A Brief History of InvSqrt

by

Matthew Robertson

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Science in Computer Science, Honours

In the Department of Computer Science and Applied Statistics

Supervisor:	Owen Kaser, PhD, Computer Science
Examining Board:	Hazel Webb, PhD, Computer Science
	Trevor Jones, PhD, Mathematics

THE UNIVERSITY OF NEW BRUNSWICK

April 24, 2012

Dedication

For Mum.

Abstract

This thesis is about the fast *InvSqrt* function found in the public domain, or more notably, the C source code for *Quake III: Arena*. The function uses some clever bit hacking to approximate the inverse (or reciprocal) square root of a 32-bit floating point number quickly. The original author of the function is still unknown; and the exact mechanics of how the function was derived is still unknown. In fact, the exact mechanics of how it even works was still not completely known, until now. This thesis will take a quick look at the history of the function, interpret the function, analyze the benefits of the function, and mathematically optimize the function even further.

Acknowledgements

I would like to thank Dr. Owen Kaser, who offered me everything I needed for my Honours degree.

Table of Contents

List of Tables

List of Figures

List of Symbols, Nomenclature or Abbreviations

$\phi(w)$ represents the *float* represented by the *word* w

$\iota(w)$ represents the *integer* represented by the bits of the *float* w

$f = \boxed{E}\boxed{M}$ represents the exponent and mantissa fields of a some float

$R = \boxed{S}\boxed{T}$ represents the *magic* constant, exponent and mantissa fields

G represents the initial guess given by the *magic* line

E_0 represents the least significant bit of E

t represents the mantissa field of R scaled to $t \in [0, 1)$

t_0 represents the optimal mantissa field of R

$y(x)$ represents the mantissa field of the approximation with $x \in [0, 1)$

$w(x)$ represents the relative error of $y(x)$

$p(x)$ represents $y(x)$ transformed and after one Newton iteration

$v(x)$ represents the relative error of $p(x)$

z_{E_0} represents the mantissa field of $1/\sqrt{f}$ with $E_0 \in \{0, 1\}$

Chapter 1

Introduction

Some unknown master bit hacker has released an amazing function into the public domain. This function, commonly called *InvSqrt*, approximates the inverse (or reciprocal) square root of a 32-bit floating point number very quickly. It can be found in many open source libraries and games on the Internet, such as the C source code for *Quake III: Arena*. This raises many questions. Why is it needed? Who wrote it? How does it work? How *well* does it work? Is it still useful with modern processors today? And finally, can it be improved to work better? This thesis will examine those questions and give a unique interpretation and optimization of the function itself.

1.1 Motivation

Games like *Quake III: Arena* were pretty impressive for their time. Their graphics required a lot of real time calculations in an age of slower processors than today. A common calculation was to *normalize* a vector, or calculate its *unit vector* $\hat{v}$ by

$$\hat{v} = \frac{\vec{v}}{||\vec{v}||}$$

where $||\vec{v}||$ is the *Euclidean norm* of $\vec{v}$

$$||\vec{v}|| = \sqrt{(v_i)^2 + (v_j)^2 + (v_k)^2}.$$

So clearly there was a need to calculate the reciprocal square root of floating point numbers quickly.

1.2 History

The fast InvSqrt function received a lot of attention on Slashdot in August of 2005 following the release of the *Quake III: Arena* source code. Calling it "one of the more famous snippets of graphics code in recent years," [8] they question how it works and who the author was. Originally, they credited it, with good reason, to John Carmack, the head developer of *Quake*; however he quickly denied it in an e-mail [8]. One can look through the article on Slashdot and quickly see how confusing the function was, and how baffled it left many people in the community.

Further digging revealed the function, in its current form, first appeared on the newsgroup comp.graphics.algorithms in January 2002. However *Quake III: Arena* had already been released, without source, in 1999. Pieces of the source code may have been leaked before it was officially released under the GNU Public License.

An interesting paper from 1997 called Floating-Point Tricks [1] contains a more general, less refined method to approximate $\log_2(x)$ and x^n. in general. The author, James F. Blinn, proposed several tricks, or hacks, to do with floating point numbers. It is possible his version of the function was the original, and had been refined and specialized specifically for the reciprocal square root by other people.

Other papers written about this subject also have had no success identifying the author of the function [2] [4]. It is possible there is no single author, and that explains why he has not revealed himself yet. The function may have just evolved into its present form by passing though several different programmers' keyboards.

Copies of this function floating around in the public domain often incorrectly call it *InvSqrt* meaning *Inverse Square Root*. The inverse square root function would just be the squared function. The function this thesis considers is, more precisely, a reciprocal square root function.

1.3 The Function

The C function in Figure 1.1, as given in the *Quake III: Arena* source code,[1] uses bit hacking to estimate the reciprocal square root of a given *float*. Then, it proceeds to use the *Newton-Raphson* method to better the estimation [3].

```
float Q_rsqrt(float number) {
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = *(long *) &y;
    i = 0x5f3759df - (i >> 1);
    y = *(float *) &i;
    y = y * (threehalfs - (x2 * y * y));
//  y = y * (threehalfs - (x2 * y * y));

    return y;
}
```

Figure 1.1: **Q_rsqrt()** as found in *Quake III: Arena*.

Researchers should probably be scratching their heads right about now. There is no obvious reason one would think this code could approximate the reciprocal square root of a number so well that the Newton step only has to be done once. Notice the Newton iteration is commented out the second time, which shows how well it works!

[1] C preprocessor directives and comments removed.

1.4 Accuracy

The first thing one might wonder is, "How good is this approximation?" The graph shown in Figure 1.2 shows the function $y = 1/\sqrt{x}$ and points taken from the **Q_rsqrt()** function at random points across all *floats*.

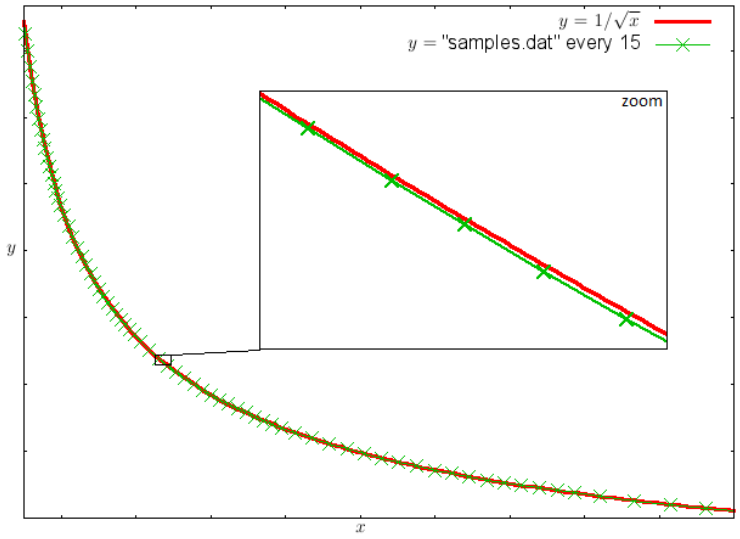

Figure 1.2: Graph of y $= 1/\sqrt{x}$ against sample data.

Clearly, the function is very accurate. In fact, it has a maximum relative error of 0.0017522874, that is less than 0.2%, given by[2]

$$\mathbf{max}\left(\sqrt{f}\cdot\mathbf{Q\_rsqrt}(f)-1\right)$$

for every positive *normalized float* f.

[2]See Appendix A.1 and A.2 for C code.

The accuracy of this function drops off for very small *denormalized floats*. Such numbers are very close to 0, so the reciprocal square root of denormalized floats approaches ∞. A very small number can be considered 0 and a very large number can be considered ∞, so the accuracy is less important in this case.

Chapter 2

Background

The fast reciprocal square root function relies on a couple big concepts such as the *Newton-Raphson* method and *IEEE 754 floating point* representation.

2.1 Newton-Raphson Method

The *Newton-Raphson* method is a method for approximating solutions of $f(x) = 0$. It takes a current approximation, or *guess*, x_n and returns some (hopefully) better approximation x_{n+1} as follows

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{2.1}$$

where $f'(x)$ is the derivative of $f(x)$ with respect to x [7].

2.2 IEEE 754 Floating Point

A 32-bit floating point number is represented in memory as

$$w = \begin{array}{|c|c|c|} \hline s & E & M \\ \hline bit\,31 & 30 \leftarrow bits \rightarrow 23 & 22 \leftarrow bits \rightarrow 0 \\ \hline \end{array}$$

where w is a *word* (or *integer*) in memory [7]. These fields can be interpreted as integers such that s is the sign bit, 1 means negative, E is the exponent field, bias $b = 127$ and finally, M is the fractional part of the normalized mantissa. So the *float* $\phi(w)$ is given by

$$\phi(w) = (-1)^s \left(1 + \frac{M}{2^{23}}\right) 2^{E-b}. \tag{2.2}$$

For convenience, this is often expressed as

$$\phi(w) = (-1)^s \, (1 + m) \, 2^{E-b} \quad m \in [0, 1) \tag{2.3}$$

where $m = M/2^{23}$. This will easily translate to other floating point representations such as *double* or *quadruple* precision; and it will more closely resemble the mathematics later on in this thesis.

Chapter 3

Interpretation

Interpreting this function may be a little difficult because of the bit hacking and lack of proper documentation. However it can be broken down into the following sections and interpreted. The casting between (**long**$*$) and (**float**$*$) is one detail. The formula $\mathbf{y} = \mathbf{y} * (\mathbf{threehalfs} - (\mathbf{x2} * \mathbf{y} * \mathbf{y}))$ is another detail. But these are simple compared to the line that should jump out; $\mathbf{i} = \mathbf{0x5f3759df} - (\mathbf{i} >> \mathbf{1})$.

3.1 Casting Pointers

The function contains two explicit castings between (**long**$*$) and (**float**$*$); for example, the line $\mathbf{i} = *(\mathbf{long}*)\&\mathbf{y}$. This line takes the address of **y**, which is of type *float*, and casts it to a pointer of type *long integer*, then stores the value at that address in **i**. This is basically interpreting the *float* bits as an

integer without converting it with any intelligence. The other casting works similarly but from *integer* back to *float*. This concept relies on how *floats* are stored in memory as explained in Section 2.2. So the *integer* interpretation of a *float's* bits, $\iota(w)$, can be expressed as

$$\iota(w) = w = 2^{31}s + 2^{23}E + M \tag{3.1}$$

where s is the sign bit, 1 means negative, E is the exponent field, bias 127 and finally, M is the fractional part of the normalized mantissa.
For example, let w_π be the closest approximation of π in single precision floating point. Then the word w_π is represented in memory as

$w_\pi =$

0	128	4788187
$bit\,31$	$30 \leftarrow bits \rightarrow 23$	$22 \leftarrow bits \rightarrow 0$

and the float $\phi(w_\pi)$ is given by

$$\phi(w_\pi) = (-1)^0 \left(1 + \frac{4788187}{2^{23}}\right) 2^{128-127} \doteq 3.14159$$

and the integer $\iota(w_\pi)$ is given by

$$\iota(w_\pi) = w_\pi = 2^{31} \cdot 0 + 2^{23} \cdot 128 + 4788187 = 1078530011.$$

3.2 Bit Shift a Float

Normally, a bit shift is an illegal operation on a *float*. But though the magic of casting pointers, this can be done indirectly. The line of code $\mathbf{i} = \mathbf{0x5f3759df} - (\mathbf{i} >> \mathbf{1})$ contains a bit shift on **i**, the *integer* interpretation of the bits from a *float* **f**. This effectively divides **i** by 2 and cuts off the trailing bit, or *floors* it. Imagine this as dividing each field in **f** by 2. But notice if E is odd, then its least significant bit will fall into M as the most significant bit! (Well, if E is even this will happen too, but it will have no effect.) This indirect bit shifting of a *float* can be viewed as

Field:	s	E	M
i	s_0	$E_7E_6...E_1E_0$	$M_{22}M_{21}...M_1M_0$
$(i >> 1)$	0	$s_0E_7...E_2E_1$	$E_0M_{22}...M_2M_1$
Range:	$bit\,31$	$30 \leftarrow bits \rightarrow 23$	$22 \leftarrow bits \rightarrow 0$

where s_0 is 0. The M_0 bit will be lost, but is the least significant bit in the entire thing, so it will not be missed. However, the E_0 bit will not be lost, instead it will fall into the mantissa field. If the bit E_0 is a 1, then it will add 2^{22} to the mantissa field and a theoretical $2^{1/2}$ will be lost. Therefore the *word* $j = (i >> 1)$ can be expressed in exponent mantissa form as

$$j = \boxed{\lfloor E/2 \rfloor} \boxed{2^{22}E_0 + \lfloor M/2 \rfloor} \tag{3.2}$$

3.3 Iteration Step

The function contains the formula $\mathbf{y} = \mathbf{y} * (\mathbf{threehalfs} - (\mathbf{x2} * \mathbf{y} * \mathbf{y}))$. This comes from the *Newton-Raphson* method on

$$f(x) = \frac{1}{x^2} - h \tag{3.3}$$

which, from Equation 2.1, yields an x_{n+1} of

$$x_{n+1} = \frac{x_n}{2}\left(3 - hx_n^2\right). \tag{3.4}$$

This can be rearranged to more closely resemble the code as

$$x_{n+1} = x_n\left(\frac{3}{2} - \frac{h}{2}x_n^2\right) \tag{3.5}$$

where x_n is $\mathbf{y}$ and $h/2$ is $\mathbf{x2}$. This step refines the answer, and can be repeated. Notice in the *Quake III: Arena* source code it is commented out the second time.

3.4 The Magic Constant

Well, the constant is not really magic! The line of code previously considered also contains the *constant* R. Let $R = \mathbf{0x5f3759df}$, which breaks down into exponent and mantissa form as

$$R = \boxed{S \mid T} = \boxed{190 \mid 3627487}.$$

This particular value for R is what leads to the good initial guess when j is subtracted from it as an *integer*. Let $G = (R - j)$ be represented as

$$G = \boxed{190 \mid 3627487} - \boxed{\lfloor E/2 \rfloor \mid 2^{22}E_0 + \lfloor M/2 \rfloor}. \tag{3.6}$$

The following interpretation of the magic constant loosely follows that of Lomont [4] and Eberly [2]. There has been a history of slightly flawed interpretations, so this interpretation is followed by a proof of correctness and then Chapter 5 will explain the previous interpretations. One major difference is this interpretation will show there are only three valid cases to consider, whereas the other papers consider a fourth impossible case.

3.4.1 The Reciprocal Square Root

For some *float* f, from Equation 2.3, $f = (1+m)2^{E-127}$. The reciprocal square root of f can be expressed as

$$\begin{aligned}
\frac{1}{\sqrt{f}} &= \frac{1}{\sqrt{1+m}}\frac{1}{\sqrt{2^{E-127}}} \\
&= \frac{1}{\sqrt{1+m}}2^{-E/2+63.5} \\
&= \frac{1}{\sqrt{1+m}}2^{-\lfloor E/2\rfloor - E_0/2+63+1/2} \\
&= \left(\frac{1}{\sqrt{2}}\right)^{E_0}\frac{\sqrt{2}}{\sqrt{1+m}}2^{-\lfloor E/2\rfloor+63} \\
&= \frac{\left(\sqrt{2}\right)^{1-E_0}}{\sqrt{1+m}}2^{-\lfloor E/2\rfloor+63}
\end{aligned}$$

This can be broken down into two factors as

$$\frac{1}{\sqrt{f}} = \underbrace{\frac{\left(\sqrt{2}\right)^{1-E_0}}{\sqrt{1+m}}}_{\text{mantissa}}\underbrace{2^{63-\lfloor E/2\rfloor}}_{\text{exponent}}. \tag{3.7}$$

The exponent and mantissa fields of G given in Equation 3.6 approximate the exponent and mantissa factors of $1/\sqrt{f}$ given in Equation 3.7. Notice the mantissa field compensates for the flooring of E in the exponent field, so the exponent field can be kept constant and the mantissa field manipulated accordingly.

3.4.2 The Exponent

Consider only the exponent factor of Equation 3.7 given by $2^e = 2^{63-\lfloor E/2 \rfloor}$. When this is interpreted as a *float* there will be a bias applied. Compensate to get $2^{e+127} = 2^{190-\lfloor E/2 \rfloor}$. Now G can be partly represented as

$$G = \boxed{190 - \lfloor E/2 \rfloor \mid ???} \tag{3.8}$$

where $\lfloor E/2 \rfloor$ can never be greater than 190, so there are no underflow issues.

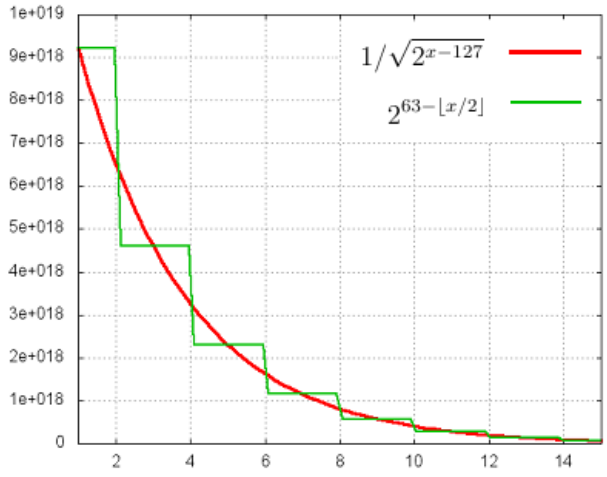

Figure 3.1: Exponent Factors

Notice the exponent factor corresponds perfectly to the exponents in Equation 3.6. The graph shown on Figure 3.1 shows the exponent part of the true reciprocal square root of f, as given in Equation 3.7, and the exponent part of the initial guess, over a small range of exponents. Clearly it fits quite nicely. So G is of the same order of magnitude as $1/\sqrt{f}$. The mantissa factor will compensate for the floored exponents to create a nearly perfect fit.

3.4.3 The Mantissa

Consider only the mantissa factor of Equation 3.7 given by

$$z_{E_0} = \frac{\left(\sqrt{2}\right)^{1-E_0}}{\sqrt{1+m}}. \tag{3.9}$$

This factor is much more complicated. Whether E is even or odd affects the mantissa, and also there is the possibility of underflow due to the subtraction $G = (R - j)$, which can cause the mantissa to borrow a bit from the exponent. So there are different cases to consider. When E is even means $E_0 = 0$.

$$z_0 = \frac{\sqrt{2}}{\sqrt{1+m}}. \tag{3.10}$$

This means no bit falls into the mantissa field of $j = (i >> 1)$. However, there is the possibility of underflow when $\lfloor M/2 \rfloor > T$, where $T = 3627487$ is the mantissa field of the magic constant.

M Small, E Even When M is small, such that $\lfloor M/2 \rfloor \leq T$, no underflow occurs so no bit is borrowed from the exponent field. Thus G is simply

$$G = \boxed{190 - \lfloor E/2 \rfloor \mid T - \lfloor M/2 \rfloor} \tag{3.11}$$

or as a *float* this is interpreted as

$$\phi(G) = \left(1 + \frac{T - \lfloor M/2 \rfloor}{2^{23}}\right) 2^{190 - \lfloor E/2 \rfloor - 127}. \tag{3.12}$$

Consider the mantissa field of $\phi(G)$, y_1, for graphing as

$$y_1(x) = 1 + t - \frac{x}{2} \quad , \; x \in [0, 2t). \tag{3.13}$$

where $x \in [0, 1)$ and $t = T/2^{23}$.

M Large, E Even When M is large, such that $\lfloor M/2 \rfloor > T$, underflow does occur, so a bit is borrowed from the exponent field of i, subtracting 1 from the exponent field and adding 2^{23} to the mantissa field. Thus G is given by

$$G = \boxed{190 - 1 - \lfloor E/2 \rfloor \mid 2^{23} + T - \lfloor M/2 \rfloor} \tag{3.14}$$

or as a *float*, this is interpreted as

$$\phi(G) = \left(1 + \frac{2^{23} + T - \lfloor M/2 \rfloor}{2^{23}}\right) 2^{189 - \lfloor E/2 \rfloor - 127}. \tag{3.15}$$

Borrowing a bit from the exponent field effectively cuts G in half. Because the exponent field of G is already fixed, divide the mantissa factor in half to compensate. So consider the line y_2 for graphing in Figure 3.2 as

$$y_2(x) = 1 + \frac{t}{2} - \frac{x}{4} \quad , \; x \in [2t, 1). \tag{3.16}$$

where $x \in [0, 1)$ and $t = T/2^{23}$.

E Odd Consider the case that E is odd means $E_0 = 1$.

$$z_1 = \frac{1}{\sqrt{1+m}}. \tag{3.17}$$

This means a 1 bit falls into the mantissa field of j. There is guaranteed underflow in the subtraction $G = (R - j)$ because the mantissa field of R is less than the smallest possible value for j, 2^{22}. So a bit is borrowed from the exponent field and G can be represented as

$$G = \boxed{190 - 1 - \lfloor E/2 \rfloor \mid 2^{23} + T - (2^{22} + \lfloor M/2 \rfloor)}. \tag{3.18}$$

Similar to y_2, a bit is borrowed from the exponent field effectively dividing G by 2. So consider y_3 for graphing in Figure 3.3 as

$$y_3(x) = \frac{3}{4} + \frac{t}{2} - \frac{x}{4} \quad , \; x \in [0, 1). \tag{3.19}$$

where $x \in [0, 1)$ and $t = T/2^{23}$.

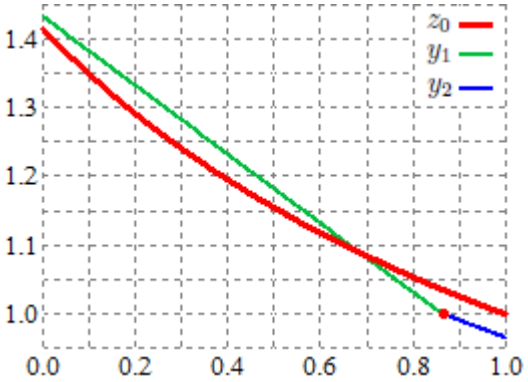

Figure 3.2: Graph of z_0, y_1, y_2.

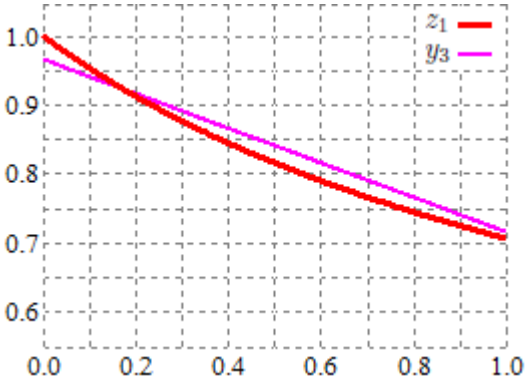

Figure 3.3: Graph of z_1, y_3.

The accuracy of the initial guess is a result of all of the previous cases together. The exponent field is shown in Figure 3.1 and the mantissa field is shown in Figures 3.2 and 3.3. Notice there are the same number of even exponents as odd exponents, so the two cases of E_0 are equally important. Consider, for Figure 3.4, the line $y_4 = \sqrt{2}y_3$, to scale to z_0. Now all cases can be graphed against z_0 and compared fairly.

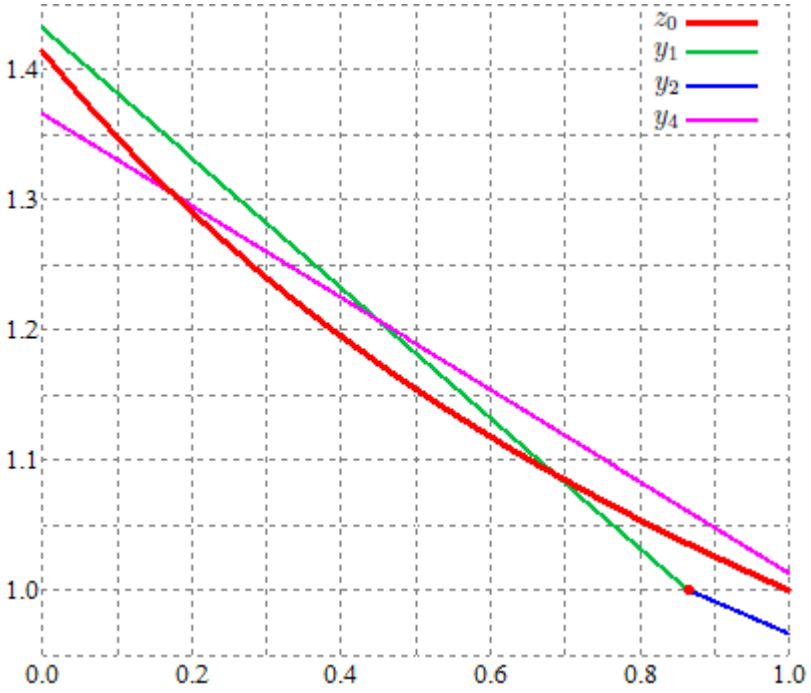

Figure 3.4: Graph of z_0, y_1, y_2, y_4

Notice at the point $x = 2t$, the line y_2 takes over from y_1 to be more accurate. When both the mantissa factor and exponent factor are accurate, overall the initial guess G will be accurate. Also notice the $y_n(x)$ functions and t are size neutral, they can easily be used to represent any precision floating point.

The pointer casting allows for the bit shifting on a *float*, then subtracting that from the *magic constant* allows for a very good initial guess. This initial guess is fed into the *Newton-Raphson method* to refine the initial guess to become a very good approximation of the reciprocal square root of a floating point number. Now the questions remain, "How was this magic constant derived?" and "Is there a better one?"

3.5 Proof of Correctness

Other papers have had mistakes in their interpretation of the line of code $\mathbf{i} = \mathbf{0x5f3759df} - (\mathbf{i} >> \mathbf{1})$ [2] [4]. So it is important to prove this interpretation is correct for all positive *floats*. Because there are only a finite number of positive *floats*, it is possible to test them all by asserting

$$\mathbf{G(i)} == \mathbf{0x5f3759df} - (\mathbf{i} >> \mathbf{1}) \tag{3.20}$$

for every *integer* (or *word*) $\mathbf{i}$, corresponding to every positive *float* f. For the C function $\mathbf{G}()$, representing precisely the cases previously given as G, this assertion does confirm[1].

[1] See Appendix A.3 for C code.

3.6 Benchmarks

In 2003, the function was considered to be about four times faster than the native *libm* **1.0/sqrt**() [4]. A test on a few different CPUs can demonstrate this. Table 3.1 shows the ratio of time to compute **1.0f/sqrtf**() over **Q_rsqrt**() across all normalized floats on different CPUs compiled with high optimization[2].

CPU	Minimum	Average	Maximum
Intel® Core™2 Duo E8400 3.00GHz	3.348315	3.352887	3.358521
Intel® Core™ i5-2415M 2.30GHz	3.960945	4.033596	4.079792
Intel® Core™2 Duo T9300 2.50GHz	3.102804	3.126771	3.153846
AMD V120 Processor	3.011129	3.014891	3.017516
AMD Sempron™ Processor 3200+	3.085374	3.096781	3.100399
Intel® Core™ i5-430M 2.27GHz	4.087017	4.098945	4.126741
Intel® Core™ i7-2640M 2.80GHz	4.081911	4.099887	4.111876

Table 3.1: Time ratio comparisons across different CPUs

On modern desktop computers it appears to be between three and four times faster. This means the function is still practical today! Running the benchmark test with the second Newton iteration enabled confirms that, as expected, adding the second iteration decreases the time by about one half. So a more accurate, two iteration version is also practical; however the great speed boost by having only one iteration may make it more attractive, even though significantly less accurate.

[2]All results compiled using the 4.x series of gcc with -O3.

Chapter 4

Analysis

It is important to generalize Section 3.4 for any *magic* constant and work for any sized floating point representation. So consider the conditional equation

$$G = \begin{cases} \boxed{S - \lfloor E/2 \rfloor \mid T - \lfloor M/2 \rfloor} & E \text{ even } M \text{ small} \\ \boxed{S - 1 - \lfloor E/2 \rfloor \mid 2^U + T - \lfloor M/2 \rfloor} & E \text{ even } M \text{ large} \\ \boxed{S - 1 - \lfloor E/2 \rfloor \mid 2^{U-1} + T - \lfloor M/2 \rfloor} & E \text{ odd} \end{cases} \tag{4.1}$$

where E being even or odd is defined by E_0 being 0 or 1, M being small or large is defined by ($\lfloor M/2 \rfloor \leq T$) or ($\lfloor M/2 \rfloor > T$) and U is the size of the mantissa field.

4.1 Exponent Field

The exponent field of R is trivial to find. Similar to the method shown in Section 3.4.2, S can be expressed as $S = \lfloor b/2 \rfloor + b$ or

$$S = \lfloor 3b/2 \rfloor \tag{4.2}$$

where b is the bias.

4.2 Mantissa Field

The mantissa field is, again, more complicated. The formula for the mantissa fraction of G can be expressed as the conditional equation combining the functions $y_1(x)$, $y_2(x)$ and $y_4(x)$,

$$y(x) = \begin{cases} 1 + t - x/2 & E_0 = 0,\ x \leq 2t \\ 1 + t/2 - x/4 & E_0 = 0,\ x > 2t \\ \sqrt{2}(3/4 + t/2 - x/4) & E_0 = 1 \end{cases} \tag{4.3}$$

where $x \in [0, 1)$ is the mantissa fraction of the input value and t is the mantissa fraction of the new *magic* constant. Minimizing the error of $y(x)$ compared to $z_0(x)$, from Section 3.4.3, will yield a new *magic* constant.

4.3 First Minimization

A natural way to make a good *magic* constant is to minimize the maximum relative error of $y(x)$. Consider the relative errors given by

$$w(x) = \left| \frac{y(x)}{z_0(x)} - 1 \right|. \tag{4.4}$$

Notice $w(x)$ is piecewise differentiable on the previously defined domains. The maximum value must occur at one of the following critical or end points

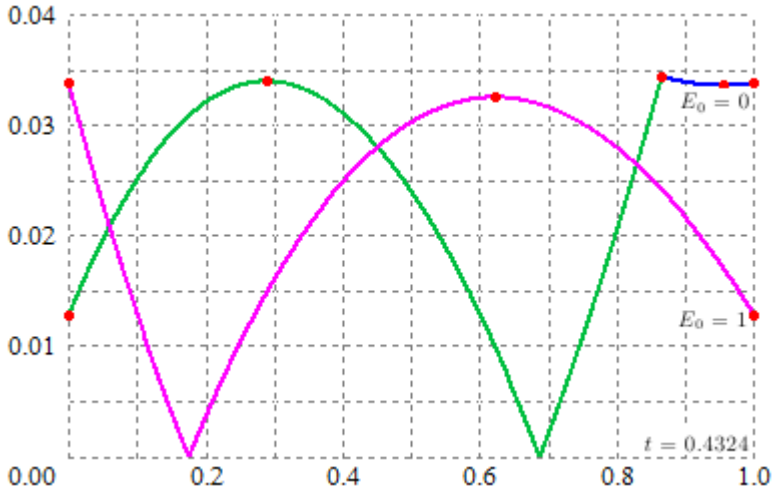

Figure 4.1: Graph of $w(x)$ with all critical and end points marked.

The critical points showing in Figure 4.1 can be expressed and labelled algebraically as

$$\underbrace{x = 0,}_{\text{end point}} \quad \underbrace{x = \frac{2t}{3},}_{\text{maximum}} \quad \underbrace{x = \frac{2t+1}{3},}_{\text{maximum}} \quad \underbrace{x = 2t,}_{\text{maximum}} \quad \underbrace{x = \frac{2(t+1)}{3},}_{\text{minimum}} \quad \underbrace{x = 1^{-}.}_{\text{end point}}$$

Taking $w(x)$ at these critical x values yields the maximum relative errors in terms of t as shown in Figure 4.1. This is a very delicate balancing act. To change t even slightly can cause one of the errors to become quite large very fast. The goal is to find the t value with the smallest maximum relative error value. Figure 4.2 shows the relative errors at the critical points resulting from different values of t.

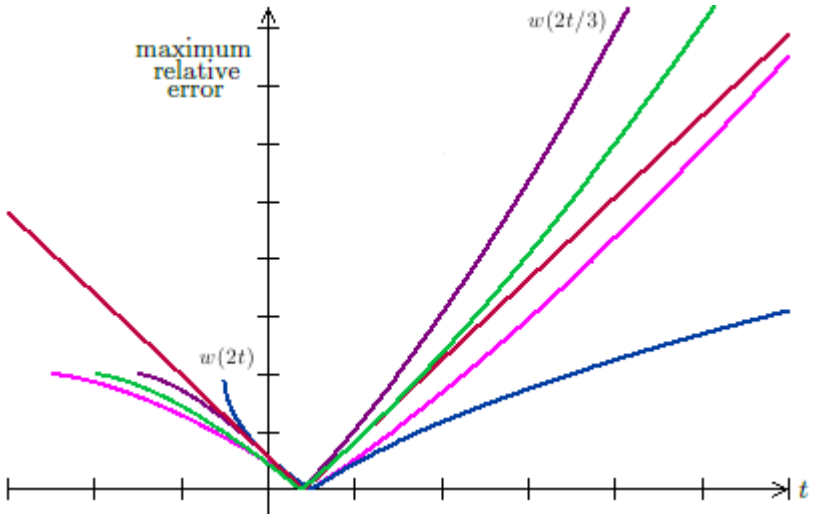

Figure 4.2: Graph of maximum relative errors against t.

Looking at Figure 3.4, it is clear that there is only a specific range t can belong to. That range is $t \in (\sqrt{2}-1, 1/2)$ because outside of this range, the even and odd exponents no longer balance each other. They both move away from the true value in the same direction. So only constants that have a mantissa field that corresponds to this range must be considered.

The optimal mantissa fraction t must be in the range $t \in (\sqrt{2}-1, 1/2)$. So consider the graph in Figure 4.3 zoomed into this range.

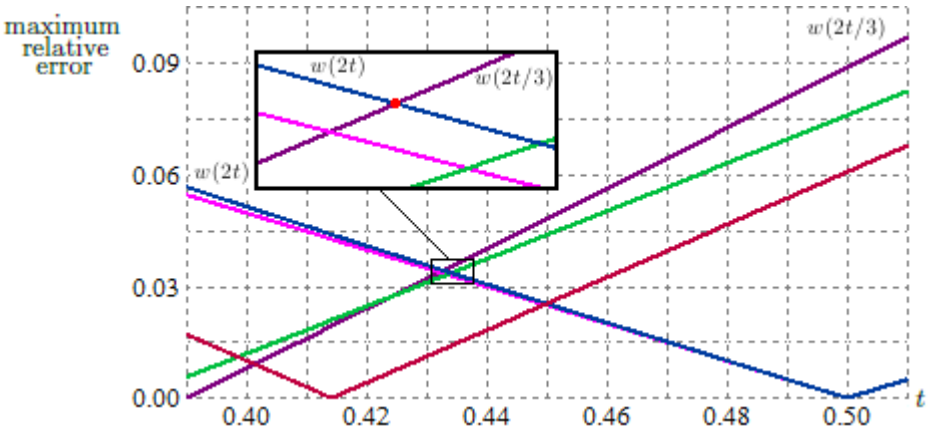

Figure 4.3: Maximum relative errors against t within the bounding lines.

It is clear that the optimal value of t is bounded only by the top two curves[1]. Therefore the t which gives the minimal maximum relative error is the intersection between the two curves $w(2t/3)$ and $w(2t)$ when E is even,

$$\left|\frac{\sqrt{6}(2t+3)^{3/2}}{18} - 1\right| = \left|\frac{\sqrt{2}\sqrt{2t+1}}{2} - 1\right|$$

where $t \in (\sqrt{2}-1, 1/2)$.

Which simplifies to be

$$4t^6 + 36t^5 + 81t^4 - 216t^3 - 972t^2 - 2916t + 1458 = 0$$

which yields

$$t \doteq 0.432744889959443195468521586996010373619 8$$

[1]See Appendix B.1 for Derive code.

This t differs from the t used in the original R, it also differs from the t calculated by Chris Lomont, in his paper Fast Inverse Square Root [4], where he uses a similar approach. The difference is only after the 26th digit, so its effect will not be seen until *quadruple precision floating point*. The difference is only significant because of how the t was obtained. Lomont used a numerical method to compute it, whereas this thesis used an algebraic and then numerical method to calculate it.

4.4 Testing

The R value corresponding to this calculated t value can be obtained by

$$R = \lfloor (190 + t)2^{23} \rfloor \tag{4.5}$$

which yields an R value of **0x5f37642f** for *floats*. However, as also noted by Lomont [4], this constant yields a maximum relative error of 0.0017758484, which is actually worse then the original **Q_rsqrt**() function. The original has a maximum relative error of 0.0017522874. However, the new constant will yield a better result before the *Newton-Raphson* iteration at 0.0342128389 when the original is 0.0343757719. So the natural question asked by Lomont was, "Why [is] the best initial approximation to the answer not the best after one Newton iteration?"

4.5 Newton Issue

All of the other explanations mentioned have made the same mistake; they find a constant that will give the closest initial guess to the real answer, but they don't care if they are smaller or larger than the answer. Because of the shape of the graph of $y = 1/\sqrt{x}$, after an iteration of the *Newton-Raphson* method from any good guess x_n, it will result in an x_{n+1} that is less than the answer. So an initial guess that is larger than the answer will not converge as fast as an initial guess that is smaller than the answer. So optimizing the magic constant before the iteration will not optimize the constant after. This can be demonstrated mathematically by considering the initial guess $x_n = (1/\sqrt{h} - \epsilon)$. So from Equation 3.4

$$x_{n+1} = \frac{(1/\sqrt{h} - \epsilon)}{2}\left(3 - h(1/\sqrt{h} - \epsilon)^2\right)$$

which can be rearranged to a more useful form as

$$x_{n+1} = \frac{1}{\sqrt{h}} - \underbrace{\left[\frac{\epsilon^2\sqrt{h}}{2}(3 - \epsilon\sqrt{h})\right]}_{\text{distance from root}}. \tag{4.6}$$

So clearly a small positive ϵ will have a smaller distance from the root, thus it is better to take an initial guess that is a little bit too small instead of a little bit too big. This also means the results of the **Q_rsqrt**() function will always give an estimate that is slightly too small.

4.6 Solution

The solution is to apply the optimization after the Newton iteration. The problem is it is hard to predict exactly how the Newton iteration will affect the float. Consider $y(x)$ from Equation 4.3 which approximates the mantissa field of $\sqrt{2}/\sqrt{1+x}$. Simply apply a transformation of $q(x) = y(x-1)/\sqrt{2}$ to approximate the mantissa field of $1/\sqrt{x}$. So $q(x)$ is given by

$$q(x) = \begin{cases} \sqrt{2}(3/4 + t/2 - x/4) & E_0 = 0,\ x \leq 1 + 2t \\ \sqrt{2}(5/8 + t/4 - x/8) & E_0 = 0,\ x > 1 + 2t \\ 1 + t/2 - x/4 & E_0 = 1 \end{cases} \tag{4.7}$$

where $x \in [1, 2)$. Because the approximation works for all valid exponents, it works for the exponents $E = 126$ and $E = 127$ specifically, that makes the exponent field in Equation 3.7 equal 1. Therefore $q(x)$ can be considered an approximation for $1/\sqrt{x}$ without worrying about the mantissa or exponent fields of the float. Now the iteration step described in Section 3.3 can be applied where $x_n = q(x)$ to give[2]

$$p(x) = q(x)\left(\frac{3}{2} - \frac{x}{2}q^2(x)\right) \tag{4.8}$$

where $x \in [1, 2)$.

[2]See Appendix B.2 for Derive code.

Clearly, from Figure 4.4, $p(x)$ is a very good approximation of $1/\sqrt{x}$.

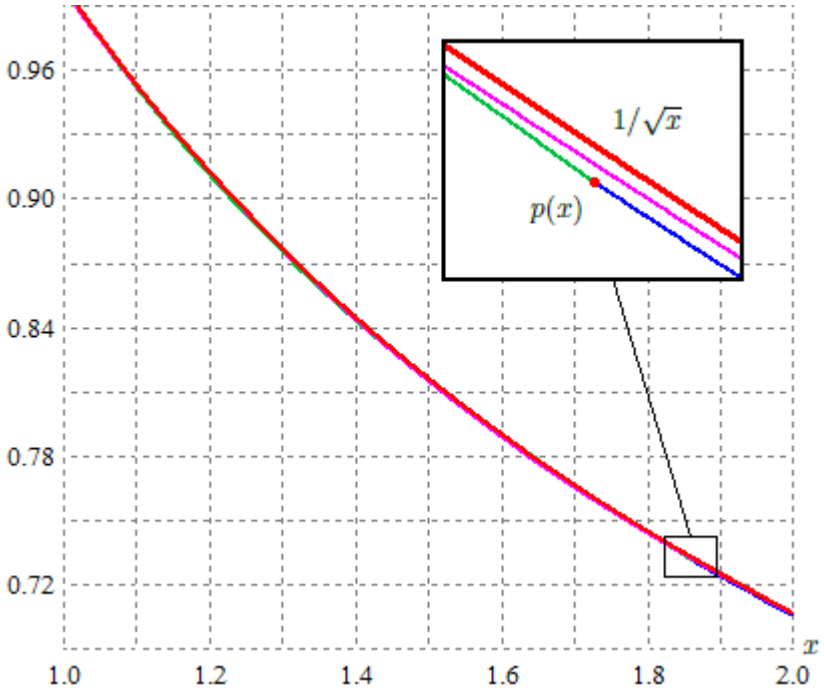

Figure 4.4: Graph $1/\sqrt{x}$ and $p(x)$

By a similar process to Section 4.3, the maximum relative error, $v(x)$, can be minimized where $v(x)$ is given by

$$v(x) = \left| \frac{p(x)}{1/\sqrt{x}} - 1 \right|. \tag{4.9}$$

The maximum value of $v(x)$ must occur at one of the following critical points

$$\underbrace{x = 1}_{\text{end point}}, \underbrace{x = \frac{2t+3}{3}}_{\text{maximum}}, \underbrace{x = \frac{2t+4}{3}}_{\text{maximum}}, \underbrace{x = 2t+1}_{\text{maximum}}, \underbrace{x = \frac{2t+5}{3}}_{\text{minimum}}, \underbrace{x = 2^{-}}_{\text{end point}}$$

Taking $v(x)$ at these critical x values yields results in terms of t as shown in Figure 4.5. Similar to Section 4.3, the goal is to find the t value with the smallest maximum relative error value. Figure 4.6 shows the relative error at the critical points resulting from different values of t.

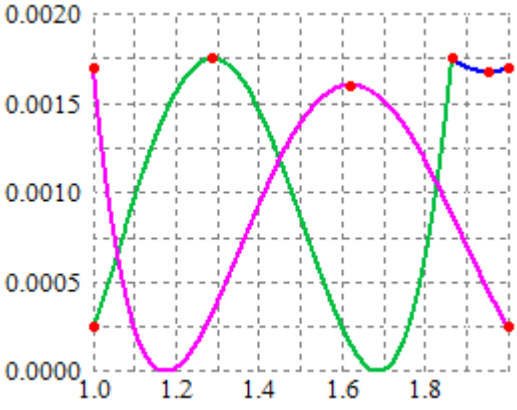

Figure 4.5: $v(x)$ with critical and end points marked.

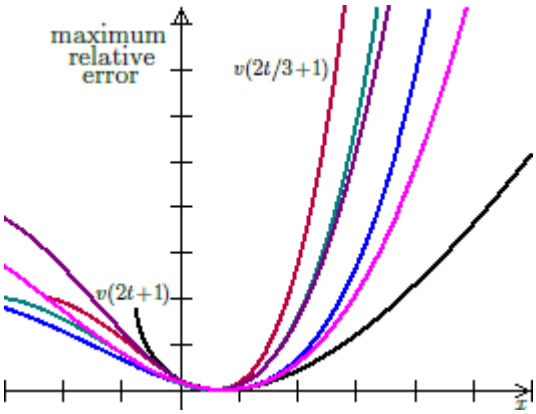

Figure 4.6: $v(x)$ taken at critical points across large range of t.

Figure 4.7 shows the maximum relative error zoomed into the optimal range.

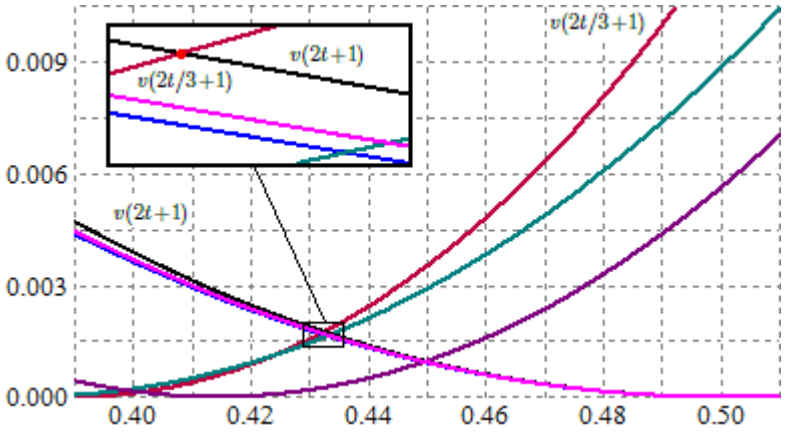

Figure 4.7: $v(x)$ taken at critical points across t bounded

The optimal t value is bounded by the top two curves[3] in Figure 4.7. Therefore the t which gives the minimal maximum relative error is the intersection between the two curves $v(2t/3+1)$ and $v(2t+1)$ when E is even,

$$\left| -\frac{\sqrt{6}(2t+3)^{3/2}(8t^3+36t^2+54t-135)}{1944} - 1 \right| = \left| \frac{\sqrt{2}(2t-5)\sqrt{2t+1}}{8} \right|$$

where $t \in (\sqrt{2}-1, 1/2)$.

Which simplifies to be

$$64t^6 + 576t^5 + 2592t^4 + 3888t^3 - 26244t + 10935 = 0$$

which yields an optimal t of about

$$t_0 \doteq 0.432450084790142642178782937496796466861 4$$

and a maximum relative error of about

$$0.001751183671220213352125174246700154536 8$$

An interesting point to mention here is that the accuracy of this method does not depend of the size of the floating point representation. So this method applied to a double or quadruple precision floating point would give the same maximum relative error to the precision of the floating point representation.

[3]See Appendix B.2 for Derive code.

4.7 Results

This optimal t value corresponds to an R value of **0x5f375a86**. This value of R has a measured maximum relative error of 0.0017512378 (and 0.0343654640 before the iteration step). The difference between this measured and the theoretical maximum relative error is due to the precision of the floating point representation. A higher precision floating point representation should have a measured maximum relative error closer to the theoretical value. This constant is better than the original for both the initial guess and after the iteration step!

4.8 Extension

On a final note, the magic constant can be extended into higher precision floating point such as *double* and *quadruple* precision using

$$R = \left\lfloor \left(\left\lfloor \frac{3b}{2} \right\rfloor + t_0 \right) 2^U \right\rfloor . \tag{4.10}$$

A *double* has a mantissa field of size $U = 52$ and a bias of $b = 1023$. Using Equation 4.10 this yields an optimal constant of $R =$ **0x5fe6eb50c7b537a9** with maximum relative error of 0.0017511837. Similarly, for *quadruple precision*, this yields a constant of $R =$ **0x5ffe6eb50c7b537a9cd9f02e504fcfbf** with a maximum relative error even closer to the theoretical. A similar method could be used to find optimal constants for two Newton iterations.

Chapter 5

Previous Explanations

As mentioned earlier, there have been previous explanations of the fast reciprocal square root function. The following authors have their own versions of the function, or their own explanations of how it works.

5.1 James Blinn

A early version of the fast reciprocal square root function is given by James F. Blinn in 1997 in his paper Floating-Point Tricks [1, 6]. His version of the function is derived from floating point approximations of $\log_2(x)$ and 2^n. These functions can be combined to give $x^{-1/2} = 2^{-1/2 \log_2(x)}$. Blinn has his own code to do this, but Figure 5.1 shows equivalent code in the same format as the *Quake III: Arena* version.

```
#define OneAsInteger 0x3F800000

float rsqrteq(float x) {
    uint32_t i;
    float y;

    y = x;
    i = *(uint32_t *) &y;
    i = (OneAsInteger + (OneAsInteger >> 1)) - (i >> 1);
    y = *(float *) &i;

    y = y * (1.47F - (0.47F * x * y * y));

    return y;
}
```

Figure 5.1: Reciprocal square root function equivalent to Blinn's version.

The first thing to note is that in place of a magic constant is the code, $(\mathbf{OneAsInteger} + (\mathbf{OneAsInteger} >> \mathbf{1}))$, which evaluates to **0x5f40000**. This is slightly larger than the optimized magic constant, but it is incredible how close it is. In fact, this magic constant is equivalent to a t value of 0.5; the point where the $y_2(x)$ case from Section 4.2 becomes impossible. The next thing to note is the modification on the Newton Iteration step; $\mathbf{y} * (\mathbf{1.47F} - (\mathbf{0.47F} * \mathbf{x} * \mathbf{y} * \mathbf{y}))$. This reduces the maximum relative error from about 1.2% using the normal Newton iteration, to about 0.6% using Blinn's mysterious modified version of the iteration step. This version of the fast reciprocal square root function is just a special case of a more generic x^n function that could be analysed and optimized in another paper.

5.2 David Eberly

A very tempting and very simple explanation is given by David Eberly in the 2002 version of his paper *Fast Inverse Square Root* [2] claiming the mantissa part of $(R >> 1)$ was simply a linear line approximating

$$y = \frac{1}{\sqrt{1+x}} \quad x \in [0, 1)$$

where S is some y-intercept and $m = -1/2$. Now this would be very nice if it was that simple, however there are two issues. Such a line is accurate close to $x = 0$ but not close to $x = 1$ because of the shape of the y function. Also this completely ignores the issues of the E_0 bit falling into the mantissa. The slope of y is given by

$$\frac{\mathrm{d}y}{\mathrm{d}x} = -\frac{1}{2}(1+x)^{-3/2}$$

so clearly at $x = 0$ the slope is $-1/2$ however as $x \to 1$ the slope $\to -\sqrt{2}/8$. To compensate for this, Eberly said take a slope of $-1/4$ by dividing the entire float in half, or subtract one from the exponent. Given the y-intercept 0.966215, this will yield a good initial guess. However this does not consider any the cases of a bit falling into the mantissa field or borrowing a bit from the exponent field. What Eberly did represents the case when E is odd, setting the foundation for future interpretations. However Eberly ignored the other two possible cases of E being even and M being small or large.

Eberly updated his paper in 2010 to include more cases after Chris Lomont provided a paper on the same subject.

5.3 Chris Lomont

In 2003, Chris Lomont, in his paper Fast Inverse Square Root [4], came to the same optimal *magic* constant for floats as this thesis, **0x5f375a86**. However, his constant was computed by brute force. He used mostly algebraic methods to come to the optimal pre-Newton iteration solution, but then resorted to brute force for the final optimal magic constant. He, as well as Eberly, considered an impossible case, when the exponent field is odd and no bit is borrowed during the subtraction from the magic constant. No such optimal constant exists that would cause that to happen. When the exponent field is odd, a bit falls into the most significant bit of the mantissa field, meaning the mantissa field is much larger than the largest possible mantissa field of an optimized magic constant.

Chapter 6

Conclusion

The magic constant can now be optimized mathematically, using the method described in this thesis, for any sized floating point representation. Figures 6.1 and 6.2 show optimized single and double precision versions of the function. The fast reciprocal square root function still receives a lot of attention from curious programmers, bit hackers, mathematicians and hobbyist; there is even a website named after the original magic constant [5]. May it inspire somebody in the future.

The fast reciprocal square root function still has room for improvement. More iterations of Newton's method will improve the accuracy dramatically; and the magic constant can be optimized again using a method similar to Section 4.6. On top of that, a very small offset constant can be added to the result to compensate for the fact that the Newton iteration(s) will always

give an estimate that is slightly too small. Now that there is a mathematical description of the function, even after Newton's iteration, the optimal offset constant could be calculated by considering the average distance distance between $1/\sqrt{x}$ and $p(x)$ given by $\int_1^2 (1/\sqrt{x} - p(x))\,\mathrm{d}x$. Also, the Newton iteration could be slightly modified to yield better results. There is potential for many other improvements as well. Such a highly accurate function could be useful for processors that have only basic arithmetic operations for double, quadruple, or any other precision floating point representation.

```
float rsqrt32(float number) {
    uint32_t i;
    float x2, y;

    x2 = number * 0.5F;
    y = number;
    i = *(uint32_t *) &y;
    i = 0x5f375a86 - (i >> 1);
    y = *(float *) &i;
    y = y * (1.5F - (x2 * y * y));

    return y;
}
```

Figure 6.1: Optimized reciprocal square root function in 32 bit.

```
double rsqrt64(double number) {
    uint64_t i;
    double x2, y;

    x2 = number * 0.5;
    y = number;
    i = *(uint64_t *) &y;
    i = 0x5fe6eb50c7b537a9 - (i >> 1);
    y = *(double *) &i;
    y = y * (1.5 - (x2 * y * y));

    return y;
}
```

Figure 6.2: Optimized reciprocal square root function in 64 bit.

Bibliography

[1] Jim Blinn, *Floating-point tricks*, IEEE Computer Graphics and Applications **17** (1997), no. 4.

[2] David Eberly, *Fast inverse square root*, Geometric Tools, LLC (2010), `http://geometrictools.com/Documentation/FastInverseSqrt.pdf`.

[3] id Software, *Quake III Arena*, `https://github.com/id-Software/Quake-III-Arena`.

[4] Chris Lomont, *Fast inverse square root*, Indiana: Purdue University (2003), `http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf`.

[5] Chris Miller, *0x5f3759df.org code faster code*, `http://0x5f3759df.org/`.

[6] Robert Munafo, personal communication, April 18 2011, `http://mrob.com/pub/math/numbers-16.html#1e009_16`.

[7] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical recipes in C: The art of scientific computing*, 2nd ed., Cambridge University Press, New York, NY, USA, 1992.

[8] Slashdot, *Quake 3: Arena source GPL'ed*, `http://games.slashdot.org/story/05/08/20/1329236/Quake-3-Arena-Source-GPLed`.

All web references last accessed April 24, 2012

Appendix A

C Code

A.1 samples.c

```
int i, j;
float f, g;
for (i = 0x00800000; i < 0x7f800000; i += 1<<15) {
    j = i | rand() & 0x7fff;
    f = *(float *) &j;
    g = Q_rsqrt(f);
    printf("%20.20f,%20.20f\n", f, g);
}
```

A.2 accuracy.c

```
int i;
float f;
double e, max = 0.0;
for (i = 0x00800000; i < 0x7f800000; i++) {
    f = *(float *) &i;
    e = fabs(sqrt(f) * Q_rsqrt(f) - 1);
    if (e > max)
        max = e;
}
printf("%f\n%e\n", max, max);
```

A.3 assert.c

```
#define S 190
#define T 3627487

int G(int w) {
    int E = (w >> 23) & 0xff; // extract E field
    int M = w & 0x7fffff; // extract M field

    int a, b; // the fields of the return float

    if ((E & 1) == 1) { // E Odd
        a = S - 1 - (E>>1);
        b = (1<<23) + T - (1<<22) - (M>>1);
    } else if ((M>>1) <= T) { // E Even M Small
        a = S - (E>>1);
        b = T - (M>>1);
    } else { // E Even M Large
        a = S - 1 - (E>>1);
        b = (1<<23) + T - (M>>1);
    }

    assert(a == (a & 0xff));
    assert(b == (b & 0x7fffff));

    // put new fields back in word
    return (a << 23) | b;
}

void checkDiscrepancies() {
    int i;
    // test all positive floats
    for (i = 0x00000001; i < 0x80000000; i++)
        assert(G(i) == 0x5f3759df - (i >> 1));
}
```

A.4 timer.c

```
double testQ_rsqrt() {
    time_t start = clock();
    int i;
    float f, g;
    for (i = 0x00800000; i < 0x7f800000; i++) {
        f = *(float *) &i;
        g = Q_rsqrt(f);
    }
    return (double) (clock() - start) / CLOCKS_PER_SEC;
}

double testSqrtf() {
    time_t start = clock();
    int i;
    float f, g;
    for (i = 0x00800000; i < 0x7f800000; i++) {
        f = *(float *) &i;
        g = 1.0f / sqrtf(f);
    }
    return (double) (clock() - start) / CLOCKS_PER_SEC;
}

void doTest() {
    int i;
    float a, b, d;
    for (i = 0; i < TESTS; i++) {
        a = testQ_rsqrt();
        printf("%f", a);
        b = testSqrtf();
        printf("\t%f", b);
        d = b / a;
        printf("\t%f\n", d);
    }
}
```

Appendix B

Derive 6 Files

B.1 firstmin.mth

```
PrecisionDigits:=100
NotationDigits:=100
z0(x):=SQRT(2)/SQRT(1+x)
z1(x):=1/SQRT(1+x)
y1(x):=1+t-x/2
y2(x):=1+t/2-x/4
y3(x):=3/4+t/2-x/4
y4(x):=SQRT(2)*y3(x)
w1(x):=y1(x)/z0(x)-1
w2(x):=y2(x)/z0(x)-1
w3(x):=y3(x)/z1(x)-1
SOLVE(DIF(w1(x),x)=0,x,Real)
  x=2*t/3
SOLVE(DIF(w2(x),x)=0,x,Real)
  x=2*(t+1)/3
SOLVE(DIF(w3(x),x)=0,x,Real)
  x=(2*t+1)/3
w1(2*t/3)
  SQRT(6)*(2*t+3)^(3/2)/18-1
w2(2*(t+1)/3)
  SQRT(6)*(2*t+5)^(3/2)/36-1
w3((2*t+1)/3)
  SQRT(6)*(t+2)^(3/2)/9-1
```

```
w1(2*t)
  SQRT(2)*SQRT(2*t+1)/2-1
w2(1)
  t/2-1/4
w3(0)
  t/2-1/4
w1(0)
  SQRT(2)*t/2+SQRT(2)/2-1
SOLVE(ABS(w1(2*t/3))=ABS(w1(2*t)),t,Real)
  4*t^6+36*t^5+81*t^4-216*t^3-972*t^2-2916*t+1458=0
NSOLVE(4*t^6+36*t^5+81*t^4-216*t^3-972*t^2-2916*t+1458=0,t,0.414,0.5)
  t=0.432744889959443195468521586996010373619782407838130499444930041043 17
FLOOR((190+t)*2^23)
  1597465647
OutputBase:=Hexadecimal
  5f37642f
```

B.2 solution.mth

```
PrecisionDigits:=100
NotationDigits:=100
z0(x):=SQRT(2)/SQRT(1+x)
z1(x):=1/SQRT(1+x)
y1(x):=1+t-x/2
y2(x):=1+t/2-x/4
y3(x):=3/4+t/2-x/4
y4(x):=SQRT(2)*y3(x)
q1(x):=y1(x-1)/SQRT(2)
  q1(x):=-SQRT(2)*(x-2*t-3)/4
q2(x):=y2(x-1)/SQRT(2)
  q2(x):=-SQRT(2)*(x-2*t-5)/8
q3(x):=y3(x-1)
  q3(x):=(2*(t+2)-x)/4
p1(x):=q1(x)*(3/2-x/2*q1(x)^2)
p2(x):=q2(x)*(3/2-x/2*q2(x)^2)
p3(x):=q3(x)*(3/2-x/2*q3(x)^2)
v1(x):=p1(x)*SQRT(x)-1
v2(x):=p2(x)*SQRT(x)-1
v3(x):=p3(x)*SQRT(x)-1
```

```
SOLVE(DIF(v1(x)=0,x),x,Real)
  x=(2*t+3)/3
SOLVE(DIF(v2(x)=0,x),x,Real)
  x=(2*t+5)/3
SOLVE(DIF(v3(x)=0,x),x,Real)
  x=2*(t+2)/3
SOLVE(v1(2*t/3+1)=v2(2*t+1),t,Real)
  64*t^6+576*t^5+2592*t^4+3888*t^3-26244*t+10935=0
NSOLVE(64*t^6+576*t^5+2592*t^4+3888*t^3-26244*t+10935=0,t,0.414,0.5)
  t=0.43245008479014264217878293749679646686135774283014672468921204774818
ABS(v1(2*t/3+1))
  0.00175118367122021335212517424670015453675424829637526886369927566607 04
FLOOR((190+t)*2^23)
  1597463174
OutputBase:=Hexadecimal
  0x5f375a86
FLOOR((FLOOR(3*1023/2)+t)*2^52)
  6910469410427058089
OutputBase:=Hexadecimal
  0x5fe6eb50c7b537a9
FLOOR((FLOOR(3*16383/2)+t)*2^112)
  12759774841085158312099207963122491795 1
OutputBase:=Hexadecimal
  0x5ffe6eb50c7b537a9cd9f02e504fcfbf
```

Vita

Candidate's full name:	Matthew Robertson
University attended:	BScCS, 2012, University of New Brunswick
Conference Presentations:	"A Brief History of InvSqrt"
	October 2011, Science Atlantic