A Brief History of InvSqrt

Matthew Robertson

University of New Brunswick

April 2, 2012

Some unknown master bit hacker has released the following amazing function into the public domain. This function, most commonly called *InvSqrt*, approximates the 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*.

q_rsqrt.c

```c
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;
}
```

This raises many questions:

- Who wrote it?
- Why is it needed?
- How accurate is it?
- How does it work?
- Is it still useful today?
- Can it be improved?

This raises many questions:

- Who wrote it?
- Why is it needed?
- How accurate is it?
- How does it work?
- Is it still useful today?
- Can it be improved?

This raises many questions:

- Who wrote it?
- Why is it needed?
- How accurate is it?
- How does it work?
- Is it still useful today?
- Can it be improved?

This raises many questions:

- Who wrote it?
- Why is it needed?
- How accurate is it?
- How does it work?
- Is it still useful today?
- Can it be improved?

This raises many questions:

- Who wrote it?
- Why is it needed?
- How accurate is it?
- How does it work?
- Is it still useful today?
- Can it be improved?

This raises many questions:

- Who wrote it?
- Why is it needed?
- How accurate is it?
- How does it work?
- Is it still useful today?
- Can it be improved?

Who wrote it?

2005 Quake III: Arena - Source code released under GPL

- "One of the more famous snippets of graphics code in recent years."

2002 Newsgroup - comp.graphics.algorithms

- Earliest known appearance in current form.

1999 Quake III: Arena - Leaked by ATI (March)

- Function in use without source.

1997 Floating-Point Tricks

- More general, less refined function for x^n.

Nobody knows...

2005 Quake III: Arena - Source code released under GPL

- "One of the more famous snippets of graphics code in recent years."

2002 Newsgroup - comp.graphics.algorithms

- Earliest known appearance in current form.

1999 Quake III: Arena - Leaked by ATI (March)

- Function in use without source.

1997 Floating-Point Tricks

- More general, less refined function for x^n.

Nobody knows...

2005 Quake III: Arena - Source code released under GPL

- "One of the more famous snippets of graphics code in recent years."

2002 Newsgroup - comp.graphics.algorithms

- Earliest known appearance in current form.

1999 Quake III: Arena - Leaked by ATI (March)

- Function in use without source.

1997 Floating-Point Tricks

- More general, less refined function for x^n.

Nobody knows...

2005 Quake III: Arena - Source code released under GPL

- "One of the more famous snippets of graphics code in recent years."

2002 Newsgroup - comp.graphics.algorithms

- Earliest known appearance in current form.

1999 Quake III: Arena - Leaked by ATI (March)

- Function in use without source.

1997 Floating-Point Tricks

- More general, less refined function for x^n.

Nobody knows...

2005 Quake III: Arena - Source code released under GPL

- "One of the more famous snippets of graphics code in recent years."

2002 Newsgroup - comp.graphics.algorithms

- Earliest known appearance in current form.

1999 Quake III: Arena - Leaked by ATI (March)

- Function in use without source.

1997 Floating-Point Tricks

- More general, less refined function for x^n.

Nobody knows...

Why is it needed?

A common real time calculation in games is 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 is a need to calculate the reciprocal square root of a number quickly.

A common real time calculation in games is to *normalize* a vector, or calculate its *unit vector* $\hat{v}$ by

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

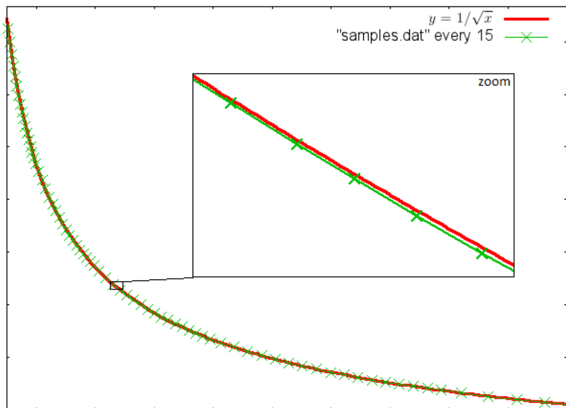
where $||\vec{v}||$ is the *Euclidean norm* of $\vec{v}$

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

So clearly there is a need to calculate the reciprocal square root of a number quickly.

How accurate is it?

Graph of y $= 1/\sqrt{x}$ along samples taken from $\mathbf{Q_rsqrt}()$



Maximum relative error: $\mathbf{0.0017522874}$

How does it work?

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

IEEE 754 Floating Point Representation

$w =$

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

where w is a *word* (or *integer*) in memory.

These fields can be interpreted as integers such that

- s is the sign bit, 1 means negative
- E is the exponent field, bias $b = 127$
- M is the fractional part of the normalized mantissa

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

IEEE 754 Floating Point Representation

$w =$

s	E	M
$bit\,31$	$30 \leftarrow bits \rightarrow 23$	$22 \leftarrow bits \rightarrow 0$

where w is a *word* (or *integer*) in memory.

These fields can be interpreted as integers such that

- s is the sign bit, 1 means negative
- E is the exponent field, bias $b = 127$
- M is the fractional part of the normalized mantissa

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

IEEE 754 Floating Point Representation

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

where w is a *word* (or *integer*) in memory.
These fields can be interpreted as integers such that

- s is the sign bit, 1 means negative
- E is the exponent field, bias $b = 127$
- M is the fractional part of the normalized mantissa

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

IEEE 754 Floating Point Representation

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

where w is a *word* (or *integer*) in memory.
These fields can be interpreted as integers such that

- s is the sign bit, 1 means negative
- E is the exponent field, bias $b = 127$
- M is the fractional part of the normalized mantissa

IEEE 754 Floating Point Representation

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

So the *float* $\phi(w)$ is given by

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

For convenience, this is often expressed as

$$\phi(w) = (-1)^s (1 + m) 2^{E-b}$$

where $m \in [0, 1)$

IEEE 754 Floating Point Representation

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

So the *float* $\phi(w)$ is given by

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

For convenience, this is often expressed as

$$\phi(w) = (-1)^s \,(1 + m)\, 2^{E-b}$$

where $m \in [0, 1)$

The *Newton-Raphson* method takes a current approximation, or guess, of the root of a function and returns some (hopefully) better approximation as follows

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

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

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:

- casting between $(\mathbf{long}*)$ and $(\mathbf{float}*)$
- the "magic" line $\mathbf{i} = \mathbf{0x5f3759df} - (\mathbf{i} >> \mathbf{1})$
- the formula $\mathbf{y} = \mathbf{y} * (\mathbf{threehalfs} - (\mathbf{x2} * \mathbf{y} * \mathbf{y}))$

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:

- casting between $(\mathbf{long}*)$ and $(\mathbf{float}*)$
- the "magic" line $\mathbf{i} = \mathbf{0x5f3759df} - (\mathbf{i} >> \mathbf{1})$
- the formula $\mathbf{y} = \mathbf{y} * (\mathbf{threehalfs} - (\mathbf{x2} * \mathbf{y} * \mathbf{y}))$

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:

- casting between $(\mathbf{long}*)$ and $(\mathbf{float}*)$
- the "magic" line $\mathbf{i} = \mathbf{0x5f3759df} - (\mathbf{i} >> \mathbf{1})$
- the formula $\mathbf{y} = \mathbf{y} * (\mathbf{threehalfs} - (\mathbf{x2} * \mathbf{y} * \mathbf{y}))$

The function contains two reinterpreting casts between (**long**$*$) and (**float**$*$). This process basically interprets the *float* bits as an *integer* without converting it intelligently.

IEEE 754 Floating Point Representation

$w =$

s	E	M
$bit\,31$	$30 \leftarrow bits \rightarrow 23$	$22 \leftarrow bits \rightarrow 0$

So the *integer* interpretation of a *floats*' bits can be expressed as

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

The function contains two reinterpreting casts between $(\mathbf{long}*)$ and $(\mathbf{float}*)$. This process basically interprets the *float* bits as an *integer* without converting it intelligently.

IEEE 754 Floating Point Representation

$w =$

s	E	M
$bit\,31$	$30 \leftarrow bits \rightarrow 23$	$22 \leftarrow bits \rightarrow 0$

So the *integer* interpretation of a *floats*' bits can be expressed as

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

For example, consider w_π to be an approximation of π in memory:

IEEE 754 Floating Point Approximation of π

Field:	s	E	M
w_π	0	128	4788187
Range:	$bit\,31$	$30 \leftarrow bits \rightarrow 23$	$22 \leftarrow bits \rightarrow 0$

as a *float*:

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

as an *integer*:

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

For example, consider w_π to be an approximation of π in memory:

IEEE 754 Floating Point Approximation of π

Field:	s	E	M
w_π	0	128	4788187
Range:	$bit\ 31$	$30 \leftarrow bits \rightarrow 23$	$22 \leftarrow bits \rightarrow 0$

as a *float*:

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

as an *integer*:

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

For example, consider w_π to be an approximation of π in memory:

IEEE 754 Floating Point Approximation of π

Field:	s	E	M
w_π	0	128	4788187
Range:	$bit\,31$	$30 \leftarrow bits \rightarrow 23$	$22 \leftarrow bits \rightarrow 0$

as a *float*:

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

as an *integer*:

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

q_rsqrt.c

```c
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;
}
```

Normally, a bit shift is an illegal operation on a *float*. But though casting pointers, this can be done indirectly. The line of code

$$\mathbf{i = 0x5f3759df - (i >> 1);}$$

contains a bit shift on $\mathbf{i}$, the *integer* interpretation of the bits from the *float* $\mathbf{f}$. This effectively divides $\mathbf{i}$ by 2.

$j = (i >> 1)$

Field:	s	E	M
i	s_0	$E_7 E_6 ... E_1 E_0$	$M_{22} M_{21} ... M_1 M_0$
$(i >> 1)$	0	$s_0 E_7 ... E_2 E_1$	$E_0 M_{22} ... M_2 M_1$

The least significant bit M_0 will be lost. However, the E_0 bit will fall into the mantissa field; possibly adding 2^{22}. Therefore the *word* $j = (i >> 1)$ can be expressed as

$$j = \boxed{\lfloor E/2 \rfloor \;\Big|\; 2^{22} E_0 + \lfloor M/2 \rfloor}$$

$j = (i >> 1)$

Field:	s	E	M
i	s_0	$E_7E_6...E_1E_0$	$M_{22}M_{21}...M_1M_0$
$(i >> 1)$	0	$s_0E_7...E_2E_1$	$E_0M_{22}...M_2M_1$

The least significant bit M_0 will be lost. However, the E_0 bit will fall into the mantissa field; possibly adding 2^{22}. Therefore the *word* $j = (i >> 1)$ can be expressed as

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

The code previously considered also contains the *constant* R

$$R = \mathbf{0x5f3759df}$$

which breaks down into exponent and mantissa form as

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

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

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

The code previously considered also contains the *constant* R

$$R = \mathbf{0x5f3759df}$$

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. Let $G = (R - j)$ be represented as

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

The code previously considered also contains the *constant* R

$$R = \mathbf{0x5f3759df}$$

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 $\mathbf{j}$ is subtracted from it. Let $G = (R - j)$ be represented as

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

q_rsqrt.c

```c
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;
}
```

For some *float* $f = (1 + m)2^{E-127}$ the reciprocal square root of f can be expressed as

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

This can be broken down into two factors as

$$\frac{1}{\sqrt{f}} = \underbrace{\frac{\sqrt{2}^{1-E_0}}{\sqrt{1+m}}}_{\text{mantissa}} \underbrace{2^{63-\lfloor E/2 \rfloor}}_{\text{exponent}}$$

Consider only the exponent factor given by

$$2^e = 2^{63-\lfloor E/2 \rfloor}$$

Compensate for the bias

$$2^{e+127} = 2^{190-\lfloor E/2 \rfloor}$$

Now G can be partly represented exactly as

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

Consider only the exponent factor given by

$$2^e = 2^{63-\lfloor E/2 \rfloor}$$

Compensate for the bias

$$2^{e+127} = 2^{190-\lfloor E/2 \rfloor}$$

Now G can be partly represented exactly as

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

Consider only the exponent factor given by

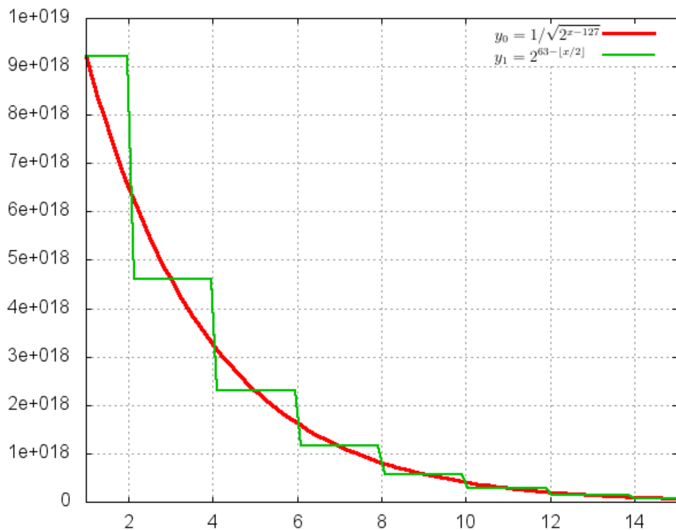
$$2^e = 2^{63 - \lfloor E/2 \rfloor}$$

Compensate for the bias

$$2^{e+127} = 2^{190 - \lfloor E/2 \rfloor}$$

Now G can be partly represented exactly as

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


1e+019
9e+018
8e+018
7e+018
6e+018
5e+018
4e+018
3e+018
2e+018
1e+018
0
2
4
6
8
10
12
14
$y_0 = 1/\sqrt{2^{x-127}}$
$y_1 = 2^{63-\lfloor x/2 \rfloor}$

Consider only the mantissa factor given by

$$Z_{E_0} = \frac{\sqrt{2}^{1-E_0}}{\sqrt{1+m}}$$

Whether E is even or odd affects the mantissa, also there is the possibility of underflow which can cause the mantissa to borrow a bit from the exponent. So there are different cases to consider.

Consider only the mantissa factor given by

$$Z_{E_0} = \frac{\sqrt{2}^{1-E_0}}{\sqrt{1+m}}$$

Whether E is even or odd affects the mantissa, also there is the possibility of underflow which can cause the mantissa to borrow a bit from the exponent. So there are different cases to consider.

Consider the case that E is even, meaning $E_0 = 0$

$$Z_0 = \frac{\sqrt{2}}{\sqrt{1+m}}$$

This means no bit falls into the mantissa field of

$$\mathbf{j} = (\mathbf{i} >> \mathbf{1})$$

However, there is the possibility of underflow when $m/2 > t$ where $t = T/2^{23} \in [0, 1)$

Consider the case that E is even, meaning $E_0 = 0$

$$Z_0 = \frac{\sqrt{2}}{\sqrt{1+m}}$$

This means no bit falls into the mantissa field of

$$\mathbf{j} = (\mathbf{i} >> \mathbf{1})$$

However, there is the possibility of underflow when $m/2 > t$ where $t = T/2^{23} \in [0, 1)$

Consider the case that E is even, meaning $E_0 = 0$

$$Z_0 = \frac{\sqrt{2}}{\sqrt{1+m}}$$

This means no bit falls into the mantissa field of

$$\mathbf{j} = (\mathbf{i} >> \mathbf{1})$$

However, there is the possibility of underflow when $m/2 > t$ where $t = T/2^{23} \in [0, 1)$

When m is small, such that $m/2 \leq t$, no underflow occurs so no bit is borrowed from the exponent field. Thus G is simply

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

or as a *float* this is interpreted as

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

The *mantissa* factor can be represented by the real line

$$y_1(x) = 1 + t - \frac{x}{2}\ , \ x \in [0, 2t]$$

When m is small, such that $m/2 \leq t$, no underflow occurs so no bit is borrowed from the exponent field. Thus G is simply

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

or as a *float* this is interpreted as

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

The *mantissa* factor can be represented by the real line

$$y_1(x) = 1 + t - \frac{x}{2}\ , \ x \in [0, 2t]$$

When m is small, such that $m/2 \leq t$, no underflow occurs so no bit is borrowed from the exponent field. Thus G is simply

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

or as a *float* this is interpreted as

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

The *mantissa* factor can be represented by the real line

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

When m is large, such that $m/2 > t$, 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

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

or as a *float*, this is interpreted as

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

When m is large, such that $m/2 > t$, 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

$$G = \boxed{\;190 - 1 - \lfloor E/2 \rfloor \;\Big|\; 2^{23} + T - \lfloor M/2 \rfloor\;}$$

or as a *float*, this is interpreted as

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

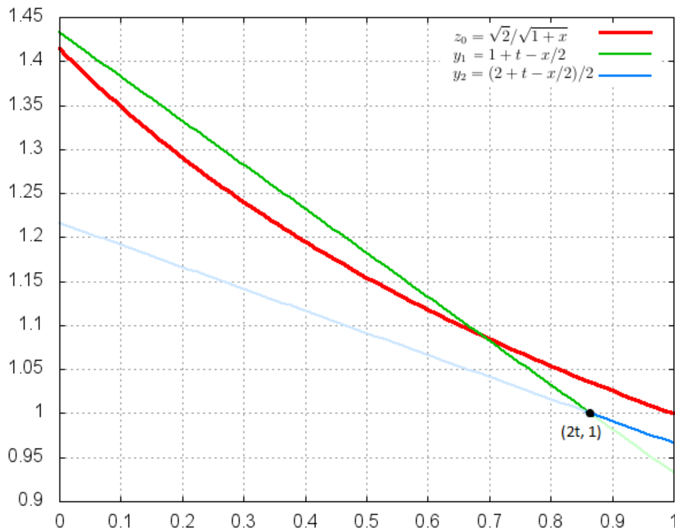
The borrowing a bit from the exponent field effectively cuts G in half. Because the exponent field of G is already fixed, divide the mantissa factor in half to compensate, giving the real line

$$y_2(x) = 1 + \frac{t}{2} - \frac{x}{4}$$

where $x \in (2t, 1)$

Interpretation

The Magic Constant - Mantissa - E is Even - Graph



Consider the case that E is odd, meaning $E_0 = 1$.

$$Z_1 = \frac{1}{\sqrt{1+m}}$$

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

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

Consider the case that E is odd, meaning $E_0 = 1$.

$$Z_1 = \frac{1}{\sqrt{1+m}}$$

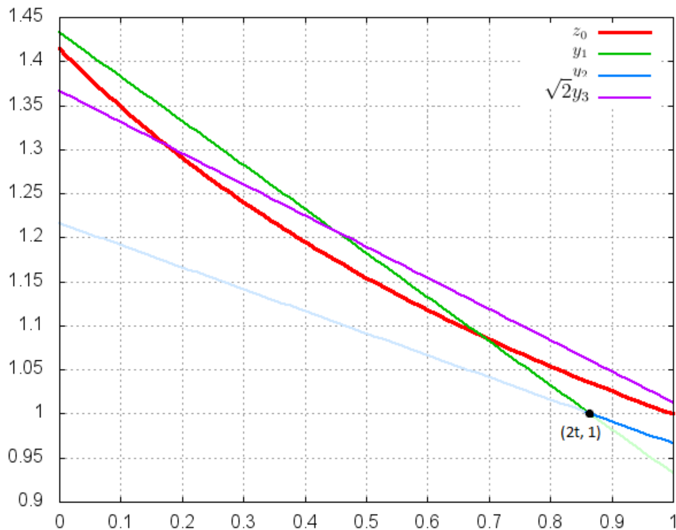
This means a 1 bit falls into the mantissa field of j. There is guaranteed underflow in the subtraction $G = (R - j)$ because R is less then the smallest possible value for j, 2^{22}. So a bit is borrowed from the exponent field and G can be represented as

$$G = \boxed{\ 190 - 1 - \lfloor E/2 \rfloor \ \big|\ 2^{23} + T - \left(2^{22} + \lfloor M/2 \rfloor\right)\ }$$

Similar to y_2, a bit is borrowed from the exponent field effectively diving G by 2. So

$$y_3(x) = \frac{3}{4} + \frac{t}{2} - \frac{x}{4}$$

where $x \in [0, 1)$


1.45
1.4
1.35
1.3
1.25
1.2
1.15
1.1
1.05
1
0.95
0.9
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
z_0
y_1
y_2
$\sqrt{2}y_3$
(2t, 1)

The particular T value of this particular magic constant is what makes the function work so well.

There has been a history of slightly flawed interpretations in the past. One major difference is this interpretation showed there are only three valid cases to consider, whereas the other papers consider a fourth, impossible case; or represented the cases differently.

Because there are only a finite number of positive *floats*, it is possible to test them all by asserting

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

For the C function $\mathrm{G}()$, representing precisely the cases previously given as G, this confirms for every *integer* $\mathbf{i}$, corresponding to every positive *float* f.

See appendix A.3 of thesis for C code.

Because there are only a finite number of positive *floats*, it is possible to test them all by asserting

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

For the C function $\mathbf{G}()$, representing precisely the cases previously given as G, this confirms for every *integer* $\mathbf{i}$, corresponding to every positive *float* f.

See appendix A.3 of thesis for C code.

q_rsqrt.c

```c
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;
}
```

Consider the *Newton-Raphson* method on

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

which yields an x_{n+1} of

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

which can be rearranged to more closely resemble the code as

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

where x is $\mathbf{y}$ and $h/2$ is $\mathbf{x2}$.

Consider the *Newton-Raphson* method on

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

which yields an x_{n+1} of

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

which can be rearranged to more closely resemble the code as

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

where x is $\mathbf{y}$ and $h/2$ is $\mathbf{x2}$.

Consider the *Newton-Raphson* method on

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

which yields an x_{n+1} of

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

which can be rearranged to more closely resemble the code as

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

where x is $\mathbf{y}$ and $h/2$ is $\mathbf{x2}$.

q_rsqrt.c

```c
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;
}
```

The function works by pointer casting, bit shifting a float, and the *Newton-Raphson* method.

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 become a very good approximation of the reciprocal square root of a floating point number.

The function works by pointer casting, bit shifting a float, and the *Newton-Raphson* method.

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 become a very good approximation of the reciprocal square root of a floating point number.

The function works by pointer casting, bit shifting a float, and the *Newton-Raphson* method.

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 become a very good approximation of the reciprocal square root of a floating point number.

Is it still useful today?

In 2003, the function was considered to be about four times faster than the native *libm* $\mathbf{1.0/sqrt(f)}$. A test on a few different CPUs can demonstrate this.

The following table shows the ratio of time to compute $\mathbf{1.0f/sqrtf}()$ over $\mathbf{Q_rsqrt}()$ across all positive floats on different CPUs:

CPU	Minimum	Average	Maximum
AMD V120 Processor	3.011129	3.014891	3.017516
AMD Sempron 3200+	3.085374	3.096781	3.100399
Intel Core2 Duo E8400 3.0Hz	3.348315	3.352887	3.358521
Intel Core i5-2415M 2.3GHz	3.960945	4.033596	4.079792
Intel Core i5-430M 2.27GHz	4.087017	4.098945	4.126741
Intel Core i7-2640M 2.8GHz	4.081911	4.099887	4.111876

On modern desktop computers it appears to be between three and four times faster. This means the function is still practical today!

Can it be improved?

It is important to generalize this interpretation to include any magic constant $R^* = \boxed{S \mid T}$ and work for any sized *floating point representation*. So consider the conditional equation

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

where U is the size of the mantissa field.

It is important to generalize this interpretation to include any magic constant $R^* = \boxed{S} \boxed{T}$ and work for any sized *floating point representation*. So consider the conditional equation

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

where U is the size of the mantissa field.

The exponent field of R is trivial to find. Similar to the method shown before, S can be expressed as

$$\begin{aligned} S &= \lfloor b/2 \rfloor + b \\ &= \lfloor 3b/2 \rfloor \end{aligned}$$

where b is the bias.

The exponent field of R is trivial to find. Similar to the method shown before, S can be expressed as

$$\begin{aligned} S &= \lfloor b/2 \rfloor + b \\ &= \lfloor 3b/2 \rfloor \end{aligned}$$

where b is the bias.

The exponent field of R is trivial to find. Similar to the method shown before, S can be expressed as

$$\begin{aligned} S &= \lfloor b/2 \rfloor + b \\ &= \lfloor 3b/2 \rfloor \end{aligned}$$

where b is the bias.

The formula for the mantissa fraction of G can be expressed as the conditional equation

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

where $x = m \in [0, 1)$
Solving $y(x)$ for t will yield a new *magic* constant.

The formula for the mantissa fraction of G can be expressed as the conditional equation

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

where $x = m \in [0, 1)$

Solving $y(x)$ for t will yield a new *magic* constant.

The formula for the mantissa fraction of G can be expressed as the conditional equation

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

where $x = m \in [0, 1)$
Solving $y(x)$ for t will yield a new *magic* constant.

A good magic constant would be one that minimizes the maximum relative error of $y(x)$. Consider the "signed" relative errors

$$w(x) = \frac{y(x)}{Z_0(x)} - 1$$

Notice $|w(x)|$ is piecewise differentiable on the previously defined domains of $y(x)$.

A good magic constant would be one that minimizes the maximum relative error of $y(x)$. Consider the "signed" relative errors

$$w(x) = \frac{y(x)}{Z_0(x)} - 1$$

Notice $|w(x)|$ is piecewise differentiable on the previously defined domains of $y(x)$.

The maximum relative error must occur at one of the following critical or end points

$$\underbrace{x = 0}_{\text{end point}} \qquad \underbrace{x = \frac{2t}{3}}_{\text{maximum}} \qquad \underbrace{x = \frac{2(t+1)}{3}}_{\text{minimum}}$$

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

The maximum relative error must occur at one of the following critical or end points

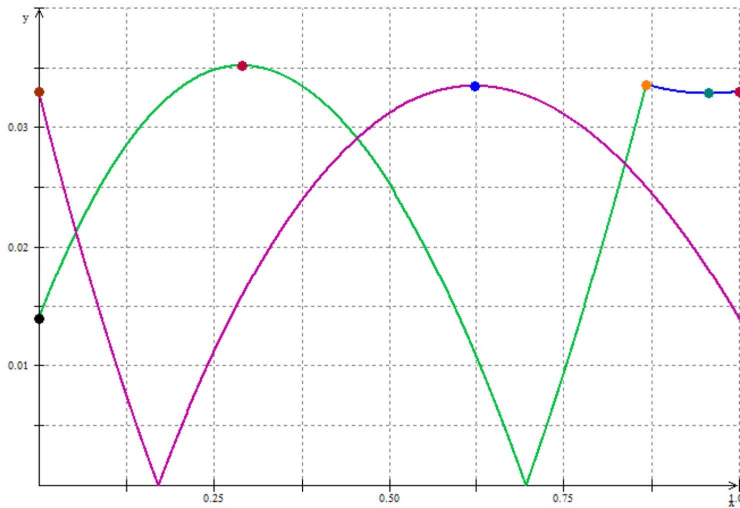
$$\underbrace{x = 0}_{\text{end point}} \qquad \underbrace{x = \frac{2t}{3}}_{\text{maximum}} \qquad \underbrace{x = \frac{2(t+1)}{3}}_{\text{minimum}}$$

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

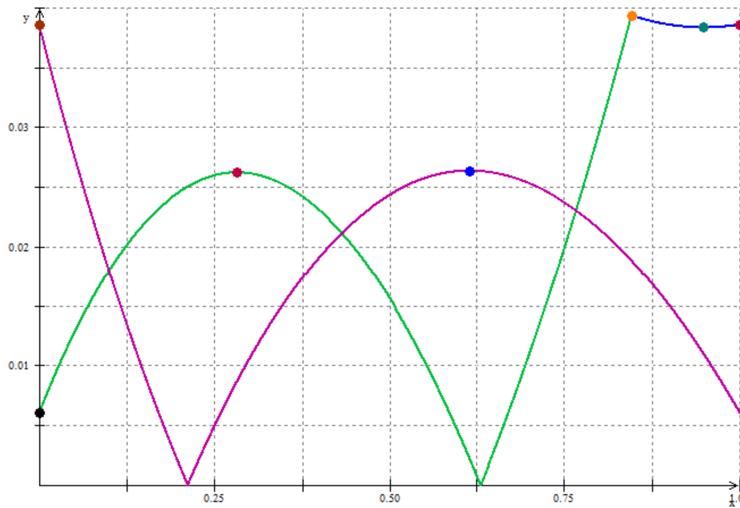
The maximum relative error must occur at one of the following critical or end points

$$\underbrace{x = 0}_{\text{end point}} \qquad \underbrace{x = \frac{2t}{3}}_{\text{maximum}} \qquad \underbrace{x = \frac{2(t+1)}{3}}_{\text{minimum}}$$

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

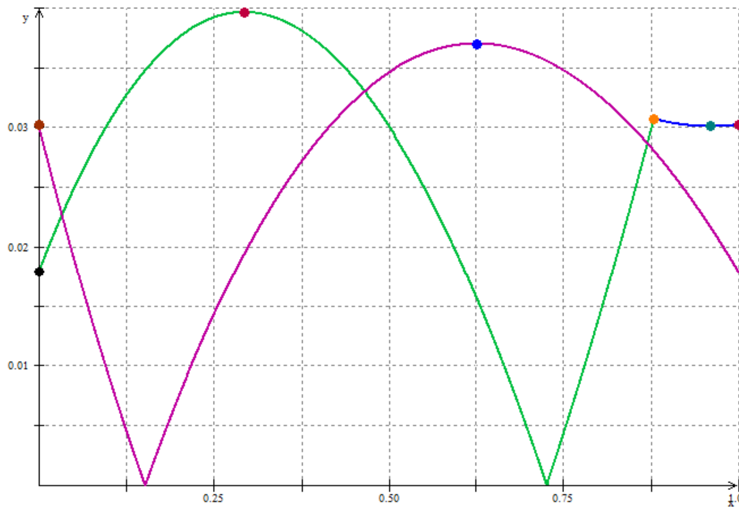


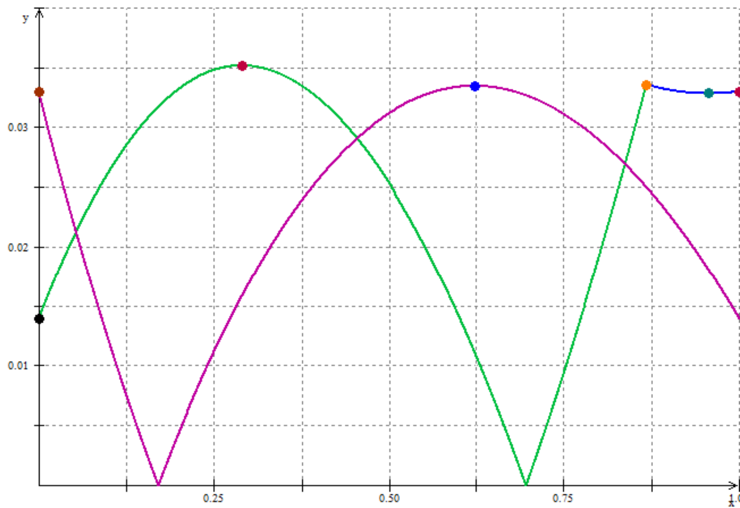
y
0.03
0.02
0.01
0.25
0.50
0.75
x


y
0.03
0.02
0.01
0.25
0.50
0.75
x

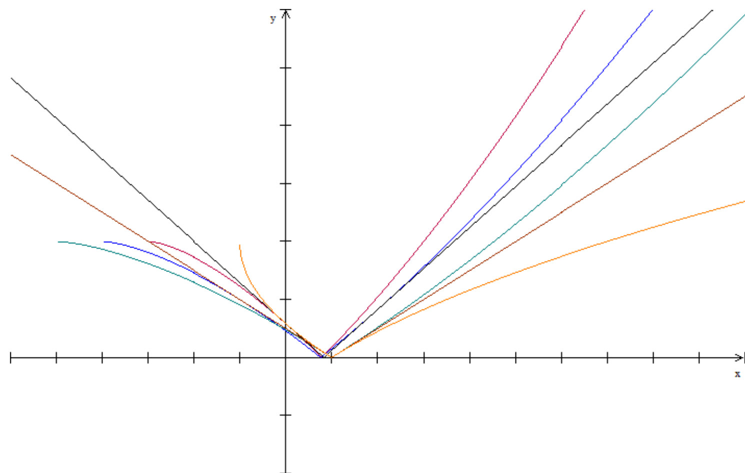
Analysis

Mantissa Field - Minimize Error - Graph

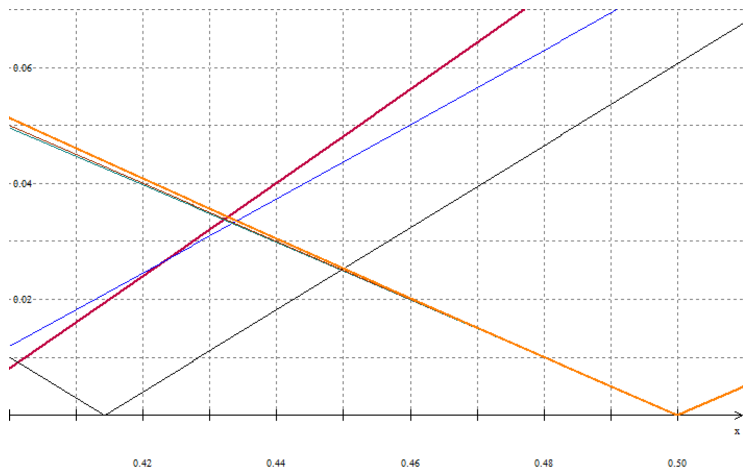


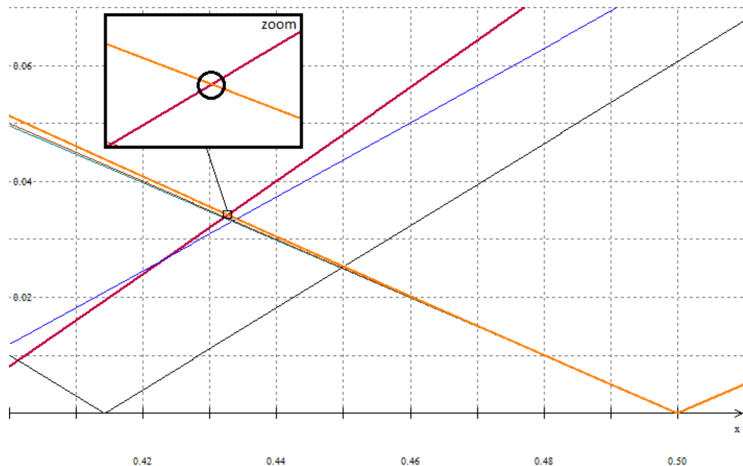


y
0.03
0.02
0.01
0.25
0.50
0.75
x



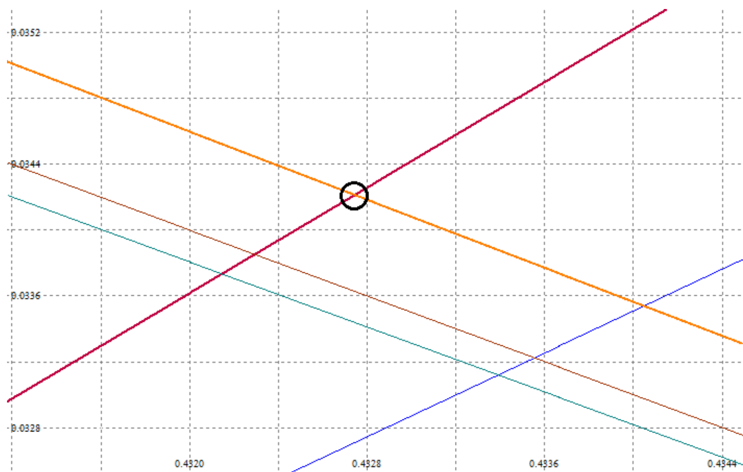
y
x


0.06
0.04
0.02
x
0.42
0.44
0.46
0.48
0.50


zoom
0.06
0.04
0.02
x
0.42
0.44
0.46
0.48
0.50

Analysis

Mantissa Field - t Values - Graph



It is clear that the optimal value of t is bound only by the top two lines of the graph. Therefore the t which gives the minimal, maximum relative error is the intercept between the two lines

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

which simplifies to be

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

which yields a t of about

$$t \doteq 0.43274488995944319546852158699601037361978241\ldots$$

It is clear that the optimal value of t is bound only by the top two lines of the graph. Therefore the t which gives the minimal, maximum relative error is the intercept between the two lines

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

which simplifies to be

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

which yields a t of about

$$t \doteq 0.432744889959443195468521586996010373619782 41\ldots$$

It is clear that the optimal value of t is bound only by the top two lines of the graph. Therefore the t which gives the minimal, maximum relative error is the intercept between the two lines

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

which simplifies to be

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

which yields a t of about

$$t \doteq 0.4327448899594431954685215869960103736197824 1\ldots$$

This t differs from the t used in the original R, it also differs from the t found by Chris Lomont.

The difference is only after the 26th digit, so its affect will not be seen until *quadruple precision floating point*.

Lomont used a numerical method to compute it, whereas this thesis used an algebraic method to calculate it.

This t differs from the t used in the original R, it also differs from the t found by Chris Lomont.

The difference is only after the 26th digit, so its affect will not be seen until *quadruple precision floating point*.

Lomont used a numerical method to compute it, whereas this thesis used an algebraic method to calculate it.

This t differs from the t used in the original R, it also differs from the t found by Chris Lomont.
The difference is only after the 26th digit, so its affect will not be seen until *quadruple precision floating point*.
Lomont used a numerical method to compute it, whereas this thesis used an algebraic method to calculate it.

The calculated t value corresponds to a R value of $\mathbf{0x5f37642f}$. However, this magic constant yields a maximum relative error of 0.0017758484, which is actually worse then the original.

The new constant will yield a better result before the Newton iteration at 0.0342128389.

Why is the best initial approximation not the best approximation after one Newton iteration? Lomont could not explain this.

The calculated t value corresponds to a R value of $\mathbf{0x5f37642f}$. However, this magic constant yields a maximum relative error of 0.0017758484, which is actually worse then the original.

The new constant will yield a better result before the Newton iteration at 0.0342128389.

Why is the best initial approximation not the best approximation after one Newton iteration? Lomont could not explain this.

The calculated t value corresponds to a R value of $\mathbf{0x5f37642f}$. However, this magic constant yields a maximum relative error of 0.0017758484, which is actually worse then the original.
The new constant will yield a better result before the Newton iteration at 0.0342128389.
Why is the best initial approximation not the best approximation after one Newton iteration? Lomont could not explain this.

The issue is a result of finding a constant that will give the closest initial guess to the real answer, but on either side.

Because of the shape of the graph of $y = 1/\sqrt{x}$, after an iteration of the *Newton-Raphson* method from any close guess x_n, it will always result in a x_{n+1} that is less than the answer.

This can be demonstrated mathematically by considering the initial guess $x_n = (1/\sqrt{h} - \epsilon)$

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

The issue is a result of finding a constant that will give the closest initial guess to the real answer, but on either side.
Because of the shape of the graph of $y = 1/\sqrt{x}$, after an iteration of the *Newton-Raphson* method from any close guess x_n, it will always result in a x_{n+1} that is less than the answer.
This can be demonstrated mathematically by considering the initial guess $x_n = (1/\sqrt{h} - \epsilon)$

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

The issue is a result of finding a constant that will give the closest initial guess to the real answer, but on either side.
Because of the shape of the graph of $y = 1/\sqrt{x}$, after an iteration of the *Newton-Raphson* method from any close guess x_n, it will always result in a x_{n+1} that is less than the answer.
This can be demonstrated mathematically by considering the initial guess $x_n = (1/\sqrt{h} - \epsilon)$

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

The issue is a result of finding a constant that will give the closest initial guess to the real answer, but on either side.
Because of the shape of the graph of $y = 1/\sqrt{x}$, after an iteration of the *Newton-Raphson* method from any close guess x_n, it will always result in a x_{n+1} that is less than the answer.
This can be demonstrated mathematically by considering the initial guess $x_n = (1/\sqrt{h} - \epsilon)$

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

This can be rearranged to a more useful form as

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

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 can be rearranged to a more useful form as

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

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.

The solution to this problem is to apply the optimization after the Newton iteration.

Consider $y(x)$ 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 $1/\sqrt{x}$.

The solution to this problem is to apply the optimization after the Newton iteration.
Consider $y(x)$ 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

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

where $x = m + 1 \in [1, 2)$.

Because the approximation works for all valid exponents, it works for the exponents $E = 126$ and $E = 127$ specifically, that makes the exponent field of $1/\sqrt{f}$ equal 1.

Therefore $q(x)$ can be considered an approximation for $1/\sqrt{x}$ without worrying about the mantissa or exponent fields.

Now the iteration step can be applied where $x_n = q(x)$ to give

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

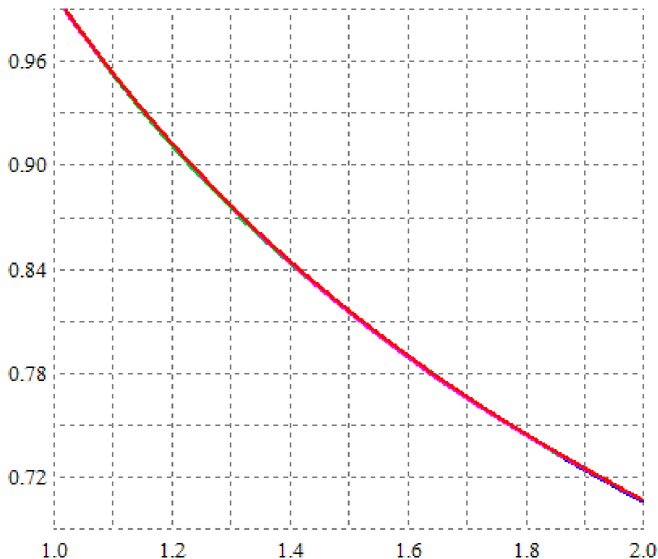
Because the approximation works for all valid exponents, it works for the exponents $E = 126$ and $E = 127$ specifically, that makes the exponent field of $1/\sqrt{f}$ equal 1.
Therefore $q(x)$ can be considered an approximation for $1/\sqrt{x}$ without worrying about the mantissa or exponent fields.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

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

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

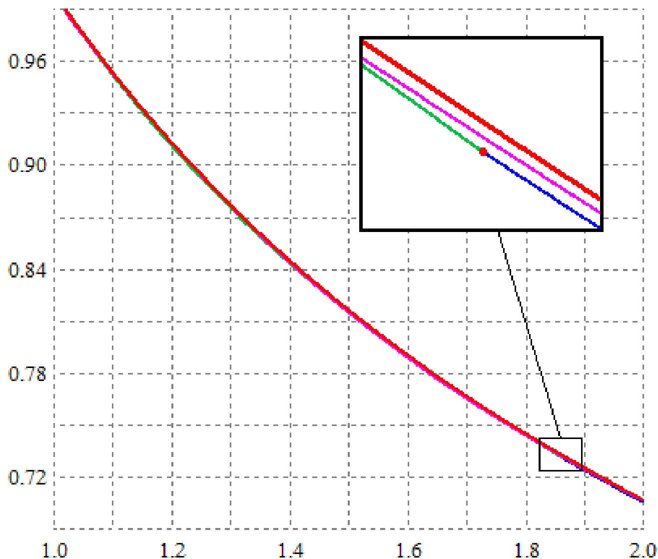
$$p(x) = q(x)\left(\frac{3}{2} - \frac{x}{2}q^2(x)\right)$$

Analysis
Mantissa Field - t Values - Newton Issue - Solution - Graph



Analysis

Mantissa Field - t Values - Newton Issue - Solution - Graph



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

$$v(x) = \frac{p(x)}{1/\sqrt{x}} - 1.$$

The maximum relative error must occur at one of the following critical or end points

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

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

The maximum relative error must occur at one of the following critical or end points

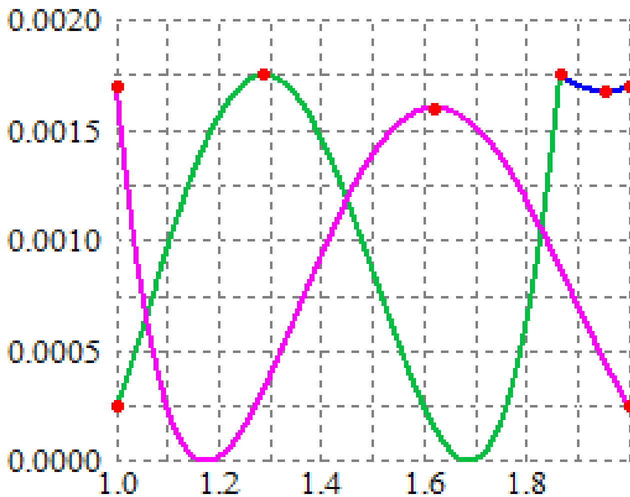
$$\underbrace{x = 1}_{\text{end point}} \quad \underbrace{x = \frac{2t+3}{3}}_{\text{maximum}} \quad \underbrace{x = \frac{2t+5}{3}}_{\text{minimum}}$$

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

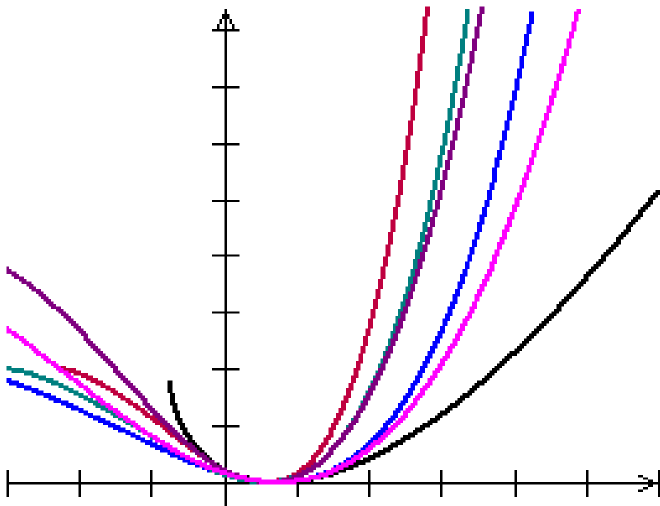
The maximum relative error must occur at one of the following critical or end points

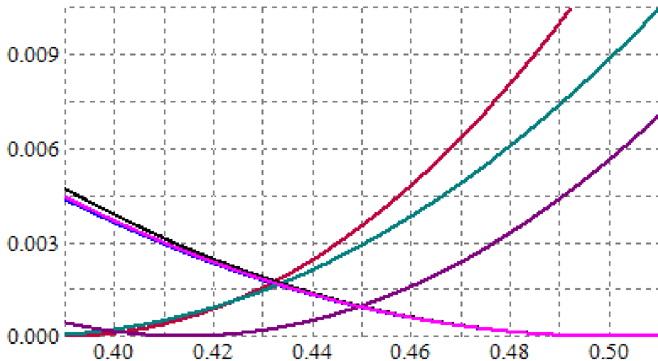
$$\underbrace{x = 1}_{\text{end point}} \quad \underbrace{x = \frac{2t+3}{3}}_{\text{maximum}} \quad \underbrace{x = \frac{2t+5}{3}}_{\text{minimum}}$$

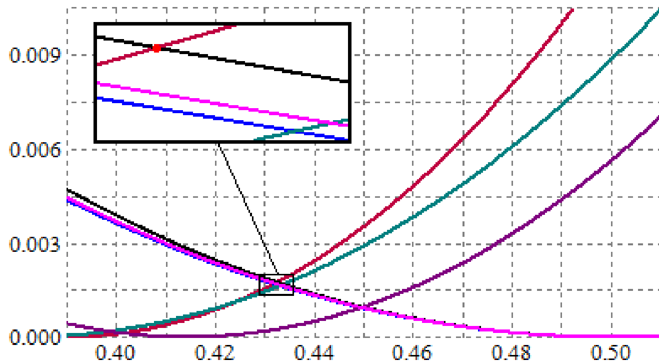
$$\underbrace{x = \frac{2t+4}{3}}_{\text{maximum}} \quad \underbrace{x = 2t+1}_{\text{end point}} \quad \underbrace{x = 2^-}_{\text{end point}}$$



0.0020
0.0015
0.0010
0.0005
0.0000
1.0
1.2
1.4
1.6
1.8




0.009
0.006
0.003
0.000
0.40
0.42
0.44
0.46
0.48
0.50


0.009
0.006
0.003
0.000
0.40
0.42
0.44
0.46
0.48
0.50

The optimal t value is bounded by the top two lines. Therefore the t which gives the minimal maximum relative error is the interception

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

which simplifies to be

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

which yields an optimal t of about

$$t_0 \doteq 0.43245008479014264217878293749679646686 13577\ldots$$

The optimal t value is bounded by the top two lines. Therefore the t which gives the minimal maximum relative error is the interception

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

which simplifies to be

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

which yields an optimal t of about

$$t_0 \doteq 0.43245008479014264217878293749679646686613577\ldots$$

The optimal t value is bounded by the top two lines. Therefore the t which gives the minimal maximum relative error is the interception

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

which simplifies to be

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

which yields an optimal t of about

$$t_0 \doteq 0.43245008479014264217878293749679646686135 77\ldots$$

The accuracy of this method does not depend of the size of the floating point representation.

The maximum relative error at the optimal t value is about

$$0.0017511836712202133521251742467001545367542482 96\ldots$$

The accuracy of this method does not depend of the size of the floating point representation.
The maximum relative error at the optimal t value is about

$$0.0017511836712202133521251742467001545367542482 96\ldots$$

This optimal t value corresponds to an R value of $\mathbf{0x5f375a86}$.

This magic constant 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.

This constant is better than the original for both the initial guess and after the iteration step!

This optimal t value corresponds to an R value of $\mathbf{0x5f375a86}$. This magic constant 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.

This constant is better than the original for both the initial guess and after the iteration step!

This optimal t value corresponds to an R value of $\mathbf{0x5f375a86}$.
This magic constant 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.
This constant is better than the original for both the initial guess and after the iteration step!

This optimal t value corresponds to an R value of $\mathbf{0x5f375a86}$.

This magic constant 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.

This constant is better than the original for both the initial guess and after the iteration step!

The magic constant can be extended into higher precision floating point such as *double* and *quadruple* precision using

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

where b is the bias, U is the size of the mantissa field, and t_0 is the optimal t value given by the root of

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

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

A *double* has a mantissa field of size $U = 52$ and a bias $b = 1023$. Using the previous equation, this yields an optimal magic constant of $\mathbf{0x5fe6eb50c7b537a9}$ with maximum relative error 0.0017511837.

Similarly, for *quadruple precision*, this yields a magic constant of $\mathbf{0x5ffe6eb50c7b537a9cd9f02e504fcfbf}$ with a maximum relative error even closer to the theoretical value.

A similar method could be used to find optimal constants for two Newton iterations.

A *double* has a mantissa field of size $U = 52$ and a bias $b = 1023$. Using the previous equation, this yields an optimal magic constant of $\mathbf{0x5fe6eb50c7b537a9}$ with maximum relative error 0.0017511837. Similarly, for *quadruple precision*, this yields a magic constant of $\mathbf{0x5ffe6eb50c7b537a9cd9f02e504fcfbf}$ with a maximum relative error even closer to the theoretical value.

A similar method could be used to find optimal constants for two Newton iterations.

A *double* has a mantissa field of size $U = 52$ and a bias $b = 1023$. Using the previous equation, this yields an optimal magic constant of $\mathbf{0x5fe6eb50c7b537a9}$ with maximum relative error 0.0017511837. Similarly, for *quadruple precision*, this yields a magic constant of $\mathbf{0x5ffe6eb50c7b537a9cd9f02e504fcfbf}$ with a maximum relative error even closer to the theoretical value.

A similar method could be used to find optimal constants for two Newton iterations.

The magic constant can now be optimized mathematically, using the method described in this thesis, for any sized floating point.

Now that there is a mathematical description of the function, even after Newton's iteration, there is a lot of potential for improvement of the function overall.

More iterations of Newton's method would improve the accuracy.

A very small offset constant could be added to the result.

The average relative error could be reduced at no cost.

May it inspire somebody in the future.

The magic constant can now be optimized mathematically, using the method described in this thesis, for any sized floating point.

Now that there is a mathematical description of the function, even after Newton's iteration, there is a lot of potential for improvement of the function overall.

More iterations of Newton's method would improve the accuracy.

A very small offset constant could be added to the result.

The average relative error could be reduced at no cost.

May it inspire somebody in the future.

The magic constant can now be optimized mathematically, using the method described in this thesis, for any sized floating point.

Now that there is a mathematical description of the function, even after Newton's iteration, there is a lot of potential for improvement of the function overall.

More iterations of Newton's method would improve the accuracy.

A very small offset constant could be added to the result.

The average relative error could be reduced at no cost.

May it inspire somebody in the future.

The magic constant can now be optimized mathematically, using the method described in this thesis, for any sized floating point.

Now that there is a mathematical description of the function, even after Newton's iteration, there is a lot of potential for improvement of the function overall.

More iterations of Newton's method would improve the accuracy.

A very small offset constant could be added to the result.

The average relative error could be reduced at no cost.

May it inspire somebody in the future.

The magic constant can now be optimized mathematically, using the method described in this thesis, for any sized floating point.

Now that there is a mathematical description of the function, even after Newton's iteration, there is a lot of potential for improvement of the function overall.

More iterations of Newton's method would improve the accuracy.

A very small offset constant could be added to the result.

The average relative error could be reduced at no cost.

May it inspire somebody in the future.

The magic constant can now be optimized mathematically, using the method described in this thesis, for any sized floating point.

Now that there is a mathematical description of the function, even after Newton's iteration, there is a lot of potential for improvement of the function overall.

More iterations of Newton's method would improve the accuracy.

A very small offset constant could be added to the result.

The average relative error could be reduced at no cost.

May it inspire somebody in the future.

rsqrt32.c

```
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;
}
```

rsqrt64.c

```c
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;
}
```