Student ID:10423513

March 15, 2020

1 Introduction

In this project I will employ Monte Carlo method to price different portfolios/options. For pricing derivatives who have analytic solutions, it is expected to compare results produced by Monte Carlo and exact values. Some improvements on Monte Carlo method will also be applied, such as **antithetic variables** and **moment matching**. After practicing Monte Carlo methods with solved portfolios/options, I will then use it to price some path-dependent options, including Asian options. Here we assume that all stock prices follow geometric Brownian motions with specified parameters given in the tasks. For path dependent options, I will explore the most accurate result within 10 seconds by using different methods.

2 Stock options

In this task, we aim to price a portfolio that

$$\Pi = 2P(X_1;T) - 2C(X_2;T) + X_2 BC(X_2;T)$$

where $P(X_1;T)$ is a European put option with strike price X_1, $C(X_2;T)$ is a European call option with strike X_2, BC is a binary call option with strike X_2 and all three expire at time T. The parameters are given as $T = 1.75, \sigma = 0.15, r = 0.04, D_0 = 0.02, X_1 = 6500 \textit{ and } X_2 = 9500$.

2.1 Approximated values VS analytic values

We know by the Law of Large Numbers that as the number of times N changes, the Monte Carlo (MC) simulation will produce different results. So here we want to investigate into how prices of the portfolio differ with various N. Here the initial time is set to $t = 0$ and initial stock price $S_0 = 6500$ and $S_0 = 9500$.

From the Figure(1) and Figure(2) in the next page, we can observe that estimated values produced by Monte Carlo converge to analytic solutions as N increases from 1000 to 100000 and 300000 respectively. And MC estimations cross the horizontal line many times which means for some cases we can obtain same values as exact solutions by MC method. For large N, estimated values are close to exact values which also verify our Monte Carlo algorithm.

•When $S_0 = 6500$, the result produced by MC method at $N = 100000$ is $V(\Pi;T) = 1021.38$.

•As for stock price $S_0 = 9500$, the result produced by MC method at $N = 300000$ is $V(\Pi;T) = 2959.10$

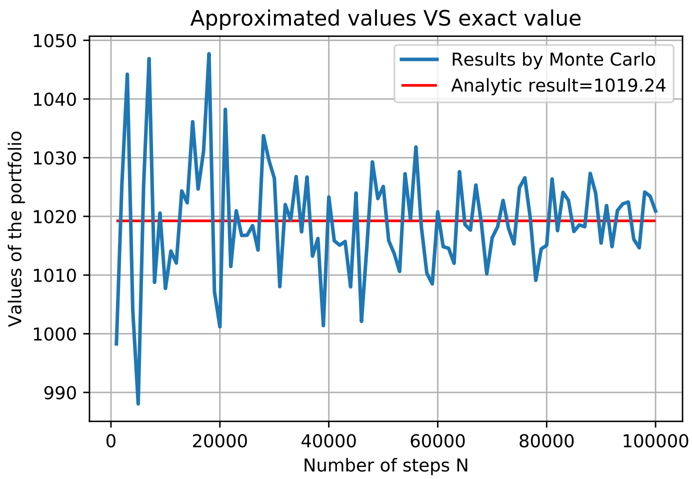

Figure 1: $S_0 = 6500$

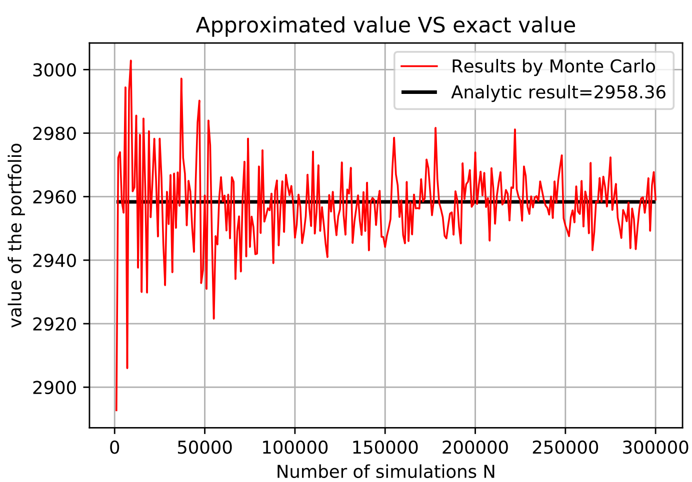

Figure 2: $S_0 = 9500$

2.2 Confidence Intervals

As we have shown above that prices of the portfolio converge when N increases, therefore by the Central Limit Theorem, we can know that

$$\frac{\sum_{i=1}^{n} V_i}{n} \sim N(\mu, \frac{\sigma^2}{n})$$

where μ is the true mean and $sigma^2$ the variance of V_i. Here we can use sample mean and sample variance to give

$$\frac{\sum_{i=1}^{n} V_i}{n} \sim N(\mu^*, \frac{\sigma^{*2}}{n})$$

. Thus we can draw a confidence interval with sample mean and variance we obtained by MC method. In this part, I will also deploy some improvements on original Monte Carlo methods such as **Antithetic variables** and **Moment matching**. These extensions will be brought up together accompanying comparison with ordinary Monte Carlo so I do not mention about extensions individually.
Here I show 95% confidence intervals obtained for option values at $S_0 = X_1(6500)$, by using three Monte Carlo methods, with 100 samples. And I will also plot the variances of samples at various N values with different methods.

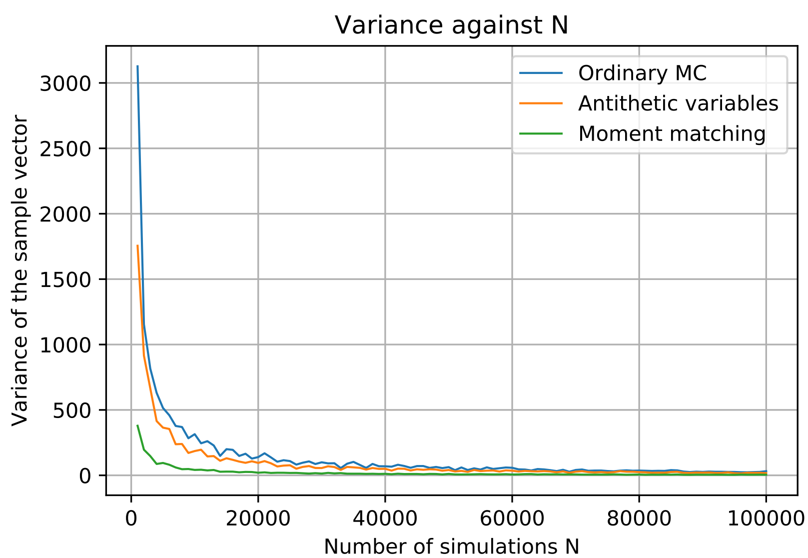

Figure 3: Sample variance plots of three methods

Number of simulations	Ordinary MC	Antithetic variables	Moment matching
N=1000	[1005.52,1027.89]	[1005.32,1022.08]	[1015.61,1023.4]
N=5000	[1016.47,1025.98]	[1015.37,1022.81]	[1017.12,1020.67]
N=10000	[1015.49,1021.87]	[1015.73,1020.81]	[1018.17,1020.56]
N=30000	[1016.98,1020.56]	[1016.39,1019.4]	[1018.39,1019.86]
N=50000	[1017.74,1020.67]	[1017.49,1019.9]	[1018.8,1020.15]
N=100000	[1018.22,1020.3]	[1018.14,1019.8]	[1018.7,1019.6]

Table 1: 95% confidence intervals by using different methods

•According to the Table(1) above, when using ordinary MC method to obtain samples, as N goes up, the confidence interval contracts. As we have already known the analytic value $V = 1019.24$, so for a large N the confidence interval approaches this analytic result, this is expected. If we look at Figure(3), we can find that when N soars, **the sample variance decreases so as its standard deviation, this causes the range of confidence interval becomes smaller**. The most accurate 95% confidence interval (at $N = 100000$) is [1018.22,1020.30]

• When applying Antithetic variables for improving MC method, we can observe that sample variance from Figure(3) is much smaller than ordinary one, it is almost only half of that using ordinary MC method. And 95% confidence interval also narrows when N jumps. From Table(1), Antithetic variables method appears more accurate than ordinary MC, mainly because of low variance. When deploying antithetic variables method at N simulations, we just need to make $N/2$ random draws from standard normal distribution, it results in less computation budget with the same accuracy as N simulations can do in a normal case. Also, the mean of random draws is zero so sample paths are correct. **The most accurate 95% confidence interval (at $N = 100000$) is [1018.14,1019.80].**

• Moment matching is an extension on antithetic variables. Based on applying antithetic random draws, moment matching makes these random draws to be exactly standard normal distributed, i.e zero mean and unit variance. Because when we take draws from standard normal distribution using built-in functions, it does not guarantee these random numbers we obtain are exactly standard normally distributed. Thus we calculate the variance (v) and replace all of random draws (ϕ) with $\frac{\phi}{\sqrt{v}}$ then the variance of new random draws is 1. **We can see a huge progress in reducing variance from Figure(3), where variance starts at a much lower position and converges to 0 quickly**. And the 95% confidence intervals are also smaller than that of previous methods at same N level. **At $N = 100000$, it produces the most accurate confidence interval [1018.7,1019.6], better than ordinary MC and antithetic variables method.**

3 Path Dependent Options

3.1 Ordinary Monte Carlo Case

For this task we are supposed to price a float-strike Asian call option with parameters $T = 1$, $r = 0.06$, $D_0 = 0.01$, $\sigma = 0.35$, $S_0 = 4150$, $K = 35$. K is equally spaced throughout the lifetime of of the option, where $t_0 = 0$ and $t_K = T$. Now we would like to investigate how values of this float-strike Asian call option vary with different N and K.

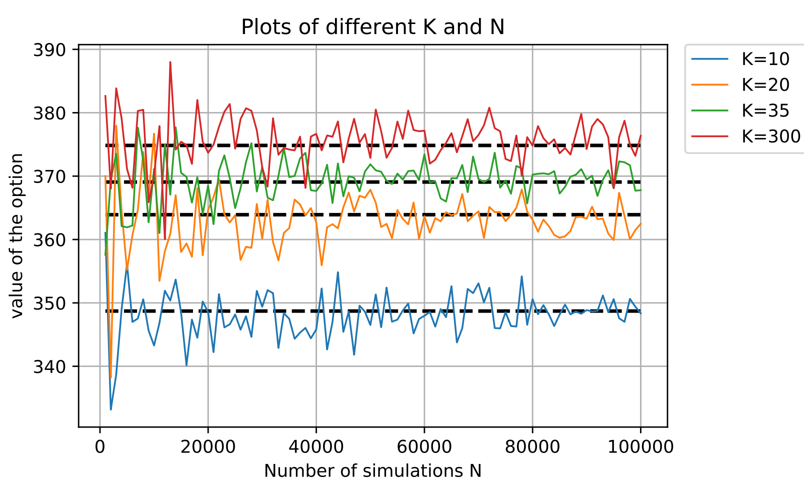

Figure 4: How value changes with various K and N

For a better comparison, I include 4 plots of $K = 10, 20, 25, 300$ in one figure, all starting from 1000 simulations to 100000 simulations. Black dashed lines represent the value of the option at $N = 100000$ for 4 different K values. I use it here to make it clear that **all 4 graphs are converging as N increases**.

•**It is easy to find out values of the path dependent call go up as K increases**. This is because when we model the stock price from Brownian motion random walk, the averaged strike $\frac{\sum_{i=1}^{K} S_i}{K}$ is likely to become smaller when K enlarges. Thus $max(S - A, 0)$ becomes larger in average, this leads to higher values of the option when we apply Monte Carlo method.

Number of simulations	K=10	K=20	K=35	K=300	K=1000
N=100000	348.712	363.914	369.569	374.842	379.487

Table 2: Estimated value at N=100000

•When $K = 10$, the estimated value of the option is $V = 348.712$.
•When $K = 20$, the estimated value of the option is $V = 363.914$.
•When $K = 35$, the estimated value of the option is $V = 369.569$.
•When $K = 300$, the estimated value of the option is $V = 374.842$.
•When $K = 1000$, the estimated value of the option is $V = 379.487$.
From Table(2) and Figure(4) we can also spot that the magnitude of increase of option values is not linear. When K is large enough, the increase will become plain.

3.2 Improvement of Monte Carlo

In this part I will apply another method to evaluate the most possible accurate value for the float-strike Asian call above with $K = 35$ within 10 seconds computation time.

Antithetic Variables

When deploying antithetic variables on MC method, it will save much computation without decreasing accuracy, which means we can have much bigger N values than using ordinary MC within 10 seconds. To apply antithetic variables on path dependent options, we choose positive random draws for all $t_i, \quad i = 0, 1, \ldots, K$ to produce one averaged strike. Then take their negative draws for another lifetime $t_i, \quad i = 0, 1, \ldots, K$ to make another strike. This is slightly different from that in ordinary MC method.

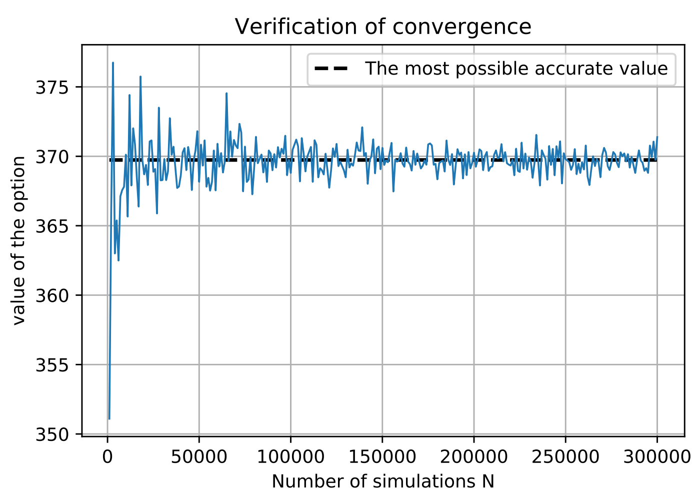

Figure 5: Verification of convergence

•The most accurate value of the option I can obtain within 10 seconds by using antithetic variables is **V=369.735**. This is achieved at $N = 3000000$, consuming about 8 seconds. And when plot out the graph, we can check from Figure(5) that this value is reasonable, since values seem to converge to it.

Moment Matching The next method I will apply is moment matching, which serves as an extension on antithetic variables method.

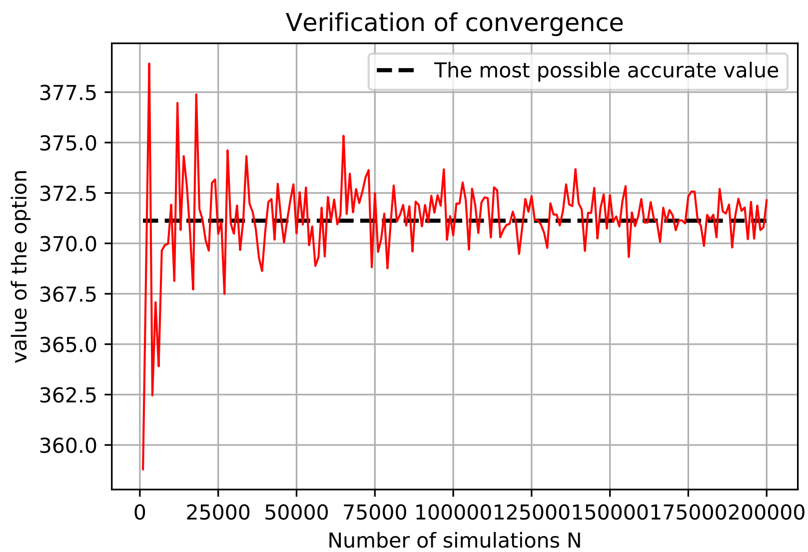

Figure 6: Verification of convergence

•The most accurate value by using moment matching method is V=371.126, which is achieved by about 7 seconds at $N = 2000000$. According to Figure(6) we find it reasonable as the graph actually converges to the value close to the dashed line.

By comparing to ordinary MC method,

categories	Ordinary MC	Antithetic variables	Moment matching
Computation time	10s	8s	7s
The most accurate value	V=369.652	V=369.735	V=371.126

Table 3: Comparison among three methods

.1 Source.cpp

```cpp
#include <iostream>
#include <random>
#include <cmath>
#include <vector>
#include <algorithm>
#include<iomanip>
#include<fstream>
#include <cassert>
using namespace std;
class Payoff
{
public:
    virtual double operator()(double ST) = 0;
    virtual double operator()(double ST, double A) = 0;
};
class portfolio :public Payoff
{
public:
    portfolio(vector<double>X_) :X(X_) {};
    virtual double operator()(double ST)
    {
        double bc = (ST > X[1]) ? 1 : 0;
        return 2. * max(X[0] - ST, 0.) - 2. * max(ST - X[1], 0.) + X[1] * bc;
    }
    virtual double operator()(double ST, double A) { return 0; }
private:
    vector<double> X;
};
class Asiancall :public Payoff
{
public:
    virtual double operator()(double ST) { return ST; }
    virtual double operator()(double ST, double A) { return max(ST - A, 0.); }
};

class pricing
{
public:
    pricing(double T_, double sigma_, double r_, double D0_, double S0_,double t_,vector<double> X_) :
        T(T_), sigma(sigma_), r(r_), D0(D0_),S0(S0_),t(t_),X(X_) {}; // constructor
    double MC(Payoff&p,int N)
    {
        static mt19937 rnd;
        normal_distribution<> ND(0.0, 1.0);
        double sum = 0;
        for (int i = 0; i < N; i++)
        {
            double Z = ND(rnd);
            double ST= S0 * exp( (T - t) * (r - D0 - 0.5 * sigma * sigma) + sigma * sqrt(T - t) * Z );
            sum += p(ST);
        }
```

```
        return exp(-r * (T - t)) * sum / N;
    }

    double AMC(Payoff& p, int N)
    {
        static mt19937 rnd;
        normal_distribution<> ND(0.0, 1.0);
        double sum = 0;
        for (int i = 0; i < N; i++)
        {
            double Z = ND(rnd);
            double ST_p = S0 * exp((T - t) * (r - D0 - 0.5 * sigma * sigma) + sigma * sqrt(T - t) * Z);
            sum += p(ST_p);
            double ST_m= S0 * exp((T - t) * (r - D0 - 0.5 * sigma * sigma) - sigma * sqrt(T - t) * Z);
            sum += p(ST_m);
        }
        return (exp(-r * (T - t)) * sum )/ (2. * N);
    }

    double MM(Payoff& p,int N)
    {
        static mt19937 ran;
        normal_distribution<> ND(0.0, 1.0);
        double sum = 0;
        vector<double> path;
        for (int i = 0; i < N; i++)
        {
            double Z = ND(ran);
            path.push_back(Z);
            sum += Z * Z;
        }
        double sd = sqrt(sum / N);
        double sum1 = 0;
        for (int i = 0; i < N; i++)
        {
            double phi = path[i]/sd;
            double ST_p = S0 * exp((T - t) * (r - D0 - 0.5 * sigma * sigma) + sigma * sqrt(T - t) * phi);
            sum1 += p(ST_p);
            double ST_m = S0 * exp((T - t) * (r - D0 - 0.5 * sigma * sigma) - sigma * sqrt(T - t) * phi);
            sum1 += p(ST_m);
        }
        return (exp(-r * (T - t)) * sum1) / (2. * N);
    }
    double PD(Payoff&p,int K, int N)
    {
        static mt19937 rnd;
        normal_distribution<> ND(0., 1.);
        double dt = (T-t) / K;
        double sum1 = 0;
        for (int i = 0; i < N; i++)
        {
            vector<double> v(K + 1);
```

```
            double sum = 0;
            v[0] = S0;
            for (int i = 1; i <= K; i++)
            {
                double Z = ND(rnd);
                v[i] = v[i - 1] * exp((dt) * (r - D0 - 0.5 * sigma * sigma) + sigma * sqrt(dt) * Z);
                sum += v[i];
            }
            double A = sum / K;
            sum1 += p(v[K],A);
        }
        return exp(-r * (T - t)) * sum1 / N;
    }

    double APD(Payoff& p, int K, int N)
    {
        static mt19937 rnd;
        normal_distribution<> ND(0., 1.);
        double dt = (T - t) / K;
        double sum1 = 0;
        for (int i = 0; i < N; i++)
        {
            vector<vector<double>> v(2);
            vector<double> q(K + 1);
            v[0] = q;v[1] = q;
            double sum = 0;
            double summ = 0;
            v[0][0] = S0; v[1][0] = S0;
            for (int i = 1; i <= K; i++)
            {
                double Z =ND(rnd);
                v[0][i] = v[0][i - 1] * exp((dt) * (r - D0 - 0.5 * sigma * sigma) +sigma * sqrt(dt) * Z);
                v[1][i] = v[1][i - 1] * exp((dt) * (r - D0 - 0.5 * sigma * sigma) - sigma * sqrt(dt) * Z);
                sum += v[0][i];
                summ += v[1][i];
            }
            double A = sum / K;
            double A1 = summ / K;
            sum1 += p(v[0][K], A)+p(v[1][K],A1);
        }
        return exp(-r * (T - t)) * sum1 / (2.*N);
    }

    double MPD(Payoff& p, int K, int N)
    {
        static mt19937 rnd;
        normal_distribution<> ND(0., 1.);
        double dt = (T - t) / K;
        double sum1 = 0;
        for (int i = 0; i < N; i++)
        {
            vector<double> path;
            vector<vector<double>> v(2);
```

```
            vector<double> q(K + 1);
            v[0] = q; v[1] = q;
            double sum = 0;
            double summ = 0;
            double ss = 0;
            v[0][0] = S0; v[1][0] = S0;
            for (int i = 0; i < K; i++)
            {
                double Z = ND(rnd);
                path.push_back(Z);
                ss += Z * Z;
            }
            double sd = sqrt(ss / K);
            for (int i = 1; i <= K; i++)
            {
                double phi = path[i - 1] / sd;
                v[0][i] = v[0][i - 1] * exp((dt) * (r - D0 - 0.5 * sigma *
    sigma) + sigma * sqrt(dt) * phi);
                v[1][i] = v[1][i - 1] * exp((dt) * (r - D0 - 0.5 * sigma *
    sigma) - sigma * sqrt(dt) * phi);
                sum += v[0][i];
                summ += v[1][i];
            }
            double A = sum / K;
            double A1 = summ / K;
            sum1 += p(v[0][K], A) + p(v[1][K], A1);
        }
        return exp(-r * (T - t)) * sum1 / (2. * N);
    }

    double analytic()
    {
        double d1 = ( log(S0 / X[0]) + (r - D0 + 0.5 * sigma * sigma)* (T - t)
     ) /  (sigma * sqrt(T - t));
        double d2 = d1 - sigma * sqrt(T - t);
        double D1= ( log(S0 / X[1]) + (r - D0 + 0.5 * sigma * sigma) * (T - t)
     ) / (sigma * sqrt(T - t));
        double D2 = D1 - sigma * sqrt(T - t);
        double P = 2. *( X[0] * exp(-r * (T - t)) * 0.5 * erfc( d2 / sqrt(2))
    - S0 * exp(-D0 * (T - t)) * 0.5 * erfc(d1 / sqrt(2)) );
        double C = -2. * (S0 * exp(-D0 * (T - t)) * 0.5 * erfc(-D1 / sqrt(2))
    - X[1] * exp(-r * (T - t)) * 0.5 * erfc(-D2 / sqrt(2)));
        double BC = X[1]*exp(-r * (T - t)) * 0.5 * erfc(-D2 / sqrt(2));
        return P + C + BC;
    }

    void CI(Payoff&p,int N,int M,int method)
    {
        double s = 0;
        double var = 0;
        vector<double> v(M);
        for (int m = 0; m < M; m++)
        {
            if (method==0){ v[m] = MC(p, N);}
            if (method==1){ v[m] = AMC(p, N);}
            if (method == 2) { v[m] = MM(p, N); }
```

```
            if (method == 3) { v[m] = PD(p, 35, N); }
            if (method == 4) { v[m] = APD(p, 35, N); }
            if (method == 5) { v[m] = MPD(p, 35, N); }
            s += v[m];
        }
        double mean = s / M;
        for (int i = 0; i < M; i++)
        {
            var += pow(v[i] - mean, 2);
        }
        double svar = var / (M - 1.);
        cout << "mean:" << mean << endl;
        cout << "variance:" << svar << endl;
        double sd = sqrt(svar / M);
        cout << "95% confidence interval: [" << mean - 2. * sd << "," << mean + 2. * sd << "]";
    }

private:
    double T, sigma, r, D0, S0, t;
    vector<double> X;
};

int main()
{
    pricing a(1.75, 0.15, 0.04, 0.02, 9500., 0., { 6500.,9500. });
    portfolio p({ 6500.,9500. });
    /*
    ofstream out("porn.csv");
    for (int i = 1; i <= 300; i++)
    {
        int N = 1000 * i;
        out << a.MC(p, N) << endl;
    }
    */
    //pricing a(1.75, 0.15, 0.04, 0.02,6500.,0., { 6500.,9500. });
    //portfolio p({ 6500.,9500. });
    cout << a.analytic();
    //pricing a1(1., 0.35, 0.06, 0.01, 4150., 0., { 0.,0. });
    //Asiancall ac;
    //cout << a1.MPD(ac, 35, 2500000);
    /*
    ofstream out("fig2.csv");
    for (int i = 1; i <= 200; i++)
    {
        int N = 1000 * i;
        out << a1.MPD(ac, 35, N) << endl;
    }
    */
    //a1.CI(ac, 10000, 100, 5);
    //cout << a1.MPD(ac,35, 10000);
    return 0;
}
```