Fast Inverse Square Root (Revisited)

David Eberly
Geometric Tools, LLC
http://www.geometrictools.com/

Created: January 26, 2002
Last Modified: July 20, 2010

Contents

1 Introduction

This document is about a fast approximation for computing the inverse square root of a 32-bit floating-point number. The original version of this document is provided here just for the historical records. I had not spent much time analyzing the algorithm, but provided enough of a hint as to how the algorithm depended on the floating-point encoding of the input number. The modified version is in the second section, and is draft material for a book I am writing entitled *Numerical Computing for Games and Science*. The book has a lot of details and algorithms regarding IEEE 754-2008 floating-point number systems.

2 The Original Version (January 26, 2002)

The following code is an edited version of code posted to `comp.graphics.algorithms` on January 9, 2002. The subject that started the thread was *Fast 2D distance approximation* of which the posted code was one of the follow-ups. The posted code is purported to be from Quake3 and provides an approximation to the inverse square root of a number, $1/\sqrt{x}$.

```
float InvSqrt (float x)
{
    float xhalf = 0.5f*x;
    int i  = *(int*)&x;
    i = 0x5f3759df - (i >> 1);  // This line hides a LOT of math!
    x = *(float*)&i;
    x = x*(1.5f - xhalf*x*x);  // repeat this statement for a better approximation
    return x;
}
```

So what does this code really do and what is that magic number `0x5f3759df`?

The idea is to specify an x and compute y so that $y = 1/\sqrt{x}$. Define $F(y) = 1/y^2 - x$. The y you want is the positive root of $F(y) = 0$. You can solve this with Newton's method. Choose an initial guess y_0. The iteration scheme is

$$y_{n+1} = y_n - F(y_n)/F'(y_n), \;\; n \geq 0$$

where $F'(y) = -2/y^3$ is the derivative of $F(y)$. The equation reduces to

$$y_{n+1} = \frac{y_n(3 - xy_n^2)}{2}.$$

In the limit as you increase n, the y_n values converge to the true value of $1/\sqrt{x}$. If y_0 is a good initial guess, then 1 or 2 iterations should give you a decent approximation. The source code had the second iteration commented out, so I suspect one iteration was good enough for Quake3's purposes.

Now the problem is selecting a good initial guess. This is where the line of code involving x, manipulated as an integer via variable i, is clever. The IEEE 32-bit float has a mantissa M filling bit positions 0 through 22, an 8-bit biased exponent E filling bits 23 through 30, and a sign bit in position 31. The function expects nonnegative input, so the sign bit is 0 for the input x. The bias is 127. The true exponent is $e = E - 127$. The corresponding number in readable form is $x = 1.M * 2^e$. You want y_0 to be a good approximation to $1/\sqrt{x} = (1/\sqrt{1.M}) * 2^{-e/2}$.

The biased exponent for $-e/2$ is $-e/2+127$. In terms of integer arithmetic, this is `0xbe - (E >> 1)` where E is the biased exponent for x. Now look at the magic number `0x5f3759df` that shows up in the code. The sign bit is 0. The next 8 bits form the hex number `0xbe`. No coincidence! The statement

```
i  = 0x5f3759df - (i >> 1);
```

implicitly computes the biased exponent $-e/2+127$.

Now you need an approximation for $1/\sqrt{1.M} = 1/\sqrt{1+M}$ where $0 \le M < 1$. Define $G(M) = 1/\sqrt{1+M}$. You can approximate this by a linear function $T(M) = 1-(M/2)$ using a Taylor series expansion at $M = 0$. The approximation is good for M near zero, but not good at $M = 1$. In fact, as M increases the difference between $G(M)$ and $T(M)$ increases (G is always larger). To try to balance the differences, you want a better fitting line, one that cuts through the graph of $G(M)$. The one corresponding to the posted code is

$$L(M) = 0.966215 - M/4$$

Figure 2.1 shows the graph of $G(M)$, the linear function $T(M)$, and the linear function $L(M)$.

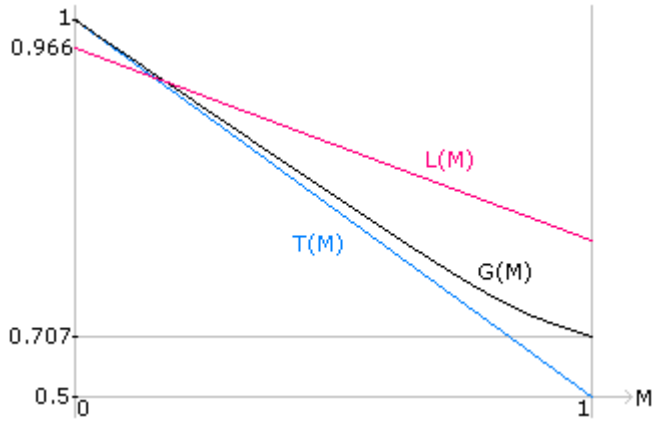


Figure 2.1

For $0 \le M < 1$, $L(M)$ produces numbers in $[0.715215, 0.966215]$ which are not normalized. Instead write

$$L(M) = (1.93243 - M/2)/2.$$

The values $1.93243 - M/2$ are 1+something, so are normalized. The actual floating point representation used for 1.93243 is `0x3ff759df`. When you subtract 1 from the exponent to account for the division by 2 in $L(M)$, you get `0x5f3759df`, the magic number in the code. Therefore, the statement

```
i  = 0x5f3759df - (i >> 1);
```

also implicitly computes the mantissa for the initial guess y_0.

I do not know why $0.966215 - M/4$ was chosen in the first place. I thought it might be based on requiring the slope to be $-1/4$ and choosing a constant C using a least squares integral approach that minimizes the integral of squared errors between $C - M/4$ and $1/\sqrt{1+M}$ where the integration is on $0 \le M \le 1$, but a quick check showed this is not the case.

3 The Modified Version (July 16, 2010)

In February 2003, Chris Lomont provided a document that was motivated by my January 2002 draft. The essence of the problem is to look at the floating-point representation for the input, but my hasty analysis of the trailing significand of the input was incomplete. I mentioned the linear approximation used for generating initial guesses for Newton's method, but as it turns out, the analysis of the trailing significand leads to a polyline approximation with three line segments.

3.1 A Minimax Approach

Lomont has a lengthy analysis of the problem as an attempt to show how the magic number might be chosen, the hope to justify the choice `0x5f3759df` in the posted source code. The analysis shows how to choose the magic number to minimize the maximum relative error over all relevant floating-point numbers. Although not explicitly mentioned, it is sufficient to analyze the problem for all floating-point numbers in the interval $[1/2, 2)$. There are $n = 2^{24}$ numbers in the interval, call them x_i for $1 \le i \le n$. If c is the magic number, the initial guess for Newton's method are generated by

```
float Initial (float x, float c)
{
    float xhalf = 0.5f*x;
    int i  = *(int*)&x;
    i = c - (i >> 1);
    x = *(float*)&i;
    return x;
}
```

The problem is to choose c such that

$$E(c) = \max_i \left| \frac{\frac{1}{x_i} - \text{Initial}(x_i, c)}{\frac{1}{x_i}} \right| = \max_i |1 - x\,\text{Initial}(x_i, c)|$$

is minimized. This is referred to as a *minimax problem*, where the minimum is

$$\varepsilon = \min_c E(c) = \min_c \max_i |1 - x\,\text{Initial}(x_i, c)|$$

The original magic number is $c_{\text{orig}} =$ `0x5f3759df`. The magic number from the minimax analysis is $c_{\text{minmax}} =$ `0x5f37642f`.

Lomont reports his numerical tests to verify that indeed, c_{minmax} leads to a smaller maximum relative error than does c_{orig}. However, he observes that after the Newton iterate, the maximum relative error for c_{orig} is smaller than that for c_{minmax}. He says: "Yet surprisingly, after one Newton iteration, it has a higher maximal relative error.", "The reason the better approximation turned out worse must be in the Newton iteration.", and "Out of curiosity, I searched numerically for a better constant." The result is not surprising. The goal is to choose the magic number so that the accuracy of the *output after the Newton iteration* is best according to some optimization criterion. Indeed, the "curious search" is where the analysis should have been in the first place—a minimax analysis after the Newton iterate is computed. This analysis leads to a magic number $c_{\text{lomont}} =$ `0x5f375a86` with minimax error (approximately) 0.00175122. This number may be obtained without understanding the line of code containing the magic number. For example,

```
// First pass:  cMin = 0x5f330000, cMax = 0x5f380000, cDelta = 0x100
// Second pass:  Select cMin and cMax to bound the minimum observed
//   from the first pass.  cMin = 0x5f375a00, cMax = 0x5f375c00, cDelta = 1

FILE* output = fopen("results.txt", "wt");
uint32_t cMin = <value>, cMax = <value>, cDelta = <value>;
for (uint32_t c = cMin; c < cMax; c += cDelta)
{
    double maxError = 0.0;
    for (uint32_t i = 0; i < (1 << 24); ++i)
    {
        union { uint32_t encoding; float number; } u;
        u.encoding = 0x3F000000u + i;  // u.number in [0.5,2)
        double x = u.number;
        double xhalf = 0.5*x;
        double z = sqrt(x);
        u.encoding = c - (u.encoding >> 1);
        double y = x*(1.5 - xhalf*x*x);

        double error = fabs(1.0 - y*z);
        if (error > maxError)
        {
            maxError = error;
        }
    }

    uint64_t uMaxError = *(uint64_t*)&maxError;
    fprintf(output, "c = 0x%.8x , error = 0x%.16I64x\n", c, uMaxError);
}
fclose(output);
```

Browsing the results after the second pass, you will find the value of c that minimizes the maximum. I actually found `0x5f375a5` to be the minimum, but a difference of one bit can be explained by round-off errors if the experiment was coded in a mathematically equivalent manner but different from a floating-point perspective (different from for expressions, use of 32-bit or 64-bit in different ways, and so on).

3.2 Analysis of the Algorithm

Let $x = 1.t \star 2^e$ be the positive input to the inverse square root function. The integer t is the trailing significand and the notation $1.t$ represents a number in the interval $[1, 2)$. The bits of t are the bits of the fractional part of the number. For a 32-bit floating-point number, t has 23 bits,

$$t = t_{22}t_{21}\cdots t_1t_0$$

and the significand $1.t$ is

$$1.t = 1 + \frac{t_{22}}{2^1} + \frac{t_{21}}{2^2} + \cdots \frac{t_1}{2^{22}} + \frac{t_0}{2^{23}}$$

The unbiased exponent e is an integer. Floating-point numbers store a biased exponent, which is $e + 127$ for 32-bit numbers.

The inverse square root of x is

$$\frac{1}{\sqrt{x}} = \frac{1}{\sqrt{1.t}} \star 2^{-e/2} = \begin{cases} \frac{\sqrt{2}}{\sqrt{1.t}} \star 2^{-e/2-1/2}, e \text{ is odd} \\ \frac{2}{\sqrt{1.t}} \star 2^{-e/2}, e \text{ is even} \end{cases}$$

The right-hand equality splits the representation based on the parity of the exponent e. The reason is that the exponent of the result must be an integer value and the significand must be a number in $[1, 2)$. Observe

that $\sqrt{2}/\sqrt{1.t} \in [1, \sqrt{2}) \subset [1, 2)$ and $2/\sqrt{1.t} \in [\sqrt{2}, 2) \subset [1, 2)$. At this early stage of the analysis, it suffices to choose approximations $1.\alpha(t)$ to $\sqrt{2}/\sqrt{1.t}$ and $1.\beta(t)$ to $2/\sqrt{1.t}$, where $\alpha(t)$ and $\beta(t)$ are nonnegative integer-valued functions of t. We may do so independently of the exponent e, and we may even do so independent of floating-point encodings. There are a *lot of possibilities*. However, the posted `FastInvSqrt` is an algorithm that implicitly imposes the approximations to the significands, and they are based on the floating-point encoding of x.

Let us look at computing the biased exponent for the output. When e is odd, say, $e = 2p + 1$, then the biased exponent $\bar{e} = e + 127$ is even. The floor of $\bar{e}$ is $\lfloor \bar{e}/2 \rfloor = p + 64$. Moreover,

$$190 - (\bar{e} >> 1) = 190 - \lfloor \bar{e}/2 \rfloor = -p - 1 + 127 = -e/2 - 1/2 + 127$$

Therefore, $190 - (\bar{e} >> 1)$ is the biased exponent for $1/\sqrt{x}$ when $\bar{e}$ is the even biased exponent for x.

When e is even, say, $e = 2p$, then the biased exponent $\bar{e} = e + 127$ is odd. The floor of $\bar{e}$ is $\lfloor \bar{e}/2 \rfloor = p + 63$. Moreover,

$$189 - (\bar{e} >> 1) = 189 - \lfloor \bar{e}/2 \rfloor = -p - 1 + 127 = -e/2 + 127$$

Therefore, $189 - (\bar{e} >> 1)$ is the biased exponent for $1/\sqrt{x}$ when $\bar{e}$ is the odd biased exponent for x.

Now consider the binary encoding of the magic number in the posted code,

```
c = 0x5f3759df = 0 [10111110] [r22 r21 ... r1 r0] = 0 [10111110] [01101110101100111011111]
```

where the bit groups make clear the biased exponent bits and the trailing significand bits. Observe that the biased exponent is 190. The binary encoding for x is i and has representation

```
i = 0 [e7 e6 e5 e4 e3 e2 e1 e0] [t22 t21 ... t1 t0]
(i >> 1) = 0 [0 e7 e6 e5 e4 e3 e2 e1] [e0 t22 ... t1]
```

The difference $c - (i >> 1)$ involves a subtraction of the trailing significands and a subtraction of the exponents. The latter subtraction is $190 - (\bar{e} >> 1)$. There is a complication, however. The subtraction of the trailing significands might require a carry out of the exponent bits, making the exponent subtraction $189 - (\bar{e} >> 1)$. We have seen these expressions previously, when representing $1/\sqrt{x}$ in normal form. A more detailed analysis is required, which involves four cases depending on the parity of $\bar{e}$ and whether or not a carry out of the exponent is required in the subtraction. The analysis will be general for $r = r_{22} \cdots r_0$.

3.2.1 Exponent e is Odd

Let e be odd, so $\bar{e}$ is even. If i is the binary encoding for x, then $i >> 1$ has trailing significand $0(t >> 1)$, which represents the integer $t = t_{22} \cdots t_1$ prepended with a zero. The subtraction $c - (i >> 1)$ is written formally as

$$\begin{array}{rccccccccccccc}
 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & r_{22} & r_{21} & \cdots & r_0 \\
- & 0 & 0 & e_7 & e_6 & e_5 & e_4 & e_3 & e_2 & e_1 & 0 & t_{22} & \cdots & t_1 \\
\hline
 & 0 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 & d_{22} & d_{21} & \cdots & d_0
\end{array}$$

and $1.d$ is the significand of the initial guess used for Newton's method. If r_{22} were chosen to be 1, then there is no carry out from the exponent bits on a subtraction. However, the original magic number has $r_{22} = 0$.

Because the bits of t vary over all possible choices of 0 and 1, there must be values of t that lead to a carry out. Specifically, $r \geq 0(t >> 1)$, then there is no carry out on subtraction and

$$1.d = 1.r - 0.0(t >> 1)$$

The initial approximation is therefore

$$\frac{1}{\sqrt{x}} \doteq (1.r - 0.0(t >> 1)) \star 2^{190-(\bar{e}>>1)}$$

If $r < 0(t >> 1)$, then there is a carry out on subtraction. It is necessary that $r_{22} = 0$ and $d_{22} = 1$ in this case. The result is

$$1.d = 2.r - 0.0(t >> 1)$$

where the 2 in $2.r$ is the consequence of the carry out. The initial approximation is therefore

$$\frac{1}{\sqrt{x}} \doteq (2.r - 0.0(t >> 1)) \star 2^{189-(\bar{e}>>1)} = \left(\frac{2.r - 0.0(t >> 1)}{2}\right) \star 2^{190-(\bar{e}>>1)}$$

where the 189 occurs because of the carry out but the right-hand side adjusts the term to 190 so that the significand is in $[1, 2)$.

3.2.2 Exponent e is Even

Let e be even, so $\bar{e}$ is odd. If i is the binary encoding for x, then $i >> 1$ has trailing significand $1(t >> 1)$, which represents the integer $t_{22} \cdots t_1$ prepended with a one. The subtraction $c-(i >> 1)$ is written formally as

$$\begin{array}{rcccccccccccc} & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & r_{22} & r_{21} & \cdots & r_0 \\ - & 0 & 0 & e_7 & e_6 & e_5 & e_4 & e_3 & e_2 & e_1 & 1 & t_{22} & \cdots & t_1 \\ \hline & 0 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 & d_{22} & d_{21} & \cdots & d_0 \end{array}$$

and $1.d$ is the significand of the initial guess used for Newton's method. Because the bits of t vary over all possible choices of 0 and 1, there must be values of t that lead to a carry out. Observe that for the original magic number, $r_{22} = 0$ in which case there must *always be a carry out*. For the sake of completeness, however, here are the two possibilities when r_{22} is chosen to be 1. If $r \geq 1(t >> 1)$, then there is no carry out on subtraction and

$$1.d = 1.r - 0.1(t >> 1)$$

The initial approximation is therefore

$$\frac{1}{\sqrt{x}} \doteq (1.r - 0.1(t >> 1)) \star 2^{190-(\bar{e}>>1)}$$

If $r < 1(t >> 1)$, then there is a carry out on subtraction. The result is

$$1.d = 2.r - 0.1(t >> 1)$$

where the 2 in $2.r$ is the consequence of the carry out. The initial approximation is therefore

$$\frac{1}{\sqrt{x}} \doteq (2.r - 0.1(t >> 1)) \star 2^{189-(\bar{e}>>1)} = \left(\frac{2.r - 0.1(t >> 1)}{2}\right) \star 2^{190-(\bar{e}>>1)}$$

where the 189 occurs because of the carry out but the right-hand side adjusts the term to 190 so that the significand is in $[1, 2)$.

3.2.3 Summary of Cases

The initial approximation for the inverse square root of $x = 1.t \star 2^e$ for a magic number c corresponding to a biased exponent 190 and a trailing significand r is summarized next, where $\bar{e} = e + 127$,

$$\frac{1}{\sqrt{x}} \doteq \begin{cases} (1.r - 0.0(t >> 1)) \star 2^{190-(\bar{e}>>1)}; & \bar{e} \text{ even}, r \geq 0.0(t >> 1) \\ \left(\frac{2.r-0.0(t>>1)}{2}\right) \star 2^{190-(\bar{e}>>1)}; & \bar{e} \text{ even}, r < 0.0(t >> 1) \\ (1.r - 0.1(t >> 1)) \star 2^{190-(\bar{e}>>1)}; & \bar{e} \text{ odd}, r \geq 0.1(t >> 1) \\ \left(\frac{2.r-0.1(t>>1)}{2}\right) \star 2^{190-(\bar{e}>>1)}; & \bar{e} \text{ odd}, r < 0.1(t >> 1) \end{cases} \tag{1}$$

Both constants c_{orig} and c_{minmax} have bit r_{22} set to zero, so the case for even $\bar{e}$ and $r \geq 0.1(t >> 1)$ cannot occur. The initial approximation is therefore a polyline with three segments. In the following images, the horizontal axis represents $x \in [1/2, 2)$. The right-most vertical gray line is at $x = 1/2$. The left-most vertical gray line is where $r = 0.0(t >> 1)$. The vertical axis represents the interval $[0.70, 1.44]$. The blue curve is $1/\sqrt{x}$ for $x \in [1/2, 1)$ and the cyan curve is $1/\sqrt{x}$ for $x \in [1, 2)$. The red segment is the initial guess for $\bar{e}$ even and $r \geq 0.0(t >> 1)$. The orange segment is the initial guess for $\bar{e}$ even and $r < 0.0(t >> 1)$. The magenta segment is the initial guess for $\bar{e}$ odd and $r < 0.1(t >> 1)$.

Figure 3.1 shows the initial approximation for c_{orig}.

Figure 3.1 Image corresponding to the initial approximation when the magic number is c_{orig}.

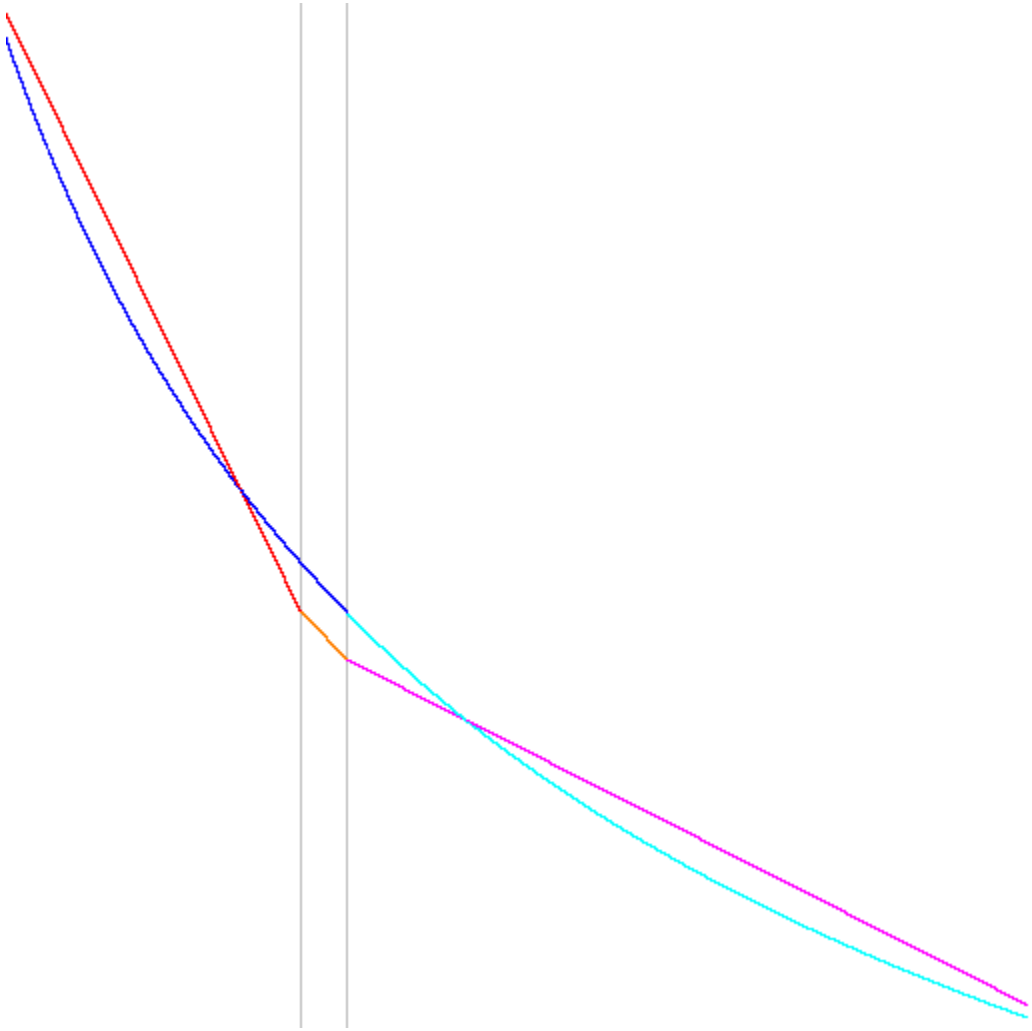


Figure 3.2 shows the approximation for c_{orig} after the Newton iterate.

Figure 3.2 Image corresponding to the output after the Newton iterate when the magic number is c_{orig}.

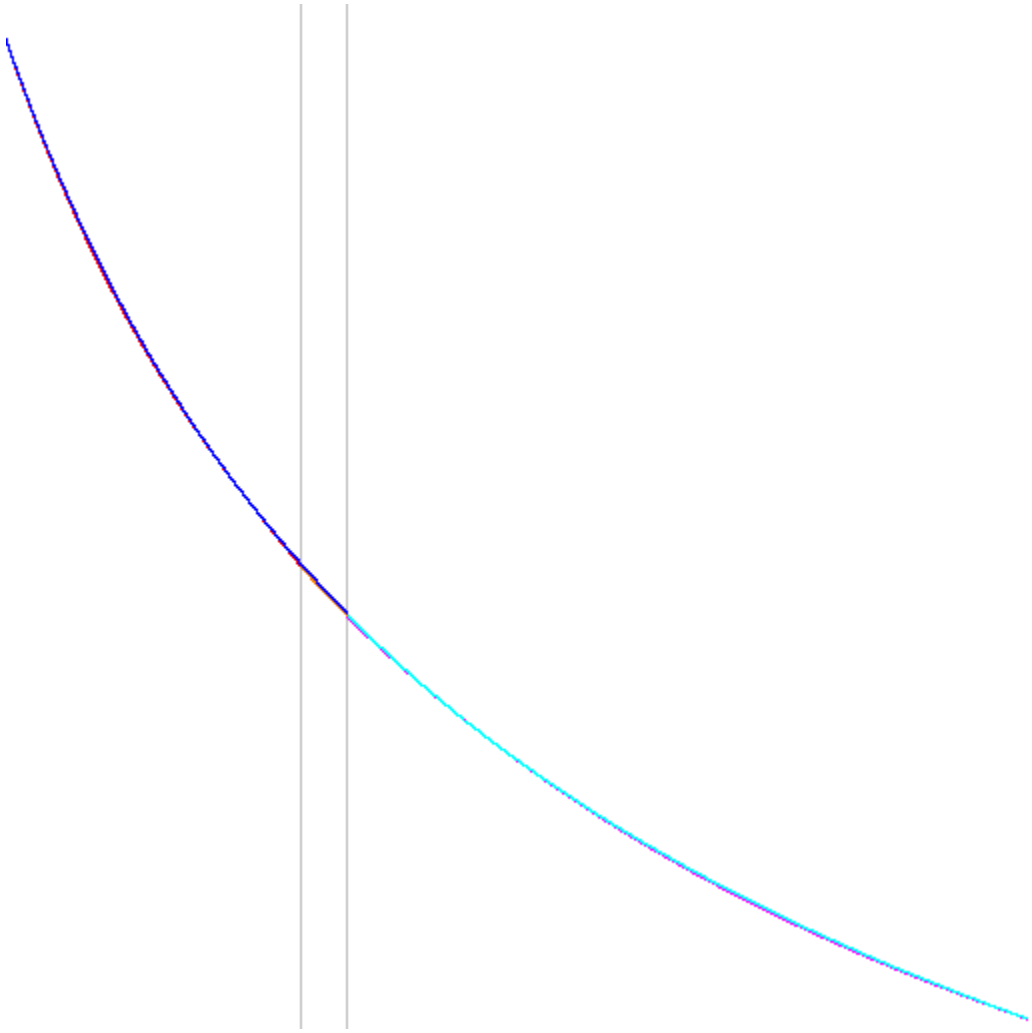


Figure 3.3 shows the initial approximation for c_{minmax}.

Figure 3.3 Image corresponding to the initial approximation when the magic number is c_{minmax}.

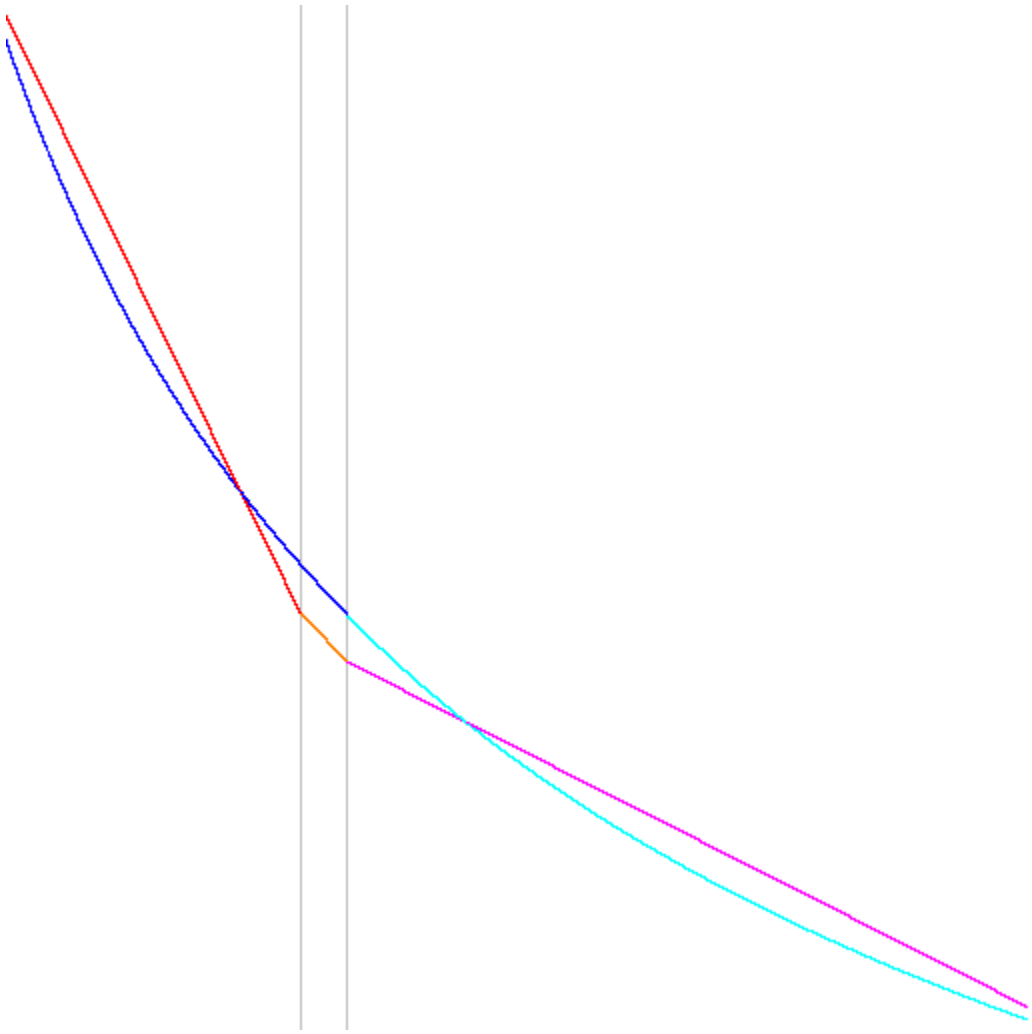
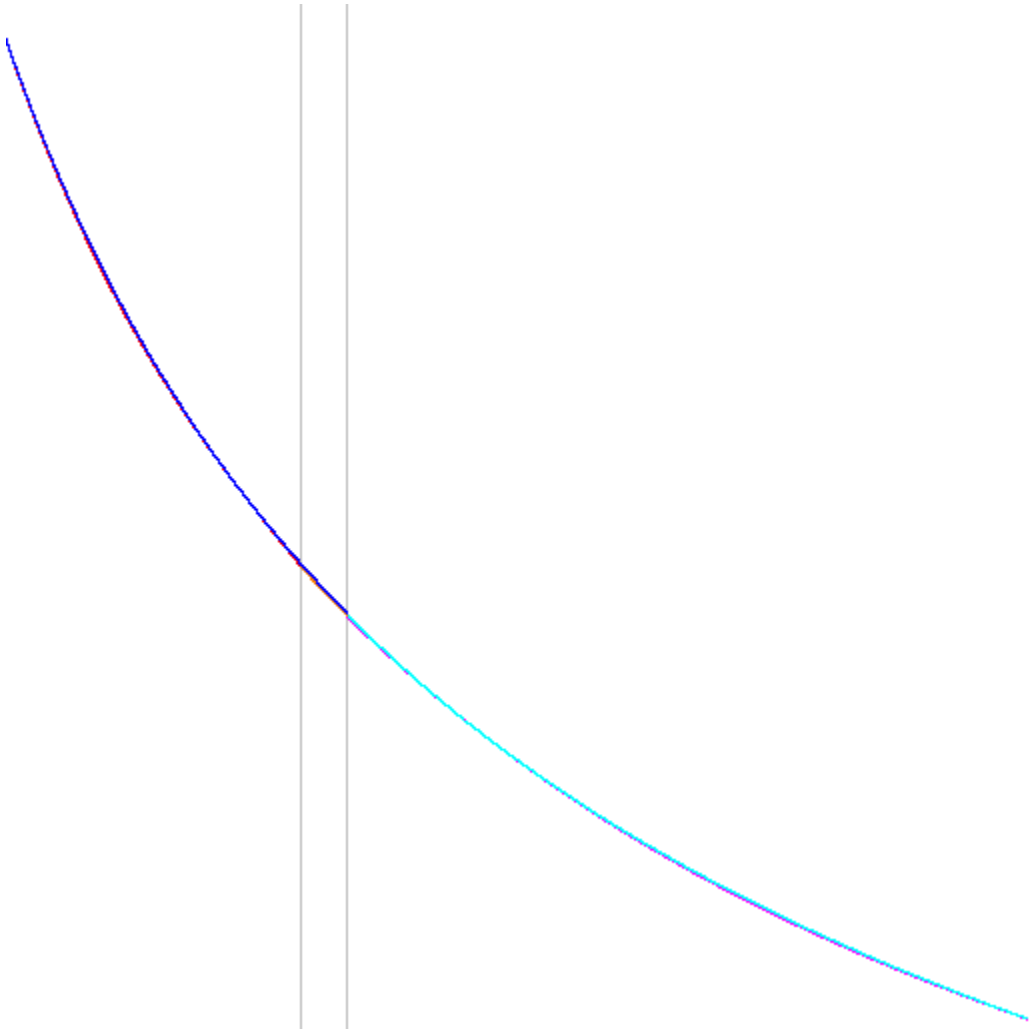


Figure 3.4 shows the approximation for c_{minmax} after the Newton iterate.

Figure 3.4 Image corresponding to the output after the Newton iterate when the magic number is c_{minmax}.



It is difficult visually to distinguish between the two images when the Newton iterate is applied.

3.3 Other Choices for Magic Numbers

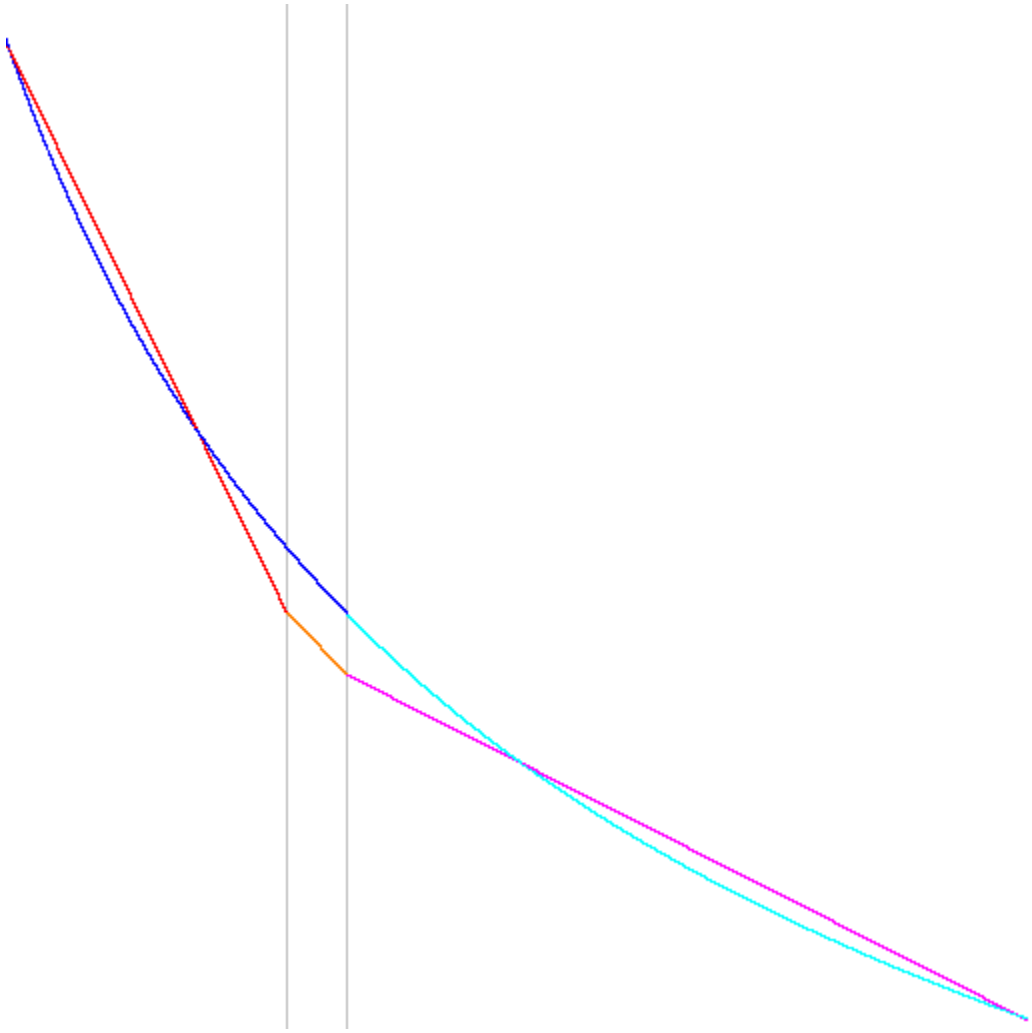
The following table shows choices of magic numbers that minimize various norms for the data.

type	minimize	magic number
original	Who knows.	`0x5f3759df`
L^1 abs	$\sum_i \lvert 1/\sqrt{x_i} - y_i \rvert$	`0x5f34ca30`
L^2 abs	$\sum_i \lvert 1/\sqrt{x_i} - y_i \rvert^2$	`0x5f360131`
L^3 abs	$\sum_i \lvert 1/\sqrt{x_i} - y_i \rvert^3$	`0x5f366be2`
L^∞ abs	$\max_i \lvert 1/\sqrt{x_i} - y_i \rvert$	`0x5f370c57`
L^1 rel	$\sum_i \lvert 1 - \sqrt{x_i} y_i \rvert$	`0x5f34bf4e`
L^2 rel	$\sum_i \lvert 1 - \sqrt{x_i} y_i \rvert^2$	`0x5f360739`
L^3 rel	$\sum_i \lvert 1 - \sqrt{x_i} y_i \rvert^3$	`0x5f3680e5`
L^∞ rel	$\max_i \lvert 1 - \sqrt{x_i} y_i \rvert$	`0x5f375a85`

The last table entry is the choice suggested by Lomont because it leads to the minimum over all magic numbers of the maximum relative error. If you minimize the L^1 norm for the relative errors, the maximum relative error is not quite twice that for the L^∞ norm for the relative errors. However, if you compute both approximations over all floating-point numbers in $[1/2, 2)$ and count how many times the `0x5f34bf4e`-based approximation has a smaller relative error than the `0x5f375a85`-based approximation, you will find that the former wins 11859800 times and the latter wins 4917417 times (for a total of $2^{24} = 166777215$ values). The `0x5f360739`-based approximation wins 12778028 times and loses 4499189 times, which amounts to winning about 3/4 of the time. Although the minimax relative error is attractive, having a more accurate answer 3/4 of the time might cause you to choose a magic number different from `0x5f375a85`. If you use the original magic number `0x5f3759df`, you win 12685492 times and lose only 4091725 times.

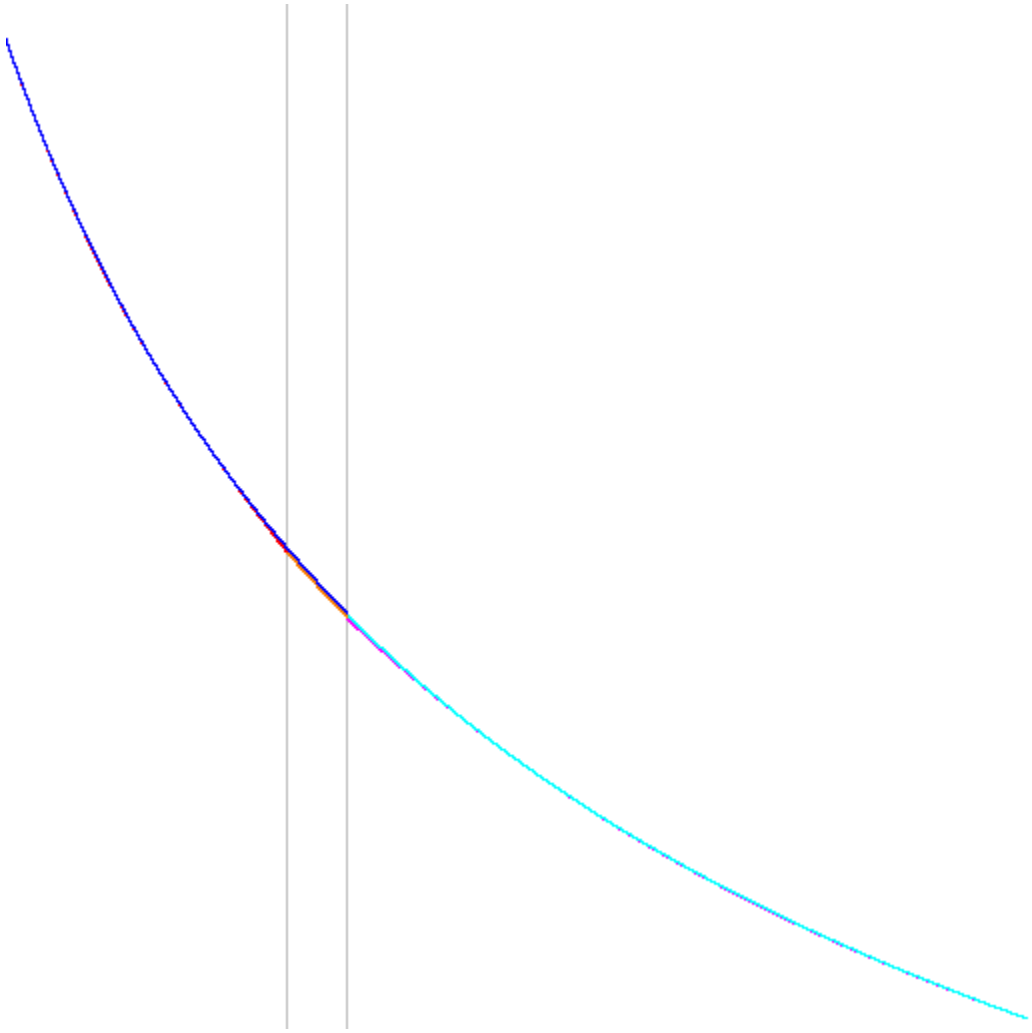
Comparing Figures 3.2 and 3.4, it is difficult to see much difference. However, now look at Figure 3.5, which shows the initial approximation using `0x5f34bf4e`,

Figure 3.5 Image corresponding to the initial approximation when the magic number is `0x5f34bf4e`.



Comparing to Figures 3.1 and 3.4, you see that the approximation is better near the endpoints $x = 0.5$ and $x = 2$ but worse near the segment between the gray vertical lines. Figure 3.6 shows the approximation for `0x5f34bf4e` after the Newton iterate.

Figure 3.6 Image corresponding to the output after the Newton iterate when the magic number is `0x5f34bf4e`.



Comparing to Figures 3.2 and 3.4, a quick glance might convince you there is no significant difference. However, based on the order of the pixel draw, notice that the images for the original and minimax magic numbers show many more red and magenta pixels than for the L^1 image when outside the strip bounded by the gray vertical lines. When inside the vertical lines, the L^1 image shows a somewhat larger error than for the other images. The L^1-based approximation wins more often, which tends to happen a lot outside the vertical strip.

One minor variation that might help improve the approximation is to add a small value to the Newton iterate,

$$y = y(3/2 - x * y * y/2) + \varepsilon$$

I have not experimented much, but did notice that for various choices of magic number, you can reduce the maximum relative error with a judicious choice of ε.

3.4 About the Original Magic Number

I tried many experiments to attempt to construct the original magic number in some formal mathematical manner. Of course, you can always choose a large enough p so that the minimum of the L^p norm occurs at the original number, but I doubt that is how the number was chosen.

The closest I came was with the following code.

```
FILE* output = fopen("results.txt", "wt");
for (uint32_t magic = 0x5f3759d0u; magic < 0x5f3759f0u; magic += 1u)
{
    double maxError = 0.0;
    const int jmax = 128;
    for (uint32_t j = 0; j <= jmax; ++j)
    {
        float x = 0.5f + 1.5f*j/(float)jmax;
        double z = sqrt((double)x);
        float xhalf = 0.5f*x;
        uint32_t bits = *(uint32_t*)&x;
        bits = magic - (bits >> 1);
        x = *(float*)&bits;
        double y = x*(1.5f - xhalf*x*x);
        double error = fabs(1.0 - y*z);
        if (error > maxError)
        {
            maxError = error;
        }
    }

    uint64_t uMaxError = *(uint64_t*)&maxError;
    fprintf(output, "magic = 0x%.8x , error = 0x%.16I64x , %10.8lf\n", magic, uMaxError, maxError);
}
fclose(output);
```

The minimum of the maximum errors occurs for magic number `0x5f3759d4`, which is quite close to `0x5f3759df`. If you compute everything using `float`, the minimum of the maximum errors has floating-point encoding `0x3ae524ea`. This number is attained by `0x5f3759d8` through `0x5f3759db`. The next maximum error larger than the minimum is `0x3ae5250b` and is attained by `0x5f3759c4` through `0x5f3759d7`. The next error larger than this one is `0x3ae5281e` and is attained by `0x5f3759dc` through `0x5f3759e5`. As you can see, the maximum-error function is quite flat and sensitive to whether you use 32-bit or 64-bit arithmetic. Moreover, a small change in `jmax` can cause the minimizing magic number to vary quite a bit. If you change `jmax` to 127, the minimum of the maximum errors is `0x3ae43e1c` and is attained by `0x5f3756de` through `0x5f3756e3`. It is quite possible that the inventor of the original magic number solved the minimax problem for only a small subset of numbers (such as 128) on the interval $[1/2, 2)$. Slight variations in the terms of the expressions, use of 32-bit or 64-bit values, and perhaps non-IEEE-compliant floating-point hardware might have contributed to the final choice of magic number (that is not possible to reproduce easily without knowledge of exactly the minimization performed). If a minimax approach is used, it should be computed over all floating-point numbers in $[1/2, 2)$.

4 Accurate Inverse Square Root

The fast approximation to the inverse square root may be used as the basis for computing more accurate values for the inverse square root function. The pseudocode is

```
typedef union { uint32_t encoding; float number; } binary32;
const uint32_t magic = <some magic number>;
binary32 x, y;
float xHalf, yPrev;
int numIterates;

x.number = <some number in [1/2,2)>;
xHalf = 0.5f*x.number;
y.encoding = magic - (x.encoding >> 1);
yPrev = 0.0f;
numIterates = 0;
while (y.number != yPrev)
{
    yPrev = y.number;
    y.number *= 1.5f - xHalf*y.number*y.number;
    ++numIterates;
}
```

The following table shows the iterations for several magic numbers. The iteration counts per line are for all the floating-point numbers in $[1/2, 2)$. The range `0x5f300000` through `0x5f3fffff` was exhaustively searched (the program ran for a few days using all four cores on a quad core machine).

type	magic number	1	2	3	4	5	6	7	8	9	total
	0x5f100000					16652454	24762				84010842
	0x5f1fffff				1838715	14938426	75				82047440
	0x5f200000				1838727	14938414	75				82047428
	0x5f2fffff			3533067	13145836	98313					63674110
	0x5f300000			3533129	13145775	98312					63674047
min average	0x5f32b693	43	148291	9498999	7111402	18481					57331635
L^1 rel	0x5f34bf4e	23	90188	4848600	11834232	4173					62083992
L^1 abs	0x5f34ca30	22	89631	4774551	11908992	4020					62159005
L^2 abs	0x5f360131	10	38130	3129923	13607506	1647					63904298
L^2 rel	0x5f360739	10	38038	3103521	13634026	1621					63930858
L^3 abs	0x5f366be2	10	36545	2690612	14048669	1380					64346512
L^3 rel	0x5f3680e5	9	36306	2613326	14126227	1348					64424247
L^∞ abs	0x5f370c57	9	34431	2218971	14522371	1434					64822438
original	0x5f3759df	8	33540	2123222	14618634	1812					64920350
L^∞ rel	0x5f375a85	9	33574	2122702	14619115	1816					64920803
	0x5f375a86	10	33568	2122712	14619110	1816					64920802
	0x5f3fffff	7	11703	688968	12826787	3249751					69646220
	0x5f400000	1	11697	688972	12826767	3249779					69646274
	0x5f4fffff					16696050	81166				83967246
	0x5f500000					16696050	81166				83967246
	0x5f5fffff					708710	16068094	412			99954998
	0x5f600000					708696	16068108	412			99955012
	0x5f6fffff						7277776	9499439	1		110162737
	0x5f700000						7277750	9499465	1		110162763
	0x5f7fffff							5947029	10829878	309	128271008

The table shows that magic number `0x5f32b693` produces the minimum average iterations, $57331635/16777216 \doteq 3.42$ iterations per floating-point number.

5 Other Algorithms

Just as we did for square roots, we may introduce a second sequence into the Newton's method, $y_{i+1} = y_i(1.5 - 0.5xy_i^2)$. Two possibilities are discussed.

Firstly, let $z_i = 1.5 - 0.5xy_i^2$, in which case $y_{i+1} = y_i z_i$. Observe that

$$z_{i+1} - z_i = -0.5x(y_{i+1}^2 - y_i^2) = -0.5xy_i^2(z_i - 1)(z_i + 1) = (z_i - 1.5)(z_i^2 - 1)$$

Choosing an initial guess y_0 and $z_0 = 1.5 - 0.5xy_0^2$, we then iterate the equations

$$y_{i+1} = y_i z_i, \;\; z_{i+1} = z_i + (z_i - 1.5)(z_i^2 - 1)$$

It is also possible to set $w_i = z_i - 1$, which implies $w_{i+1} = w_i^2(w_i + 1.5)$. In either case, the number of arithmetic operations is larger than that for the y-only equation, so this is most likely not a useful alternative in a floating-point system because of the additional costs per floating-point number.

Secondly, let $z_i = 0.5xy_i$, in which case $y_{i+1} = y_i(1.5 - y_i z_i)$. Observe that

$$z_{i+1} - z_i = 0.5x(y_{i+1} - y_i) = 0.5xy_i(0.5 - y_i z_i) = z_i(0.5 - y_i z_i)$$

Choosing an initial guess y_0 and $z_0 = 0.5xy_0^2$, we then iterate the equations

$$y_{i+1} = y_i(1.5 - y_i z_i), \;\; z_{i+1} = z_i(1.5 - y_i z_i)$$

The expression $(1.5 - y_i z_i)$ need only be evaluated once per iteration. This requires two arithmetic operations. The computation of y_{i+1} and z_{i+1} requires two more arithmetic operations, a total of four operations that is the same as the y-only method. Unfortunately, numerical experiments showed that at least one more iteration was required for convergence compared to the y-only method. Floating-point round-off errors adversely affected this implementation, even though the algorithm is mathematically equivalent to the y-only method.