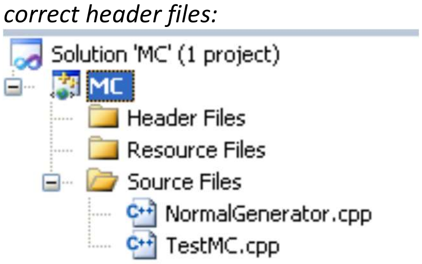
GROUP C&D – MONTE CARLO PRICING METHODS

1. Answers to questions in write-up.

<u>Group C - Monte Carlo 101</u>

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.

<u>Analysis:</u>
In studying the source code in the file TestMC.cpp, I have the following observations:

The write-up discusses how to replace continuous time by discrete time. In doing so, one must 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 $\Delta t_n \equiv t_{n+1} - t_n, \quad 0 \le n \le N-1.$

In contrast, the code in *TestMC.cpp* prompts the user to input the number of simulations (NSim), as well as the number of time steps (NT). A mesh point array is then instantiated via Range.cpp as [0, T] / time steps, where T is the option's expiry. We can therefore assume that the N subintervals are in fact a uniform mesh.

Next, the code in *TestMC.cpp* initializes a pointer (myNormal) to boost libraries lagged_fibonacci607 and normal_distribution for purposes of generating a uniform random number. Purpose of this is to construct a simulated path of the underlying stock X.

$$dX = aXdt + bXdW, \quad a, b \text{ constant}$$

Because we can think of ***dW*** as being a normally distribution random variable with zero mean and variance ***dt,*** we can confirm the code is correct in using normal_distribution of (0,1), as based on the z_n provided below.

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

The code then sets up a *for* loop – iterating through every NSim. The key to Monte Carlo is to make use of the "law of large numbers" to approximate the price of the option. Thus, the purpose of this *for* loop is to iterate through as many simulations from the normal distribution N(0,1), as the variable *X*, and then compute the pay-off, followed by averaging the sum of these pay-offs, and then finally taking the risk-free discount to obtain the approximate price of the option.

Continuing to follow through the code in *TestMC.*cpp, the code then sets the initial boundary condition of VOld equal to the underlying asset price S_0, followed by a *for* loop to discretize time T. The code utilizes a Vector object. As part of the *for* loop, $t_0 = 0$ in the mesh is stored in the Vector's index value of 1 (in contrast with C++ arrays which typically begin at 0).

We now turn our attention to lines 189 – 191 in *TestMC.cpp*. This code represents the implementation of the Explicit Euler method. Variable *k* is meant to represent Δt_n , while variable *sqrk* is mean to represent ΔW_n as shown in the equation below. The Z_n, as already stated above, is our randomly generated number from a normal distribution with mean 0 and variance 1.

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

We arrive at this conclusion based on our understanding that the mesh size is constant, and that the factors *a* and *b* are constant. The use of the drift() function represents the non-stochastic drift coefficient, while the diffusion() method represents the coefficient of volatility, multiplied by the stochastic *dW* term, or in our particular case, ΔW_n, whereby:

$$\Delta W_n = W(t_{n+1}) - W(t_n), \quad 0 \le n \le N-1.$$

Turning our attention to the code snippet: `drift(x[index-1], VOld),` we can make several observations. First, the argument t in function diffusion isn't even used within the function definition. Argument *t* is meant to represent x_{n-1}.. Second, the calculation in function drift() specifies that the risk-free rate be multiplied by the underlying asset price, VOld. This would imply that this is a risk-neutral pricing model, whereby the drift term is equal to the risk-free rate, multiplied by VOld, which represents X_n.

Moving onto the diffusion() function, we observe that argument *double t* is not referenced within the function definition. We can also observe that VOld is passed in argument *double X*¸and is multiplied by a volatility factor of data->sig (which we designate via the OptionData struct). Given that coefficients *a* and *b* are constant in the geometric Brownian motion model, we can therefore, view the diffusion() function as a constant. The same can be said of the drift() function as well.

The rest of the code in *TestMC.cpp* follows the write-up whereby the call price is calculated at *t=T* via the payoff function. The code then calculates the average call price at t=T, and then is discounted to *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?*

Given that the Monte Carlo is a simulation, we will never obtain the exact same answer as the exact solution. As shown in the four screenshots below, we can see that by increasing NT (time steps) to around 500, and by increasing NSIM (simulations or draws) towards infinity, that the simulated prices continue to asymptotically approach the exact price. 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.

Time to expiry also appear to be a factor, whereby, the longer the time to expiry, the higher NT/NSIM must be in order to achieve a relatively equivalent measure of accuracy. This pattern is evident in the data shown below. Batch 1 has T = 0.25, whereas in Batch 2, T = 1.0.

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.

NT	NSIM	Closed Solution	Value - Call	Absolute Error	SD	SE
300	15,000,000	2.13337	2.13179	0.0015800	4.51307	0.00116527
500	**15,000,000**	**2.13337**	**2.13335**	**0.0000200**	**4.51475**	**0.0011657**
500	3,000,000	2.13337	2.13232	0.0010500	4.51043	0.0026041
500	1,000,000	2.13337	2.13071	0.0026600	4.51286	0.00451286
300	1,000,000	2.13337	2.1347	(0.0013300)	4.51766	0.00451766
500	900,000	2.13337	2.13058	0.0027900	4.51349	0.00475764
500	500,000	2.13337	2.1253	0.0080700	4.51365	0.00638326

Figure 1 – Batch 1 Call

NT	NSIM	Closed Solution	Value - Put	Absolute Error	SD	SE
500	**500,000**	**5.84628**	**5.84624**	**0.0000400**	**6.0481**	**0.00156161**
500	**900,000**	**5.84628**	**5.84504**	**0.0012400**	**6.04849**	**0.00156171**
300	1,000,000	5.84628	5.84109	0.0051900	6.04822	0.00349194
500	1,000,000	5.84628	5.84125	0.0050300	6.04743	0.00604743
500	3,000,000	5.84628	5.85369	(0.0074100)	6.05714	0.00605714
500	15,000,000	5.84628	5.84038	0.0059000	6.04769	0.00637483
300	15,000,000	5.84628	5.85493	(0.0086500)	6.05373	0.00856126

Figure 2 – Batch 1 Put

NT	NSIM	Closed Solution	Value - Call	Absolute Error	SD	SE
300	**15,000,000**	**7.96557**	**7.96437**	**0.0012**	**13.1427**	**0.0033934**
500	15,000,000	7.96557	7.96672	-0.00115	13.1473	0.0033946
500	3,000,000	7.96557	7.96675	-0.00118	13.1372	0.0075848
500	1,000,000	7.96557	7.96142	0.00415	13.1421	0.0131421
300	1,000,000	7.96557	7.97235	-0.00678	13.1535	0.0131535
500	900,000	7.96557	7.96172	0.00385	13.1433	0.0138542
500	700,000	7.96557	7.94876	0.01681	13.1404	0.0157058
500	500,000	7.96557	7.9418	0.02377	13.1421	0.0185857

Figure 3 – Batch 2 Call

NT	NSIM	Closed Solution	Value - Put	Absolute Error	SD	SE
500	15,000,000	7.96557	7.9666	-0.00103	10.4055	0.002687
300	**15,000,000**	**7.96557**	**7.9652**	**0.00037**	**10.4069**	**0.002687**
500	3,000,000	7.96557	7.95794	0.00763	10.4058	0.006008
500	1,000,000	7.96557	7.95663	0.00894	10.4052	0.0104052
300	1,000,000	7.96557	7.98455	-0.01898	10.4229	0.0104229
500	900,000	7.96557	7.95525	0.01032	10.4058	0.0109687
500	700,000	7.96557	7.97051	-0.00494	10.4154	0.0124488
500	500,000	7.96557	7.97869	-0.01312	10.4208	0.0147373

Figure 4 – Batch 2 Put

c) *Now we do some stress-testing of the MC method. Take Batch 4. What values do we need to assign to NT and NSIM in order to get an accuracy to two places behind the decimal point? How is the accuracy affected by different values for NT/NSIM?*

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.

NT	NSIM	Closed Solution	Value - Put	Absolute Error	SD	SE
1000	1,000,000	1.2475	**1.24861**	-0.00111	2.4550400	0.0024550
950	1,000,000	1.2475	**1.24942**	-0.00192	2.4551600	0.0024552
1000	5,000,000	1.2475	1.25218	-0.00468	2.4567800	0.0010987
900	1,000,000	1.2475	1.2511	-0.0036	2.4571400	0.0024571
700	1,000,000	1.2475	1.25214	-0.00464	2.4571700	0.0024572
500	5,000,000	1.2475	1.25478	-0.00728	2.4605100	0.0011004
500	1,000,000	1.2475	1.25428	-0.00678	2.4606800	0.0024607
500	30,000,000	1.2475	1.25582	-0.00832	2.46112	0.000449337
500	15,000,000	1.2475	1.25606	-0.00856	2.4612900	0.0006355
500	10,000,000	1.2475	1.25594	-0.00844	2.4614800	0.0007784

Figure 5 – Batch 4 Put

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. By increasing these parameters, the absolute error worsened while the SE decreased (since a higher NSIM inversely lowers SE).

NT	NSIM	Closed Solution	Value - Call	Absolute Error	SD	SE
500	30,000,000	92.1757	91.5686	0.6071	359.504	0.06564
500	10,000,000	92.1757	91.6058	0.5699	367.038	0.11607
600	6,000,000	92.1757	91.734	0.4417	364.188	0.14868
1000	5,000,000	92.1757	91.7444	0.4313	361.598	0.161711
970	5,000,000	92.1757	91.8001	0.3756	368.538	0.164815
500	5,000,000	92.1757	91.7312	0.4445	378.661	0.16934
600	1,000,000	92.1757	91.5996	0.5761	351.59	0.35159
1000	1,000,000	92.1757	91.5646	0.6111	352.824	0.352824
900	1,000,000	92.1757	91.8968	0.2789	358.897	0.358897
975	1,000,000	92.1757	92.0891	0.0866	367.844	0.367844
970	1,000,000	92.1757	92.2432	-0.0675	368.056	0.368056
500	1,000,000	92.1757	91.845	0.3307	375.215	0.37522
974	1,000,000	92.1757	92.281	-0.1053	381.853	0.381853
700	**1,000,000**	**92.1757**	**92.2405**	**-0.0648**	**392.991**	**0.392991**
950	1,000,000	92.1757	92.5465	-0.3708	398.527	0.398527
500	500,000	92.1757	91.858	0.3177	372.554	0.52687

Figure 6 – Batch 4 Call

Group D - Advanced Monte Carlo

a) *Create generic functions to compute the standard deviation and standard error based on the above formulae. The inputs are a vector of size M (M = NSIM), the interest-free rate and expiry time T. Integrate this new code into TestMC.cpp. Make sure that the code compiles.*

Working Excel code is saved in the CODE\Group D folder.

By referring to the screenshot below, you will note that I cross-verified the C++ standard deviation function I wrote with that of the STD.S excel function. In order to have the numbers match, I multiplied the output of the Excel STD.S by exp(-rT). You can find this in the excel file - *Group C_D Analysis*, tab Standard Deviation, located in the Documentation\Group C_D folder.

	Call Output Price stored in vector	
[0]	6.313752835	39.86347487
[1]	0	0
[2]	9.404999369	88.45401314
[3]	7.855368726	61.70681781
[4]	0	0
[5]	3.845698511	14.78939704
[6]	10.7438853	115.4310712
[7]	0	0
[8]	21.89795392	479.5203861
[9]	0	0
[10]	0	0
[11]	0	0
[12]	6.801831741	46.26491504
[13]	0	0
[14]	12.38221042	153.3191348
[15]	1.157845557	1.340606334
[16]	5.45578342	29.76557272
[17]	18.35551051	336.9247661
[18]	0	0
[19]	0	0
	Sum	**SumSquares**
	104.2148403	**1367.380155**
Excel STD.S		
6.5868		
x exp(-0.08*0.25)		
6.4564		
Standard Deviation coded in TestMC.cpp		
6.4564		

Figure 7 – Verifying Standard Deviation C++ code with Excel

b) *Run the MC program again with data from Batches 1 and 2. Experiment with different values of NT (time steps) and NSIM (simulations or draws). How do SD and SE react for these different run parameters, and is there any pattern in regards to the accuracy of the MC (when compared to the exact method)?*

SD and SE appear to decrease as NSIM approaches infinity, with SE having a more direct correlation with NSIM approaching infinity, given how SE is derived. This makes sense – the more data we have, the more precise our estimate is. We can also observe that NSIM impacts SE more so than NT.

NT	NSIM	Closed Solution	Value - Call	Absolute Error	SD	SE
300	1,000,000	2.13337	2.1347	(0.0013300)	4.51766	0.00451766
500	15,000,000	**2.13337**	**2.13335**	**0.0000200**	4.51475	0.0011657
500	3,000,000	2.13337	2.13232	0.0010500	4.51043	0.0026041
300	15,000,000	2.13337	2.13179	0.0015800	4.51307	0.00116527
500	1,000,000	2.13337	2.13071	0.0026600	4.51286	0.00451286
500	900,000	2.13337	2.13058	0.0027900	4.51349	0.00475764
500	500,000	2.13337	2.1253	0.0080700	4.51365	0.00638326

Figure 8 – Batch 1 Call

NT	NSIM	Closed Solution	Value - Put	Absolute Error	SD	SE
500	500,000	5.84628	5.85493	(0.0086500)	6.05373	0.00856126
300	1,000,000	5.84628	5.85369	(0.0074100)	6.05714	0.00605714
500	**15,000,000**	**5.84628**	**5.84624**	**0.0000400**	**6.0481**	**0.00156161**
300	15,000,000	5.84628	5.84504	0.0012400	6.04849	0.00156171
500	1,000,000	5.84628	5.84125	0.0050300	6.04743	0.00604743
500	3,000,000	5.84628	5.84109	0.0051900	6.04822	0.00349194
500	900,000	5.84628	5.84038	0.0059000	6.04769	0.00637483

Figure 9 – Batch 1 Put

NT	NSIM	Closed Solution	Value - Call	Absolute Error	SD	SE
300	**15,000,000**	**7.96557**	**7.96437**	**0.0012**	**13.1427**	**0.0033934**
500	15,000,000	7.96557	7.96672	-0.00115	13.1473	0.0033946
500	3,000,000	7.96557	7.96675	-0.00118	13.1372	0.0075848
500	1,000,000	7.96557	7.96142	0.00415	13.1421	0.0131421
300	1,000,000	7.96557	7.97235	-0.00678	13.1535	0.0131535
500	900,000	7.96557	7.96172	0.00385	13.1433	0.0138542
500	700,000	7.96557	7.94876	0.01681	13.1404	0.0157058
500	500,000	7.96557	7.9418	0.02377	13.1421	0.0185857

Figure 10 – Batch 2 Call

NT	NSIM	Closed Solution	Value - Put	Absolute Error	SD	SE
500	15,000,000	7.96557	7.9666	-0.00103	10.4055	0.002687
300	**15,000,000**	**7.96557**	**7.9652**	**0.00037**	**10.4069**	**0.002687**
500	3,000,000	7.96557	7.95794	0.00763	10.4058	0.006008
500	1,000,000	7.96557	7.95663	0.00894	10.4052	0.0104052
300	1,000,000	7.96557	7.98455	-0.01898	10.4229	0.0104229
500	900,000	7.96557	7.95525	0.01032	10.4058	0.0109687
500	700,000	7.96557	7.97051	-0.00494	10.4154	0.0124488
500	500,000	7.96557	7.97869	-0.01312	10.4208	0.0147373

Figure 11 – Batch 2 Put

NT	NSIM	Closed Solution	Value - Call	Absolute Error	SD	SE
500	10,000,000	0.204058	0.203442	0.000616	1.02022	0.000322623
600	6,000,000	0.204058	0.20349	0.000568	1.0208	0.000416741
500	5,000,000	0.204058	0.203531	0.000527	1.02188	0.00045700
950	5,000,000	0.204058	0.204532	-0.000474	1.02572	0.000458717
900	1,000,000	0.204058	0.203715	0.000343	1.02036	0.00102036
600	1,000,000	0.204058	0.203489	0.000569	1.02074	0.00102074
500	1,000,000	0.204058	0.203168	0.00089	1.0244	0.0010244
700	**1,000,000**	**0.204058**	**0.20433**	**-0.000272**	**1.02704**	**0.00102704**

Figure 12 – Batch 3 Call

NT	NSIM	Closed Solution	Value - Put	Absolute Error	SD	SE
500	10,000,000	4.07326	4.0723	0.00096	2.09642	0.000662945
600	6,000,000	4.07326	4.07148	0.00178	2.09632	0.000855821
500	5,000,000	4.07326	4.0715	0.00176	2.09584	0.00093729
950	**5,000,000**	**4.07326**	**4.07327**	**-1E-05**	**2.09617**	**0.000937434**
500	1,000,000	4.07326	4.07227	0.00099	2.09477	0.00209477
600	1,000,000	4.07326	4.07234	0.00092	2.09507	0.00209507
700	1,000,000	4.07326	4.07147	0.00179	2.09552	0.00209552
900	1,000,000	4.07326	4.06948	0.00378	2.09718	0.00209718

Figure 13 – Batch 3 Put

NT	NSIM	Closed Solution	Value - Call	Absolute Error	SD	SE
500	30,000,000	92.1757	91.5686	0.6071	359.504	0.06564
500	10,000,000	92.1757	91.6058	0.5699	367.038	0.11607
600	6,000,000	92.1757	91.734	0.4417	364.188	0.14868
1000	5,000,000	92.1757	91.7444	0.4313	361.598	0.161711
970	5,000,000	92.1757	91.8001	0.3756	368.538	0.164815
500	5,000,000	92.1757	91.7312	0.4445	378.661	0.16934
600	1,000,000	92.1757	91.5996	0.5761	351.59	0.35159
1000	1,000,000	92.1757	91.5646	0.6111	352.824	0.352824
900	1,000,000	92.1757	91.8968	0.2789	358.897	0.358897
975	1,000,000	92.1757	92.0891	0.0866	367.844	0.367844
970	1,000,000	92.1757	92.2432	-0.0675	368.056	0.368056
500	1,000,000	92.1757	91.845	0.3307	375.215	0.37522
974	1,000,000	92.1757	92.281	-0.1053	381.853	0.381853
700	**1,000,000**	**92.1757**	**92.2405**	**-0.0648**	**392.991**	**0.392991**
950	1,000,000	92.1757	92.5465	-0.3708	398.527	0.398527
500	500,000	92.1757	91.858	0.3177	372.554	0.52687

Figure 14 – Batch 4 Call

NT	NSIM	Closed Solution	Value - Put	Absolute Error	SD	SE
1000	1,000,000	1.2475	**1.24861**	-0.00111	2.4550400	0.0024550
950	1,000,000	1.2475	**1.24942**	-0.00192	2.4551600	0.0024552
1000	5,000,000	1.2475	1.25218	-0.00468	2.4567800	0.0010987
900	1,000,000	1.2475	1.2511	-0.0036	2.4571400	0.0024571
700	1,000,000	1.2475	1.25214	-0.00464	2.4571700	0.0024572
500	5,000,000	1.2475	1.25478	-0.00728	2.4605100	0.0011004
500	1,000,000	1.2475	1.25428	-0.00678	2.4606800	0.0024607
500	30,000,000	1.2475	1.25582	-0.00832	2.46112	0.000449337
500	15,000,000	1.2475	1.25606	-0.00856	2.4612900	0.0006355
500	10,000,000	1.2475	1.25594	-0.00844	2.4614800	0.0007784

Figure 15 – Batch 4 Put