Scott Sidoli

6/12/19

Group C: Monte Carlo 101

Part A)

We study the code in TestMC.cpp and relate it to the given theory. First, we discuss the include files. OptionData.hpp contains the struct which packages the relevant data used for pricing options including the *T*, *K*, *σ*, *r*, *b*, and *S*, which are the expiry time, strike price, volatility, risk-free interest rate, cost-of-carry, and current stock price, respectively. This is necessary for any option pricing calculator. Next, we have UtilitiesDJD/RNG/NormalGenerator.hpp contains the base class that allows us to generate random number according to a normal distribution with mean 0 and standard deviation 1. This will be used in simulating the stock path. Continuing, we have UtilitiesDJD/Geometry/Range.cpp, which contains the functionality to create the time interval which we will break into subintervals of equal width in accordance with the mesh function. The library, cmath, is include for computations, and iostream to handle user input/output.

Next, we templated method and the SDE namespace. We have a method to print the size of a given vector and is respective entries. We have the definition of the stochastic differential equation which contains the option data, the drift term, the diffusion term, and the diffusion term needed for the Milstein method, a technique for approximating numerical solutions to stochastic differential equations.

After the conclusion of the namespace, we have the main function. An option is created whose values are set. The determination of put or call is made and the initial stock price, S\_0, is set. The number of subintervals for the time interval is initialized as N and is set to 100, but the user can input their choice of N. On line 79, we have the range functionality used to create the time interval going from *t=0* to *t=T*, the expiry time. VOld is initialized as the current stock price and VNew is introduced. On line 83, a vector, *x*, is given by the mesh function and presumably divides the time interval into N subintervals of equal width. From line 87-89, the number of simulations is set. This is *M* in the text on Monte Carlo. On line 91, k is the width of one subinterval (*Δt*) and sqrk is . On line 95 and 96, we initialize *dW* (*N(0, 1)*) and the option price, denoted *price*. We create a copy of the standard normal distribution, *N(0, 1)*, denoted myNormal. Next, we define a stochastic differential equation whose solution will simulate a path taken by the stock price. The initial option data is incorporated with a pass by reference on line 102. A count is created to track the number of times the stock price hits 0. We then have a for loop which is designed to create a path for the stock price with each iteration. We will have NSim total number of paths. After 10000 iteration, the user is notified of the milestone. Recall that the vector, *x*, is the mesh of the time interval. We have a for loop (nested) which creates the path that the stock takes. Using the finite difference method (explicit), the new stock price is calculated based on the previous:

This is analogous to both equations (9) and (10) in the HW. VOld is set to the new stock price and the method repeats until we reach the end of the mesh. We count the number of times that the price hits 0. We exit the loop with the final VNew and compute the payoff. The price is then given by the average value of these outputs which is then discounted on line 139. We cleanup and print the results of the Monte Carlo simulation.

Part B)

We run TestMC.cpp with batch 1 and experiment with different values of NSim and N. Batch 1 has values: T = 0.25, K = 65, sig = 0.30, r = 0.08, S = 60, and b = 0.08 (standard Black-Scholes). The exact call price is 2.13337. We obtain the following result: (left most column is N, top row is NSim)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 15,000 | 30,000 | 45,000 | 60,000 | 75,000 | 90,000 | 105,000 |
| 25 | 2.04754 | 2.10394 | 2.10194 | 2.09274 | 2.09509 | 2.0997 | 2.10123 |
| 40 | 2.08397 | 2.10046 | 2.0929 | 2.10134 | 2.10706 | 2.10934 | 2.1181 |
| 55 | 2.10596 | 2.0785 | 2.08258 | 2.09548 | 2.10307 | 2.12024 | 2.11816 |
| 70 | 2.11091 | 2.07539 | 2.09685 | 2.11165 | 2.11607 | 2.1244 | 2.12836 |
| 85 | 2.11827 | 2.07519 | 2.09794 | 2.12171 | 2.13297 | 2.13527 | 2.12562 |
| 100 | 2.0918 | 2.08685 | 2.11886 | 2.11739 | 2.12827 | 2.12415 | 2.12883 |
| 115 | 2.09673 | 2.08128 | 2.12171 | 2.1316 | 2.12863 | 2.13201 | 2.13165 |

We observe that in the lower right corner of the table we appear to be closer to the exact call price of 2.13337. Except for a few instances, as one increases the number of simulations (i.e. as one moves across a row from left to right) we get closer to the exact call price. Similarly, as one moves down a column (increasing the number of time steps) we get closer to the exact call price. Again, this is not without exception, but when running these simulations, one expects some variability. We also perform this with N = 200 and NSim = 1000000. We get Call Price = 2.13356. We try again with N = 200 and NSim = 2000000. We get Call Price = 2.13343. Even with N = 200, NSim = 10,000,000, we do not get the exact answer. In this last case, we get 2.13276. Due to sampling variability, it is unlikely that any sample average will be identical to the population average (i.e. the output of BS). Also worth noting is that the vast majority of Monte Carlo simulations produced call prices that were below the exact price.

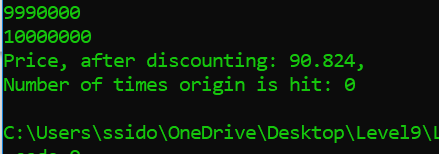
For batch 2, we experiment with different values of N and NSim. For batch 2 the parameter values are T = 1.0, K = 100, sig = 0.2, r = 0.0, S = 100, and b = 0.0. We have an exact call price of 7.96557. We use a somewhat smaller to table for this batch.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 100000 | 500000 | 1000000 | 2000000 |
| 50 | 7.90297 | 7.97618 | 7.97678 | 7.96661 |
| 100 | 7.94362 | 7.98116 | 7.9625 | 9.95934 |
| 150 | 8.00869 | 7.97718 | 7.9577 | 7.96207 |
| 200 | 8.0006 | 7.96235 | 7.96113 | 7.96492 |
| 250 | 8.0538 | 7.95764 | 7.95752 | 7.96358 |

Again, we notice that none of the Monte Carlo outputs exactly mirror the output of BS. General trends are still apparent. We notice that the last column was accurate to two decimal places in 80% of Monte Carlo simulation outputs. As this wasn’t observed in the table for batch 1, we can conclude that we get our most accurate estimations when NSim >= 2,000,000. In reviewing both tables, we can conclude that as NSim approaches infinity, the outputted option prices approach the exact solution to Black-Scholes.

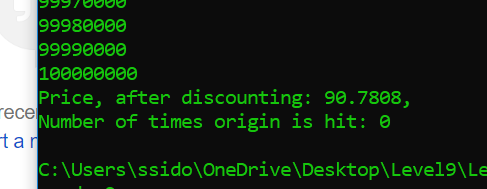
Part C)

For batch 4, we start with NSim = 100,000 and N = 100. The exact call price is 92.17570. We get 89.4248. This is particularly far off. We take NSim to be 2,000,000 and keep N at 100. We get 89.2492. We take N to be 200 and keep NSim at 2,000,000. We get 90.9017. We run with NSim = 10,000,000:



We notice that even for high NSim, we are not that close to the exact price. We ran

NSim = 100,000,000 and found



This is actually further from the exact price than what we observed previously. We can conclude that we should perhaps alter our NStep. For this case, it was left at 200.