平方根的计算方法

鲁尚宗

一、 介绍

开方是数值计算的一个基本运算。平方根的近似计算在中学教科书和实际中都有着重要的地位。本文详细论述了近似计算平方根的几种方法，并通过程序来比较不同计算方法的时间和精度。

二、 算法详解

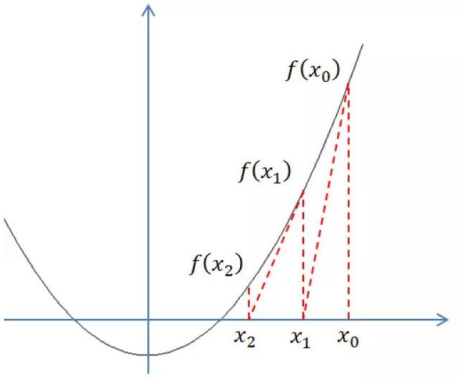
1、笔算

考虑计算 $\sqrt{23}$ 的近似值：由 $16<23<25$，得 $4<\sqrt{23}<5$，从而 $\sqrt{23}$ 介于 4 与 5 之间，因此可设 $\sqrt{23}=4+x$（其中 $0<x<1$）。两边平方，有 $23=(4+x)^2$，整理得 $23=16+8x+x^2$。因为 $0<x<1$，则 x^2 很小可忽略不计，那么有 $23\approx 16+8x$，解得 $x\approx 0.8$，于是 $\sqrt{23}\approx 4.8$。

为得到更精确的近似值，可再设 $\sqrt{23}=4.8+y$（y 是纯小数，可正可负）。两边平方有 $23=(4.8+y)^2$，整理得 $23=23.04+9.6y+y^2$，将 y^2 忽略不计得 $23\approx 23.04+9.6y$，解得 $y\approx -0.004$。于是 $\sqrt{23}\approx 4.8+(-0.004)=4.796$，与 $\sqrt{23}$ 的真值 4.79583...已经比较接近了。

2、牛顿法

牛顿法是一种在实数域和复数域上近似求解方程的方法。首先我们先来看函数图像。



首先，选择一个接近函数 **f(x)**零点的 **x0**,计算相应的 **f(x0)**和切线斜率 **f'(x0)**（这里 **f'**表示函数 **f** 的导数)。

也就是求如下方程的解:

$$0=(x-x_0)\cdot f'(x_0)+f(x_0)$$

我们将新求得的点 x 坐标命名为 x1，通常 x1 会比 x0 更接近方程 f(x)=0 的解。因此我们现在可以利用 x1 开始下一轮迭代。迭代公式可化简为如下所示：

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

而求平方根的方程我们可以看成 **f(x) = x^2 - a**,a 即为我们要求平方根的常数。

3、二分法

具体的做法如下：

这个非形式化过程可以用如下更形式化的自然语言表示：

1) 为了计算数x的平方根，从选择一个任意的猜测值g开始。一种可能就是将g设为x，尽管也可以选择任何其他的正数值。

2) 如果猜测值g足够接近于正确的平方根，算法结束，函数将g作为结果返回。

3) 如果g不够精确，用g和x/g的平均值作为新的猜测值。因为这两个值中的一个小于确切的平方根，另一个则大于确切的平方根，选择平均值会使你得到一个更接近于正确答案的值。

4) 把新的猜测值存入变量g，从第二步开始重复这个过程。

计算公式如下：

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$$

二分法的基本思想是逐步将非线性方程 f(x)=O 的有根区间（或隔根区间）二分，通过判断函数值的符号，逐步对半缩小有根区间（或隔根区间），直到区间缩小到容许误差范围之内，然后取区间的中点为根 x 的近似值。

4、级数逼近法

微积分中的泰勒级数如下：

$$f(x) \cong f(a) + f'(a)(x-a) + f''(a)\frac{(x-a)^2}{2!} + f'''(a)\frac{(x-a)^3}{3!} + \cdots + f^{(n)}(a)\frac{(x-a)^n}{n!}$$

在这个公式中，符号a表示一个常量，记号f'、f''和f'''表示函数f的一阶、二阶和三阶导数，以此类推。这个公式称为**泰勒公式**（Taylor's formula）。

$$\sqrt{x} \cong 1 + \frac{1}{2}(x-1) - \frac{1}{4}\frac{(x-1)^2}{2!} + \frac{3}{8}\frac{(x-1)^3}{3!} - \frac{15}{16}\frac{(x-1)^4}{4!} + \cdots$$

但是这儿存在一个问题，就是这个公式的收敛问题。它是存在收敛区间的。

作为泰勒级数展开，平方根函数的公式仅当参数值位于一个有限范围内时才有效，在该范围内计算趋于收敛。该范围被称为**收敛半径**（radius of convergence）。对平方根函数用$a=1$计算，泰勒级数公式希望x处于范围

$$0 < x < 2$$

之间。如果x在收敛半径外，则展开式中的项会越来越大，泰勒级数离答案也就越来越远。这个限制减小了TSqrt函数的用途，虽然在这个范围内它还是有效的。

当面临这种限制时，应该想一种方法把这个通用问题转化成某一特定问题，这个问题正好能满足你手上的这个方案的需要。因此，你需要找到一种根据落在范围内的平方根计算落在范围外的大数的平方根的方法。

为了实现这个问题的转换，参考下式：

$$\sqrt{4x} = \sqrt{4}\sqrt{x} = 2\sqrt{x}$$

由此可知，平方根内的4可以转换为平方根外的2。

这个结论提供了一种完成这个方案的工具。如果有一个极大的数，你可以用4去除它，用TSqrt计算余数的平方根，然后再用2乘以这个结果。如果除一次4还不能使它落在所期望的范围内，则在调用TSqrt前多除几次4，只要记住最后把结果再乘同样次数的2即可。

5、平方根倒数速算法（卡马克反转）

平方根倒数速算法（英语：Fast Inverse Square Root，亦常以“Fast InvSqrt()”或其使用的十六进制常数 0x5f3759df 代称）是用于快速计算平方根的倒数的一种算法。此算法最早可能是于 90 年代前期由 SGI 所发明，后来则于 1999 年在《雷神之锤 III 竞技场》的源代码中应用，但直到 2002－2003 年间才在 Usenet 一类的公共论坛上出现。这一算法的优势在于减少了求平方根倒数时浮点运算操作带来的巨大的运算耗费，而在计算机图形学领域，若要求取照明和投影的波动角度与反射效果，就常需计算平方根倒数。

一个浮点数是由32位二进制位表示的有理数，分为三部分。其中**符号**占1位，表示正负，记为**Si**；**指数**占接下来的8位，表示经过偏移处理后的指数，即实际表示E（如图中为124），需要偏移B（图中为2的8次方减1,127。B为一个固定值），最后得指数值为**E-B**；**有效数字（除最高位以外）**占剩下的23位，记为**m**（0<m<1）,图中的$m=1\times2^{-2}=0.250$。

所以浮点数的结构公式为：$x=(-1)^{\mathrm{Si}}\cdot(1+m)\cdot2^{(E-B)}$，图中$x=(1+0.250)\cdot2^{-3}=0.15625$

整数的表示相对简单，符号占1位，数值占剩下的31位。如果用上图的浮点数字节序列来表示整数，那么$I=E\times2^{23}+M$，即$I = 124 \times 2^{23} + 2^{21}$.平方根倒数函数仅能处理正数，所以符号位均为0。

小结：对于同样的32位二进制数码，若为浮点数表示时实际数值为$x=(1+m_x)2^{e_x}$，而若为整数表示时实际数值则为$I_x=E_xL+M_x$，其中$L=2^{n-1-b}$，

$$m_x = \frac{M_x}{L}$$

$$e_x = E_x - B，\text{其中}B = 2^{b-1} - 1$$

进一步说明：

$$y = \frac{1}{\sqrt{x}}$$

对等式的两边取二进制对数（$\log_2$，即函数$f(n) = 2^n$的反函数），有

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

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

以如上方法，就能将浮点数x和y的相关指数消去，从而将乘方运算化为加法运算。而由于$\log_2(x)$与$\log_2(x^{-1/2})$线性相关，因此在x与y_0（即输入值与首次近似值）间就可以线性组合的方式创建方程，在此McEniry再度引入新数σ描述$\log_2(1+x)$与近似值R间的误差：由于$0\le x<1$，有$\log_2(1+x)\approx x$，则在此可定义σ与x的关系为$\log_2(1+x)\cong x+\sigma$，这一定义就能提供二进制对数以首次精度值（此处$0 \le \sigma \le \frac{1}{3}$；当R为0x5f3759df时，有$\sigma=0.045046187579168701 1756$）。由此将$\log_2(1+x)=x+\sigma$代入上式，有：

$$m_y + \sigma + e_y = -\frac{1}{2}m_x - \frac{1}{2}\sigma - \frac{1}{2}e_x$$

参照首段等式代入M_x，E_x，B与L，有：

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

移项整理得：

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

如上所述，对于以浮点规格存储的正浮点数x，若将其作为长整型表示则示值为$I_x = E_xL + M_x$，由此即可根据x的整数表示导出y（在此$y = \frac{1}{\sqrt{x}}$，亦即x的平方根倒数的首次近似值）的整数表示值，也即：

$$I_y = E_yL + M_y = R - \frac{1}{2}(E_xL + M_x) = R - \frac{1}{2}I_x，\text{其中}R = \frac{3}{2}(B-\sigma)L.$$

此算法首先接收一个 32 位带符浮点数，然后将之作为一个 32 位整数看待，以将其向右进行一次逻辑移位的方式将之取半，并用十六进制“魔术数字”0x5f3759df 减之，**如此即可得对输入的浮点数的平方根倒数的首次近似值；而后重新将其作为浮点数，以牛顿法反复迭代，以求出更精确的近似值，直至求出匹配精确度要求的近似值**。在计算浮点数的平方根倒数的同一精度的近似值时，此算法比直接使用浮点数除法要快四倍。

此算法最早被认为是由约翰·卡马克所发明，但后来的调查显示，该算法在这之前就于计算机图形学的硬件与软件领域有所应用，如 SGI 和 3dfx 就曾在产品中应用此算法。而就现在所知，此算法最早由加里·塔罗利（Gary Tarolli）在 SGI Indigo 的开发中使用。虽说随后的相关研究也提出了一些可能的来源，但至今为止仍未能确切知晓算法中所使用的特殊常数的起源。

《雷神之锤 III》的代码直到 QuakeCon 2005 才正式放出，但早在 2002 年（或 2003 年）时，平方根倒数速算法的代码就已经出现在 Usenet 与其他论坛上了。最初人们猜测是卡马克写下了这段代码，但他在回复询问他的邮件时否定了这个观点，并猜测可能是先前曾帮 id Software 优化雷神之锤的资深汇编程序员 Terje Mathisen 写下了这段代码；而在 Mathisen 的邮件里，他表示，在 1990 年代初，他只曾作过类似的实现，确切来说这段代码亦非他所作。现在所知的最早实现是由 Gary Tarolli 在 SGI Indigo 中实现的，但他亦坦承他仅对常数 R 的取值做了一定的改进，实际上他也不是作者。在向以发明 MATLAB 而闻名的 Cleve Moler 查证后，Rys Sommefeldt 则认为原始的算法是 Ardent Computer 公司的 Greg Walsh 所发明，但他也没有任何决定性的证据能证明这一点。

不仅该算法的原作者不明，人们也仍无法确定当初选择这个“魔术数字”的方法。Chris Lomont 曾做了个研究：他推算出了一个函数以讨论此速算法的误差，并找出了使误差最小的最佳 R 值 0x5f37642f（与代码中使用的 0x5f3759df 相当接近），但以之代入算法计算并进行一次牛顿迭代后，所得近似值之精度仍略低于代入 0x5f3759df 的结果；因此 Lomont 将目标改为查找在进行 1－2 次牛顿迭代后能得到最大精度的 R 值，在暴力搜索后得出最优 R 值为 0x5f375a86，以此值代入算法并进行牛顿迭代，所得的结果都比代入原始值（0x5f3759df）更精确，于是他的论文最后提到“如果可能我想询问原作者，此速算法是以数学推导还是以反复试错的方式求得？”。在论文中，Lomont 亦指出，64 位的 IEEE754 浮点数（即双精度类型）所对应的魔术数字是 0x5fe6ec85e7de30da，但后来的研究表明，代入 0x5fe6eb50c7aa19f9 的结果精确度更高（McEniry 得出的结果则是 0x5FE6EB50C7B537AA，精度介于两者之间）。在 Charles McEniry 的论文中，他使用了一种类似 Lomont 但更复杂的方法来优化 R 值：他最开始使用穷举搜索，所得结果与 Lomont 相同；而后他尝试用带权二分法寻找最优值，所得结果恰是代码中所使用的魔术数字 0x5f3759df，因此，McEniry 认为，这一常数最初或许便是以“在可容忍误差范围内使用二分法”的方式求得。

牛顿迭代法是一种求方程的近似根的方法。首先要估计一个与方程的根比较靠近的数值，然后根据公式推算下一个更加近似的数值，不断重复直到可以获得满意的精度。其公式如下：

函数：y=f(x) 其一阶导数为：y'=f'(x)

则方程：f(x)=0 的第 n+1 个近似根为

x[n+1] = x[n] - f(x[n]) / f'(x[n])

牛顿迭代法最关键的地方在于估计第一个近似根。如果该近似根与真根足够靠近的话，那么只需要少数几次迭代，就可以得到满意的解。 现在回过头来看看如何利用牛顿法来解决我们的问题。求平方根的倒数，实际就是求方程 1/(x^2)-a=0 的解。将该方程按牛顿迭代法的公式展开为：

x[n+1]=1/2*x[n]*(3-a*x[n]*x[n])

三、 算法比较

用 c 语言编写程序用 2~5 方法计算平方根，与内置的函数进行比较。开平方的数在一万以内，为了统计时间每种方法循环 100000 次。将结果打印出来。迭代精度设置为 1e-10。

```
x=8145.00 y=  90.24965373895 <  2.00>,y1=  90.24965373895 < 24.00>,y2=  90.24908578247 <450.00>,y3=  90.24964904785 <  8.00>,y4=  90.24965373896 < 69.00>
x=3281.00 y=  57.28001396648 <  1.00>,y1=  57.28001396648 < 22.00>,y2=  57.28001396858 < 45.00>,y3=  57.28000640869 <  9.00>,y4=  57.28001396656 < 79.00>
x=6827.00 y=  82.62566187329 <  2.00>,y1=  82.62566187329 < 28.00>,y2=  82.62566187073 <133.00>,y3=  82.62535095215 <  8.00>,y4=  82.62566187320 < 67.00>
x=9961.00 y=  99.80480950335 <  1.00>,y1=  99.80480950335 < 25.00>,y2=  99.80480951645 < 63.00>,y3=  99.80441284180 <  8.00>,y4=  99.80480950336 < 81.00>
x= 491.00 y=  22.15851980616 <  1.00>,y1=  22.15851980616 < 24.00>,y2=  22.15851980534 <486.00>,y3=  22.15850830078 <  9.00>,y4=  22.15851980620 < 73.00>
x=2995.00 y=  54.72659317005 <  1.00>,y1=  54.72659317005 < 27.00>,y2=  54.72659317486 < 54.00>,y3=  54.72648620605 <  9.00>,y4=  54.72659316998 < 68.00>
x=1942.00 y=  44.06812907306 <  1.00>,y1=  44.06812907306 < 26.00>,y2=  44.06812907154 <369.00>,y3=  44.06809234619 <  9.00>,y4=  44.06812907309 < 83.00>
x=4827.00 y=  69.47661477073 <  2.00>,y1=  69.47661477073 < 31.00>,y2=  69.47661476813 < 49.00>,y3=  69.47660827637 < 10.00>,y4=  69.47661477079 < 88.00>
x=5436.00 y=  73.72923436467 <  2.00>,y1=  73.72923436467 < 28.00>,y2=  73.72923436850 < 58.00>,y3=  73.72920227051 <  9.00>,y4=  73.72923436464 < 84.00>
```

X 为计算的数值 ，y 为内置的函数计算结果， y1 为牛顿法计算结果，y2 为级数逼近法计算结果，y3 为平方根倒数法计算结果，y4 为二分法计算结果。<>里的数字为每种方法对应的时间。

可以看出内置的函数速度最快。牛顿法和二分法结果精确到了小数点后 10 位，平方根倒数法结果精确到小数点后 4 位，而级数逼近法的精度却不太稳定。从时间上来看，除了内置函数。平方根倒数的时间最短，牛顿法时间其次，级数逼近和二分法的时间最长。

四、 结论

平方根倒数法倒数的方法比较巧妙，在所需精度不高的情况下可以采用。牛顿法精度可以保证。在上个世纪 90 年代，平方根倒数的方法要比内置的函数快四倍。但随着计算机的发展，现代的 x86 和 x64cpu 都内置了一个 sqrt 指令，编译器开优化的话，就是一条汇编，这极大的提高了内置函数的效率。

五、 参考文献

赖志柱,吴德宝.平方根的几种近似计算方法[J].毕节学院学报,2013,31(04):43-50.

https://www.cnblogs.com/xkfz007/archive/2012/05/15/2502348.html

https://zh.wikipedia.org/wiki/%E5%B9%B3%E6%96%B9%E6%A0%B9%E5%80%92%E6%95%B0%E9%80%9F%E7%AE%97%E6%B3%95

https://diducoder.com/sotry-about-sqrt.html