Quake 3 Reciprocal Square Root: The Fun Parts

Jerome Coonen

16 April 2022

One liner

Given x, compute $y = \frac{1}{\sqrt{x}}$ with an ingenious first guess for y and then one step of classic Newton's Iteration to improve it. This short code in the video game Quake 3 has gained cult fame over the past 20 years. Line for line, it may be the most discussed code in history.

A quick preview

Here is the code, adapted from the Wikipedia page *Fast Inverse Square Root*.

```
            float Q_sqrt( float x ) {
                long i;
                float y;
(1)             i  = * ( long * ) &x;       // evil floating point bit-level hacking
(2)             i  = 0x5f3759df - ( i >> 1 );    // WTF?
(3)             y  = * ( float * ) &i;
(4)             y  = y * ( 1.5F - ( 0.5F * x * y * y ) );   // 1st iteration
                // y  = y * ( 1.5F - ( 0.5F * x * y * y ) );   skip 2nd iteration
            }
```

You don't have to be a C expert to see something fishy here in lines (1) and (3). The variables `x` and `y` are 32-bit floating point numbers in IEEE Standard 754 format, and `i` is a 32-bit integer. The "`F`" suffix on the decimal constants indicates they should be 32-bit floating point values, too.

Line (1) says, "just put the bits of `x` into `i`, I know what I'm doing." The pejorative comment exposes this as potentially dangerous behavior. Line (2) is the marvelous first guess of the inverse square root via subtraction and a 1-bit shift, with a mysterious hex constant thrown in. Line (2) is the primary subject of this note.

Then line (3) simply puts the bits of `i` back into `y`, where they regain their IEEE Standard interpretation. Lines (1) and (3) don't compute anything. They permit 32-bit values to pass between integer and floating point contexts. Finally, line (4) improves the accuracy of `y` with an everyday numerical technique. Normally, one performs two or more Newton steps, but the designers commented out the second step when they determined that one step delivered sufficient accuracy for Quake3.

The Wikipedia page above is an excellent resource, featuring a mix of Quake3 history, background on the importance of the reciprocal square root, and further references.

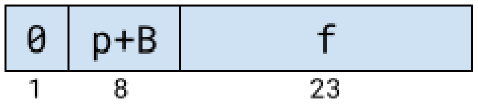
The trick

The trick behind line (2) is easy to state but less easy to see: compute the first guess of **y** in the parallel universe of base-2 logarithms. The formula

$$log2(y) \;=\; log2(1\,/\,\sqrt{x}) \;=\; -\tfrac{1}{2}log2(x)$$

illustrates how straightforward the reciprocal root is in the log world – change sign and divide by 2. The question becomes how we can pass quickly between $x \leftrightarrow log2(x)$ for any x. How do we enter the parallel universe?

A logarithm hiding right inside



A positive normal number z in IEEE 32-bit format has the form shown, with $x = 2^p \times 1.f$. Ignoring B for a moment, take the log:

$$log2(x) = log2\left(2^p \times 1.f\right) \;=\; p \;+\; log2(1 + f)\;.$$

Convert to the natural log with a bit more high school algebra:

$$log2(x) = p + log2(e)log(1 + f)\;.$$

$log2(e)$ is about 1.44. It’s the log base 2 of $e = 2.71828...$ the base of the natural logarithms. The Taylor Series expansion of the natural log is helpful, too. Note how the 1 in the $1.f$ vanishes, using this approximation.

$$log(1 + f) \;=\; f \;-f^2/2 + f^3/3 \;-\; ... \approx f \quad \text{for small } f\,.$$

All this work brings us to

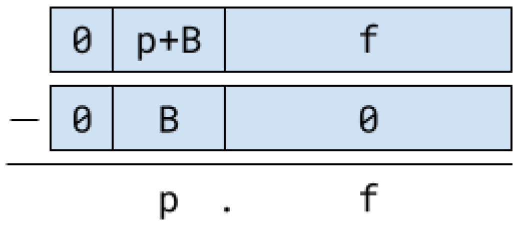
$$log2(x) \approx p + 1.44 \times f \approx p + f\;.$$

This is a rough approximation! We casually tossed the factor 1.44, but we’ll revisit that matter later. The beauty is that the expression $p + f$ is just $p.f$, namely, the bits of the 32-bit field with **B** removed and a binary point insinuated between the **p** and **f** fields. The approximation is sitting right there!

This is totally unintuitive. It’s why people write code comments like, “evil floating point bit-level hacking.”

A little bit-level hacking to remove B

The algebra of the last section offered a different interpretation of the bits of a 32-bit floating point number. Here’s how to pass through the portal into the parallel universe.



The number **B** is a bias added to the signed exponent to bring it into the nonnegative range `0x00` to `0xff`. The bias is 127, or hex `0x7f`. We can now compute an approximate $log2(x)$ by simply subtracting **B**, aligned 23 bits leftward. In code this could look like

```
i = * (long *) x;
i  = i - 0x3f800000;   // namely 0x7f << 23
```

The result is approximate $log2(x)$ as a signed 32-bit fixed-point binary value. A great deal of conversation led to this one 32-bit subtraction.

An example at the speed of light

Let's try our log skills with c the speed of light. $c \approx 3 \times 10^8 \, m/s$. Its 32-bit floating point value is **`4d8f0d18`** which is printed by this bit of code.

```
float c = 3.0e8;
printf("c = %.1e = 0x%08x\n", c, *(long *) &c);
```

The exponent bits are **`4d800000`** so we right shift by 23 bits to **`9b`**, which we convert to 155 decimal, subtract the bias 127, and arrive at 28. The bits of the fraction **`f`** are **`8f0d18`**, with the binary point just right of the leading bit. This is **not** obvious the first twenty times you see it, and we're not making a fuss about the *implicit* leading bit, but shifting the significant bits left one bit gives us the convenient form $c \approx 2^{28} \times 1.1e1a3_{16}$. In the second previous section, we saw that lurking within is $log2(c) \approx 1c.1e1a3_{16} = 28.1176$. The true value is

$log2(c) = 1c.290f2..._{16} = 28.1603...$ so we have a ballpark estimate.

As a sanity check, we could restore the factor $log2(e) = 1.44$ to the fraction part to improve the approximation. Th result is a more credible $log2(c) \approx 28.1693$. Below we'll find an even better way to improve the $log2()$ first approximation.

Approximating the reciprocal root

The next step is to approximate $log2(y) = -\frac{1}{2}log2(x)$. This line of code shifts the log right one bit and negates it. The binary shift effects the division by 2.

```
i  = -((i - 0x3f800000) >> 1);
```

This is the first hint of line (2) in the Quake3 code.

To retrieve the floating point value y from $log2(y)$ we run the process backward – add the bias back in! The following blend of C programming and mathematical rearrangement simplifies the formula.

```
i  = 0x3f800000 -((i - 0x3f800000) >> 1);  // add the bias to “make” it floating
   = 0x3f800000 + (0x3f800000 >> 1) -(i >> 1);  // gather the hex constants
   = 0x5f400000 - (i >> 1);
```

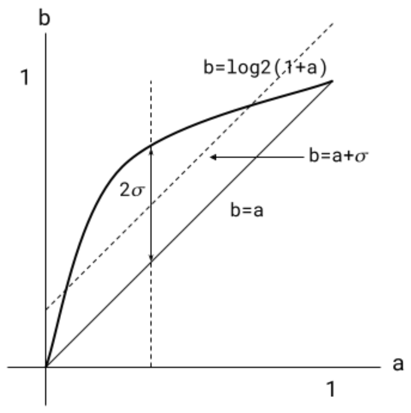
By just thinking about the bits of x in a different way, we've gotten very close to the initial approximation in line (2) of the Quake3 code. Now it's time to improve the guess.

A better log2()

The exaggerated graph to the right shows how $b = log2(1 + a) \approx a$. It suggests an improvement: $log2(x) = log2\left(2^{p} \times 1.f\right) \approx e + f + \sigma$ where σ reduces the maximum deviation from $b = log2(a)$. The best value is $\sigma = 0.04304$. An improved approximate $log2()$ can be had by adding a small nudge. The value of σ is the maximum of

$$f(z) = lo2(1 + z) - z \qquad z \in [0, 1]$$

a simple calculus exercise.



The Quake3 designers did something less obvious. They improved σ by looking at it in the larger context of how it behaved through two steps of Newton's method.

Newton's method

Isaac Newton's name is attached to a powerful technique that emerged in several parts of the world in the 17th century. The method finds points where a function is zero, so we recast our inverse square root problem in this form $f(y) = 1/y^{2} - x = 0$.

The Newton iteration improves a guess with the formula

$$y_2 = y_1 - \frac{f(y_1)}{f'(y_1)}$$

where $f'(y) =- 2y^{-3}$ so

$$y_2 = y_1 - \frac{y_1^{-2} - x}{-2y_1^{-3}} = y_1\left(\frac{3}{2} - \frac{1}{2}xy_1^{2}\right)$$

This is line (4) of the Quake3 code Nothing fancy, just a line of numerics:

```
(4)        y  = y * ( 1.5F - ( 0.5F * x * y * y ) );   // 1st iteration
```

The Quake3 root

The final step of this saga is to refine the initial approximation in line (2):

```
             float ALMOST_Q_sqrt( float x ) {
                 long i;
                 float y;
(1)              i  = * ( long * ) &x;       // evil floating point bit-level hacking
(2)              i  = 0x5f400000 - ( i >> 1 );    // there must be a better constant
(3)              y  = * ( float * ) &i;
(4)        y  = y * ( 1.5F - ( 0.5F * x * y * y ) );   // 1st iteration
                 // y  = y * ( 1.5F - ( 0.5F * x * y * y ) );   skip 2nd iteration
             }
```

For lack of a better term, let's call the value in line (2) the *magic constant.* Improving the $log2()$ approximation helps, but what the Quake3 designers did was find the magic constant by looking at the behavior of `Q_sqrt()` over a range of inputs. Some of the references discuss tools like Matlab, which provide built-in methods to automate such searching.

We went DIY, investigating `Q_sqrt()` over the range `[0.001, 6.0]` with a grid of 4,001 pints. In 13 steps, using binary search, we arrived at `0x5f3759ef`, which differs from the Quake3 value in the 5th bit position from the right.

This magic constant is peculiar to the circumstance of using two steps of the Newton iteration. But after arriving at their magic constant, the Quake3 designers backed off to using just one step of Newton, commenting out the second line. The magic constant for that situation differs 4 bits farther leff of the published value, but it's still very close.

Bit-level hacking not considered evil

Programmers know that the expressive, low-level features of the C language offer unlimited opportunity to get their bits in a knot. Style guides preach simplicity and transparency. Perceived "optimizations" are so often irrelevant – after the bugs get wrung out. Then along comes the reciprocal square root, a vital function at the heart of a game's rendering engine. Every cycle counts, so it's worth looking up a decades-old connection between logarithms and binary floating point encodings. The result was a 4-line code sequence with a bit-level hack talked about for twenty years.

Some ancient history

The guess-and-iterate algorithm here dates back at least to *reciproot* on the Ferranti-Manchester Mk 1 computer. Wikipedia and other articles trace the origin of the Quake 3 algorithm to Cleve Moler and colleagues, who point to notes by William Kahan, emeritus professor of everything computational at U.C. Berkeley.

Kahan saw the algorithm on the Mk 1 in 1953 and he suggests it might date to 1949. He implemented it on the IBM 7090 at University of Toronto in 1962 and on the CDC 6400 at Berkeley in 1969. It migrated back to the PDP-10 at MIT, where it was known as *Kahan's Magic Square Root.* Kahan's square root papers of 1980 and 1999 discuss his improved initial guess, which fed into the Quake 3 code.