MATH 6767 Homework 4

Yuliang Li GTID# 903012703

Project Object

In this project, we use Monte Carlo Method to price a “down and in barrier” option on the underlying “shares of MPC”. Some parameters are like,

Initial spot: 70.86
Maturity: 181 days
Strike price: 70.86
Contingency: Down and In Barrier at 56.69, sampled daily
Risk free interest rate: 0.09%
Dividends: Dividends of 0.42% paid on Nov 18 and Feb 18 (day 18 and day 79)

The project prices the option in two market scenarios:
1. Volatility is 30.71%
2. Volatility is a function of S and t given by

$$\sigma(S,t) = (30 - 0.6(S - 70)e^{t-T})\%$$

where T-t is time to expiration.

Project Structure

Main.cpp: The main file. Complete pricing the European Vanilla option and Down and In Barrier option with Monte Carlo Method (the stochastic process generated by GBM and Euler-Marayama method; the random numbers generated by Halton numbers and some library functions). Compare the price of a European Vanilla option completed by Black-Scholes Formula and Monte Carlo Method.

BSE.h and **BSE.cpp:** Black Sholes Model to price European Option, written in the previous homework.

NormDistIntegral.h and **NormDistIntegral.cpp**: generate a norm distribution, also written in the previous homework.

Simulator.h: Simulate the stochastic processes with generating random numbers. The processes include Geometric Brownian Motion (GBM), Euler Marayama method. The random number is generated by BoxMuller method, which is completed by BoxMuller.h, BoxMuller.cpp, mt19937ar.h and mt19937ar.cpp.

MCEngine.h: Monte Carlo Engines, which price options with various payoffs by Monte Carlo Methods simulated by different stochastic processes.

OptionPay.h and **OptionPay.cpp**: calculate the payoff of vanilla option and down and in barrier option.

IO.h and **IO.cpp**: input file contains parameters for pricing and output the results to csv files.

Project Implementation

Halton numbers

Halton number is a low discrepancy series, which satisfy uniform distribution in interval [0, 1]. We can use Halton number to get the random number satisfy the standard normal distribution in interval [0, 1]. Then we get the factor for GBM exponent part. A objective function completes all this part. The code is like below,

```
1. double operator()(int N) // argument gives step size
2. {
3.     //this function's copyright belongs to Prof. Demko
4.     if(N == 1) return 1.0 / N;
5.     if(N < base) return (N % base) / (double)(base);
6.     int q = N;
7.     double power = 1.0;
8.     double return_value = 0.0;
9.     while(q > base - 1)
10.    {
11.        double r = q % base;
12.        return_value += r / pow(base, power);
13.        q = q / base;
14.        power += 1.0;
15.    }
16.     return_value += q / (double)pow(base, power);
17.     return_value = InvStdNormal(return_value); // inverse the
   Halton number to a number satisfy Normal distribution in [0, 1]
18.     return Ito*exp(return_value * variance);
19. }
```

Euler-Marayama Method

Use this class we generate a routine and store the routine in an array. Then return the array. Each routine is completed by objective function.

```
1. double* operator()(double  intecp) // the argument is the
   intecept of the volatility
2. {
3.         double x = spot; // current spot price
4.         double *price = new double [NumSteps];
5.         for(int i = 0;  i< NumSte ps; ++i){
6.                 double z = genrand_std_normal();
7.                 x *= 1 + det + var * z;
8.                 price[i] = x;
9.                 sigma = (intecp - 0.6 * (x - 70.0) * exp(i+1-
   NumSteps)) / 100.0;
10.                var = sigma * sqrt(dT);
11.        }
12.        return price;
13.}
```

Monte Carlo Engine

We use various Monte Carlo Engines to simulate the Monte Carlo Method and price the options.

We use template function to code the Monte Carlo Engine. In the template, Payoff is the data type presents the payoff of option, and Simulator is the data type presents the stochastic process. The number of variables differentiates each template.

Price the European Vanilla option whose underlying price moves like GBM. We simulate N samples and consider the average of the each sample's price as the result.

```
1. //for GBM
2. template <typename Payoff, typename Simulator>
3. double MonteCarloPricer(int N, Payoff p, Simulator
   s, double discount, double spot)
4. {
5.    if(N <= 0){ std::cerr << "\n\t illegal number of simulations
   "<< std::endl; exit(1);}
6.    double sum = 0.0;
7.    for(int i = 0; i< N; ++i){ double z = s(); sum += p(z*spot);
    }
8.    return discount*sum/N;
9. }
```

To price the Down and In Barrier option, we should simulate each day's underlying price and simulate N sample routines. The average the price of option on each routine is considered as the result. To save the memory, each price routine is stored in the dynamic array and it is deleted after that the routine has been priced.

```
1. //MC Engine for barrier option without dividends
2. template <typename Payoff, typename Simulator>
3. double MonteCarloPricer(int N, Payoff p, Simulator
   s, double discount, double spot, intmaturity)
4. {
5.         if(N <= 0){ std::cerr << "\n\t illegal number of
   simulations "<< std::endl;exit(1);}
6.         double sum = 0.0;
7.         for(int i = 0; i< N; ++i)
8.         {
9.                 double spot_temp = spot; // initial each
   rountine's initial spot price
10.                 double* price = new double[maturity]; // set a
   dynamic array to store routine price
11.                 for(int j = 1; j <= maturity; j++)
12.                 {
13.                         double z = s(); // GBM exponent part
14.                         spot_temp *= z;
15.                         price[j-1] = spot_temp;
16.                 }
17.                 sum += p(price, maturity);
18.                 delete[] price; // release the memory
19.          }
20.          return discount*sum/N; // the price of option
21. }
```

For each routine, we sort the underlying price to find whether it has been lower than the barrier. We use quick sort to complete it. This part is completed in the class BarrierPutPay.

```
1. // quick sort to order the price from lowest to highest
2. void BarrierPutPay::sort(double* price, int l, int r)
3. {
4.     if(l < r)
5.     {
6.         int i = r,j = r+1;
7.         double pivot = price[l];
8.         double temp;
9.         for(i = r; i >l; i --)
10.         {
11.             if(price[i] >= pivot)
12.             {
13.                 j--;
14.                 if(i != j)
15.                 {
16.                     temp = price[j];
17.                     price[j] = price[i];
18.                     price[i] = temp;
```

```
                }
            }
        }
        temp = price[j-1];
        price[j-1] = pivot;
        price[l] = temp;
        sort(price, l, j-1);
        sort(price, j+1, r);
    }
}
```

Payoff of the Down and In Barrier Option

The class of this payoff is a derived class of class Payoff written before. In this class, we use a function objective, which receives an array of stock price to get the option price.

```
double BarrierPutPay::operator() (double* price, int size) // arguments are spot price and teh array's size
{
        double temp = price[size-1];
        sort(price, 0, size-1);
        if(price[0] <= barrier) // if down and in the barrier, the option take effect
        {
                return (strike - temp > 0.0)?(strike - temp):0.0;
        }
        else
         {
                 return 0.0;
        }
}
```

Results

In this project, the number of test Monte Carlo method is 32000.

1. The price of European Vanilla Put option under Market Scenario I without dividends.

Methods	Black-Scholes Formula	Monte Carlo Methods (system random number generate function)	Monte Carlo Methods (Halton numbers)
Option Price	6.0844	6.0483	6.0845

2. The convergence of Monte Carlo Methods

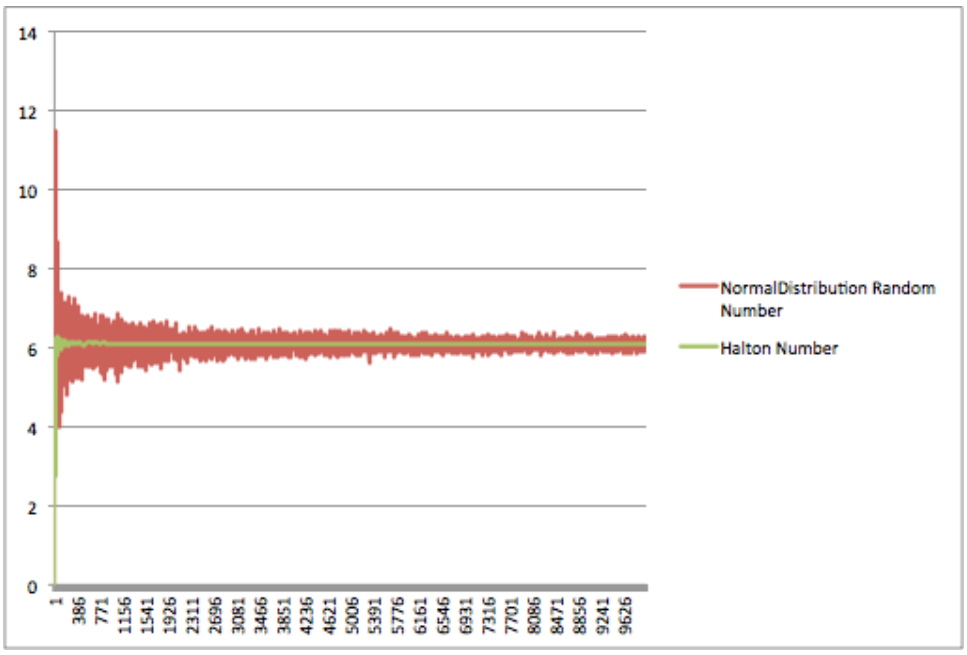

Fig. 1 The convergence of two methods for generating random numbers to simulate Monte Carlo Method

Concluded from the figure, the speed of convergence of using Halton number is much better than using the normal distribution random number which is generated by the function in library.

3. The Down and In Barrier option price under Market Scenario I

	No Dividends	Dividends
Option Price	4.6860	4.9883

Because dividend means some certain revenue, the price of option with dividends should be higher than those ignoring dividends. The result also reflects the fact.

4. The Down and In Barrier option price under Market Scenario II ignoring dividends

	Market Scenario I	Market Scenario II
Option Price	4.6860	4.6883

Because the volatility under Market Scenario II is unstable, the volatility has probability that to be greater than 30.71%. Thus, the price of the barrier option under Market Scenario II is higher than that under Market Scenario II.

5. Confidence level of the option price with dividends under Market Scenario I

To test our result statistically, we test the confident interval of the barrier option price with dividends. The 95% confident interval of the option price that is calculated by Monte Carlo Method is [4.8973, 5.0792]. For there are 32000 samples, but the interval is narrow enough, which means the standard deviation of our sample is small, so we are able to consider our result is good.

6. About JP Morgan's security

Under the market described above, the underlying price has less probability to cross the barrier, and that leads the barrier option price is much lower than European Vanilla option price.

However, if the volatility is no longer stable, and the volatility of it is large, the unstable market will contribute to raising the option's price, which has been reflected in the 4^{th} question.