
Fast Inverse Square Root

Prepared by Mark on September 25, 2025

Section 1: Introduction

In 2005, ID Software published the source code of *Quake III Arena*, a popular game released in 1999. This caused quite a stir: ID Software was responsible for many games popular among old-school engineers (most notably *Doom*, which has a place in programmer humor even today).

Naturally, this community immediately began dissecting *Quake*'s source. One particularly interesting function is reproduced below, with original comments:

```
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 ); // [redacted]
    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;
}
```

This code defines a function `Q_sqrt`, which was used as a fast approximation of the inverse square root in graphics routines. (in other words, `Q_sqrt` efficiently approximates $1 \div \sqrt{x}$)

The key word here is “fast”: *Quake* ran on very limited hardware, and traditional approximation techniques (like Taylor series)¹ were too computationally expensive to be viable.

Our goal today is to understand how `Q_sqrt` works.

To do that, we’ll first need to understand how computers represent numbers.

We’ll start with simple binary integers—turn the page.

¹Taylor series aren’t used today, and for the same reason. There are better ways.

Section 2: Integers

Definition 1:

A *bit string* is a string of binary digits.

In this handout, we'll denote bit strings with the prefix `0b`.

This prefix is only notation—it is *not* part of the string itself.

For example, `1001` is the number “one thousand and one,” while `0b1001` is the string of bits “1 0 0 1”.

We will separate long bit strings with underscores for readability.

Underscores have no meaning: `0b1111_0000` = `0b11110000`.

Problem 2:

What is the value of the following bit strings, if we interpret them as integers in base 2?

- `0b0001_1010`
- `0b0110_0001`

Definition 3:

We can interpret a bit string in any number of ways.

One such interpretation is the *unsigned integer*, or `uint` for short.

`uints` allow us to represent positive (hence “unsigned”) integers using 32-bit strings.

The value of a `uint` is simply its value as a binary number:

- `0b00000000_00000000_00000000_00000000` = 0
- `0b00000000_00000000_00000000_00000011` = 3
- `0b00000000_00000000_00000000_00100000` = 32
- `0b00000000_00000000_00000000_10000010` = 130

Problem 4:

What is the largest number we can represent with a 32-bit `uint`?

Problem 5:

Find the value of each of the following 32-bit unsigned integers:

- `0b00000000_00000000_00000101_00111001`
- `0b00000000_00000000_00000001_00101100`
- `0b00000000_00000000_00000100_10110000`

Hint: The third conversion is easy—look carefully at the second.

Definition 6:

In general, fast division of `uints` is difficult².

Division by powers of two, however, is incredibly easy:

To divide by two, all we need to do is shift the bits of our integer right.

For example, consider `0b0000_0110 = 6`.

If we insert a zero at the left end of this string and delete the zero at the right (thus “shifting” each bit right), we get `0b0000_0011`, which is 3.

Of course, we lose the remainder when we right-shift an odd number:

9 shifted right is 4, since `0b0000_1001` shifted right is `0b0000_0100`.

Problem 7:

Right shifts are denoted by the `>>` symbol:

`00110 >> n` means “shift `0b0110` right n times.”

Find the value of the following:

- `12 >> 1`
- `27 >> 3`
- `16 >> 8`

Naturally, you’ll have to convert these integers to binary first.

²One may use repeated subtraction, but this isn’t efficient.

Section 3: Floats

Definition 8:

*Binary decimals*³ are very similar to base-10 decimals.

In base 10, we interpret place value as follows:

- $0.1 = 10^{-1}$
- $0.03 = 3 \times 10^{-2}$
- $0.0208 = 2 \times 10^{-2} + 8 \times 10^{-4}$

We can do the same in base 2:

- $0.1 = 2^{-1} = 0.5$
- $0.011 = 2^{-2} + 2^{-3} = 0.375$
- $101.01 = 5.125$

Problem 9:

Rewrite the following binary decimals in base 10:

You may leave your answer as a fraction.

- 1011.101
- 110.1101

³Note that “binary decimal” is a misnomer—“deci” means “ten”!

Definition 10:

Another way we can interpret a bit string is as a *signed floating-point decimal*, or a **float** for short. Floats represent a subset of the real numbers, and are interpreted as follows: The following only applies to floats that consist of 32 bits. We won't encounter any others today.

$$\begin{array}{c} 0\ b\ 0_00000000_00000000_00000000_00000000 \\ \hline \begin{array}{ccc} s & \text{exponent} & \text{fraction} \end{array} \end{array}$$

- The first bit denotes the sign of the float's value We'll label it s .
If $s = 1$, this float is negative; if $s = 0$, it is positive.
- The next eight bits represent the *exponent* of this float. (we'll see what that means soon)
We'll call the value of this eight-bit binary integer E .
Naturally, $0 \leq E \leq 255$ (since E consist of eight bits)
- The remaining 23 bits represent the *fraction* of this float.
They are interpreted as the fractional part of a binary decimal.
For example, the bits `0b10100000_00000000_00000000` represent $0.5 + 0.125 = 0.625$.
We'll call the value of these bits as a binary integer F .
Their value as a binary decimal is then $F \div 2^{23}$. (convince yourself of this)

Problem 11:

Consider `0b01000001_10101000_00000000_00000000`.

Hint: The underscores here do *not* match those in Definition 10

Find the s , E , and F we get if we interpret this bit string as a float.

Leave F as a sum of powers of two.

Definition 12:

The final value of a float with sign s , exponent E , and fraction F is

$$(-1)^s \times 2^{E-127} \times \left(1 + \frac{F}{2^{23}}\right)$$

Notice that this is very similar to base-10 scientific notation, which is written as

$$(-1)^s \times 10^e \times (f)$$

We subtract 127 from E so we can represent positive and negative numbers.

E is an eight bit binary integer, so $0 \leq E \leq 255$ and thus $-127 \leq (E - 127) \leq 127$.

Problem 13:

Consider `0b01000001_10101000_00000000_00000000`.

This is the same bit string we used in Problem 11.

What value do we get if we interpret this bit string as a float?

Hint: $21 \div 16 = 1.3125$

Problem 14:

Encode 12.5 as a float.

Hint: $12.5 \div 8 = 1.5625$

Definition 15:

Say we have a bit string x .

We'll let x_f denote the value we get if we interpret x as a float, and we'll let x_i denote the value we get if we interpret x as an integer.

Problem 16:

Let $x = 0b01000001_01001000_00000000_00000000$.

What are x_f and x_i ? As always, you may leave big numbers as powers of two.

Section 4: Integers and Floats

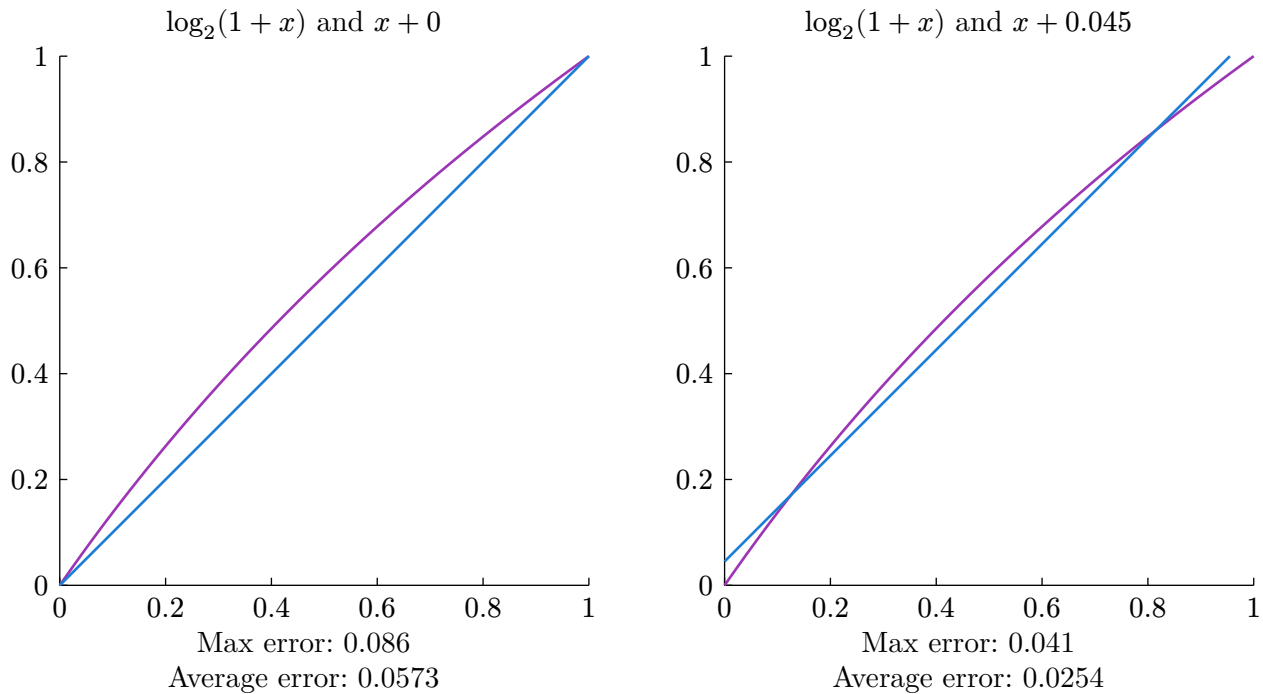
Observation:

If x is smaller than 1, $\log_2(1+x)$ is approximately equal to x .

Note that this equality is exact for $x = 0$ and $x = 1$, since $\log_2(1) = 0$ and $\log_2(2) = 1$.

We'll add the *correction term* ε to our approximation: $\log_2(1+a) \approx a + \varepsilon$.

This allows us to improve the average error of our linear approximation:



A suitable value of ε can be found using calculus or with computational trial-and-error.

We won't bother with this—we'll simply leave the correction term as an opaque constant ε .

Note: "Average error" above is simply the area of the region between the two graphs:

$$\int_0^1 | \log(1+x)_2 - (x + \varepsilon) |$$

Feel free to ignore this note, it isn't a critical part of this handout.

Problem 17:

Use the fact that $\log_2(1 + a) \approx a + \varepsilon$ to approximate $\log_2(x_f)$ in terms of x_i .
Namely, show that

$$\log_2(x_f) = \frac{x_i}{2^{23}} - 127 + \varepsilon$$

In other words, we're finding an expression for x as a float in terms of x as an int.

Problem 18:

Using basic log rules, rewrite $\log_2\left(\frac{1}{\sqrt{x}}\right)$ in terms of $\log_2(x)$.

Section 5: The Fast Inverse Square Root

A simplified version of the *Quake* routine we are studying is reproduced below.

```
float Q_rsqrt( float number ) {  
    long i = * ( long * ) &number;  
    i = 0x5f3759df - ( i >> 1 );  
    return * ( float * ) &i;  
}
```

This code defines a function `Q_rsqrt` that consumes a `float number` and approximates its inverse square root. If we rewrite this using notation we're familiar with, we get the following:

$$Q_sqrt(n_f) = 6240089 - (n_i \div 2) \approx \frac{1}{\sqrt{n_f}}$$

`0x5f3759df` is 6240089 in hexadecimal.

Ask an instructor to explain if you don't know what this means.

It is a magic number hard-coded into `Q_sqrt`.

Our goal in this section is to understand why this works:

- How does *Quake* approximate $\frac{1}{\sqrt{x}}$ by simply subtracting and dividing by two?
- What's special about 6240089?

Remark 19:

For those that are interested, here are the details of the “code-to-math” translation:

- “`long i = * (long *) &number`” is C magic that tells the compiler to set `i` to the `uint` value of the bits of `number`.
“`long`” refers to a “long integer”, which has 32 bits.
Normal ints have 16 bits, short ints have 8.
In other words, `number` is n_f and `i` is n_i .
- Notice the right-shift in the second line of the function.
We translated `(i >> 1)` into $(n_i \div 2)$.
- “`return * (float *) &i`” is again C magic.
Much like before, it tells us to return the value of the bits of `i` as a float.

Setup:

We are now ready to show that `Q_sqrt` (x) effectively approximates $\frac{1}{\sqrt{x}}$.

For convenience, let's call the bit string of the inverse square root r .

In other words,

$$r_f := \frac{1}{\sqrt{n_f}}$$

This is the value we want to approximate.

Problem 20:

Find an approximation for $\log_2(r_f)$ in terms of n_i and ε

Remember, ε is the correction constant in our approximation of $\log_2(1+x)$.

Problem 21:

Let's call the "magic number" in the code above κ , so that

$$\text{Q_sqrt}(n_f) = \kappa - (n_i \div 2)$$

Use Problem 17 and Problem 20 to show that $\text{Q_sqrt}(n_f) \approx r_i$

Note: If we know r_i , we know r_f .

We don't even need to convert between the two—the underlying bits are the same!

Problem 22:

What is the exact value of κ in terms of ε ?

Hint: Look at Problem 21. We already found it!

Remark 23:

In Problem 22 we saw that $\kappa = (3 \times 2^{22})(127 - \varepsilon)$.

Looking at the code again, we see that $\kappa = 0x5f3759df$ in *Quake*:

```
float Q_rsqr( float number ) {
    long i = * ( long * ) &number;
    i = 0x5f3759df - ( i >> 1 );
    return * ( float * ) &i;
}
```

Using a calculator and some basic algebra, we can find the ε this code uses:

Remember, $0x5f3759df$ is 6240089 in hexadecimal.

$$\begin{aligned}(3 \times 2^{22})(127 - \varepsilon) &= 6240089 \\ (127 - \varepsilon) &= 126.955 \\ \varepsilon &= 0.0450466\end{aligned}$$

So, 0.045 is the ε used by *Quake*.

Online sources state that this constant was generated by trial-and-error, though it is fairly close to the ideal ε .

Remark 24:

And now, we're done!

We've shown that $Q_sqrt(x)$ approximates $\frac{1}{\sqrt{x}}$ fairly well.

Notably, Q_sqrt uses *zero* divisions or multiplications ($>>$ doesn't count).

This makes it *very* fast when compared to more traditional approximation techniques (i.e, Taylor series).

In the case of *Quake*, this is very important. 3D graphics require thousands of inverse-square-root calculations to render a single frame⁴, which is not an easy task for a Playstation running at 300MHz.

⁴e.g, to generate normal vectors

Section 6: Bonus – More about Floats

Problem 25:

Convince yourself that all numbers that can be represented as a float are rational.

Problem 26:

Find a rational number that cannot be represented as a float.

Problem 27:

What is the smallest positive 32-bit float?

Problem 28:

What is the largest positive 32-bit float?

Problem 29:

How many floats are between -1 and 1 ?

Problem 30:

How many floats are between 1 and 2 ?

Problem 31:

How many floats are between 1 and 128 ?