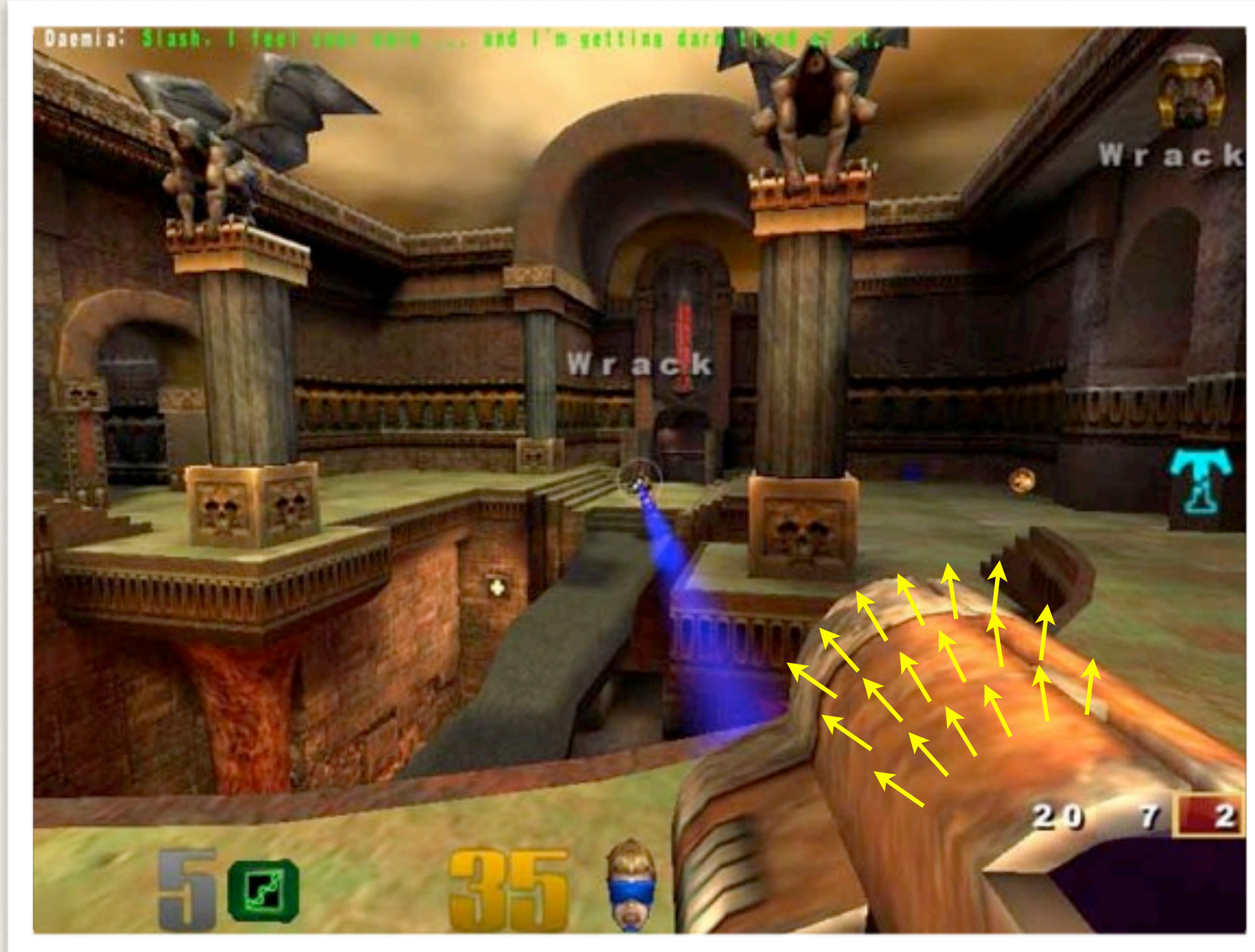


Way To Normal

Fast Inverse Square Root magic trick

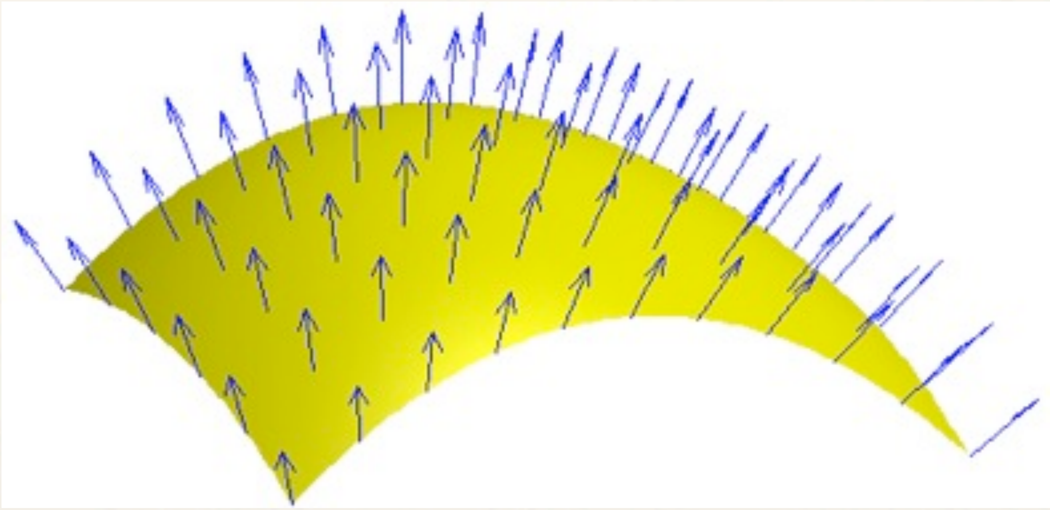
Loïc Matthey

Tea Talk n°11

17th January 2013

Introduction

- How to compute $\frac{1}{\sqrt{x}}$ quickly and efficiently?
- Cool piece of code found in Quake III source code, attributed to John Carmack.
  - Actually going back to SGI, 3dfx and first written in mid 1980s by Greg Walsh
- Needed to compute normals, used extensively in lighting and shading



The piece of code

```
float FastInvSqrt(float x) {
    long i;
    float x2, y;
    float xhalf = 0.5f * x;
    y  = x;

    i  = *( long * ) &y;                     // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );            // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( 1.5f - ( x2 * y * y ) );  // 1st iteration

    return y;
}
```

- (with original comments of Quake III progammers)

Piece of code

```
float FastInvSqrt(float x) {
    long i;
    float x2, y;
    float xhalf = 0.5f * x;
    y  = x;

    i  = *( long * ) &y;                        // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( 1.5f - ( x2 * y * y ) );         // 1st iteration

    return y;
}
```

- Reinterprets bits of floating-point number as an integer

Piece of code

```
float FastInvSqrt(float x) {
    long i;
    float x2, y;
    float xhalf = 0.5f * x;
    y  = x;

    i  = *( long * ) &y;                     // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );            // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( 1.5f - ( x2 * y * y ) );      // 1st iteration

    return y;
}
```

- Does integer arithmetic on it, produces an approximation to inverse square root

Piece of code

```
float FastInvSqrt(float x) {
    long i;
    float x2, y;
    float xhalf = 0.5f * x;
    y  = x;

    i  = *( long * ) &y;                      // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );             // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( 1.5f - ( x2 * y * y ) );       // 1st iteration

    return y;
}
```

- Reinterprets bits as floating-point number

Piece of code

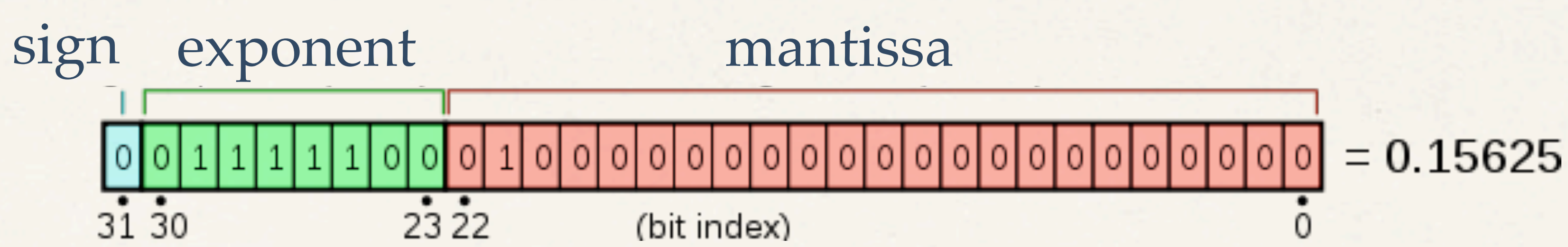
```
float FastInvSqrt(float x) {
    long i;
    float x2, y;
    float xhalf = 0.5f * x;
    y  = x;

    i  = *( long * ) &y;                      // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );             // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( 1.5f - ( x2 * y * y ) );       // 1st iteration

    return y;
}
```

- Single iteration of Newton's Method to improve the approximation

Floating point representation



- Floating point representation: $[x_F]_2 = (-1)^s(1+m)2^e$
- Exponent: add bias, signed integer [-127, 128].
- Mantissa: normalised between 0 and 1

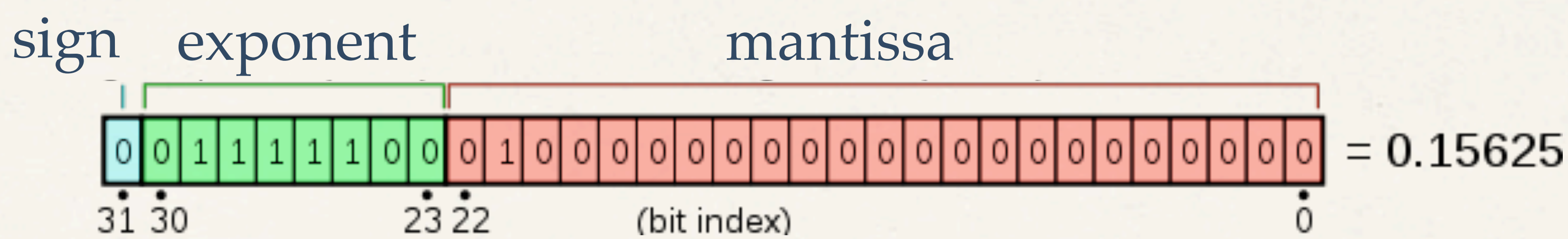
- Integer views of exponent and mantissa: E and M
- Floating point interpretation:

$$m = \frac{M}{L}$$

$$e = E - B$$

  - 32 bit floating points: $L = 2^{23}$, $B = 127$

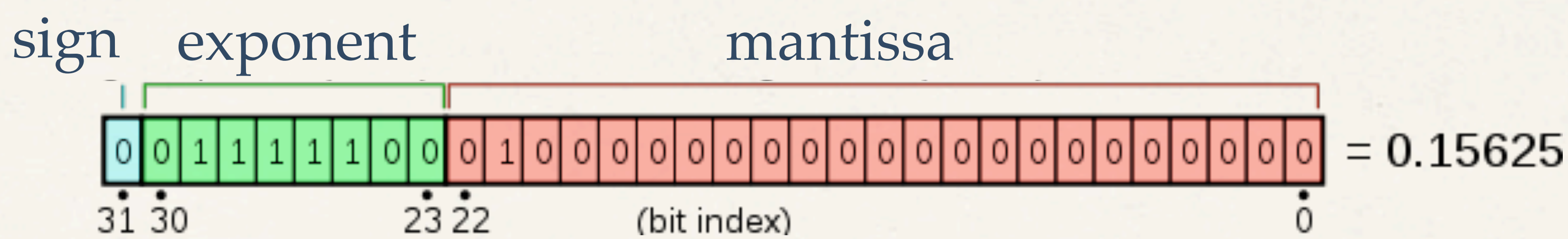
Floating point representation



- Here:
$$[x_F]_2 = (-1)^s(1+m)2^e$$
$$M = 2^{21}$$
$$E = 124$$
$$m = \frac{M}{L} = \frac{2^{21}}{2^{23}} = 0.01$$
$$e = E - B = 124 - 127 = -3$$

$$x_F = (-1)^s(1+0.01)2^{-3} = 0.00101$$
$$x = [x_F]_{10} = 0.15625$$

Floating point representation



- Corresponding integer interpretation of the same bits:

$$z_I = M + LE$$

- So here:

$$M = 2^{21}$$

$$E = 124$$

$$z_I = 2^{21} + 124 \cdot 2^{23} = 1042284544$$

Inverse square root

- Back to our function:

$$y = \frac{1}{\sqrt{x}} = x^{-\frac{1}{2}}$$

- Take the log:

$$\log_2 y = -\frac{1}{2} \log_2 x$$

- Replace by floating point representation:

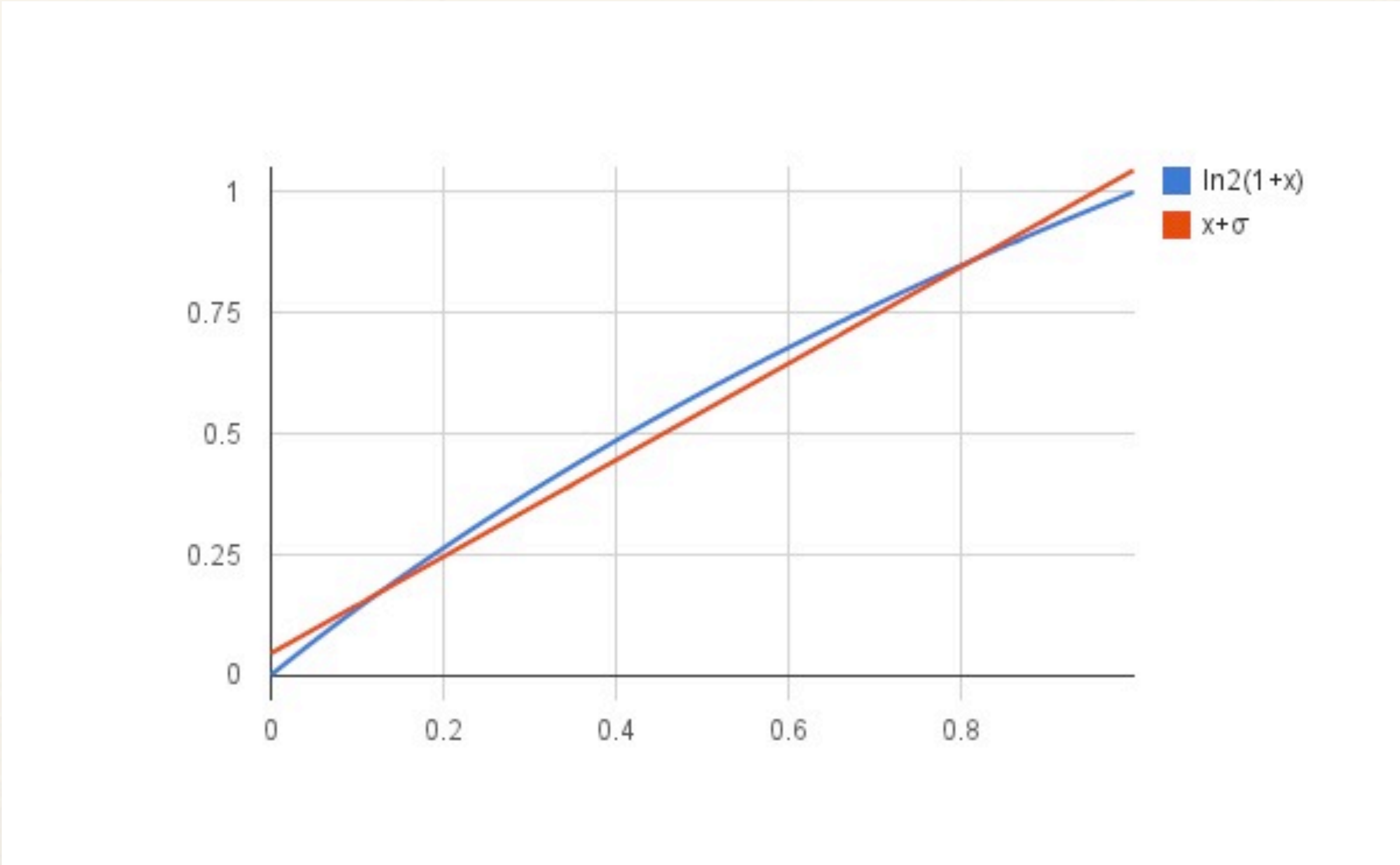
$$x = (1 + m_x) 2^{e_x} \qquad y = (1 + m_y) 2^{e_y}$$

$$\log_2 (1 + m_y) + e_y = -\frac{1}{2} (\log_2 (1 + m_x) + e_x)$$

Inverse square root

- The trick.
  - Linear approximation of log. Choose best σ

$$\log_2(1 + v) \approx v + \sigma$$



Inverse square root

- Approximate.

$$\log_2(1+m_y)+e_y = -\frac{1}{2}(\log_2(1+m_x)+e_x)$$

$$\approx \;\; m_y+\sigma+e_y = -\frac{1}{2}(m_x+\sigma+e_x)$$

- Replace by integer view of exponent and mantissa:

$$\frac{M_y}{L}+\sigma+E_y-B = -\frac{1}{2}(\frac{M_x}{L}+\sigma+E_x-B)$$

$$\frac{M_y}{L}+E_y = -\frac{1}{2}(\frac{M_x}{L}+E_x)-\frac{3}{2}(\sigma-B)$$

$$M_y+LE_y = \frac{3}{2}L(B-\sigma)-\frac{1}{2}(M_x+LE_x)$$

$$\mathbf{I_y} = \frac{3}{2}L(B-\sigma)-\frac{1}{2}\mathbf{I_x}$$

Inverse square root

- Integer representation operation:
  divide by two, add some constant.

$$\mathbf{I_y} = \frac{3}{2}L(B-\sigma) - \frac{1}{2}\mathbf{I_x}$$

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

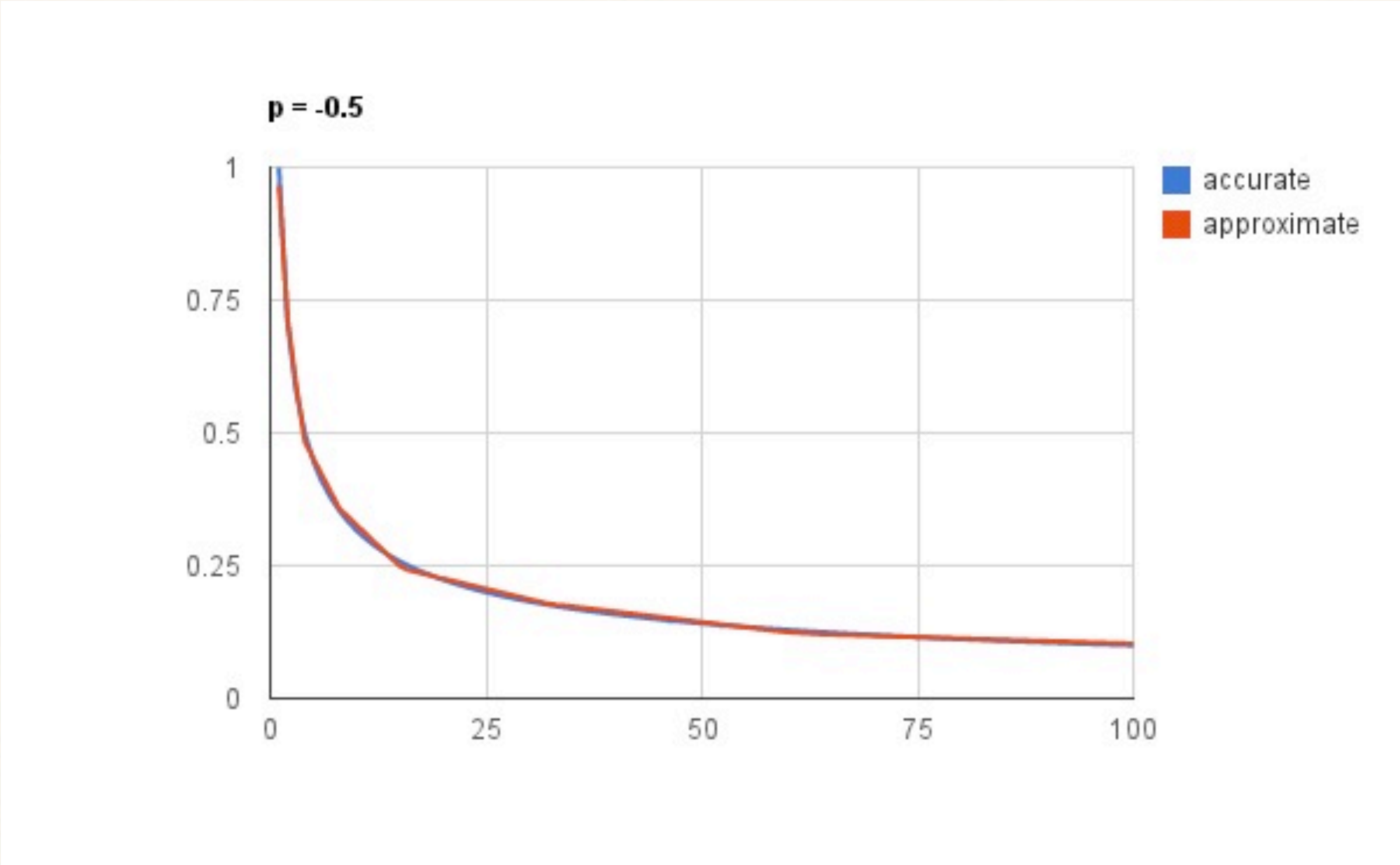
- L and B known. σ chosen to give best approximation to log.
  Here:

$$\sigma = 0.0450465$$

$$\frac{3}{2}L(B-\sigma) = 1597463007 = [5f3759df]_{hex}$$

Inverse square root

- Precision of approximation?



- Newton step added to be even more precise
  - 10% error -> 0.6% error

Conclusion

- Cool trick, floating-point operation transformed into integer addition and shift (fast).
- Can be extended to any power of x, actually.
- But SSE hardware instruction faster now, less critical!

The End

- Questions?

- References:
    - http://blog.quenta.org/2012/09/0x5f3759df.html
    - http://en.wikipedia.org/wiki/Fast_inverse_square_root

- Supplementary slides

Newton step

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}$$

$$f(y) = \frac{1}{y^2} - x = 0$$

$$f'(y) = -\frac{2}{y^3}$$

$$y_{n+1} = y_n + \frac{y_n^{-2} - x}{2y_n^{-3}}$$

$$= \frac{2y_n^{-2} + y_n^{-2} - x}{2y_n^{-3}} = \frac{3y_n - xy_n^3}{2}$$

$$= y_n(\frac{3}{2} - \frac{x}{2}y_n^2)$$

$$\mathbf{I_y} \approx (1-p)L(\sigma - B) + p\mathbf{I_x}$$

Extend to any power

$$\mathbf{I_y} \approx (1-p)L(\sigma - B) + p\mathbf{I_x}$$

$$\mathbf{I_y} \approx K_{\frac{1}{2}} + \frac{1}{2}\mathbf{I_x}$$

$$K_{\frac{1}{2}} = \frac{1}{2}L(B-\sigma) = \frac{1}{2}2^{23}(127 - 0.0450465) = \texttt{0x1fbd1df5}$$