

## Introduction

This report is a two part study of:

- I) Montecarlo techniques used to value a portfolio of european, path independent options. Extensions of this method such as antithetic variables and moment-matching were also investigated vis-a-vis confidence intervals and variance reduction.
- II) Montecarlo techniques (and extensions) used to value a discrete, fixed-strike Asian call option which is an example of a path dependent option. Particular study is also carried out on computational processing time required and algorithm efficiency.

## Task 1

This task consisted in valuing a portfolio comprising of shorting a call option with strike price  $X_1$ , longing a call option with strike price  $X_2$ , longing  $2X_2$  binary cash or nothing call options with strike price  $X_2$  and unit payoff and longing a call option with strike price equal to zero with parameters:

$T=0.5$ ,  $\sigma=0.41$ ,  $r=0.03$ ,  $D_0=0.04$ ,  $X_1=70.0$  and  $X_2=100.0$ .

Montecarlo methods of estimation have at their core the law of large numbers which states that for a number  $N$  of trials tending to infinity, the average of all the observations tends to the expected value. Formally,

**Theorem 1** *The sample average  $\bar{X}$  converges almost surely to the expected value  $\mu$ .  $\bar{X}_n \rightarrow \mu$  for  $n \rightarrow \infty$*

Hence, Montecarlo methods are a technique in which a large quantity of randomly generated numbers (usually denoted by  $N$ ) are studied using a probabilistic model to find an approximate solution to a numerical problem that would be difficult to solve by other methods.

## Analytic Values

Before attempting numerical methods of solving the valuation of the portfolio it is important we have a value to check our results with. Luckily there are analytic solutions to the different options in the portfolio currently being analysed and these combine to produce our final value of  $V(S_0, t = 0)$ . The following represent the analytic formulae used:

- I) A call option with value of underlying  $S$  at time  $t$ , strike price  $X$  at maturity  $T$  and interest rate  $r$ :

$$C(S, t) = Se^{-D_0(T-t)}N(d_1) - Xe^{-r(T-t)}N(d_2) \quad (1)$$

- II) A binary call option with unit payoff at maturity  $T$ :

$$BC(S, t) = e^{-r(T-t)}N(d_2) \quad (2)$$

The values  $d_1$  and  $d_2$  are variables calculated from the other parameters with formulae:

$$d_1 = \frac{\ln(\frac{S}{X}) + (r - D_0 + \frac{1}{2}\sigma^2)}{\sigma\sqrt{T-t}} \quad (3)$$

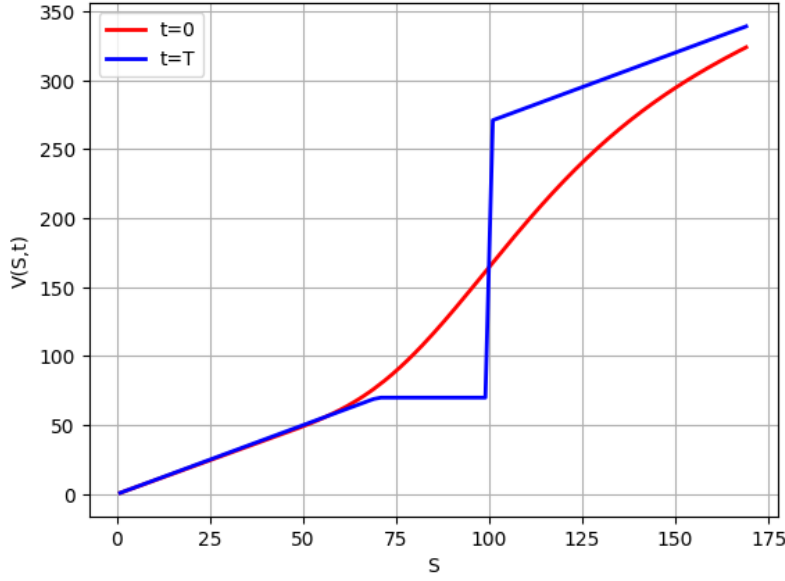


Figure 1: Plots of the price of the portfolio at times  $t = 0$  and  $t = T$  using analytic solutions.

and

$$d_2 = d_1 - \sigma\sqrt{T - t} \quad (4)$$

Figure 1 illustrates the value of the portfolio calculated using the analytic Formulae 1 to 4. We are particularly interested in the  $t = 0$  case since we will be checking the estimators used later on with these values to measure accuracy.

## Analysing Montecarlo Methods

The first and most basic method of estimation attempted in this report was the 'normal' Montecarlo method. For a particular path  $m$  (where  $m$  runs from 1 to  $M$ ), a random number is generated from a pseudorandom number generator (PRNG) of choice  $N$  times. In this case, a Mersenne Twister was chosen which is based on the Mersenne prime  $2^{19937} - 1$ . This particular PRNG was chosen as it has very long periods, passes most statistical tests for randomness and has fast random number generation. The price of the stock is let to fluctuate and the average value after  $N$  fluctuations is taken. From Theorem 1 we expect the value of the portfolio to tend to the analytic value as  $N$  tends to a very large number. The results for this method are shown in Table 1. The Montecarlo estimate value for the portfolio does

$N$ (thousands)	$V(S_0 = X_1, t = 0)_{MC}$	$V(S_0 = X_1, t = 0)_A$	$V(S_0 = X_2, t = 0)_{MC}$	$V(S_0 = X_2, t = 0)_A$
1	77.8243	78.1939	164.73	164.564
2	78.3079	78.1939	164.662	164.564
5	78.1757	78.1939	164.694	164.564
10	78.1372	78.1939	164.546	164.564
50	78.2078	78.1939	164.615	164.564
100	78.1909	78.1939	164.615	164.564

Table 1: A table showing Montecarlo (MC) estimated values for the portfolio  $V(S_0, t = 0)$  compared to the analytic value (A) for different values of Montecarlo iterations  $N$ .

indeed tend to analytic value which is a good sign that this estimator works. However careful attention must be paid to what confidence one can put on such method and its results. A few good metrics for this are the variance (or sample variance) and confidence intervals. In the next subsection we analyze more in detail the confidence in the results of this methods and investigate possible extensions to this which make thus confidence better.

## Accuracy and Confidence Intervals

A corollary of Theorem 1 is that, assuming finite variance

$$Var(X_i) = \sigma^2 = \frac{\sum X - \mu^2}{M} \quad (5)$$

for all  $i$  and no correlation between random variables, the variance for the average value of  $M$  paths of our Montecarlo sampling is given by

$$Var(\bar{X}_M) = \frac{\sigma^2}{M} \quad (6)$$

but since we are sampling a finite number of paths and not taking an infinite number of paths we use the sample variance given by

$$Var(\bar{X}_M) = \frac{\sigma^2}{M-1}. \quad (7)$$

Finally, we can use the Central Limit Theorem which states that when independent random variables are added, their properly normalized sum tends toward a normal distribution. For us it means we can relate the sample variance to the standard deviation which implies about 95% confidence that repeated runs of the trial would result within 2 standard deviations of the mean. Applying these to the basic Montecarlo method explained previously produces the plot seen in Figure 2d. The red envelope is the 95% confidence level and one immediate observation is that this covers completely the results of the simulation. This mean that the variance is very high and our results are far from reliable.

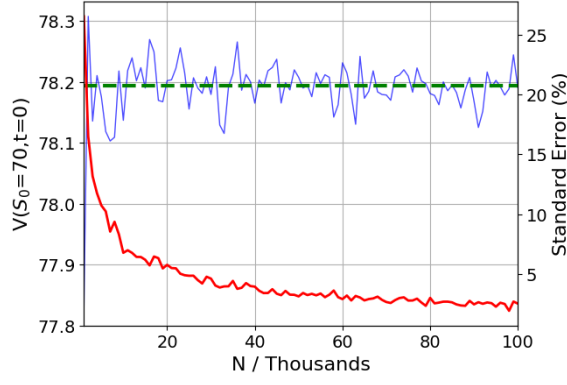
Thus, there is a need for variance reducing methods which build upon Montecarlo. One such method is the use of **antithetic variables**. The idea is that instead of drawing only a normally distributed  $\phi$  we now draw a pair  $-\phi$  and add it to the sum and at the end divide by  $2N$ . The advantage of using this method is that the sum is ensured to be 0 as required and that variance be 1 ( by matching the mean and skewness to the required one,the variance is thus ensured). To keep everything comparable  $\frac{N}{2}$  such numbers were drawn so the total remains  $N$  as before. The results can be seen in Figure 2e.

One further method investigated was **moment matching**. Similarly to antithetic variables,  $\frac{N}{2}$  random numbers are sampled together with their pair. The total variance of these is calculated and then the whole set of numbers is divided element-wise by the square root of the variance. This ensures a variance of 1 as required. The results can be seen in Figure 2f.

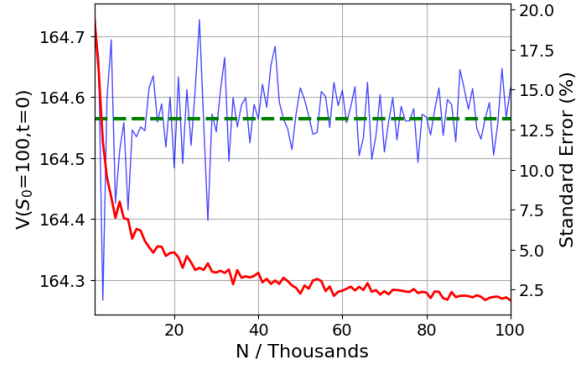
$N$ (thousands)	$V_{MC}$	$V_{AV}$	$V_{MM}$	$V_A$
1	77.8243±20.637	78.163±15.9292	78.2931±10.1584	78.1939
2	78.3079±12.9237	78.1904±9.58466	78.26±6.21086	78.1939
5	78.1757±8.3882	78.2445±7.62567	78.2246±4.64797	78.1939
10	78.1372±5.30014	78.1608±5.31964	78.2444±2.63015	78.1939
50	78.2078±2.45959	78.1939±2.62684	78.1851±1.57154	78.1939
100	78.1909±1.98767	78.1924±1.40195	78.2065±0.909534	78.1939

Table 2: A table showing basic Montecarlo (MC) estimated values for the portfolio  $V(S_0 = X_1, t = 0)$  compared to the antithetic variables (AV), moment matching (MM) and analytic (A) methods for different values of Montecarlo iterations  $N$ .

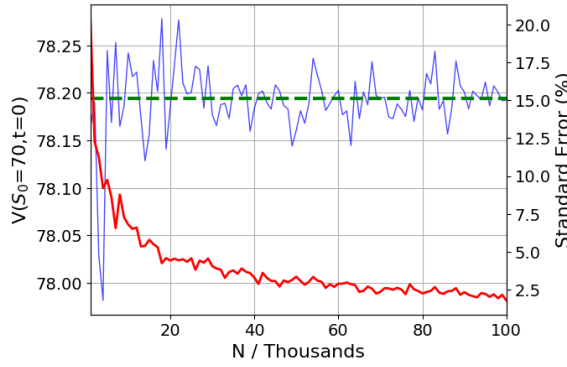
The results, shown in Figure 2 and compared in the Table 3, indicate a clear trend of increasing confidence and accuracy for the methods described. Moment matching appears to be the best of these methods, precision-wise, achieving sub 2% error on the mean. Ensuring the mean and variance match the required ones from our assumptions of Gaussian noise means the results not only converge better toward the analytic value but the precision they carry with them increases.



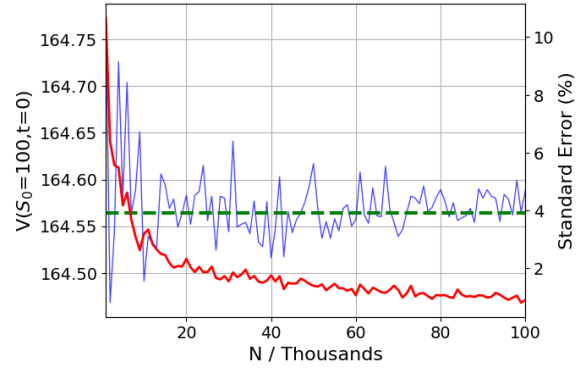
(a) Basic Monte Carlo method with  $S_0=70$



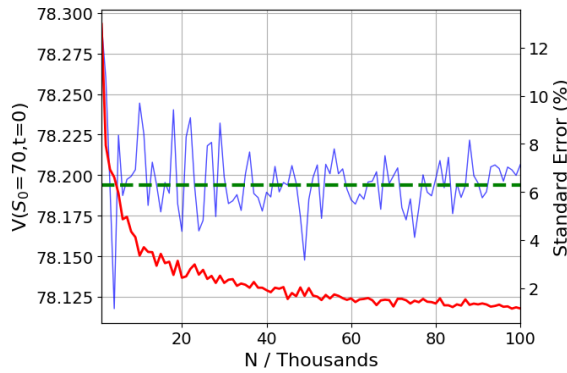
(d) Basic Monte Carlo method with  $S_0=100$



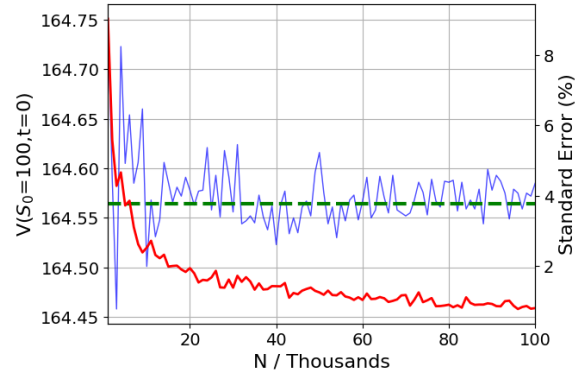
(b) Antithetic Variables method with  $S_0=70$



(e) Antithetic Variables method with  $S_0=100$



(c) Moment Matching method with  $S_0=70$



(f) Moment Matching method with  $S_0=100$

— Monte Carlo Average      — Analytic Value      — (Standard Error)

Figure 2: Plots of the price of the portfolio at time  $t = 0$  for  $S_0 = X_1$  (a-c) and  $S_0 = X_2$  (d-f) using normal (a,d), antithetic variable (b,e) and moment matching (c,f) methods respectively.

## Computational Efficiency

One final remark should be made about the efficiency in processing time required by the three estimation methods described previously. As described before, moment matching and antithetic variables require  $\frac{N}{2}$  random numbers while basic Montecarlo requires  $N$  which explains the shapes of the curves in Figure 3. The difference between moment matching and antithetic variables might be explained by the requirement of the former of one more for loop than the latter. Due to the results in Figure 2 one can say that using moment matching is the best method since the relatively small increase in computing time required produces very precise and accurate results when compared to the analytic values.

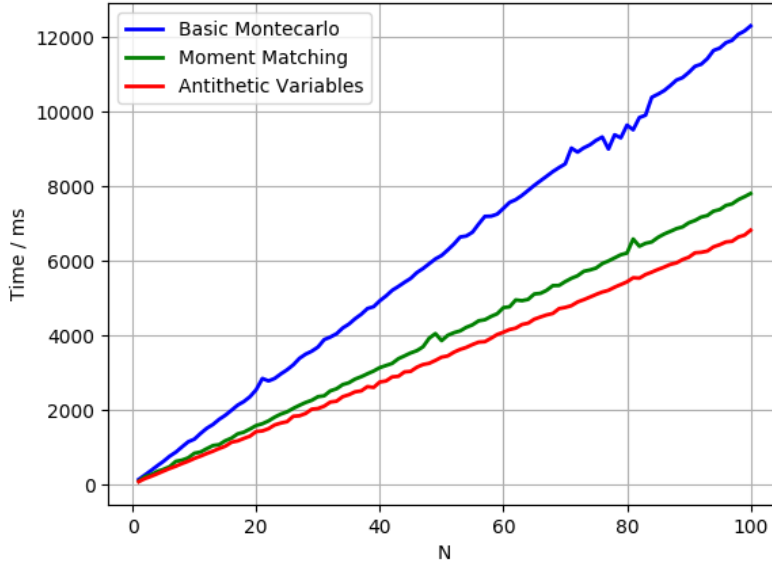


Figure 3: Plots of the time taken by the particular estimation method for  $M = 100$  paths as a function of Montecarlo random numbers generated  $N$ .

## Task 2

This task consisted in valuing a discrete fixed-strike Asian call option with the following parameters:

$$T=2, \sigma=0.42, r=0.03, D_0=0.02, X=64000$$

Due to the results shown in the previous task (Figure 2), I decided to use antithetic paths to estimate this option thus using an efficient Montecarlo extension. As before the method is the same when it comes to generating  $N$  random numbers and fluctuating the stock using this parameter. The difference in these options as the name implies is that since they are functions of the average of the price of the stock at some predetermined time intervals, the path the stock takes changes the final result. Thus the method is to subdivide time from 0 to maturity in  $K+1$  time steps and for a discrete fixed-strike Asian call option the 'floating' strike price is the average of sampling at  $K+1$  intervals. The natural first trend to investigate is the correlation (or lack) between the parameters  $K$ ,  $N$  and the value of the option.

$N$ (thousands)	$K$	$V_{MC}$	95 % Confidence Levels	
			Lower Level	Upper Level
1	20	8838.6	8712.99	8964.21
50	20	8864.3	8847.72	8880.88
100	20	8863.08	8849.54	8876.63
1	50	8734.19	8624.81	8843.56
50	50	8671.95	8656.07	8687.84
100	50	8670.35	8659.22	8681.49

Table 3: Table showing antithetic paths extended Montecarlo (MC) estimated values for the option. The number of paths is kept constant to  $M=50$ .

The table highlights the most important trends, further pictorially illustrated by Figures 4 and Y. **An increase in  $K$  appears to decrease the value of the option.** This can be interpreted by remembering that  $K + 1$  is the number of intervals into which the time between 0 and maturity is set and by which the 'floating' strike value is calculated. By taking more samples the option is more susceptible to volatility within those intervals. Having more intervals means betting on a higher set of possible intervals which make up the final price against which the strike price is tested. Thus, with higher volatility susceptibility come lower prices to account for it.

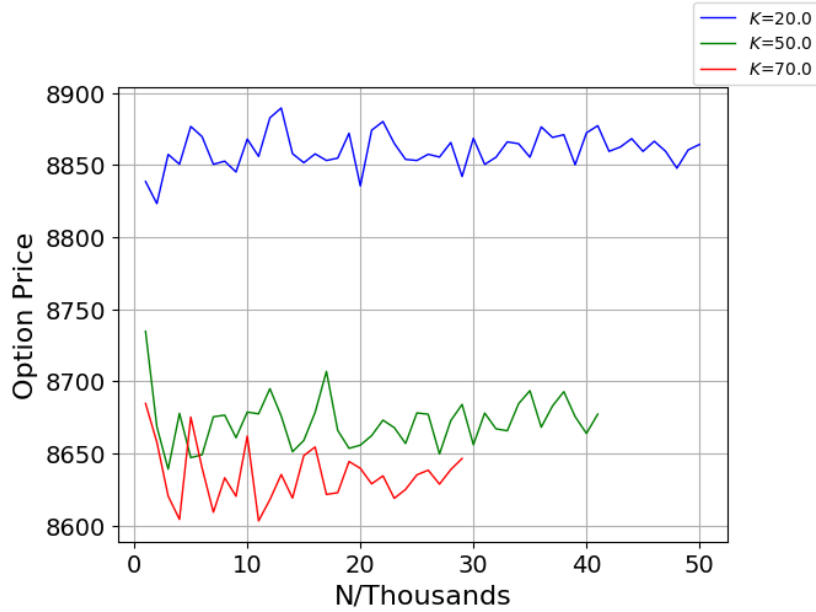
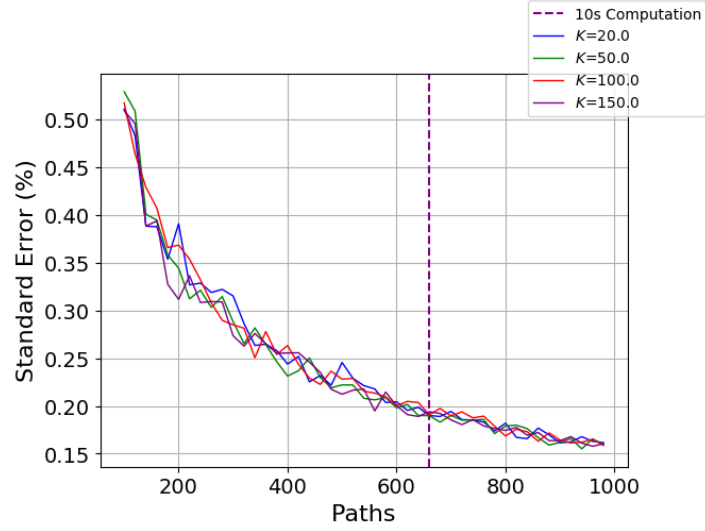


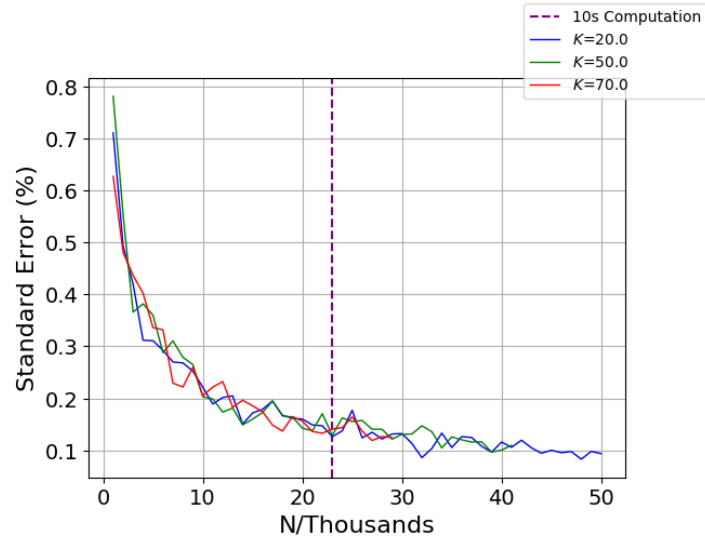
Figure 4: Value of the option as a function of increasing  $N$  for different values of  $K$ . The number of paths is fixed to  $M=100$ . Larger  $K$  values were stopped earlier when the simulation took  $> 30$ s since the point was to observe trends in option value not time.

## Strict Timing

The final part of this task was to obtain the most accurate value possible for the option using any Montecarlo method of choice, in this case antithetic paths, within 10 seconds of computation for a fixed  $K = 20$ . Figures 5a and 5b were plotted to understand which parameter between the number of paths  $M$  and the number of random numbers generated  $N$  was best to increase and focus on. The vertical, purple dashed lines show the limit at which 10s of computation were exceeded. Given these figures, I decided to compromise between number of paths and Montecarlo iterations to produce the final results in Table 4.



(a) Plot of the error on the mean value for the option for increasing number of paths  $M$ . The value of  $N$  was fixed to 1000.



(b) Plot of the error on the mean value for the option for increasing number of Monte Carlo loops  $N$ . The value of  $M$  was fixed to 50.

Figure 5: Plots of the error on the mean value for the option for increasing parameters  $N$  and  $M$ .

$N$ (thousands)	$K$	$M$	Time(ms)	$V_{MC}$	95 % Confidence Levels	
					Lower Level	Upper Level
9000	20	300	9987	8857.54	8839.64	8875.45

Table 4: Table showing final result for antithetic paths extended Monte Carlo (MC) estimated values for the option.

# Appendix

## Portfolio Pricing Program Listing

```
1 #include <algorithm>
2 #include <chrono>
3 #include <cmath>
4 #include <cstdlib>
5 #include <fstream>
6 #include <functional>
7 #include <iomanip>
8 #include <iostream>
9 #include <random>
10 #include <vector>
11
12 double generateGaussianNoise(double mu, double sigma, int i) {
13     auto halton_seq = [] (int index, int base = 2) {
14         double f = 1, r = 0;
15         while (index > 0) {
16             f = f / base;
17             r = r + f * (index % base);
18             index = index / base;
19         }
20         return r;
21     };
22     static const double two_pi = 2.0 * 3.14159265358979323846;
23
24     double u1, u2;
25     u1 = halton_seq(i, 2);
26     u2 = halton_seq(i, 3);
27
28     double z1;
29     thread_local bool generate;
30     generate = !generate;
31
32     if (!generate)
33         z1 = sqrt(-2.0 * log(u1)) * sin(two_pi * u2);
34     return z1 * sigma + mu;
35
36     double z0;
37     z0 = sqrt(-2.0 * log(u1)) * cos(two_pi * u2);
38     return z0 * sigma + mu;
39 }
40
41 double normalDistribution(double x) { return 0.5 * erfc(-x / sqrt(2.)); }
42
43 double european_options_monteCarlo(
44     const double stock0, const double strikePrice, const double D,
45     const double interestRate, const double sigma, const double maturity,
46     const std::function<double(double, double)> payoff, const int N) {
47     // declare the random number generator
48     static std::mt19937 rng;
49     std::normal_distribution<> ND(0, 1);
50
51     double sum = 0.;
52     for (int i = 0; i < N; i++) {
53         double phi = ND(rng);
54         double ST =
```



```

55     stock0 * exp((interestRate - D - 0.5 * sigma * sigma) * maturity +
56         phi * sqrt(maturity) * sigma);
57     sum += payoff(ST, strikePrice);
58 }
59 return sum / N * exp(-interestRate * maturity);
60 }
61
62 double european_options_halton(
63     const double stock0, const double strikePrice, const double D,
64     const double interestRate, const double sigma, const double maturity,
65     const std::function<double(double, double)> payoff, const int N) {
66     double sum = 0.;
67     for (int i = 1; i <= N; i++) {
68         double phi = generateGaussianNoise(0, 1, i);
69         double ST =
70             stock0 * exp((interestRate - D - 0.5 * sigma * sigma) * maturity +
71                 phi * sqrt(maturity) * sigma);
72         sum += payoff(ST, strikePrice);
73     }
74     return sum / N * exp(-interestRate * maturity);
75 }
76
77 double european_options_monteCarlo_antithetic(
78     const double stock0, const double strikePrice, const double D,
79     const double interestRate, const double sigma, const double maturity,
80     const std::function<double(double, double)> payoff, const int N) {
81     // declare the random number generator
82     static std::mt19937 rng;
83     std::normal_distribution<> ND(0, 1);
84
85     double sum = 0.;
86     for (int i = 0; i < int(N / 2); i++) {
87         double phi = ND(rng);
88         double ST1 =
89             stock0 * std::exp((interestRate - D - 0.5 * sigma * sigma) *
90                 maturity +
91                 phi * std::sqrt(maturity) * sigma);
92         double ST2 =
93             stock0 * std::exp((interestRate - D - 0.5 * sigma * sigma) *
94                 maturity +
95                 (-phi) * std::sqrt(maturity) * sigma);
96         sum += payoff(ST1, strikePrice);
97         sum += payoff(ST2, strikePrice);
98     }
99     return sum / N * std::exp(-interestRate * maturity);
100 }
101
102 double european_options_monteCarlo_moment_match(
103     const double stock0, const double strikePrice, const double D,
104     const double interestRate, const double sigma, const double maturity,
105     const std::function<double(double, double)> payoff, const int N) {
106     // declare the random number generator
107     static std::mt19937 rng;
108     std::normal_distribution<> ND(0, 1);
109     std::vector<double> PLN(N);
110     double sum = 0.0;
111     for (int i = 0; i < int(N / 2); i += 1) {
112         double phi = ND(rng);

```

```

111     PLN[i] = phi;
112     sum += 2 * std::pow(phi, 2);
113 }
114 double sample_variance = std::sqrt(sum / (N - 1));
115 sum = 0.0;
116 for (int i = 0; i < int(N / 2); i++) {
117     double phi = PLN[i] / sample_variance;
118     double ST1 =
119         stock0 * exp((interestRate - D - 0.5 * sigma * sigma) * maturity +
120                     phi * sqrt(maturity) * sigma);
121     double ST2 =
122         stock0 * exp((interestRate - D - 0.5 * sigma * sigma) * maturity +
123                     (-phi) * sqrt(maturity) * sigma);
124     sum += payoff(ST1, strikePrice);
125     sum += payoff(ST2, strikePrice);
126 }
127 return sum / N * exp(-interestRate * maturity);
128 }
129
130 double european_options_analytic(
131     const double stock0, const double strikePrice, const double D,
132     const double interestRate, const double sigma, const double maturity,
133     const std::function<double(double, double, double, double, double,
134                                double, double)>
135     payoff,
136     const double time) {
137
138     const double d1 =
139         (std::log(stock0 / strikePrice) +
140          ((interestRate - D + (pow(sigma, 2) / 2)) * (maturity - time))) /
141         (std::sqrt(maturity - time) * sigma);
142     const double d2 = d1 - sigma * std::sqrt(maturity - time);
143     return payoff(strikePrice, interestRate, maturity, time, d1, d2, D,
144                  stock0);
145 }
146 double path_dependent_options_monteCarlo_momentmatching(double T, double K,
147                                                         double N, double S0,
148                                                         double r, double D,
149                                                         double sigma,
150                                                         double X) {
151     static std::mt19937 rng;
152     std::normal_distribution<> ND(0, 1);
153     std::vector<std::vector<double>> phi_vector(int(N / 2),
154                                                std::vector<double>(int(K)));
155     double sum = 0.0;
156     for (int i = 0; i < int(N / 2); i += 1) {
157         for (int k = 0; k < K; k++) {
158             double phi = ND(rng);
159             phi_vector[i][k] = phi;
160             sum += 2 * std::pow(phi, 2);
161         }
162     }
163     double sample_variance = std::sqrt(sum / ((N)*2 * (K)));
164     sum = 0.;
165     for (int i = 0; i < int(N / 2); i++) {

```

```

166     double dt = T / K;
167     std::vector<double> stockPath1(K + 1);
168     std::vector<double> stockPath2(K + 1);
169     // initialise first value
170     stockPath1[0] = S0;
171     stockPath2[0] = S0;
172     double A1 = 0.;
173     double A2 = 0.;
174     for (int k = 1; k <= K; k++) {
175         double phi = phi_vector[i][(k - 1)];
176         stockPath1[k] =
177             stockPath1[k - 1] *
178             exp((r - D - 0.5 * sigma * sigma) * dt + sigma * sqrt(dt) * phi);
179         stockPath2[k] =
180             stockPath2[k - 1] *
181             exp((r - D - 0.5 * sigma * sigma) * dt + sigma * sqrt(dt) * (-phi
182         ));
183     }
184     for (int k = 1; k <= K; k++) {
185         A1 += stockPath1[k];
186         A2 += stockPath2[k];
187     }
188     A1 /= K;
189     A2 /= K;
190     sum += std::max(A1 - X, 0.);
191     sum += std::max(A2 - X, 0.);
192 }
193 double VMC = (sum / (N)) * exp(-r * T);
194 return VMC;
195 }
196
197 double path_dependent_options_monteCarlo(double T, double K, double N,
198                                           double S0, double r, double D,
199                                           double sigma, double X) {
200     static std::mt19937 rng;
201     std::normal_distribution<> ND(0., 1.);
202     double sum = 0.;
203     for (int n = 0; n < N; n++) {
204         // now create a path
205         double dt = T / K;
206         std::vector<double> stockPath(K + 1);
207         stockPath[0] = S0;
208         for (int i = 1; i <= K; i++) {
209             double phi = ND(rng);
210             stockPath[i] = stockPath[i - 1] * exp((r - D - 0.5 * sigma * sigma) *
211             dt +
212                                     phi * sigma * sqrt(dt));
213         }
214         // and calculate A
215         double A = 0.;
216         for (int i = 1; i <= K; i++) {
217             A += (stockPath[i]);
218         }
219         A /= K;
220         sum += std::max(A - X, 0.);
221     }
222     return sum / N * exp(-r * T);
223 }

```

```

222
223 void path_independent_options() {
224     // path_dependant_option();
225     const double T = 0.5, sigma = 0.41, r = 0.03, D = 0.04, X1 = 70., X2 =
        100.;
226
227     auto long_call_payoff = [](const double S, const double X) {
228         return std::max(S - X, 0.);
229     };
230     auto short_call_payoff = [](const double S, const double X) {
231         return -std::max(S - X, 0.);
232     };
233     auto binary_call_payoff = [](const double S, const double X) {
234         if (S <= X) {
235             return 0.;
236         } else {
237             return 1.;
238         }
239     };
240     auto analytic_long_call_payoff =
241         [](const double X, const double r, const double T, const double t,
242            const double d1, const double d2, const double D, const double S)
243         {
244             return -X * std::exp(-r * (T - t)) * normalDistribution(d2) +
245                 S * std::exp(-D * (T - t)) * normalDistribution(d1);
246         };
247     auto analytic_short_call_payoff =
248         [](const double X, const double r, const double T, const double t,
249            const double d1, const double d2, const double D, const double S)
250         {
251             return +X * std::exp(-r * (T - t)) * normalDistribution(d2) -
252                 S * std::exp(-D * (T - t)) * normalDistribution(d1);
253         };
254     auto analytic_long_binary_call_payoff =
255         [](const double X, const double r, const double T, const double t,
256            const double d1, const double d2, const double D, const double S)
257         {
258             return std::exp(-r * (T - t)) * normalDistribution(d2);
259         };
260
261     const double min_N = 1000;
262     const double max_N = 100000;
263     const size_t N_data_points = 100;
264     const double N_interval = (max_N - min_N) / (N_data_points - 1);
265     const double min_S0 = 70.;
266     const double max_S0 = 100.;
267     const size_t S0_data_points = 2;
268     const double S0_interval = (max_S0 - min_S0) / (S0_data_points - 1);
269     const double M = 100;
270     double S0 = min_S0;
271
272     std::ofstream output1("./Assignment_3/outputs/analytic.task.2.1.csv");
273     for (size_t S{1}; S < 170; S += 1) {
274         if (S == X1 || S == X2) {
275             continue;
276         }
277         double analytic_pi_0 = 0;
278         analytic_pi_0 += european_options_analytic(S, X1, D, r, sigma, T,

```

```

276         analytic_short_call_payoff ,
    0);
277     analytic_pi_0 += european_options_analytic(S, X2, D, r, sigma, T,
278         analytic_long_call_payoff ,
    0);
279     analytic_pi_0 +=
280         2 * X2 *
281         european_options_analytic(S, X2, D, r, sigma, T,
282             analytic_long_binary_call_payoff , 0);
283     analytic_pi_0 += european_options_analytic(S, 0., D, r, sigma, T,
284         analytic_long_call_payoff ,
    0);
285     double analytic_pi_T = 0;
286     analytic_pi_T += european_options_analytic(S, X1, D, r, sigma, T,
287         analytic_short_call_payoff ,
    T);
288     analytic_pi_T += european_options_analytic(S, X2, D, r, sigma, T,
289         analytic_long_call_payoff , T
    );
290     analytic_pi_T +=
291         2 * X2 *
292         european_options_analytic(S, X2, D, r, sigma, T,
293             analytic_long_binary_call_payoff , T);
294     analytic_pi_T += european_options_analytic(S, 0., D, r, sigma, T,
295         analytic_long_call_payoff , T
    );
296     output1 << S << "," << analytic_pi_0 << "," << analytic_pi_T << std::
endl;
297 }
298
299 std::ofstream output2("./Assignment_3/outputs/momentmatch.task.2.1.csv");
300 for (size_t i{}; i < S0_data_points; i += 1) {
301     double N = min_N;
302     for (size_t j{}; j < N_data_points; j += 1) {
303         // Carry out M calculations of montecarlo simulations for the
portfolio
304         // with N random generations
305
306         std::vector<double> PI_N(M);
307         double sum = 0;
308         double variance = 0;
309         double sample_variance = 0;
310         double lower_confidence_limit = 0;
311         double upper_confidence_limit = 0;
312         double mean = 0;
313         auto t1 = std::chrono::high_resolution_clock::now();
314         for (size_t k{}; k < M; k += 1) {
315             double montecarlo_pi = 0;
316             montecarlo_pi += european_options_monteCarlo_moment_match(
317                 S0, X1, D, r, sigma, T, short_call_payoff , N);
318             montecarlo_pi += european_options_monteCarlo_moment_match(
319                 S0, X2, D, r, sigma, T, long_call_payoff , N);
320             montecarlo_pi += 2 * X2 *
321                 european_options_monteCarlo_moment_match(
322                     S0, X2, D, r, sigma, T, binary_call_payoff , N)
;
323             montecarlo_pi += european_options_monteCarlo_moment_match(
324                 S0, 0., D, r, sigma, T, long_call_payoff , N);

```

```

325     sum += montecarlo_pi;
326     PI_N[k] = montecarlo_pi;
327 }
328 auto t2 = std::chrono::high_resolution_clock::now();
329 auto time_taken =
330     std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
331     .count();
332 mean = sum / M;
333 sum = 0.;
334 for (int l{0}; l < M; l += 1) {
335     sum += std::pow((PI_N[l] - mean), 2);
336 }
337 sample_variance = sum / M * (M - 1);
338 lower_confidence_limit = mean - 2 * std::sqrt(sample_variance);
339 upper_confidence_limit = mean + 2 * std::sqrt(sample_variance);
340 // Carry out analytic value of portfolio
341 double analytic_pi = 0;
342 analytic_pi += european_options_analytic(S0, X1, D, r, sigma, T,
343                                           analytic_short_call_payoff,
0);
344 analytic_pi += european_options_analytic(S0, X2, D, r, sigma, T,
345                                           analytic_long_call_payoff,
0);
346 analytic_pi +=
347     2 * X2 *
348     european_options_analytic(S0, X2, D, r, sigma, T,
349                               analytic_long_binary_call_payoff, 0);
350 analytic_pi += european_options_analytic(S0, 0., D, r, sigma, T,
351                                           analytic_long_call_payoff,
0);
352 // Output to file
353 output2 << time_taken << "," << N << "," << S0 << "," << mean << ","
354 << std::sqrt(sample_variance) << "," <<
lower_confidence_limit
355 << "," << upper_confidence_limit << "," << analytic_pi
356 << std::endl;
357 N += N_interval;
358 }
359
360 S0 += S0_interval;
361 }
362 }
363
364 void path_dependent_options() {
365
366     std::mt19937 rng;
367     std::normal_distribution<> ND(0, 1.);
368
369     double S0 = 64000, sigma = 0.42, r = 0.03, T = 2, X = 64000, D = 0.02;
370     int K = 20;
371     const double min_N = 1000;
372     const double max_N = 50000;
373     const size_t N_data_points = 3;
374     double N_Values[3] = {9000, 10000, 8000};
375     double M_Values[3] = {300, 500, 600};
376     const double N_interval = (max_N - min_N) / (N_data_points - 1);
377     // const double N_interval = 0;
378     std::ofstream output1("./Assignment_3/outputs/final.task.2.2.csv");

```

```

379 double K_Values[4] = {70, 150};
380 size_t max_paths = 70;
381 double N = min_N;
382
383 for (size_t j{}; j < N_data_points; j += 1) {
384     N = N_Values[j];
385     // for (size_t current_k{1}; current_k < 100; current_k += 1) {
386     for (size_t current_path{0}; current_path < 3; current_path += 1) {
387         // double K = K_Values[current_k];
388         K = 20.0;
389         double sum = 0;
390         double variance = 0;
391         double sample_variance = 0;
392         double lower_confidence_limit = 0;
393         double upper_confidence_limit = 0;
394         auto t1 = std::chrono::high_resolution_clock::now();
395         size_t paths = M_Values[current_path];
396         std::vector<double> PI_N(paths);
397         for (size_t p{0}; p < paths; p += 1) {
398             double montecarlo = 0;
399             montecarlo = path_dependent_options_monteCarlo_momentmatching(
400                 T, K, N, S0, r, D, sigma, X);
401             sum += montecarlo;
402             PI_N[p] = montecarlo;
403         }
404         auto t2 = std::chrono::high_resolution_clock::now();
405         auto time_taken =
406             std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
407                 .count();
408         double mean = sum / paths;
409         sum = 0.;
410         for (int l{0}; l < paths; l += 1) {
411             sum += std::pow((PI_N[l] - mean), 2);
412         }
413         sample_variance = sum / (paths * (paths - 1));
414         lower_confidence_limit = mean - 2 * std::sqrt(sample_variance);
415         upper_confidence_limit = mean + 2 * std::sqrt(sample_variance);
416         // Output to file
417         output1 << time_taken << "," << N << "," << K << "," << paths << ","
418             << mean << "," << std::sqrt(sample_variance) << ","
419             << lower_confidence_limit << "," << upper_confidence_limit
420             << std::endl;
421         // N += N_interval;
422     }
423     //}
424 }
425 }
426
427 int main() {
428     path_dependent_options();
429     // path_independent_options();
430 }

```

## Graphing Program Listing

```

1 import matplotlib.pyplot as plt
2 import csv
3 import numpy as np
4

```

```

5
6 def firstTask():
7     analyticData = {'S': [], 'analytic_pi_0': [], 'analytic_pi_T': []}
8     with open('outputs/analytic.task.2.1.csv', newline='\n') as csvfile:
9         reader = csv.DictReader(csvfile, fieldnames=[
10             'S', 'analytic_pi_0', 'analytic_pi_T'], quoting=csv.
QUOTENONNUMERIC)
11         for row in reader:
12             analyticData['S'].append(row['S'])
13             analyticData['analytic_pi_0'].append(row['analytic_pi_0'])
14             analyticData['analytic_pi_T'].append(row['analytic_pi_T'])
15         fig = plt.figure()
16         ax = fig.add_subplot(1, 1, 1)
17         plt.grid(True)
18         ax.plot(analyticData['S'], analyticData['analytic_pi_0'],
19                 label=r't=0', color='red', linewidth=2)
20         ax.plot(analyticData['S'], analyticData['analytic_pi_T'],
21                 label=r't=T', color='blue', linewidth=2)
22         plt.xlabel('S')
23         plt.ylabel('V(S,t)')
24         plt.legend()
25         plt.savefig('Solution/analytic_values.png',
26                     bbox_inches='tight', pad_inches=0.25)
27
28     files = ['normal.task.2.1', 'antithetic.task.2.1', 'momentmatch.task
.2.1']
29     allData = {'70': {'normal': {}, 'antithetic': {}, 'momentmatch': {}},
30                '100': {'normal': {}, 'antithetic': {}, 'momentmatch': {}}}
31     for file in files:
32         counter = 0
33         with open('outputs/'+file+'.csv', newline='\n') as csvfile:
34             # N, S0, Montecarlo Avrg PI, Lower confidence limit, Upper
confidence limit, Analytic PI
35             reader = csv.DictReader(csvfile, fieldnames=[
36                 'time_taken', 'n', 's0', 'montecarlo_pi
', 'std', 'lcl', 'ucl', 'analytic_pi'], quoting=csv.QUOTENONNUMERIC)
37             currentFileData = {'time_taken': [], 'n': [], 's0': [], '
montecarlo_pi': [
38                 ], 'std': [], 'error-bars': [[], []], 'analytic_pi': []}
39             for row in reader:
40                 counter += 1
41                 if(counter == 101):
42                     allData[str(int(currentFileData['s0'][0]))]
43                         [[file.split(sep='.')[0]] = currentFileData
44                         currentFileData = {'time_taken': [], 'n': [], 's0': [],
'montecarlo_pi': [
45                             ], 'std': [], 'error-bars': [[], []], 'analytic_pi':
[]}]
46                     currentFileData['n'].append(row['n']/1000)
47                     currentFileData['time_taken'].append(row['time_taken'])
48                     currentFileData['s0'].append(row['s0'])
49                     currentFileData['montecarlo_pi'].append(row['montecarlo_pi'
])
50                     currentFileData['std'].append(
51                         100*row['std']/row['montecarlo_pi'])
52                     currentFileData['error-bars'][0].append(
53                         row['montecarlo_pi']-2*row['std'])
54                     currentFileData['error-bars'][1].append(

```



```

55         row[ 'montecarlo-pi' ]+2*row[ 'std' ])
56         currentFileData[ 'analytic-pi' ].append(row[ 'analytic-pi' ])
57         allData[ str(int(currentFileData[ 's0' ][0]))
58                 ][ file.split(sep='.')[0]] = currentFileData
59
60     locations = { '70_': '', }
61     for dataKey, dataframe in allData.items():
62         for subKey, subFrame in dataframe.items():
63             fig, ax1 = plt.subplots()
64             plt.grid(True)
65             ax2 = ax1.twinx()
66             ax2.tick_params('both', labelsize=14)
67             ax1.tick_params('both', labelsize=14)
68
69             ax1.set_xlabel('N / Thousands', fontsize=16)
70             ax1.set_ylabel('V($S_0$='+dataKey+', t=0)', fontsize=16)
71             ax2.set_ylabel('Standard Error (%)', fontsize=16)
72             # p3 = np.poly1d(np.polyfit(subFrame[ 'n' ], subFrame[ 'std' ], 5))
73             ax2.plot(subFrame[ 'n' ], subFrame[ 'std' ], label=r'(Standard
Error)',
74                     ls='-', color='red', linewidth=2)
75             # This is about confidence intervals
76             n = np.array(subFrame[ 'n' ], dtype=np.float64)
77             mean = np.array(subFrame[ 'montecarlo-pi' ], dtype=np.float64)
78             lower = np.array(subFrame[ 'error-bars' ][0], dtype=np.float64)
79             upper = np.array(subFrame[ 'error-bars' ][1], dtype=np.float64)
80             analytic = np.array(subFrame[ 'analytic-pi' ], dtype=np.float64)
81             ax1.plot(n, mean,
82                     label=r'Montecarlo Average', color='blue', alpha=0.7,
linewidth=1)
83             ax1.plot(n, analytic,
84                     label=r'Analytic Value', ls='dashed', color='green',
linewidth=3)
85             ax1.set_xlim(1, subFrame[ 'n' ][-1])
86             handles, labels = [
87                 (a + b) for a, b in zip(ax1.get_legend_handles_labels(),
ax2.get_legend_handles_labels())]
88             plt.savefig('Solution/confidence_'+dataKey+'_'+subKey+'.png',
89                     bbox_inches='tight', pad_inches=0.25)
90
91             axe = plt.axes(frameon=False)
92             axe.figure.set_size_inches(8, 1)
93             axe.legend(handles, labels, ncol=3, loc='center',
94                     mode="expand", fancybox=False, framealpha=0.0)
95             axe.xaxis.set_visible(False)
96             axe.yaxis.set_visible(False)
97             plt.savefig('Solution/legend.png',
98                     bbox_inches='tight', pad_inches=0)
99
100         labels = { 'antithetic': 'Antithetic Variables',
101                   'normal': 'Basic Montecarlo', 'momentmatch': 'Moment Matching'
102         }
103         colors = { 'antithetic': 'red',
104                   'normal': 'blue', 'momentmatch': 'green' }
105
106     fig_70 = plt.figure()
107     fig_100 = plt.figure()
108     ax_70 = fig_70.add_subplot(1, 1, 1)

```

```

108     ax_100 = fig_100.add_subplot(1, 1, 1)
109
110     for dataKey, dataframe in allData.items():
111         for subDataKey, subDataFrame in dataframe.items():
112             # This is about timing efficiency
113             ax_70.plot(allData["70"][subDataKey]['n'], allData["70"][
subDataKey]['time_taken'],
114                       label=labels[subDataKey], color=colors[subDataKey],
linewidth=2)
115
116             ax_100.plot(allData["100"][subDataKey]['n'], allData["100"][
subDataKey]['time_taken'],
117                        label=labels[subDataKey], color=colors[subDataKey],
linewidth=2)
118             break
119
120     plt.legend()
121     plt.grid(True)
122     plt.xlabel('N')
123     plt.ylabel('Time / ms')
124     fig_70.savefig('Solution/timing_efficiency_70.png',
125                  bbox_inches='tight', pad_inches=0.25)
126     fig_100.savefig('Solution/timing_efficiency_100.png',
127                    bbox_inches='tight', pad_inches=0.25)
128
129 # Fixed N,K and vary M
130
131
132 def vary_M():
133     files = ['paths.task.2.2']
134     allData = {'K': []}
135     value_of_M = 0
136
137     for file in files:
138         counter = 0
139         with open('outputs/'+file+'.csv', newline='\n') as csvfile:
140             # N, S0, Montecarlo Avg PI, Lower confidence limit, Upper
confidence limit, Analytic PI
141             reader = csv.DictReader(csvfile, fieldnames=[
142                 'time_taken', 'n', 'k', 'm', '
montecarlo_pi', 'std', 'lcl', 'ucl'], quoting=csv.QUOTE_NONNUMERIC)
143             currentFileData = {'time_taken': [], 'n': [], 'k': [], 'm': [],
'montecarlo_pi': [
144                 ], 'std': [], 'errorBars': [[], []]}
145             found = False
146             for row in reader:
147                 counter += 1
148                 if(counter == 46):
149                     counter = 1
150                     allData['K'].append(currentFileData)
151                     currentFileData = {'time_taken': [], 'n': [], 'k': [],
'm': [], 'montecarlo_pi': [
152                         ], 'std': [], 'errorBars': [[], []]}
153                     currentFileData['n'].append(row['n']/1000)
154                     currentFileData['time_taken'].append(row['time_taken'])
155                     currentFileData['k'].append(row['k'])
156                     currentFileData['montecarlo_pi'].append(row['montecarlo_pi'
])

```

```

157         currentFileData[ 'std' ].append(
158             100*row[ 'std' ]/row[ 'montecarlo_pi' ])
159         currentFileData[ 'error_bars' ][0].append(
160             row[ 'montecarlo_pi' ]-2*row[ 'std' ])
161         currentFileData[ 'error_bars' ][1].append(
162             row[ 'montecarlo_pi' ]+2*row[ 'std' ])
163         currentFileData[ 'm' ].append(row[ 'm' ])
164         if(found != True and row[ 'time_taken' ] >= 10000):
165             found = True
166             value_of_M = row[ 'm' ]
167         allData[ 'K' ].append(currentFileData)
168
169     fig, ax1 = plt.subplots()
170     plt.grid(True)
171     ax1.tick_params('both', labelsize=14)
172
173     ax1.set_xlabel('Paths', fontsize=16)
174     ax1.set_ylabel('Option Price', fontsize=16)
175     colors = { '20': 'blue', '50': 'green', '100': 'red', '150': 'purple' }
176     for dataKey, dataframe in allData.items():
177         for subFrame in dataframe:
178             # This is about number of paths
179             m = np.array(subFrame[ 'm' ], dtype=np.float64)
180             mean = np.array(subFrame[ 'montecarlo_pi' ], dtype=np.float64)
181             ax1.plot(m, mean,
182                     label=r'$K$='+str(subFrame[ 'k' ][0]), color=colors[ str(
183 int(subFrame[ 'k' ][0])) ], alpha=1, linewidth=1)
184             handles, labels = ax1.get_legend_handles_labels()
185             fig.legend(handles, labels)
186             fig.savefig('Solution/task_2_2-plot_1.png',
187                         bbox_inches='tight', pad_inches=0.25)
188
189     fig, ax2 = plt.subplots()
190     plt.grid(True)
191     ax2.tick_params('both', labelsize=14)
192
193     ax2.set_xlabel('Paths', fontsize=16)
194     ax2.set_ylabel('Standard Error (%)', fontsize=16)
195     ax2.axvline(x=value_of_M, ymin=0, ls='—',
196                color='purple', label=r"10s Computation")
197     found = False
198     for dataKey, dataframe in allData.items():
199         for subFrame in dataframe:
200             # This is about standard error
201             ax2.plot(subFrame[ 'm' ], subFrame[ 'std' ],
202                     label=r'$K$='+str(subFrame[ 'k' ][0]), color=colors[ str(
203 int(subFrame[ 'k' ][0])) ], linewidth=1)
204             handles, labels = ax2.get_legend_handles_labels()
205             fig.legend(handles, labels)
206             fig.savefig('Solution/task_2_2-plot_2.png',
207                         bbox_inches='tight', pad_inches=0.25)
208
209     # Fixed N,M and vary K
210
211     def vary_K():
212         files = [ 'k.task.2.2' ]
213         allData = {}

```

```

213     for file in files:
214         counter = 0
215         with open('outputs/'+file+'.csv', newline='\n') as csvfile:
216             # N, S0, Montecarlo Avg PI, Lower confidence limit, Upper
confidence limit, Analytic PI
217             reader = csv.DictReader(csvfile, fieldnames=[
218                 'time_taken', 'n', 'k', 'm', '
montecarlo_pi', 'std', 'lcl', 'ucl'], quoting=csv.QUOTE_NONNUMERIC)
219             allData = {'time_taken': [], 'n': [], 'k': [], 'm': [], '
montecarlo_pi': [
220                 ], 'std': [], 'errorBars': [[], []]}
221             for row in reader:
222                 allData['n'].append(row['n']/1000)
223                 allData['time_taken'].append(row['time_taken'])
224                 allData['k'].append(row['k'])
225                 allData['montecarlo_pi'].append(row['montecarlo_pi'])
226                 allData['std'].append(
227                     100*row['std']/row['montecarlo_pi'])
228                 allData['errorBars'][0].append(
229                     row['montecarlo_pi']-2*row['std'])
230                 allData['errorBars'][1].append(
231                     row['montecarlo_pi']+2*row['std'])
232                 allData['m'].append(row['m'])
233
234     fig, ax1 = plt.subplots()
235     plt.grid(True)
236     ax1.tick_params('both', labelsize=14)
237
238     ax1.set_xlabel('K', fontsize=16)
239     ax1.set_ylabel('Option Price', fontsize=16)
240
241     # This is about number of paths
242     k = np.array(allData['k'], dtype=np.float64)
243     mean = np.array(allData['montecarlo_pi'], dtype=np.float64)
244     ax1.scatter(k, mean)
245     #ax1.set_xlim(20, allData['k'][-1])
246     fig.savefig('Solution/task_2_2-plot_3.png',
247                 bbox_inches='tight', pad_inches=0.25)
248
249     fig, ax2 = plt.subplots()
250     plt.grid(True)
251     ax2.tick_params('both', labelsize=14)
252     ax2.set_xlabel('K', fontsize=16)
253     ax2.set_ylabel('Standard Error (%)', fontsize=16)
254     #ax2.set_xlim(20, allData['k'][-1])
255     # This is about standard error
256     ax2.plot(allData['k'], allData['std'], linewidth=1)
257
258     handles, labels = ax2.get_legend_handles_labels()
259     fig.legend(handles, labels)
260     fig.savefig('Solution/task_2_2-plot_4.png',
261                 bbox_inches='tight', pad_inches=0.25)
262
263
264 def vary_N():
265     files = ['n.task.2.2']
266     allData = {'K': []}
267     value_of_N = 0

```

```

268     for file in files:
269         counter = 0
270         found = False
271         with open('outputs/'+file+'.csv', newline='\n') as csvfile:
272             # N, S0, Montecarlo Avg PI, Lower confidence limit, Upper
confidence limit, Analytic PI
273             reader = csv.DictReader(csvfile, fieldnames=[
274                 'time_taken', 'n', 'k', 'm', '
montecarlo_pi', 'std', 'lcl', 'ucl'], quoting=csv.QUOTE_NONNUMERIC)
275             currentFileData = {'time_taken': [], 'n': [], 'k': [], 'm': [],
'montecarlo_pi': [
276                 ], 'std': [], 'error_bars': [[], []]}
277             for row in reader:
278                 counter += 1
279                 if(counter == 51 or counter == 92):
280                     allData['K'].append(currentFileData)
281                     currentFileData = {'time_taken': [], 'n': [], 'k': [],
'm': [], 'montecarlo_pi': [
282                         ], 'std': [], 'error_bars': [[], []]}
283                     currentFileData['n'].append(row['n']/1000)
284                     currentFileData['time_taken'].append(row['time_taken'])
285                     currentFileData['k'].append(row['k'])
286                     currentFileData['montecarlo_pi'].append(row['montecarlo_pi'
])
287                     currentFileData['std'].append(
288                         100*row['std']/row['montecarlo_pi'])
289                     currentFileData['error_bars'][0].append(
290                         row['montecarlo_pi']-2*row['std'])
291                     currentFileData['error_bars'][1].append(
292                         row['montecarlo_pi']+2*row['std'])
293                     currentFileData['m'].append(row['m'])
294                     if(found != True and row['time_taken'] >= 10000):
295                         found = True
296                         value_of_N = row['n']/1000
297                     allData['K'].append(currentFileData)
298
299     fig, ax1 = plt.subplots()
300     plt.grid(True)
301     ax1.tick_params('both', labelsize=14)
302
303     ax1.set_xlabel('N/Thousands', fontsize=16)
304     ax1.set_ylabel('Option Price', fontsize=16)
305     colors = {'20': 'blue', '50': 'green', '70': 'red'}
306     for dataKey, dataframe in allData.items():
307         for subFrame in dataframe:
308             # This is about number of paths
309             n = np.array(subFrame['n'], dtype=np.float64)
310             mean = np.array(subFrame['montecarlo_pi'], dtype=np.float64)
311             ax1.plot(n, mean,
312                 label=r'$K$='+str(subFrame['k'][0]), color=colors[str(
int(subFrame['k'][0]))], alpha=1, linewidth=1)
313             handles, labels = ax1.get_legend_handles_labels()
314             fig.legend(handles, labels)
315             fig.savefig('Solution/task_2_2-plot_5.png',
316                 bbox_inches='tight', pad_inches=0.25)
317     fig, ax2 = plt.subplots()
318     plt.grid(True)
319     ax2.tick_params('both', labelsize=14)

```

```

320
321 ax2.set_xlabel('N/Thousands', fontsize=16)
322 ax2.set_ylabel('Standard Error (%)', fontsize=16)
323 ax2.axvline(x=value_of_N, ymin=0, ls='—',
324             color='purple', label=r"10s Computation")
325 for dataKey, dataFrame in allData.items():
326     for subFrame in dataFrame:
327         # This is about standard error
328         ax2.plot(subFrame['n'], subFrame['std'],
329                 label=r'$K$='+str(subFrame['k'][0]), color=colors[str(
330 int(subFrame['k'][0]))], linewidth=1)
331
332     handles, labels = ax2.get_legend_handles_labels()
333     fig.legend(handles, labels)
334     fig.savefig('Solution/task_2_2_plot_6.png',
335                 bbox_inches='tight', pad_inches=0.25)
336
337 vary_M()
338 vary_K()
339 vary_N()

```