MATH60082
Examples Sheet 3

Contents

3.1 Generating Random Numbers 1
3.2 Monte Carlo on Uniform Random Numbers 3
3.3 Are you Normal? 4
3.4 European Options 8
3.5 Analysing Monte-Carlo Results 10

3.1 Generating Random Numbers

To use the new random number generator we need to include the `random` library, the `cmath` library for any calculations and also `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 `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 `operator()` function, or as can be seen in the program below.

```
1 int main()
2 {
3   // declare a random number generator
4   // here we choose the merton twister engine (32bit)
5   mt19937 rng;
6   cout << rng() << endl;
7   // output is 3499211612
8 }
```

download

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

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

download

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

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

download

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

```
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!!!
```

download

From here on in we will be using the default seed which should be consistent across platforms.

> TASKS:
>
> (i) Use a loop to output the first 1000 numbers in the sequence.
>
> (ii) Try resetting the seed using the number 123 inside the loop – what happens?
>
> (iii) Try resetting the seed using a `random_device` inside the loop – what happens?

3.2 Monte Carlo on Uniform Random Numbers

> TASKS:
>
> (i) Write a program to generate random number from the uniform distribution between 0 and 1.
>
> (ii) Can you calculate the probability that a random draw u is between 0.25 and 0.5?

For the uniform distribution we use the syntax as shown in this program

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

download

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

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

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

```
1  // total number of simulations
2  int N=1000;
3  // keep track of payoff
4  double sum=0;
5  for(int i=0;i<N;i++)
6  {
7    double u=U(rng); // generate the random number
8    if(0.25 < u && u < 0.5) // split the interval condition
9    {
10     sum += 1;// add in payoff
11   }
12 }
13
14 cout << " P(0.25 < u < 0.5) = " << sum/N << endl;
15 return 0;
```

download

TASKS:

(i) Run the code above and check the result makes sense.

(ii) Put the algorithm into a function like this:

```
double calcProb(double a,double b,int N)
{
// calculate probability a<u<b
}
```

(iii) Return the calculated probability for different values of N, what can you say about the results as $N \to \infty$?

3.3 Are you Normal?

TASKS:

(i) Write a program to generate random numbers from the normal distribution with mean 0 and variance 1.

(ii) Can you test that the distribution is normal?

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

```
// 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;
```

download

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 $i = -10, -9, \dots, 9, 10$ 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

```
// input parameters
int totalRuns=1e6; // total number of simulations
// interval specification
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
```

download

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

$$x_j = a + \frac{h}{2} + jh \text{ for } j = 0, 1, \dots, 20 \tag{1}$$

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

```
1   double phi=Phi(rng); // get a number from the normal distribution
2   int jStar = floor( (phi-a)/h ); // use the floor function to get j* for the
        nearest point to x_j* to phi
3   cout << phi << " " << jStar << " " << a+jStar*h << " " << a+(jStar+1)*h <<
        endl;
4 // output is
5 //    0.13453 10 -0.2 0.2
```

download

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) in figure 1 and the few lines from my code to generate this plot are given below. Click on download to get the full code.

```
1   for(int run=0;run<totalRuns;run++) // run over all simulations
2   {
3     double phi=Phi(rng); // get a number from the normal distribution
4     int jStar = floor( (phi-a)/h ); // use the floor function to get j* for
         the nearest point to x_j* to phi
5     if(jStar>=n || jStar<0)continue; // if you are outside the grid continue
6     counter[jStar]++; // add one to the correct counter
7   }
```

download

TASKS:

(i) Create a loop in your program to generate n draws from $\phi \sim N(0,1)$.

(ii) Use the interval selection code to find which *bin* each random number lands in (`jStar`)

(iii) increment (or add one) to the appropriate *bin* (`counter[jStar]`)

(iv) Plot out your results of `x` vs `counter` – does it look normal?

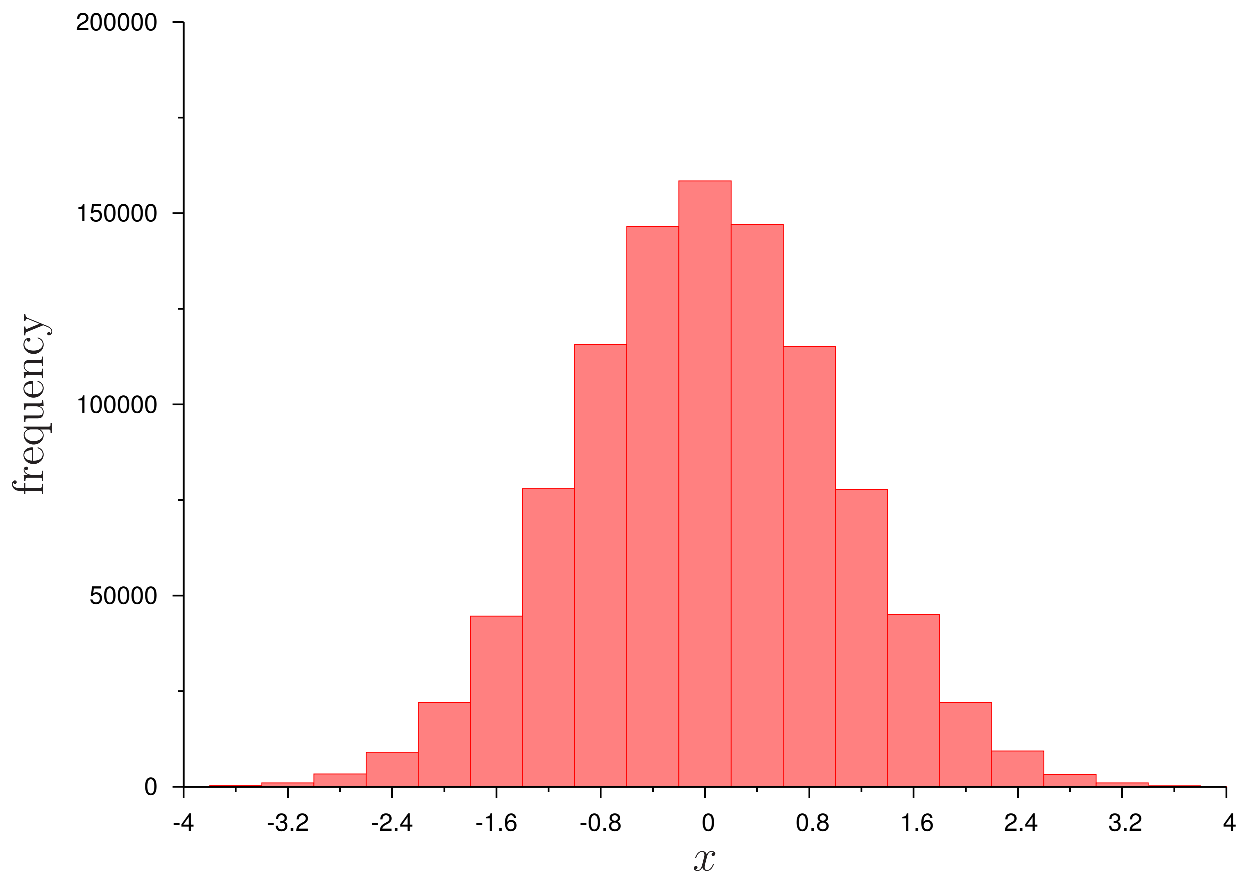

Figure 1: A histogram of calls to the standard normal distribution function in c++

3.4 European Options

TASKS:
Write a program to calculate the value of a Black Scholes call option C with strike price $X = 10$ and a maturity $T = 0.75$ that is 9 months from now, given the current stock price $S_0 = 9.576$. Assume we can price under the risk neutral measure, and that S follows a GBM with $\sigma = 0.4$, the interest rate is $r = 0.05$.

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, click on download to get the full code.

```
1  // declare the random number generator
2  mt19937 rng;
```

download

Note that 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

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

download

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)

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

download

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

```
1  // initialise sum to zero
2  double sum=0.;
3  // run through each simulation
4  for(int i=0;i<N;i++)
```

```
5    {
6      double phi=ND(rng); // get a random draw from N(0,1)
7      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
8      sum = sum + max( ST - strikePrice , 0. ); // add in the payoff of the
         option in that case
9    }
10   cout << sum/N*exp(-interestRate*maturity) << endl; // output the average
       value over all paths
```

download

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

```
1 double monteCarlo(double S0,double strikePrice,double interestRate,double
     sigma,double maturity,int N)
2 {
3
4 }
```

download

and copy/paste everything from main into this function (except the parameter declarations). The first few lines from the Monte Carlo function should now look like this. Click on download to get the full code.

```
1 double monteCarlo(double S0,double strikePrice,double interestRate,double
     sigma,double maturity,int N)
2 {
3   // declare the random number generator
4   mt19937 rng;
5   // declare the distribution
6   normal_distribution<> ND(0.,1.);
7  ND(rng);
8   // initialise sum
9   double sum=0.;
```

download

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)

```
1   cout << monteCarlo(9.576,10.,0.05,0.4,0.75,1000) << endl;
2   cout << monteCarlo(9.576,10.,0.05,0.4,0.75,1000) << endl;
3   cout << monteCarlo(9.576,10.,0.05,0.4,0.75,1000) << endl;
4   cout << monteCarlo(9.576,10.,0.05,0.4,0.75,1000) << endl;
5   // output is
6   // 1.2671
7   // 1.2671
```

```
8  // 1.2671
9  // 1.2671
```

download

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

```
1  static mt19937 rng;
```

download

and when we run the program again we get

```
1.2671
1.33243
1.22289
1.31114
```

TASKS:
Run this code for $N = 100, 200, 300, \dots$ and plot out the results for different N. What does it look like?

3.5 Analysing Monte-Carlo Results

TASKS:
How accurate is your Monte Carlo estimate of the value?

Okay, so 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. Click on download to get the full code.

```
1  // we want to run M calculations
2  int M=100;
```

```
3    // now store all the results
4    vector<double> samples(M);
5    // number of paths in each calculation
6    int N=1000;
7    // run some calculations
8    for(int i=0;i<M;i++)
9    {
10     cout << monteCarlo(9.576,10.,0.05,0.4,0.75,N) << endl;
11   }
```

download

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

```
1  // run some calculations
2    for(int i=0;i<M;i++)
3    {
4      // cout << monteCarlo(9.576,10.,0.05,0.4,0.75,N) << endl;
5      samples[i] =  monteCarlo(9.576,10.,0.05,0.4,0.75,N);
6    }
```

download

Calculate the mean of the vector

```
1    double sum=0.;
2    for(int i=0;i<M;i++)
3    {
4      sum+=samples[i];
5    }
6    double mean = sum/M;
7    cout << " sample mean = " << mean << endl;
```

download

and variance

```
1    double sumvar=0.;
2    for(int i=0;i<M;i++)
3    {
4      sumvar+=(samples[i]-mean)*(samples[i]-mean);
5    }
6    double variance = sumvar/(M-1);
7    cout << " sample variance = " << variance << endl;
```

download

and the output is

```
sample mean = 1.28323
sample variance = 0.00609616
```

From simple statistics (Central Limit Theorem) 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 σ^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

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

download

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 each simulation for V is an independent identically distributed random number, so the Central Limit Theorem applies. However, if a more complex option is solved in which a regression is performed over some fixed number of paths each simulation will no longer be *i.i.d.*. This also could apply to antithetic paths as well (they are not *i.i.d.*). 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. Click on download to get the full code.

```
#include <iostream>
#include <random>
#include <cmath>
#include <vector>
#include <algorithm>
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)
{
```

download

TASKS:

(i) Run this analysis to get confidence intervals for a range of values for N and M

(ii) If $N * M$ stays the same, what can you say about the confidence interval? Why?

(iii) Try different payoff functions, say put options, binary options, different parameters.

(iv) What has the most effect on the results?

References