Bits and Pieces

A breakdown of the digits

Ciara

```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;
}
```

'Fast' inverse square root funtion

Root-finding problem

- Current problem: $x = \frac{1}{\sqrt{input}}$
- We want: find x where $f(x) = 0$

Root-finding problem

- Current problem: $x = \frac{1}{\sqrt{input}}$
- We want: find x where $f(x) = 0$

Rearrange

$$x^2 = \frac{1}{input}$$

Root-finding problem

- Current problem: $x = \frac{1}{\sqrt{input}}$
- We want: find x where $f(x) = 0$

Rearrange

$x^2 = \frac{1}{input}$

$0 = \frac{1}{x^2} - input$

Root-finding problem

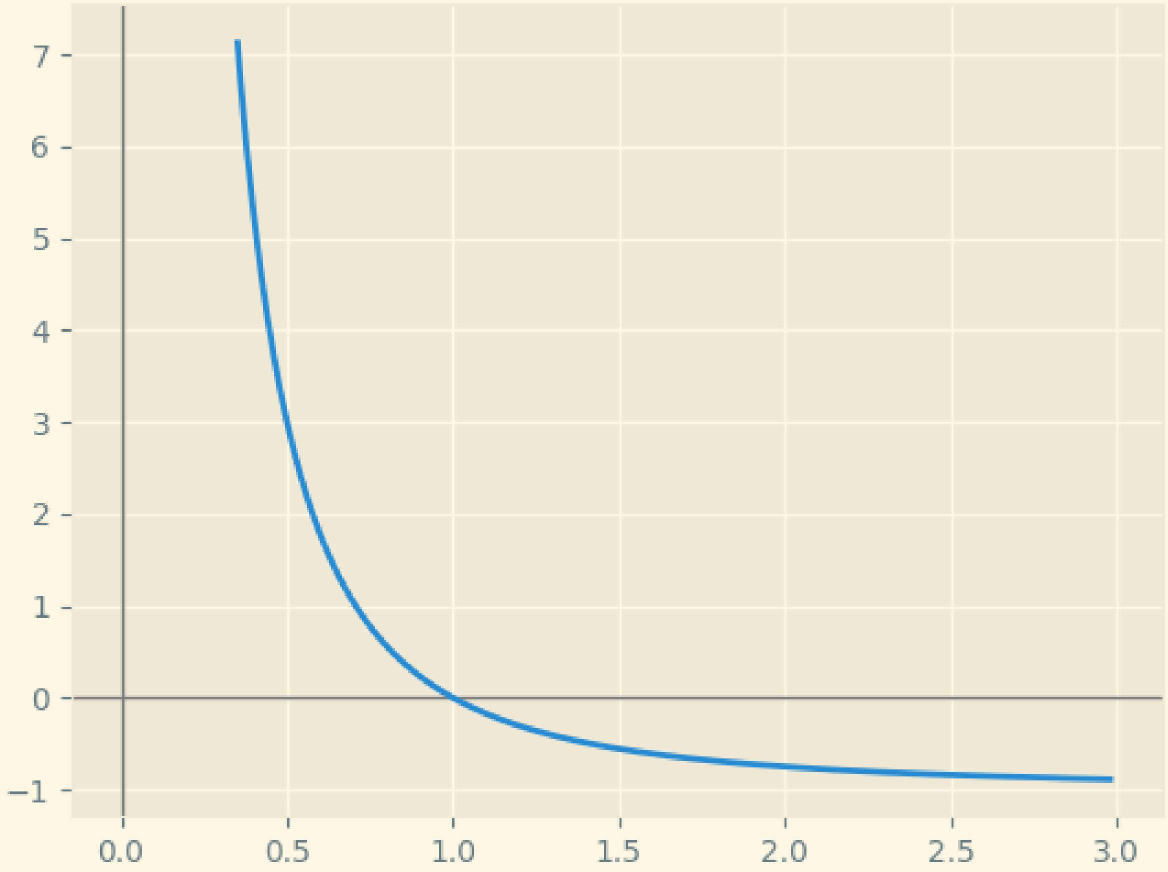
- Current problem: $x = \frac{1}{\sqrt{input}}$
- We want: find x where $f(x) = 0$

Rearrange

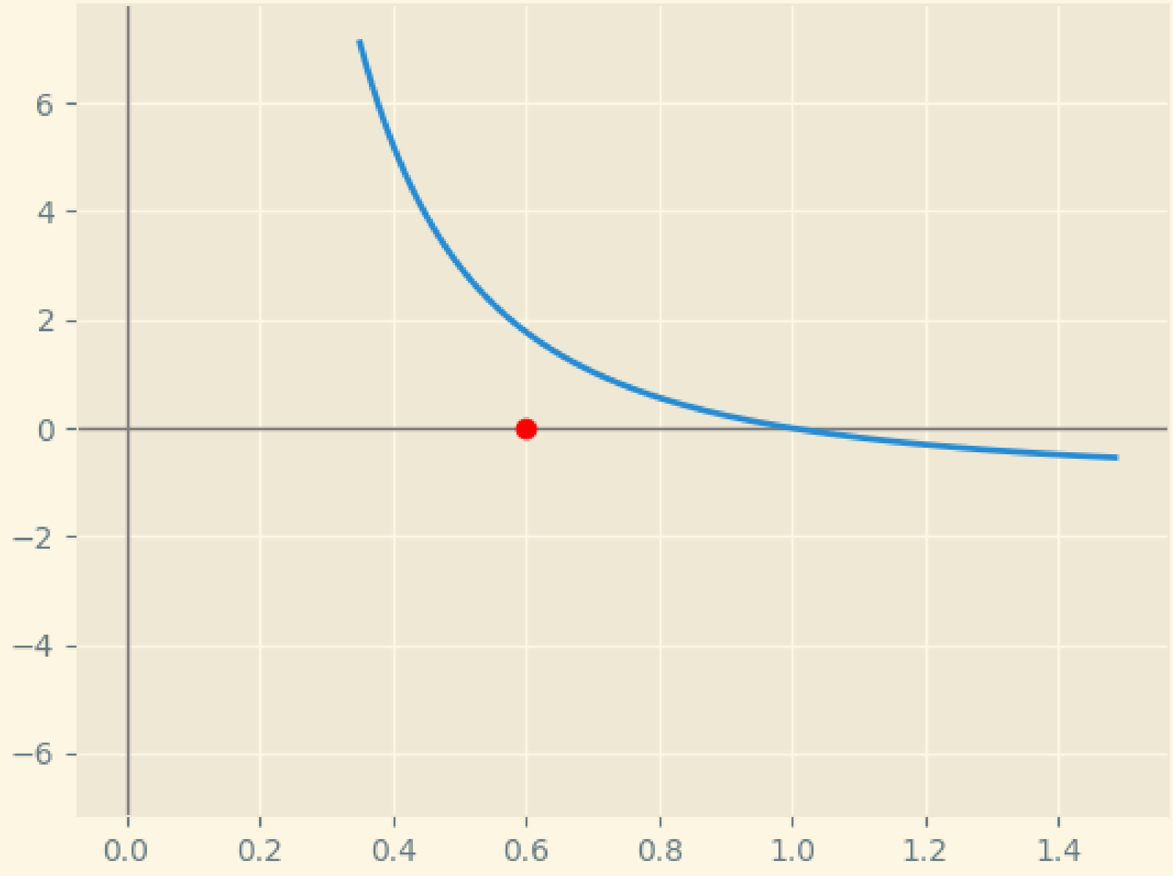
$$x^2 = \frac{1}{input}$$

$$\boxed{f(x) = \frac{1}{x^2} - input}$$

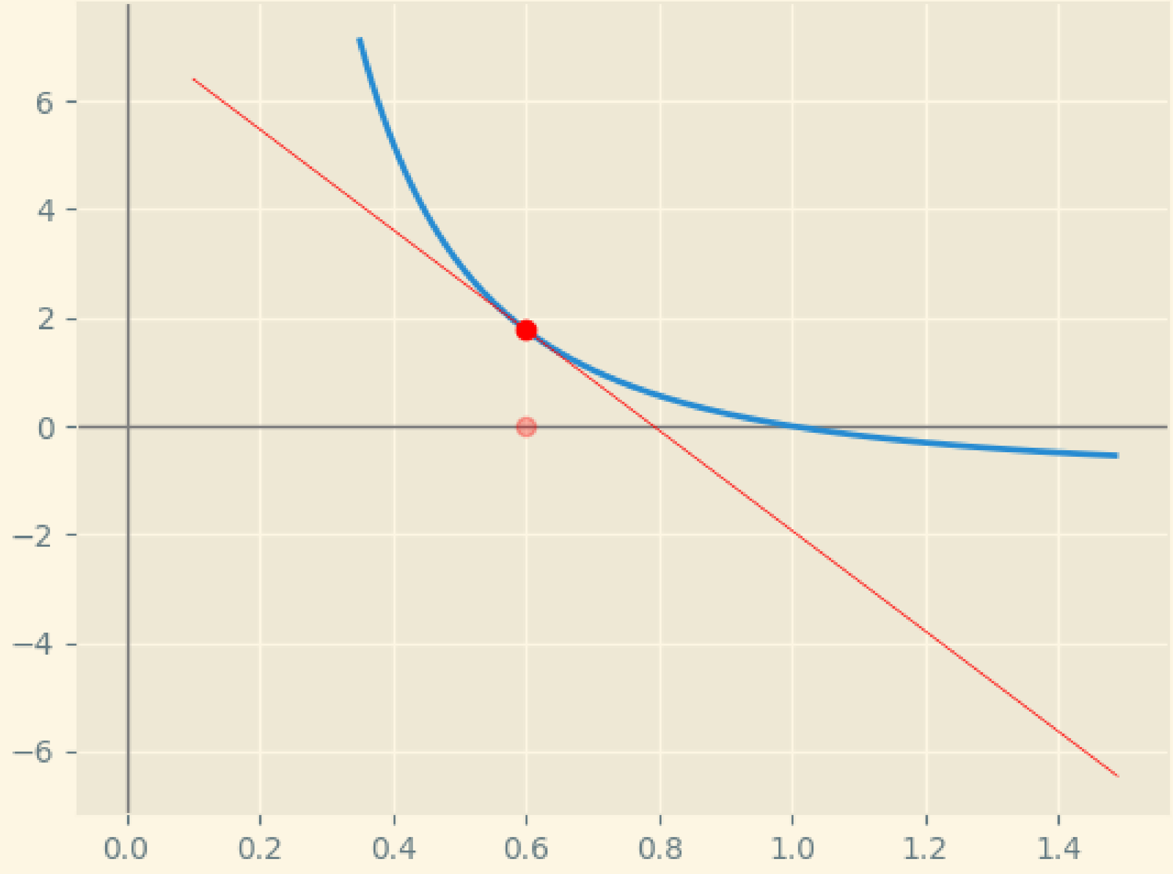
Newton Raphson



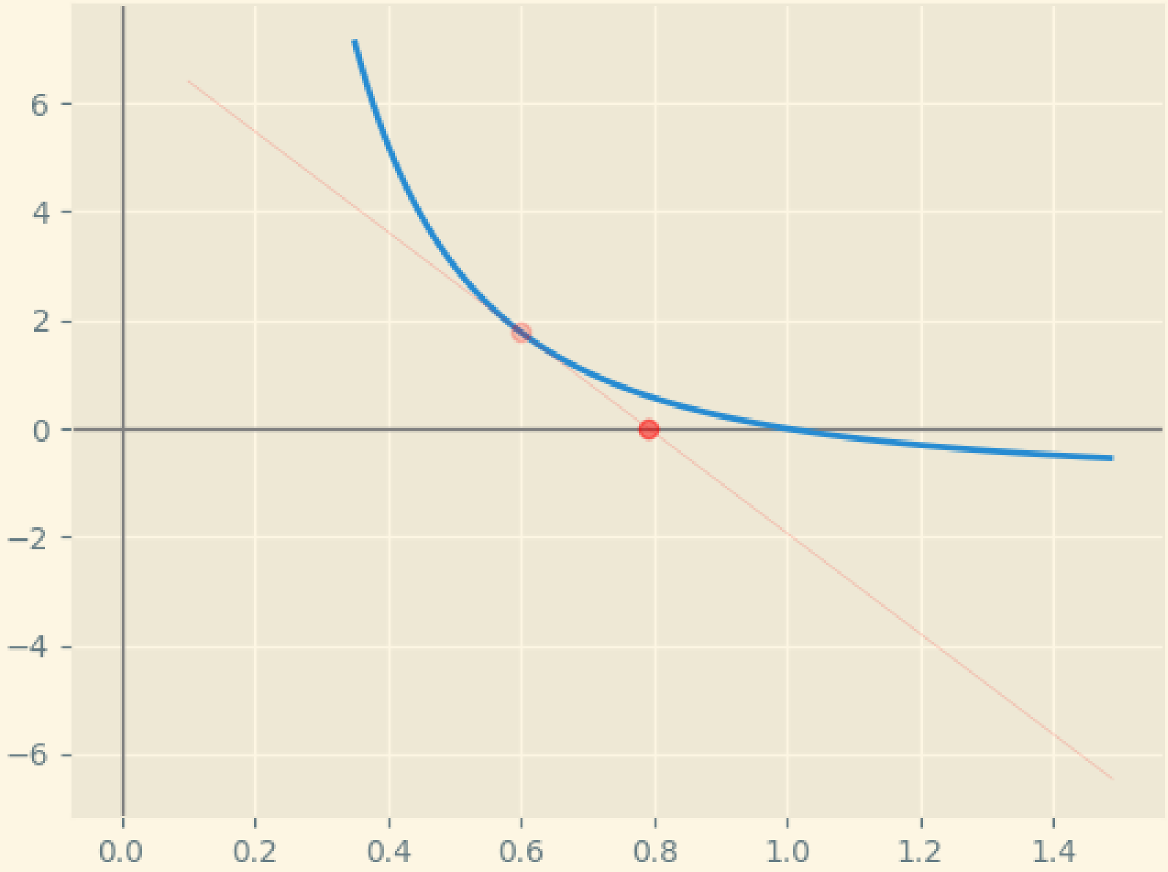
Newton Raphson



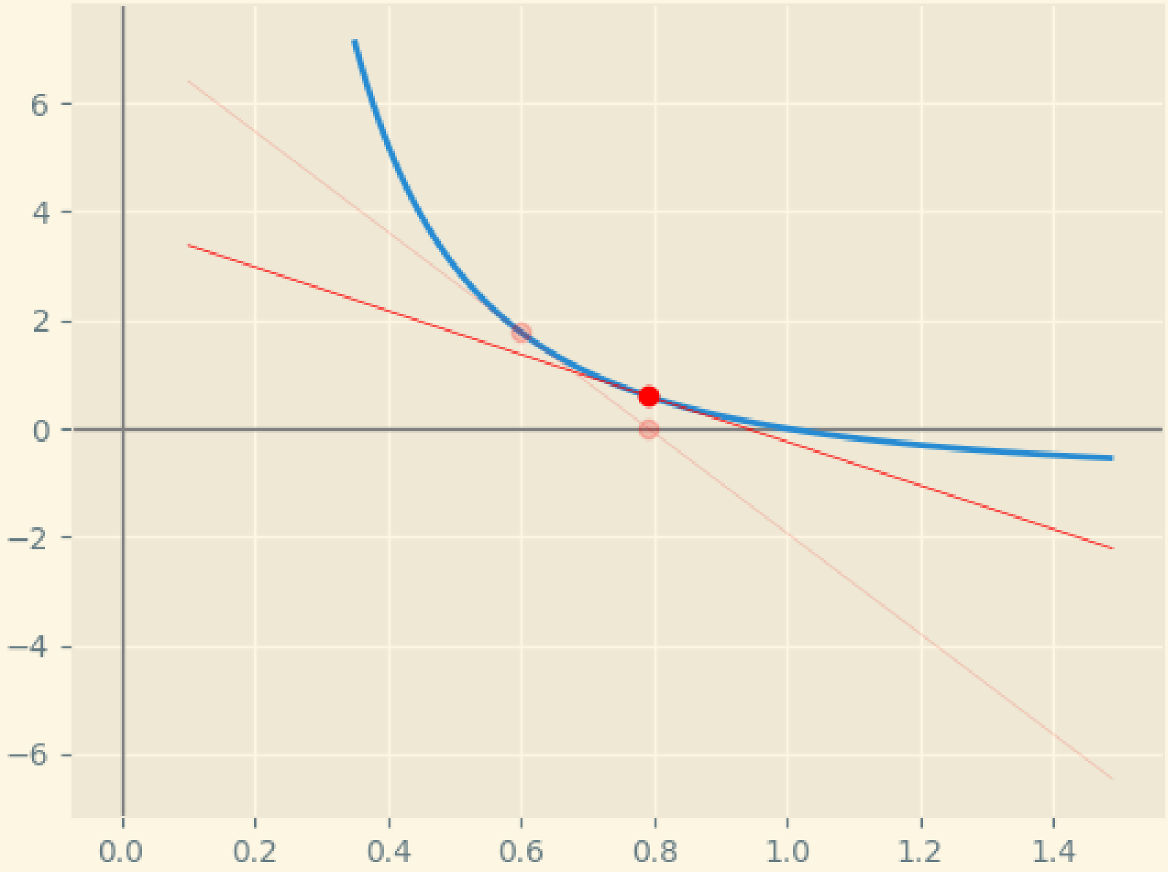
Newton Raphson



Newton Raphson



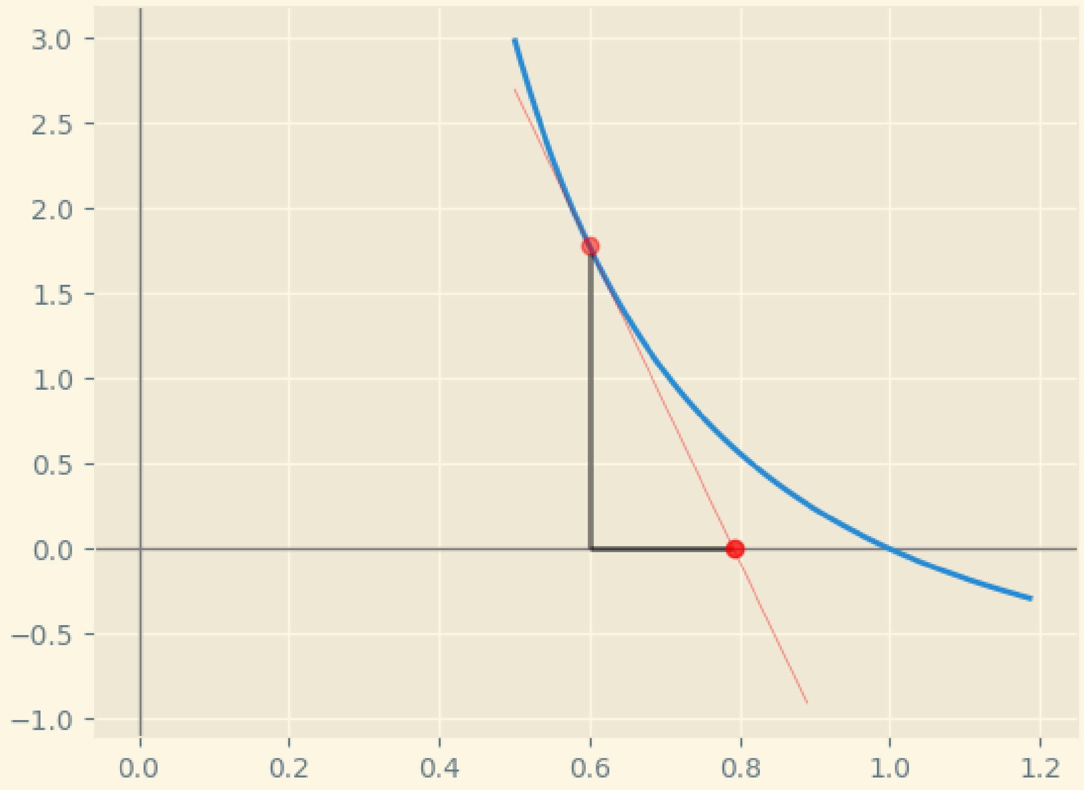
Newton Raphson



Newton Raphson

Iteratively

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$



Newton Raphson

$$f(x) = \frac{1}{x^2} - input$$

$$f(x) = x^{-2} - input$$

$$\boxed{f'(x) = -2x^{-3}}$$

Newton Raphson

$$f(x) = x^{-2} - input$$

$$f'(x) = -2x^{-3}$$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

$$\boxed{x_1 = x_0 - \frac{x_0^{-2} - input}{-2x_0^{-3}}}$$

Newton Raphson

Rearrange

$$x_1 = x_0 - \frac{x_0^{-2} - input}{-2x_0^{-3}}$$

$$x_1 = x_0 + \frac{1}{2}x_0 - \frac{1}{2}input * x_0^3$$

$$x_1 = \frac{3}{2}x_0 - \frac{1}{2}input * x_0^3$$

Newton Raphson

Rearrange

$$x_1 = \frac{3}{2}x_0 - \frac{1}{2}input * x_0^3$$

$$\boxed{x_1 = x_0 * \left(\frac{3}{2} - \frac{1}{2}input * x_0^2\right)}$$

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

- $y \longrightarrow x_0$ and $x_2 \longrightarrow \frac{1}{2}input$

Getting our initial estimate

Bitwise Operations

Bitwise Operations

&	AND operator
|	OR operator
^	XOR operator
~	NOT operator (one's complement)
<<	Left shift operator
>>	Right shift operator

Right shift on unsigned integers

$$6$$

```
0000110
```

$$6 >> 1 \longrightarrow 3$$

```
0000011
```

IEEE Floats

Scientific notation

223	$2.23 * 10^2$
0.03	$3.0 * 10^{-2}$

Base 2

12	$1.5 * 2^3$
0.875	$1.75 * 2^{-1}$

Base 2

12	$(1 + 0.5) * 2^3$
0.875	$(1 + 0.75) * 2^{-1}$

$$-1^{sign} * (1 + mantissa) * 2^{exponent}$$

Base 2

12	$(1+0.5)*2^3$
0.875	$(1+0.75)*2^{-1}$

$$-1^{sign}*(1+mantissa)*2^{exponent}$$

Sign	Exp. bits	Fraction (mantissa) bits
0	0000 0000	0000 0000 0000 0000 0000 000

Value	Sign	Exp. bits	Fraction (mantissa) bits
0	0	0000 0000	0000 0000 0000 0000 0000 000
$1.0 * 2^{-126}$	0	0000 0001	0000 0000 0000 0000 0000 000
$1.0 * 2^{0}$	0	0111 1111	0000 0000 0000 0000 0000 000
$1.0 * 2^{1}$	0	1000 0000	0000 0000 0000 0000 0000 000
$1.0 * 2^{127}$	0	1111 1110	0000 0000 0000 0000 0000 000
inf	0	1111 1111	0000 0000 0000 0000 0000 000
NaN	0	1111 1111	1000 0000 0000 0000 0000 000

$$-1^{sign} * (1 + mantissa) * 2^{exponent}$$

Value	Sign	Exp. bits (E)	Fraction (mantissa) bits (M)
$1.5 * 2^3$	0	1000 0010	1000 0000 0000 0000 0000 000

e = exponent
m = mantissa

E = exponent bits as integer
M = mantissa bits as integer

B = bias (127)
L = smallest fractional part (2^{23})

Value	Sign	Exp. bits (E)	Fraction (mantissa) bits (M)
$1.5 * 2^3$	0	1000 0010	1000 0000 0000 0000 0000 000

$e = E - B$ where $B = 127$

$m = \frac{M}{L}$ where $L = 2^{23}$

$value = (1 + m)2^e$

Value	Sign	Exp. bits (E)	Fraction (mantissa) bits (M)
$1.5 * 2^3$	0	1000 0010	1000 0000 0000 0000 0000 000

- e.g. value 12 = $(1 + 0.5) * 2^3$
  - $e = 3$,
    - $e = E - 127$ so $E = 130$
  - $m = 0.5$,
    - $m = \frac{M}{2^{23}}$ so $M = 2^{22}$

One final piece of notation

Value	Sign	Exp. bits (E)	Fraction (mantissa) bits (M)
$1.5 * 2^3$	0	1000 0010	1000 0000 0000 0000 0000 000

Float bits as integer

$$\boxed{I = M + LE}$$

Scale up the exponential bits by L (2^{23}), assuming positive

Getting an estimate

Reintepret bits as long *(use uint32_t these days)*

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

Clever bit trickery

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

- Assumptions
  - Assume input normal positive value - not zero, NaN or +/-inf

Thinking in terms of logs

$$result = \frac{1}{\sqrt{input}} = input^{-\frac{1}{2}}$$

$$\log_2(result) = -\frac{1}{2}\log_2(input)$$

Rewrite and use log rules

$$\log_2(result) = -\frac{1}{2}\log_2(input)$$

- $result = (1 + m_{result}) * 2^{e_{result}}$
- $input = (1 + m_{input}) * 2^{e_{input}}$

Apply log rules

$$\log_2(1 + m_{result}) + e_{result} = -\frac{1}{2}(\log_2(1 + m_{input}) + e_{input})$$

Getting rid of the logs

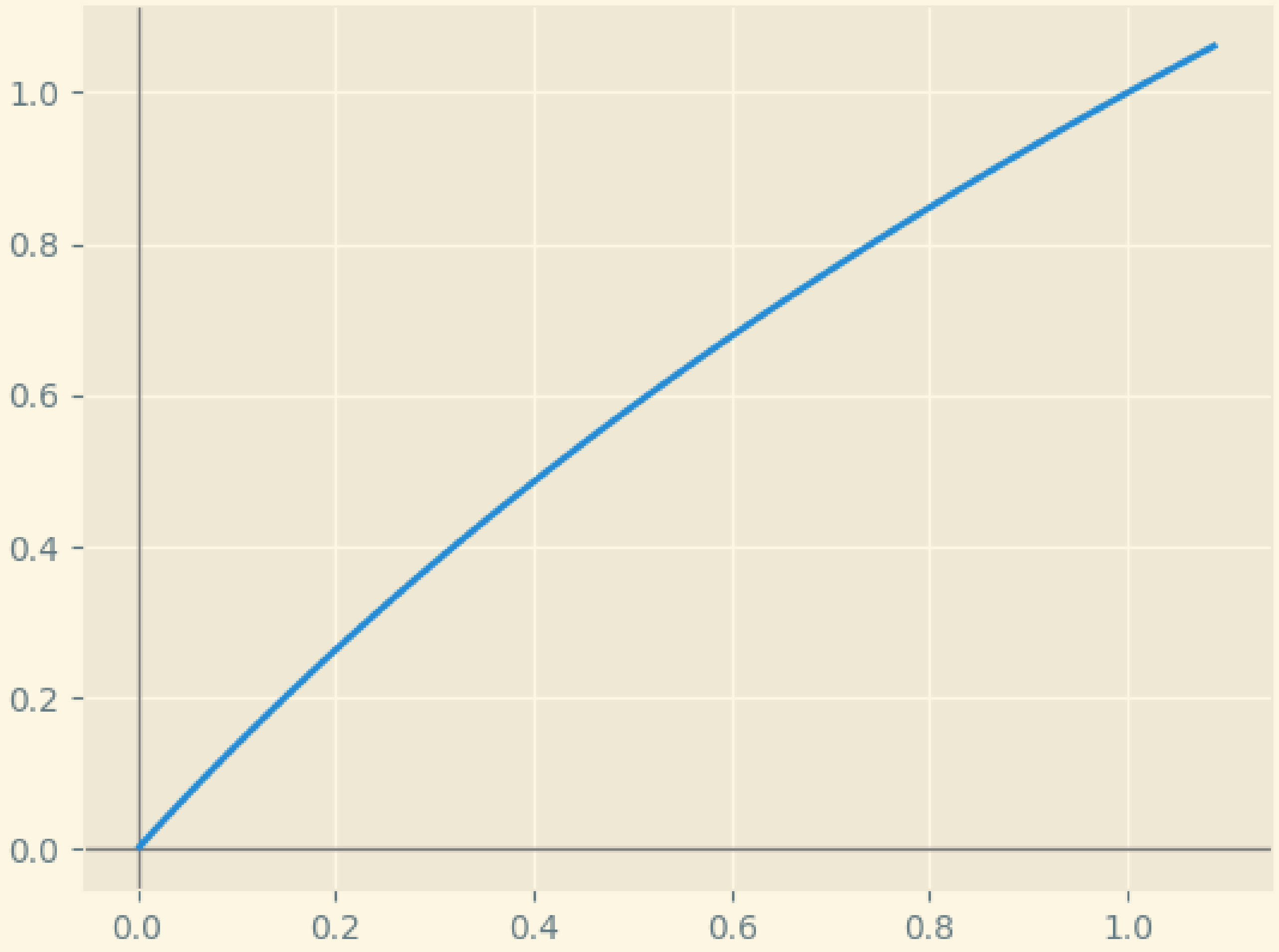
$$\boxed{\log_2\left(1 + m_{result}\right)} + e_{res} =$$
$$-\frac{1}{2}\left(\boxed{\log_2\left(1 + m_{input}\right)} + e_{input}\right)$$

$$0.0 \leq m < 1$$

Graph of log(1+x)

$$y = \log_2(1 + m)$$

$$0.0 \leq m < 1$$

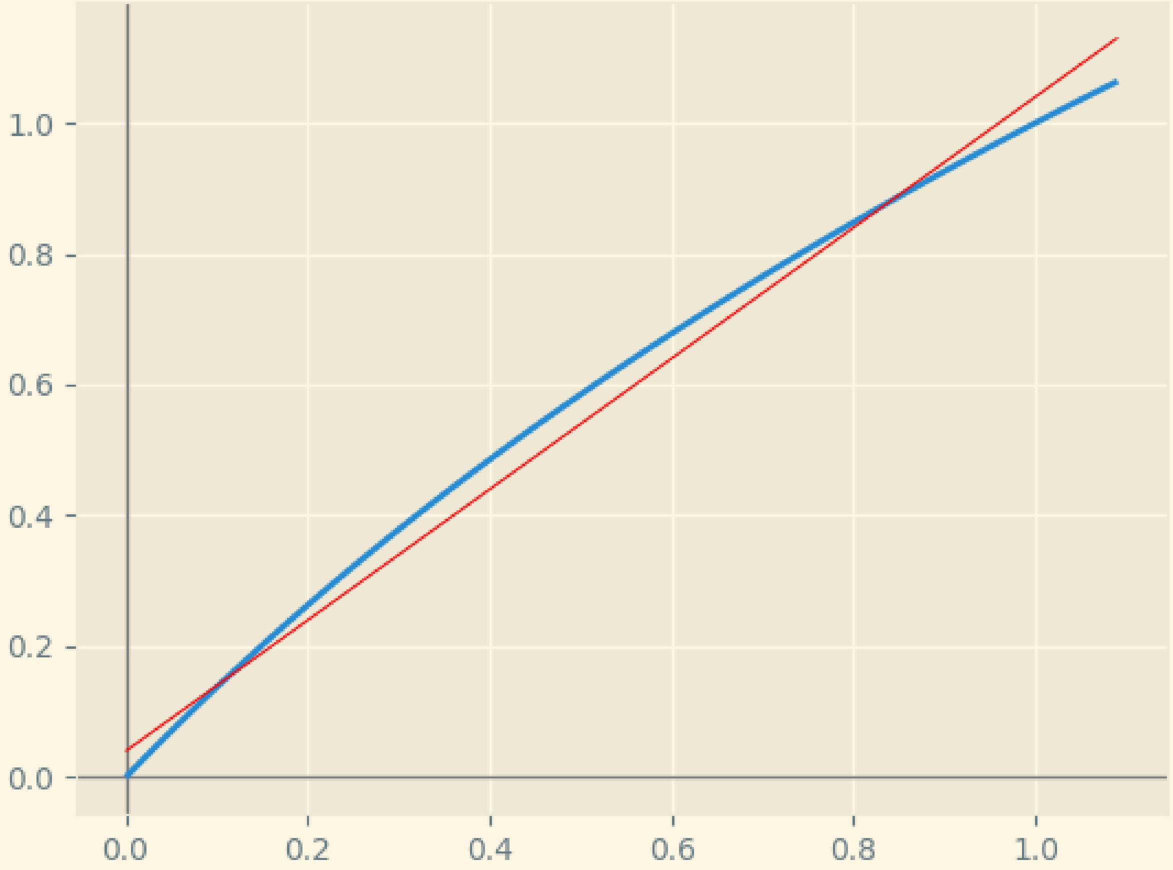


Approximate as straight line

$$y = \log_2(1 + m)$$

$$0.0 \leq m < 1$$

$$\log_2(1 + m) \approxeq m + \sigma$$



Substitute $m + \sigma$ into our equation

$$\log_2\left(1 + m_{result}\right) + e_{result} = -\tfrac{1}{2}\left(\log_2\left(1 + m_{input}\right) + e_{input}\right)$$

$$\boxed{m_{result} + \sigma + e_{result} \approxeq -\frac{1}{2}\left(m_{input} + \sigma + e_{input}\right)}$$

What about our actual float bits?

$$\frac{M_{result}}{2^{23}} + \sigma + E_{result} - 127 \approx -\frac{1}{2}\left(\frac{M_{input}}{2^{23}} + \sigma + E_{input} - 127\right)$$

Multiple both side by 2_{23} (aka L), and simplify

$$M_{res} + 2^{23}E_{res} \approx \frac{3}{2} * 2^{23}\left(127 - \sigma\right) - \frac{1}{2}\left(M_{input} + 2^{23}E_{input}\right)$$

Spot the $I = M + LE$ values!

$$\boxed{M_{res} + 2^{23} E_{res}} \approx \tfrac{3}{2} * 2^{23} \left(127 - \sigma\right) - \tfrac{1}{2} \left(\boxed{M_{in} + 2^{23} E_{in}}\right)$$

$$\boxed{\boxed{I_{result} \approx \frac{3}{2} 2^{23} (127 - \sigma) - \frac{1}{2} I_{input}}}$$

We have our magic code!

$$\boxed{I_{result} \approx \frac{3}{2} 2^{23} (127 - \sigma) - \frac{1}{2} I_{input}}$$

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

- Choosing
  - $\sigma = 0.0450465$
- Halving input as int by bit shifting, just like earlier

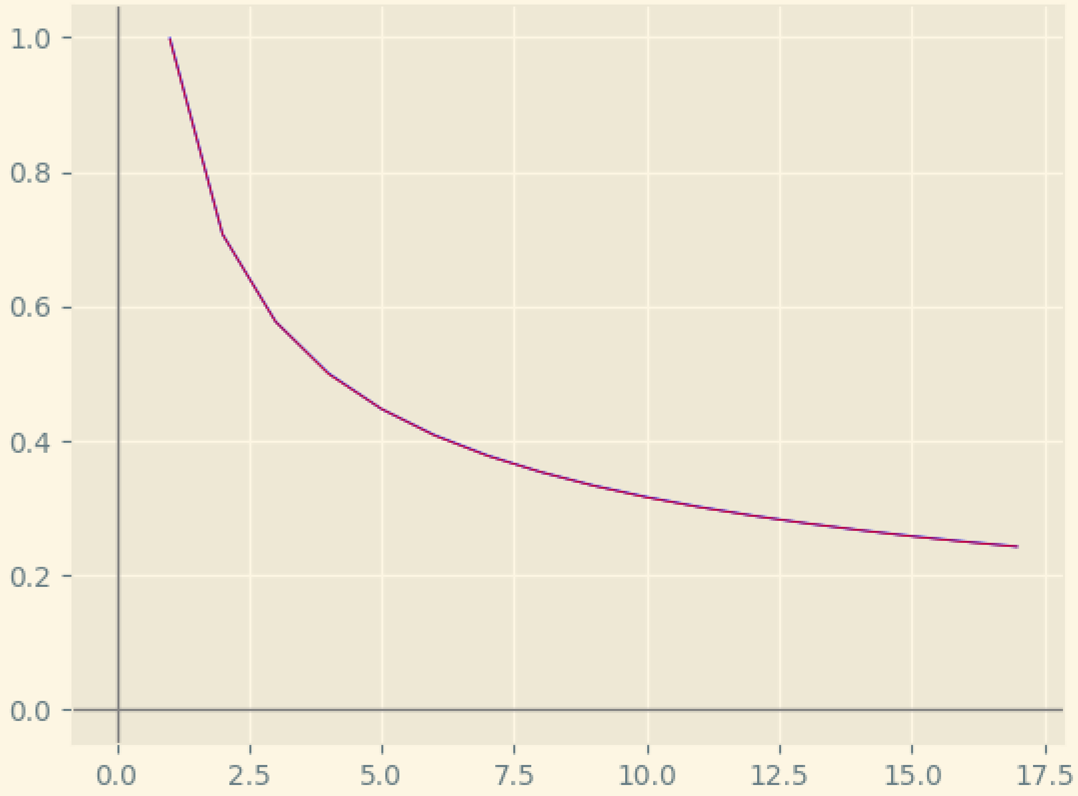
We have our magic code!

$$\boxed{I_{result} \approx \frac{3}{2}2^{23}(127 - \sigma) - \frac{1}{2}I_{input}}$$

```
i  = 0x5f3759df - ( i >> 1 );  // Gosh this makes perfect sense!
```

- Choosing
  - $\sigma = 0.0450465$
- Halving input as int by bit shifting, just like earlier

And it works!



Is it fast?

Depends what you're comparing against...

```
result = 1.0 / std::sqrt(input);
```

- Q_rsqrt slightly faster (≈ 0.9x) than standard library when floating point model is `Precise (/fp:precise)`
- BUT 3x **slower** if floating point model is `Fast (/fp:fast)`

Should we ever use it?

Equivalent code works for any power $\left(Quoted\ online\ as\ between\ -1\ and\ 1\ I\ think\ because\ you\ can\ extract\ the\ int\right)$

$$\boxed{I_{result} \approx (1-p)\,L(B-\sigma) + pI_{input}}$$

e.g.

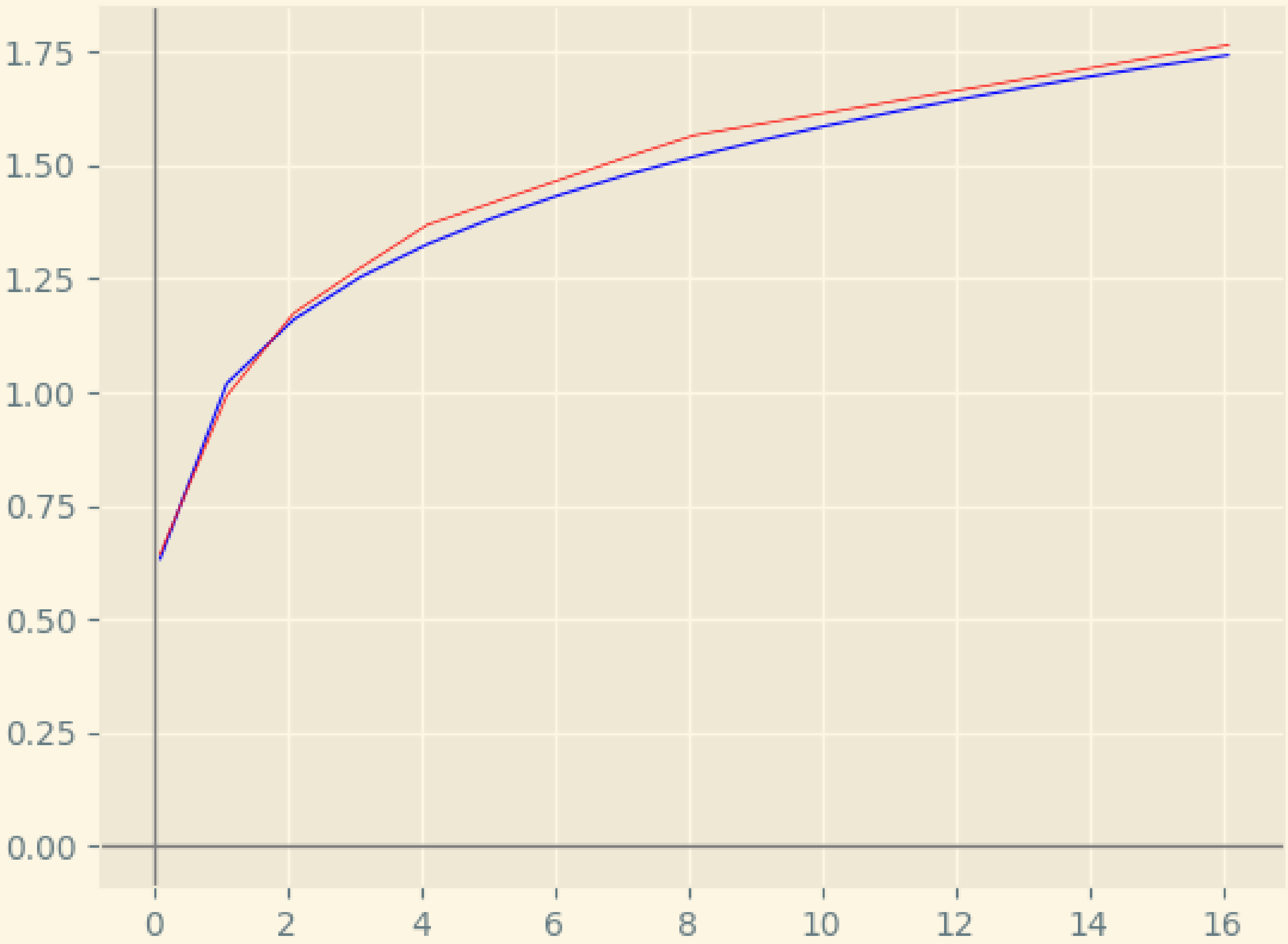
- $result = input^{\frac{1}{5}}$
- $p = \frac{1}{5}$

$$I_{result} \approx (1 - p)\, L(B - \sigma) + pI_{input}$$

$$I_{result} \approx \left(1 - \tfrac{1}{5}\right) 2^{23}(127 - 0.0450465) + \tfrac{1}{5} I_{input}$$

```
i = 0x32C82FEF + (0.2 * i);    // Wow it also works for 1/5
```

- No newton raphson but still not bad!
- ~4x faster than `std::pow(input, 0.2)`


1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
2
4
6
8
10
12
14
16

Conclusions

- Floating bit representations are fun!
- Log rules are useful
- Just because it says it's fast doesn't mean it is

Useful links

- Christian Plesner Hansen, 2012 blog post
- Charles McEniry, The Mathematics Behind the Fast Inverse Square Root Function Code, 2007
  - *I couldn't find the paper anywhere but references to it litter in the internet - if anyone has a copy please send my way!*
- Chris Lomont, 2003 Paper
- H-schmidt Float converter
- Sean Eron Anderson - Bit Twiddling Hacks

Test power of two

- Test power of two: `isPowTwo = !(val & (val - 1));` or with `val &&` so 0 returns false

Val	16	17	18
Bits	0001 0000	0001 0001	0001 0010
-1	0000 1111	0001 0000	0001 0001
& with val	0000 0000	0001 0000	0001 0000

Reverse all the bits

```
unsigned int v;     // input bits to be reversed
unsigned int r = v; // r will be reversed bits of v; first get LSB of v
int s = sizeof(v) * CHAR_BIT - 1; // extra shift needed at end

for (v >>= 1; v; v >>= 1)
{
  r <<= 1;
  r |= v & 1;
  s--;
}
r <<= s; // shift when v's highest bits are zero
```

Any questions?