Survey of the Fast Computation of the Reciprocal Square Root through Integer Operations on Floating-Point Values

Thomas Nelson

University of Massachusetts Lowell

Published July 27, 2017

Abstract

Finding a value's reciprocal square root has many uses in vector-based calculations, but arithmetic calculations of finding a square root and performing division are too computationally expensive for typical large-scale real-time performance constraints. This survey examines the "Fast Inverse Square Root" algorithm and explores the techniques of its implementation through examination of the Newton-Raphson Method of Approximation and the *magic-number* **0x5f3759df** which allow for the reciprocal square root to be calculated with only multiplication and subtraction operations.

1 Overview

The reciprocal of the square root of a value, $\frac{1}{\sqrt{x}}$, also called the "inverse square root" is necessary for vector calculations which are instrumental in common 3D-rendering applications. Due to the time-sensitive nature of these applications, there has been a great deal of development over the past 20 years in providing fast approximations of this reciprocal value. While hardware manufacturers began including instruction-set level methods of performing this calculation in 1999, it has taken some time for these to become widespread in end-user machines [5]. An algorithm to quickly find this reciprocal, known as the "Fast Inverse Square Root" algorithm, was popularized and publicized in the late 90s as a general-use solution.

This survey explores this "Fast Inverse Square Root" algorithm in detail, providing a full examination of the mathematical formula it employs as well as a derivation of the notorious *magic number* **0x5f3759df** found therein. The Newton-Raphson Method of Approximation, which may be used in a general sense to refine approximate solutions to a formula into more precise approximations, will also be explored in detail due to its core relevance to the "Fast Inverse Square Root" algorithm. A brief overview of IEEE 754 single-precision floating-point number representation will be made due to its fundamental relevance to the "floating-point bit-level hacking" employed by the algorithm, before the ramifications of this are made clear in the derivation of the *magic number*.

2 The Fast Inverse Square Root Algorithm

The "Fast Inverse Square Root Algorithm" (**FastInvSqrt()**) has been in use since the mid-to-late-90s, and became popularly known of and analyzed through the dissemination of its usage in the engine of the video game *Quake III Arena*. This algorithm was pivotal in the early development of real-time 3D graphics technologies. 3D graphics engines are required to calculate "surface normals" in order to express lighting and reflection (see figure 1). The calculation for a surface normal involves dividing a length of 1 by the square root of each of its coordinates squared $\frac{1}{\sqrt{x^2+y^2+z^2}}$ [6]. Traditional means of finding the square root and dividing it are computationally expensive, especially as compared to calculating $x^2+y^2+z^2$, but the "Fast Inverse Square Root Algorithm" makes this entire computation very fast.

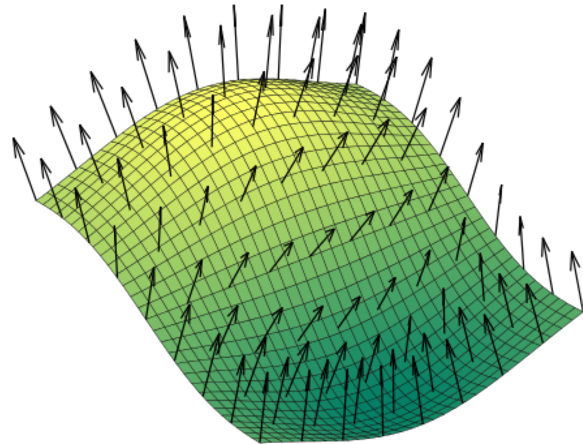

Figure 1: Example of Surface Normals[7]

Algorithmically, **FastInvSqrt()** is an implementation of the Newton-Raphson Approximation Method which is performed once from a very good initial guess, which has in turn been derived through a bit-level manipulation of the floating point representation of the input value [4]. This simple to compute yet high-quality initial guess for the Newton-Raphson Approximation Method is what makes this algorithm novel and innovative, as it allows for a degree of precision comparable to multiple passes of the approximation method for

arbitrary input values - with only a single pass required. In rare cases where higher-precision is demanded, this algorithm has an asymptotic complexity of $\mathbf{O(n\ lg(n))}$ for n passes of the Newton-Raphson Method [3].

This survey will provide an overview of the "Fast Inverse Square Root Algorithm" through analysis of its published source code, before explaining the mechanisms involved - the Newton-Raphson Approximation Method and the *magic number* which delivers such a good initial guess.

2.1 Algorithm Source Code

The notorious source code for "Fast Inverse Square Root", simplified slightly for brevity from the published *Quake III Arena* source of **Q_rsqrt()**, is as follows (original comments preserved)[6]:

```
1  float FastInvSqrt(float x) {
2    float xhalf = 0.5f * x;
3    int i = *(int*)&x;  // evil floating point bit level hacking
4    i = 0x5f3759df - (i >> 1);  // what the fuck?
5    x = *(float*)&i;
6    x = x*(1.5f-(xhalf*x*x));
7    return x;
8  }
```

2.2 Source Code Analysis

The source code may be described step-by-step as follows:

Step 1: Take an input floating-point number x.
Step 2: Calculate and store half of the input number as a floating point **xhalf**.
Step 3: Cast x to int32 format and store as i.
Step 4: Bitshift i to the right by one, and arithmetically subtract this bitshifted value from the *magic number* **0x5f3759df**.
Step 5: Recast i as a floating point, storing as x
Step 6: Perform the Newton-Raphson Approximation using the values of **xhalf** and x, overwriting the value of x.
Step 7: Return the value of x.

It is noteworthy that in order to increase precision, the Newton-Raphson Approximation method may be repeatedly invoked by repetition of Step 6. However, the full version of the most notorious implementations of this algorithm only perform one Newton-Raphson Approximation pass, and comments out the performance of a second pass (with the note to uncomment if additional precision is desired).

2.3 Algorithm Analysis

Without worrying about the details of the low-level implementation, the algorithm itself may be thought of as a two-step process:

Step 1: Use "evil floating-point bit-level hacking" involving a *magic number* to manipulate the input number into a very good initial guess.

Step 2: Use the Newton-Raphson Approximation Method to refine the precision from this very good initial guess, and return it.

Some academics consider the *magic number* **0x5f3759df** to be sub-optimal, in truth there exist other magic numbers aside from the one employed in this popular implementation[1][6]. As is, experimentation reveals this implementation of the "Fast Inverse Square Root" algorithm to perform roughly four times faster than `(float)(1.0/sqrt(x))` [1]. Working backwards from the calculated solution, this survey will first explain how the Newton-Raphson Method is used to refine a guess before exploring how the initial guess is found.

3 Fast Precision: The Newton-Raphson Method of Approximation

The Newton-Raphson Method of Approximation is a means of refining a guess at the root of a function (the value of x where $f(x) = 0$). The formula requires taking a guess, g_n, and subtracting from it the value of $f(g_n)$ over its derivative, $f'(g_n)$, in order to get the new guess g_{n+1} which is closer to the root of the function (that is to say, $f(g_{n+1})$ is closer to 0 than $f(g_n)$ was), as follows:

Newton-Raphson Method of Approximation Formula
$g_{n+1} = g_n - \frac{f(g_n)}{f'(g_n)}$

It should be made clear, that this is a general-purpose formula which can be adapted to any function with a known derivative in order to find the roots of the function. Nevertheless, this method can be easily used for refining a guess at the inverse square root problem: by finding a function $error(x)$ which measures how erroneous a guess x is for an initial input. Thus, in the "Fast Inverse Square Root" algorithm, we use the Newton-Raphson Method to minimize our error at guessing the inverse square root [4]. If we take i to be our initial value for the inverse square root, such that we are trying to solve for $\frac{1}{\sqrt{i}}$, we may use the following proof to find a means of minimizing $error(x)$:

Step	Equation	Operation
0	$x = \frac{1}{\sqrt{i}}$	Begin with initial function definition
1	$x^2 = \frac{1}{i}$	Square each side
2	$\frac{1}{x^2} = i$	Divide 1 by each side
3	$\frac{1}{x^2} - i = 0$	Subtract i from each side
4	$error(x) = \frac{1}{x^2} - i$	Use as function to find root of $error(x)$
5	$error(x) = x^{-2} - i$	Simplified error formula

With the chain-rule, we may now find the derivative of the error formula:

Error Formulae
$error(x) = x^{-2} - i$
$error'(x) = -2x^{-3}$

With those in hand, we may incorporate them into the Newton-Raphson Method of Approximation in order to find a means of minimizing the error of our guess (g_n) at the inverse square root of i and thus find a better guess (g_{n+1}):

Step	Equation	Operation
0	$g_{n+1} = g_n - \frac{error(g_n)}{error'(g_n)}$	Begin with the Newton-Raphson Method of Approximation, using $error(g_n)$ as $f(g_n)$
1	$g_{n+1} = g_n - \frac{g_n^{-2} - i}{error'(g_n)}$	Substitute in the error formula
2	$g_{n+1} = g_n - \frac{g_n^{-2} - i}{-2g_n^{-3}}$	Substitute in the derivative of the error formula
3	$g_{n+1} = g_n + \frac{g_n - ig_n^3)}{2}$	Combine numerator and denominator using exponent manipulation
4	$g_{n+1} = g_n(\frac{3}{2} - \frac{1}{2}ig_n^2)$	Simplify expression

From this point, one may refine a guess g_n into a better guess g_{n+1} using only g_n and the initial value i, which as used to minimize error corresponds to the initial value x in $\frac{1}{\sqrt{x}}$. This is a direct parallel to the code:

```
x = x(1.5f-(xhalf*x*x));
```

wherein **xhalf** is the stored value of half the original input (i). This final formula can be performed with only computationally cheap multiplication and subtraction. Now, it must be understood that if the initial guess is of poor quality then the Newton-Raphson Method requires multiple passes in order to ensure precision (see figure 2). It is assured that with the Newton-Raphson Method a guess at the inverse square root may be easily made more precise: all this approach requires for speed is a good initial guess.

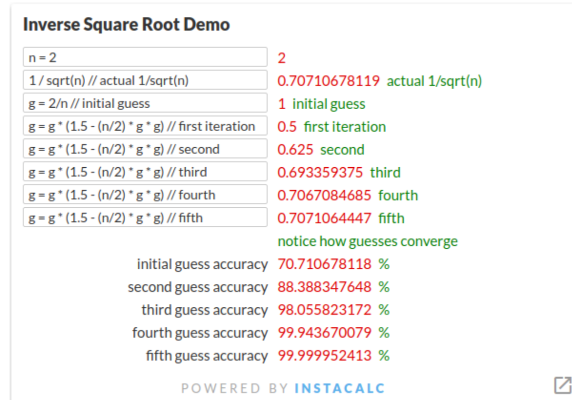

Figure 2: Multiple Newton-Raphson passes from a poor quality initial guess [4]

4 A Good Initial Guess: The Magic Number Explained

In order for the Newton-Raphson Method of Approximation to arrive at an effective solution quickly, it must be given as good of a quality initial guess as possible. This is where the *magic number* **0x5f3759df** comes into play, as this constant provides a means of arriving at a very good guess with nothing more than a bitshift operation and a subtraction operation.

4.1 IEEE 754 Floating-Point Representation

Because the "Fast Inverse Square Root" algorithm works on bit-level manipulation of floating-point values, it is critical that we have an understanding of how numbers are stored within the IEEE 754 32-bit single-precision floating-point standard. In this standard, the 32-bit **float** is partitioned into three sections, the sign bit s, the 8-bits for the exponent e, and 23-bits for the mantissa m, as shown below:

s	e	m
31	30 29 29 27 26 25 24 23	22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

In accordance to the IEEE 754 standard these values are not interpreted as-is, which is to say that each partition is not interpreted as a bitwise integer value. Instead, each partition is interpreted across a standardized range of values, in accordance to the IEEE 754 standard. This discrepancy is what facilitates the speed of the “Fast Inverse Square Root Algorithm”, which manipulates a cast bitwise integer interpretation of the floating-point value’s bit pattern in order to arrive at a very good guess very quickly. As our analysis of the algorithm has shown, the “evil floating-point bit-level hacking” involves subtracting a bitshifted original value from a *magic number* - which may be derived from this IEEE 754 implementation’s rules along with a few algebraic approximations.

4.2 Preparing to Derive the Magic Number

The *magic number* of the “Fast Inverse Square Root Algorithm” is derivable through algebraic manipulation of the floating-point representation, bitwise conversion between integer and floating-point of the mantissa and exponent, and the inverse square root function. In order to derive the *magic number* **0x5f3759df** we must provide representations of the exponent and mantissa elements of a floating-point number, e and m respectively, with the mantissa representing a value between 0 and 1, and the exponent representing a value between -127 and 128. The algorithm is dependent on the bitwise relation between the floating-point value and the integer value of the same bits, and the conversion equations between these interpretations. Thus, while e and m represent the exponent and mantissa as interpreted by floating-point representation, E and M will be used to represent the exponent and mantissa as interpreted by integer representation. Then, these formulae may be used for conversion between the floating-point and integer interpretations[6]:

Conversion	Equation
floating-point mantissa from bitwise integer mantissa	$m = \frac{M}{L}$
floating-point exponent from bitwise integer exponent	$e = E - B$

Variable	Value	Range
m	floating-point interpretation of the mantissa	$[0, 1]$
M	bitwise integer interpretation of the mantissa	$[0, (2^{23} - 1)]$
e	floating-point interpretation of the exponent	$[-127, 128]$
E	bitwise integer interpretation of the exponent	$[0, 255]$
L	mantissa range: 2^{23} for IEEE 754 floating-point	-
B	exponent bias: 127 for IEEE 754 floating-point	-

We may thus calculate the value of a floating-point number or its bitwise integer interpretation using these formula:

Equivalent to	Calculation
floating-point number value (F)	$F = (1 + m)2^e$
integer interpretation value (I)	$I = M + LE$

As part of the derivation of the *magic number*, we must employ a Logarithm-Linear Approximation Formula. This Logarithm-Linear Approximation Formula posits that the base 2 logarithm of $1 + x$ is very close to a straight line for values of x between 0 and 1 (see figure 3), and that this straight line's formula is approximately x plus some offset value σ [6].

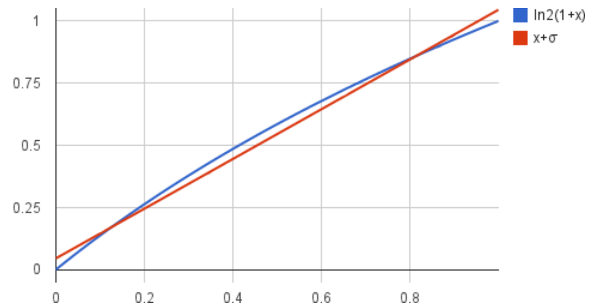

Figure 3: $log_2(1 + x)$ versus $x + \sigma$ [6]

Logarithm-Linear Approximation Formula
$log_2(1 + x) \approx x + \sigma$

4.3 Derivation of the Magic Number

With all of this prepared, we may conduct the derivation:

Step	Equation	Operation
0	$y = \frac{1}{\sqrt{x}} = x^{-\frac{1}{2}}$	Begin with equation for reciprocal square root
1	$log_2 y = -\frac{1}{2} log_2 x$	Take the base 2 logarithm of both sides
2	$log_2((1+m_y)2^{e_y}) = -\frac{1}{2}(log_2((1+m_x)2^{e_x}))$	Substitute in floating-point components
3	$log_2(1+m_y) + e_y = -\frac{1}{2}(log_2(1+m_x) + e_x)$	Carry through logarithm of exponent
4	$m_y + \sigma + e_y \approx -\frac{1}{2}(m_x + \sigma + e_x)$	Employ Logarithm-Linear Approximation Formula
5	$\frac{M_y}{L} + \sigma + E_y - B \approx -\frac{1}{2}(\frac{M_x}{L} + \sigma + E_x - B)$	Substitute in bitwise integer representations of mantissa and exponent
6	$\frac{M_y}{L} + E_y \approx -\frac{1}{2}(\frac{M_x}{L} + \sigma + E_x - B) - \sigma + B$	Add $(-\sigma + B)$ to both sides
7	$\frac{M_y}{L} + E_y \approx -\frac{1}{2}(\frac{M_x}{L} + E_x) - \frac{3}{2}(\sigma - B)$	Combine $-\frac{1}{2}(\sigma - B)$ with $-\sigma + B$
8	$M_y + LE_y \approx \frac{3}{2}L(B - \sigma) - \frac{1}{2}(M_x + LE_x)$	Multiply both sides by L
9	$I_y \approx \frac{3}{2}L(B - \sigma) - \frac{1}{2}I_x$	Substitute in integer representation value I for $M + LE$
10	$I_y \approx K - \frac{1}{2}I_x$	Substitute in constant K

At this point, we have found that the integer representation of y is an unknown constant minus half of the integer representation of x:
$I_y \approx K - \frac{1}{2}I_x$
which is a direct parallel to the code:

```
i = K - (i >> 1);
```

This value K is our *magic number*, a constant from which we subtract half the original integer representation in order to calculate a very good guess at the reciprocal square root. Thus, our derivation continues by finding the value of K for IEEE 754 floating-point, substituting the proper values for L and B into the expression $\frac{3}{2}L(B-\sigma)$:

Step	**Equation**	**Operation**
11	$K = \frac{3}{2}L(B-\sigma)$	Begin with constant expression from previous step
12	$K = \frac{3}{2}2^{23}(B-\sigma)$	Substitute in 2^{23} for value of L in IEEE 754 floating-point
13	$K = \frac{3}{2}2^{23}(127-\sigma)$	Substitute in 127 for value of B in IEEE 754 floating-point

From this point, a value must be chosen for σ which produces the best approximation. While alternative σ values have been suggested by various scholars [1], this survey will focus on the ramifications of the σ value chosen for the "Fast Inverse Square Root" algorithm implementation, $\sigma = 0.0450465$:

Step	**Equation**	**Operation**
14	$K = \frac{3}{2}2^{23}(127-0.0450465)$	Substitute in 0.0450465 for value of σ
15	$K = 1597463007$	Simplify expression
16	$K =$ **0x5f3759df**	Convert to hexadecimal

Once we bring this constant K back to our integer representation value I_y approximation formula, we are given our final derivation:

Step	**Equation**	**Operation**
17	$I_y \approx$ **0x5f3759df** $-\frac{1}{2}I_x$	Substitute in *magic number* for K

From this point, the "Fast Inverse Square Root" algorithm runs this initial guess value of I_y through the Newton-Raphson Method of Approximation, in order to improve the level of precision. The actual utilization of the *magic number* involves a couple datatype recasts, a bitshift, and a subtraction operation. From this initial guess value, the algorithm is as simple as multiplication and subtraction.

5 Conclusion

The “Fast Inverse Square Root” algorithm is remarkable in its approach to solving what seems to be an arithmetically difficult problem by employing only bitshift, multiplication, recast, and subtraction operations. It is able to perform this through careful sacrifices of precision by means of sufficiently accurate approximation formulae. By using the Logarithm-Linear Approximation Formula, $log_2(1+x) \approx x + \sigma$ with a suitable σ value, a very good derived *magic number* may be used to generate a very good initial guess using nothing but two recast operations, a bitshift, and a subtraction operation. From here, the algorithm’s derivation from the Newton-Raphson Approximaton Method formula demands nothing more than four multiplication operations and a subtraction operation. Truly, this is an impressive feat of design through novel and innovative mathematical derivation.

References

[1] Chris Lomont, "FAST INVERSE SQUARE ROOT," Department of Mathematics, Perdue University, West Lafayette, Feb. 28, 2003. Available: http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf [Accessed 2 June 2017].

[2] Saad Zafar and Raviteja Adapa, "Hardware architecture design and mapping of 'Fast Inverse Square Root' algorithm," in *The 2014 International Conference on Advances in Electrical Engineering (ICAEE)*, India, Jan, 2014. Available: http://www.dline.info/ed/fulltext/v3n1/4.pdf [Accessed 8 June 2017].

[3] Alexander Yee, "Internals - Inverse Square Root," Jan 29, 2017. [Online]. Available: http://www.numberworld.org/y-cruncher/internals/invsqrt.html [Accessed 10 June 2017].

[4] Kalid Azad, "Understanding Quake's Fast Inverse Square Root," [Online]. Available: https://betterexplained.com/articles/understanding-quakes-fast-inverse-square-root [Accessed 11 June 2017].

[5] Stuart F. Oberman, "Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7™ Microprocessor," [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.7202&rep=rep1&type=pdf [Accessed 11 June 2017].

[6] Christian P. Hansen. "0x5f3759df," [Online]. Available: http://h14s.p5r.org/2012/09/0x5f3759df.html [Accessed 23 July 2017].

[7] Nicoguaro. "Vectores normales a la superficie $xe^{-x^2-y^2}$," [Online]. Available: https://commons.wikimedia.org/wiki/File:Surface_normals.svg [Accessed 26 July 2017].

University of Massachusetts Lowell: Department of Computer Science,
220 Pawtucket St, Lowell, MA 01854
Email address: Thomas_Nelson@student.uml.edu
Web address: www.tjourney.com

First written: July 2017