Fast Inverse Square Root

Hassam Uddin

Outline

Representing the Reals

Abusing IEEE-754 for fun and profit

Quake's Fast Inverse-Square-Root

Section 1

Representing the Reals

Bases

Usually, we represent numbers using their bases.

Bases

Usually, we represent numbers using their bases.

- $(425.91)_{10} = 4 \cdot 10^2 + 2 \cdot 10^1 + 5 \cdot 10^0 + 9 \cdot 10^{-1} + 1 \cdot 10^{-2}$

Bases

Usually, we represent numbers using their bases.

- $(425.91)_{10} = 4 \cdot 10^2 + 2 \cdot 10^1 + 5 \cdot 10^0 + 9 \cdot 10^{-1} + 1 \cdot 10^{-2}$
- $(1011.01)_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = (11.25)_{10}$

Bases

Usually, we represent numbers using their bases.

- $(425.91)_{10} = 4 \cdot 10^2 + 2 \cdot 10^1 + 5 \cdot 10^0 + 9 \cdot 10^{-1} + 1 \cdot 10^{-2}$
- $(1011.01)_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = (11.25)_{10}$

In a computer, this has some downsides though.

Fixed-Point Representations

Say we had a fixed 32 bits to represent a decimal number. What are our options?

Fixed-Point Representations

Say we had a fixed 32 bits to represent a decimal number. What are our options?
A few choices:

Fixed-Point Representations

Say we had a fixed 32 bits to represent a decimal number. What are our options?
A few choices:

- 16 bits for the integer portion, 16 bits for the decimal portion. This means we can only represent up to 65535, but have a precision of $\approx 1.5 \cdot 10^{-5}$

Fixed-Point Representations

Say we had a fixed 32 bits to represent a decimal number. What are our options?
A few choices:

- 16 bits for the integer portion, 16 bits for the decimal portion. This means we can only represent up to 65535, but have a precision of $\approx 1.5 \cdot 10^{-5}$
- 24 bits for the integer portion, 8 for the decimal? We can go up to 16777215, but our precision is only ≈ 0.004.

Fixed-Point Representations

Say we had a fixed 32 bits to represent a decimal number. What are our options?
A few choices:

- 16 bits for the integer portion, 16 bits for the decimal portion. This means we can only represent up to 65535, but have a precision of $\approx 1.5 \cdot 10^{-5}$
- 24 bits for the integer portion, 8 for the decimal? We can go up to 16777215, but our precision is only ≈ 0.004.

None of these are particularly ideal, we are either severely limiting the largest number we can represent, or the smallest magnitude of precision we have.

Floating Point Representations

Floating point representations are quite similar to scientific notation.

Floating Point Representations

Floating point representations are quite similar to scientific notation.

- We can represent 37.56 as $3.756 \cdot 10^1$.

Floating Point Representations

Floating point representations are quite similar to scientific notation.

- We can represent 37.56 as $3.756 \cdot 10^1$.
- We can represent $(1011.011)_2$ as $1.011011 \cdot 2^3$.

Floating Point Representations

Floating point representations are quite similar to scientific notation.

- We can represent 37.56 as $3.756 \cdot 10^1$.
- We can represent $(1011.011)_2$ as $1.011011 \cdot 2^3$.
- A nice property of binary is that the first bit of a number in this scientific notation will *always* be 1.

Floating Point Representations

Floating point representations are quite similar to scientific notation.

- We can represent 37.56 as $3.756 \cdot 10^1$.
- We can represent $(1011.011)_2$ as $1.011011 \cdot 2^3$.
- A nice property of binary is that the first bit of a number in this scientific notation will *always* be 1.
- We represent a floating point number x as $\pm q \cdot 2^m$, where m is the exponent, and q is the "significand" of the form $1.f$. We refer to f as the fraction, or mantissa.

Floating Point Representations

Our range of representations is much larger, but we aren't as precise.

Floating Point Representations

Our range of representations is much larger, but we aren't as precise.

- Consider for example $m \in [-4, 4]$ with two bits of the "fraction," giving us 6 total bits for our representation. What are the smallest and largest values we can represent?
- **NOTE**: For simplicity, although we can represent 15 values with 4 bits in the exponent, we're limiting it to 8 (between -4 and 4).

Floating Point Representations

Our range of representations is much larger, but we aren't as precise.

- Consider for example $m \in [-4, 4]$ with two bits of the "fraction," giving us 6 total bits for our representation. What are the smallest and largest values we can represent?
- **NOTE**: For simplicity, although we can represent 15 values with 4 bits in the exponent, we're limiting it to 8 (between -4 and 4).
- $x = 1.b_1 b_2 \cdot 2^m$: Smallest is $(1.00)_2^{-4} = 0.0625$, and the largest is $(1.11)_2 \cdot 2^4 = 28$.

Floating Point Representations

Our range of representations is much larger, but we aren't as precise.

- Consider for example $m \in [-4, 4]$ with two bits of the "fraction," giving us 6 total bits for our representation. What are the smallest and largest values we can represent?
- **NOTE**: For simplicity, although we can represent 15 values with 4 bits in the exponent, we're limiting it to 8 (between -4 and 4).
- $x = 1.b_1 b_2 \cdot 2^m$: Smallest is $(1.00)_2^{-4} = 0.0625$, and the largest is $(1.11)_2 \cdot 2^4 = 28$.
- We can represent much larger values with the same number of bits as a fixed-point, regardless of how we split the fixed-point, while still being able to represent smaller values as well. The only downside is, we're not as precise. How do we represent 27.0?

Floating Point Representations

Our range of representations is much larger, but we aren't as precise.

- Consider for example $m \in [-4, 4]$ with two bits of the "fraction," giving us 6 total bits for our representation. What are the smallest and largest values we can represent?
- **NOTE**: For simplicity, although we can represent 15 values with 4 bits in the exponent, we're limiting it to 8 (between -4 and 4).
- $x = 1.b_1 b_2 \cdot 2^m$: Smallest is $(1.00)_2^{-4} = 0.0625$, and the largest is $(1.11)_2 \cdot 2^4 = 28$.
- We can represent much larger values with the same number of bits as a fixed-point, regardless of how we split the fixed-point, while still being able to represent smaller values as well. The only downside is, we're not as precise. How do we represent 27.0?

We cannot represent 27, we're stuck approximating it as 28 or 24.

IEEE-754 Single Precision

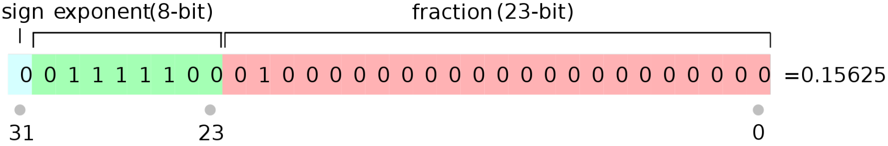
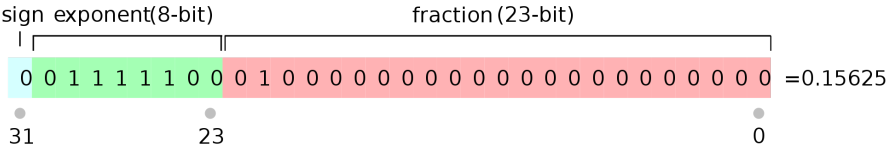


Figure: ~~Stolen~~ Borrowed from CS 357 Notes

IEEE-754 is *very* similar to a floating point representation but with a few tweaks.

$$x = (-1)^s 1.f \cdot 2^m$$

Down to the bits



- We use 1 bit for the sign, s, leaving us 31 bits.
- We use 8 bits for the exponent, giving us 255 possible exponents. We write $m = c - 127$, where c is the actual exponent stored in the binary representation. We also reserve $c = 0$ and $c = 255$ for special cases. The largest exponent is 127 and the smallest exponent is -126.
- The remaining 23 bits are the fractional part, also known as the mantissa.

$$x = (-1)^s 1.f \cdot 2^m$$

Special Cases

- Zero?

Special Cases

- Zero? We make all the bits in the exponent and the mantissa, or fraction, 0 to represent 0. Since we don't impact the sign, this means that floats can be -0 or $+0$.

Special Cases

- Zero? We make all the bits in the exponent and the mantissa, or fraction, 0 to represent 0. Since we don't impact the sign, this means that floats can be -0 or $+0$.
- Infinity?

Special Cases

- Zero? We make all the bits in the exponent and the mantissa, or fraction, 0 to represent 0. Since we don't impact the sign, this means that floats can be -0 or $+0$.
- Infinity? If a number is outside our range, we store it as infinity, which we represent as 255 in the exponent, or by setting all the bits to 1. We leave the mantissa set to all 0s.

Special Cases

- Zero? We make all the bits in the exponent and the mantissa, or fraction, 0 to represent 0. Since we don't impact the sign, this means that floats can be -0 or $+0$.
- Infinity? If a number is outside our range, we store it as infinity, which we represent as 255 in the exponent, or by setting all the bits to 1. We leave the mantissa set to all 0s.
- NaN?

Special Cases

- Zero? We make all the bits in the exponent and the mantissa, or fraction, 0 to represent 0. Since we don't impact the sign, this means that floats can be -0 or $+0$.
- Infinity? If a number is outside our range, we store it as infinity, which we represent as 255 in the exponent, or by setting all the bits to 1. We leave the mantissa set to all 0s.
- NaN? We set all 1s in the exponent again, but make the mantissa non-zero.

Rounding values is an important consideration in most cases, take CS 357 (or just watch the two lectures associated with floating point numbers) to understand how we use floating points and how we should be careful with them.

Matching bits

What happens if we take the bits in a floating point number and just "pretend" that it's an integer? What if we do the opposite?

Matching bits

What happens if we take the bits in a floating point number and just "pretend" that it's an integer? What if we do the opposite?

Let us ignore the sign bit for a moment: $x = 1.f \cdot 2^{c-127}$. Reinterpreting these bits as an integer, we get $x_{int} = c \cdot 2^{23} + f$.

Matching bits

What happens if we take the bits in a floating point number and just "pretend" that it's an integer? What if we do the opposite?

Let us ignore the sign bit for a moment: $x = 1.f \cdot 2^{c-127}$. Reinterpreting these bits as an integer, we get $x_{int} = c \cdot 2^{23} + f$.

Converting an integer to a float is less clean, so I'll leave that to you.

Section 2

Abusing IEEE-754 for fun and profit

Fast Logarithms

- Let's take the logarithm of our float representation.

Fast Logarithms

- Let's take the logarithm of our float representation.
- If we consider the mantissa and the exponent as integers, we can write $x_{float} = (1 + \frac{f}{2^{23}}) \cdot 2^{c-127}$. If we take the logarithm of this:

$$\log_2((1 + \frac{f}{2^{23}}) \cdot 2^{c-127}) = \log_2(1 + \frac{f}{2^{23}}) + c - 127$$

Fast Logarithms

- Let's take the logarithm of our float representation.
- If we consider the mantissa and the exponent as integers, we can write $x_{float} = (1 + \frac{f}{2^{23}}) \cdot 2^{c-127}$. If we take the logarithm of this:

$$\log_2((1 + \frac{f}{2^{23}}) \cdot 2^{c-127}) = \log_2(1 + \frac{f}{2^{23}}) + c - 127$$

- Recall that we can approximate $\log(1 + x) \approx \log(x)$, and we can add an error correction factor μ, to make our approximation even tighter.

Fast Logarithms

- Let's take the logarithm of our float representation.
- If we consider the mantissa and the exponent as integers, we can write $x_{float} = (1 + \frac{f}{2^{23}}) \cdot 2^{c-127}$. If we take the logarithm of this:

$$\log_2((1 + \frac{f}{2^{23}}) \cdot 2^{c-127}) = \log_2(1 + \frac{f}{2^{23}}) + c - 127$$

- Recall that we can approximate $\log(1 + x) \approx \log(x)$, and we can add an error correction factor μ, to make our approximation even tighter.
- So our log is now:

$$\frac{f}{2^{23}} + \mu + c - 127 = \frac{1}{2^{23}}(f + c \cdot 2^{23}) + \mu - 127$$

Hmm

- Something suspicious has appeared. Our logarithm is of the form $k_1(f + c \cdot 2^{23}) + k_2$, where k_1 and k_2 are constants.

Hmm

- Something suspicious has appeared. Our logarithm is of the form $k_1(f + c \cdot 2^{23}) + k_2$, where k_1 and k_2 are constants.
- Recall, however, that the integer reinterpretation of our floating point number is $f + c \cdot 2^{23}$.

Hmm

- Something suspicious has appeared. Our logarithm is of the form $k_1(f + c \cdot 2^{23}) + k_2$, where k_1 and k_2 are constants.
- Recall, however, that the integer reinterpretation of our floating point number is $f + c \cdot 2^{23}$.
- We can approximate $\log_2(x_{float})$ as a linear transformation of x_{int}. No computation needed!

Hmm

- Something suspicious has appeared. Our logarithm is of the form $k_1(f + c \cdot 2^{23}) + k_2$, where k_1 and k_2 are constants.
- Recall, however, that the integer reinterpretation of our floating point number is $f + c \cdot 2^{23}$.
- We can approximate $\log_2(x_{float})$ as a linear transformation of x_{int}. No computation needed!
- How easy would it be to go from $\log_2(x_{float})$ back to the regular number?

Hmm

- Something suspicious has appeared. Our logarithm is of the form $k_1(f + c \cdot 2^{23}) + k_2$, where k_1 and k_2 are constants.
- Recall, however, that the integer reinterpretation of our floating point number is $f + c \cdot 2^{23}$.
- We can approximate $\log_2(x_{float})$ as a linear transformation of x_{int}. No computation needed!
- How easy would it be to go from $\log_2(x_{float})$ back to the regular number?
- We just undo the linear transform: we've gotten all the log properties for free!

Square root a log-approximated number

- Let's take the square root of x_{float} by abusing these properties.

Square root a log-approximated number

- Let's take the square root of x_{float} by abusing these properties.
- First, recall that $k \log(x) = \log\left(x^k\right)$, so $x_{float}^{1/2} = 2^{1/2 \cdot \log_2(x_{float})}$.

Square root a log-approximated number

- Let's take the square root of x_{float} by abusing these properties.
- First, recall that $k \log(x) = \log\left(x^k\right)$, so $x_{float}^{1/2} = 2^{1/2 \cdot \log_2(x_{float})}$.
- So, we can compute $2^{1/2 \cdot (k_1 x_{int} + k_2)}$

Square root a log-approximated number

- Let's take the square root of x_{float} by abusing these properties.
- First, recall that $k \log(x) = \log\left(x^k\right)$, so $x_{float}^{1/2} = 2^{1/2 \cdot \log_2(x_{float})}$.
- So, we can compute $2^{1/2 \cdot (k_1 x_{int} + k_2)}$
- How do we find our constants k_1, k_2, and do this computation quickly?

Square root a log-approximated number

- Let's take the square root of x_{float} by abusing these properties.
- First, recall that $k \log(x) = \log\left(x^k\right)$, so $x_{float}^{1/2} = 2^{1/2 \cdot \log_2(x_{float})}$.
- So, we can compute $2^{1/2 \cdot (k_1 x_{int} + k_2)}$
- How do we find our constants k_1, k_2, and do this computation quickly?
- Let's detour into taking the *inverse* square root

Section 3

Quake's Fast Inverse-Square-Root

A detour into history

```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;
}
```

Modernize

```
constexpr float Q_rsqrt(float number) noexcept
{
  // only allow on IEEE-754 floats
  static_assert(std::numeric_limits<float>::is_iec559);

  // what the fuck? (left for historical accuracy)
  // make use of std::bit_cast to avoid undefined behavior
  float const y = std::bit_cast<float>(
    0x5f3759df - (std::bit_cast<std::uint32_t>(number) >> 1));
  return y * (1.5f - (number * 0.5f * y * y));
}
```

evil floating point bit level hacking

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

or

```
std::bit_cast<std::uint32_t>(number)
```

what the fuck?

```
i  = 0x5f3759df - ( i >> 1 );
```

what the fuck?

```
i  = 0x5f3759df - ( i >> 1 );
```

- Recall that $-\frac{1}{2}\log(y) = \log\left(\frac{1}{\sqrt{y}}\right)$

what the fuck?

```
i  = 0x5f3759df - ( i >> 1 );
```

- Recall that $-\frac{1}{2}\log(y) = \log\left(\frac{1}{\sqrt{y}}\right)$
- Call $\frac{1}{\sqrt{y}} = Y$, and let us substitute the bit representation of each in place of their log:

what the fuck?

```
i  = 0x5f3759df - ( i >> 1 );
```

- Recall that $-\frac{1}{2}\log(y) = \log\left(\frac{1}{\sqrt{y}}\right)$
- Call $\frac{1}{\sqrt{y}} = Y$, and let us substitute the bit representation of each in place of their log:

$$\frac{1}{2^{23}}(f_Y + c_Y \cdot 2^{23}) + \mu - 127 = -\frac{1}{2}\left(\frac{1}{2^{23}}(f_y + c_y \cdot 2^{23}) + \mu - 127\right)$$

- Let's solve for the bit representation of Y: $f_Y + c_Y \cdot 2^{23}$:

what the fuck?

```
i  = 0x5f3759df - ( i >> 1 );
```

- Recall that $-\frac{1}{2}\log(y) = \log\left(\frac{1}{\sqrt{y}}\right)$
- Call $\frac{1}{\sqrt{y}} = Y$, and let us substitute the bit representation of each in place of their log:

$$\frac{1}{2^{23}}(f_Y + c_Y \cdot 2^{23}) + \mu - 127 = -\frac{1}{2}\left(\frac{1}{2^{23}}(f_y + c_y \cdot 2^{23}) + \mu - 127\right)$$

- Let's solve for the bit representation of Y: $f_Y + c_Y \cdot 2^{23}$:

$$f_Y + c_Y \cdot 2^{23} = \frac{3}{2}2^{23}(127 - \mu) - \frac{1}{2}(f_y + c_y \cdot 2^{23})$$

0x5f3759df?

```
i  = 0x5f3759df - ( i >> 1 );
```

$$f_Y + c_Y \cdot 2^{23} = \frac{3}{2} 2^{23}(127 - \mu) - \frac{1}{2}(f_y + c_y \cdot 2^{23})$$

0x5f3759df?

```
i  = 0x5f3759df - ( i >> 1 );
```

$$f_Y + c_Y \cdot 2^{23} = \frac{3}{2} 2^{23} (127 - \mu) - \frac{1}{2} (f_y + c_y \cdot 2^{23})$$

0x5f3759df?

```
i  = 0x5f3759df - ( i >> 1 );
```

$$f_Y + c_Y \cdot 2^{23} = \frac{3}{2} 2^{23}(127 - \mu) - \frac{1}{2}(f_y + c_y \cdot 2^{23})$$

0x5f3759df?

```
i  = 0x5f3759df - ( i >> 1 );
```

$$f_Y + c_Y \cdot 2^{23} = \frac{3}{2} 2^{23}(127 - \mu) - \frac{1}{2}(f_y + c_y \cdot 2^{23})$$

- How did we choose the “magic constant” μ?

0x5f3759df?

```
i  = 0x5f3759df - ( i >> 1 );
```

$$f_Y + c_Y \cdot 2^{23} = \frac{3}{2} 2^{23}(127 - \mu) - \frac{1}{2}(f_y + c_y \cdot 2^{23})$$

- How did we choose the "magic constant" μ?
- Historically, it's unknown, and the choice of constant used in Quake is actually not optimal.

0x5f3759df?

```
i  = 0x5f3759df - ( i >> 1 );
```

$$f_Y + c_Y \cdot 2^{23} = \frac{3}{2} 2^{23}(127 - \mu) - \frac{1}{2}(f_y + c_y \cdot 2^{23})$$

- How did we choose the "magic constant" μ?
- Historically, it's unknown, and the choice of constant used in Quake is actually not optimal.
- If you were doing this in your own program, plot the error and minimize.

Casting back

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

or

```
float const y = std::bit_cast<float>(...);
```

Casting back

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

or

```
float const y = std::bit_cast<float>(...);
```

- Are we done?

Casting back

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

or

```
float const y = std::bit_cast<float>(...);
```

- Are we done?
- We are quite close, but we've introduced a decent amount of error in our assumptions.

Newton's Method, another detour

The goal: find c such that $f(c) = 0$

Newton's Method, another detour

The goal: find c such that $f(c) = 0$

- We want to make a guess that is close to c

Newton's Method, another detour

The goal: find c such that $f(c) = 0$

- We want to make a guess that is close to c
- Find the tangent line and solve for its 0:
  $0 = f(x_0) + f'(x_0)(x_1 - x_0) \implies x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$

Newton's Method, another detour

The goal: find c such that $f(c) = 0$

- We want to make a guess that is close to c
- Find the tangent line and solve for its 0:
  $0 = f(x_0) + f'(x_0)(x_1 - x_0) \implies x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$
- Repeat until we're happy

Newton's Method, another detour

The goal: find c such that $f(c) = 0$

- We want to make a guess that is close to c
- Find the tangent line and solve for its 0:
  $0 = f(x_0) + f'(x_0)(x_1 - x_0) \implies x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$
- Repeat until we're happy

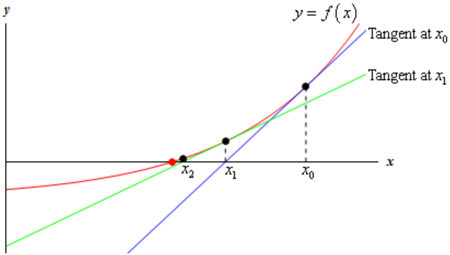


Figure: Paul's Math Notes

Newton's method, on inverse square root

- We want to find $\frac{1}{\sqrt{x}}$, so minimize $\mathrm{error}(y) = \frac{1}{y^2} - x$
- Plugging into Newton's method, we have:

$$y_1 = y_0 - \frac{{y_0}^{-2} - x}{-2{y_0}^{-3}} = \frac{1}{2} y_0 (3 - x y_0^2)$$

Another look

$$\frac{1}{2}y_0(3 - xy_0^2)$$

```cpp
constexpr float Q_rsqrt(float number) noexcept
{
  // only allow on IEEE-754 floats
  static_assert(std::numeric_limits<float>::is_iec559);

  // what the fuck? (left for historical accuracy)
  // make use of std::bit_cast to avoid undefined behavior
  float const y = std::bit_cast<float>(
    0x5f3759df - (std::bit_cast<std::uint32_t>(number) >> 1));
  return y * (1.5f - (number * 0.5f * y * y));
}
```

Does the fun stop here?

- The inverse square root does not have any divisions, so it is “fast.”

Does the fun stop here?

- The inverse square root does not have any divisions, so it is "fast."
- Quake uses this for the inverse square root because taking the inverse square root of a vector's length is a common operation to normalize a vector.

Does the fun stop here?

- The inverse square root does not have any divisions, so it is "fast."
- Quake uses this for the inverse square root because taking the inverse square root of a vector's length is a common operation to normalize a vector.
- We can approximate a *lot* of functions using this approach while avoiding any divisions.

ECE Majors strike again

- Unfortunately, we're not allowed to have fun in a world with hardware engineers.

ECE Majors strike again

- Unfortunately, we're not allowed to have fun in a world with hardware engineers.
- Intel SSE (found on any computer made after 1999), has the RSQRTSS instruction.

ECE Majors strike again

- Unfortunately, we're not allowed to have fun in a world with hardware engineers.
- Intel SSE (found on any computer made after 1999), has the RSQRTSS instruction.

```cpp
#include <bit>
#include <limits>
#include <cstdint>
#include <cmath>

float Q_rsqrt(float number) noexcept
{
  static_assert(std::numeric_limits<float>::is_iec559);

  float const y = std::bit_cast<float>(
    0x5f3759df - (std::bit_cast<std::uint32_t>(number) >> 1));
  return y * (1.5f - (number * 0.5f * y * y));
}

float inverse_sqrt(float f) {
    return 1 / sqrtf(f);
}
```

x86-64 gcc (trunk) (Editor #1)

x86-64 gcc (trunk) -O2 -ffast-math -std=c++20

A Output... Filter... Libraries Overrides + Add new... Add tool...

```asm
Q_rsqrt(float):
        movd    edx, xmm0
        mov     eax, 1597463007
        mulss   xmm0, DWORD PTR .LC0[rip]
        shr     edx
        sub     eax, edx
        movd    xmm2, eax
        movaps  xmm1, xmm2
        mulss   xmm1, xmm2
        mulss   xmm0, xmm1
        movss   xmm1, DWORD PTR .LC1[rip]
        subss   xmm1, xmm0
        mulss   xmm1, xmm2
        movaps  xmm0, xmm1
        ret
inverse_sqrt(float):
        movaps  xmm1, xmm0
        rsqrtss xmm1, xmm1
        mulss   xmm0, xmm1
        mulss   xmm0, xmm1
        mulss   xmm1, DWORD PTR .LC3[rip]
        addss   xmm0, DWORD PTR .LC2[rip]
        mulss   xmm0, xmm1
        ret
```

All is not lost

- Inverse square root is such a common operation that it is built into modern hardware

All is not lost

- Inverse square root is such a common operation that it is built into modern hardware
- But, keep in mind, when you're doing any computation, logs and powers are just a cast and linear transformation away.

Questions?

Truth is much too complicated to allow anything but approximations.

— John Von Nuemann (1947)