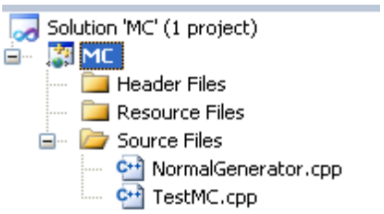
Groups C&D: Monte Carlo Pricing Methods

C. Monte Carlo 101

Answer the following questions:

a) Study the source code in the file *TestMC.cpp* and relate it to the theory that we have just discussed. The project should contain the following source files and you need to set project settings in VS to point to the correct header files:



Compile and run the program *as is* and make sure there are no errors.

Ans :

The code upon compilation works perfectly fine after adding the paths to our respective boost directories.The main intent of the code provided inside the file TestMC is to provide an one factor Monte Carlo with explicit Euler.

Firstly inside the main method; we define a Put option with some given parameters and ask for the number of simulations to be done as input.

The first step is to replace continuous time by discrete time. To this end, we divide the interval [0, T] (where T is the expiry date) into a number of subintervals as shown in Figure 1. We define $N + 1$ *mesh points* as follows:

$$0 = t_0 < t_1 < .. < t_n < t_{n+1} < .. < t_N = T.$$

In this case we define a set of *subintervals* (t_n, t_{n+1}) of size $\Delta t_n \equiv t_{n+1} - t_n, \quad 0 \leq n \leq N-1.$

Now we relate with the above given text which basically describes the discretization of continuous time into a bunch of mesh points.These mesh points are implemented by first taking a range of sub-intervals over the expiry time T which was hardcoded into the main method initially and storing these mesh points into a data structure called Range.This Range data structure stores both the start and finish of the range and is also responsible for returning a mesh of n+1 points which is equally divided called the mesh array.Because the subintervals inside this Range is calculated by taking out the average of the overall range size.The mesh is uniform.

Next, the code in TestMC.cpp initializes a myNormal pointer to boost libraries and normal_distribution for generating a uniform random number. Purpose of this is to construct a simulated path of the underlying stock X.

Having defined how to subdivide $[0,T]$ into subintervals, we are now ready to motivate our finite difference schemes; for the moment we examine the scalar linear SDE with constant coefficients:

$$\begin{aligned} dX &= aXdt + bXdW, \quad a,b \text{ constant} \\ X(0) &= A. \end{aligned} \tag{6}$$

While trying to define the trajectory ; we realise using a normal distribution with mean 0 and variance 1 will be a right fit because of the following description of the wiener increments :

$$\begin{cases} \Delta t_n = \Delta t = T/N, \quad 0 \le n \le N-1 \\ \Delta W_n = \sqrt{\Delta t} z_n, \text{ where } z_n \sim N(0,1). \end{cases}$$

Post this ; we start running a for loop for all the provided simulations.We also do checkpointing by logging the state of the loop control variable after every 1000 iterations. Then inside every simulation ; we iterate over all the n+1 mesh points to reach Xn+1 from Xo As mentioned below in the doc :

$$\begin{cases} X_{n+1} = X_n + aX_n\Delta t_n + bX_n\Delta W_n \\ X_0 = A. \end{cases} \tag{10}$$

The code actually inside the simulation and mesh points loop implements the formula for explicit Euler as

<u>VNew = VOld + (k * drift(x[index-1], VOld)) + (sqrk * diffusion(x[index-1], VOld) * dW);</u>

As we can see Vold and Vnew are the old and new prices after every subinterval.
K is nothing but Δ **t***n*. And sqrk is nothing but square root of Δt.
dW alone is nothing but the normal random number which multiplying with sqrk gives us our Δ**W***n*.

<u>Now coming to the drift and diffusion part :</u>

The drift takes two parameters viz. t and X (**VOld**,***Xn-1).***However the drift does not even use this t which is Xn-1 in this case.The drift simply multiplies the underlying stock price with the risk free rate which tells us the coefficient ***a*** used in the explicit euler equation is risk neutral. The diffusion part also takes the exactly same parameters without using t but uses a local variable called betacev whose value is 1.The diffusion method basically returns σ*(X^bcev) which is nothing but σ*(X) and this represents the coefficient ***b*** in the equation 10 for explicit euler. And given that a and b are constants in geometric brownian motion we can assume both diffusion and drift serve as those constants.

So we implement this and at every simulation take the average of the payoff value provided by this latest value of Xn+1 and after all the simulations are completed; we discount the average payoff at time t = T by using the risk free discount rate (r) to obtain the approximate price of the option at time t = 0.

b) Run the MC program again with data from Batches 1 and 2. Experiment with different value of NT (time steps) and NSIM (simulations or draws). In particular, how many time steps and draws do you need in order to get the same accuracy as the exact solution? How is the accuracy affected by different values for NT/NSIM?

However there is no solution with the same accuracy as the exact solution ; with certain number of simulations you can reach to a point where error is only around 0.00002 as we could achieve with 500 NT and 15 million NSims for a call option for batch 1.Accuracy is affected by different values mainly viz. NSims and NTs but they are not related in a linear fashion.We can also observe that by decreasing the NT to around 300, yet increasing NSIM to 15,000,000 (as an example), negatively impacts rate of convergence.

Batch #1 - call	NT	NSIM	Closed Solution	Value -(Call)	Error
	500	500,000	2.13337	2.1253	0.00807
	500	900,000	2.13337	2.13058	0.00279
	300	1,000,000	2.13337	2.1347	-0.00133
	500	1,000,000	2.13337	2.13071	0.00266
	500	3,000,000	2.13337	2.13232	0.00105
	300	15,000,000	2.13337	2.13179	0.00158
	500	15,000,000	2.13337	2.13335	0.00002
Batch #2 - Call	500	500,000	7.96557	7.9418	0.02377
	500	700,000	7.96557	7.94876	0.01681
	500	900,000	7.96557	7.96172	0.00385
	300	1,000,000	7.96557	7.97235	-0.00678
	500	1,000,000	7.96557	7.96142	0.00415
	500	3,000,000	7.96557	7.96675	-0.00118
	300	15,000,000	7.96557	7.96437	0.0012
	500	15,000,000	7.96557	7.96672	-0.00115

	Closed Solution	Value - (Put)	Error
Batch #1 - Put	5.84628	5.85493	-0.00865
	5.84628	5.84038	0.0059
	5.84628	5.85369	-0.00741
	5.84628	5.84125	0.00503
	5.84628	5.84109	0.00519
	5.84628	5.84504	0.00124
	5.84628	5.84624	0.00004
Batch #2 - Put	7.96557	7.97869	-0.01312
	7.96557	7.97051	-0.00494
	7.96557	7.95525	0.01032
	7.96557	7.98455	-0.01898
	7.96557	7.95663	0.00894
	7.96557	7.95794	0.00763
	7.96557	7.9652	0.00037
	7.96557	7.9666	-0.00103

Having said that, we can also observe that there is not necessarily a linear relationship between NT and error (accuracy). Meaning, sometimes too high of an NT can lead to inaccuracies as well. From a theoretical point of view, the error does decrease as NT approaches infinity, but in reality, this may hit a limit due to round-off errors. Additionally, we can observe that accuracy might worsen as NSIM approaches infinity. This too, is due to the non-linear nature of Monte Carlo.

c.)

With reference to the call option in batch 4, a 2 decimal accuracy doesn't seem to be achievable. The closest I was able to get was by using NT = 700 and NSIM = 1,000,000.

Batch #4 - call	NT	NSIM	Closed Solution	Value -(Call)	Error
batch 4	500	500,000	92.1757	91.858	0.3177
	500	1,000,000	92.1757	91.845	0.3307
	500	5,000,000	92.1757	91.7312	0.4445
	500	10,000,000	92.1757	91.6058	0.5699
	700	1,000,000	92.1757	92.2405	-0.0648
	900	1,000,000	92.1757	91.8968	0.2789
	950	1,000,000	92.1757	92.5465	-0.3708
	1000	1,000,000	92.1757	91.5646	0.6111
	1000	5,000,000	92.1757	91.7444	0.4313
	500	30,000,000	92.1757	91.5686	0.6071

When simulating the put option in batch 4, a 2 decimal accuracy can be achieved when using NT = 950 and NSIM = 1,000,000. Above this lower bound, and in particular reference to NT, we can only expect better accuracy.

Batch #4 - put	NT	NSIM	Closed Solution	Value - (Put)	Error	
	500	1,000,000	1.2475	1.25428	-0.00678	
	500	5,000,000	1.2475	1.25478	-0.00728	
	500	10,000,000	1.2475	1.25594	-0.00844	
	500	15,000,000	1.2475	1.25606	-0.00856	
	700	1,000,000	1.2475	1.25214	-0.00464	
	900	1,000,000	1.2475	1.2511	-0.0036	
	950	1,000,000	1.2475	**1.24942**	-0.00192	
	1000	1,000,000	1.2475	**1.24861**	-0.00111	
	1000	5,000,000	1.2475	1.25218	-0.00468	
	500	30,000,000	1.2475	1.25582	-0.00832	