Monte Carlo 101

Running the code

The program successfully ran after including the boost libraries to the project.

Theory Analysis

The theory starts by talking about scalar linear SDE with constant coefficients, i.e.,

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

The solution of this SDE is the value of X at time step T. To solve this, the time interval ([0, T]) is broken down into N equal timesteps Δt_n . The integral of this equation is on a continuous time range which is discretized and what we get is the **explicit Euler-Maruyama** scheme which is a popular method for approximating the solution of SDEs.

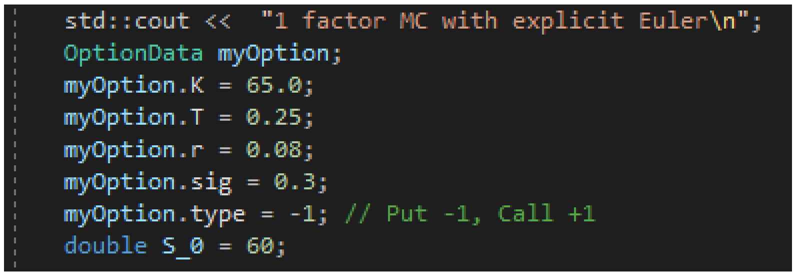
$$\begin{cases} X_{n+1} = X_n + aX_n\Delta t_n + bX_n\Delta W_n \\ X_0 \quad = A. \end{cases} \tag{10}$$

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

The stock price at t_{n+1} is calculated using the stock price at t_n with random movement through ΔW_n .This random movement is called as increments of the Wiener process which are generated using random number generators. Here we use the normal random number generator for this purpose. Through the computed stock price for t=T, the option price is computed using the payoff function and discounted exponentially over time T (starting time t=0). This process is repeated over many iterations for accuracy and an average is taken.

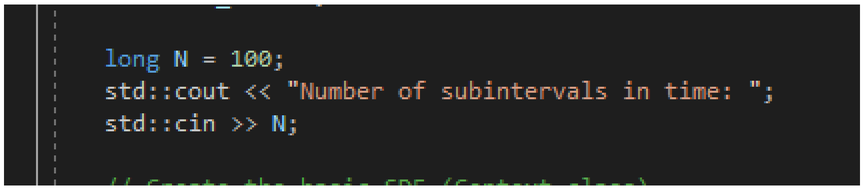
Mapping code to theory

1. Option params initialization



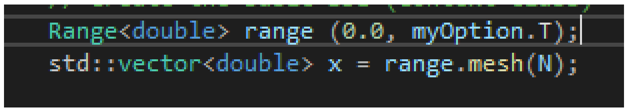
```
std::cout <<  "1 factor MC with explicit Euler\n";
OptionData myOption;
myOption.K = 65.0;
myOption.T = 0.25;
myOption.r = 0.08;
myOption.sig = 0.3;
myOption.type = -1; // Put -1, Call +1
double S_0 = 60;
```

2. Number of time steps (N)



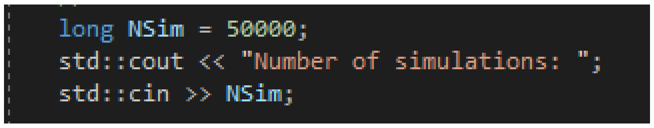
```
long N = 100;
std::cout << "Number of subintervals in time: ";
std::cin >> N;
```

3. Range denoting $t \epsilon [0, T]$ and meshing it in a vector for time steps



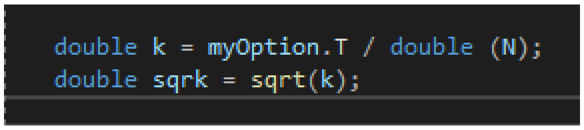
```
Range<double> range (0.0, myOption.T);
std::vector<double> x = range.mesh(N);
```

4. Number of simulations for MC



```
long NSim = 50000;
std::cout << "Number of simulations: ";
std::cin >> NSim;
```

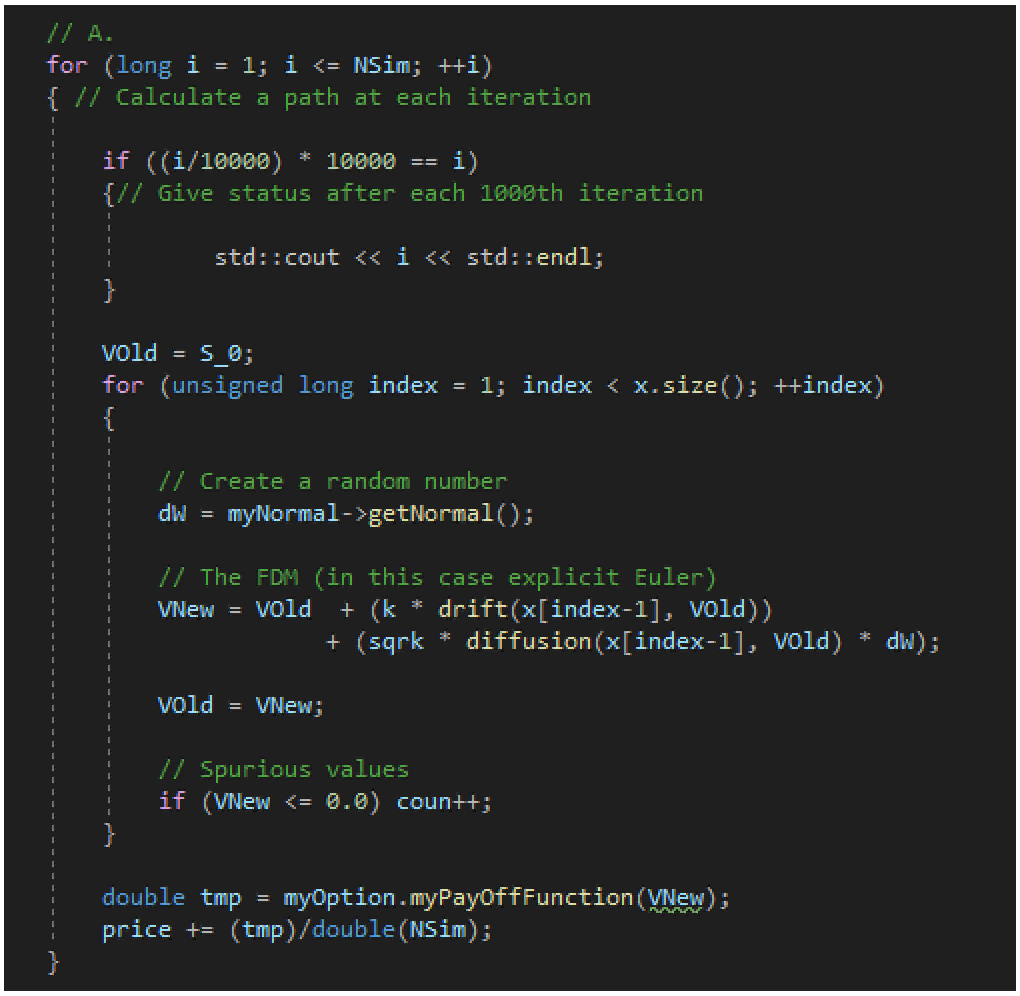
5. Δt and $\sqrt{\Delta t}$ (equal timesteps)



```
double k = myOption.T / double (N);
double sqrk = sqrt(k);
```

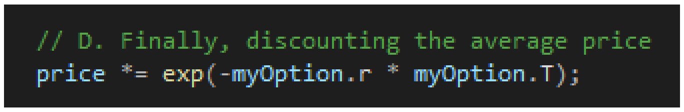
6. Run NSim iterations. For every iteration:
    a. Set starting stock price to S_0
    b. Loop over all N time steps. For every time step:
        i. Get a normal random number
        ii. Apply explicit euler to calculate stock price at current time step using stock price from previous time step. The *drift* serves as $a * V_{old}$ and *diffusion* serves as $b * V_{old}$.
        iii. Update the previous stock price with the newly calculated stock price which will be used in the next iteration of this loop.

c. Calculate option price using stock price at t=T from this iteration.
d. Avg the price and add to the avg price across all simulations.



```
// A.
for (long i = 1; i <= NSim; ++i)
{ // Calculate a path at each iteration

    if ((i/10000) * 10000 == i)
    {// Give status after each 1000th iteration

            std::cout << i << std::endl;
    }

    VOld = S_0;
    for (unsigned long index = 1; index < x.size(); ++index)
    {

        // Create a random number
        dW = myNormal->getNormal();

        // The FDM (in this case explicit Euler)
        VNew = VOld  + (k * drift(x[index-1], VOld))
                    + (sqrk * diffusion(x[index-1], VOld) * dW);

        VOld = VNew;

        // Spurious values
        if (VNew <= 0.0) coun++;
    }

    double tmp = myOption.myPayOffFunction(VNew);
    price += (tmp)/double(NSim);
}
```

7. Discount option price using continuous discounting using risk free discount rate to get option price at t=0



```
// D. Finally, discounting the average price
price *= exp(-myOption.r * myOption.T);
```

This concludes the Monte Carlo execution and we get an option price at t=0.

Experimenting with NSim & NT

Batch 1

Experimenting with Call Option

NSim	NT	Call Price (MC)	Call Price (Exact)	Error
50000	100	2.11675	2.13337	-0.016624
50000	50	2.07387	2.13337	-0.0594969
100	50000	1.63003	2.13337	-0.503336
100000	10	2.11117	2.13337	-0.0222026
100000	1000	2.14465	2.13337	0.0112828
1000000	100	2.13271	2.13337	-0.000657763
500000	1000	2.13279	2.13337	-0.000578231

Put Option

NSim	NT	Put Price (MC)	Put Price (Exact)	Error
500000	1000	5.84122	5.84628	-0.00505653

Batch 2

Experimenting with Put Option

NSim	NT	Put Price (MC)	Put Price (Exact)	Error
50000	100	8.0132	7.96557	0.0476321
50000	50	8.01619	7.96557	0.0506155
100	50000	9.03346	7.96557	1.06789
100000	10	8.0067	7.96557	0.0411303
1	100000	4.75373	7.96557	-3.21184
1000000	100	7.97439	7.96557	0.00882419
10000000	10	7.98315	7.96557	0.0175806

Call Option

NSim	NT	Put Price (MC)	Put Price (Exact)	Error
1000000	100	7.9625	7.96557	-0.00306876

Observations around NSim & NT

Reaching to the exact solution seems impossible using the Monte Carlo method discussed here. Although the solution does converge and gets really close to the exact solution with a high number of simulations and a high number of time steps.
Interestingly, the number of simulations (NSim) seems to matter much more than the number of time steps (NT) with regards to accuracy. This makes sense as a large number of simulations allow for better approximation of the stochastic process. However, it is noteworthy, the improvement in accuracy with increasing number of simulations seems to be diminishing (non-linearly) in nature.
One test case (highlighted yellow) upon comparison also suggests that increasing NSim a lot can negatively impact accuracy.

Stress Testing

Batch 4:

Call Option

NSim	NT	Call Price (MC)	Call Price(Exact)	Error
1000000	10	71.6595	92.1757	-20.5162
1000000	100	89.5241	92.1757	-2.65163
1000000	400	91.5014	92.1757	-0.674281
1000000	700	92.2405	92.1757	0.0647586
1000000	1000	91.5646	92.1757	-0.611135

Convergence to two decimal places wasn't achieved for the call option with the above values for NSim and NT. Closest came with NSim = 1,000,000 and NT = 700.

Put Option

NSim	NT	Call Price (MC)	Call Price(Exact)	Error
100000	100	1.29604	1.2475	0.0485408
300000	100	1.28682	1.2475	0.0393248
1000000	100	1.29275	1.2475	0.0452478
1000000	1000	1.24861	1.2475	0.00110596

Convergence to two decimal places was achieved for the put option with the highlighted set of values.