Vectors

Make sure that you include, at least, some comments in your code. Remember to use optimizations on your codes. There is information on the web but ask in class if you are unsure. In order to use vectors you need to include vector libraries at the top of your code:

```cpp
#include <vector>
```

4.1 Declare a vector of doubles in you code. The syntax is as follows:

```cpp
vector<double> a(4), b, c(6, 3.);
```

where \(a\) is a vector with 4 elements, \(b\) is an empty vector, and \(c\) is a vector with 6 elements each with the value 3. Set values to your vector and print them out to the screen, using the access operator \([\]) to reference an element of the vector, then print them to the screen

```cpp
a[0] = 1.;
```

4.2 Now declare a 2D vector of doubles (see Tutorial on the web) using the syntax:

```cpp
vector<vector<double>> doubleVector;
```

This generates an empty vector. Resize it using the following syntax or that on the web:

```cpp
// size of first dimension is 3
doubleVector.resize(3);
// setup size for each of the elements in that dimension
// they need not be the same size
for(int i=0;i<doubleVector.size();i++)doubleVector[i].resize(5);
```

Again we use the access operator \([\]) to access elements
4.3 Practice assigning values to the vector, printing out elements to the screen and performing calculations on it (e.g. find the average of all the elements).

**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 \( T = 1 \).

4.4 You should solve this problem using a path dependent Monte Carlo code. The algorithm for a path in the code will look some thing like:
\[
X^{k+1} = X^k + \mu dt + \sigma \phi^k \sqrt{dt} \quad \text{for } k = 0, 1, ..., K - 1
\]
where \( dt = T/K \) is a small time increment and \( X^k = X(t^k) = X(kdt) \). The value of \( dt \) in this case has to be chosen small enough so as not to influence the results. As the path is updated, you will need to:-

- calculate the new position of \( X \)
- add a time increment \( dt \) to total time spent in interval
- if \( X^{k+1} < 0 \) or \( X^{k+1} > 1 \) then exit and return hitting time \( t^{k+1} \)
- if path reaches \( t^K = T \) then return \( T \)

The expected value will just be the average of a large number of sample paths.

4.5 Once the algorithm has been developed inside main, move it out into its own function. Think about the inputs and outputs from the function.

4.6 Check your algorithm for convergence in both the number of time steps and number of paths. How paths are needed to get a result correct to 3d.p.??