# FAST INVERSE SQUARE ROOT

CHRIS LOMONT

ABSTRACT. Computing reciprocal square roots is necessary in many applications, such as vector normalization in video games. Often, some loss of precision is acceptable for a large increase in speed. This note examines and improves a fast method found in source-code for several online libraries, and provides the ideas to derive similar methods for other functions.[1]

## 1. INTRODUCTION

Reading the math programming forum on `www.gamedev.net` [1], I ran across an interesting method to compute an inverse square root. The C code was essentially (my comments):

```
float InvSqrt(float x)
      {
      float xhalf = 0.5f*x;
      int i = *(int*)&x;       // get bits for floating value
      i = 0x5f3759df - (i>>1); // gives initial guess y0
      x = *(float*)&i;         // convert bits back to float
      x = x*(1.5f-xhalf*x*x);  // Newton step, repeating increases accuracy
      return x;
      }
```

The interesting part was the constant `0x5f3759df`: where did it come from and why does the code work? Some quick testing in Visual C++.NET [2] showed the code above to be roughly 4 times faster than the naive `(float)(1.0/sqrt(x))`, and the maximum relative error over all floating point numbers was 0.00175228, so the method seems very useful. Three immediate questions were: 1) how does it work, 2) can it be improved, and 3) what bit master designed such an incredible hack?

A quick search on Google for `0x5f3759df` returned several hits, the most relevant being a reference to a thread about this code on `comp.graphics.algorithms` from Jan 9, 2002 [3], and an (incorrect,

[1] *2000 Mathematics Subject Classification.* Primary 65G99, 65Y99; Secondary 68W40.

*Key words and phrases.* Inverse square root, speed tradeoffs, IEEE 754.

but close) explanation by D. Eberly [4]. However his explanation is illuminating. Further digging found no correct explanation of this code. It appears in the sourcecode for *Quake 3*, written by legendary game programmer John Carmack, but someone on the net (I cannot now find the reference) credited it to Gary Tarolli who was at Nvidia. Can anyone confirm authorship? It also appears in the *Crystal Space* sourcecode [5], the *Titan Engine* [6], and the *Fast Code Library*, although each seems to derive from *Quake 3*.

The motivation to try such an algorithm is more clearly explained in Eberly [4], where he assumes the shift creates a linear interpolation to the inverse square root. Note there are several ways to speed up this code, but this note will not go into further optimizations. There are also faster methods, perhaps table lookup tricks, but most of them have less accuracy than this method.

This note assumes PC architecture (32 bit floats and ints) or similar. In particular the floating point representation is IEEE 754-1985 [7]. This C code has been reported to be endian-neutral (supposedly tested it on a Macintosh). I have not verified this. Since the method works on 32 bit numbers it seems (surprisingly) endian-neutral. It is easy to extend the code to other situations and bit sizes (such as type `double`) using the ideas in this note. Anyway, on to the problems 1), 2), and 3).

## 2. Background

Floating point numbers are stored in the PC as 32 bit numbers;

| s | E | M |
|---|---|---|
| *bit* 31 | $30 \leftarrow bits \rightarrow 23$ | $22 \leftarrow bits \rightarrow 0$ |

where $s$ is a 1 bit sign (1 denotes negative), $E$ is an 8 bit exponent, and $M$ is a 23 bit mantissa. The exponent is biased by 127 to accommodate positive and negative exponents, and the mantissa does not store the leading 1, so think of $M$ as a binary number with the decimal point to the left, thus $M$ is a value in $I = [0, 1)$. The represented value is

$$x = (-1)^s(1 + M)2^{E-127}$$

These bits can be viewed as the floating point representation of a real number, or thinking only of bits, as an integer. Thus $M$ will be considered a real number in $I$ or as an integer, depending on context. $M$ as a real number is $M$ as an integer divided by $2^{23}$.

## 3. The Algorithm

The main idea is Newton approximation, and the magic constant is used to compute a good initial guess. Given a floating point value $x > 0$, we want to compute $\frac{1}{\sqrt{x}}$. Define $f(y) = \frac{1}{y^2} - x$. Then the value we seek is the positive root of $f(x)$. Newton's root finding method, given a suitable approximation $y_n$ to the root, gives a better one $y_{n+1}$ using

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}, \quad n \geq 0$$

For the $f(y)$ given, this simplifies to $y_{n+1} = \frac{1}{2}y_n(3 - xy_n^2)$ which corresponds to the line of code `x = x*(1.5f-xhalf*x*x)` , where `x` is the initial guess, which hereafter will be called $y_0$.

The line of code `i = 0x5f3759df - (i>>1)` computes this initial guess $y_0$, roughly by multiplying the exponent for $x$ by $-\frac{1}{2}$, and then picking bits to minimize error. `i>>1` is the *shift right operator* from C, which shifts the bits of an integer right one place, dropping the least significant bit, effectively dividing by 2. Eberly's explanation was that this produced linear approximation, but is incorrect; we'll see the guess is piecewise linear, and the function being approximated is not the same in all cases. However I would guess his explanation was the inspiration for creating this algorithm.

## 4. The Magic Number(s)

Thus we are left with finding an initial guess. Suppose we are given a floating point value $x > 0$, corresponding to the 32 bits

| | | |
|---|---|---|
| 0 | E | M |

as above. Let the exponent $e = E - 127$. Then $x = (1+M)2^e$, and the desired value $y = \frac{1}{\sqrt{x}} = \frac{1}{\sqrt{1+M}}2^{-e/2}$, treating $e$ and $M$ as real numbers, *NOT* as integers. For the general case we take a magic constant $R$, and analyze $y_0 = R-$`(i>>1)`, where the subtraction is as integers, `i` is the bit representation of $x$, but we view $y_0$ as a real number. $R$ is the integer representing a floating point value with bits

| | | |
|---|---|---|
| 0 | $R_1$ | $R_2$ |

i.e., $R_1$ in the exponent field and $R_2$ in the mantissa field. When we shift `i` right one bit, we may shift an exponent bit into the mantissa field, so we will analyze two cases.

For the rest of this note, for a real number $\alpha$ let $\lfloor \alpha \rfloor$ denote the integer less than or equal to $\alpha$.

4.1. **Exponent E Even.** In this case, when we use `i>>1`, no bits from the exponent $E$ are shifted into the mantissa $M$, and $\lfloor E/2 \rfloor = E/2$. The true exponent $e = E - 127$ is odd, say $e = 2d + 1$, $d$ an integer. The bits representing the initial guess give the new exponent

$$\begin{aligned} R_1 - \lfloor E/2 \rfloor &= R_1 - E/2 \\ &= R_1 - \frac{e+127}{2} \\ &= R_1 - \frac{2d+1+127}{2} \\ &= R_1 - 64 - d \end{aligned}$$

*We require this to be positive.* If it were negative the resulting sign bit would be 1, and the method fails to return a positive number. If this result is 0, then the mantissa part could not borrow from the exponent, which we will see below is necessary. Since this must be true for any even $E \in [0..254]$, we require $R_1 \geq 128$.

Since the exponent $E$ is even, no bits from it shift into the mantissa under `i>>1`, so the new mantissa is $R_2 - \lfloor M/2 \rfloor$ (as integers), assuming $R_2 \geq \lfloor M/2 \rfloor$. The initial guess is then

$$\begin{aligned} y_0 &= (1 + R_2 - M/2)2^{(R_1-64-d)-127} \\ &= (1 + R_2 - M/2)2^{R_1-191-d} \\ &= (2 + 2R_2 - M)2^{R_1-192-d} \end{aligned}$$

where $M/2$ replaced $\lfloor M/2 \rfloor$, since the resulting error is at most $2^{-24}$ in the mantissa (as a real number), which is insignificant compared to other errors in the method.

If $R_2 < M/2$, then upon subtracting $R-$`(i>>1)` as integers, the bits in the mantissa field will have to borrow one from the exponent field (this is why we needed the new exponent *positive* above), thus dividing $y_0$ by two. The *bits* in the mantissa field will then be $(1+R_2) - \lfloor M/2 \rfloor$, which are still in $I$. Thus if $R_2 < M/2$

$$\begin{aligned} y_0 &= (1 + (1 + R_2 - M/2))2^{(R_1-64-d)-127-1} \\ &= (2 + R_2 - M/2)2^{R_1-192-d} \end{aligned}$$

where we write the exponent the same as in the non-carrying case.

If we define

$$\beta_M^{R_2} = \begin{cases} 2R_2 - M & : \quad R_2 \geq M/2 \\ R_2 - M/2 & : \quad R_2 < M/2 \end{cases}$$

then we can combine these two $y_0$ equations into one:

$$y_0 = (2 + \beta_M^{R_2})2^{R_1-192-d}$$

Note that $\beta_M^{R_2}$ is continuous, and differentiable except along $R = M/2$. Substituting $e = 2d + 1$, the value $y = \frac{1}{\sqrt{x}}$ we are trying to approximate is

$$\frac{1}{\sqrt{1+M}}2^{-e/2} = \frac{1}{\sqrt{1+M}}2^{-d-1/2} = \frac{1}{\sqrt{2}\sqrt{1+M}}2^{-d}$$

The relative error $\left|\frac{y-y_0}{y}\right|$, which is how we will measure the quality of the method, is

$$\left|\frac{\frac{1}{\sqrt{2}\sqrt{1+M}}2^{-d} - (2+\beta_M^{R_2})2^{R_1-192-d}}{\frac{1}{\sqrt{2}\sqrt{1+M}}2^{-d}}\right|$$

which simplifies to $|\varepsilon_0(M, R)|$,

$$\varepsilon_0(M, R) = 1 - \sqrt{2}\sqrt{1+M}(2 + \beta_M^{R_2})2^{R_1-192}$$

Note that this no longer depends on $d$, and that $M$ can be any value in $I = [0, 1)$. This formula is only valid if $E$ is even, thus has a hidden dependence on $E$.

Suppose we want $R_2$ so that the relative error $\max_{M\in I} |\varepsilon_0| < 1/8$. Then

$$\begin{aligned}
\frac{1}{8} &> \max_{M\in I} |\varepsilon_0| \\
&\geq |\varepsilon_0(0, R_2)| \\
&= \left|1 - \sqrt{2}(2 + 2R_2)2^{R_1-192}\right| \\
&= \left|1 - (1 + R_2)2^{R_1-190.5}\right|
\end{aligned}$$

Expanding,

$$-1/8 < 1 - (1 + R_2)2^{R_1-190.5} < 1/8$$

$$\frac{9}{8} \geq \frac{9}{8(1+R_2)} > 2^{R_1-190.5} > \frac{7}{8(1+R_2)} > \frac{7}{16}$$

where the last step used the fact that $R_2 \in I \Rightarrow (1 + R_2) \in [1, 2)$. Taking $\log_2$ and adding 190.5 gives $190.7 > R_1 > 189.2$, so $R_1 = 190 =$ `0xbe` is the unique integer that has any *chance* of keeping the relative error below 0.125 for even $E$. In bit positions 24 through 30, this gives the leading `(0xbe>>1) = 0x5f` part of the magic constant $R$ (as well as requiring bit 23 to be 0).

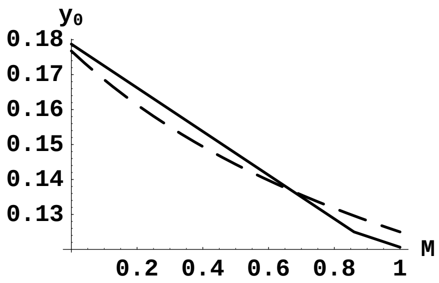


Figure 1

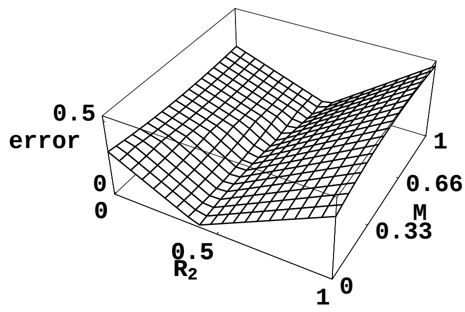


Figure 2

To illustrate, Figure 1 shows $y_0$ as a solid line and the function $(2+2M)^{-1/2}2^{-d}$ needing approximated as a dashed line, for $R_2 = 0.43$, $d = 2$. It is clear that $y_0$ is a nonlinear approximation; however, by being nonlinear, it actually fits better! Figure 2 shows the relative error $|\varepsilon_0(M, R_2)|$ for $R_2 \in I$ and $M \in I$. It is clear that the constant $R_2$ cross section with the smallest maximal error is approximately $R_2 = 0.45$

4.2. **Exponent E Odd.** With the previous case under our belt, we analyze the harder case. The difference is that the odd bit from the exponent $E$ shifts into the high bit of the mantissa from the code `i>>1`. This adds $\frac{1}{2}$ to the real value of the shifted mantissa, which becomes $\lfloor M/2 \rfloor + \frac{1}{2}$, where the truncation is as integers and the addition is as real numbers. $e = E - 127 = 2d$ is even. Similar to above the new exponent is

$$\begin{aligned} R_1 - \lfloor E/2 \rfloor &= R_1 - \left\lfloor \frac{e+127}{2} \right\rfloor \\ &= R_1 - \left\lfloor \frac{2d+127}{2} \right\rfloor \\ &= R_1 - 63 - d \end{aligned}$$

This must be positive as in the previous case.

Again write $M/2$ instead of $\lfloor M/2 \rfloor$ for the reasons above. The new mantissa is $R_2 - (M/2 + 1/2)$ as real numbers when $R_2 \geq \frac{M+1}{2}$, requiring no borrow from the exponent. Then

$$\begin{aligned} y_0 &= (1 + R_2 - (M/2 + 1/2))2^{(R_1 - 63 - d) - 127} \\ &= (1/2 + R_2 - M/2)2^{R_1 - 190 - d} \\ &= (2 + 4R_2 - 2M)2^{R_1 - 192 - d} \end{aligned}$$

Again we choose the same exponent for $y_0$ as in the case when $E$ is even. If $R_2 < \frac{M+1}{2}$, the subtraction $R-$`(i<<1)` needs to borrow one from the (positive) exponent, which divides $y_0$ by 2, and the bits in the mantissa field are then $1 + R_2 - (\frac{M}{2} + \frac{1}{2})$ as real numbers, which is still in $I$. So if $R_2 < \frac{M+1}{2}$ we get for the initial guess

$$\begin{aligned} y_0 &= (1 + (1 + R_2 - (M/2 + 1/2)))2^{(R_1-63-d)-127-1} \\ &= (3/2 + R_2 - M/2)2^{R_1-191-d} \\ &= (3 + 2R_2 - M)2^{R_1-192-d} \end{aligned}$$

Defining (similarly to $\beta_M^{R_2}$)

$$\gamma_M^{R_2} = \begin{cases} 4R_2 - 2M & : \quad 2R_2 \geq M + 1 \\ 1 + 2R_2 - M & : \quad 2R_2 < M + 1 \end{cases}$$

we can write both of these $y_0$ at once as

$$y_0 = (2 + \gamma_M^{R_2})2^{R_1-192-d}$$

Note that $\gamma_M^{R_2}$ is continuous, and differentiable except along $2R_2 = M + 1$, so $y_0$ is also. Using $e = 2d$, we want $y_0$ to approximate

$$y = \frac{1}{\sqrt{x}} = \frac{1}{\sqrt{1+M}}2^{-e/2} = \frac{1}{\sqrt{1+M}}2^{-d}$$

Note this is *not* the same function we were approximating in the case $E$ even; it is off by a factor of $\sqrt{2}$.

The relative error (which we want to minimize) simplifies as above to $|\varepsilon_1(M,R)|$, where

$$\varepsilon_1(M,R) = 1 - \sqrt{1+M}(2 + \gamma_M^{R_2})2^{R_1-192}$$

again independent of d (but with the same hidden dependence on $E$ as above).

Also as before, if we want the relative error $\max_{M\in I} |\varepsilon_1| < 1/8$ for $E$ odd, we can take $M = 0$ (or $M = 1$) and analyze, but since we want the same constant $R_1$ to work for $E$ even and for $E$ odd, we take $R_1 = 190$ from above for both cases. Note that this satisfies the earlier requirement that $R_1 \geq 128$. So for the rest of this note assume $R_1 = 190 =$ `0xbe` and redefine the two error functions as

$$\begin{aligned} \varepsilon_0(M,R_2) &= 1 - \sqrt{2}\sqrt{1+M}(2 + \beta_M^{R_2})/4 \\ \varepsilon_1(M,R_2) &= 1 - \sqrt{1+M}(2 + \gamma_M^{R_2})/4 \end{aligned}$$

depending only on $M$ and $R_2$, each taking values in I. At this point we thoroughly understand what is happening, and computer tests confirm the accuracy of the model for the approximation step, so we have achieved goal 1).

4.2.1. *An Example.* When $x = 16$, $E = 4 + 127 = 131$, $M = 0$, and a bit from the exponent shifts into the mantissa. So after the line `i = 0x5f3759df - (i>>1)`, the initial guess $y_0$ is `0x3E7759DF`. The new exponent is $190 - \lfloor 131/2 \rfloor = 125 =$ `0x7d`, which corresponds to $e = 125 - 127 = -2$. The new mantissa needs to borrow a bit from the exponent, leaving $e = -3$, and is `0xb759df - 0x4000000 = 0x7759DF`, corresponding to $M = 0.932430148$. Thus $y_0 = (1 + M)2^{-3} = 0.241553$, which is fairly close to $\frac{1}{\sqrt{x}} = 0.25$.

4.3. **Optimizing the Mantissa.** We want the value of $R_2 \in I$ that minimizes $\max_{M \in I}\{|\varepsilon_0(M, R_2)|, |\varepsilon_1(M, R_2)|\}$. Fix a value of $R_2$, and we will find the value(s) of $M$ giving the maximal error. Since $\varepsilon_0$ and $\varepsilon_1$ are continuous and piecewise differentiable, this $M$ will occur at an endpoint of a piece or at critical point on a piece.

4.3.1. *Maximizing* $|\varepsilon_0|$. Start with the endpoints for $\varepsilon_0$: $M = 0$, $M = 1$, and $R_2 = M/2$ (where $\beta_M^{R_2}$ is not differentiable). When $M = 0$, $\beta_M^{R_2} = 2R_2$. Let

$$f_1(R_2) = \varepsilon_0(0, R_2) = 1 - \sqrt{2}\sqrt{1+0}(2 + 2R_2)/4 = 1 - \frac{1 + R_2}{\sqrt{2}}$$

Similarly, when $R_2 = M/2$, $\beta_M^{R_2} = 0$;

$$f_2(R_2) = \varepsilon_0(2R_2, R_2) = 1 - \sqrt{2}\sqrt{1 + 2R_2}(2 + 0)/4 = 1 - \sqrt{\frac{1 + 2R_2}{2}}$$

When $M = 1$ we need to consider the two cases for $\beta_M^{R_2}$, giving

$$f_3(R_2) = \varepsilon_0(1, R_2) = \begin{cases} \frac{1}{2}(1 - 2R_2) & : \quad R_2 \geq 1/2 \\ \frac{1}{4}(1 - 2R_2) & : \quad R_2 < 1/2 \end{cases}$$

Checking the critical points takes two cases. Assuming $R_2 \geq M/2$, solving $\frac{\partial \varepsilon_0}{\partial M} = 0$ gives the critical point $M = \frac{2}{3}R_2$, giving

$$f_4(R_2) = \varepsilon_0(\frac{2R_2}{3}, R_2) = 1 - (1 + \frac{2}{3}R_2)^{3/2}/\sqrt{2}$$

When $R_2 < M/2$, the critical point is $M = \frac{2}{3}(1+R_2)$. This can only happen if $R_2 \in [0, \frac{1}{2})$, so define

$$f_5(R_2) = \begin{cases} 0 & : \quad R_2 \geq 1/2 \\ \varepsilon_0(\frac{2}{3}(1+R_2), R_2) = 1 - \frac{(5+2R_2)^{3/2}}{6\sqrt{6}} & : \quad R_2 < 1/2 \end{cases}$$

4.3.2. *Maximizing* $|\varepsilon_1|$. Similar analysis on $\varepsilon_1$ yields

$$f_6(R_2) = \varepsilon_1(0, R_2) = \begin{cases} \frac{1}{2} - R_2 & : \quad R_2 \geq 1/2 \\ \frac{1}{4}(1 - 2R_2) & : \quad R_2 < 1/2 \end{cases}$$

$$f_7(R_2) = \varepsilon_1(1, R_2) = \begin{cases} 1 - \sqrt{2}R_2 & : \quad R_2 \geq 1 \\ 1 - (1+R_2)/\sqrt{2} & : \quad R_2 < 1 \end{cases}$$

At the point $R = \frac{M+1}{2}$, $\gamma_M^{R_2} = 2$, but this can only occur when $R_2 \geq 1/2$, so we get

$$f_8(R_2) = \begin{cases} \varepsilon_1(2R_2 - 1, R_2) = 1 - \sqrt{2R_2} & : \quad R_2 \geq 1/2 \\ 0 & : \quad R_2 < 1/2 \end{cases}$$

The critical points again need two cases. If $R_2 \geq M/2 + 1/2$ the critical point is $M = \frac{2R_2-1}{3}$. This only occurs if $R_2 \geq \frac{1}{2}$. Similarly, for $R_2 < \frac{M+1}{2}$ we obtain the critical point $M = \frac{2R_2+1}{3}$, which requires $R_2 < \frac{1}{2}$. So we define both at once with

$$f_9(R_2) = \begin{cases} \varepsilon_1(\frac{2R_2-1}{3}, R_2) = 1 - (\frac{2}{3}(1+R_2))^{3/2} & : \quad R_2 \geq 1/2 \\ \varepsilon_1(\frac{2R_2+1}{3}, R_2) = 1 - \sqrt{2}(\frac{2+R_2}{3})^{3/2} & : \quad R_2 < 1/2 \end{cases}$$

Note all $f_i$ are continuous except $f_9$.

4.3.3. *Combining Errors.* So, for a given $R_2$, the above 9 functions give 9 values, and the max of the absolute values is the max error for this $R_2$. So we now vary $R_2 \in I$ to find the best one.

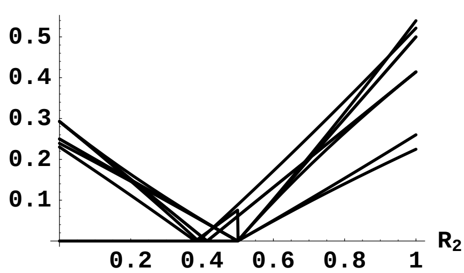


Figure 3

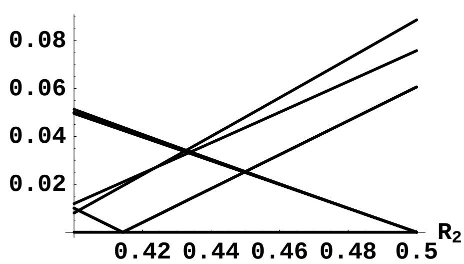


Figure 4

Figure 3 is a Mathematica [8] plot of the error functions $|f_i(R_2)|$. The max of these is lowest near $R_2 = 0.45$; Figure 4 is a closer look. The numerical solver `FindMinimum` in Mathematica finds the common value of $R_2$ that minimizes $\max_i |f_i|$, returning

$$r_0 = 0.432744889959443195468521587014$$

where this many digits are retained since this constant can be applied to any bitlength of floating point number: 64 bit doubles, 128 bit future formats, etc. See the conclusion section. Note $r_0$ is very close to the original value in the code which is approximately 0.432430148. $r_0$ corresponds to the integer $R_2 = \lfloor 2^{23} r_0 + 0.5 \rfloor =$ `0x37642f`. Attaching $R_1 =$ `0xbe`, shifted to `0x5f`, the optimal constant becomes $R =$ `0x5f37642f`.

## 5. Testing

We test the analysis using the following function that only computes the *linear* approximation step (the Newton step is removed!).

```
float InvSqrtLinear(float x, int magic)
      {
      float xhalf = 0.5f*x;
      int i = *(int*)&x;       // get bits for floating value
      i = magic - (i>>1);      // gives initial guess y0
      x = *(float*)&i;         // convert bits back to float
      return x;
      }
```

We apply the function to the initial constant, the constant derived above, and the constant `0x5f375a86` (the reason will be explained below). Also, we test each constant with the Newton step performed 1 and 2 iterations (the added time to run the second Newton step was very small, yet the accuracy was greatly increased). Each test is over all floating point values. The maximal relative error percentages are

| Value | Predicted | Tested | 1 Iteration | 2 Iterations |
|---|---|---|---|---|
| `0x5f3759df` | 3.43758 | 3.43756 | 0.175228 | 4.66e-004 |
| `0x5f37642f` | 3.42128 | 3.42128 | 0.177585 | 4.77521e-004 |
| `0x5f375a86` | 3.43655 | 3.43652 | 0.175124 | 4.65437e-004 |

So the analysis was correct in predicting that the new constant would approximate better in practice. Yet surprisingly, after one Newton iteration, it has a higher maximal relative error. Which again raises the question: how was the original code constant derived?

The reason the better approximation turned out worse must be in the Newton iteration. The new constant is clearly better in theory and practice than the original for initial approximation, but after 1 or 2 Newton steps the original constant performed better. Out of curiosity, I searched numerically for a better constant. The results of the theory implied any good approximations should be near those few above, thus limiting the range of the search.

Starting at the initial constant, and testing all constants above and below until the maximal relative error exceeds 0.00176 gives the third constant `0x5f375a86` as the best one; each was tested over all floating point values. The table shows it has a smaller maximal relative error than the original one. So the final version I propose is

```
float InvSqrt(float x)
      {
      float xhalf = 0.5f*x;
      int i = *(int*)&x;         // get bits for floating value
      i = 0x5f375a86- (i>>1);  // gives initial guess y0
      x = *(float*)&i;           // convert bits back to float
      x = x*(1.5f-xhalf*x*x);  // Newton step, repeating increases accuracy
      return x;
      }
```

Thus the initial goal 2) is reached, although not as directly as planned. Goal 3) remains open :)

The C++ code and Mathematica 4.0 code are available online [9].

## 6. Conclusion and Open Questions

This note explains the “magic” behind the constant in the code, and attempts to understand it in order to improve it if possible. The new constant `0x5f375a86` appears to perform slightly better than the original one. Since both are approximations, either works well in practice. I would like to find the original author if possible, and see if the method was derived or just guessed and tested.

The utility of this note is explaining how to go about deriving such methods for other functions and platforms. With the reasoning methods above, almost any such method can be analyzed. However the analysis should be thoroughly tested on a real machine, since hardware often does not play well with theory.

The above derivations are almost size neutral, so can be applied to 64 bit `double`, or even longer packed types to mimic SIMD instructions without hardware support. The analysis allows this method to be extended to many platforms and functions.

For example, the `double` type has an 11 bit exponent, biased by 1023, and a 52 bit mantissa. After deriving $y_0$ with the same form for the mantissa as above, only the bound on $R_1$ needs reworked, and the rest of the analysis will still hold. The same constant $r_0$ can used for any bit size in this manner. Here $R_1 = 1534$, and the best initial approximation is $R = R_1 2^{52} + \lfloor r_0 * 2^{52} \rfloor =$ `0x5fe6ec85e7de30da`. A quick test shows the relative error to be around 0.0342128 for the initial approximation, and 0.0017758 after 1 Newton iteration.

A final question is to analyze why the best initial approximation to the answer is not the best after one Newton iteration.

6.1. **Homework:** A few problems to work on:

1. Derive a similar method for `sqrt(x)`
2. Which is better (faster? more accurate?): a version that works for `double` or 2 Newton steps?
3. Confirm the best initial approximation for 64 bit IEEE 754 size type `double`.

## References

[1] www.gamedev.net/community/forums/topic.asp?topic_id=139956
[2] www.msdn.microsoft.com/visualc/
[3] `comp.graphics.algorithms`, Search google with "0x5f3759df group:comp.graphics.algorithms"
[4] David Eberly, www.magic-software.com/Documentation/FastInverseSqrt.pdf
[5] crystal.sourceforge.net/
[6] talika.eii.us.es/ titan/titan/index.html
[7] IEEE, grouper.ieee.org/groups/754/
[8] Mathematica - www.mathematica.com
[9] Chris Lomont, www.math.purdue.edu/∼ clomont, look under math, then under papers

Department of Mathematics, 150 North University Street, Purdue University, West Lafayette, Indiana 47907-2067
*Email address*: clomont@math.purdue.edu
*Web address*: www.math.purdue.edu/∼ clomont
First written: Feb 2003