快速求解平方根倒数

参考链接

- Fast Inverse Square Root — A Quake III Algorithm – YouTube
- Fast inverse square root – Wikipedia

按照Wikipedia的说法，这个算法1999年就出现在Quake III Arena代码仓库中，这个项目代码在2005年开源。但是在此之前的2002-2003年间，这段代码就已经出现在了Usenet上。最开始这个算法的作者被认为是Id-Software co-founder John Carmack，但是后来有发现说这个算法可能在更早的时候就被发明出来了。

问题起源

在游戏中有很多地方需要对空间矢量进行归一化，比如 (x, y, z) 可以归一化为

$$(x,y,z)/\sqrt{(x*x+y*y+z*z)}$$

那其中后半部分就是平方根倒数的由来，快速计算这个值对游戏性能至关重要。

在游戏编程中，包括现在的深度学习，为了性能大部分浮点都是使用float类型表示。后面说到的浮点数都是32-bits的float类型，然后long类型也是32-bits，并且假设运行在little-endian X86系统上。

如何实现

通常实现方式是调用库函数 $1/\sqrt{x}$. 这种实现性能上有两个问题：sqrt本身实现(todo: cycles?) 和外面除法（~10 cycles).

而Quake III里面的实现则是下面这样

```
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;
}
```

代码非常简单，指令也好计算：

- 4条乘法指令(4 * 3 = 12 cycles)
- 1条移位指令(1 cycle)
- 1条减法指令(1 cycle)

不考虑流水线，整体计算下来只需要14 cycles. 所以这也是它计算快的原因。

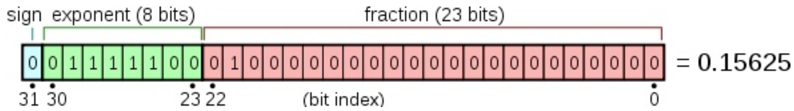
要理解它需要了解三个知识点：

1. IEEE 754 float表示
2. $log2(1+x)$ 的近似
3. 牛顿迭代法（用于最后修正）

IEEE 754 float表示

https://en.wikipedia.org/wiki/IEEE_754

https://www.h-schmidt.net/FloatConverter/IEEE754.html



二进制的表示分为3个部分：

1. sign bit. 对于我们这里来说，sign bit = 0
2. exponent bits. 指数部分，后面我们使用E表示，它表示的数值是 $2^{(E-127)}$
3. fraction bits. 小数部分，后面我们使用M(Mantissa)表示，它表示的数值是 $1 + M/(2^{23})$

最终的浮点值是三个部分相乘得到的，以上面为例：

- 指数部分的值是 $2^{(E-127)}$. E部分是124, 那么指数部分的值就是 2^{-3} = 0.125
- 小数部分的值是 $1 + M/(2^{23})$. M部分是 2^{21}，所以小数部分就是 1.25
- 最后相乘在一起就是 0.125 * 1.25 = 0.15625

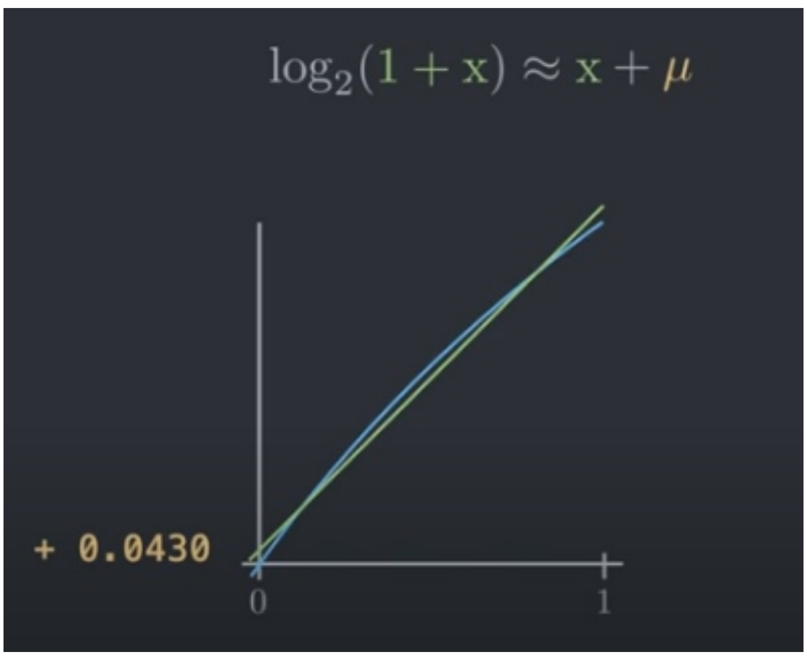
写成形式化的公式就是 $1 + M/(2^{23}) * 2^{(E-127)}$. 这个公式后面还会使用到。另外需要注意的是，这个二进制对应的整数是 $(E << 23) + M$

log(1+x)的近似

对于x在[0,1]区间内的值，$log2(1 + x)$ 函数曲线如下，可以看到它近似地等于 (x+u)，其中这个u是用来进行修正的。

至于为什么 $log2(1 + x) \ = (x + u)$，大家可以去找找麦克劳林公式/泰勒级数展开的相关材料。

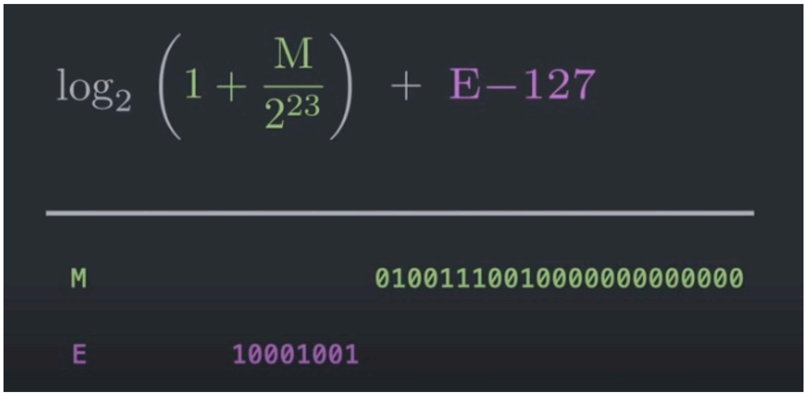
下图显示的是u选择在0.0430的话，那么还是可以比较好地拟合到 $log2(1 + x)$ 这个曲线的。



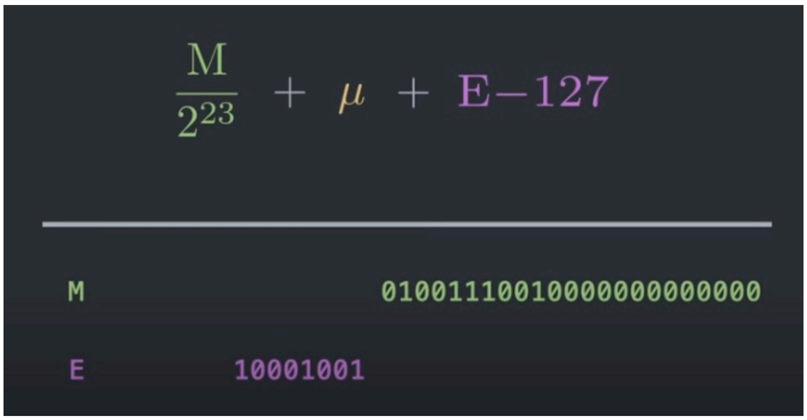
数值展开

假设 $x = 1/\sqrt{y} = y^{-1/2}$, 我们在上面取log2操作的话，可以得到 $log2(x) = -0.5 * log2(y)$

这里我们先y替换成为浮点数表示，我们将y写成 $1 + M/(2^{23}) * 2^{(E-127)}$ 的话，那么 $log2(y) = (E - 127) + log2(1 + M/(2^{23}))$

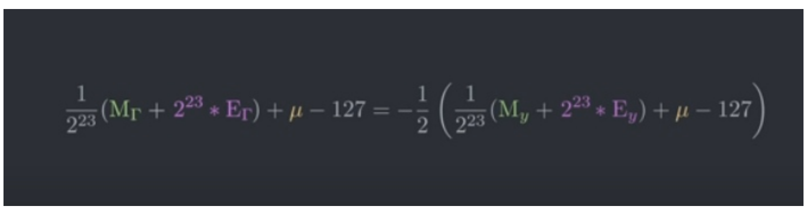


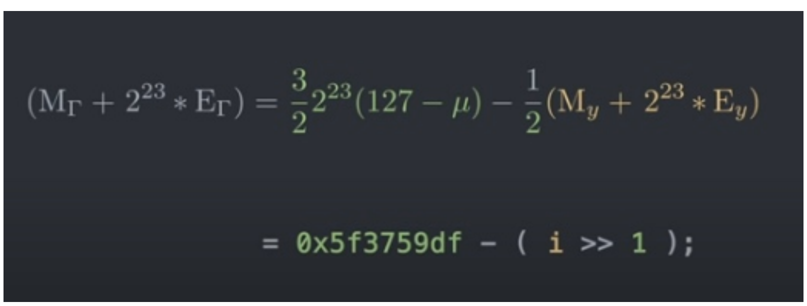
接着我们使用 log2(1+x) 的近似 $log2(1+x) \;= (x+u)$ 就可以得到 $(E-127\,)+M/(2^{23})+u$



可以看到通过这些变化，原本比较难以求解的 $log(y)$ 变为了 $E-127+M/(2^{23})+u.$ 这样的形式

如果我们对x, y两边同时做类似的展开的话，那么可以得到下图这样的等式：





简化到最后可以发现，左右两边都有我们希望的 **浮点数的整数表示** $(E<<23)+M.$

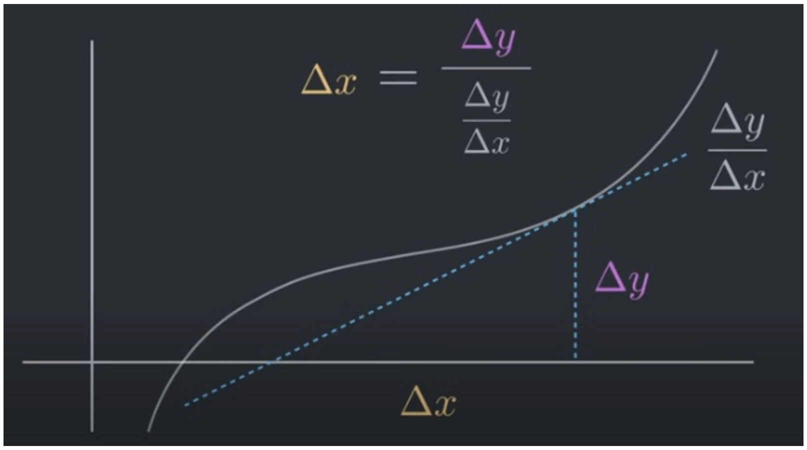
回忆之前我们假设u = 0.0430, 带入后可以得到这个数值是 ~= 1597488759 (0x5f37be77)。这里得到的数值和截图稍微有点差异，实际选择u很有讲究，有人还专门为此写篇论文来说如何选择这个u ([http://www.matrix67.com/data/InvSqrt.pdf)。](http://www.matrix67.com/data/InvSqrt.pdf)

牛顿迭代法

牛顿迭代法主要是用来求解 `f(x) = 0` 的，大致思路是:

1. 假设我们在 (x0, y0) 点上，这个点上存在一条切线
2. 沿着切线向着 y = 0 的方向进行滑动 dx. 其中这个dx计算公式如下图
3. 然后我们就到达了 x0 – dx. 接着重复第一步直到 y0 结果可以接受 (~=0)

图中 $dy/dx = f'(x), dy = f(x)$, 所以 $dx = f(x)/f'(x)$



在这个函数中，牛顿迭代法主要是用来做最后一次修正的，有助于减少偏差。

在这个问题里面要求解 $x = 1/\sqrt{y}$, 所以

- $f(x) = 1/(x^2\ ) - y = 0$
- $f'(x) = -2 * x^{-3}$
- $x+ = 1/2 * x * (1 - x^2 * y)$

所以迭代公式是 $x' = x * (3/2 - 1/2 * (x^2) * y)$

整合优化

```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;                       // 将输入变为二进制整数
    i  = 0x5f3759df - ( i >> 1 );               // 利用log2(1+x)展开
    y  = * ( float * ) &i;                      // 将输出变为浮点数
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 一轮牛顿迭代进行修正
    return y;
}
```

看看上面代码，都是简单的计算，也没有任何分支代码，我们是否可以继续优化呢？SIMD. 整个实现几乎就是直接翻译。

```
void Q_rsqrt_simd(float* number, float* output, size_t n) {
    assert(n % 8 == 0);
    // 32 * 8 = 256 bits.
    size_t loop = n / 8;

    static const int MAG = 0x5f3759df;
    static const float THREEHALFS = 1.5f;
    static const float HALF = 0.5f;
    __m256i mag = _mm256_set_epi32(MAG, MAG, MAG, MAG, MAG, MAG, MAG, MAG);
    __m256 threehalfs = _mm256_set_ps(THREEHALFS,THREEHALFS,THREEHALFS,THREEHALFS,THREEHALFS,THREEHALFS,THREEHALFS,THREEHALFS);
    __m256 half = _mm256_set_ps(HALF,HALF,HALF,HALF,HALF,HALF,HALF,HALF);
    __m128i srl = _mm_set_epi64x(0x0, 0x1);

    for(size_t i = 0; i < loop; i++ ) {
        // x2 = number * 0.5f
        // output = t0
        __m256 t0 = _mm256_loadu_ps(number);
        t0 = _mm256_mul_ps(t0, half);

        // i  = * ( long * ) &y;
        // i  = 0x5f3759df - ( i >> 1 );
```

```
        // y  = * ( float * ) &i;
        // output = t1
        __m256i t1 = _mm256_loadu_si256((__m256i const*)number);
        t1 = _mm256_srl_epi32(t1, srl);
        t1 = _mm256_sub_epi32(mag, t1);
        __m256 t2 = (__m256i)t1;

        // y  = y * ( threehalfs - ( x2 * y * y ) );
        // output = t2
        __m256 t3 = _mm256_mul_ps(t2, t2);
        t3 = _mm256_mul_ps(t0, t3);
        t3 = _mm256_sub_ps(threehalfs, t3);
        t3 = _mm256_mul_ps(t2, t3);

        _mm256_storeu_ps(output, t3);
        number += 8;
        output += 8;
    }
    return;
}
```

性能对比

代码放在了 GitHub 上面，这里就列举一下最后的测试结果。在我的Mac上使用clang，和在开发机器上使用gcc10分别编译，运行时间差距还是蛮大的。Benchmark使用了4中方法来计算：

1. 1/std::sqrt(x)
2. `_mm256_rsqrt_ps`（内置SIMD指令）
3. Q_rsqrt
4. Q_rsqrt_simd

在我的mac上使用clang 12.0.0运行结果如下：使用std::sqrt结果最慢，其他三个没有太大差别

```
mbp :: .codes/cc/misc ‹master› » g++ --version

Configured with: --prefix=/Library/Developer/CommandLineTools/usr --with-gxx-include-dir=/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/c++/4.2.1
Apple clang version 12.0.0 (clang-1200.0.32.2)
Target: x86_64-apple-darwin19.6.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
mbp :: .codes/cc/misc ‹master› » g++ BenchRSqrt.cpp -mavx2  -std=c++11  -lbenchmark  -lbenchmark_main -O3
mbp :: .codes/cc/misc ‹master› » ./a.out
2021-08-17T14:37:04+08:00
Running ./a.out
Run on (8 X 2000 MHz CPU s)
CPU Caches:
  L1 Data 48 KiB (x4)
  L1 Instruction 32 KiB (x4)
  L2 Unified 512 KiB (x4)
  L3 Unified 6144 KiB (x1)
Load Average: 2.11, 2.18, 2.26
------------------------------------------------------------------
Benchmark                        Time             CPU   Iterations
------------------------------------------------------------------
run_std_rsqrt/100000         42782 ns        42787 ns        15987
run_rsqrt_simd/100000        16750 ns        16801 ns        41947
run_Q_rsqrt/100000           16944 ns        16993 ns        40784
run_Q_rsqrt_simd/100000      18073 ns        18086 ns        40247
```

在开发机器上运行结果如下：反而是 Q_rsqrt 运行时间最长（不知道为什么），std::sqrt时间尚可，内置的SIMD指令运行时间最短。

```
(py3env) sandbox-sql :: ~ » g++ test.cpp -mavx2  -std=c++11  -lbenchmark -lbenchmark_main -O2 -I${DORIS_THIRDPARTY}/installed/include -L${DORIS_THIRDPARTY}/installed/lib64 -lpthread
(py3env) sandbox-sql :: ~ » ./a.out
2021-08-17 14:31:23
Running ./a.out
Run on (104 X 3800 MHz CPU s)
```

```
CPU Caches:
  L1 Data 32 KiB (x52)
  L1 Instruction 32 KiB (x52)
  L2 Unified 1024 KiB (x52)
  L3 Unified 36608 KiB (x2)
Load Average: 0.15, 0.22, 0.28
-----------------------------------------------------------------
Benchmark                       Time             CPU   Iterations
-----------------------------------------------------------------
run_std_rsqrt/100000        189042 ns       189008 ns         3704
run_rsqrt_simd/100000        13898 ns        13868 ns        50074
run_Q_rsqrt/100000         1098357 ns      1098262 ns          637
run_Q_rsqrt_simd/100000      22751 ns        22707 ns        31140
```