LGS: C++ QuickStart

Introduction

The idea of this lecture is to get you from possibly knowing nothing about the language to being able to work out the Black-Scholes model in C++. This presentation is of necessity a little skeletal - I will try to get across some key facts and constructions that will get you going as quickly as possible, but you should flesh this out by reading around in due course.

We will use the Bloodshed compiler so start it right away.

Once you have got it started, in a browser navigate to the secure area of the web site:

http://www.mth.kcl.ac.uk/~shaww/web_page/lgs/notes/cppfiles

In here you can find a zip files called cppexamples.zip. Ideally you will download and unzip this file and get all the example programs at once. Otherwise we can retrieve them one at a time, starting with

hello.cpp

from the link indicated under "C++ programming material". **Save it in your "My Documents" area or somewhere else you can remember and open it from within Dev C++.** This file contains the following program

```
#include <iostream>
using namespace std;

int main()
{
    char name;
    cout << " Hello Compfin student " << endl;
    cout << " Hit any key+<RET> to finish \n " ;
    cin >> name;
    return(0);
}
```

First we shall compile and run this program, the talk about how it works.

Go to the Dev-C++ menu "Execute" and pick the "Compile and Run" option. There is a shortcut which is to just hit F9. Let's get this going....

Comments on this program:

1. The basic C++ system does not have a lot of things you need to get going. So you ALWAYS have to literally "include" some library functions. To comunicate with a terminal (here DOS) window you have the following command, which includes the standard (<>) library iostream.

#include <iostream>

2. The using namespace std; line allows you to use the short form of standard functions such as cout and endl

3. The next line is int main(). All C++ programs have this for the main part of the code. Programs return a value, here an integer denoted int, or nothing, in which case they are declared as void. Here the main block ends with return(0) which means that the program returns the integer zero.

4. All program statements are separated with a semi-colon, ;

5. << is used to separate items in an output stream.

6. Text is delimited with " marks.

7. >> is used to separate items in an input stream.

8. endl ends a line in the output.

9. \n is a newline character you can put in a string of text.

10. To stop this program ending before you can even see what has happened I have put in a line that wants an input in the form of a key stroke; to this end I have declared a character variable name so as to serve as the keystroke variable.

Strings

Close hello.cpp and go and fetch helloname.cpp. Here it is

```
#include <iostream>
#include <string>
using namespace std;


int main()
{
    string myname;
    char keystroke;
    cout << " What is your name" << endl;
    cin >> myname;
    cout << " Hello " << myname << endl ;
    cout << " Hit any key+<RET> to finish \n" ;
    cin >> keystroke;
    return(0);
}
```

If we include the string library we can use a string variable which does NOT require us to specify how long it might be. Your name might be very long for example. Ever had an internet form which stops you typing in your full name or address??

Numbers 1: Integers and float types and basic arithmetic

So the first thing a Quant needs to be able to do is have numbers and arithmetic in their program. The simplest relevant data types are int and float and you can do simple arithmetic with the operators +,-,*,/ for addition, subtraction, multiplication and division. SO go and get arithmeticone.cpp, which has the following code in it:

```
#include <iostream>

using namespace std;


int main()

{

    char keystroke;

    float a,b,sum,product,difference,ratio,average;

    int n=2;

    cout << " Type in two positive numbers \n" ;

    cin >> a >> b;

    sum = a+b;

    product = a*b;

    difference = a-b;

    ratio = a/b;

    average = sum/n;

    cout << " Your numbers were " << a << " and " << b << endl ;

    cout << " Their sum is " << sum << endl ;

    cout << " Their product is " << product << endl ;

    cout << " Their difference is " << difference << endl ;

    cout << " Their ratio is " << ratio << endl ;

    cout << " Their arithmetical average is " << average << endl ;

    cout << " Hit any key+<RET> to finish \n" ;

    cin >> keystroke;

    return(0);

}
```

Saving your results to a file

You will very rapidly progress to have screenfulls of unsaved data once you get going, so it is a good idea early on to have a basic idea of writing the results to a file. Including one further library makes this possible in exactly the same way you write to the screen. You just have to name the file. The example in arithmeticonefile.cpp makes this clear, and also adds the idea of putting in comments. Here it is:

```
// This program does some basic arithmetic and writes the result to a file

// It has comments like this one!

// The double slash denotes precisely one line of comments.

#include <iostream>

#include <fstream>

// The latter include statement allows file management
```

```
using namespace std;


int main()
{
/*
If I want my comments to
span several lines I use these star-slash delimiters
*/
    char keystroke;
    float a,b,sum,product,difference,ratio,average;
    int n=2;
    cout << " Type in two positive numbers \n" ;
    cin >> a >> b;
    sum = a+b;
    product = a*b;
    difference = a-b;
    ratio = a/b;
    average = sum/n;
// Now deal with output to a file - hard coded for now:
    ofstream out("myoutput.txt");
// send the results to it:
    out << " Your numbers were " << a << " and " << b << endl ;
    out << " Their sum is " << sum << endl ;
    out << " Their product is " << product << endl ;
    out << " Their difference is " << difference << endl ;
    out << " Their ratio is " << ratio << endl ;
    out << " Their arithmetical average is " << average << endl ;
// Use ordinary terminal output for the closing keystroke
    cout << " Hit any key+<RET> to finish \n" ;
    cin >> keystroke;
    return(0);
}
```

Numbers, Arithmetic and Elementary Mathematics in C++

So at this point you have seen how to do some very basic input and output, basic arithmetic and save the results to a file. Now we need to expand our understanding of basic arithmetic and mathematics. We first need to get a grip on the types of numbers. The matter is frankly somewhat unsatisfactory as

(a) the standard for C++ does not enforce differences between type variations that you would imagine is a basic requirement;

(b) complex numbers are not managed as a standard at all

(c) there is increasing chaos and variation as you introduce vectors, matrices and higher order stuff.

For "whole numbers" I will NOT get into the variations on integers and will just use the type "int" everywhere in these notes. You might like to look up short int and long int and unsigned variations in the notes and in books.

For real numbers there are some important things to think about. There are three basic types:

float

double

long double

The C++ standard does not require that these things are necessarily different from one another. It is a good idea to run the following code on any system you plan to use to get an idea of what is going on! This is realtypes.cpp:

```
#include <iostream>


using namespace std;


int main()
{
    char keystroke;
    float num = 1.0/7;
    double num_d=1.0/7;
    long double num_ld=1.0/7;
    cout.precision(20);
    cout << " The float is       : " << num << endl ;
    cout << " On the fly gives   : " << 1.0/7 << endl ;
    cout << " The double is      : " << num_d << endl ;
    cout << " The long double is : " << num_ld << endl ;
    cout << " Hit any key+<RET> to finish \n" ;
    cin >> keystroke;
    return(0);
}
```

If you look at the output it is clear that there is nothing extra in long double in this system. We shall use double everywhere from now on. You should at some point read up on ranges of numbers and associated issues. I sneaked in a statement: cout.precision(20) in order to make it possible to see the difference. Find out what happens if you comment out this line (Do it NOW!).

Arithmetical Operations

We already looked at the basics of

+ ,addition,

- subtraction,

* times,

/ divide.

It is possible to combine these with assignment (= in C++) to write shorter code: So if, e.g. i is an integer and you want to add two to i, you do not have to write

i = i+2

rather, you can say

i+=2

Simlarly with the other operators. Frequently you may be stepping through something in steps of one, in which case you would say not

i=i+1

or indeed, not

i+=1

but just

i++

Run the following example (incrementing.cpp) just to make sure of things

```
#include <iostream>
using namespace std;
int main()
{
    char keystroke;
    int i=0, j=0, k=0;
    i=i+1;
    j+=1;
    k++;
    cout << i << j << k << endl;
    cout << " Hit any key+<RET> to finish \n" ;
    cin >> keystroke;
    return(0);
}
```

You often want to use these shorthands for incrementation in loops/iterations.

"Standard" Maths functions

To do just about anything mathematical you will need to load the math library with a statement of the form

#include <cmath>

at the top of the code. Just about every program will want to have this loaded as it contains the really basic stuff like sqrt for square root! Here are the most useful bits - see

www.cplusplus.com/reference/clibrary/cmath for some more details

cos, sin, tan,

acos, asin, atan, atan2

cosh, sinh, tanh

exp, log, log10, modf

pow, sqrt

ceil,fabs,floor,fmod

You have to have this loaded (again!). Note that stuff we might consider basic like the cumulative normal distribution is not there, let alone stuff like Bessel functions or hypergeometric functions.....(ever valued an Asian option?). Don't get lazy with these functions. E.g. the polynomial $x^2 + x + 1$should be worked out NOT with

```
pow(x,2)+x+1
```

but rather with

```
1+x*(1+x)
```

and this type of nesting is always desirable. Rather than doing some boring examples calling various functions we shall see some of them in use later.

Basic Control Structures in C++

Programs often need to do similar things many times or make choices or do something until a test fails, and so on. All of this is managed by a variety of control structures. The first thing you need to understand is Boolean structures, and these are best illustrated by their use within an if statement. A Boolean statement should be decidable to be true or false. For example, if i and j are integers precisely one of the following is true:

i>j

i==j

i<j

but your decisions could also be based on >= or <= for example and note that just as in *Mathematica* the double equals plays the role of denoting equality, as opposed to a single = that is used for assignment. Here is a simple program, ifdeciding.cpp that makes a simple comment about the size of a number and accordingly decides to square or cube it:

If then else example

```
#include <iostream>
/* This program illustrates the use of the if statement;
Note: A. blocks are grouped with {}; B. else is optional
*/
using namespace std;

int main()
{
    char keystroke;
    int i;
    cout << " Enter an integer \n ";
    cin >>i;
    if (i>= 100)
       {cout << "that is quite a big number \n";
       cout << "its square is " << i*i << endl;}
       else
       {cout << "that is not a big number \n";
       cout << "its cube is " << i*i*i << endl;}
    cout << "Hit any key+<RET> to finish \n" ;
    cin >> keystroke;
    return(0);
}
```

Switch case example

You can of course build up some complicated logic based on a collection of if-else statements. But if you have three or more possibilities to contend with it MIGHT be more elegant to use the "switch" structure PROVIDED the choices can be simplified to just matching to a finite set of constants. This can be particularly useful if you have an algorithm that may take simple forms for certain paramaters otherwise it reverts to a general system. You need to explicitly "break" out of the tests once you have carried out the branch of interest. Go get switchdeciding.cpp

```
#include <iostream>
/* This program illustrates the use of the switch case statement;
*/
using namespace std;

int main()
{
    char keystroke;
    int i;
    cout << " Enter a positive integer \n ";
```

```
    cin >>i;
    switch (i)
    {
      case 1: cout << "You entered one \n";
      break;
      case 2: cout << "You entered two \n";
      break;
      case 3: cout << "You entered three \n";
      break;
      case 4: cout << "You entered four \n";
      break;
      default: cout << "You entered something other than one to four\n";
      break;
    }
    cout << "Hit any key+<RET> to finish \n" ;
    cin >> keystroke;
    return(0);
}
```

While examples

This is very important in financial maths as it is frequently used in iterative processes, for example to solve an equation numerically. A program might keep executing "While" an error is bigger than some tolerance. A classic example would be to find the square root of a number. If we input a number c and want to solve the equation

$$f(x) = x^2 - c = 0 \tag{1}$$

This equation may be solved iteratively by Newton-Raphson methods with the scheme

$$x_n = \frac{x_{n-1}}{2} + \frac{c}{2\,x_{n-1}} \tag{2}$$

We will give two programs to solve this. The first is whileiteration.cpp and is as follows:

```
#include <iostream>
/* This program uses the while statement to solve an equation iteratively;
*/
using namespace std;

int main()
{
    char keystroke;
    double c, x, y=1.0;
    cout.precision(20);
    cout << " Enter a positive number \n ";
    cin >> c;
    while (x!=y)
    {
```

```
            x=y;
            y = x/2+c/(2*x);
      }
    cout << "the square root of your number is " << y <<endl;
    cout << "its square gives back " << y*y << endl;
    cout << "Hit any key+<RET> to finish \n" ;
    cin >> keystroke;
    return(0);
}
```

It will often be the case that you want to stop at a certain pre-defined tolerance, in which case the following slightly cruder while statement would be used. This is a very common construction. This is whileiterationcruder.cpp:

```
#include <iostream>
#include <cmath>
/* This program uses the while statement to solve an equation iteratively;
The termination criteris is now set more tolerantly with the fabs math function.
*/
using namespace std;

int main()
{
    char keystroke;
    double c, x, y=1.0;
    cout.precision(20);
    cout << " Enter a positive number \n ";
    cin >> c;
    while (fabs(x-y)>1.0e-2)
    {
            x=y;
            y = x/2+c/(2*x);
      }
    cout << "the square root of your number is " << y <<endl;
    cout << "its square gives back " << y*y << endl;
    cout << "Hit any key+<RET> to finish \n" ;
    cin >> keystroke;
    return(0);
}
```

Do-While example

Note that in the examples of while the execution might never take place - the test is at the beginning. There is a variation on this which says

do

{commands}

while (test);

and the loop keeps on executing until the test fails. The commands will always run at least once. You can usually achieve this within an ordinary while by initialising matters suitably, as in the above example. But let's just rewrite our solver to use do while to see it working! This is dowhileiteration.cpp

```
#include <iostream>
#include <cmath>
/* This program uses the do while statement to solve an equation iteratively;
*/
using namespace std;

int main()
{
    char keystroke;
    double c, x, y=1.0;
    cout.precision(20);
    cout << " Enter a positive number \n ";
    cin >> c;
    do
    {
          x=y;
          y = x/2+c/(2*x);
     }
    while (fabs(x-y)>1.0e-2);
    cout << "the square root of your number is " << y <<endl;
    cout << "its square gives back " << y*y << endl;
    cout << "Hit any key+<RET> to finish \n" ;
    cin >> keystroke;
    return(0);
}
```

For loops

These can be used to do all types of numerical modelling where one wants to set up tree or finite-difference models and work around a grid. For now we just see its use in setting up a loop of a DEFINED length. In foriteration.cpp we outupt powers of the integers from 0 to 9.

```
#include <iostream>

/* This program uses the for statement to table some numbers

*/

using namespace std;


int main()

{

    char keystroke;

    int i,j;

    for (i=0; i<10; i++)

    {     j=i*i;

          cout << "number " << i << ", square "<< j << endl;

          cout << "cube " << i*j << ", fourth power " << j*j << endl;

     }

    cout << "Hit any key+<RET> to finish \n" ;

    cin >> keystroke;

    return(0);

}
```

Simple User-Defined Functions in C++

There are many levels at which one can consider this topic. To get going on e.g. working out the Black-Scholes model we really need two functions

1. A function for the cumulative Normal Distribution;

2. A function for the Black-Scholes formula.

The latter can be extended to all kinds of other functions, e.g. for the partial derivatives.

Rather than faff with some contrived basic examples let's go straight for the Normal Distribution function first. The we shall have a function calling this function which may as well be the Black-Scholes model.

Joshi's and Marsaglia's C++ Implementation of the Normal CDF

A C++ version of the second rational approximation has been made publicly available by M. Joshi (though he has said he used something else professionally!). Below in the program I haved included the relevant *extract* from Normals.cpp, downloadable from his web site (the link to it is in the /notes/ directory). I have deleted a couple of blocks not relevant to this discussion. Note that Joshi (a) takes special care of the tails (b) works out the polynomial piece more efficiently.

Marsalglia's Series

The matter of approximating the cumulative Normal was revisited and surveyed by G. Marsaglia in 2004 (Journal of Statistical Software, July 2004, Vol 11 Issues 4), available on the web at:

www.jstatsoft.org/v11/i04/v11i04.pdf

and he came up with a C programme based on the result that

$$\Phi(x) = \frac{1}{2} + \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \left(x + \frac{x^3}{3} + \frac{x^5}{3\times 5} + \frac{x^7}{3\times 5\times 7} + \frac{x^9}{3\times 5\times 7\times 9} + \ldots \right) \tag{3}$$

This has been known for a long time (see formula 26.2.11 of Abramowitz and Stegun). Here is the c code for it:

```
double PhiMarsaglia(double  x)


{long double s=x,t=0,b=x,q=x*x,i=1;
  while(s≠t)  s=(t=s)+(b*=q/(i+=2));
  return  0.5+s*exp(-0.5*q-0.91893853320467274178L);}
```

We have these two functions together in a program where the main block works out the values of the CDFs with the two models and outputs them to a file. The code is cumulativenormals.cpp:

```
//        CumulativeNormals.cpp
// code to work out the normal pdf and cdf
// the latter via a rational approximation (Joshi's code) and Marsaglia's series


//First include is all that is needed for function definitions
// Rest is for I/O


#include <cmath>
#include <iostream>
#include <fstream>


using namespace std;
```

```
// phinorm is constant value of one over root 2 pi.

const double phinorm = 0.398942280401433;

// pdf
double phi(double  x)
{
    return phinorm*exp(-x*x/2);
}

double PhiRational(double  x)
{
    static double a[5] = { 0.319381530,
                          -0.356563782,
                           1.781477937,
                          -1.821255978,
                           1.330274429};

    double result;

    if (x<-7.0)
        result = phi(x)/sqrt(1.+x*x);

    else
    {
        if (x>7.0)
            result = 1.0 - PhiRational(-x);
        else
        {
            double tmp = 1.0/(1.0+0.2316419*fabs(x));

            result=1-phi(x)*
                     (tmp*(a[0]+tmp*(a[1]+tmp*(a[2]+tmp*(a[3]+tmp*a[4])))));

            if (x<=0.0)
                result=1.0-result;

        }
    }

    return result;
}

/* code for PhiRational is renamed version of
```

```
 * code Copyright (c) 2002
 * Mark Joshi
 *
 * Permission to use, copy, modify, distribute and sell this
 * software for any purpose is hereby
 * granted without fee, provided that the above copyright notice
 * appear in all copies and that both that copyright notice and
 * this permission notice appear in supporting documentation.
 * Mark Joshi makes no representations about the
 * suitability of this software for any purpose. It is provided
 * "as is" without express or implied warranty.
*/

/* The following is the compact code provided by Marsaglia in his 2004 paper!
*/

double PhiMarsaglia(double  x)

{long double s=x,t=0,b=x,q=x*x,i=1;
while(s!=t) s=(t=s)+(b*=q/(i+=2));
return 0.5+s*exp(-0.5*q-0.91893853320467274178L)
    ;
}



#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    double result;
    char keystroke;
    int N;
    int j;
    double range;
    double x;
    double ratphi;
    double marsphi;

    cout << "Test computations of cumulative normal distributions \n ";
    cout << "Rational approximation (Joshi implementation) \n";
    cout << "Marsaglia approximation (Marsaglia implmentation) \n";
    cout << "Enter number of positive and negative sample points (N) \n ";
    cin >> N;
```

```
    cout << "Enter range parameter (range) \n ";
    cin >> range;
    cout << " Result being written to normaloutput.txt \n " << endl;

    ofstream out("normaloutput.txt");
    for (j=0; j<= 2*N; j++)
    {
        x = -range + j*range/N;
        ratphi = PhiRational(x);
        marsphi = PhiMarsaglia(x);

    out.precision(15);
        out << x << " " << ratphi << " " << marsphi << "\n";
        }
    cout << "Hit any key+<RET> to finish \n" ;
    cin >> keystroke;
    return(0);
}
```

We proceed to compile and run the C code cumulativenormals.cpp and look at the output, with N=1000, range = -7 to 7. This code runs both the rational approximation and the Marsaglia formula over the range and outputs triples {x, rational(x), Marsaglia(x)}.

```
SetDirectory[
 "C:\Documents and Settings\William Shaw\My Documents\LGS0708"]
```

C:\Documents and Settings\William Shaw\My Documents\LGS0708

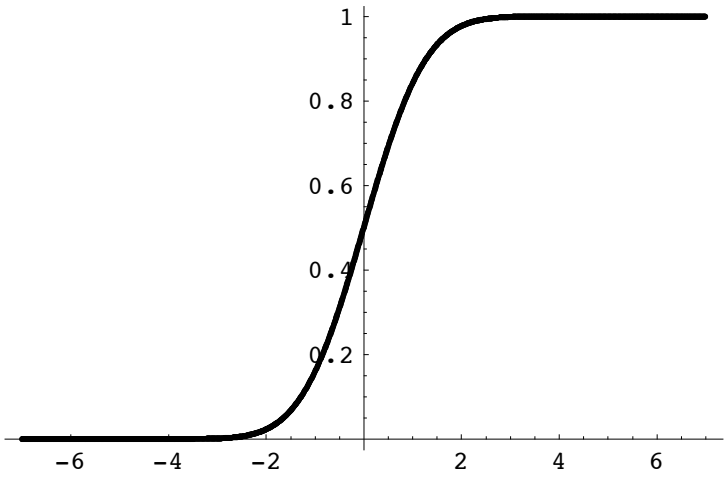
```
FileNames[]
```

{arithmeticone.cpp, arithmeticone.exe, arithmeticonefile.cpp, arithmeticonefile.exe, blackscholes.cpp, cppexamples.zip, cumulativenormals.cpp, cumulativenormals.exe, dowhileiteration.cpp, foriteration.cpp, hello.cpp, hello.exe, helloname.cpp, helloname.exe, ifdeciding.cpp, incrementing.cpp, Lecture3_pottedCPP.nb, myoutput.txt, normaloutput.txt, realtypes.cpp, switchdeciding.cpp, whileiteration.cpp, whileiterationcruder.cpp}

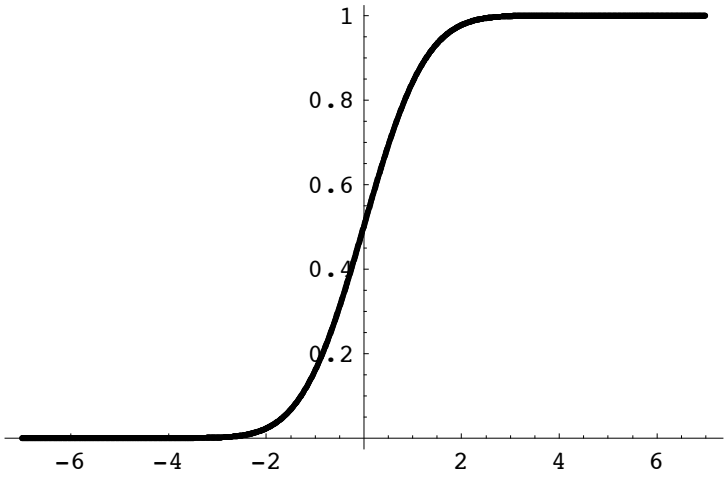
```
cnormaldata = ReadList["normaloutput.txt", Number];
tabular = Partition[cnormaldata, 3];
len = Length[tabular]
```

2001

```
ratdata = Table[{tabular[[i, 1]], tabular[[i, 2]]}, {i, 1, len}];
ListPlot[ratdata];
```




```
marsdata = Table[{tabular[[i, 1]], tabular[[i, 3]]}, {i, 1, len}];
ListPlot[marsdata];
```



These look OK - how are we going to check them out for quality and correctness of the implementation. I will reload some *Mathematica* implementations for comparison.

```
Ncdf[(z_)?NumberQ] := N[0.5*Erf[z/Sqrt[2]] + 0.5];
Ncdf[x_] := (1 + Erf[x/Sqrt[2]])/2
```

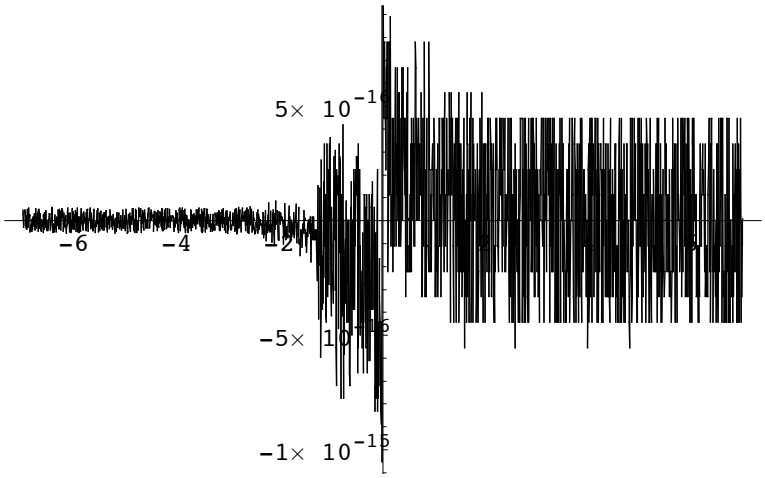
```
HFuncTwoBase[x_] :=
    Module[{γ = 0.2316419, a = 0.319381530, b = -0.356563782,
   c = 1.781477937, d = -1.821255978, e = 1.330274429, k},
        k = 1 / (1 + γ x);
        u = Exp[-x^2 / 2] / Sqrt[2 Pi] *
    (a * k + b * k^2 + c * k^3 + d k^4 + e k^5)]
```

```
HFuncTwo[x_] :=
    If[x >= 0, 1 - HFuncTwoBase[x], HFuncTwoBase[-x]]
```

```
MarsagliaNcdf[x_] :=
 Module[{s = x, t = 0, b = x, q = x * x, i = 1},
  While[Abs[(s - t)] > 10^(-19), (t = s; b *= q / (i += 2); s = t + b)];
  1 / 2 + s / Sqrt[2 Pi] Exp[-q / 2]]
```

■ **Joshi vs *Mathematica* on the rational approximation**

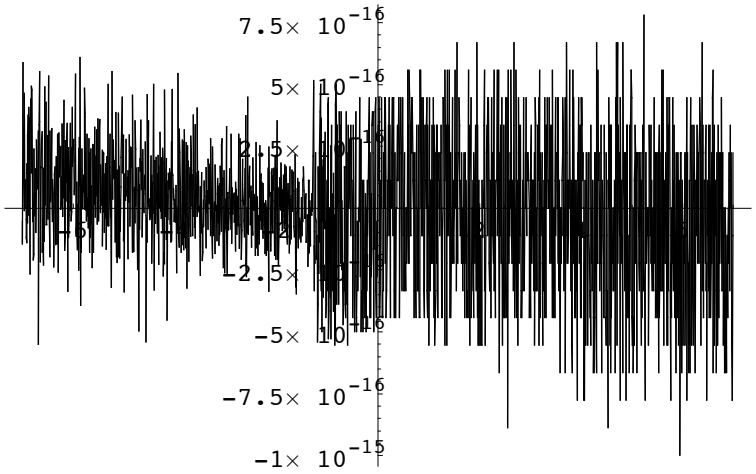
```
ratcomp =
  Table[{tabular[[i, 1]],
    HFuncTwo[tabular[[i, 1]]] - tabular[[i, 2]]}, {i, 1, len}];
ListPlot[ratcomp, PlotJoined → True];
```



So these are indeed the SAME rational approximation!

■ **Marsaglia vs *Mathematica* Marsaglia on the series approximation**

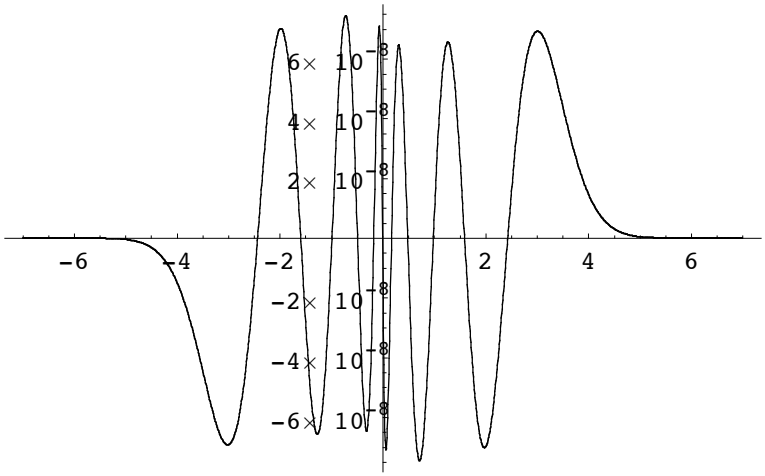
```
marscomp =
  Table[{tabular[[i, 1]], MarsagliaNcdf[tabular[[i, 1]]] -
     tabular[[i, 3]]}, {i, 1, len}];
ListPlot[marscomp, PlotJoined → True];
```



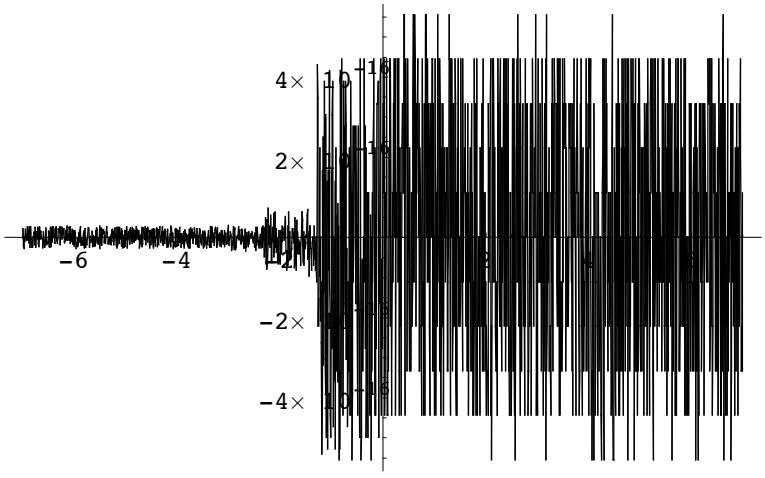
So these are indeed there to as much precision as we care about. Two more comparison of *Mathematica*'s built in gadget against the external C. First we do built in *Mathematica* Phi against the C code with the rational approximation:

```
ratcompb =
  Table[{tabular[[i, 1]], Ncdf[tabular[[i, 1]]] - tabular[[i, 2]]},
   {i, 1, len}];
ListPlot[ratcompb, PlotJoined → True];
```

General::spell1 : Possible spelling error: new symbol
name "ratcompb" is similar to existing symbol "ratcomp". More…




```
marscompb =
  Table[{tabular[[i, 1]], Ncdf[tabular[[i, 1]]] - tabular[[i, 3]]},
   {i, 1, len}];
ListPlot[marscompb, PlotJoined → True];
```



This is about as far as I can be bothered to go! The Marsaglia function will do just fine.

Black-Scholes Modelling in C++

Go run the following - coding the model is easy (error-checking on inputs aside) once you have a decent normal CDF.

> **K = 100, vol = 0.2, r = 0.1, q = 0.05, T = 2**

```
// blackscholes.cpp
// code to work out the standard model using
// Marsaglia's normal function

#include <cmath>
#include <iostream>
#include <fstream>
using namespace std;
const double phinorm = 0.398942280401433;

// pdf
double phi(double x)
{
    return phinorm*exp(-x*x/2);
}

/* The following is the compact code provided by Marsaglia in his 2004 paper!
*/

double PhiMarsaglia(double  x)
{long double s=x,t=0,b=x,q=x*x,i=1;
while(s!=t) s=(t=s)+(b*=q/(i+=2));
return 0.5+s*exp(-0.5*q-0.91893853320467274178L);}

double BSCall(double  S, double K, double sigma, double r, double q, double t)
{
    double sd = sigma*sqrt(t);
    double d1 =(log(S/K)+(r-q)*t+0.5*sd*sd)/sd;
    double d2 = d1-sd;
    return S*exp(-q*t)*PhiMarsaglia(d1)-K*exp(-r*t)*PhiMarsaglia(d2);
}

double BSPut(double  S, double K, double sigma, double r, double q, double t)
{
    double sd = sigma*sqrt(t);
    double d1 =(log(S/K)+(r-q)*t+0.5*sd*sd)/sd;
```

```
    double d2 = d1-sd;
    return K*exp(-r*t)*PhiMarsaglia(-d2)-S*exp(-q*t)*PhiMarsaglia(-d1);
}

using namespace std;
int main()
{
    double result;
    char name[5];
    int N;
    int j;
    double range;
    double s;
    double k;
    double vol;
    double r;
    double q;
    double t;
    double bsc;
    double bsp;

    cout << "Test computations of Black-Scholes pricing \n ";
    cout << "Marsaglia's method for Phi \n";
    cout << "Enter number of positive and negative sample points (N) \n ";
    cin >> N;
    cout << "Enter range parameter (range) for prices above and below strike \n ";
    cin >> range;
    cout << "Enter strike \n ";
    cin >> k;
    cout << "Enter volatility \n ";
    cin >> vol;
    cout << "Enter risk-free rate (continuously compounded) \n ";
    cin >> r;
    cout << "Enter continuous yield (continuously compounded) \n ";
    cin >> q;
    cout << "Enter time to maturity in years \n ";
    cin >> t;

    cout << " Result being written to bsoutput.txt \n " << endl;

    ofstream out("bsoutput.txt");
    for (j=0; j<= 2*N; j++)
    {
        s = k -range + j*range/N;
        bsc = BSCall(s,k,vol,r,q,t);
```

```
        bsp = BSPut(s,k,vol,r,q,t);
        out.precision(15);
        out << s << " " << bsc << " " << bsp << "\n";
        }

    return(0);
}
```

```
SetDirectory[
 "C:\Documents and Settings\William Shaw\My Documents\LGS0708"]
```

C:\Documents and Settings\William Shaw\My Documents\LGS0708

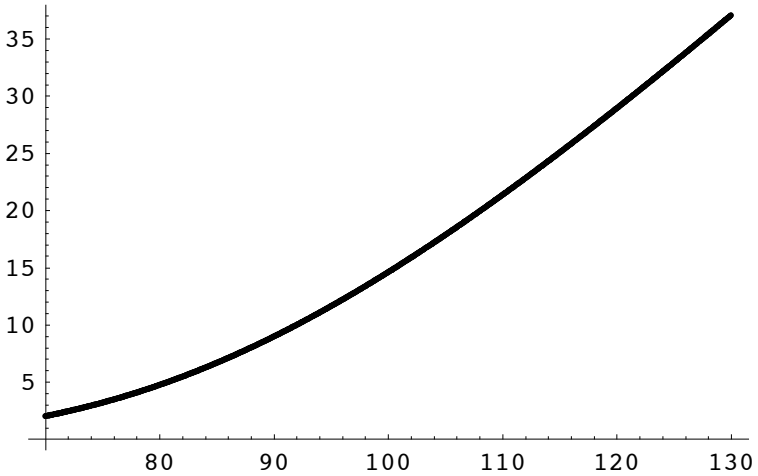
```
FileNames[]
```

{arithmeticone.cpp, arithmeticone.exe, arithmeticonefile.cpp, arithmeticonefile.exe, blackscholes.cpp, cppexamples.zip, cumulativenormals.cpp, cumulativenormals.exe, dowhileiteration.cpp, foriteration.cpp, hello.cpp, hello.exe, helloname.cpp, helloname.exe, ifdeciding.cpp, incrementing.cpp, Lecture3_pottedCPP.nb, myoutput.txt, normaloutput.txt, realtypes.cpp, switchdeciding.cpp, whileiteration.cpp, whileiterationcruder.cpp}

```
bsdata = ReadList["bsoutput.txt", Number];
tabular = Partition[bsdata, 3];
len = Length[tabular]
```

2001

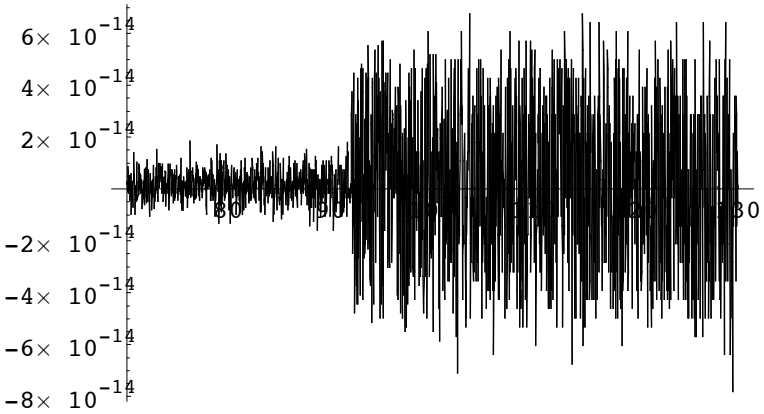
```
bscdata = Table[{tabular[[i, 1]], tabular[[i, 2]]}, {i, 1, len}];
ListPlot[bscdata];
```



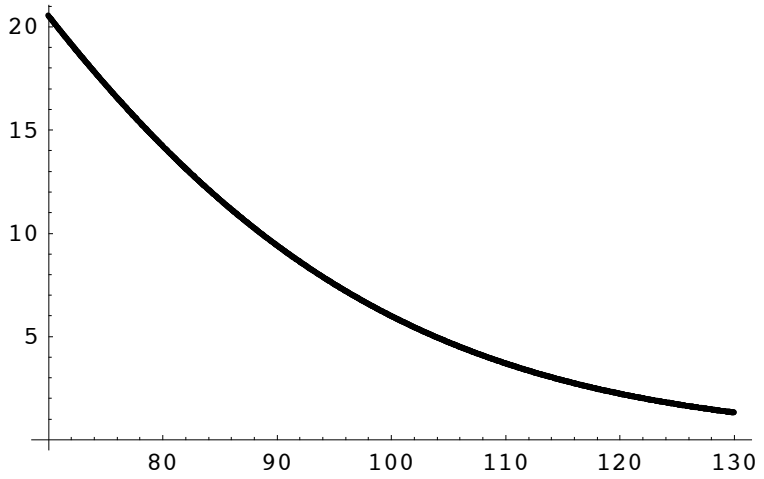
```
done[s_, σ_, k_, t_, r_, q_] :=
((r - q)*t + Log[s/k])/(σ*Sqrt[t]) + (σ*Sqrt[t])/2;
dtwo[s_, σ_, k_, t_, r_, q_] :=
((r - q)*t + Log[s/k])/(σ*Sqrt[t]) - (σ*Sqrt[t])/2;
```

```
BlackScholesCall[s_, k_, σ_, r_, q_, t_] :=
s*Exp[-q*t]*Ncdf[done[s, σ, k, t, r, q]] - k*Exp[-r*t]*Ncdf[dtwo[s, σ,
k, t, r, q]];
BlackScholesPut[s_, k_, σ_, r_, q_, t_] :=
k*Exp[-r*t]*Ncdf[-dtwo[s, σ, k, t, r, q]] - s*Exp[-q*t]*Ncdf[-done[s, σ,
k, t, r, q]]
```

```
callerrorcheck =
  Table[{tabular[[i, 1]],
    BlackScholesCall[tabular[[i, 1]], 100, 0.2, 0.1, 0.05, 2] -
     tabular[[i, 2]]}, {i, 1, len}];
ListPlot[callerrorcheck, PlotJoined → True];
```




```
bspdata = Table[{tabular[[i, 1]], tabular[[i, 3]]}, {i, 1, len}];
ListPlot[bspdata];
```



```
puterrorcheck =
  Table[{tabular[[i, 1]],
    BlackScholesPut[tabular[[i, 1]], 100, 0.2, 0.1, 0.05, 2] -
     tabular[[i, 3]]}, {i, 1, len}];
ListPlot[puterrorcheck, PlotJoined → True, PlotRange → All];
```