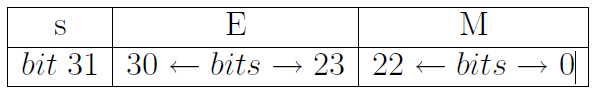
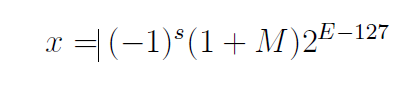
平方根倒数速算法  
  
原文：http://www.matrix67.com/data/InvSqrt.pdf  
  
摘要：很多程序不可避免的要计算平方根倒数，例如3D游戏中的正规化矢量。通常，为了大幅提高运算速度而损失一些精度是可以接受的。本文是对一些网上链接库的源码中发现的快速算法的讨论和改进，并提供了推导出其他类似算法的思路。  
  
1.引言  
 在www.gamedev.net[1]的数学编程论坛上，我看到一个有趣的方法计算平方根倒数。下面是C的代码和我加的注释：  
float InvSqrt(float x)  
{  
 float xhalf = 0.5f\*x;  
 int i = \*(int\*)&x; // get bits for floating value  
 i = 0x5f3759df - (i>>1); // gives initial guess y0  
 x = \*(float\*)&i; // convert bits back to float  
 x = x\*(1.5f-xhalf\*x\*x); // Newton step, repeating increases accuracy(牛顿迭代，用重复计算来增加精度)  
 return x;  
}  
0x5f3759df这个常量非常有趣：它是哪来的，工作原理是什么？通过VC++.Net[2]做的一些快速测试显示这个方法速度比原生的(float)(1.0/sqrt(x))方法快近乎4倍，并且所有浮点数中最大相对误差仅有0.00175228，所以这个方法看上去非常有用。对此有三个直接的问题：1).它是怎么工作的，2).是否可以改进，3).哪个高手设计了这样一个难以置信的东西？  
 用Google搜索0x5f3759df ，最贴近的是2002.1.9[3]的一段线程comp.graphics.algorithms的一个引用了它的线程的代码，和一个D. Eberly[4]的解释（错误，但是贴近）。然而他的解释是有启发性的。更进一步的挖掘也没有找到正确的说明。一个传说中的程序员John Carmack在Quake 3的源码里写了它，但是有人（我不知道出处）在往上说是Nvidia的Gary Tarolli。有人能确定么？Crystal Space[5]的源码里也出现了它，Titan引擎[6]，这个快速代码的库似乎都来之Quake 3。  
 Eberly[4]对平方根倒数插值的线性变化更清楚的解释了使用这个算法的动机。本文记录了一些提升这段代码速度的方法，但是并不会更进一步推进优化。有一些例如表查找的类似加速方法，但是其中大多数的精度都不如这个方法。  
 本文假定PC使用32位浮点和整型或类似架构。特别之处浮点表示标准使用的是IEEE 754-1985[7]。这段C的代码确认是端中性（译注：并不事先确定是big还是little）的（据说在Macintosh上测试过）。不过我没有验证过。由于方法是以32位工作的它应该（令人惊讶的）是端中性的。本文的观点是它很容易扩展为适应其他场景（比如double类型）。无论如何，主要还是那三个问题。  
  
2.背景

浮点在PC中是32位的；

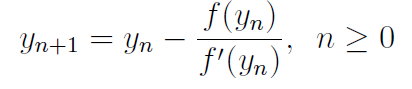


S是占1bit的符号位（1为负）, E是8bits的指数，M是23bits的有效数字。指数存储时会偏移127以保存负数（译注：E是无符号的，但科学计数法允许符号，浮点的E取值范围为0~255，于是取中间值127作为偏移量，比如-1保存为-1+127=126，这样就可以保存负数了），有效数字不保存首位的1（译注：IEEE 754，因为第一位在PC中保存总是1，所以忽略），所以M可以认为是以小数点左边表示的二进制数，取值范围是大于等于0且小于1。存储的值计算公式为：



这些为可以用来表示浮点实数，但看这些位的时候也可以表示整型。M究竟是表示[0，1)区间的浮点还是整型取决于上下文。M由浮点实数转为整型需要乘以2的23次方。

3.算法

主要思路是牛顿迭代，并以这个神奇的数字估算一个初始的近似值。给定一个浮点值x>0，我们想计算x的平方根倒数。定义。我们求的是f(x)的正根。牛顿的算法是首先假定一个近似根的值,并使用公式来计算更贴近的值。对于给定的f（y）,简化为对应代码x = x\*(1.5f-xhalf\*x\*x)，当x为初始估算值时，称结果为。

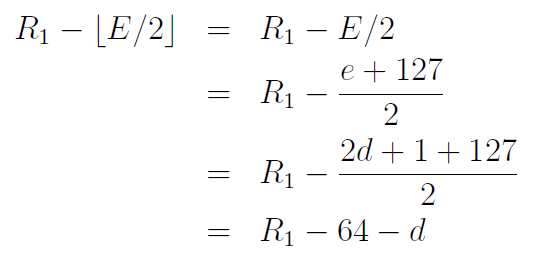
代码i = 0x5f3759df - (i>>1)估算初始近似值，大致为x的指数乘以负二分之一，并拼接位来减小误差。Eberly的解释是这种方法会使估算值线性接近标准值，不过这是错的；这个估算值的线性是分段的，这个算法的表现并不总是一样的。尽管如此，我通过推测他的解释得到了创建这个算法的灵感。

4．神奇的数字(译注: 0x5f3759df)

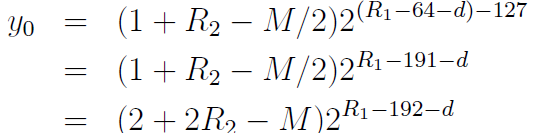
于是，我们还需要做的就是找一个初始的预估值。假设一个如之前背景中介绍的浮点32位格式的大于0的值x，指数e=E位的值-127，于是，于是目标值，以浮点实数的形式处理。一般情况下，我们使用一个神奇的常量R来计算，其中减法应用于整型，i是x的bit形式（译注：\*(int\*)&x），不过结果看做浮点实数。也就是说，R是以浮点型表示的整型，R1是指数部分R2是尾数部分。我们将i右移一位的时候，有可能会将指数的bit移到尾数部分中去，所以我们应该以两种不同的情况来进行分析。

在文章之后的部分，我们使用一个浮点实数，并用表示小于或等于的整型。

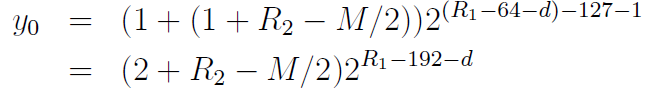
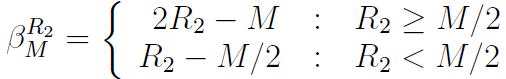
4.1 指数E没有位移进尾数部分。这种情况下，i>>1不会引发指数E的位移动到尾数M部分，。计算真正的指数e=E-127，假设e=2d+1,并且d是整型。位移后的预估值指数：

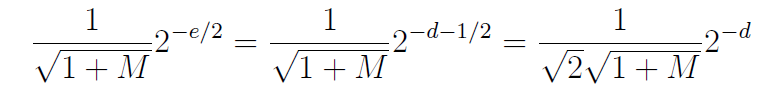


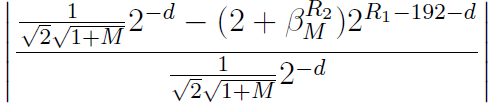
这个结果值必须是正数。因为如果是负数，标志位将会变为1，将无法返回正数。而结果如果是0，尾数部分将无法从指数借位，后面会说明借位是必须的。由于E取值范围是0到254，那么R1必须大于等于128。

由于没有位会因为执行i>>1而位移进尾数部分，于是新的尾数在的情况下是（整型）。于是，初始预估值是。

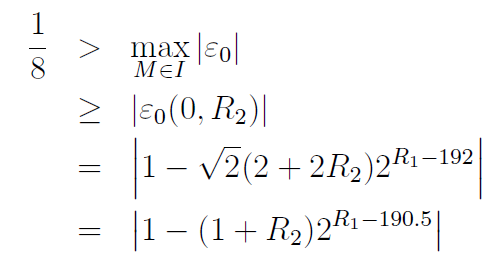
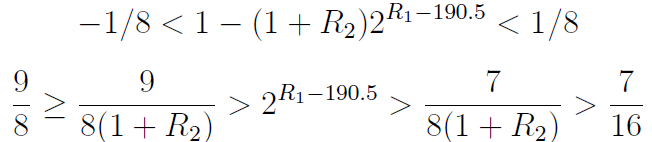
使用替换 ，尾数（作为浮点实数）最大的误差为，相比其他方法，这个误差是可以忽略的。

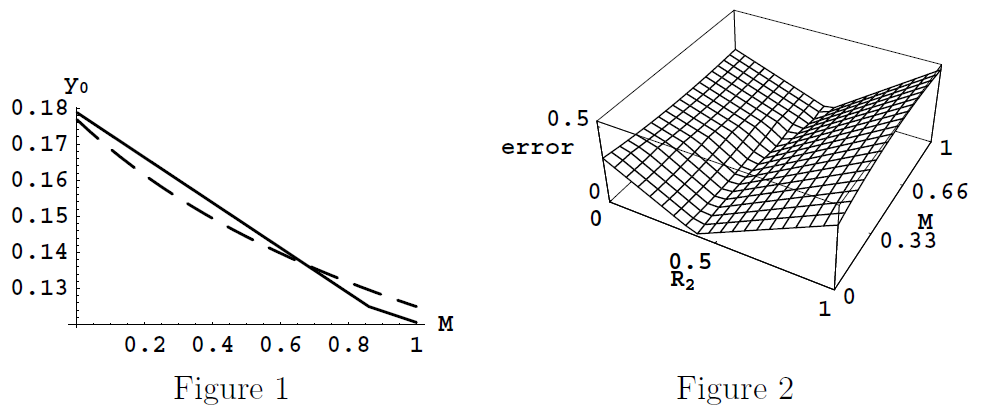
如果，在R−(i>>1)以整型做减法后，尾数部分会从指数部分借一位（这也是上面说的必须借位的原因），于是y0分为了两种情况。仍在I中的尾数部分是。当的这种情况。定义就可以将这两种情况合成一个：。

值得注意的是是连续的，并且除了R = M/2之外是可微的（译注：假设函数y=f(x)在x+Δx处可导，可导是指自变量X产生变化时x的增量Δx与y的增量Δy比值连续(Δy与Δx的比值在Δx趋于0时的极限存在，比值即为x+Δx的导数，也是函数在标轴上形成的曲线在x+Δx点的切线斜率，而微分是对这种局部变化的描述当Δx变化足够小时Δy的变化情况），Δx有两部分变化，一是：在一维情况下变化量即微元dx线性正比于Δx，在更广泛的情况下，它是一个线性映射作用在△x上的值；二是：比△x更高阶的无穷小，也就是说除以△x后仍然会趋于零。当改变量很小时，第二部分可以忽略不计，函数的变化量约等于第一部分，也就是函数在x处的微分，记作df(x)或f'(x)dx。有这两种变化即为可微）。代入e = 2d + 1，平方根倒数y近似为。

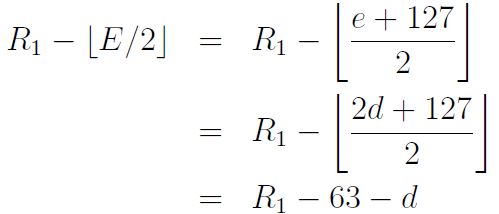
相对误差，这个误差也是我们用来衡量算法好坏的依据：，可简化为，。

注意，已经与d没有关系了，并且M可以取任何大于等于0小于1的值。在E位移正常的情况下，公式有效性只间接依赖于E。

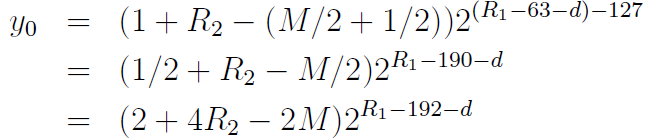
假设我们想要一个相对误差的R2。，展开：最后这步是基于(译注：>=1&&<2)。对上面展开的公式应用log2（译注：*logarithms*。如果ab=n，那么logan=b。其中，a叫做“底数”，n叫做“真数”，b叫做“以a为底的n的对数”。f(x)=logax函数叫做对数函数。对数函数中x的定义域是x>0，零和负数没有对数；a的定义域是a>0且a≠1）并增加190.5使190.7 > R1 > 189.2，于是R1 = 190 = 0xbe是唯一保持E的误差低于0.125的整型。第24到30位提供了这个神奇常量R的重要部分(0xbe>>1) = 0x5f（我们还需要23到0的部分）。

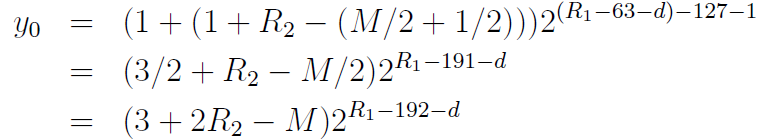


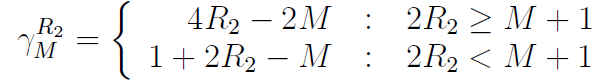
说明，Figure 1以实线表示y0，函数的近似值以虚线表示，当R2=0.43，d=2。显然y0不是线性递进的；然而，非线性实际上更合适！Figure 2展示了时的误差，同样显而易见最大误差的最小值约为R2=0.45，并且是值分布的分界线。

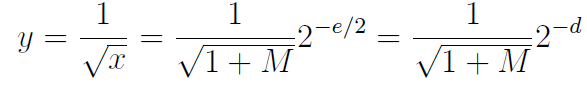
4.2 指数E位移进了尾数部分。与之前我们分析的情况相比，这个会复杂一些。i>>1执行后指数E位会移到尾数的高位部分。改变的尾数的真实值增加了0.5，变成了，其中部分作为整型（译注：此处不知道我是否理解正确了，原文是the truncation is as integers，我觉得是因为E位移导致尾数部分也被位移除以2了），增加部分作为浮点。同理，新的指数是。

前面讨论过这个结果必须是正数。

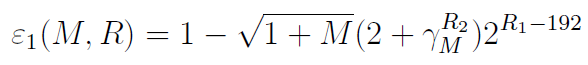
同理(译注:误差忽略不计)用替代，当，不借位的情况下尾数变成浮点的。于是。

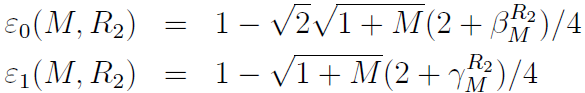
我们为y0选择同4.1中相同的指数。如果，减法需要从指数（正）中借一位，这个运算是的y0被除以2，并且i中尾数部分的位是浮点的。于是我们得到了时的初始估算值。

（同）通过我们可以将两种情况的y0一起计算。

注意是连续的，并且除了2R2 =M + 1以外是可微的，y0同样如此。通过e=2d,y0近似于。

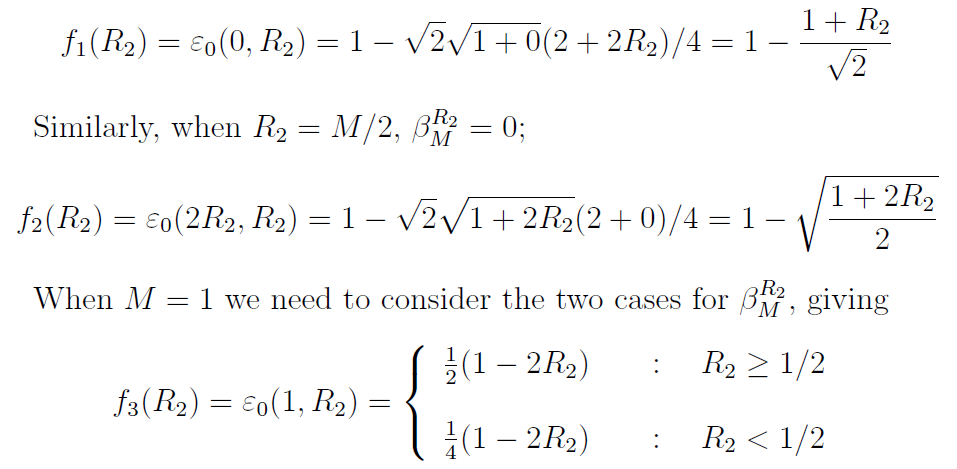
注意，这与我们在4.1中的近似算法不同；这里没有。

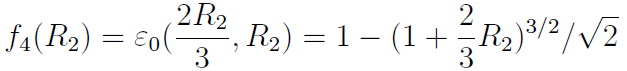
容忍相对误差（希望尽可能小）来简化为同前面的，再次与d无依赖（同样对E有隐含依赖）。

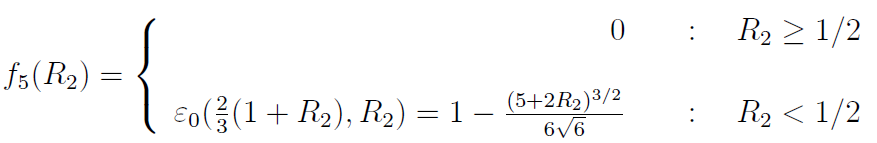
像之前一样，我们想让最大相对误差，我们可以用M=0（或1）去分析，但是既然我们想让相同的常数R1在这两种情况（译注：4.1,4.2）下工作，那么使用R1=190代入这两种情况。注意首要满足的条件是。于是对于R1 = 190 = 0xbe(译注：24-30)之外剩余的部分定义两个误差函数依赖于M和R2，取值范围在I中。我们已经清楚的明白了发生了什么，并且测试仔细确认了近似计算迭代模型，完美的完成了目标1）。

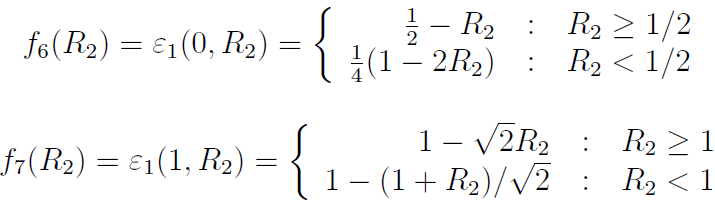
4.2.1 例子。X = 16,E= 4 + 127 = 131, M=0，并且指数中的一位位移到了尾数中。所以执行i = 0x5f3759df - (i>>1)后，初始预估值y0是0x3E7759DF。新的指数是，相当于e = 125 − 127 = −2。新的尾数需要从指数中借位，剩余e = -3，于是0xb759df - 0x4000000= 0x7759DF，对应的M = 0.932430148。于是y0 = (1 +M) 2-3 = 0.241553，这非常接近。

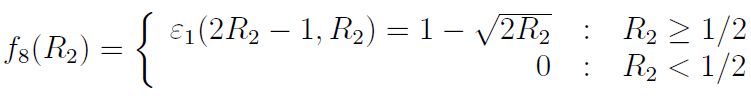
4.3 优化尾数。我们希望将情况下的最大误差降低。优化R2，我们会发现M的最大误差值。从到连续并且分段可微，这个M将出现在某个连接的断点或临界点上。

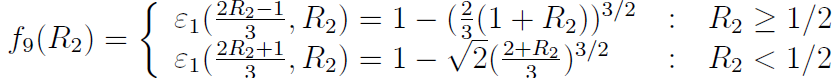
4.3.1 最大化。从的端点开始：M = 0, M =1, R2 = M/2(不可微)。当M = 0，。

检查两种情况的临界点。如果，发现临界点时，。

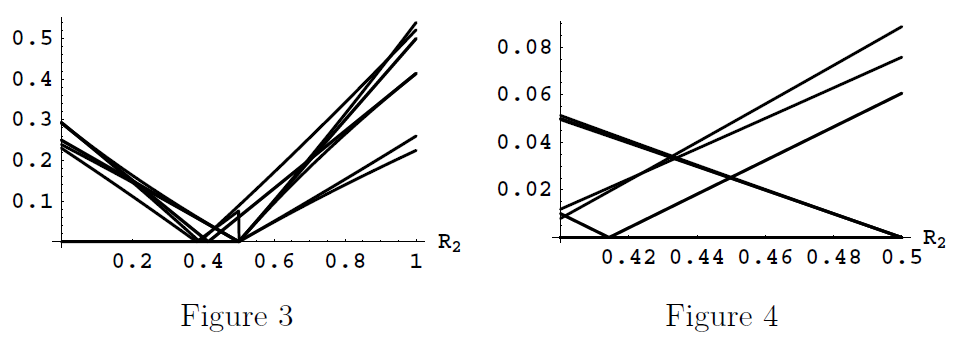
时，临界点是。只有R2取值范围在大于等于零小于二分之一是发生，于是。

4.3.2 最大化。相似的分析方法可得

在，，这只会在时发生，于是有

临界点也需要分两种情况。的临界点是。这只会出现在时。相似的，的情况下，的临界点是。两种情况合成（注意fi除f9外连续）

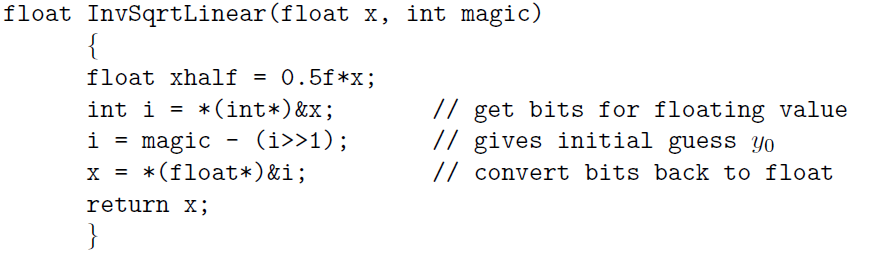
4.3.3 关于误差。对于给定的R2，以上9个函数可以得到9个值，最大的绝对值就是对于给定R2的最大误差。现在我们可以让R2在I的取值区间内变化以获得最好的取值。



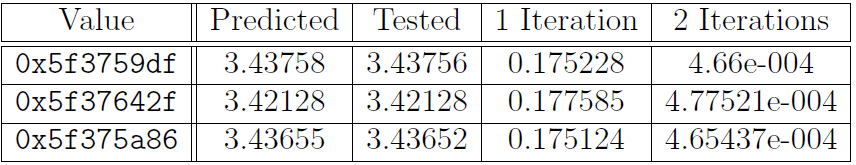
图三是一个数学软件[8]对误差函数的绘图。最大值在接近R2 = 0.45处最低；图四是对这部分的放大。使用Mathematica的FindMinimum功能求解R2对的结果中重叠的最小值是r0 = 0.432744889959443195468521587014保留这么多小数位是为了以后应用于任意位的浮点型：64位、128位等格式。参考结果部分。注意r0非常接近代码起初得到的近似值0.432430148。r0对应的整型。加上R1 = 0xbe，位移得0x5f（译注：0xbe>>1），得到最佳常量R = 0x5f37642f。

5.测试

我们使用下面这个方法来测试这个分析结果，只计算线性近似迭代（不用牛顿迭代！）。



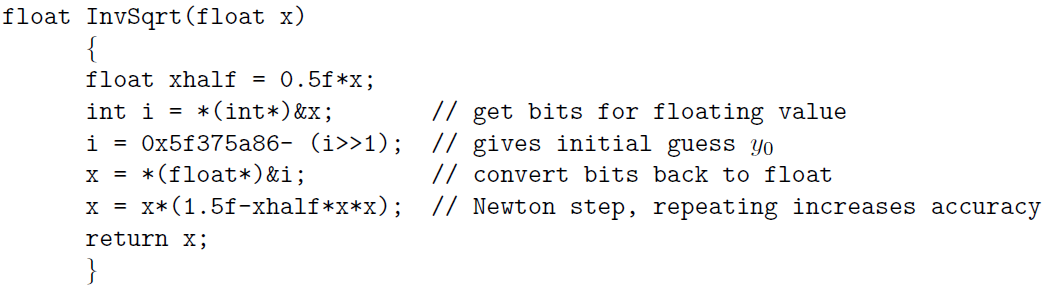
我们将这个方法应用到初始常量，推导出的常量和0x5f375a86（后面会解释原因）。同样，我们会用牛顿迭代对每个常量进行一到两个迭代（增加牛顿迭代次数消耗的时间很少，但是精度会大大增加）。每次测试都会覆盖所有浮点值。最大相对误差的比例是



这样在执行中得出的新常量来证明分析的推测是正确的。然而令人惊讶的是，在一次牛顿迭代后相对误差更大了。那么问题来了：最初的常量是怎么得到的？

牛顿迭代一定是的让更好的近似结果变坏的原因。新的常量理论和实际上都比初始的更好，但是在一两次牛顿迭代后初始的常量却表现的更好。出于好奇，我试验了更好的常量值。理论表明，任何好的近似值应该符合上面那几点，于是限制查找的范围。

从初始常量开始，测试所有常量值附近的值直到最大相对误差超过0x5f375a86的值0.00176。结果显示0x5f375a86相比0x5f3759df有更小的最大相对误差。因此，我建议优化的最终版本是



于是，最初的目标2）达到了，虽然并不完全符合计划。目标3）暂时保留：）

C++的代码和Mathematica 4.0参考[9]。

6.结论与问题

本文解释了代码中常量背后的魔法，并且试图理解以尽可能优化它。新的常量0x5f375a86表现似乎略有提高。由于两个相近，所以哪个执行的都很好。如果可能，我更想找到原作者，并且确认常量导出的方法或者只是通过猜测和测试得出的。

本文的作用是解释了面对不同的功能和平台如何去推导一个合适的常量值。用以上的推理方法，可以对几乎所有类似的功能进行分析。然而，分析应该在真实的机器上经过彻底的测试，因为真实的执行并不总和推测一样。

上述推论是大多数情况下是有兼容性的，可以使用与64位的double，或者通用的(译注：原文or even longer packed types to mimic SIMD instructions

without hardware support.我觉得是对硬件通用的，不需要特定硬件支持的意思)模拟SIMD指令集的更长包装类型。分析表明这方法可以扩展到很多平台和功能。

例如，double类型有11位指数，偏移量1023，并且有52位尾数。在通过相同形式的推导得到y0后，只需要结合R1，其余的分析是一样的。这种方式下常数r0可以应用到任意位大小。给定R1 = 1534，最好的初始近似值是。简单测试一下得到初始近似值的相对误差大约0.0342128，进行一次牛顿迭代后是0.0017758。

还有最后一个问题是为什么最好的初始近似值在一个牛顿迭代后不是最好的。

6.1家庭作业:几个问题：

1.用类似的方法推导sqrt(x)；

2.哪个更好（快？精确？）：double类型牛顿迭代两次后（译注：相对于一次）；

3.确认对于IEEE 754标准的64位double类型最好的初始近似值。

参考：

[1] www.gamedev.net/community/forums/topic.asp?topic id=139956

[2] www.msdn.microsoft.com/visualc/

[3] comp.graphics.algorithms, Search google with “0x5f3759df

group:comp.graphics.algorithms”

[4] David Eberly, www.magic-software.com/Documentation/FastInverseSqrt.pdf

[5] crystal.sourceforge.net/

[6] talika.eii.us.es/ titan/titan/index.html

[7] IEEE, grouper.ieee.org/groups/754/

[8] Mathematica - www.mathematica.com

[9] Chris Lomont, www.math.purdue.edu/\_clomont, look under math,

then under papers