Level 9 HW: Part C

David Leather

April 1st, 2025

Part C: MC 101

Question C.a

> *Study the source code in the file TestMC.cpp and relate it to the theory that we have just discussed.*

```
namespace SDEDefinition
{ // Defines drift + diffusion + data

   OptionData* data;            // The data for the option MC

   double drift(double t, double X)
   { // Drift term

      return (data->r)*X; // r - D
   }


   double diffusion(double t, double X)
   { // Diffusion term

      double betaCEV = 1.0;
      return data->sig * pow(X, betaCEV);

   }

   double diffusionDerivative(double t, double X)
   { // Diffusion term, needed for the Milstein method

      double betaCEV = 1.0;
      return 0.5 * (data->sig) * (betaCEV) * pow(X, 2.0 * betaCEV - 1.0);
   }
} // End of namespace
```

The above code parameterizes the SDE in Equation (6) and Equation (9) which shows that that the drift and diffusion coefficients are proportional to the current value, `X`.

$$dX = \alpha X dt + bX dW$$

Now the code moves on to the main function.

```
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;

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

First, parameters are set, including the option contract and the quantity of subintervals. N, needed to be created.

```
   // Create the basic SDE (Context class)
   Range<double> range (0.0, myOption.T);
   double VOld = S_0;
   double VNew;

   std::vector<double> x = range.mesh(N);
```

Then, the grid over time is represented by range, which is an object of class `Range<double>`. Then $N+1$ grid points are generated by code, `x = range.mesh(N)` , which results in a vector of doubles.

$$0 = t_0 < t_1, \cdots < t_n < t_{n+1} < \cdots < t_N = T$$

```
// V2 mediator stuff
long NSim = 50000;
std::cout << "Number of simulations: ";
std::cin >> NSim;

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

// Normal random number
double dW;
double price = 0.0;    // Option price

// NormalGenerator is a base class
NormalGenerator* myNormal = new BoostNormal();

using namespace SDEDefinition;
SDEDefinition::data = &myOption;
```

```
std::vector<double> res;
int coun = 0; // Number of times S hits origin
```

Next, parameters are defined for the Monte Carlo simulation and useful expressions are precomputed such as `double sqrk = sqrt(k)`. The user is asked to input the number of simulations. Variables that will be used inside the scope of the MC loop are initialized, such as std::vector<double> res, which stores the results of the MC simulates (the value that is to the averaged over), and a counter variable for the number of times the underlying stock price hits 0. If I remember correctly, it's possible for S to hit 0, but only once. It is an observing state. If we instead had the CIR model the variance would go to 0 as $S \searrow 0$ making this less likely to occur (maybe impossible).

```
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);
}

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

// Cleanup; V2 use scoped pointer
delete myNormal;
```

Above is the main loop for implementing the Monte Carlo (MC) pricing of a European Call. The outer loop, `for (long i = 1; i <= NSim; ++i)`, computes the MC price `NSIM` times. Each simulated results in accumulating the MC price divided

by a normalizing factor `NSIM` so the resulting sum at the end of the loop is the average price, `price += (tmp)/double(NSim)` .

The inner-loop is more interesting as it mimics the structure of the option. The idea is to simulate the stock price forward to period T. Because there is no early-exercise we only need to consider the payout of the option in period T.

To increment forward by one time-step we make a draw from the normal distribution to match ΔW_n in Equation (10). We substitute the integral of the continuous path of the Wiener process with a draw from the Normal distribution, which exactly captures its distribution qualities, `dW = myNormal->getNormal()` .

The we compute the new value of the underlying `VNew` based the Euler-Maruyama method, `VOld` , `VNew = VOld  + (k * drift(x[index-1], VOld)) + (sqrk * diffusion(x[index-1], VOld) * dW)` .

Finally, we set `VOld` to `VNew` for the next iteration of the for loop, and increment the counter that tracks times the underlying hit 0.

Once the inner-loop is finished we compute the payoff and increment `price` as discussed previously.

At the end of the simulation, we need to discount the price back to time t_0 to get the final price of the call, `price *= exp(-myOption.r * myOption.T)` .

```
// Cleanup; V2 use scoped pointer
delete myNormal;

std::cout << "Price, after discounting: " << price << ", " << std::endl;
std::cout << "Number of times origin is hit: " << coun << endl;
```

Lastly, we delete the object `myNormal` from memory, and display the results of the MC simulation to the user.

Question C.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). 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` ?*

Batch 1: Accuracy of European Call: $T = 0.25$, $K = 65$, $\sigma = 0.30$, $r = 0.08$, $S = 60$

Reference Call Price = 2.13337

Table values are absolute errors.

`NT` , `NSIM`	`NSIM` =10	`NIM` =100	`NSIM` = 1_000	`NSIM` =10,000	`NIM` =100_000	`NSIM` =1_000_000
`NT` = 5	2.024499e-01	6.718065e-01	1.043850e-02	8.194448e-03	3.693774e-02	3.394879e-02
`NT` = 10	5.047594e-02	2.518777e-01	5.335607e-02	1.040406e-02	2.220257e-02	1.662575e-02
`NT` = 25	8.388749e-01	4.280889e-02	1.820541e-01	6.395586e-02	3.201728e-02	3.427768e-03
`NT` = 50	1.077734e+00	1.086020e-01	8.705175e-02	6.530664e-02	2.685270e-02	1.377353e-04
`NT` = 75	6.487481e-02	2.116941e-01	5.879898e-02	1.245244e-02	5.169615e-03	1.694676e-03

NT = 100	6.434635e-01	3.844738e-02	5.927214e-02	4.429913e-03	2.944562e-03	6.577634e-04
NT = 1000	8.123001e-01	4.187805e-01	3.704392e-02	1.746732e-03	1.128285e-02	1.189853e-03

Batch 2: Accuracy of European Call: $T = 1.0$, $K = 100$, $\sigma = 0.30$, $r = 0.08$, $S = 100$

Reference Call Price = 7.96557

Table values are absolute errors.

NT, NSIM	NSIM =10	NIM =100	NSIM = 1_000	NSIM =10,000	NIM =100_000	NSIM =1_000_000
NT = 5	1.144529e-01	1.987845e+00	1.403461e-02	7.290750e-02	8.822713e-03	1.128689e-02
NT = 10	6.253628e-01	4.771328e-01	9.580447e-02	1.511568e-02	4.269981e-03	3.653877e-03
NT = 25	2.818773e+00	1.705325e-01	3.679201e-01	9.967205e-02	7.558557e-02	9.635184e-03
NT = 50	4.200774e+00	9.950853e-02	8.227195e-02	1.805510e-01	6.259688e-02	1.120600e-02
NT = 75	3.492576e-02	1.886736e-01	2.200147e-01	8.501766e-03	1.018476e-03	1.100765e-03
NT = 100	1.589411e+00	1.904438e-01	2.299962e-01	2.460466e-02	2.195357e-02	6.577634e-04
NT = 1000	2.613153e+00	9.024101e-01	9.392745e-02	4.331281e-02	1.627487e-02	6.111349e-01

Conclusion: I need something like NT=75 and NSIM=100000 to get close to the accuracy of the closed-form solution

Question C.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?*

Ok, so I ran NT=1_000 and and NSIM=100_000_000 , and the resulting pricing error was $3.23e-01$, so I think it suffice to say this batch would need an extreme number of simulations to reach $1e-02$.

```
Simulation number: 99930000
Simulation number: 99940000
Simulation number: 99950000
Simulation number: 99960000
Simulation number: 99970000
Simulation number: 99980000
Simulation number: 99990000
Simulation number: 100000000
Pricing Error: 3.230219e-01
Number of times origin is hit: 0
```

I would think based on what I observed NT = 1_000 and NSIM = 1_000_000_000 should get close $1e-02$, as $10^{-\frac{1}{2}} \approx 3.23 \times 10^{-1}$.