4.1 Time To First Exit

Calculate the expected hitting time $E[t^*]$ for a Brownian motion $X(t)$, where the process must hit either $X(t^*) = 0$ or $X(t^*) = 1$ for $t^* < T$. If neither boundary is hit within the time $T$ then $t^* = T$.

Assume that the process $X$ follows the SDE

$$dX = \mu dt + \sigma dW$$

where $\mu = 0.01$ is the drift and $\sigma = 0.75$ is the standard deviation of the Brownian motion. We have that the initial start point of the Brownian motion is $X(t = 0) = 0.56$, and we set $T = 1$.

Now we need to consider the problem, which is to track the path of Brownian motion through time, and check at each point in time whether or not the process has crossed either one or zero. So there are obvious stages in which we can build up our code, namely

- Stage 1: Generate random numbers from a normal distribution
- Stage 2: Generate a single path in $X$
- Stage 3: Stop the path when $X < 0$ or $X > 1$ and output the current time
- Stage 5: Calculate expected value
- Stage 6: Move into function returning expected value for $N$ runs
- Stage 7: Test and validate the code

Stage 1 has already been dealt with in a previous example. You can use this sample code to start.
```cpp
#include <iostream>
#include <random>
#include <cmath>
using namespace std;

int main()
{
    // declare the random number generator
    mt19937 rng;
    // declare the distribution
    normal_distribution<> ND(0., 1.);
}
```

Now we need to think about variables and parameters for this problem. The drift, standard deviation, maximum time $T$ and the initial value of $X$ are the input parameters, as well as possibly the position of the bounds. For a given path we also need to specify the discretisation size $dt$. Declare all of these variables and assign them with some trivial values to begin with.

```cpp
// declare parameters
double X0=0.56, mu=0.01, sigma = 0.75, T=1.;
// grid params
int K=100;
// local params
double dt=T/K;
```

Compile and run the program to check everything is ok so far. Next we need to keep track of the current value of $X(t)$ independently of storing the initial value which will become clearer later on. Declare another variable $X_t$ to store $X(t)$, initialise it with an appropriate value. Output to screen with the command `cout << X_t << endl;`, you should get 0.56.

Now we want to calculate the path using a recurrence relation for $X$. We choose to use a `for` loop for this, which must run between $k = 0$ and $k = K - 1$. Input the equation for the next value of $X_t$ overwriting the previous value. It should look something like this.

```cpp
// first generate a path in X(t) and print to screen]
// initialise value of X at time 0
double t=0.;
double X_t=X0;

cout << t " | X(t) " <<endl;
cout << t << " | " << X_t << endl;
for(int k=0;k<K;k++)
{
    X_t = X_t + mu*dt + sqrt(dt)*sigma*ND(rng);
    t = (k+1)*dt;
    cout << t " | X(t) " <<endl;
    cout << t << " | " << X_t << endl;
}
```
cout <<t<< " \t| " << Xt << endl;
}

The final value should \( t = 1 \) and \( X(1) = 0.412219 \). Inspecting the path we can see that it hits a value of \( X(t) = 1.06307 \) at \( t = 0.54 \).

Now we want to implement that inspection of the path within the code. To make the code more flexible for future modifications and make out code look more like a standard Monte Carlo algorithm we are going to first reimplement the path generation to store all the values in an array. We choose to use a stl vector to store the values, again for flexibility. First step is to delete the declaration for \( \texttt{Xt} \) and redefine it

\[
\texttt{vector<double> \ Xt(K+1);}
\]

which is a vector with \( K + 1 \) elements. Now we initialise the first value in the vector to \( \texttt{Xt[0]} = X0; \) and then use appropriate values in the recurrence relation. This part of your code should now look like

```cpp
// now generate the path in a vector \( X(K+1) \) and print to screen
vector<double> Xt(K+1);
// initialise value of \( X \) at time 0
double t = 0.;
\texttt{Xt[0]} = X0;

\texttt{cout} << " t \t| X( t ) " <<\texttt{endl};
\texttt{cout} << "−−−−−−−−−−−−−−−−−−−−−−−jh" <<\texttt{endl};
\texttt{cout} << \texttt{t} << " \t| " \texttt{Xt[0]} <<\texttt{endl};
\texttt{for(int k=0;k<K;k++)}
{
  \texttt{t=k}\ast dt;
  \texttt{cout} << \texttt{t} << " \t| " \texttt{Xt[k]} <<\texttt{endl};
  \texttt{Xt[k+1]} = \texttt{Xt[k]} + \texttt{mu}\ast dt + \texttt{sqrt(dt)}\ast\texttt{sigma}\ast\texttt{ND(rng)};
  \texttt{t} = \texttt{(k+1)}\ast dt;
  \texttt{cout} << \texttt{t} << " \t| " \texttt{Xt[k+1]} <<\texttt{endl};
}
```

Now we want to implement the payoff, defined as the first hitting time. To do this we separate the generation of the path from evaluating the payoff. So once the path has been generated, we move back through the path and calculate the payoff. For each element in the vector \( \texttt{Xt} \), we check if it is above or below the bounds. If either condition is satisfied we break from the loop and return the current time. If we get to the final element just return \( T \). My implementation of this looks like this

```cpp
// now calculate first exit time from the path
double exitTime = 0.;
\texttt{for(int k=0;k<=K;k++)}
{
  \texttt{t} = \texttt{k}\ast \texttt{dt};
  \texttt{cout} << \texttt{t} << " \t| " \texttt{Xt[k]} <<\texttt{endl};
  \texttt{if(k==K)} exitTime = \texttt{t};
  \texttt{if(\texttt{Xt[k]} < 0.)} // set exit time and stop
```

3
Now we have generated a single path we check and run the code. For the default seed (none supplied), we get $t^* = 0.54$ as expected. We can run the code for another path by resetting the seed (place `rng.
seed(1)` directly underneath where it is declared). By changing the value of the seed you should see a range of values are possible. Once we are happy that this is behaving correctly add a Monte Carlo loop to calculate the average exit time over $N$ samples. Underneath where you have declared the normal distribution declare $N$, your `sum` variable and start the Monte Carlo loop:

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

// add in the monte carlo loop
int N = 1000;
double sum = 0.;

for (int i = 0; i < N; i++)
{
    // now generate the path in a vector X(M+1) and print to screen
}
```

Close the loop towards the bottom of your program and output the average exit time:

```cpp
} // finished getting exit time
// cout << " Exit time " << exitTime << endl;
sum = sum + exitTime;

} // finish Monte Carlo loop

cout << " Expected exit time " << sum/N << endl;
```

Again we should test now with different values of $N$ and $K$ to ensure that things are working properly. After I have removed unnecessary screen outputs my code looks something like this:

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

int main()
{
    // declare parameters
    double X0=0.56 , mu=0.01 , sigma =0.75 , T=1.;
    // grid params
    int M=100;
    // local params
    double dt=T/K;
    // declare the random number generator
    mt19937 rng ;
    // declare the distribution
    normal_distribution<> ND(0.,1.);
    // monte carlo loop
    int N=1000;
    double sum=0.;
    for (int i=0;i<N;i++)
    {
        // now generate the path in a vector X(K+1) and print to screen
        vector<double> Xt(K+1);
        // initialise value of X at time 0
        double t=0.;
        Xt[0]=X0;
        for (int k=0;k<K;k++)
        {
            Xt[k+1] = Xt[k] + mu*dt + sqrt(dt)*sigma*ND(rng);
        }
        // now calculate first exit time from the path
        double exitTime=0.;
        for (int k=0;k<=K;k++)
        {
            // get current time this loop
            t=k*dt;
            if (k==K) exitTime = t;
            if (Xt[k]<0.) // set exit time and stop
            {
                exitTime = t;
                break;
            }
            if (Xt[k]>1.) // set exit time and stop
            {
                exitTime = t;
                break;
            }
        }
    }
}
```
Once we are convinced we can put the Monte Carlo part in its own function. Remember that once it goes inside the function we need to make the random number generator static so that the sequence is not reset every time the function is called. We can now analyse the results, taking different value of $N$ and $K$ and trying to get confidence interval just like we did in a previous example. Finally your code might look something like:

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

double monteCarlo(double X0, double mu, double sigma, double T, int K, int N)
{
    // local params
    double dt=T/K;

    // declare the random number generator
    static mt19937 rng;
    // declare the distribution
    normal_distribution<> ND(0.,1.);
    double sum=0.;
    
    for(int i=0;i<N;i++)
    {
        // now generate the path in a vector X(K+1) and print to screen
        vector<double> Xt(K+1);
        // initialise value of X at time 0
        double t=0.;
        Xt[0]=X0;
        for(int k=0;k<K;k++)
        {
            Xt[k+1] = Xt[k] + mu*dt + sqrt(dt)*sigma*ND(rng);
        }

        // now calculate first exit time from the path
        double exitTime=0.;
        for(int k=0;k<=K;k++)
        {
            // get current time this loop
            t=k*dt;
            if(k==K)exitTime = t;
        }

        sum = sum + exitTime;
    }

    // finished getting exit time
    sum = sum + exitTime;
}

// finish Monte Carlo loop
cout << " Expected exit time " << sum/N <<endl;
}```
```c
if (Xt[k] < 0.) // set exit time and stop
{
    exitTime = t;
    break;
}
if (Xt[k] > 1.) // set exit time and stop
{
    exitTime = t;
    break;
} // finished getting exit time

sum = sum + exitTime;

} // finish Monte Carlo loop

return sum/N;
```

```c
int main()
{
    // declare parameters
    double X0=0.56, mu=0.01, sigma=0.75, T=1.;
    // grid params
    int K=1000;
    cout << " Solve for the first exit time with parameters " << endl;
    cout << " X0=" << X0 << " , mu=" << mu << " , sigma=" << sigma << " , T=" << T << " and K=" << K << " . " << endl;
    cout << " ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" << endl;
    // run for different
    for (int M=10; M<=1000; M*=10)
    {
        // now store all the results
        vector<double> samples(M);

        // monte carlo loop
        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(X0, mu, sigma, T, K, 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