3.1 Random Numbers

To use the new random number generator we need to include the \texttt{random} library, the \texttt{cmath} library for any calculations and also \texttt{iostream} to show results onscreen. We first create a new project with an empty program with the correct libraries, and then declare a variable of type \texttt{mt19937}. This declares a new random number generator, which generates pseudo random sequence of integers defined by the Mersenne Twister algorithm. A computer can only generate a random sequence of integers, but of course we can then take that sequence of integers and convert it to any required distribution. Some of the conversions are simple but others are more complex, luckily we now have inbuilt C++ conversion to all standard distributions (more on this later). This means you will always have to create a generator to pass as an argument to the probability distribution you want to generate.

Once you have declared the generator, the next number in the sequence is called with the \texttt{operator()} function, or as can be seen in the program below.

```cpp
#include <iostream>
#include <cmath>
#include <random>
using namespace std;

int main()
{
    // declare a random number generator
    // here we choose the merton twister engine (32bit)
    mt19937 rng;
    cout << rng() << endl;
    // output is 3499211612
}
```
We have called this function once here to return the value 3499211612. You will notice that the number is the same every time the program is run. This is because a random number generator always follows a predefined sequence, and the default start point in the sequence is always the same. We can call the next few numbers by writing a loop or by simple copy/pasting the `cout` line a few times to get

```cpp
cout << rng() << endl;
cout << rng() << endl;
cout << rng() << endl;
// output is
// 3499211612
// 581869302
// 3890346734
```

This repetition in the sequence is in fact extremely useful when bug checking your code as you can eliminate the randomness as the reason your results change. When doing numerical computations it is hardly ever required to start the sequence of with a random start point, since you are only ever interested in what happens on average over a large number of simulations. If you wish to change the starting point of sequence to get a different result or to rerun calculations on the same set of random numbers, then we need to set a seed. You set the seed with the function `seed(int)`, or as follows

```cpp
rng.seed(123);
cout << rng() << endl;
```

Note that the 123 here does not refer the 123rd number in the sequence, but rather this refers to the number 123 in the sequence, and the next result will be generated with that number as input. This means that inputting 124 as the seed will put you in a completely unrelated position in the sequence. Seeds will normally only need to be set once (and only once) in your programs, unless you want to compare results in some way.

There maybe times where you want to set a really random sequence (to generate demonstrations, different results etc) then the best way is to use the type `random_device`. Declare a variable of that type and you can set any number of random seeds with the `operator()` function, the next code demonstrates this

```cpp
random_device randomSeed; // this uses hardware parameters (clocks etc) to generate a random seed
rng.seed(randomSeed()); // set the seed from the random device
cout << rng() << endl;
// my output this time is 1776944864
// but it’s different for each run!!!
```

From here on in we will be using the default seed which should be consistent across platforms.
For the uniform distribution we use the syntax as shown in this program

```cpp
#include <iostream>
#include <cmath>
#include <random>
using namespace std;

int main() {
    // declare a random number generator
    // here we choose the merton twister engine (32 bit)
    mt19937 rng;
    // a uniform distribution
    uniform_real_distribution<double> U(0.,1.);
    // get a number from the uniform distribution
    cout << U(rng) << endl;
    return 0;
}
```

To calculate the probability using simulations, note that if \( u \) is a random draw from the uniform distribution on \((0,1)\) then probability and expected value (average) are linked as follows

\[
\text{Prob}(0.25 < u < 0.5) = E[1_{0.25<u<0.5}]
\]

So to calculate this with simulations we need to count how many times \( u \) lands in the interval, and then divide by the total simulations. We simply need then to store \( N \) the total number of simulations as an integer, and store the running sum of the payoff which in this case is 1 if \( 0.25 < u < 0.5 \) and 0 otherwise. Your code might look a bit like this

```cpp
#include <iostream>
#include <cmath>
#include <random>
using namespace std;

int main() {
    // declare a random number generator
    // here we choose the merton twister engine (32 bit)
    mt19937 rng;
    // a uniform distribution
    uniform_real_distribution<double> U(0.,1.);
    // total number of simulations
    int N=1000;
    // keep track of payoff
    double sum=0;
    for(int i=0;i<N;i++)
    {
        double u=U(rng); // generate the random number
        if(0.25 < u && u < 0.5) // split the interval condition
```

3
Once we are satisfied with the uniform random generator we can move onto the normal random generator. Again this is simply a matter now of declaring the type `normal_distribution<double>` to enable us to convert random integers into draws from a normal distribution. For the code we get

```cpp
#include <iostream>
#include <cmath>
#include <random>

using namespace std;

int main()
{
  // declare a random number generator
  // here we choose the merton twister engine (32 bit)
  mt19937 rng;
  // a normal distribution with mean 0 and variance 1
  normal_distribution<double> Phi(0.,1.);
  // get a number from the normal distribution
  cout << Phi(rng) << endl;
  return 0;
}
```

In the next part we are asked to build up a histogram plot of the normal distribution over the range of intervals

\[
a = -4.2 \leq (i - 1/2)h < x < (i + 1/2)h \leq b = 4.2
\]

with \(i = -10, -9, \ldots, 9, 10\) and \(h = 0.4\). Here we have 21 intervals, so we are going to have to store the central position of each of the intervals as well as keep count of the frequency at which a random draw lands in the interval. For simplicity we shall use arrays for storage, with a double for the interval position and integer for counting. The variable declarations should be, noting here that \(n\) must be constant to use with arrays

```cpp
// input parameters
int totalRuns=1e6; // total number of simulations
const int n=21; // number of intervals
double a=-4.2,b=4.2;// start/end points
double h=(b-a)/n; // fixed interval width
```
double x[n]; // center of intervals

// local storage
int counter[n]; // number of times landing in interval

and you should then initialise the values. Because array indices work starting from 0 it will be better if we index the interval by j=0,1,...,20. Then the centre of the interval is given by the formula

\[ x_j = a + \frac{h}{2} + jh \] for \( j = 0,1,\ldots,20 \) (1)

It is also good practice to reset the value of the counter to zero.

Next we want to generate a random number and then find which interval it sits in, and to do this we need the floor function from the math library. We can rearrange (1) to find that a number \( \phi \) sits in the \( j^* \)th interval given by

\[ j^* = \left\lfloor \frac{\phi - a}{h} \right\rfloor \]

We can test this with some code

```c
double phi=Phi(rng); // get a number from the normal distribution
int jStar = floor((phi-a)/h); // use the floor function to get j* for the nearest point to x_j* to phi
cout << phi << " " << jStar << " " << a+jStar*h << " " << a+(jStar+1)*h << endl;
// output is
// 0.13453 10 -0.2 0.2
```

The interval selected is the 10th interval which corresponds to \([-0.2,0.2]\), and clearly \( \phi=0.13453 \) is in this interval. When you are coding this up you will need to check that you are within the bounds.

The final solution can be plotted (with gnuplot for example as I have used)
and the code to generate this plot is

```cpp
#include <iostream>
#include <cmath>
#include <random>

using namespace std;

int main()
{
    // declare a random number generator
    // here we choose the merton twister engine (32 bit)
    mt19937 rng;
    // a uniform distribution
    normal_distribution<double> Phi(0,1);
    // get a number from the normal distribution

    // create a histogram and output to screen
    // number of intervals
    const int n=21;
    int totalRuns=1e6;
    // start/end points
    double a=-4.2,b=4.2;
    // fixed interval width
    double h=(b-a)/n;
    // center of intervals
```
double x[n];
// number of times landing in interval
int counter[n];

for(int j=0; j<n; j++)
{
    x[j] = a+h/2.+ j*h; // setup the position of the central points
    counter[j]=0; // reset the counter
}

for(int run=0; run<totalRuns; run++) // run over all simulations
{
    double phi=Phi(rng); // get a number from the normal distribution
    int jStar = floor( (phi-a)/h ); // use the floor function to get j* for
    // the nearest point to x_j* to phi
    if(jStar>n || jStar<0) continue; // if you are outside the grid continue
    counter[jStar]++; // add one to the correct counter
}

for(int j=0; j<n; j++) // output results
{
    // output i, midpoint, frequency ( phi \in [ x_j-0.5*h , x_j+0.5*h ] )
    cout << j << " " << x[j] << " " << counter[j] << endl;
}
// output looks like
// 0  -4.66
// 1   -3.6 291
// 2   -3.2 1022
// 3  -2.8 3340
// 4  -2.4 9043
// 5   -2 22022
// 6   -1.6 44645
// 7   -1.2 77928
// 8  -0.8 115622
// 9  -0.4 146577
// 10  0 158419
// 11  0.4 147059
// 12  0.8 115190
// 13  1.2 77761
// 14  1.6 44979
// 15  2 22082
// 16  2.4 9372
// 17  2.8 3252
// 18  3.2 1015
// 19  3.6 245
// 20  4.50
return 0;
}
3.2 Monte Carlo

Now we are going to value an European call option using Monte-Carlo. The setup is very simple, we just need to sum up the payoffs from a bunch of sample paths and then take the average. First start with an empty program except for the random number generator, as follows

```cpp
#include <iostream>
#include <random>
#include <cmath>
using namespace std;

int main()
{
    // declare the random number generator
    mt19937 rng;
}
```

where we are including the `random` library, as well as `cmath` for access to mathematical functions required later. Next declare the variable which contains the methods to get random draws from a normal distribution

```cpp
// declare the distribution
normal_distribution<> ND(0.,1.);
```

Now we have everything we need to start, declare your parameters for the option at the top of your main function alongside the number of simulations we are going to run (a counting number)

```cpp
// parameters
double stock0=9.576, strikePrice=10., interestRate=0.05, sigma=0.4, maturity=0.75;
// number of paths
int N=1000;
```

We put these at the top to indicate that they will be input variables when we move this out to its own function later on. Now we can write the Monte-Carlo algorithm in less than 10 lines, start by initialising a sum variable with zero and then run through each path, adding in the payoff of a call for each particular stock price at maturity

```cpp
// initialise sum to zero
double sum=0.;
// run through each simulation
for(int i=0;i<N;i++)
{
    // // calculate stock price
    double stock=stock0*exp((interestRate-0.5*sigma*sigma)*maturity+
                        sigma*sqrt(maturity)*ND(rng));

    // calculate call payoff
    double payoff=max(stock-strikePrice,0.);

    // add in the payoff
    sum+=payoff;
}
```

```cpp`
```
double phi=ND(rng); // get a random draw from N(0,1)
double ST=S0 * exp( (interestRate - 0.5*sigma*sigma)*maturity + phi*sigma
 * sqrt(maturity) ); // get a random draw for the value of stock price
 at maturity
sum = sum + max( ST - strikePrice , 0. ); // add in the payoff of the
option in that case
}
cout << sum/N*exp(-interestRate*maturity) << endl; // output the average
value over all paths

Running this code should return a value of 1.2671, which we can check is reasonably close
to the analytic value we would expect.

In order to do any sort of meaningful analysis, we must now put this algorithm into a
function. Use the following definition of the function

double monteCarlo(double S0 , double strikePrice , double interestRate , double
sigma , double maturity , int N)
{
// declare the random number generator
    mt19937 rng;
// declare the distribution
    normal_distribution<> ND(0.,1.);
    ND(rng);
// initialise sum
    double sum=0.;
    for(int i=0; i<N; i++)
    {
        double phi=ND(rng);
        double ST=S0 * exp( (interestRate - 0.5*sigma*sigma)*maturity + phi*sigma
 * sqrt(maturity) );
        sum = sum + max( ST - strikePrice , 0. );
    }
    return sum/N*exp(-interestRate*maturity);
}

and copy/paste everything from main into this function (except the parameter declarations). Our full program at this stage should now look like

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

double monteCarlo(double S0 , double strikePrice , double interestRate , double
sigma , double maturity , int N)
{
    // declare the random number generator
    mt19937 rng;
    // declare the distribution
    normal_distribution<> ND(0.,1.);
    ND(rng);
    // initialise sum
    double sum=0.;
    for(int i=0; i<N; i++)
    {
        double phi=ND(rng);
        double ST=S0 * exp( (interestRate - 0.5*sigma*sigma)*maturity + phi*sigma
 * sqrt(maturity) );
        sum = sum + max( ST - strikePrice , 0. );
    }
    return sum/N*exp(-interestRate*maturity);
}
```cpp
int main()
{
    cout << monteCarlo(9.576,10.,0.05,0.4,0.75,1000) << endl;
}
```

Run it and check you get the same result as last time. Now to do some analysis, we are going to want to run this algorithm lots of times with a different number of paths. So try running the function several times in `main` (copy/paste the single line of code)

```cpp
cout << monteCarlo(9.576,10.,0.05,0.4,0.75,1000) << endl;
cout << monteCarlo(9.576,10.,0.05,0.4,0.75,1000) << endl;
cout << monteCarlo(9.576,10.,0.05,0.4,0.75,1000) << endl;
cout << monteCarlo(9.576,10.,0.05,0.4,0.75,1000) << endl;
// output is
// 1.2671
// 1.2671
// 1.2671
// 1.2671
```

The output is the same each time, but why? We are supposed to be generating random numbers. Unfortunately because of the way we declare the random number generator, each time we call this function a new version of the random number generator is created which gets reset back to the first number in the sequence. To avoid this we need to make sure that one and only one random number generator is ever created for this function, so we use the keyword `static` to tell the compiler this. The appropriate line in your code is changed to

```cpp
static mt19937 rng;
```

and when we run the program again we get

1.2671
1.33243
1.22289
1.31114

Ok now we are confident that we can generate random samples of the solution for a given \( N \), we want to see what happens expected distribution of the results as we increase the number of paths. We are going to do some analysis now which requires the stl storage container `vector` which allows for dynamic allocation of memory. They can perform all of the same things a simple array can do and much more. In the `main` function, we declare `vector<double> samples(M)` which is an array of type double intialised with \( M \) values. First check you can run the results by outputting to screen

```cpp`
```
```
// now store all the results
vector<double> samples(M);
// number of paths in each calculation
int N=1000;
// run some calculations
for(int i=0;i<M;i++)
{
    cout << monteCarlo(9.576,10.,0.05,0.4,0.75,N) << endl;
}

Now rather than output to screen write them into the vector samples using the array syntax like this

// run some calculations
for(int i=0;i<M;i++)
{
    // cout << monteCarlo(9.576,10.,0.05,0.4,0.75,N) << endl;
    samples[i] = monteCarlo(9.576,10.,0.05,0.4,0.75,N);
}

Calculate the mean of the vector

double sum=0.;
for(int i=0;i<M;i++)
{
    sum+=samples[i];
}
double mean = sum/M;
cout << " sample mean = " << mean << endl;

and variance

double sumvar=0.;
for(int i=0;i<M;i++)
{
    sumvar+=(samples[i]-mean)*(samples[i]-mean);
}
double variance = sumvar/(M-1);
cout << " sample variance = " << variance << endl;

and the output is

sample mean = 1.28323
sample variance = 0.00609616

From simple statistics we know that each estimate of the solution $V_N$ is

$$V_N \sim N(V^*, \sigma^2)$$
where $V^*$ is the true solution and $\sigma^2$ is the variance. The if we choose to take a sample mean from that distribution $\bar{V}$ with $M$ samples we have

$$\bar{V} \sim N \left( V^*, \frac{\sigma^2}{M} \right)$$

So to get a confidence interval for our result we use this result to output the following

```cpp
double sd = sqrt(variance/M);
cout << "95% confident result is in ["<<mean-2.*sd << "," << mean+2.*sd << "] with " << N*M << " total paths. " << endl;
```

and the output is

95% confident result is in [1.26761,1.29884] with 100000 total paths.

We can now run this for a range of values for $N$ and $M$. Note first that the sample mean estimate uses $NM$ total paths to get the confidence interval. We could set $N = 1$ and $M = 100000$ and you should get a similar result to the one above. This is because even with a single path the European put/call option will be approximately normally distributed. For other more complex options this might not be the case and you will need to ensure both $N >> 1$ and $M >> 1$. Finally we present some code to run different values of $M$ for fixed $N$ on a European call option. Note here that I have included a payoff function, so that if you need to solve for a different option you should change the payoff here and it should work fine.

```cpp
#include <iostream>
#include <random>
#include <cmath>
using namespace std;

// payoff from the European call option
double payoff(double S, double strikePrice)
{
    return max(S - strikePrice, 0.); // change this line here to solve for different European options
}

double monteCarlo(double S0, double strikePrice, double interestRate, double sigma, double maturity, int N)
{
    // declare the random number generator
    static mt19937 rng;
    // declare the distribution
    normal_distribution<> ND(0., 1.);
    ND(rng);
    // initialise sum
    double sum=0.;
    for(int i=0;i<N;i++)
```
double phi = ND(rng);
// calculate stock price at T
double ST = S0 * exp((interestRate - 0.5 * sigma * sigma) * maturity + phi * sigma * sqrt(maturity));
// add in payoff
sum = sum + payoff(ST, strikePrice);
}
// return discounted value
return sum / N * exp(-interestRate * maturity);

int main()
{
    // run for different
    for (int M = 100; M <= 100000; M *= 10)
    {
        // now store all the results
        vector<double> samples(M);
        // number of paths in each calculation
        int N = 1000;
        cout << "------------" << endl;
        cout << "Run results with M=" << M << " samples from V_N, where N=" << N << "." << endl;
        // run some calculations
        for (int i = 0; i < M; i++)
        {
            samples[i] = monteCarlo(9.576, 10.0, 0.05, 0.4, 0.75, N);
        }
        // estimate the mean from the sample
        double sum = 0.;
        for (int i = 0; i < M; i++)
        {
            sum += samples[i];
        }
        double mean = sum / M;
        cout << " mean = " << mean << endl;
        // estimate the variance from the sample
        double sumvar = 0.;
        for (int i = 0; i < M; i++)
        {
            sumvar += (samples[i] - mean) * (samples[i] - mean);
        }
        double variance = sumvar / (M - 1);
        cout << " variance = " << variance << endl;
        // get the standard deviation of the sample mean
        double sd = sqrt(variance / M);
    }
cout << " 95% confident result is in ["<<mean-2.*sd << "," << mean+2.*sd << "] with "<< N*M << " total paths." << endl;
}

References