Title: The Fast Inverse Square Root Algorithm

Full Name: KOUA Mohamed Anis

Level: 1CP

Group: 3

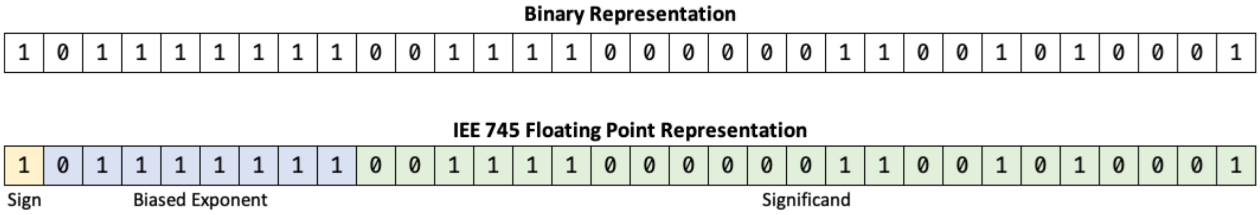
Speciality: CP

Key-words: square root, newton’s method, linear approximations, floating-point, taylor’s theorem

In 2005, game company id released the source code for the iconic 1999 game Quake 3 Arena. Soon after, programmers discovered a tiny algorithm hidden within it—a brilliant routine that was significantly faster than any other similar algorithm. This algorithm computes an approximation of the multiplicative inverse of the square root of x, a computation essential in computer graphics. It is called dozens of times per second, particularly for operations such as vector normalization, character controllers, collision detection, view frustum culling...

If someone asked me to calculate the inverse square root today, I’d probably just use the built-in sqrt function from the math.h header. However, that approach involves floating-point division and computing the square root—usually implemented with Newton iterations, which converge iteratively (often in O(logn) time with respect to the required precision). In contrast, the Fast Inverse Square Root (FISR) algorithm operates in constant time, O(1). So how does it function? and Is it still relevent today?That’s what we’re going to dive into

Before delving into the algorithm's core, it is essential to revisit some fundamental concepts from, computer architecture—particularly, the representation of floating-point numbers. Floating-point representation is a method for encoding real numbers in memory efficiently. Inspired by scientific notation, it expresses numbers as approximations using a sign, a significand (or mantissa), and an exponent within the binary numeral system, allowing for a wide range of values with varying precision, In this article we will be working with 32-bit floats which are stored in memory as follows:



Binary Representation

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

IEE 745 Floating Point Representation

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

-1.2345678

We conclude that a floating-point x is extracted as:

$$x = (-1)^s \cdot (1 + \frac{M}{2^{23}}) \cdot 2^{(E-127)} \in \mathbb{R}$$

Where: s sign-bit, M: Mantissa (23 bits), E: biased exponent (8 bits, offset by 127 to handle negative exponents). A crucial detail is that, before the mantissa, there is always an implicit leading 1, which doesn't need to be stored—saving us 1 bit! Consequently, the bit-level representation of a single-precision floating-point number can be expressed as (Notice that $x > 0$):

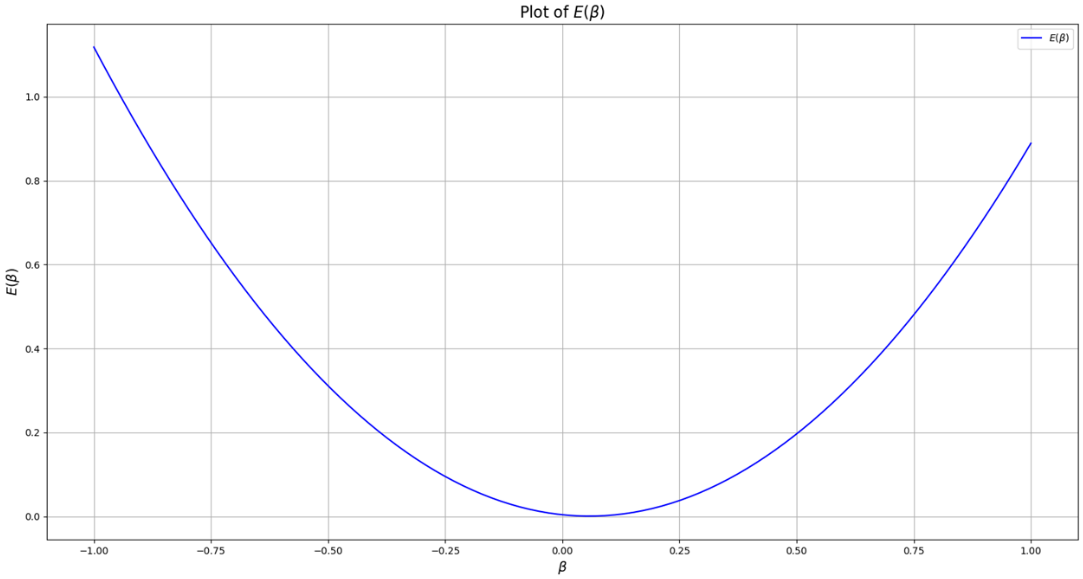
$$I_x = bit(x) = M + E \cdot 2^{23} \in N$$

By establishing a relationship between a number and its bit representation via logarithms and assuming x is positive we obtain:

$$\log_2(x) = \log_2(1 + \frac{M}{2^{23}}) + E - 127$$

To simplify the function $\log_2(1 + \frac{M}{2^{23}})$, we can employ a linear approximation over the inteval $I = [0,1]$. $\log_2(1 + x) \approx \alpha x + \beta$, for simplicity we will let $\alpha = 1$ as it's important for our proof and will make life way much eaiser, one way of finding β is to calculate the he total squared error: $E(\beta) = \int_0^1 (\log_2(x+1) - x - \beta)^2 dx$

We are looking to minimize the function $E(\beta)$. If we plot the graph for $E(\beta)$ we will find it a strictly positive polynomial of degree 2 so the point with the minimal y is the only critical point (0 derivative):



$\frac{dE}{d\beta}(\beta) = \int_0^1 \frac{\partial}{\partial\beta}(\log_2(x+1) - x - \beta)^2\, dx = 0$ (Leibniz integral rule)

$$\beta = \int_0^1 (\log_2(x+1) - x)\, dx \Rightarrow \beta = \log_2(x+1)(x+1) - \frac{x+1}{\log_2(2)} - \frac{x^2}{2}|_0^1 = \frac{3}{2} - \frac{1}{\ln 2}$$

$$\approx 0.05730495911$$

Thus we can express:

$$\log_2(x) = \frac{M}{2^{23}} + E + \beta - 127 = \frac{(M + E \cdot 2^{23})}{2^{23}} + \beta - 127$$

$\log_2(x) = \frac{Ix}{2^{23}} + \beta - 127$ …(A.1)

Let Γ be the approximation of y:

$$\log_2\left(\frac{1}{\sqrt{x}}\right) = -\frac{1}{2}\log_2(x) \approx -\frac{\beta - 127}{2} - \frac{I_x}{2^{24}} = \log_2(\Gamma)$$

Having established this, we now seek the bit-level representation for the value Γ. We can write:

$\log_2(\Gamma) = \frac{I_\Gamma}{2^{23}} + \beta - 127 = -\frac{I_x}{2^{24}} - \frac{\beta - 127}{2}$ return to (A.1)

This leads to the expression (1):

$$I_\Gamma = 3 \cdot (127 - \beta) \cdot 2^{22} - \frac{Ix}{2} = R - \frac{I_x}{2}$$

Let's take a moment to analyze the following result. Instead of relying on complex operation. We are only shifting and offsetting I_x by some constants, to get an intution on why we do that, we need to refer to our result $y = \frac{1}{\sqrt{x}}$, but $x = (1+m)2^e$ where $m = \frac{M}{2^{23}} \in \mathbb{R}$ and $e = E - 127 \in \mathbb{N}$, so $\frac{1}{\sqrt{x}} = \frac{1}{\sqrt{1+m}}2^{-e/2}$, notice that $\frac{1}{\sqrt{1+m}} \approx 1 - \frac{1}{2}m$, so applying a right shift on I_x multiplying by -1 will yield: $-I_x \gg 1 = \frac{-M}{2} - 2^{22}E$ which yields $decode(-I_x \gg 1) = (1 - m/2)2^{-e/2} = \mathit{\Gamma}$

Though some edge cases need to be handled—such as when shifting I_x right by one bit, where the leading bit from the exponent might spill over into the mantissa (Consider E = 1001011 when we right shift E >> 2 = 0100101 + 1 to the mantissa) this is corrected by the R constant. We won't delve into the details for the sake of simplicity, but for a rigorous mathematical analysis, refer to *Chris Lomont's on "Fast Inverse Square Root"*.

This approximation yields an acceptable error but its still noticeable. For that, we can further refine the result by applying a single iteration of Newton's numerical method. Recall that Newton's method is given by

$$\mathit{\Gamma}_{n+1} = \mathit{\Gamma}_n - \frac{f(\mathit{\Gamma}_n)}{f'(\mathit{\Gamma}_n)} \quad n \in N$$

To effectively apply Newton's method, we define a differentiable function that becomes zero once $\mathit{\Gamma} = \frac{1}{\sqrt{x}}$. Although one might initially consider $f(\mathit{\Gamma}) = \mathit{\Gamma} - \frac{1}{\sqrt{x}}$, however if we try to apply the method we'll find $\mathit{\Gamma}_1 = \frac{1}{\sqrt{x}}$ which defeats the hole purspose of the approximation. With some simple algebra we can derive another function that works well, yielding advantagous approximations for the inverse square root function: $f(\mathit{\Gamma}) = \frac{1}{\mathit{\Gamma}^2} - x$. Applying Newton's method with this function gives:

$$\mathit{\Gamma}_1 = \mathit{\Gamma} - \frac{f(\mathit{\Gamma})}{f'(\mathit{\Gamma})} = \mathit{\Gamma} + \mathit{\Gamma}\frac{(1 - x^2\mathit{\Gamma})}{2} = \frac{\mathit{\Gamma}}{2}(3 - x\mathit{\Gamma}^2)$$

This single iteration reduces the error to under 1%.

After establishing the theoretical framework, we are ready to implement our highly efficient algorithm. Initially, we require both the floating-point number x and its corresponding bit-level representation, Ix. A common challenge is to extract the bit representation of x and store it as a 32-bit integer. One common technique is as follows:

```
float x = 5.2F; // Example
int i = *((int*)&x); // Our Trick
```

This method coerces the compiler into interpreting x as a 32-bit integer, thus enabling access to its bit representation. Next, we compute our initial approximation for Γ. According to equation (1), this involves halving the bits of x and applying an offset. The offset is exactly $1597308823=3\cdot(127-\beta)\cdot 2^{22}$, or, more familiarly 0x5F34FF97. Thus, we have:

```
i = 0x5F34FF97 - (i >> 1); // First approximation
```

We then reverse the process to recover the floating-point value from its bit-level representation:

```
x = *((float*)&x);
```

Subsequently, we refine the approximation by applying Newton's method, as delineated in equation (2):

```
x = x * (1.5F - 0.5F * x * x * x);
printf(“%f\n”, x); // Prints 0.438508, real value: 0.4385290097 close enough!!
```

For clarity and enhanced readability, here is the final implementation:

```
float Q_rsqrt(float x) {
   float x2 = 0.5F * x, threeHalfs = 1.5F;
   int i = 0;
   i = *((int*)&x);
   i = 0x5F34FF97 - (i >> 1);
   x = *((float*)&i);
   x = x * (threeHalfs - x2 * x * x);
   return x;
}
```

Before wrapping up, we must address an important question: **Is the Fast Inverse Square Root (FISR) algorithm still relevant today?**

The answer is **yes, but with caveats**. While FISR was groundbreaking in its time, its significance has diminished with advancements in modern hardware. Today's processors feature highly optimized floating-point units (FPUs) capable of computing square roots efficiently. Many compilers and instruction sets now include hardware-accelerated alternatives that either match or outperform FISR in practical applications.

For instance, modern Intel processors leverage the **SSE instruction set**, which includes a dedicated instruction for estimating the reciprocal square root. This hardware-based method is reportedly **four times faster** than FISR. However, what remains unknown is whether Intel's implementation is based on the same principles as FISR or employs an entirely different approach.

Regardless, understanding FISR is still valuable. Beyond its historical significance, it provides deep insight into numerical optimization, low-level computing, and the trade-offs between precision and performance. Now, you are among the few developers who grasp both the **mathematical foundations** and **computational intricacies** of this legendary algorithm—knowledge that remains relevant in specialized domains such as graphics programming, physics simulations, and real-time computing.

BIBLIOGRAPHY

[1]: C. LOMONT, <<FAST INVERSE SQUARE ROOT>>, IEEE 754, Aug 2005.

[2]: https://www.youtube.com/watch?v=Fm0vygzVXeE, 14 Aug 2021

[3]: https://www.youtube.com/watch?v=p8u_k2LIZyo, 28 Nov 2020

[4]: https://www.youtube.com/watch?v=tmb6bLbxd08, 18 Nov 2023

[5]:https://github.com/francisrstokes/githublog/blob/main/2024/5/29/fast-inverse-sqrt.md, 29 May 2024

[6]: A.Micheal, <<Zen of Code Optimization: The Ultimate Guide to Writing Software That Pushes PCs to the Limit>>, USA, Scottsdale, Ariz, 1994