Assignment 1: Monte Carlo Methods

10134621

School of Mathematics

The University of Manchester

Fourth Year
Computational Finance

April 2021

Abstract

In this assignment we look to use the Monte Carlo method to value various types of options. We begin by attempting to value a portfolio composed of different standard European options such as call options, put options and their binary alternatives. We calculate confidence intervals for our numerical results and compare to the analytic solution where possible. To improve the accuracy and efficiency of our results we also explore some extensions to the base Monte Carlo method, namely using antithetic variables or quasi-Monte Carlo methods such as the Halton sequence. We go onto explore path dependent options, specifically floating strike Asian call options, and how our result varies with the model parameters. Finally, we attempt to obtain the most accurate value of this option in a time-limited scenario.

1. Stock options

1.1. Problem setup

The portfolio we seek to value is comprised of long two put options with strike price X_1, long a call option with strike price X_2, short X_2 binary put options with strike price X_2 and unit payoff and short a call option with strike price equal to zero. The parameters are: expiry time $T = 0.5$, volatility $\sigma = 0.25$, interest rate $r = 0.03$, dividend rate $D_0 = 0.01$, $X_1 = 450$ and $X_2 = 700$. Note that all code referenced in this report can be found in the appendicies.

1.2. Portfolio value at expiry

As an initial check, to ensure our portfolio is setup correctly, we plot the value of the portfolio at expiry for various current share prices, S_0, in the range 0 to 1150 as shown in Figure (1). This matches the plot given in the problem.

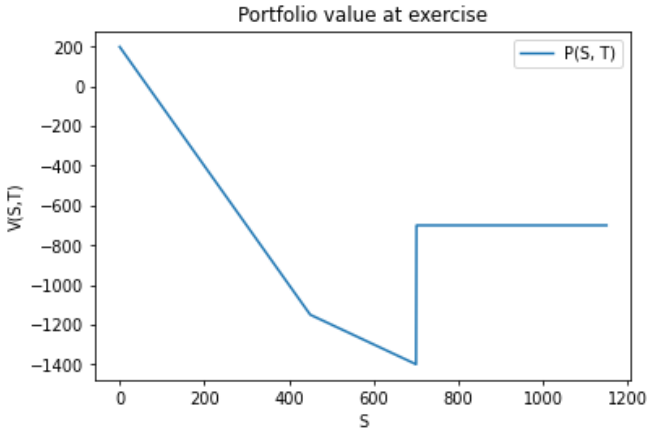

Figure 1: An image showing the value of the portfolio at exercise, $V(S,T)$, for various current values of the share price, S_0.

1.3. Current portfolio value

We can use Monte Carlo simulation to approximate the value of the portfolio at the current time. We do this for an increasing number of simulation paths and compare to the analytical solution. The function that performed the Monte Carlo simulation is shown in Listing (1).

Figure (2) shows the case for $S_0 = X_1$ and Figure (3) shows it for $S_0 = X_2$. In both scenarios we see that as N increases, our numerical approximation converges to the analytical solution.

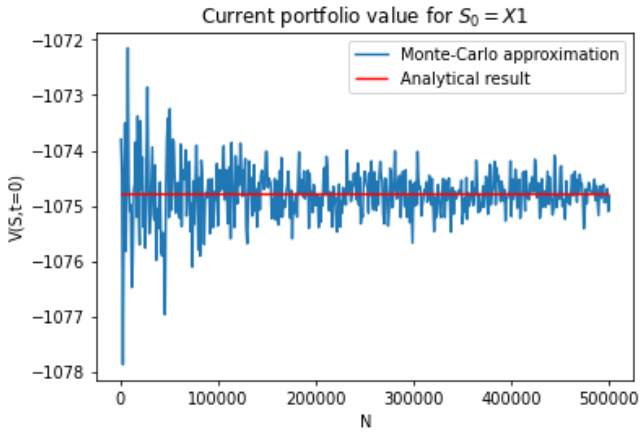

Figure 2: An image showing the current value of the portfolio, for $S_0 = X_1$, calculated using Monte Carlo simulation for an increasing number of paths N (solid blue line). The analytical solution (solid red line) is shown for comparison and has exact value $\Pi(S,T) = -1074.78$.

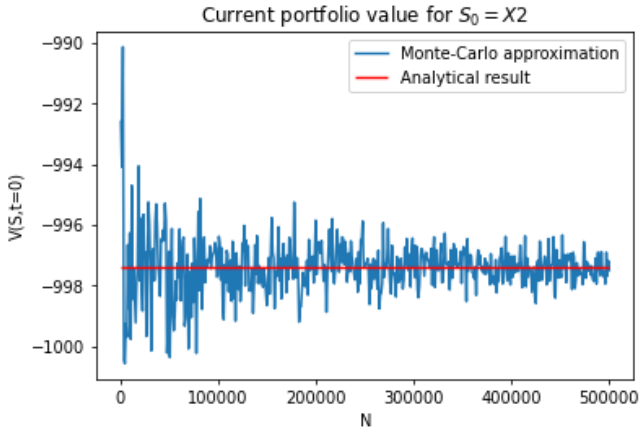

Figure 3: An image showing the current value of the portfolio, for $S_0 = X_2$, calculated using Monte Carlo simulation for an increasing number of paths N (solid blue line). The analytical solution (solid red line) is shown for comparison and has exact value $\Pi(S,T) = -997.365$.

1.4. Confidence intervals

For the case $S_0 = X_1$ we obtain confidence intervals by repeating the Monte Carlo simulation M times and making use of the Central Limit Theorem.

Setting $M = 100$ we calculate the 95% confidence interval to be $[-1074.81, -1074.72]$. The code for calculating confidence intervals is shown in Listing (2).

We can also look at how the confidence interval changes as we increase the number of Monte Carlo paths N as shown in Figure (4). Due to the computational expense we restrict $M = 100$. As N increases the confidence interval narrows but has significant diminishing returns past $N \approx 200,000$.

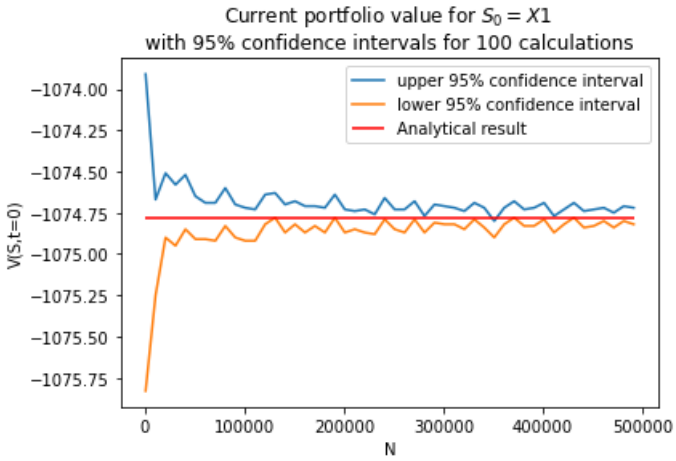

Figure 4: An image showing the the upper (solid blue line) and lower (solid orange line) 95% confidence intervals, using Monte Carlo simulation, for an increasing number of paths N. The analytical solution (solid red line) is shown for comparison and has exact value $\Pi(S,T) = -1074.78$.

1.5. Antithetic variables

We can slightly improve our Monte Carlo method by making use of antithetic variables. Here we draw the N independent samples in pairs effectively giving us double the amount of random paths for each N. This improvement only requires one additional line of code, which is shown in Listing (3). Analogously to Figure (4), Figure (5) shows the Monte Carlo approximation to the current portfolio value for $S_0 = X_1$ compared to the analytical result, along with the 95% confidence intervals, but this time using antithetic variables.

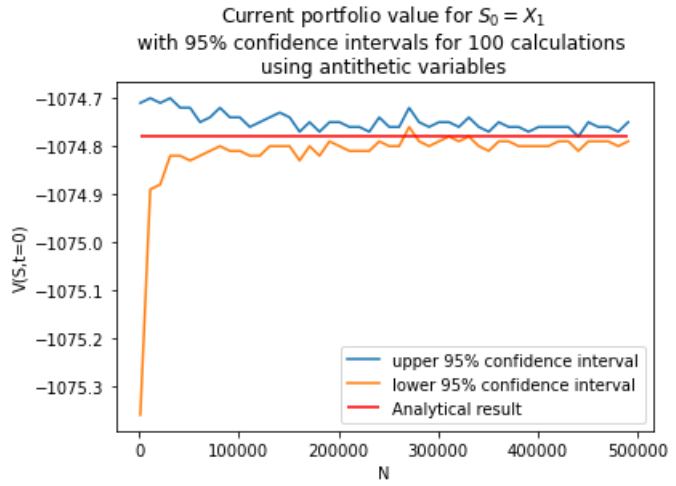

Figure 5: An image showing the the upper (solid blue line) and lower (solid orange line) 95% confidence intervals, using Monte Carlo simulation with antithetic variables, for an increasing number of paths N. The analytical solution (solid red line) is shown for comparison and has exact value $\Pi(S,T) = -1074.78$.

Looking at both of these figures, beyond $N \approx 200,000$ the convergence rate is significantly slower so choosing this value of N should offer good accuracy to computation speed ratio.

Table (1) shows the percentage error and computation time for both the standard Monte Carlo method and the method using antithetic variables for $N = 200,000$. We see that the standard method is approximately 20% faster for this calculation but using antithetic variables offers almost quadruple accuracy. This is not a general comment on the methods since it is only for one chosen N value and the methods both employ random numbers. This is explored more in Sub-section (1.7).

1.6. The Halton Sequence

To improve the Monte Carlo further we discard the notion of random numbers and instead draw them in a deterministic fashion which can improve the convergence by a factor of $\frac{1}{\sqrt{N}}$. First we generate two uniformly distributed numbers, x_1 and x_2, by choosing two prime numbers as bases, for example 2 and 3. The sequence is generated by writing the base 10 integers in the chosen basis, reflecting in the decimal point and then converting back to base 10. Now we pair up our uniform numbers and input them into the Box-Muller formulas [1]

$$y_1 = \cos(2\pi x_2)\sqrt{-2\log(x_1)} \qquad y_2 = \sin(2\pi x_1)\sqrt{-2\log(x_2)},$$

which generate numbers, y_1 and y_2, that are normally distributed. The code used to generate a Halton sequence of particular size given a prime basis is shown in Listing (4).

Figure (6) shows the Halton sequence approximation to the current portfolio value converging to the analytical result. Similarly to before, the convergence levels off around $N \approx 200,000$. The accuracy and efficiency for this N are show in Table (1) which suggests that the accuracy of this method is worse than antithetic variables at twice the computation time. Again however, comparison at one N is not a good metric and we explore this further in Sub-section (1.7).

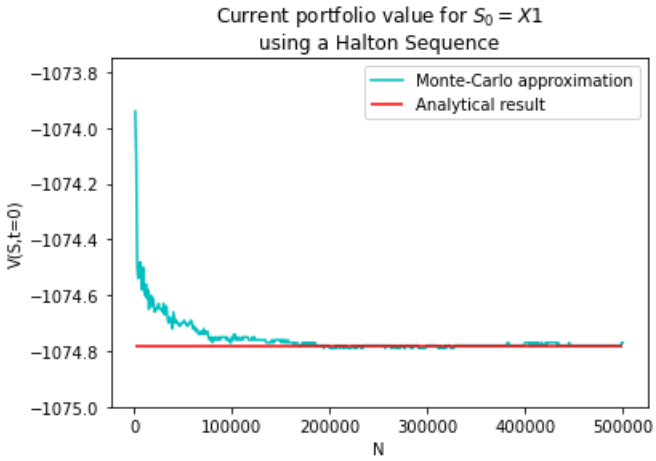

Figure 6: An image showing the current value of the portfolio calculated, for $S_0 = X_2$, using Monte Carlo simulation with a Halton sequence, for an increasing number of paths N (solid blue line). The analytical solution (solid red line) is shown for comparison and has exact value $\Pi(S,T) = -1074.78$.

We cannot calculate confidence intervals for this method as it deterministic. However, we can find the convergence rate, c, of the method using

$$V_N - V_{exact} = O\left(\frac{1}{N^c}\right). \tag{1.6.1}$$

Therefore a plot of $\ln(V_N - V_{exact})$ against $\ln(N)$ should give a straight line with gradient $-c$. Figure (7) shows such a plot with a best fit line, calculated using linear regression. The r^2 value for this linear regression is 0.83 so we are fairly confident in the linear relationship and the gradient is $c_{\text{Halton}} = 1.02$, which is very close to the true Halton convergence of 1 [2].

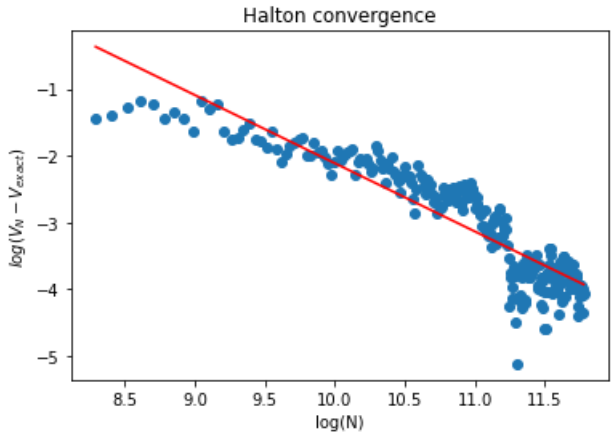

Figure 7: A plot of the logarithmic difference between our numerical approximation and the exact value, with the number of Monte Carlo paths. The relationship is approximately linear and the best bit line has gradient equal to the convergence of the Halton method, $c = 1.02$.

We can compare this to the standard Monte Carlo method, which gives $c_{\text{Standard}} = 0.48$, with the true value being 0.5 [2]. So the Halton sequence improves convergence by a factor $\frac{1}{\sqrt{N}}$.

1.7. Comparing methods

As mentioned when focusing on each method, a value of $N \approx 200,000$ appears to be a good choice for balancing accuracy and speed. Table (1) compares the methods for this value of N.

However, this single value of N doesn't paint a full picture. In figure (8) we plot the relative error for increasing N and see that the Halton sequence method is superior overall, despite the antithetic variable method randomly offering more accurate results at particular values.

We perform a similar investigation for the computation time by making plots of the different method times for increasing values of N as shown in Figure (9). The standard MC and antithetic method take approximately the same amount of time but the Halton sequence appears to take

Method	Percentage error	Computation time (s)	Upper confidence 95%	Lower confidence 95%
Standard MC	$2.412\,08 \times 10^{-5}$	0.0541109	-1074.82	-1074.69
Antithetic Variables	$6.615\,96 \times 10^{-6}$	0.0479431	-1074.80	-1074.75
Halton Sequence	$9.816\,18 \times 10^{-6}$	0.116226	N/A	N/A

Table 1: Comparison of accuracy and efficiency between three numerical methods: a standard Monte Carlo, a one using antithetic variables and the Halton sequence, all for $N = 200,000$.

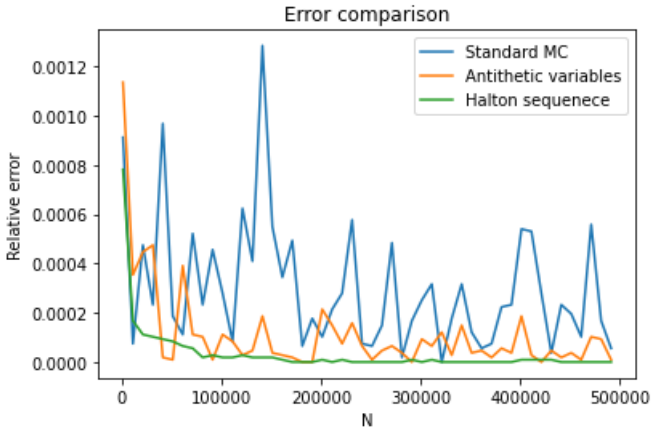

Figure 8: An image showing the relative errors of three numerical methods: a standard Monte Carlo, a one using antithetic variables and the Halton sequence, when compared with the analytic result.

just over twice as long. This is actually the behaviour we expect, as our standard random number generator method is $O(N)$ but the Halton sequence method is at least $O(2N)$. This is because we have to both generate the Halton sequence, which is at least $O(N)$, and then convert it using the Box-Muller method, which is $O(N)$.

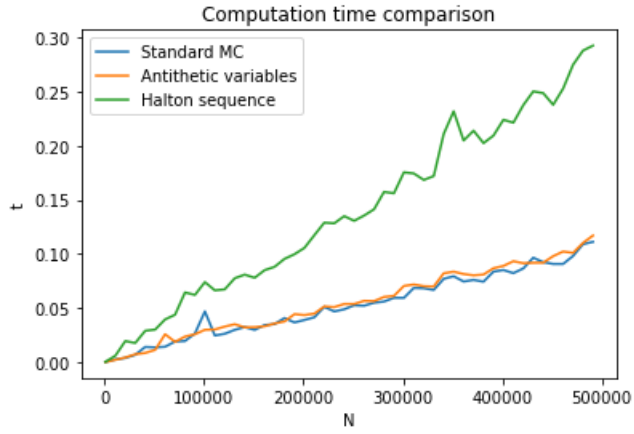

Figure 9: An image showing the computation time for an increasing number of paths, N, for three different methods: the standard Monte Carlo, a one using antithetic variables and the Halton sequence based method.

Overall, when performing a single calculation with a specific value of N, the Halton sequence is arguably the best method. Despite taking over twice the computation time, the accuracy is far more reliable. Using one of the other methods might speed up the calculation, and potentially allow you to use a higher N in a given time, but due to the pseudo-random nature of the methods, their accuracy is often several times worse than that of the Halton sequence.

2. Path dependent options

2.1. Valuing a floating strike Asian call option

We are tasked with pricing a discrete floating strike Asian call option. The function to perform this calculation is shown in Listing (5).

2.2. Exploring path dependence

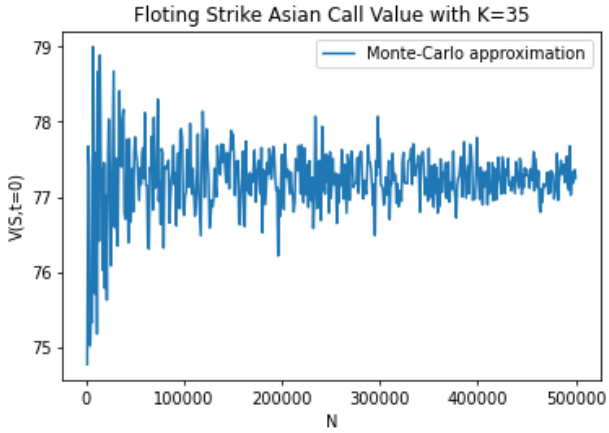

Figure 10: An image showing the current value of a floating strike Asian call option varying with the number of Monte Carlo paths. The parameters used were $S_0 = 900$, $T = 1.25$, $r = 0.03$, $D_0 = 0.04$ and $\sigma = 0.37$.

We can now explore how changing parameters in our path dependent problem, such as N and K, changes our results. Firstly, Figure (10) shows how our approximation to the option value varies as a function of N. As we increase N the value converges to $V(S, t = 0) \approx 77$, but the rate of convergence shows some diminishing returns past $N \approx 200,000$. So there may not be much benefit to using a value of N beyond this value.

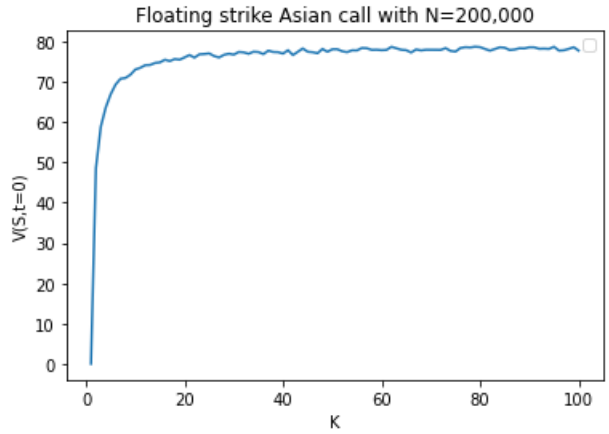

Figure 11: An image showing the current value of a floating strike Asian call option varying with the number of points on each stock path. The parameters used were $S_0 = 900$, $T = 1.25$, $r = 0.03$, $D_0 = 0.04$ and $\sigma = 0.37$.

Additionally, we can increase the number of points on our generated stock paths. We take $N = 200,000$ and then vary K as shown in Figure (11). We see that at very small values of K, the value of the option is wildly inaccurate, but rapidly increases and levels off around $V(S,T) \approx 77$, as before, for all $K \gtrsim 20$. Thus this seems to be the lower limit for the value of K we can choose. Higher values of K offer negligible accuracy increase and the computation time has linear dependence on K so those high values should be avoided to decrease the computation time.

2.3. Time limited valuation

We are tasked with obtaining the most accurate value of our floating strike Asian call option within a 10 seconds interval, fixing $K = 35$. Of course we want to maximise the computation time to get the most accurate result. For the non-path dependent case, we showed in Figure (9), that the computation time of all our methods was linear and the time for any one path only takes fractions of a second. Hence with a 10 second limit, we can likely use a very large N. Given this, the Halton sequence would be the preferable method, as it has a better convergence rate and is more accurate than pseudo-random number methods at high N. However, for ease of implementation, and the fact that the ability to use a higher N coming from having better than half the computation time may offset the methods flaws, we will use the antithetic variable method. The modified path dependent valuation function for use with antithetic variables is shown in Listing (6).

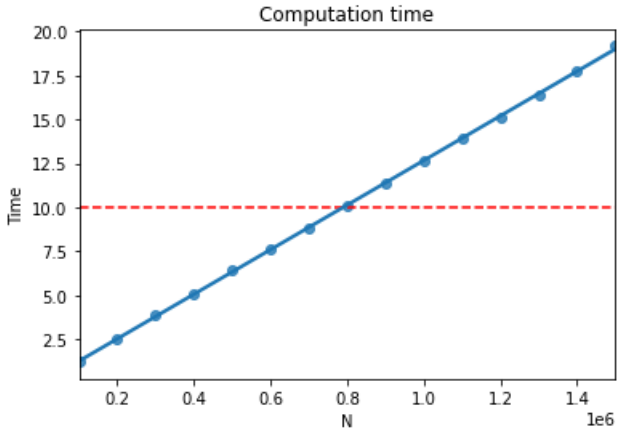

Figure 12: A plot showing the average time to value a floating strike Asian call option over 100 calculations varying with the number of Monte Carlo paths using antithetic variables. The best fit line and confidence intervals are determined using linear regression.

Since the antithetic variable method utilises random numbers it will produce a different best value of N every time, so averaging over multiple calculations will give a more reliable result. Figure (12) shows the computation time for various values of N for 100 calculations. The code used to generate this data is shown in Listing (7). We use linear regression to determine a best fit line along with 95% confidence intervals (Note that the confidence region is not actually visible as the data has an almost perfect linear relationship with an $r^2 = 0.9998$). Then we reverse engineer this to find that the value of N giving a calculation time of 10 seconds is $N = 788,096$.

Thus the most accurate average option value achievable in the time limit is $V(S, t = 0) = 77.2558$ and we are 95% confident that the most accurate value lies within the range $[77.2359, 77.2757]$.

3. Length and date

The number of words in this document is 1606
This document was submitted on 09/04/21 at 17:15

References

[1] G. E. Box, "A note on the generation of random normal deviates," *Ann. Math. Statist.*, vol. 29, pp. 610–611, 1958.

[2] R. E. Caflisch *et al.*, "Monte carlo and quasi-monte carlo methods," *Acta numerica*, vol. 1998, pp. 1–49, 1998.

A. Stock options code

Listing 1: C++ code showing the function used to perform a Monte Carlo simulation to value a portfolio

```
// perform monte carlo
double MonteCarlo(const double& initial_share_price, const double& interest_rate, const double& dividend_rate, const double& volatility,
    const double& expiration, const int& N, const int& put_number, const int& call_number, const int& binary_put_number,
    const int& binary_call_number, const int& zero_strike_call_number, const double& put_strike, const double& call_strike,
    const double& binary_put_strike, const double& binary_call_strike)
{
    // declare random number generator
    static std::mt19937 rng;

    // declare the normal distrubtion
    std::normal_distribution<double> ND(0., 1.);

    // initialise sum to 0
    double sum = 0;

    // run the simulations
    for (int i{ 0 }; i < N; i++) {

        // draw a random normally distributed number
        double phi = ND(rng);

        // get random value of stock value at maturity
        double final_share_price = initial_share_price * exp((interest_rate - dividend_rate - 0.5 * pow(volatility, 2)) * expiration + volatility * phi * pow(expiration, 0.5));

        // increment the sum
        sum += portfolio_payoff(put_number, call_number, binary_put_number, binary_call_number, zero_strike_call_number, put_strike,
            call_strike, binary_put_strike, binary_call_strike, final_share_price);
    }

    // output average over all paths
    return exp(-interest_rate * expiration) * sum / N;
}
```

Listing 2: C++ code showing how to calculate confidence intervals

```
// calculate the mean
    double sum_mean{ 0 };
    for (int i{ 0 }; i < samples.size(); i++) sum_mean += samples[i];
    double sample_mean = sum_mean / M;

    // calculate the variance
    double sum_var{ 0 };
```

```
    for (int i{ 0 }; i < samples.size(); i++) sum_var += pow(samples[i] - sample_mean, 2);
    double sample_variance = sum_var / (M - 1.);

    // calculate population mean and std
    double pop_mean = sample_mean;
    double pop_std = sqrt(sample_variance / M);

    // calculate confidence intervals
    double upper_95 = pop_mean + 2 * pop_std;
    double lower_95 = pop_mean - 2 * pop_std;
```

Listing 3: C++ code showing modification for a calculation using antithetic variables

```
// get random value of stock value at maturity
        double final_share_price_plus = initial_share_price * exp((interest_rate - dividend_rate - 0.5 * pow(volatility, 2)) *
             expiration + volatility * phi * pow(expiration, 0.5));
        double final_share_price_minus = initial_share_price * exp((interest_rate - dividend_rate - 0.5 * pow(volatility, 2)) *
             expiration - volatility * phi * pow(expiration, 0.5));
```

Listing 4: C++ code showing how to generate a Halton sequence of particular size given a prime basis.

```
// generate Halton sequence
std::vector<double> Halton_sequence(const int& basis, const int& size)
{
    // declare vector to return
    std::vector<double> Halton;

    // generate vector of size N
    for (int i{ 1 }; i <= size; i++) {

        // initialise variables
        double temp{ 1 };
        double Halton_number{ 0 };
        int index{ i };

        // calculate Halton number at index
        while (index > 0) {

            temp /= basis;
            Halton_number += temp * (index % basis);
            index /= basis;
        }

        // record the number
        Halton.push_back(Halton_number);
    }

    return Halton;
}
```

B. Path dependent options code

Listing 5: C++ code showing the function used to value a floating strike Asian call option

```
// value Asian call
double value_Asian_call(const double& initial_share_price, const double& interest_rate, const double& dividend_rate, const
     double& volatility,
    const double& expiration, const int& N, const double& K)
{
    // declare random number generator
    static std::mt19937 rnd;

    // declare the normal distrubtion
    std::normal_distribution<double> ND(0., 1.);

    // initalise sum to zero
    double sum{ 0 };

    // loop over all MC paths
    for (int i{ 0 }; i < N; i++) {

        // create a sample path
        double dt{ expiration / K };
        std::vector<double> stock_path;
        stock_path.push_back(initial_share_price);
        for (int i{ 1 }; i <= K; i++) {

            // generate random number
            double phi = ND(rnd);
```

```
25
26              // gemerate stock path
27              stock_path.push_back(stock_path[i - 1] * exp((interest_rate - dividend_rate -
28                  0.5 * pow(volatility, 2)) * dt + volatility * phi * pow(dt, 0.5)));
29          }
30
31          // calculate A
32          double A;
33          double A_sum{ 0 };
34          for (int i{ 1 }; i <= K; i++) A_sum += stock_path[i];
35          A = A_sum / K;
36
37          // add in the payoff
38          sum += std::max(stock_path.back() - A, 0.);
39      }
40
41      // average over all paths
42      return exp(-interest_rate * expiration) * sum / N;
43  }
```

Listing 6: C++ code showing the function used to value a floating strike Asian call option utilising antithetic variables

```
1   // value Asian call
2   double value_Asian_call(const double& initial_share_price, const double& interest_rate, const double& dividend_rate, const
        double& volatility,
3       const double& expiration, const int& N, const double& K)
4   {
5       // declare random number generator
6       static std::mt19937 rnd;
7
8       // declare the normal distrubtion
9       std::normal_distribution<double> ND(0., 1.);
10
11      // initalise sum to zero
12      double sum{ 0 };
13
14      // loop over all MC paths
15      for (int i{ 0 }; i < N; i++) {
16
17          // time step
18          double dt{ expiration / K };
19
20          // containers for stock path
21          std::vector<double> stock_path1;
22          std::vector<double> stock_path2;
23
24          // record initial values
25          stock_path1.push_back(initial_share_price);
26          stock_path2.push_back(initial_share_price);
27
28          // generate stock path
29          for (int i{ 1 }; i <= K; i++) {
30
31              // generate random number
32              double phi = ND(rnd);
33
34              // gemerate stock path
35              stock_path1.push_back(stock_path1[i - 1] * exp((interest_rate - dividend_rate -
36                  0.5 * pow(volatility, 2)) * dt + volatility * phi * pow(dt, 0.5)));
37              stock_path2.push_back(stock_path2[i - 1] * exp((interest_rate - dividend_rate -
38                  0.5 * pow(volatility, 2)) * dt - volatility * phi * pow(dt, 0.5)));
39          }
40
41          // calculate A
42          double A1, A2;
43          double A1_sum{ 0 }, A2_sum{ 0 };
44          for (int i{ 1 }; i <= K; i++) {
45              A1_sum += stock_path1[i];
46              A2_sum += stock_path2[i];
47          }
48          A1 = A1_sum / K;
49          A2 = A2_sum / K;
50
51          // add in the payoff
52          sum += std::max(stock_path1.back() - A1, 0.);
53          sum += std::max(stock_path2.back() - A2, 0.);
54      }
55
56      // average over all paths
57      return exp(-interest_rate * expiration) * sum / (2.*N);
58  }
```

Listing 7: C++ code showing the function used to value a floating strike Asian call option utilising antithetic variables

```cpp
double K{ 35 };  // points in the sample path

    // containers
    std::vector<double> N_store;
    std::vector<std::vector<double>> master_time_store;

    // loop over number of calculations
    for (int i{ 0 }; i < 100; i++) {

        std::vector<double> time_store;

        for (int N{ 100000 }; N <= 1500000; N += 100000) {

            if (i == 0) N_store.push_back(N);

            auto start = std::chrono::steady_clock::now();  // get start time
            double value = value_Asian_call(initial_share_price, interest_rate, dividend_rate, volatility, expiration, N, K);
            auto finish = std::chrono::steady_clock::now();  // get finish time
            auto elapsed = std::chrono::duration_cast<std::chrono::duration<double>>(finish - start);  // convert into seconds

            time_store.push_back(elapsed.count());
        }

        master_time_store.push_back(time_store);
    }

    // calculate average
    std::vector<double> average;
    for (int i{ 0 }; i < N_store.size(); i++) {

        double sum{ 0 };

        for (int j{ 0 }; j < master_time_store.size(); j++) {
            sum += master_time_store[j][i];
        }

        average.push_back(sum / master_time_store.size());
    }
```