Fast inverse square-root program

Andrei Seymour-Howell

March 12, 2021

Why would one need to compute $1/\sqrt{x}$?

The inverse square root is used to normalise vectors. Normalised vectors are needed for 3D graphics programs to determine angles of incidence and reflection.

As a reminder a vector $\mathbf{v} = (v_1, v_2, v_3) \in \mathbb{R}^3$ can be normalised by dividing by it's norm, that is

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{||\mathbf{v}||} = \frac{\mathbf{v}}{\sqrt{v_1^2 + v_2^2 + v_3^2}}.$$

Why would one need to compute $1/\sqrt{x}$?

The inverse square root is used to normalise vectors. Normalised vectors are needed for 3D graphics programs to determine angles of incidence and reflection.

As a reminder a vector $\mathbf{v} = (v_1, v_2, v_3) \in \mathbb{R}^3$ can be normalised by dividing by it's norm, that is

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{||\mathbf{v}||} = \frac{\mathbf{v}}{\sqrt{v_1^2 + v_2^2 + v_3^2}}.$$

3D graphics programs must normalise millions of vectors every second to simulate lighting. In the early 1990's, the code that did this for decimal numbers was computationally expensive, especially when dealing with a large amount of vectors.

Quake 3 Arena

Quake 3 Arena is a first-person shooter multiplayer game released in 1999 based on the famous id tech engine.

Quake 3 Arena

Quake 3 Arena is a first-person shooter multiplayer game released in 1999 based on the famous id tech engine.

Now why is this product placement necessary? Well within the source code in the following quite interesting code to compute the inverse square root.

Magic code

```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;                       // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed

    return y;
}
```

Newton's method

Newton's method is a root-finding algorithm that successively computes better approximations to a root of a function $f(x)$.

Newton's method

Newton's method is a root-finding algorithm that successively computes better approximations to a root of a function $f(x)$. More explicitly, if $f(x) = 0$ for some $x \in \mathbb{R}$, then for some sufficiently close number x_0 to x, the following sequence

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

will converge to x quadratically.

Newton's method

Newton's method is a root-finding algorithm that successively computes better approximations to a root of a function $f(x)$. More explicitly, if $f(x) = 0$ for some $x \in \mathbb{R}$, then for some sufficiently close number x_0 to x, the following sequence

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

will converge to x quadratically.
For nice-enough functions, the hardest part is finding the initial guess.

Newton's method for $1/\sqrt{x}$

For our case we wish to calculate $\frac{1}{\sqrt{x}}$. To do this we consider the function $f(y) = \frac{1}{y^2} - x$, whose positive root is exactly $\frac{1}{\sqrt{x}}$.

Newton's method for $1/\sqrt{x}$

For our case we wish to calculate $\frac{1}{\sqrt{x}}$. To do this we consider the function $f(y) = \frac{1}{y^2} - x$, whose positive root is exactly $\frac{1}{\sqrt{x}}$. Applying Newton's method to this we get the sequence

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

Newton's method for $1/\sqrt{x}$

For our case we wish to calculate $\frac{1}{\sqrt{x}}$. To do this we consider the function $f(y) = \frac{1}{y^2} - x$, whose positive root is exactly $\frac{1}{\sqrt{x}}$. Applying Newton's method to this we get the sequence

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

This is the step occurring in the line:

```
y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
```

Storing integers on a computer

Computers store information using **bits**, which are logical states that can have 2 possible values, think 1 or 0.

Storing integers on a computer

Computers store information using **bits**, which are logical states that can have 2 possible values, think 1 or 0.
Numerically computers work in base 2, so to store an integer on a computer we convert the number to base 2 and keep a track of the sign of the integer. For example 190 is $+10111110$, or -50 is -110010.

Storing integers on a computer

Computers store information using **bits**, which are logical states that can have 2 possible values, think 1 or 0.

Numerically computers work in base 2, so to store an integer on a computer we convert the number to base 2 and keep a track of the sign of the integer. For example 190 is $+10111110$, or -50 is -110010.

Since computers are limited by how many bits it can store, we limit the numbers bits we allow to store each number. If we stick to the same rules throughout the code, we can gain vast speed advantages. This is implemented in programming languages by **data types**.

Storing integers on a computer

Computers store information using **bits**, which are logical states that can have 2 possible values, think 1 or 0.

Numerically computers work in base 2, so to store an integer on a computer we convert the number to base 2 and keep a track of the sign of the integer. For example 190 is $+10111110$, or -50 is -110010.

Since computers are limited by how many bits it can store, we limit the numbers bits we allow to store each number. If we stick to the same rules throughout the code, we can gain vast speed advantages. This is implemented in programming languages by **data types**.

Examples in C include **int** for 16-bit integer, **char** for text characters/strings and **float** for 32-bit decimal numbers.

Long data type

The data type for integers appearing in the code is **long**, which stores whole numbers in 32-bits of memory. The first bit is used to store the sign of the number, 0 for $+$ and 1 for $-$ and the other bits are used to number.

Long data type

The data type for integers appearing in the code is **long**, which stores whole numbers in 32-bits of memory. The first bit is used to store the sign of the number, 0 for $+$ and 1 for $-$ and the other bits are used to number.

For example the number 190 would be stored in memory as

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	1	0

Long data type

The data type for integers appearing in the code is **long**, which stores whole numbers in 32-bits of memory. The first bit is used to store the sign of the number, 0 for $+$ and 1 for $-$ and the other bits are used to number.
For example the number 190 would be stored in memory as

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	1	0

This allows us to store all integers in the range
$[-2^{31}+1, 2^{31}-1] = [-2,147,483,647, +2,147,483,647]$.

Long data type

The data type for integers appearing in the code is **long**, which stores whole numbers in 32-bits of memory. The first bit is used to store the sign of the number, 0 for $+$ and 1 for $-$ and the other bits are used to number.
For example the number 190 would be stored in memory as

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	1	0

This allows us to store all integers in the range
$[-2^{31}+1, 2^{31}-1] = [-2,147,483,647, +2,147,483,647]$.
On computers we actually store negative numbers in **two's complement** form. Since we'll only be dealing with positive numbers we wont need to look at it here.

How to store decimal values?

The first simplest way to thing about doing this is to just also consider negative powers of 2. For example allow the first 16-bits to be positive powers of 2 and the rest be negative powers of 2 to give you the decimal digits.

How to store decimal values?

The first simplest way to thing about doing this is to just also consider negative powers of 2. For example allow the first 16-bits to be positive powers of 2 and the rest be negative powers of 2 to give you the decimal digits.
The main drawback of this is the numbers we can store are quite small, only up to $2^{15} - 1$ if we also include a sign bit.

How to store decimal values?

The first simplest way to thing about doing this is to just also consider negative powers of 2. For example allow the first 16-bits to be positive powers of 2 and the rest be negative powers of 2 to give you the decimal digits.

The main drawback of this is the numbers we can store are quite small, only up to $2^{15} - 1$ if we also include a sign bit.

Thankfully some clever people at the Institute of Electrical and Electronic Engineers(IEEE) came up with a standard to store these numbers more efficiently.

IEEE 754-1985 Floating point numbers

The main idea is that we already have a way to minimise how we write numbers by using scientific notation. That is we tend to write a number as

$$x = \pm d_0.d_1 d_2 d_3 \ldots \times 10^e$$

where $1 \leq d_0 < 10$.

IEEE 754-1985 Floating point numbers

The main idea is that we already have a way to minimise how we write numbers by using scientific notation. That is we tend to write a number as

$$x = \pm d_0.d_1 d_2 d_3 \ldots \times 10^e$$

where $1 \leq d_0 < 10$. The exponent e tends to be quite small so when we store this number we can add more bits to describe the d_i's.

IEEE 754-1985 Floating point numbers

The main idea is that we already have a way to minimise how we write numbers by using scientific notation. That is we tend to write a number as

$$x = \pm d_0.d_1 d_2 d_3 \ldots \times 10^e$$

where $1 \leq d_0 < 10$. The exponent e tends to be quite small so when we store this number we can add more bits to describe the d_i's.

In base 2 we would write our number as

$$x = \pm 1.b_1 b_2 b_3 \ldots \times 2^{e_x} = \pm(1 + m_x)2^{e_x}$$

where e_x is an integer.

IEEE 754-1985 Floating point numbers

The main idea is that we already have a way to minimise how we write numbers by using scientific notation. That is we tend to write a number as

$$x = \pm d_0.d_1 d_2 d_3 \ldots \times 10^e$$

where $1 \leq d_0 < 10$. The exponent e tends to be quite small so when we store this number we can add more bits to describe the d_i's.

In base 2 we would write our number as

$$x = \pm 1.b_1 b_2 b_3 \ldots \times 2^{e_x} = \pm(1 + m_x)2^{e_x}$$

where e_x is an integer. We call e_x the **exponent** and $1 + m_x$ the **mantissa** or **significand**. On a computer we would only need to store the numbers e_x and m_x and the sign since we will only ever work in base 2.

IEEE 754-1985 Floating point numbers

The IEEE 754 standard states that if you want to store a number in this way in 32-bits:

- 1-bit should be given to the sign,
- 8-bits should given to the exponent e_x,
- 23-bits should be given to the mantissa m_x.

IEEE 754-1985 Floating point numbers

The IEEE 754 standard states that if you want to store a number in this way in 32-bits:

- 1-bit should be given to the sign,
- 8-bits should given to the exponent e_x,
- 23-bits should be given to the mantissa m_x.

To actually store this physically in the memory, we have to first change how we write e_x and m_x.

- We write $E_x = e_x + B$ where $B = 127 = 2^7 - 1$ called the **exponent bias**.

IEEE 754-1985 Floating point numbers

The IEEE 754 standard states that if you want to store a number in this way in 32-bits:

- 1-bit should be given to the sign,
- 8-bits should given to the exponent e_x,
- 23-bits should be given to the mantissa m_x.

To actually store this physically in the memory, we have to first change how we write e_x and m_x.

- We write $E_x = e_x + B$ where $B = 127 = 2^7 - 1$ called the **exponent bias**.
- We write $M_x = m_x \times L$ where $L = 2^{23}$, then round the number to an integer.

IEEE 754-1985 Floating point numbers

The IEEE 754 standard states that if you want to store a number in this way in 32-bits:

- 1-bit should be given to the sign,
- 8-bits should given to the exponent e_x,
- 23-bits should be given to the mantissa m_x.

To actually store this physically in the memory, we have to first change how we write e_x and m_x.

- We write $E_x = e_x + B$ where $B = 127 = 2^7 - 1$ called the **exponent bias**.
- We write $M_x = m_x \times L$ where $L = 2^{23}$, then round the number to an integer.

This means we only need to 3 integers that are all positive E_x, M_x and the sign.

Floating point example

Let's look at $x = \pi = 3.1415926535897 93\ldots$. In fixed point arithmetic in base 2 we have $x = 11.001001000011111101 1011\ldots$.

Floating point example

Let's look at $x = \pi = 3.141592653589793\ldots$. In fixed point arithmetic in base 2 we have
$x = 11.001001000011111101101 1\ldots$.
In the normalised form, we would get

$$\begin{aligned} x &= +(1 + 0.5707964897155761 7188\ldots) \times 2^1 \\ &= +(1 + 0.10010010000111111011011\ldots) \times 2^1. \end{aligned}$$

Floating point example

Let's look at $x = \pi = 3.141592653589793\ldots$. In fixed point arithmetic in base 2 we have $x = 11.001001000011111101101 1\ldots$.
In the normalised form, we would get

$$
\begin{aligned}
x &= +(1 + 0.5707964897155761718 8\ldots) \times 2^1 \\
&= +(1 + 0.10010010000111111011011\ldots) \times 2^1.
\end{aligned}
$$

Hence $e_x = 1$, which means $E_x = 1 + B = 128 = 10000000_2$ and $m_x = 0.10010010000111111011011\ldots$, which gives $M_x = m_x \times L = 10010010000111111011011 = 4788187$ after rounding.

Floating point example

Let's look at $x = \pi = 3.141592653589793\ldots$. In fixed point arithmetic in base 2 we have
$x = 11.001001000011111101101 1\ldots$.
In the normalised form, we would get

$$
\begin{aligned}
x &= +(1 + 0.5707964897155761718 8\ldots) \times 2^1 \\
&= +(1 + 0.10010010000111111011011\ldots) \times 2^1.
\end{aligned}
$$

Hence $e_x = 1$, which means $E_x = 1 + B = 128 = 10000000_2$ and $m_x = 0.10010010000111111011011\ldots$, which gives $M_x = m_x \times L = 10010010000111111011011 = 4788187$ after rounding.

0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

Initialisation step

```
long i;
float x2, y;
const float threehalfs = 1.5F;

x2 = number * 0.5F;
y  = number;
```

evil floating point bit level hack

The claim of this line of code, is that treating the bits of a positive floating point number x as a long type gives a rough approximation to $\log_2(x)$.

evil floating point bit level hack

The claim of this line of code, is that treating the bits of a positive floating point number x as a long type gives a rough approximation to $\log_2(x)$.

To begin we shall write $x = 2^{e_x}(1 + m_x)$, then

$$\log_2(x) = e_x + \log_2(1 + m_x).$$

evil floating point bit level hack

The claim of this line of code, is that treating the bits of a positive floating point number x as a long type gives a rough approximation to $\log_2(x)$.
To begin we shall write $x = 2^{e_x}(1 + m_x)$, then

$$\log_2(x) = e_x + \log_2(1 + m_x).$$

Since $0 \leq m_x < 1$, the logarithm on the right can be approximated by

$$\log_2(1 + m_x) \approx m_x + \sigma$$

where σ is a free parameter used to tune the approximation.

evil floating point bit level hack

The claim of this line of code, is that treating the bits of a positive floating point number x as a long type gives a rough approximation to $\log_2(x)$.
To begin we shall write $x = 2^{e_x}(1 + m_x)$, then

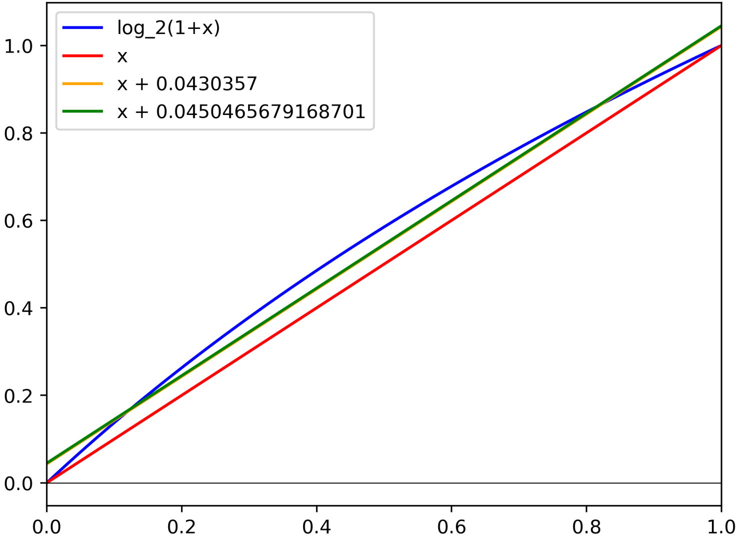
$$\log_2(x) = e_x + \log_2(1 + m_x).$$

Since $0 \leq m_x < 1$, the logarithm on the right can be approximated by

$$\log_2(1 + m_x) \approx m_x + \sigma$$

where σ is a free parameter used to tune the approximation.
It turns out that $\sigma \approx 0.0430357\ldots$ gives the best approximation for the uniform error along the interval. For historical purposes we shall let $\sigma = 0.0450465679168701$.

$\log_2(1 + m_x)$ error



evil floating point bit level hack

We now do the "evil floating point bit level hack" and interpret the floating point bits of x as a long type. Since the mantissa is already an integer it doesn't change, so all we really do is add on the exponent to the 24-th bit onwards, numerically this means just multiplying it by $L = 2^{23}$.

evil floating point bit level hack

We now do the “evil floating point bit level hack” and interpret the floating point bits of x as a long type. Since the mantissa is already an integer it doesn’t change, so all we really do is add on the exponent to the 24-th bit onwards, numerically this means just multiplying it by $L = 2^{23}$.

In the code this is given by

```
i  = * ( long * ) &y;                       // evil floating point bit level hacking
```

evil floating point bit level hack

We now do the "evil floating point bit level hack" and interpret the floating point bits of x as a long type. Since the mantissa is already an integer it doesn't change, so all we really do is add on the exponent to the 24-th bit onwards, numerically this means just multiplying it by $L = 2^{23}$.

In the code this is given by

```
i  = * ( long * ) &y;                       // evil floating point bit level hacking
```

Mathematically, this means

$$\begin{aligned} I_x &= E_x L + M_x \\ &= L(e_x + B + m_x) \\ &= L(e_x + m_x + \sigma + B - \sigma) \\ &\approx L \log_2(x) + L(B - \sigma) \end{aligned}$$

evil floating point bit level hack

We now do the "evil floating point bit level hack" and interpret the floating point bits of x as a long type. Since the mantissa is already an integer it doesn't change, so all we really do is add on the exponent to the 24-th bit onwards, numerically this means just multiplying it by $L = 2^{23}$.

In the code this is given by

```
i  = * ( long * ) &y;                       // evil floating point bit level hacking
```

Mathematically, this means

$$\begin{aligned} I_x &= E_x L + M_x \\ &= L(e_x + B + m_x) \\ &= L(e_x + m_x + \sigma + B - \sigma) \\ &\approx L \log_2(x) + L(B - \sigma) \end{aligned}$$

Rearranging, we see that I_x is a linear approximation to $\log_2(x)$

$$\log_2(x) \approx \frac{I_x}{L} - (B - \sigma).$$

WTF

If we let $y = \frac{1}{\sqrt{x}}$, the magic number that I alluded to earlier, actually isn't that magic, it just comes from the identity

$$\log_2(y) = -\frac{1}{2}\log_2(x).$$

WTF

If we let $y = \frac{1}{\sqrt{x}}$, the magic number that I alluded to earlier, actually isn't that magic, it just comes from the identity

$$\log_2(y) = -\frac{1}{2}\log_2(x).$$

Using our approximation from before we can write this as

$$\frac{I_y}{L} - (B - \sigma) \approx -\frac{1}{2}\left(\frac{I_x}{L} - (B - \sigma)\right)$$

which yields

$$I_y \approx \frac{3}{2}L(B - \sigma) - \frac{1}{2}I_x.$$

Magic number reveal

This first term is just a constant given by

$$\frac{3}{2}L(B-\sigma) = 1597463007$$

Magic number reveal

This first term is just a constant given by

$$\frac{3}{2}L(B-\sigma) = 1597463007 = \mathrm{0x5f3759df}$$

in base 16 or hexadecimal. This is written in the code in the line

```
i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
```

Magic number reveal

This first term is just a constant given by

$$\frac{3}{2}L(B-\sigma) = 1597463007 = \text{0x5f3759df}$$

in base 16 or hexadecimal. This is written in the code in the line

```
i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
```

The term $\frac{1}{2}I_x$ is computed by shifting all the bits to the right by one. Since we work base 2 this will be the same as dividing by 2.

Magic number reveal

This first term is just a constant given by

$$\frac{3}{2}L(B-\sigma) = 1597463007 = \text{0x5f3759df}$$

in base 16 or hexadecimal. This is written in the code in the line

```
i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
```

The term $\frac{1}{2}I_x$ is computed by shifting all the bits to the right by one. Since we work base 2 this will be the same as dividing by 2.

Then the last line

```
y  = * ( float * ) &i;
```

just converts I_y back to y and gives us the initial guess of $y = \frac{1}{\sqrt{x}}$.

How good is this approximation?

Let's consider again $x = 3.14159\ldots$, our memory for y looks like

0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

How good is this approximation?

Let's consider again $x = 3.14159\ldots$, our memory for y looks like

0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

Now interpreting this as an long type and storing this in i, the memory address for i looks like

0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

How good is this approximation?

Let's consider again $x = 3.14159\ldots$, our memory for *y* looks like

0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

Now interpreting this as an long type and storing this in *i*, the memory address for *i* looks like

0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

Shifting the bits by one place to the right, i.e the line $i >> 1$, *i* becomes

0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1

How good is this approximation?

Let's consider again $x = 3.14159\ldots$, our memory for y looks like

0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

Now interpreting this as an long type and storing this in i, the memory address for i looks like

0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

Shifting the bits by one place to the right, i.e the line $i >> 1$, i becomes

0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1

Then subtracting this from our "magic number" 0x5f3759df, we get

0	0	1	1	1	1	1	1	0	0	0	1	0	0	1	0	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0

How good is this approximation?

Let's consider again $x = 3.14159\ldots$, our memory for y looks like

0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

Now interpreting this as an long type and storing this in *i*, the memory address for *i* looks like

0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

Shifting the bits by one place to the right, i.e the line $i >> 1$, *i* becomes

0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1

Then subtracting this from our "magic number" 0x5f3759df, we get

0	0	1	1	1	1	1	1	0	0	0	1	0	0	1	0	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0

Finally reinterpreting this as a float and storing this back in *y* we get

0	0	1	1	1	1	1	1	0	0	0	1	0	0	1	0	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0

How good is this approximation?

Let's consider again $x = 3.14159\ldots$, our memory for y looks like

0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

Now interpreting this as an long type and storing this in i, the memory address for i looks like

0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

Shifting the bits by one place to the right, i.e the line $i >> 1$, i becomes

0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1

Then subtracting this from our "magic number" 0x5f3759df, we get

0	0	1	1	1	1	1	1	0	0	0	1	0	0	1	0	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0

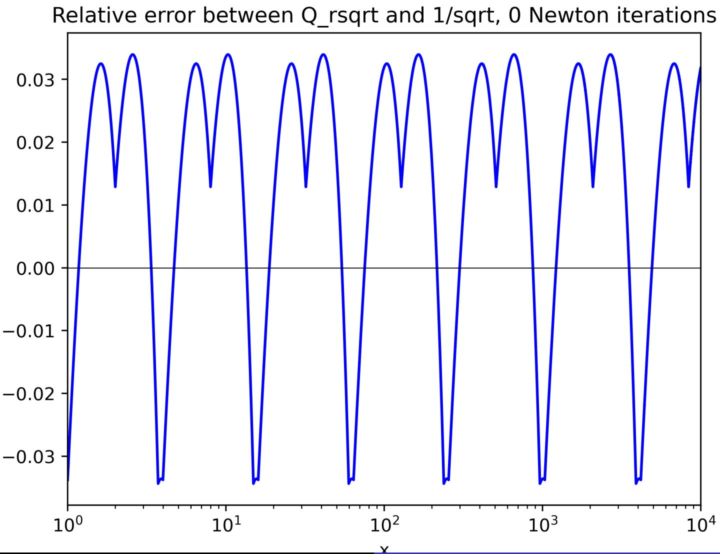
Finally reinterpreting this as a float and storing this back in y we get

0	0	1	1	1	1	1	1	0	0	0	1	0	0	1	0	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0

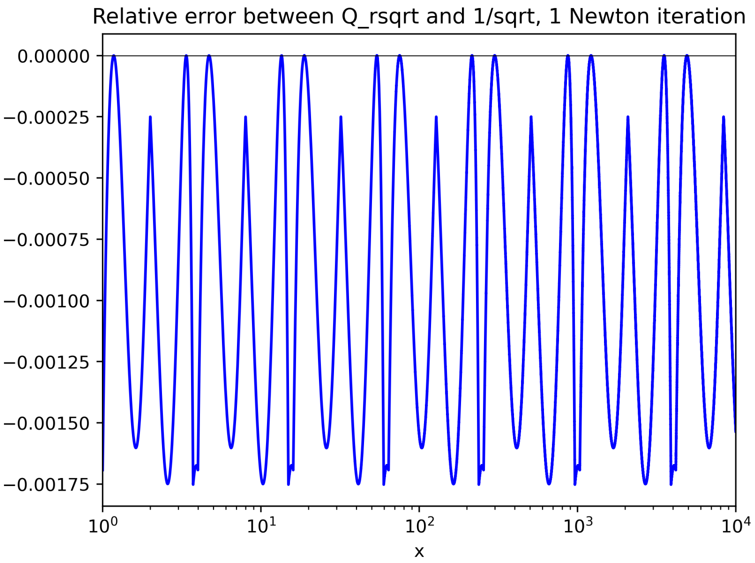
As a decimal number this gives $y = 0.5735160112$. After 1 Newton iteration we get $y = 0.5639570355$. The actual value is $0.564189583547750\ldots$.

Error graph

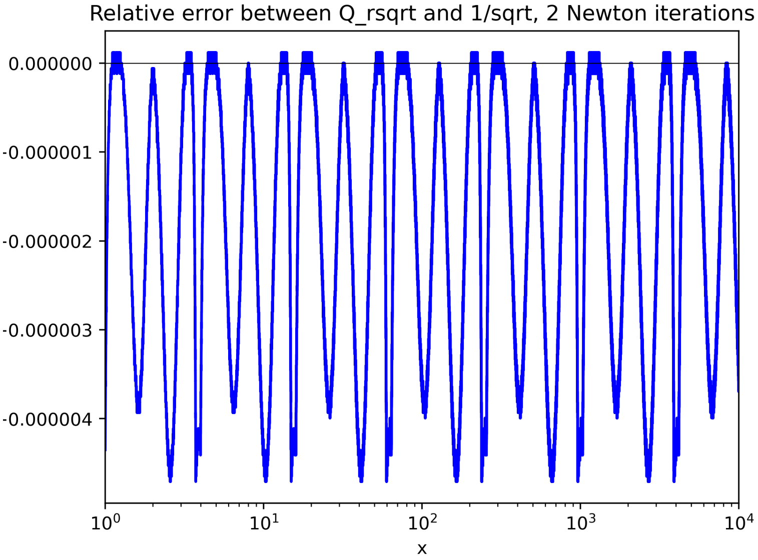
Relative error $= \frac{v_A - v_E}{v_E}$



Better Error graph



Even Better Error graph



The big question, should you use this?

No.

The big question, should you use this?

No. Probably.

Reasons not to use this code

1. The main reason is this is no longer the fastest method. In 1999 the “Streaming SIMD Extensions” (SSE) were added to x86 architecture CPU, effectively allowing certain operations, like square-rooting directly on the CPU without needing to do anything in software. One of the functions include “rsqrtss” which computes the inverse-square root considerably faster and to the full 11 decimal accuracy. Most modern compilers will automatically choose the SSE functions even when you type the software version.

Reasons not to use this code

1. The main reason is this is no longer the fastest method. In 1999 the "Streaming SIMD Extensions" (SSE) were added to x86 architecture CPU, effectively allowing certain operations, like square-rooting directly on the CPU without needing to do anything in software. One of the functions include "rsqrtss" which computes the inverse-square root considerably faster and to the full 11 decimal accuracy. Most modern compilers will automatically choose the SSE functions even when you type the software version.
2. A second slightly less important reason is that this reinterpreting floats and integers using pointers is considered undefined behaviour and some computers probably wont like it. (As of a few months ago C++ defined correct behaviour using C++20's std::bit_cast function).

History

- John Carmack

History

- John Carmack
- Terje Mathisen

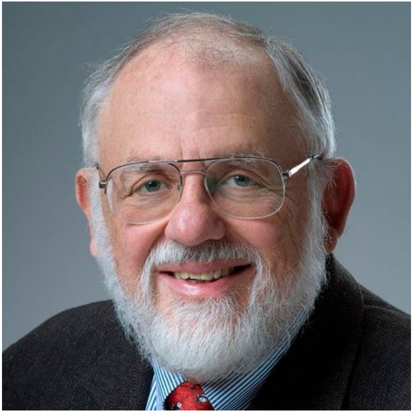
History

- John Carmack
- Terje Mathisen
- Gary Tarolli

History

- John Carmack
- Terje Mathisen
- Gary Tarolli
- Greg Walsh

Greg Walsh's inspiration



Cleve Moler

Thanks for listening!

Main references:

- FAST INVERSE SQUARE ROOT - Chris Lomont
- M.Robertson: A Brief History of InvSqrt, Bachelor Thesis, Univ. of New Brunswick 2012.