and it convincing the employee but here it is C we fool:

```c
#include <stdio.h>
int main() {
    long i;
    float y = 3.14159265;
    i = *(long*)&y;
    printf("%lu", i);
    return 0;
}
```

This code snippet would return the number 1078530011.

So What Is It?

The 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;                       // Step 1
    i  = 0x5f3759df - ( i >> 1 );               // Step 2 <--
    y  = * ( float * ) &i;                      // Reversing Step 1
    y  = y * ( threehalfs - ( x2 * y * y ) ); // Step 3
//  y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
}
```

Step 2 - Applying our Knowledge

y = 0x5F3759DF - (i >> 1);

Fun fact

0x just means "The following is a hexadecimal number" and it was changed to this when B, the predecessor to C was written.

C was originally called New B.

Step 2 - Applying our Knowledge

y = 0x5F3759DF - (i >> 1);

This Might look a bit more familiar,

$$y_{int} \approx \frac{3}{2}L(127 - \sigma) - \frac{1}{2}(x_{int})$$

Reminder

We mentioned earlier that a bit shift to the left divides by 2, so we subtract that to get our $-\frac{1}{2}(x_{int})$ term.

y = 0x5F3759DF - (i >> 1);

Reminder

We mentioned earlier that a bit shift to the left divides by 2, so we subtract that to get our $-\frac{1}{2}(x_{int})$ term.

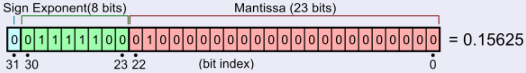
y = 0x5F3759DF - (i >> 1);

$$y_{int} \approx \underbrace{\frac{3}{2}L(127-\sigma)}_{?} \underbrace{-\frac{1}{2}(x_{int})}_{-(i>>1)}$$

$$C = \frac{3}{2}L(B-\sigma)$$

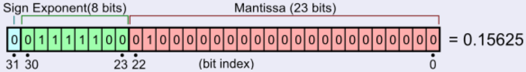
The challenge of finding C

You may have noticed, when we shift our float 1 to the right, for odd exponents we push a 1 into the Mantissa.



The challenge of finding C

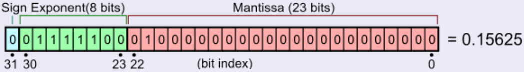
You may have noticed, when we shift our float 1 to the right, for odd exponents we push a 1 into the Mantissa.



Lucky for us, we know this only adds up to 0.5 in the Mantissa but for our exponent we end up with:

The challenge of finding C

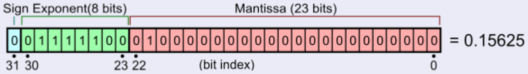
You may have noticed, when we shift our float 1 to the right, for odd exponents we push a 1 into the Mantissa.



Lucky for us, we know this only adds up to 0.5 in the Mantissa but for our exponent we end up with:

$$2^{((E>>1)-127)}$$

The challenge of finding C



We could also lose a 1 from our Mantissa during a shift.

This leaves us with 3 scenarios for our both our exponent and mantissa:

Recall $x_{dec} = 2^{E_x - B}(1 + \frac{M_x}{2^{23}})$

$let\ e_x = E_x - B,\ e_x \in [-126, 127]$

$let\ m_x = 1 + \frac{M_x}{2^{23}},\ m_x \in [1, 2)$

For some $n \in [-63, 63], k \in \mathbb{Z}^+$ we get the cases,

$$e_x = 2n,\ m_x = 0$$
$$e_x = 2n + 1,\ m_x = 2k + 1$$
$$e_x = 2n + 1,\ m_x = 2(k + 1)$$

*There are many ways to explain this problem, this portion is a simplified version of the explanation in [7. Moroz et al]

The challenge of finding C

Now for each case lets look at the what happens to the components of our shifted value y:

$$e_y = \begin{cases} -n, & e_x = 2n \\ -n-1, & e_x = 2n \\ -n-1, & e_x = 2n+1 \end{cases} \qquad m_y = \begin{cases} 0 & , m_x = 0 \\ \frac{2}{\sqrt{1+m_x}} - 1 & , m_x = 2k+1 \\ \frac{\sqrt{2}}{\sqrt{1+m_x}} - 1 & , m_x = 2(k+1) \end{cases}$$

Notes:

- $\frac{e_x}{2}$ is atleast twice overpowered by the bias
- For even e_x, the bit shifted into m_x means you've added 0.5
- For odd e_x, m_x becomes $\frac{1}{\sqrt{\frac{(1+m_x)}{2}}} - 1 = \frac{\sqrt{2}}{\sqrt{m_x}} - 1$

*This portion is a simplified version of the explanation in [7. Moroz et al]

Choosing C

Today we won't be searching for a new Magic Number. However:

- σ is actually tiny

Choosing C

Today we won't be searching for a new Magic Number. However:

- σ is actually tiny
- We can set $\sigma = 0$ and get close

Choosing C

Today we won't be searching for a new Magic Number. However:

- σ is actually tiny
- We can set $\sigma = 0$ and get close

With $\sigma = 0$,

Choosing C

Today we won't be searching for a new Magic Number. However:

- σ is actually tiny
- We can set $\sigma = 0$ and get close

With $\sigma = 0$,

$$\frac{3}{2}L(127 - \sigma) = \frac{3}{2}L(127 - 0)$$

Choosing C

Today we won't be searching for a new Magic Number. However:

- σ is actually tiny
- We can set $\sigma = 0$ and get close

With $\sigma = 0$,

$$\frac{3}{2}L(127 - \sigma) = \frac{3}{2}L(127 - 0)$$

$$= \frac{3}{2}(2^{23})(127) = 1598029824$$

Choosing C

Today we won't be searching for a new Magic Number. However:

- σ is actually tiny
- We can set $\sigma = 0$ and get close

With $\sigma = 0$,

$$\frac{3}{2}L(127 - \sigma) = \frac{3}{2}L(127 - 0)$$

$$= \frac{3}{2}(2^{23})(127) = 1598029824$$

$$= 5F400000 \text{ or } \mathrm{0x5F400000}$$

Choosing C

Today we won't be searching for a new Magic Number. However:

- σ is actually tiny
- We can set $\sigma = 0$ and get close

With $\sigma = 0$,

$$\frac{3}{2}L(127 - \sigma) = \frac{3}{2}L(127 - 0)$$

$$= \frac{3}{2}(2^{23})(127) = 1598029824$$

$$= 5F400000 \text{ or } 0x5F400000$$

Surprisingly close to the original magic number 0x5F3759DF

Our First Approximation

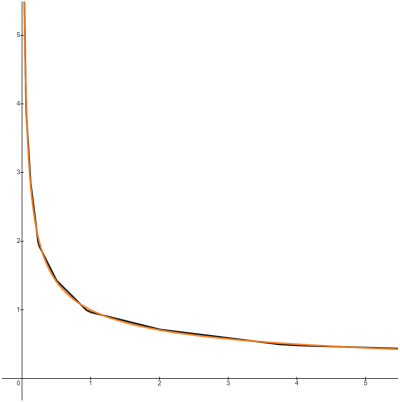


Figure: Our approximation using $\sigma = 0.0450465$ in black compared to $\frac{1}{\sqrt{x}}$ in green.

The 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;                    // Step 1
    i  = 0x5f3759df - ( i >> 1 );            // Step 2
    y  = * ( float * ) &i;                   // Reversing Step 1
    y  = y * ( threehalfs - ( x2 * y * y ) ); // Step 3 <--
//  y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
}
```

Step 3 - Newton's Method

y = y * (threehalfs - (x2 * y * y));

Step 3 - Newton's Method

```
y = y * ( threehalfs - ( x2 * y * y ) );
```

Newton's Method

Newton's method is a root finding algorithm that approximates the roots of a function, if the initial guess is good then after applying the formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

x_{n+1} will be a better guess.

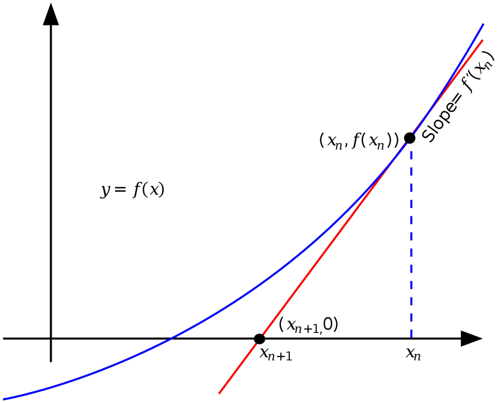


Figure: Newton's Method

Wait, Roots?

Zeros? I thought we were looking of inverse square roots? Well if we define a function $f(y)$:

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

The roots of this function would be when our guess y is exactly $\frac{1}{\sqrt{x}}$

Newton's method applied to y using $f(y) = \frac{1}{y^2} - x$:

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}$$

Wait, Roots?

Zeros? I thought we were looking of inverse square roots? Well if we define a function $f(y)$:

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

The roots of this function would be when our guess y is exactly $\frac{1}{\sqrt{x}}$

Newton's method applied to y using $f(y) = \frac{1}{y^2} - x$:

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}$$

$$y_{n+1} = y - \frac{\frac{1}{y^2}+x}{\frac{d}{dy}(\frac{1}{y^2}+x)}$$

Wait, Roots?

Zeros? I thought we were looking of inverse square roots? Well if we define a function $f(y)$:

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

The roots of this function would be when our guess y is exactly $\frac{1}{\sqrt{x}}$

Newton's method applied to y using $f(y) = \frac{1}{y^2} - x$:

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}$$

$$y_{n+1} = y - \frac{\frac{1}{y^2}+x}{\frac{d}{dy}(\frac{1}{y^2}+x)}$$

$$y_{n+1} = y - \frac{\frac{1}{y^2}+x}{\frac{-2}{y^3}} = y + \frac{y^3(\frac{1}{y^2}+x)}{2}$$

Wait, Roots?

Zeros? I thought we were looking of inverse square roots? Well if we define a function $f(y)$:

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

The roots of this function would be when our guess y is exactly $\frac{1}{\sqrt{x}}$

Newton's method applied to y using $f(y) = \frac{1}{y^2} - x$:

$$y_{n+1} = y + \frac{y}{2} - \frac{y^3 x}{2}$$

Wait, Roots?

Zeros? I thought we were looking of inverse square roots? Well if we define a function $f(y)$:

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

The roots of this function would be when our guess y is exactly $\frac{1}{\sqrt{x}}$

Newton's method applied to y using $f(y) = \frac{1}{y^2} - x$:

$$y_{n+1} = y + \frac{y}{2} - \frac{y^3 x}{2}$$

$$y_{n+1} = y(1 + \frac{1}{2} - \frac{y^2 x}{2})$$

Wait, Roots?

Zeros? I thought we were looking of inverse square roots? Well if we define a function $f(y)$:

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

The roots of this function would be when our guess y is exactly $\frac{1}{\sqrt{x}}$

Newton's method applied to y using $f(y) = \frac{1}{y^2} - x$:

$$y_{n+1} = y + \frac{y}{2} - \frac{y^3 x}{2}$$

$$y_{n+1} = y(1 + \frac{1}{2} - \frac{y^2 x}{2})$$

$$y_{n+1} = y(\frac{3}{2} - \frac{y^2 x}{2})$$

Wait, Roots?

Zeros? I thought we were looking of inverse square roots? Well if we define a function $f(y)$:

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

The roots of this function would be when our guess y is exactly $\frac{1}{\sqrt{x}}$

Newton's method applied to y using $f(y) = \frac{1}{y^2} - x$:

$$y_{n+1} = y(\frac{3}{2} - \frac{y^2 x}{2})$$

```
y = y * ( threehalfs - ( x2 * y * y ) );
```

The New Approximation

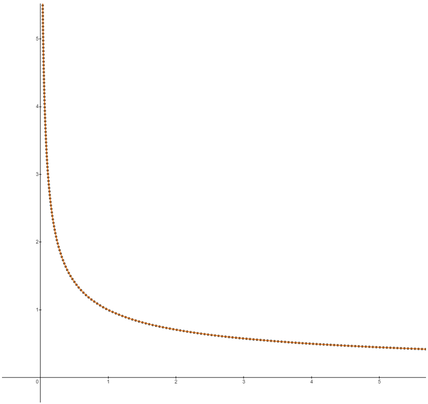


Figure: Our new values in black compared to $\frac{1}{\sqrt{x}}$ in green.

The 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;                    // Step 1
    i  = 0x5f3759df - ( i >> 1 );            // Step 2
    y  = * ( float * ) &i;                   // Reversing Step 1
    y  = y * ( threehalfs - ( x2 * y * y ) ); // Step 3
//  y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
}
```

Outline

Modern Day

Where is it today?

As of 2020, there is still papers being written trying to find more efficient magic constants but outside of theory the algorithm is now mostly obsolete.

Modern Day

Where is it today?

As of 2020, there is still papers being written trying to find more efficient magic constants but outside of theory the algorithm is now mostly obsolete.

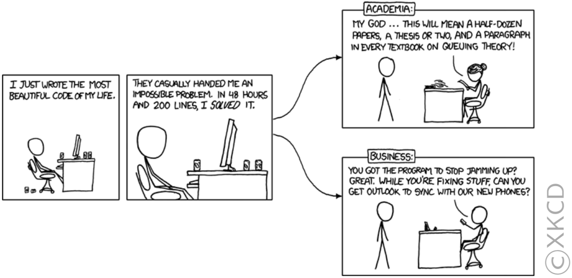
Why?

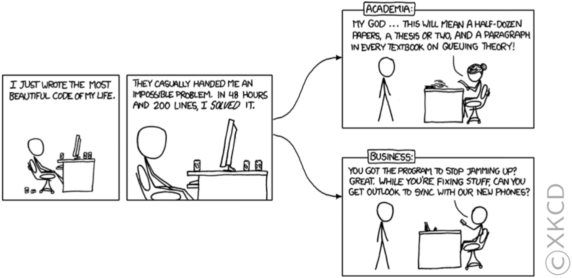
Around 1999, new hardware came out supporting rsqrtss, an instruction for computing inverse square roots.

Modern Day

So why explore it?

The algorithm may be obsolete now, but it serves as a reminder not to be so hasty in assuming that all the low-hanging fruit has been picked in our fields.





Tile text: Some engineer out there has solved P=NP and it's locked up in an electric eggbeater calibration routine. For every 0x5f375a86 we learn about, there are thousands we never see.

More on the Topic

- Cool Raytracing Demo - `https://www.youtube.com/watch?v=V2YsxqI-x64`
- Interactive IEEE-754 Tool - `https://www.h-schmidt.net/FloatConverter/IEEE754.html`
- Desmos Graphs of the accuracy at each step - `https://www.desmos.com/calculator/yoz6n1wlvu`
- Github of the Quake III source code - `https://github.com/id-Software/Quake-III-Arena`

Outline

References

1. "Fast Inverse Square Root — A Quake III Algorithm" -
   https://www.youtube.com/watch?v=p8u_k2LIZyo

2. "The Fast Inverse Square Root – An intuitive explanation" -
   https://www.youtube.com/watch?v=NCuf2tjUsAY

3. Hansen.Hummus and Magnets."0x5f3759df"-
   http://h14s.p5r.org/2012/09/0x5f3759df.html?mwh=1

4. Munafo."Notable Properties of Specific Numbers"-
   https://mrob.com/pub/math/numbers-16.html#le009$_$16

5. Eberly."Fast Inverse Square Root" -
   http://web.archive.org/web/20030426190503/http://www.magic-software.com/Documentation/FastInverseSqrt.pdf

References

6. Lomont. "FAST INVERSE SQUARE ROOT" - `http://www.lomont.org/papers/2003/InvSqrt.pdf`
7. Moroz et al. "Fast calculation of inverse square root with the use of magic constant analytical approach" - `https://arxiv.org/abs/1603.04483`
8. Rhys. "Origin of Quake3's Fast InvSqrt() - Part 1"- `https://www.beyond3d.com/content/articles/8/`
9. Rhys. "Origin of Quake3's Fast InvSqrt() - Part Two - Page 1"- `https://www.beyond3d.com/content/articles/15/`
10. Wikipedia. "Fast Inverse Square Root" - `https://en.wikipedia.org/wiki/Fast_inverse_square_root`