Student Id: 9910821

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, σ=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 $\overline{X}$ converges almost surely to the expected value μ. $\overline{X}_n \to \mu$ for $n \to \infty$*

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:

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

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

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

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

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

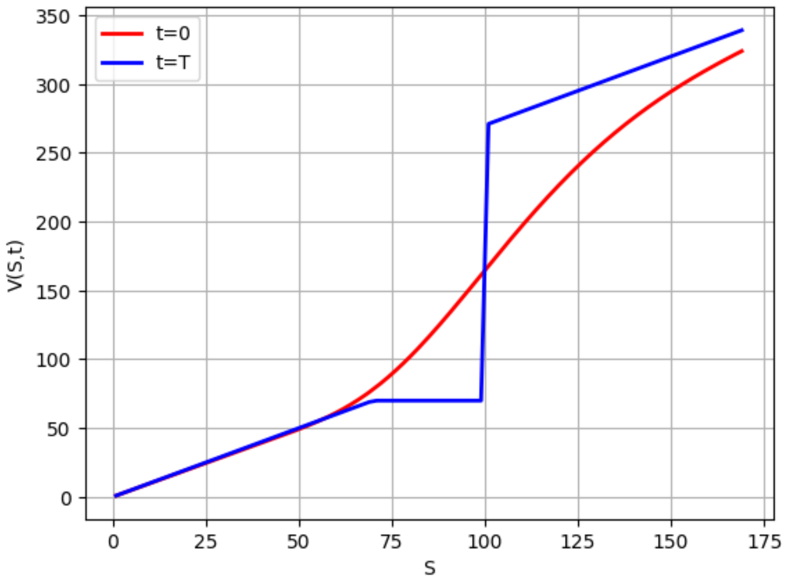


Figure 1: Plots of the price of the portfolio at times $t = 0$ and $t = T$ using analytic solutions.

and

$$d_2 = d_1 - \sigma\sqrt{T-t} \tag{4}$$

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 portofolio $V(S_0, t = 0)$ 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

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

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

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

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

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

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 simulatiom. This mean that the variance is very high and our results are far from reliable.

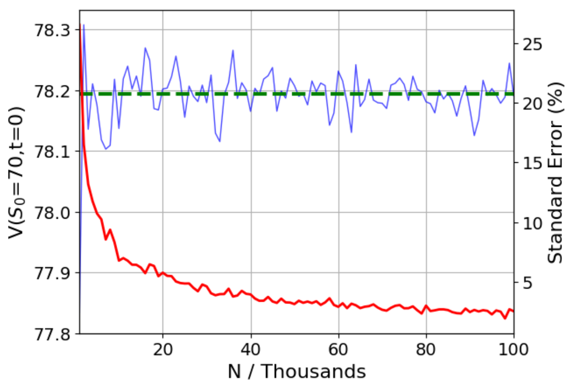
Thus, there is a need for variance reducing methods which build upon Montecarlo. One such method is the use of **antithetic variables**. The idea is that instead of drawing only a normally distributed ϕ we now draw a pair $-\phi$ and add it to the sum and at the end divide by $2N$. The advantage of using 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.

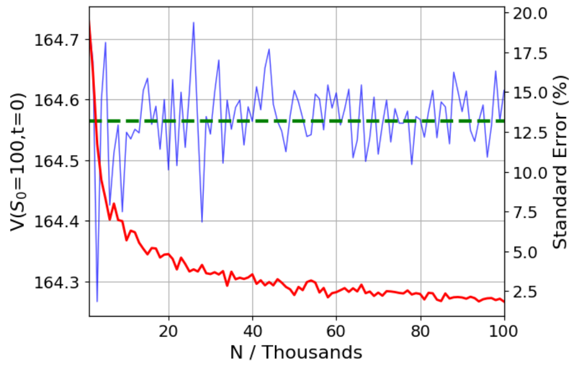
N (thousands)	V_{MC}	V_{AV}	V_{MM}	V_A
1	77.8243±20.637	78.163±15.9292	78.2931±10.1584	78.1939
2	78.3079±12.9237	78.1904±9.58466	78.26±6.21086	78.1939
5	78.1757±8.3882	78.2445±7.62567	78.2246±4.64797	78.1939
10	78.1372±5.30014	78.1608±5.31964	78.2444±2.63015	78.1939
50	78.2078±2.45959	78.1939±2.62684	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 portofolio $V(S_0 = X_1, t = 0)$ compared to the antithetic variables (AV), moment matching (MM) and analytic (A) methods for different values of Montecarlo iterations N.

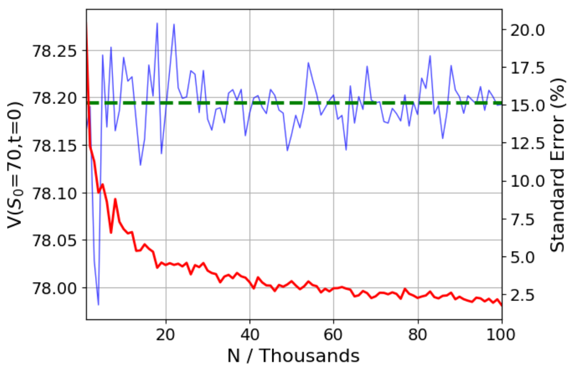
The results, shown in Figure 2 and compared in the Table 3, indicate a clear trend of increasing confidence and accuracy for the methods described. Moment matching appears to be the best of these methods, precision-wise, achieving sub 2% error on the mean. Ensuring the mean and variance match the required 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.



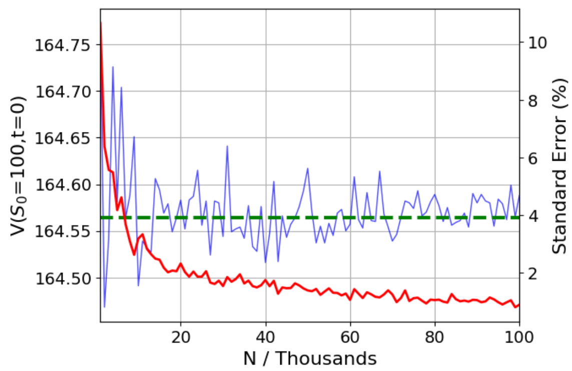
(a) Basic Montecarlo method with S_0=70



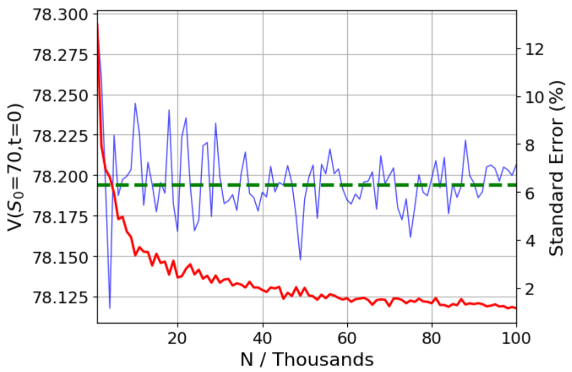
(d) Basic Montecarlo method with S_0=100



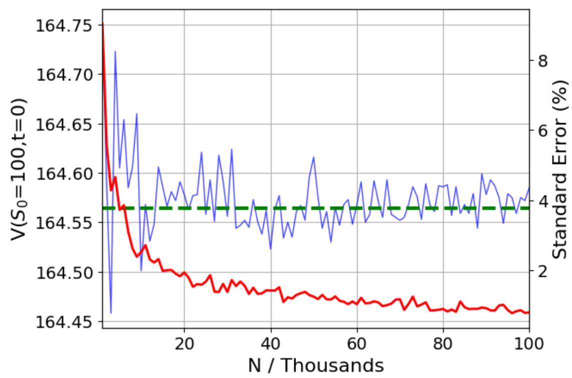
(b) Antithetic Variables method with S_0=70



(e) Antithetic Variables method with S_0=100



(c) Moment Matching method with S_0=70



(f) Moment Matching method with S_0=100

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), anithetic 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.

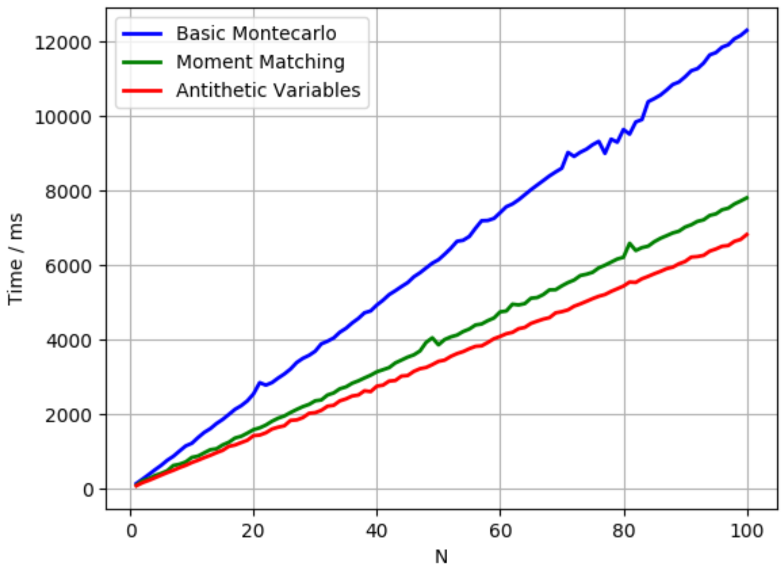


Figure 3: Plots of the time taken by the particular estimation method for $M = 100$ paths as a function of Montecarlo random numbers generated N.

Task 2

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

T=2, σ=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.

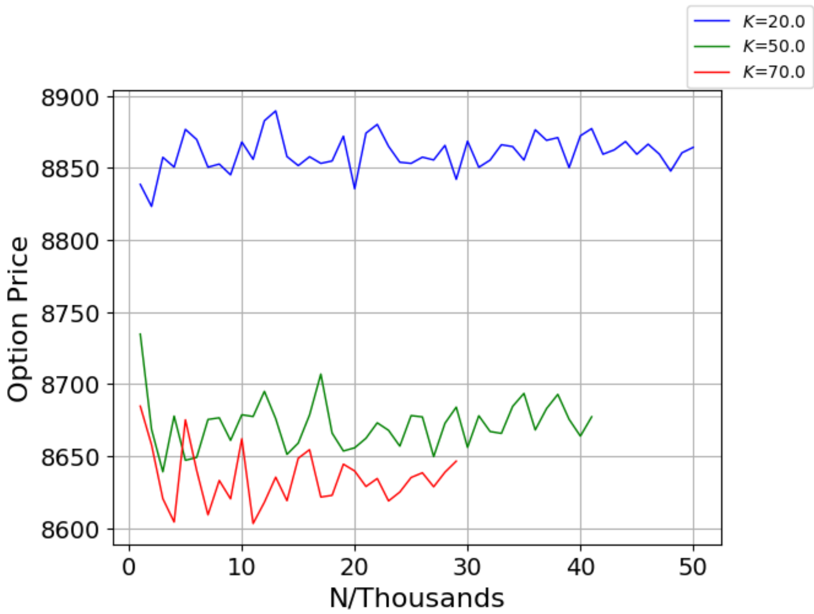
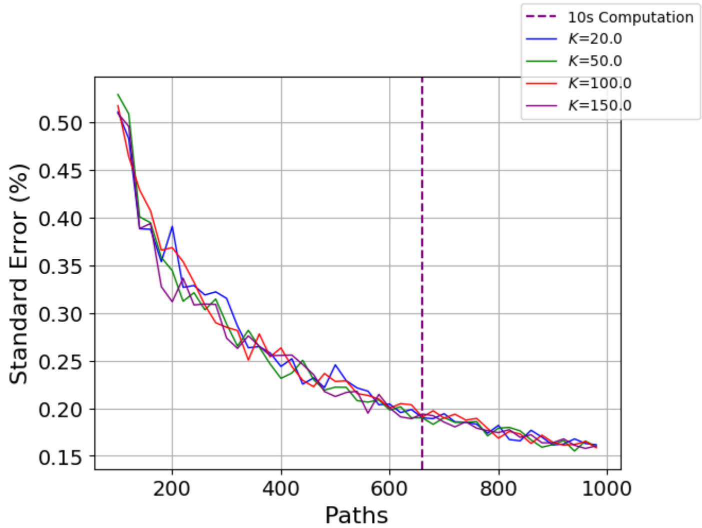


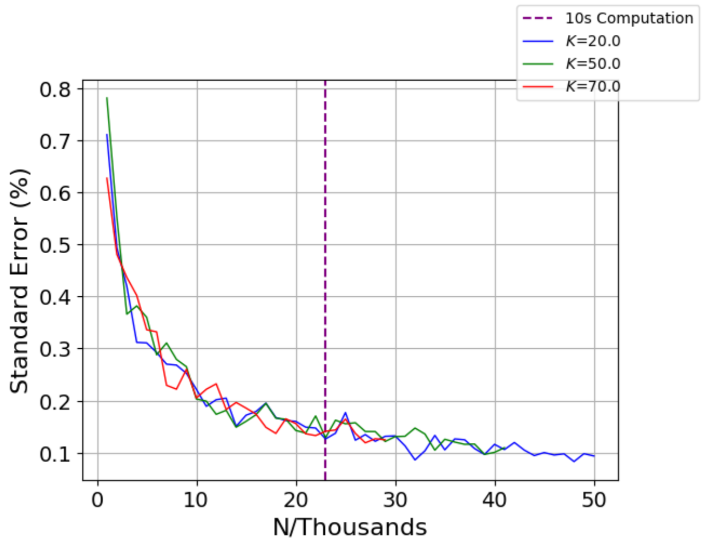
Figure 4: Value of the option as a function of increasing N for different values of K. The number of paths is fixed to M=100. Larger K values were stopped earlier when the simulation took > 30s since the point was to observe trends in option value not time.

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 Montecarlo 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 Montecarlo (MC) estimated values for the option.

Appendix

Portfolio Pricing Program Listing

```
#include <algorithm>
#include <chrono>
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <functional>
#include <iomanip>
#include <iostream>
#include <random>
#include <vector>

double generateGaussianNoise(double mu, double sigma, int i) {
  auto halton_seq = [](int index, int base = 2) {
    double f = 1, r = 0;
    while (index > 0) {
      f = f / base;
      r = r + f * (index % base);
      index = index / base;
    }
    return r;
  };
  static const double two_pi = 2.0 * 3.14159265358979323846;

  double u1, u2;
  u1 = halton_seq(i, 2);
  u2 = halton_seq(i, 3);

  double z1;
  thread_local bool generate;
  generate = !generate;

  if (!generate)
    z1 = sqrt(-2.0 * log(u1)) * sin(two_pi * u2);
  return z1 * sigma + mu;

  double z0;
  z0 = sqrt(-2.0 * log(u1)) * cos(two_pi * u2);
  return z0 * sigma + mu;
}

double normalDistribution(double x) { return 0.5 * erfc(-x / sqrt(2.)); }

double european_options_monteCarlo(
    const double stock0, const double strikePrice, const double D,
    const double interestRate, const double sigma, const double maturity,
    const std::function<double(double, double)> payoff, const int N) {
  // declare the random number generator
  static std::mt19937 rng;
  std::normal_distribution<> ND(0, 1);

  double sum = 0.;
  for (int i = 0; i < N; i++) {
    double phi = ND(rng);
    double ST =
```

```
        stock0 * exp((interestRate - D - 0.5 * sigma * sigma) * maturity +
                      phi * sqrt(maturity) * sigma);
    sum += payoff(ST, strikePrice);
  }
  return sum / N * exp(-interestRate * maturity);
}

double european_options_halton(
    const double stock0, const double strikePrice, const double D,
    const double interestRate, const double sigma, const double maturity,
    const std::function<double(double, double)> payoff, const int N) {
  double sum = 0.;
  for (int i = 1; i <= N; i++) {
    double phi = generateGaussianNoise(0, 1, i);
    double ST =
        stock0 * exp((interestRate - D - 0.5 * sigma * sigma) * maturity +
                      phi * sqrt(maturity) * sigma);
    sum += payoff(ST, strikePrice);
  }
  return sum / N * exp(-interestRate * maturity);
}

double european_options_monteCarlo_antithetic(
    const double stock0, const double strikePrice, const double D,
    const double interestRate, const double sigma, const double maturity,
    const std::function<double(double, double)> payoff, const int N) {
  // declare the random number generator
  static std::mt19937 rng;
  std::normal_distribution<> ND(0, 1);

  double sum = 0.;
  for (int i = 0; i < int(N / 2); i++) {
    double phi = ND(rng);
    double ST1 =
        stock0 * std::exp((interestRate - D - 0.5 * sigma * sigma) * maturity +
                              phi * std::sqrt(maturity) * sigma);
    double ST2 =
        stock0 * std::exp((interestRate - D - 0.5 * sigma * sigma) * maturity +
                              (-phi) * std::sqrt(maturity) * sigma);
    sum += payoff(ST1, strikePrice);
    sum += payoff(ST2, strikePrice);
  }
  return sum / N * std::exp(-interestRate * maturity);
}

double european_options_monteCarlo_moment_match(
    const double stock0, const double strikePrice, const double D,
    const double interestRate, const double sigma, const double maturity,
    const std::function<double(double, double)> payoff, const int N) {
  // declare the random number generator
  static std::mt19937 rng;
  std::normal_distribution<> ND(0, 1);
  std::vector<double> PI_N(N);
  double sum = 0.0;
  for (int i = 0; i < int(N / 2); i += 1) {
    double phi = ND(rng);
```

```
    PI_N[i] = phi;
    sum += 2 * std::pow(phi, 2);
  }
  double sample_variance = std::sqrt(sum / (N - 1));
  sum = 0.0;
  for (int i = 0; i < int(N / 2); i++) {
    double phi = PI_N[i] / sample_variance;
    double ST1 =
        stock0 * exp((interestRate - D - 0.5 * sigma * sigma) * maturity +
                     phi * sqrt(maturity) * sigma);
    double ST2 =
        stock0 * exp((interestRate - D - 0.5 * sigma * sigma) * maturity +
                     (-phi) * sqrt(maturity) * sigma);
    sum += payoff(ST1, strikePrice);
    sum += payoff(ST2, strikePrice);
  }
  return sum / N * exp(-interestRate * maturity);
}

double european_options_analytic(
    const double stock0, const double strikePrice, const double D,
    const double interestRate, const double sigma, const double maturity,
    const std::function<double(double, double, double, double, double,
   double,
                               double, double)>
        payoff,
    const double time) {

  const double d1 =
      (std::log(stock0 / strikePrice) +
       ((interestRate - D + (pow(sigma, 2) / 2)) * (maturity - time))) /
      (std::sqrt(maturity - time) * sigma);
  const double d2 = d1 - sigma * std::sqrt(maturity - time);
  return payoff(strikePrice, interestRate, maturity, time, d1, d2, D,
   stock0);
}

double path_dependent_options_monteCarlo_momentmatching(double T, double K,
                                                        double N, double S0
   ,
                                                        double r, double D,
                                                        double sigma,
                                                        double X) {
  static std::mt19937 rng;
  std::normal_distribution<> ND(0, 1);
  std::vector<std::vector<double>> phi_vector(int(N / 2),
                                              std::vector<double>(int(K)));
  double sum = 0.0;
  for (int i = 0; i < int(N / 2); i += 1) {
    for (int k = 0; k < K; k++) {
      double phi = ND(rng);
      phi_vector[i][k] = phi;
      sum += 2 * std::pow(phi, 2);
    }
  }
  double sample_variance = std::sqrt(sum / ((N)*2 * (K)));
  sum = 0.;
  for (int i = 0; i < int(N / 2); i++) {
```

```cpp
    double dt = T / K;
    std::vector<double> stockPath1(K + 1);
    std::vector<double> stockPath2(K + 1);
    // initialise first value
    stockPath1[0] = S0;
    stockPath2[0] = S0;
    double A1 = 0.;
    double A2 = 0.;
    for (int k = 1; k <= K; k++) {
      double phi = phi_vector[i][(k - 1)];
      stockPath1[k] =
          stockPath1[k - 1] *
          exp((r - D - 0.5 * sigma * sigma) * dt + sigma * sqrt(dt) * phi);
      stockPath2[k] =
          stockPath2[k - 1] *
          exp((r - D - 0.5 * sigma * sigma) * dt + sigma * sqrt(dt) * (-phi
   ));
    }
    for (int k = 1; k <= K; k++) {
      A1 += stockPath1[k];
      A2 += stockPath2[k];
    }
    A1 /= K;
    A2 /= K;
    sum += std::max(A1 - X, 0.);
    sum += std::max(A2 - X, 0.);
  }
  double V_MC = (sum / (N)) * exp(-r * T);
  return V_MC;
}

double path_dependent_options_monteCarlo(double T, double K, double N,
                                         double S0, double r, double D,
                                         double sigma, double X) {
  static std::mt19937 rng;
  std::normal_distribution<> ND(0., 1.);
  double sum = 0.;
  for (int n = 0; n < N; n++) {
    // now create a path
    double dt = T / K;
    std::vector<double> stockPath(K + 1);
    stockPath[0] = S0;
    for (int i = 1; i <= K; i++) {
      double phi = ND(rng);
      stockPath[i] = stockPath[i - 1] * exp((r - D - 0.5 * sigma * sigma) *
     dt +
                                            phi * sigma * sqrt(dt));
    }
    // and calculate A
    double A = 0.;
    for (int i = 1; i <= K; i++) {
      A += (stockPath[i]);
    }
    A /= K;
    sum += std::max(A - X, 0.);
  }
  return sum / N * exp(-r * T);
}
```

```cpp

void path_independent_options() {
  // path_dependant_option();
  const double T = 0.5, sigma = 0.41, r = 0.03, D = 0.04, X1 = 70., X2 =
    100.;

  auto long_call_payoff = [](const double S, const double X) {
    return std::max(S - X, 0.);
  };
  auto short_call_payoff = [](const double S, const double X) {
    return -std::max(S - X, 0.);
  };
  auto binary_call_payoff = [](const double S, const double X) {
    if (S <= X) {
      return 0.;
    } else {
      return 1.;
    }
  };
  auto analytic_long_call_payoff =
      [](const double X, const double r, const double T, const double t,
         const double d1, const double d2, const double D, const double S)
    {
        return -X * std::exp(-r * (T - t)) * normalDistribution(d2) +
               S * std::exp(-D * (T - t)) * normalDistribution(d1);
      };
  auto analytic_short_call_payoff =
      [](const double X, const double r, const double T, const double t,
         const double d1, const double d2, const double D, const double S)
    {
        return +X * std::exp(-r * (T - t)) * normalDistribution(d2) -
               S * std::exp(-D * (T - t)) * normalDistribution(d1);
      };
  auto analytic_long_binary_call_payoff =
      [](const double X, const double r, const double T, const double t,
         const double d1, const double d2, const double D, const double S)
    {
        return std::exp(-r * (T - t)) * normalDistribution(d2);
      };

  const double min_N = 1000;
  const double max_N = 100000;
  const size_t N_data_points = 100;
  const double N_interval = (max_N - min_N) / (N_data_points - 1);
  const double min_S0 = 70.;
  const double max_S0 = 100.;
  const size_t S0_data_points = 2;
  const double S0_interval = (max_S0 - min_S0) / (S0_data_points - 1);
  const double M = 100;
  double S0 = min_S0;

  std::ofstream output1("./Assignment_3/outputs/analytic.task.2.1.csv");
  for (size_t S{1}; S < 170; S += 1) {
    if (S == X1 || S == X2) {
      continue;
    }
    double analytic_pi_0 = 0;
    analytic_pi_0 += european_options_analytic(S, X1, D, r, sigma, T,
```

```
                                                  analytic_short_call_payoff, 0);
    analytic_pi_0 += european_options_analytic(S, X2, D, r, sigma, T,
                                                  analytic_long_call_payoff, 0);
    analytic_pi_0 +=
        2 * X2 *
        european_options_analytic(S, X2, D, r, sigma, T,
                                  analytic_long_binary_call_payoff, 0);
    analytic_pi_0 += european_options_analytic(S, 0., D, r, sigma, T,
                                                  analytic_long_call_payoff, 0);
    double analytic_pi_T = 0;
    analytic_pi_T += european_options_analytic(S, X1, D, r, sigma, T,
                                                  analytic_short_call_payoff, T);
    analytic_pi_T += european_options_analytic(S, X2, D, r, sigma, T,
                                                  analytic_long_call_payoff, T);
    analytic_pi_T +=
        2 * X2 *
        european_options_analytic(S, X2, D, r, sigma, T,
                                  analytic_long_binary_call_payoff, T);
    analytic_pi_T += european_options_analytic(S, 0., D, r, sigma, T,
                                                  analytic_long_call_payoff, T);
    output1 << S << "," << analytic_pi_0 << "," << analytic_pi_T << std::endl;
  }

  std::ofstream output2("./Assignment_3/outputs/momentmatch.task.2.1.csv");
  for (size_t i{}; i < S0_data_points; i += 1) {
    double N = min_N;
    for (size_t j{}; j < N_data_points; j += 1) {
      // Carry out M calculations of montecarlo simulations for the portfolio
      // with N random generations

      std::vector<double> PI_N(M);
      double sum = 0;
      double variance = 0;
      double sample_variance = 0;
      double lower_confidence_limit = 0;
      double upper_confidence_limit = 0;
      double mean = 0;
      auto t1 = std::chrono::high_resolution_clock::now();
      for (size_t k{}; k < M; k += 1) {
        double montecarlo_pi = 0;
        montecarlo_pi += european_options_monteCarlo_moment_match(
            S0, X1, D, r, sigma, T, short_call_payoff, N);
        montecarlo_pi += european_options_monteCarlo_moment_match(
            S0, X2, D, r, sigma, T, long_call_payoff, N);
        montecarlo_pi += 2 * X2 *
                         european_options_monteCarlo_moment_match(
                             S0, X2, D, r, sigma, T, binary_call_payoff, N);
        montecarlo_pi += european_options_monteCarlo_moment_match(
            S0, 0., D, r, sigma, T, long_call_payoff, N);
```

```
        sum += montecarlo_pi;
        PI_N[k] = montecarlo_pi;
      }
      auto t2 = std::chrono::high_resolution_clock::now();
      auto time_taken =
          std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
              .count();
      mean = sum / M;
      sum = 0.;
      for (int l{0}; l < M; l += 1) {
        sum += std::pow((PI_N[l] - mean), 2);
      }
      sample_variance = sum / M * (M - 1);
      lower_confidence_limit = mean - 2 * std::sqrt(sample_variance);
      upper_confidence_limit = mean + 2 * std::sqrt(sample_variance);
      // Carry out analytic value of portfolio
      double analytic_pi = 0;
      analytic_pi += european_options_analytic(S0, X1, D, r, sigma, T,
                                               analytic_short_call_payoff, 0);
      analytic_pi += european_options_analytic(S0, X2, D, r, sigma, T,
                                               analytic_long_call_payoff, 0);
      analytic_pi +=
          2 * X2 *
          european_options_analytic(S0, X2, D, r, sigma, T,
                                    analytic_long_binary_call_payoff, 0);
      analytic_pi += european_options_analytic(S0, 0., D, r, sigma, T,
                                               analytic_long_call_payoff, 0);
      // Output to file
      output2 << time_taken << "," << N << "," << S0 << "," << mean << ","
              << std::sqrt(sample_variance) << "," << lower_confidence_limit
              << "," << upper_confidence_limit << "," << analytic_pi
              << std::endl;
      N += N_interval;
    }

    S0 += S0_interval;
  }
}

void path_dependent_options() {

  std::mt19937 rng;
  std::normal_distribution<> ND(0, 1.);

  double S0 = 64000, sigma = 0.42, r = 0.03, T = 2, X = 64000, D = 0.02;
  int K = 20;
  const double min_N = 1000;
  const double max_N = 50000;
  const size_t N_data_points = 3;
  double N_Values[3] = {9000, 10000, 8000};
  double M_Values[3] = {300, 500, 600};
  const double N_interval = (max_N - min_N) / (N_data_points - 1);
  // const double N_interval = 0;
  std::ofstream output1("./Assignment_3/outputs/final.task.2.2.csv");
```

```cpp
  double K_Values[4] = {70, 150};
  size_t max_paths = 70;
  double N = min_N;

  for (size_t j{}; j < N_data_points; j += 1) {
    N = N_Values[j];
    // for (size_t current_k{1}; current_k < 100; current_k += 1) {
    for (size_t current_path{0}; current_path < 3; current_path += 1) {
      // double K = K_Values[current_k];
      K = 20.0;
      double sum = 0;
      double variance = 0;
      double sample_variance = 0;
      double lower_confidence_limit = 0;
      double upper_confidence_limit = 0;
      auto t1 = std::chrono::high_resolution_clock::now();
      size_t paths = M_Values[current_path];
      std::vector<double> PI_N(paths);
      for (size_t p{0}; p < paths; p += 1) {
        double montecarlo = 0;
        montecarlo = path_dependent_options_monteCarlo_momentmatching(
            T, K, N, S0, r, D, sigma, X);
        sum += montecarlo;
        PI_N[p] = montecarlo;
      }
      auto t2 = std::chrono::high_resolution_clock::now();
      auto time_taken =
          std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
              .count();
      double mean = sum / paths;
      sum = 0.;
      for (int l{0}; l < paths; l += 1) {
        sum += std::pow((PI_N[l] - mean), 2);
      }
      sample_variance = sum / (paths * (paths - 1));
      lower_confidence_limit = mean - 2 * std::sqrt(sample_variance);
      upper_confidence_limit = mean + 2 * std::sqrt(sample_variance);
      // Output to file
      output1 << time_taken << "," << N << "," << K << "," << paths << ","
              << mean << "," << std::sqrt(sample_variance) << ","
              << lower_confidence_limit << "," << upper_confidence_limit
              << std::endl;
      // N += N_interval;
    }
    //}
  }
}

int main() {
  path_dependent_options();
  // path_independent_options();
}
```

Graphing Program Listing

```python
import matplotlib.pyplot as plt
import csv
import numpy as np

```

```python

def firstTask():
    analyticData = {'S': [], 'analytic_pi_0': [], 'analytic_pi_T': []}
    with open('outputs/analytic.task.2.1.csv', newline='\n') as csvfile:
        reader = csv.DictReader(csvfile, fieldnames=[
            'S', 'analytic_pi_0', 'analytic_pi_T'], quoting=csv.
QUOTE_NONNUMERIC)
        for row in reader:
            analyticData['S'].append(row['S'])
            analyticData['analytic_pi_0'].append(row['analytic_pi_0'])
            analyticData['analytic_pi_T'].append(row['analytic_pi_T'])
        fig = plt.figure()
        ax = fig.add_subplot(1, 1, 1)
        plt.grid(True)
        ax.plot(analyticData['S'], analyticData['analytic_pi_0'],
                label=r't=0', color='red', linewidth=2)
        ax.plot(analyticData['S'], analyticData['analytic_pi_T'],
                label=r't=T', color='blue', linewidth=2)
        plt.xlabel('S')
        plt.ylabel('V(S,t)')
        plt.legend()
        plt.savefig('Solution/analytic_values.png',
                    bbox_inches='tight', pad_inches=0.25)

    files = ['normal.task.2.1', 'antithetic.task.2.1', 'momentmatch.task
.2.1']
    allData = {'70': {'normal': {}, 'antithetic': {}, 'momentmatch': {}},
               '100': {'normal': {}, 'antithetic': {}, 'momentmatch': {}}}
    for file in files:
        counter = 0
        with open('outputs/'+file+'.csv', newline='\n') as csvfile:
            # N, S0, Montecarlo Avrg PI, Lower confidence limit,Upper
confidence limit, Analytic PI
            reader = csv.DictReader(csvfile, fieldnames=[
                                    'time_taken', 'n', 's0', 'montecarlo_pi
', 'std', 'lcl', 'ucl', 'analytic_pi'], quoting=csv.QUOTE_NONNUMERIC)
            currentFileData = {'time_taken': [], 'n': [], 's0': [], '
montecarlo_pi': [
            ], 'std': [], 'error_bars': [[], []], 'analytic_pi': []}
            for row in reader:
                counter += 1
                if(counter == 101):
                    allData[str(int(currentFileData['s0'][0]))
                            ][file.split(sep='.')[0]] = currentFileData
                    currentFileData = {'time_taken': [], 'n': [], 's0': [],
 'montecarlo_pi': [
                    ], 'std': [], 'error_bars': [[], []], 'analytic_pi':
[]}
                currentFileData['n'].append(row['n']/1000)
                currentFileData['time_taken'].append(row['time_taken'])
                currentFileData['s0'].append(row['s0'])
                currentFileData['montecarlo_pi'].append(row['montecarlo_pi'
])
                currentFileData['std'].append(
                    100*row['std']/row['montecarlo_pi'])
                currentFileData['error_bars'][0].append(
                    row['montecarlo_pi']-2*row['std'])
                currentFileData['error_bars'][1].append(
```

```
                    row['montecarlo_pi']+2*row['std'])
                currentFileData['analytic_pi'].append(row['analytic_pi'])
            allData[str(int(currentFileData['s0'][0]))
                    ][file.split(sep='.')[0]] = currentFileData

    locations = {'70_': '', }
    for dataKey, dataFrame in allData.items():
        for subKey, subFrame in dataFrame.items():
            fig, ax1 = plt.subplots()
            plt.grid(True)
            ax2 = ax1.twinx()
            ax2.tick_params('both', labelsize=14)
            ax1.tick_params('both', labelsize=14)

            ax1.set_xlabel('N / Thousands', fontsize=16)
            ax1.set_ylabel('V($S_0$='+dataKey+',t=0)', fontsize=16)
            ax2.set_ylabel('Standard Error (%)', fontsize=16)
            # p3 = np.poly1d(np.polyfit(subFrame['n'], subFrame['std'], 5))
            ax2.plot(subFrame['n'], subFrame['std'], label=r'(Standard Error)',
                     ls='-', color='red', linewidth=2)
            # This is about confidence intervals
            n = np.array(subFrame['n'], dtype=np.float64)
            mean = np.array(subFrame['montecarlo_pi'], dtype=np.float64)
            lower = np.array(subFrame['error_bars'][0], dtype=np.float64)
            upper = np.array(subFrame['error_bars'][1], dtype=np.float64)
            analytic = np.array(subFrame['analytic_pi'], dtype=np.float64)
            ax1.plot(n, mean,
                     label=r'Montecarlo Average', color='blue', alpha=0.7, linewidth=1)
            ax1.plot(n, analytic,
                     label=r'Analytic Value', ls='dashed', color='green', linewidth=3)
            ax1.set_xlim(1, subFrame['n'][-1])
            handles, labels = [
                (a + b) for a, b in zip(ax1.get_legend_handles_labels(), ax2.get_legend_handles_labels())]
            plt.savefig('Solution/confidence_'+dataKey+'_'+subKey+'.png',
                        bbox_inches='tight', pad_inches=0.25)

            axe = plt.axes(frameon=False)
            axe.figure.set_size_inches(8, 1)
            axe.legend(handles, labels, ncol=3, loc='center',
                       mode="expand", fancybox=False, framealpha=0.0)
            axe.xaxis.set_visible(False)
            axe.yaxis.set_visible(False)
            plt.savefig('Solution/legend.png',
                        bbox_inches='tight', pad_inches=0)

    labels = {'antithetic': 'Antithetic Variables',
              'normal': 'Basic Montecarlo', 'momentmatch': 'Moment Matching'}
    colors = {'antithetic': 'red',
              'normal': 'blue', 'momentmatch': 'green'}

    fig_70 = plt.figure()
    fig_100 = plt.figure()
    ax_70 = fig_70.add_subplot(1, 1, 1)
```

```python
    ax_100 = fig_100.add_subplot(1, 1, 1)

    for dataKey, dataFrame in allData.items():
        for subDataKey, subDataFrame in dataFrame.items():
            # This is about timing efficiency
            ax_70.plot(allData["70"][subDataKey]['n'], allData["70"][subDataKey]['time_taken'],
                           label=labels[subDataKey], color=colors[subDataKey], linewidth=2)

            ax_100.plot(allData["100"][subDataKey]['n'], allData["100"][subDataKey]['time_taken'],
                            label=labels[subDataKey], color=colors[subDataKey], linewidth=2)
        break

    plt.legend()
    plt.grid(True)
    plt.xlabel('N')
    plt.ylabel('Time / ms')
    fig_70.savefig('Solution/timing_efficiency_70.png',
                   bbox_inches='tight', pad_inches=0.25)
    fig_100.savefig('Solution/timing_efficiency_100.png',
                    bbox_inches='tight', pad_inches=0.25)

# Fixed N,K and vary M


def vary_M():
    files = ['paths.task.2.2']
    allData = {'K': []}
    value_of_M = 0

    for file in files:
        counter = 0
        with open('outputs/'+file+'.csv', newline='\n') as csvfile:
            # N, S0, Montecarlo Avrg PI, Lower confidence limit,Upper confidence limit, Analytic PI
            reader = csv.DictReader(csvfile, fieldnames=[
                                    'time_taken', 'n', 'k', 'm', 'montecarlo_pi', 'std', 'lcl', 'ucl'], quoting=csv.QUOTE_NONNUMERIC)
            currentFileData = {'time_taken': [], 'n': [], 'k': [], 'm': [], 'montecarlo_pi': [
            ], 'std': [], 'error_bars': [[], []]}
            found = False
            for row in reader:
                counter += 1
                if(counter == 46):
                    counter = 1
                    allData['K'].append(currentFileData)
                    currentFileData = {'time_taken': [], 'n': [], 'k': [], 'm': [], 'montecarlo_pi': [
                    ], 'std': [], 'error_bars': [[], []]}
                currentFileData['n'].append(row['n']/1000)
                currentFileData['time_taken'].append(row['time_taken'])
                currentFileData['k'].append(row['k'])
                currentFileData['montecarlo_pi'].append(row['montecarlo_pi'])
```

```python
                currentFileData['std'].append(
                    100*row['std']/row['montecarlo_pi'])
                currentFileData['error_bars'][0].append(
                    row['montecarlo_pi']-2*row['std'])
                currentFileData['error_bars'][1].append(
                    row['montecarlo_pi']+2*row['std'])
                currentFileData['m'].append(row['m'])
                if(found != True and row['time_taken'] >= 10000):
                    found = True
                    value_of_M = row['m']
            allData['K'].append(currentFileData)

    fig, ax1 = plt.subplots()
    plt.grid(True)
    ax1.tick_params('both', labelsize=14)

    ax1.set_xlabel('Paths', fontsize=16)
    ax1.set_ylabel('Option Price', fontsize=16)
    colors = {'20': 'blue', '50': 'green', '100': 'red', '150': 'purple'}
    for dataKey, dataFrame in allData.items():
        for subFrame in dataFrame:
            # This is about number of paths
            m = np.array(subFrame['m'], dtype=np.float64)
            mean = np.array(subFrame['montecarlo_pi'], dtype=np.float64)
            ax1.plot(m, mean,
                     label=r'$K$='+str(subFrame['k'][0]), color=colors[str(
   int(subFrame['k'][0]))], alpha=1, linewidth=1)
        handles, labels = ax1.get_legend_handles_labels()
        fig.legend(handles, labels)
        fig.savefig('Solution/task_2_2_plot_1.png',
                    bbox_inches='tight', pad_inches=0.25)
    fig, ax2 = plt.subplots()
    plt.grid(True)
    ax2.tick_params('both', labelsize=14)

    ax2.set_xlabel('Paths', fontsize=16)
    ax2.set_ylabel('Standard Error (%)', fontsize=16)
    ax2.axvline(x=value_of_M, ymin=0, ls='--',
                color='purple', label=r"10s Computation")
    found = False
    for dataKey, dataFrame in allData.items():
        for subFrame in dataFrame:
            # This is about standard error
            ax2.plot(subFrame['m'], subFrame['std'],
                     label=r'$K$='+str(subFrame['k'][0]), color=colors[str(
   int(subFrame['k'][0]))], linewidth=1)

        handles, labels = ax2.get_legend_handles_labels()
        fig.legend(handles, labels)
        fig.savefig('Solution/task_2_2_plot_2.png',
                    bbox_inches='tight', pad_inches=0.25)

# Fixed N,M and vary K


def vary_K():
    files = ['k.task.2.2']
    allData = {}
```

```
    for file in files:
        counter = 0
        with open('outputs/'+file+'.csv', newline='\n') as csvfile:
            # N, S0, Montecarlo Avrg PI, Lower confidence limit,Upper confidence limit, Analytic PI
            reader = csv.DictReader(csvfile, fieldnames=[
                                         'time_taken', 'n', 'k', 'm', 'montecarlo_pi', 'std', 'lcl', 'ucl'], quoting=csv.QUOTE_NONNUMERIC)
            allData = {'time_taken': [], 'n': [], 'k': [], 'm': [], 'montecarlo_pi': [
            ], 'std': [], 'error_bars': [[], []]}
            for row in reader:
                allData['n'].append(row['n']/1000)
                allData['time_taken'].append(row['time_taken'])
                allData['k'].append(row['k'])
                allData['montecarlo_pi'].append(row['montecarlo_pi'])
                allData['std'].append(
                    100*row['std']/row['montecarlo_pi'])
                allData['error_bars'][0].append(
                    row['montecarlo_pi']-2*row['std'])
                allData['error_bars'][1].append(
                    row['montecarlo_pi']+2*row['std'])
                allData['m'].append(row['m'])

    fig, ax1 = plt.subplots()
    plt.grid(True)
    ax1.tick_params('both', labelsize=14)

    ax1.set_xlabel('K', fontsize=16)
    ax1.set_ylabel('Option Price', fontsize=16)

    # This is about number of paths
    k = np.array(allData['k'], dtype=np.float64)
    mean = np.array(allData['montecarlo_pi'], dtype=np.float64)
    ax1.scatter(k, mean)
    #ax1.set_xlim(20, allData['k'][-1])
    fig.savefig('Solution/task_2_2_plot_3.png',
                bbox_inches='tight', pad_inches=0.25)

    fig, ax2 = plt.subplots()
    plt.grid(True)
    ax2.tick_params('both', labelsize=14)
    ax2.set_xlabel('K', fontsize=16)
    ax2.set_ylabel('Standard Error (%)', fontsize=16)
    #ax2.set_xlim(20, allData['k'][-1])
    # This is about standard error
    ax2.plot(allData['k'], allData['std'], linewidth=1)

    handles, labels = ax2.get_legend_handles_labels()
    fig.legend(handles, labels)
    fig.savefig('Solution/task_2_2_plot_4.png',
                bbox_inches='tight', pad_inches=0.25)


def vary_N():
    files = ['n.task.2.2']
    allData = {'K': []}
    value_of_N = 0
```

```
    for file in files:
        counter = 0
        found = False
        with open('outputs/'+file+'.csv', newline='\n') as csvfile:
            # N, S0, Montecarlo Avrg PI, Lower confidence limit,Upper confidence limit, Analytic PI
            reader = csv.DictReader(csvfile, fieldnames=[
                                        'time_taken', 'n', 'k', 'm', 'montecarlo_pi', 'std', 'lcl', 'ucl'], quoting=csv.QUOTE_NONNUMERIC)
            currentFileData = {'time_taken': [], 'n': [], 'k': [], 'm': [], 'montecarlo_pi': [
            ], 'std': [], 'error_bars': [[], []]}
            for row in reader:
                counter += 1
                if(counter == 51 or counter == 92):
                    allData['K'].append(currentFileData)
                    currentFileData = {'time_taken': [], 'n': [], 'k': [], 'm': [], 'montecarlo_pi': [
                    ], 'std': [], 'error_bars': [[], []]}
                currentFileData['n'].append(row['n']/1000)
                currentFileData['time_taken'].append(row['time_taken'])
                currentFileData['k'].append(row['k'])
                currentFileData['montecarlo_pi'].append(row['montecarlo_pi'])
                currentFileData['std'].append(
                    100*row['std']/row['montecarlo_pi'])
                currentFileData['error_bars'][0].append(
                    row['montecarlo_pi']-2*row['std'])
                currentFileData['error_bars'][1].append(
                    row['montecarlo_pi']+2*row['std'])
                currentFileData['m'].append(row['m'])
                if(found != True and row['time_taken'] >= 10000):
                    found = True
                    value_of_N = row['n']/1000
            allData['K'].append(currentFileData)

    fig, ax1 = plt.subplots()
    plt.grid(True)
    ax1.tick_params('both', labelsize=14)

    ax1.set_xlabel('N/Thousands', fontsize=16)
    ax1.set_ylabel('Option Price', fontsize=16)
    colors = {'20': 'blue', '50': 'green', '70': 'red'}
    for dataKey, dataFrame in allData.items():
        for subFrame in dataFrame:
            # This is about number of paths
            n = np.array(subFrame['n'], dtype=np.float64)
            mean = np.array(subFrame['montecarlo_pi'], dtype=np.float64)
            ax1.plot(n, mean,
                     label=r'$K$='+str(subFrame['k'][0]), color=colors[str(int(subFrame['k'][0]))], alpha=1, linewidth=1)
        handles, labels = ax1.get_legend_handles_labels()
        fig.legend(handles, labels)
        fig.savefig('Solution/task_2_2_plot_5.png',
                    bbox_inches='tight', pad_inches=0.25)
    fig, ax2 = plt.subplots()
    plt.grid(True)
    ax2.tick_params('both', labelsize=14)
```

```

    ax2.set_xlabel('N/Thousands', fontsize=16)
    ax2.set_ylabel('Standard Error (%)', fontsize=16)
    ax2.axvline(x=value_of_N, ymin=0, ls='--',
                color='purple', label=r"10s Computation")
    for dataKey, dataFrame in allData.items():
        for subFrame in dataFrame:
            # This is about standard error
            ax2.plot(subFrame['n'], subFrame['std'],
                     label=r'$K$='+str(subFrame['k'][0]), color=colors[str(
    int(subFrame['k'][0]))], linewidth=1)

        handles, labels = ax2.get_legend_handles_labels()
        fig.legend(handles, labels)
        fig.savefig('Solution/task_2_2_plot_6.png',
                    bbox_inches='tight', pad_inches=0.25)


vary_M()
vary_K()
vary_N()
```