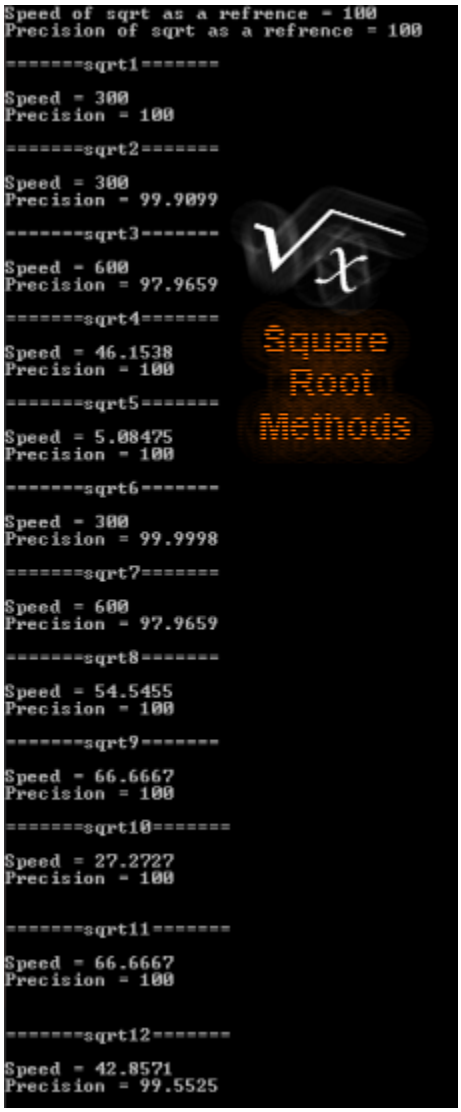
Best Square Root Method - Algorithm - Function (Precision VS Speed)

https://www.codeproject.com/Articles/69941/Best-Square-Root-Method-Algorithm-Function-Precisi

Mahmoud Hesham El-Magdoub Tue Sep, 07 02:00

- Download source - 5.28 KB



Introduction

I enjoy Game Programming with Directx and I noticed that the most called method throughout most of my games is the standard **sqrt** method in the *Math.h* and this made me search for faster functions than the standard **sqrt**. And after some searching, I found lots of functions that were much much faster but it's always a compromise between speed and precision. The main purpose of this article is to help people choose the best square-root method that suits their program.

Background

In this article, I compare 14 different methods for computing the square root with the standard **sqrt** function as a reference, and for each method I show its precision and speed compared to the **sqrt** method.

What this Article is Not About

1. Explaining how each method works
2. New ways to compute the square root

Using the Code

The code is simple, it basically contains:

1. main.cpp

Calls all the methods and for each one of them, it computes the speed and precision relative to the **sqrt** function.

2. SquareRootmethods.h

This Header contains the implementation of the functions, and the reference of where I got them from.

First I calculate the Speed and Precision of the **sqrt** method which will be my reference.

For computing the Speed, I measure the time it takes to call the **sqrt** function (M-1) times and I assign this value to `RefSpeed` which will be my reference.

And for computing the Precision, I add the current result to the previous result in `RefTotalPrecision` every time I call the method. `<code>RefTotalPrecisi` on will be my reference.

For measuring runtime duration (Speed) of the methods, I use the `CDuration` class found in this link, and I use `dur` as an instance of that class.

```
for(int j=0;j<AVG;j++)
{
  dur.Start();

        for(int i=1;i<M;i++)
           RefTotalPrecision+=sqrt((float) i);

  dur.Stop();

  Temp+=dur.GetDuration();
}

RefTotalPrecision/=AVG;
Temp/=AVG;

RefSpeed=(float)(Temp)/CLOCKS_PER_SEC;
```

And for the other methods I do the same calculations, but in the end, I reference them to the **sqrt**.

```
        for(int j=0;j<AVG;j++)
        {
          dur.Start();

                for(int i=1;i<M;i++)
                    TotalPrecision+=sqrt1((float) i);

          dur.Stop();

          Temp+=dur.GetDuration();
        }

        TotalPrecision/=AVG;
        Temp/=AVG;

        Speed=(float)(Temp)/CLOCKS_PER_SEC;

cout<<'Precision = '
<<(double)(1-abs((TotalPrecision-RefTotalPrecision)/(RefTotalPrecision)))*100<<endl;
```

NOTES:

1. I assume that the error in Precision whether larger or smaller than the reference is equal, that's why I use 'abs'.
2. The Speed is referenced as the actual percentage, while the Precision is referenced as a decrease percentage.

You can modify the value of `M` as you like, I initially assign it with 10000.

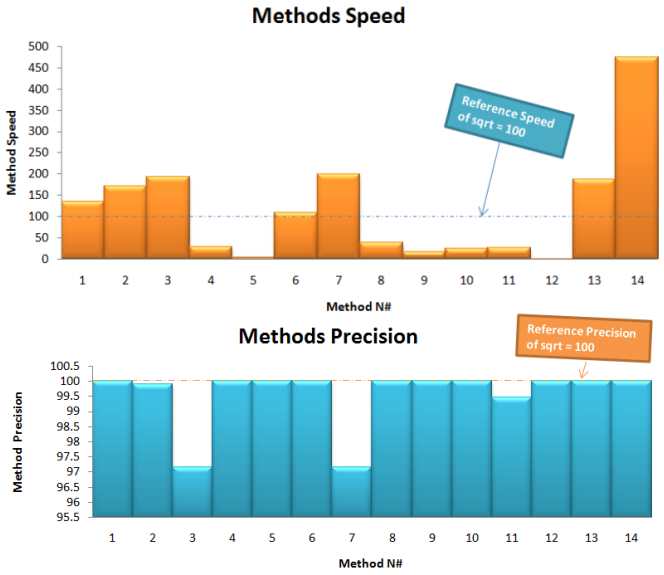
You can modify `AVG` as well, the higher it is, the more accurate the results.

```
#define M 10000
#define AVG 10
```

Points of Interest

Precision wise, the **sqrt** standard method is the best. But the other functions can be much faster even 5 times faster. I would personally choose Method N# 14 as it has high precision and high speed, but I'll leave it for you to choose. :)

I took 5 samples and averaged them and here is the output:



According to the analysis the above Methods Performance Ranks (Speed x Precision) is:

Performance3.PNG

NOTE: The performance of these methods depends highly on your processor and may change from one computer to another.

The METHODS

Sqrt1

Reference: http://ilab.usc.edu/wiki/index.php/Fast_Square_Root

Algorithm: Babylonian Method + some manipulations on IEEE 32 bit floating point representation

```
float sqrt1(const float x)
{
  union
  {
    int i;
    float x;
  } u;
  u.x = x;
  u.i = (1<<29) + (u.i >> 1) - (1<<22);

        u.x =       u.x + x/u.x;
  u.x = 0.25f*u.x + x/u.x;

  return u.x;
}
```

Sqrt2

Reference: http://ilab.usc.edu/wiki/index.php/Fast_Square_Root

Algorithm: The Magic Number (Quake 3)

```
#define SQRT_MAGIC_F 0x5f3759df
 float  sqrt2(const float x)
{
  const float xhalf = 0.5f*x;

  union   {
    float x;
    int i;
  } u;
  u.x = x;
  u.i = SQRT_MAGIC_F - (u.i >> 1);    return x*u.x*(1.5f - xhalf*u.x*u.x);}
```

Sqrt3

Reference: http://ilab.usc.edu/wiki/index.php/Fast_Square_Root

Algorithm: Log base 2 approximation and Newton's Method

```
float sqrt3(const float x)
{
  union
  {
    int i;
    float x;
  } u;

  u.x = x;
  u.i = (1<<29) + (u.i >> 1) - (1<<22);
  return u.x;
}
```

Sqrt4

Reference: I got it a long time a go from a forum and I forgot, please contact me if you know its reference.

Algorithm: Bakhsali Approximation

```
float sqrt4(const float m)
{
   int i=0;
   while( (i*i) <= m )
          i++;
    i--;
   float d = m - i*i;
 float p=d/(2*i);
 float a=i+p;
   return a-(p*p)/(2*a);
}
```

Sqrt5

Reference: http://www.dreamincode.net/code/snippet244.htm

Algorithm: Babylonian Method

```
   float sqrt5(const float m)
{
   float i=0;
   float x1,x2;
   while( (i*i) <= m )
          i+=0.1f;
   x1=i;
   for(int j=0;j<10;j++)
   {
       x2=m;
      x2/=x1;
      x2+=x1;
      x2/=2;
      x1=x2;
   }
   return x2;
}
```

Sqrt6

Reference: http://www.azillionmonkeys.com/qed/sqroot.html#calcmeth

Algorithm: Dependant on IEEE representation and only works for 32 bits

```
double sqrt6 (double y)
{
    double x, z, tempf;
    unsigned long *tfptr = ((unsigned long *)&tempf) + 1;
    tempf = y;
   *tfptr = (0xbfcdd90a - *tfptr)>>1;
 x =  tempf;
 z =  y*0.5;
 x = (1.5*x) - (x*x)*(x*z);                                    x = (1.5*x) -
(x*x)*(x*z);
  return x*y;
  }
```

Sqrt7

Reference: http://bits.stephan-brumme.com/squareRoot.html

Algorithm: Dependant on IEEE representation and only works for 32 bits

```
float sqrt7(float x)
 {
   unsigned int i = *(unsigned int*) &x;
      i  += 127 << 23;
      i >>= 1;
   return *(float*) &i;
 }
```

Sqrt8

Reference: http://forums.techarena.in/software-development/1290144.htm

Algorithm: Babylonian Method

```
double sqrt9( const double fg)
{
 double n = fg / 2.0;
 double lstX = 0.0;
 while(n != lstX)
 {
 lstX = n;
 n = (n + fg/n) / 2.0;
 }
 return n;
 }
```

Sqrt9

Reference: http://www.functionx.com/cpp/examples/squareroot.htm

Algorithm: Babylonian Method

```
 double Abs(double Nbr)
{
 if( Nbr >= 0 )
  return Nbr;
 else
  return -Nbr;
}

double sqrt10(double Nbr)
{
 double Number = Nbr / 2;
 const double Tolerance = 1.0e-7;
 do
 {
  Number = (Number + Nbr / Number) / 2;
 }while( Abs(Number * Number - Nbr) > Tolerance);

 return Number;
}
```

Sqrt10

Reference: http://www.cs.uni.edu/~jacobson/C++/newton.html

Algorithm: Newton's Approximation Method

```
double sqrt11(const double number)e
{
const double ACCURACY=0.001;
double lower, upper, guess;

 if (number < 1)
 {
  lower = number;
  upper = 1;
 }
 else
 {
  lower = 1;
  upper = number;
 }

 while ((upper-lower) > ACCURACY)
 {
  guess = (lower + upper)/2;
  if(guess*guess > number)
   upper =guess;
  else
   lower = guess;
 }
 return (lower + upper)/2;

}
```

Sqrt11

Reference: http://www.drdobbs.com/184409869;jsessionid=AIDFL0EBECDYLQE1GHOSKH4ATMY32JVN

Algorithm: Newton's Approximation Method

```
 double sqrt12( unsigned long N )
{
    double n, p, low, high;
    if( 2 > N )
        return( N );
    low  = 0;
    high = N;
    while( high > low + 1 )
    {
        n = (high + low) / 2;
        p = n * n;
        if( N < p )
            high = n;
        else if( N > p )
            low = n;
        else
            break;
    }
    return( N == p ? n : low );
}
```

Sqrt12

Reference: http://cjjscript.q8ieng.com/?p=32

Algorithm: Babylonian Method

```
double sqrt13( int n )
{
                double a = (double) n;
        double x = 1;

                for( int i = 0; i < n; i++)
        {
                x = 0.5 * ( x+a / x );
        }

        return x;
}
```

Sqrt13

Reference: N/A

Algorithm: Assembly fsqrt

```
double sqrt13(double n)
{
  __asm{
     fld n
     fsqrt
   }
}
```

Sqrt14

Reference: N/A

Algorithm: Assembly fsqrt 2

```
double inline __declspec (naked) __fastcall sqrt14(double n)
{
        _asm fld qword ptr [esp+4]
        _asm fsqrt
        _asm ret 8
}
```

History

1.3 (15 September 2010)

- Added Method N#14 (which is the best method till now)
- Added modified source code

1.2 (24 June 2010)

- Added Method N#13
- Added the Methods Performance Rank
- Added modified source code

1.1 (3 April 2010)

- Added Precision Timer instead of clock because it's more precise
- Added the average feature

1.0 (31 March 2010)

- Initial release

I hope that this article would at least slightly help those who are interested in this issue.