MATH40082 (Computational Finance)
Main Assignment: Simulation Methods

10458528

March 2024

1 Theory

1.1 Stock Options

When the price of a stock is said to follow a risk-neutral distribution, we have that its price at a time t is given by

$$S_t \approx N(f(S_0,t), v^2(S_0,t)t) \tag{1}$$

where S_0 is the current stock price and $f(S_0,t)$ and $v(S_0,t)$ are functions calibrated for the specific context. For the financial contract to be valued in this report, the price at time t is given by

$$S_t = S_0(\cosh(2\beta T - \alpha T) - 1) + \theta(3 - e^{\alpha T} - e^{\beta T}) + \sigma(1+\alpha T)\frac{1}{2}(S_0+\theta)^{\gamma}\sqrt{T}\phi \tag{2}$$

where $\theta = 70100$, $\alpha = 0.01$, $\beta = 0.01$, $\gamma = 1.05$, $\sigma = 0.26$, $T = 2$ and

$$\phi = N(0,1). \tag{3}$$

If we consider a financial contract $C(S,T)$ written on the underlying stock S, which has a payoff given by

$$C(S,T) = g(S) = \begin{cases} X_2 - S_T & \text{if } S_T < X_1 \\ S_T - X_2 & \text{if } X_1 \le S_T < X_2 \\ X_1 - S_T & \text{if } S_T \ge X_2 \end{cases} \tag{4}$$

Then the analytical solution for any given payoff is given by the numerical integration

$$C(S_0, t=0) = \frac{e^{-rT}}{v\sqrt{2\pi T}}\int_{-\infty}^{\infty} g(z)\exp\left[-\frac{(z-f)^2}{2v^2T}\right]dz \tag{5}$$

We can carry out a Monte-Carlo valuation for the option by sampling from the normal distribution ϕ and calculating the stock price several times, with the i^{th} stock price given by

$$S_T^i = f(S_0,T) + v(S_0,T)\sqrt{T}\phi_i \tag{6}$$

We then average over the n calculated stock prices to approximate the value of the financial contract as

$$C(S_0, t=0) \approx e^{-rT}\frac{1}{n}\sum_{i=1}^{n} g(S_T^i) \tag{7}$$

1.2 Path Dependent Options

Assuming that the risk-neutral stochastic process follows the SDE

$$ds = f(S,t)dt + v(S,t)dW. \tag{8}$$

A path-dependent option depends on all of the share prices, $S(t_k)$, at $K+1$ equally spaced sampling times $t_0, t_1, ..., t_k = k\Delta t, ..., t_K$ with $t_0 = 0$, $t_K = T$ and

$$\Delta t = \frac{T}{K}. \tag{9}$$

In this report, the SDE in equation (8) takes the specific form

$$ds = (\alpha\theta - \beta S)dt + \sigma(|S|)^{\gamma}dW, \tag{10}$$

where $\theta = 70100$, $\alpha = 0.01$, $\beta = 0.01$, $\gamma = 1.05$, $\sigma = 0.26$, $T = 1$.
Since the options are path-dependent, We must approximate the price at each of K time steps between the initial time and the time of maturity T. If the time step is given by ΔT, then the share price at each discreet point in time is given by

$$S^i(t_k) = S^i(t_{k-1}) + f(S^i(t_{k-1}), t_{k-1})\Delta t + v(S^i(t_{k-1}), t_{k-1})\sqrt{\Delta t}\phi_{i,k-1} \tag{11}$$

In this report, we value a minimum floating strike lookback call option, the payoff, G, is found by first calculating the minimum share price

$$A = \min_k S(t_k) \tag{12}$$

and then

$$G(S, A) = \max(S - A, 0) \tag{13}$$

To calculate the value of the contract at $t = 0$ using Monte Carlo simulation, we average over n approximations and apply a discounting factor to get

$$C(S_0, t = 0) \approx e^{-rT} \frac{1}{n} \sum_{i=1}^{n} G(S^i, A) \tag{14}$$

2 Results

2.1 Stock Options

The analytical solution to the value of the first financial contract was calculated from equation (5) using numerical quadrature using functions from the Python package SciPy. This value was found to be 7784.1000 with an error of 0.01661. Using a Monte-Carlo simulation, utilising equation (6), the value of the financial contract was found to be 7734.2 $\pm$ 51.6 when the number of simulated stock prices was $N = 1000000$, in good agreement with the analytical solution.

To investigate how the number of simulated stock prices affects the accuracy of the Monte-Carlo simulation, the value of the financial contract was calculated for a range of numbers of simulated stock prices N. Figure 1 below shows how the value of the financial contract converges to the analytical value with increasing N.

As a result of the central limit theorem, the error of the Monte-Carlo simulation is expected to scale proportionally to $N^{-1/2}$. To demonstrate this, The plot in Figure 2 was created which shows the standard deviation against $N^{-1/2}$ and a linear fit was calculated to follow the relationship of $error = 51000N^{-1/2} + 4.99$.

There exist a few methods by which the standard Monte-Carlo method can be improved that I investigated for this report. Firstly, Antithetic variables were used by calculating the stock price with the randomly sampled ϕ of equation (3) as well as its negative $-\phi$ to ensure the distribution is centred around zero. This is expanded upon by moment matching, in which each of the ϕ and $-\phi$ values sampled are divided by the square root of the variance of the entire sample to ensure the sample also has a variance of one. if N values of ϕ are sampled for these two methods, the resulting final sample will have $2N$ samples. Finally, The Halton Sequence was used to create a set of coordinates (x_1, x_2), uniformly distributed within the unit square. This process is started by selecting two prime numbers a and b, before representing a series of numbers in the base a^{-1} to give the x_1 values and b^{-1} to get the x_2 values. A set of $2N$ normally distributed numbers can be created from N (x_1, x_2) coordinates using the Box-Muller method with the equations

$$y_1 = \cos(2\pi x_2)\sqrt{-2\log(x_1)},\ y_2 = \cos(2\pi x_1)\sqrt{-2\log(x_2)} \tag{15}$$

These three methods were compared to the original Monte-Carlo method by using them, in turn, to calculate the value of the financial contract for a range of values of N. The results of these simulations can be seen in Figure 3. It can be seen from Figure 3 that the introduction of the extra methods dramatically increases the efficiency of the Monte-Carlo simulations, with all three methods showing a similar level of improvement.

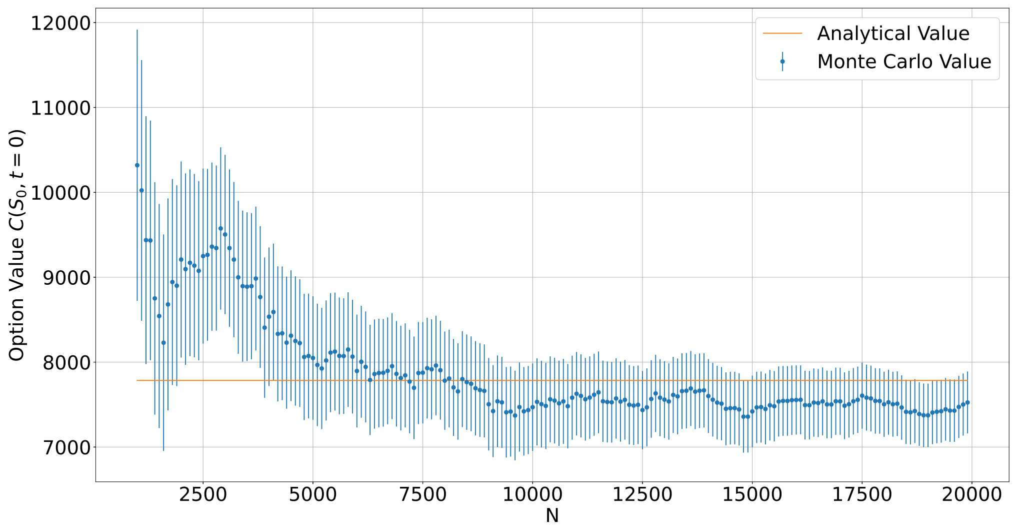

Figure 1: A plot of the option value at t=0 for a range of values of the number of sample paths N, with error bars showing the standard deviation of the simulations ran for each value of N.

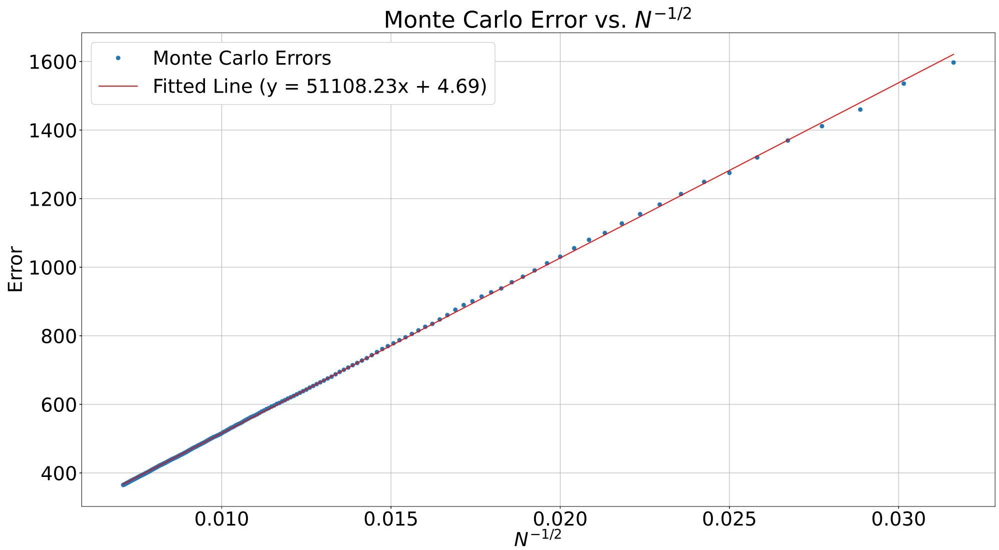

Figure 2: A plot of the standard deviation of the option values against the reciprocal of the square root of the number of sample paths. along with a linear fit.

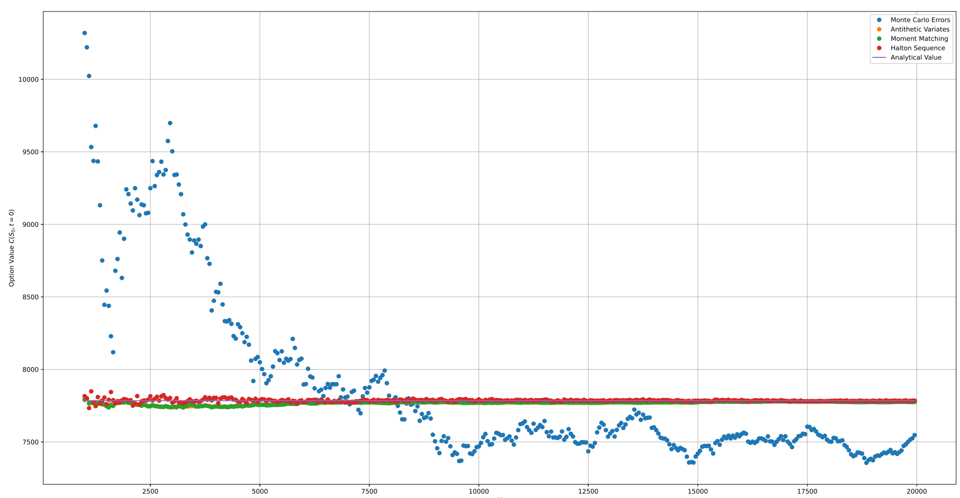

Figure 3: A plot of the option value for a range of values of the number of sample paths N for the difference Monte-Carlo techniques

Figure 4 shows a plot of the errors against $N^{-1/2}$ for each of the methods over a range of values of N. For each method, a linear fit has been calculated. All three improvement methods have very similar convergence rates, however at small N, moment matching and Halton sequence methods slightly outperform the antithetic variables method.

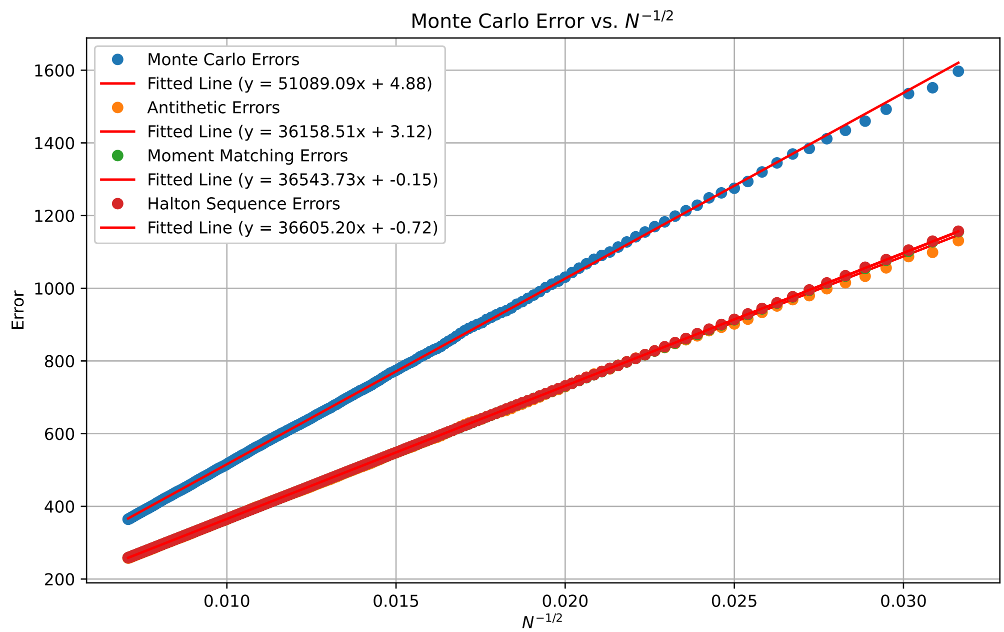

Figure 4: A plot of the error for the option value against $N^{-1/2}$ for the different techniques.

I also investigated the difference in time taken for each of the Monte-Carlo methods to run for a range of values of N. As can be seen, the addition of Antithetic variables and Moment Matching doubled the time taken to complete the Monte-Carlo simulation, whilst the Halton method significantly increased the time taken.

N	Standard Time	Antithetic Time	Moment-Matching Time	Halton Time
10000	0.400	0.770	0.772	4.466
20000	0.788	1.565	1.534	9.681
30000	1.169	2.419	2.411	15.13
40000	1.619	3.263	3.376	20.33

Table 1: A table showing the time taken in seconds for each of the Monte-Carlo methods to run for a range of values of N.

2.2 Path Dependent Options

For the discrete minimum floating-strike lookback call option the value calculated for t=0 through the standard Monte Carlo simulation was $13898.1 \pm 15.693966670393191$. This was calculated with $N = 1000000$ simulations with the stock price observed at $k = 30$ equally spaced times. There is no analytical solution to the minimum floating-strike lookback call option with which to compare the results of the Monte-Carlo simulation.

I ran a Monte-Carlo simulation to value the path-dependent option for a range of values of N. The results of this analysis are shown below in Figure 5.

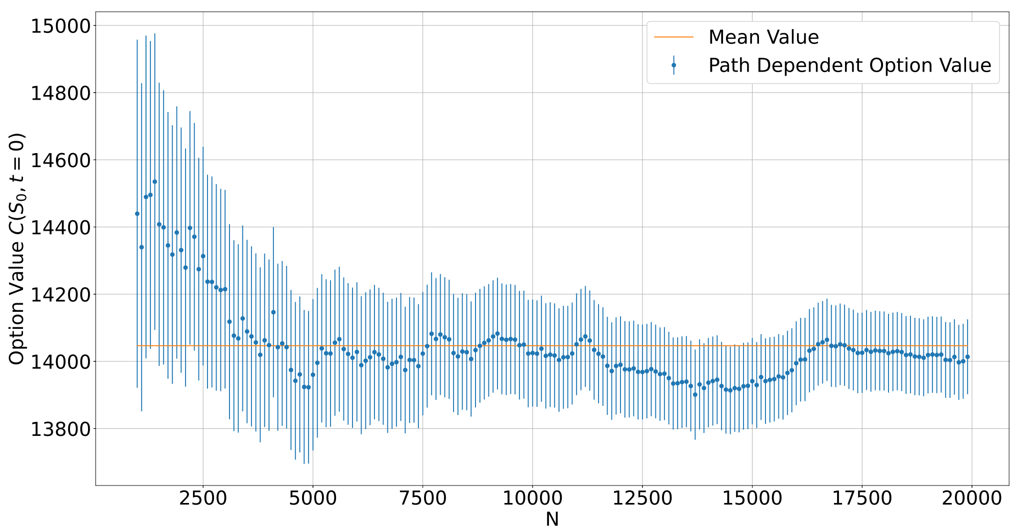

Figure 5: A plot of the Asian put option value at t=0 for a range of values of the number of sample paths N as well as the mean of all the values

Figure 6 shows that the relation between the standard deviation and $N^{-1/2}$ is also linear. It was found that the relation was given by $error = 16603N^{-1/2} - 7.04$.

Finally, I calculated the derivative of the minimum floating-strike lookback call option with respect to α using the formula

$$\frac{dV}{d\sigma} \approx \frac{V(S_0, t=0; \sigma = 0.26 + d\sigma) - V(S_0, t=0, \sigma = 0.26)}{d\sigma}. \tag{16}$$

The best estimate for the derivative is when $d\sigma$ approaches zero, so I plotted the value of the derivative for a range of values of $d\sigma$ and calculated a linear fit, as can be seen in Figure 7. Taking the derivative axis intercept to be equal to the derivative suggests that the derivative has the value $dV/d\sigma = 49290.6$

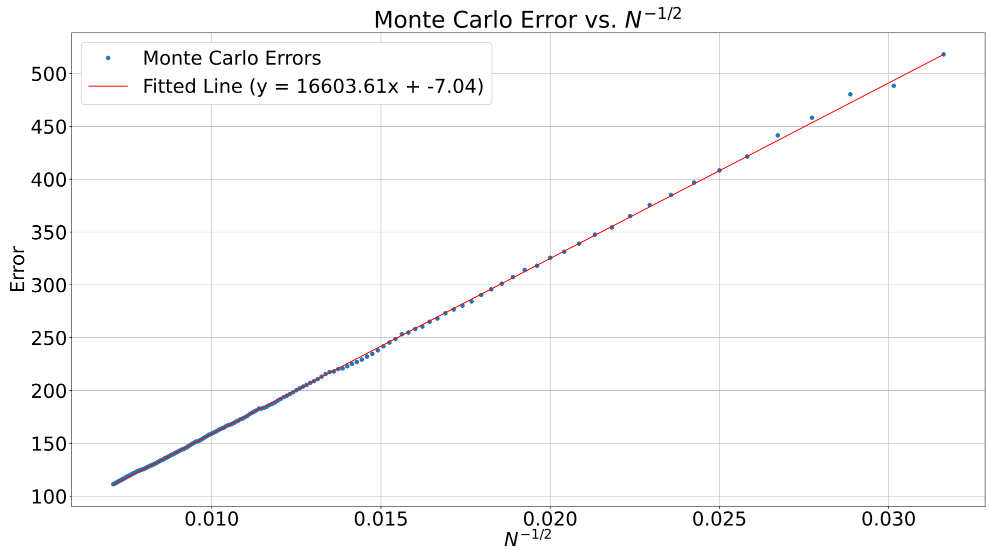

Figure 6: A plot of the standard deviation of the path dependent option values against the reciprocal of the square root of the number of sample paths

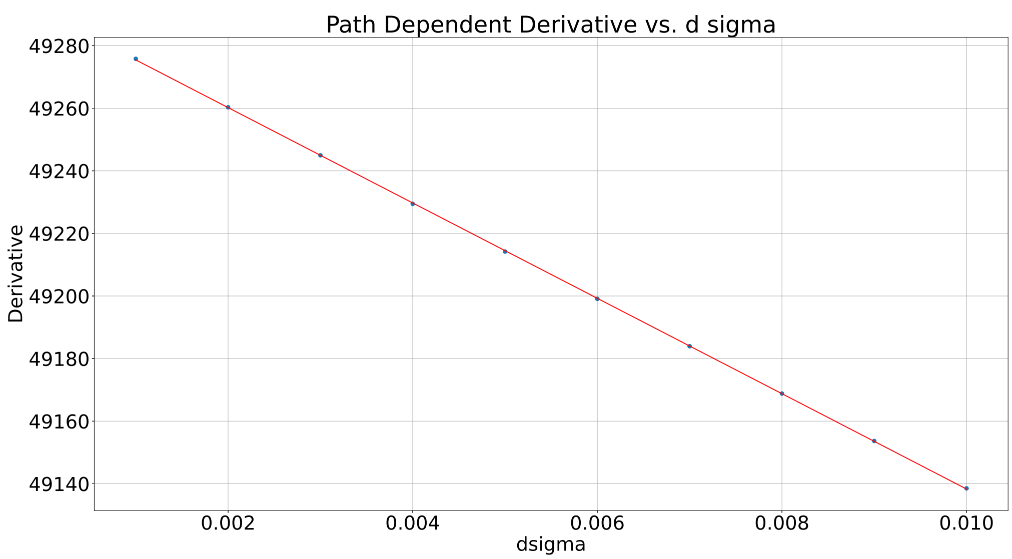

Figure 7: A plot of the derivative with respect to $d\sigma$, as calculated by the finite difference method for a range of $d\sigma$.

Appendix A Source code listings

A.1 `main_task_1.py`

```python
import numpy as np
from scipy.integrate import quad as QUAD
import matplotlib.pyplot as plt
from scipy.stats import linregress
from timeit import timeit

plt.rcParams.update({'font.size': 30})

t = 0
S_0 = 70154
r = 0.01
X_1 = 70000
X_2 = 80000
theta = 70100
alpha = 0.01
beta = 0.01
gamma = 1.05
sigma = 0.26

def g(S):
    if S<X_1:
        return X_2 - S
    elif S < X_2:
        return S-X_2
    else:
        return X_1 - S

def g_vectorised(S):
    result_list = [X_2 - s if s < X_1 else s - X_2 if s < X_2 else X_1 - s for s in S]
    return np.array(result_list)

def f(S_0, T):
    return S_0*(np.cosh(2*beta*T - alpha*T) - 1) + theta*(3-np.exp(alpha*T) - np.exp(beta*T))

def v(S_0, T):
    return sigma*(1+alpha*T)*(1/2)*(S_0+theta)**gamma

def calculate_analytical_value():
    C_integrand = lambda z: g(z)*np.exp(-1 * (z-f(S_0, T))**2 / (2*(v(S_0, T)**2)*T))
    I1 = QUAD(C_integrand , -9900000.0, X_1)
    I2 = QUAD(C_integrand , X_1, X_2)
    I3 = QUAD(C_integrand , X_2, 9900000.0)
    V_exact = np.exp(-r*T) * (I1[0]+I2[0]+I3[0]) /  (v(S_0, T)*np.sqrt(2*np.pi*T))
    print(" V_exact:=",V_exact," with error ", I1[1]+I2[1]+I3[1])
    return V_exact

def monte_carlo(N):
    rng = np.random.default_rng(seed=0)
    phi = rng.normal(0.0, 1.0, size=(N))

    S_T = f(S_0, T) + v(S_0, T) * np.sqrt(T) * phi
    mean_vals = np.mean(g_vectorised(S_T))
```

```python
    error = np.sqrt(np.var(g_vectorised(S_T))) / np.sqrt(N)

    return mean_vals * np.exp(-r * T), error

def monte_carlo_antithetic(N):
    rng = np.random.default_rng(seed=0)
    phi = rng.normal(0.0, 1.0, size=(N))

    # Generate antithetic variates
    antithetic_phi = np.concatenate([phi, -phi])

    # Compute S_T for both phi and -phi
    S_T = f(S_0, T) + v(S_0, T) * np.sqrt(T) * antithetic_phi

    # Apply the vectorized g function to the entire S_T array
    sum_vals = np.mean(g_vectorised(S_T))
    error = np.sqrt(np.var(g_vectorised(S_T))) / np.sqrt(2*N)

    return sum_vals * np.exp(-r * T), error

def monte_carlo_moment_matching(N):
    rng = np.random.default_rng(seed=0)

    # Generate phi and adjust for moment matching
    phi = rng.normal(0.0, 1.0, size=(N))
    phi_variance = np.var(phi)
    adjusted_phi = phi / np.sqrt(phi_variance)

    # Generate antithetic variates for the adjusted phi
    antithetic_phi = np.concatenate([adjusted_phi, -adjusted_phi])

    # Compute S_T for both adjusted_phi and -adjusted_phi
    S_T = f(S_0, T) + v(S_0, T) * np.sqrt(T) * antithetic_phi

    # Apply the vectorized g function to the entire S_T array
    sum_vals = np.mean(g_vectorised(S_T))
    error = np.sqrt(np.var(g_vectorised(S_T))) / np.sqrt(2*N)

    return sum_vals * np.exp(-r * T), error

def halton_sequence(i, base):
    f = 1
    r = 0
    while i > 0:
        f = f / base
        r = r + f * (i % base)
        i = int(i / base)
    return r

def halton_vector_func(a, b, N):
    halton_vector = np.zeros((N, 2))

    for i in range(N):
        halton_vector[i][0] = halton_sequence(i+1, a)
        halton_vector[i][1] = halton_sequence(i+1, b)
    return halton_vector
```

```

def box_muller_vector(halton_vector):
    x1 = halton_vector[:, 0]
    x2 = halton_vector[:, 1]

    r = np.sqrt(x1**2 + x2**2)

    transformed_1 = np.cos(2 * np.pi * x2) * np.sqrt(-2 * np.log(x1))
    transformed_2 = np.sin(2 * np.pi * x1) * np.sqrt(-2 * np.log(x2))

    # Interleave transformed_1 and transformed_2
    box_muller_vector = np.empty((x1.size * 2,), dtype=x1.dtype)
    box_muller_vector[0::2] = transformed_1
    box_muller_vector[1::2] = transformed_2

    return box_muller_vector

def monte_carlo_halton(a, b, N):
    # Generate Halton sequence
    h_vector = halton_vector_func(a, b, N)
    # Apply Box-Muller transform vectorized
    phi_values = box_muller_vector(h_vector)

    # Calculate S_T using vectorized operations
    S_T = f(S_0, T) + v(S_0, T) * np.sqrt(T) * phi_values

    # Apply the vectorized g function to S_T and calculate the mean
    mean_val = np.mean(g_vectorised(S_T))
    error = np.sqrt(np.var(g_vectorised(S_T))) / np.sqrt(2*N)

    # Adjust the result for discounting and return
    return mean_val * np.exp(-r * T), error

def path_dependent_option(N, k, da=None):
    rng = np.random.default_rng(seed=0)
    phi = np.array(rng.normal(0.0,1.0,size=((N,k))))
    sig = sigma + da if da is not None else sigma
    delta_t = T/k

    S_t = np.zeros((N,k))
    S_t[:, 0] = S_0

    delta_t = T/k

    for i in range(k-1):
        S_t[:, i+1] = S_t[:,i] + (alpha*theta - beta*S_t[:,i])*delta_t + sig*np.abs(S_t[:,i])**gamma*np.sqrt(delta_t)*phi[:, i]

    min_values = np.min(S_t[:, 1:], axis=1)
    payout = np.maximum(0, S_t[:, -1] - min_values)

    value = np.exp(-r * T) * np.mean(payout)
    error = np.sqrt(np.var(payout)) / np.sqrt(N)
    return value, error

def path_dependent_derivative(N, k, da):
```

```
    derivative = (path_dependent_option(N, k, da)[0] - path_dependent_option(N, k)[0]) / da
    return derivative

def path_dep_N_analysis():
    N_values = np.arange(1000, 20000, 100)
    path_dep_vals = np.zeros(N_values.size)
    path_dep_errors = np.zeros(N_values.size)


    for i, N in enumerate(N_values):
        print(i)
        path_dep_vals[i], path_dep_errors[i] = path_dependent_option(N, 30)

    mean = np.mean(path_dep_vals)
    print(f"path dependent mean: {mean}")

    plt.errorbar(N_values, path_dep_vals, yerr=path_dep_errors, fmt='o', label="Path Dependent Option Value")
    plt.plot(N_values, [mean] * N_values.size, label="Mean Value")
    plt.xlabel("N")
    plt.ylabel("Option Value $C(S_0, t=0)$")
    plt.grid()
    plt.legend()
    plt.show()

    plt.errorbar(N_values, path_dep_errors, fmt='o', label="Path Dependent errors")
    plt.xlabel("N")
    plt.ylabel("error")
    plt.grid()
    plt.legend()
    plt.show()

    errror_analysis(N_values, path_dep_errors)

def N_anaylsis(analytical_value):

    N_values = np.arange(1000, 20000, 100)
    monte_carlo_vals = np.zeros(N_values.size)
    monte_carlo_errors = np.zeros(N_values.size)
    monte_carlo_anti_vals = np.zeros(N_values.size)
    monte_carlo_anti_errors = np.zeros(N_values.size)
    monte_carlo_moment_vals = np.zeros(N_values.size)
    monte_carlo_moment_errors = np.zeros(N_values.size)
    monte_carlo_halton_vals = np.zeros(N_values.size)
    monte_carlo_halton_errors = np.zeros(N_values.size)


    for i, N in enumerate(N_values):
        print(i)
        monte_carlo_vals[i], monte_carlo_errors[i] = monte_carlo(N)
        monte_carlo_anti_vals[i], monte_carlo_anti_errors[i] = monte_carlo_antithetic(N)
        monte_carlo_moment_vals[i], monte_carlo_moment_errors[i] = monte_carlo_moment_matching(N)
```

```
            monte_carlo_halton_vals[i], monte_carlo_halton_errors[i] = monte_carlo_halton(2, 3, N)

    plt.errorbar(N_values, monte_carlo_vals, yerr=monte_carlo_errors, fmt='o', label="Monte Carlo Value")
    plt.plot(N_values, [analytical_value] * N_values.size, label="Analytical Value")
    plt.xlabel("N")
    plt.ylabel("Option Value $C(S_0, t=0)$")
    plt.grid()
    plt.legend()
    plt.show()

    plt.plot(N_values, monte_carlo_vals, 'o', label="Monte Carlo Errors")
    plt.plot(N_values, monte_carlo_anti_vals, 'o', label="Antithetic Variates")
    plt.plot(N_values, monte_carlo_moment_vals, 'o', label="Moment Matching")
    plt.plot(N_values, monte_carlo_halton_vals, 'o', label="Halton Sequence")
    plt.plot(N_values, [analytical_value] * N_values.size, label="Analytical Value")
    plt.xlabel("N")
    plt.ylabel("Option Value $C(S_0, t=0)$")
    plt.grid()
    plt.legend()
    plt.show()

    errror_analysis(N_values, monte_carlo_errors)
    multi_error_analysis(N_values, monte_carlo_errors, monte_carlo_anti_errors, monte_carlo_moment_errors, monte_carlo_halton_errors)

def errror_analysis(N_values, errors):
    N_inv_sqrt = N_values**-0.5

    slope, intercept, _, _, _ = linregress(N_inv_sqrt, errors)
    fitted_errors = slope * N_inv_sqrt + intercept

    plt.figure(figsize=(10, 6))
    plt.plot(N_inv_sqrt, errors, 'o', label="Monte Carlo Errors")
    plt.plot(N_inv_sqrt, fitted_errors, 'r', label=f"Fitted Line (y = {slope:.2f}x + {intercept:.2f})")

    plt.xlabel("$N^{-1/2}$")
    plt.ylabel("Error")
    plt.title("Monte Carlo Error vs. $N^{-1/2}$")
    plt.legend()
    plt.grid(True)
    plt.show()

def multi_error_analysis(N_values, monte_errors, anti_errors, moment_errors, halton_errors):
    N_inv_sqrt = N_values**-0.5

    monte_slope, monte_intercept, _, _, _ = linregress(N_inv_sqrt, monte_errors)
    monte_fitted_errors = monte_slope * N_inv_sqrt + monte_intercept

```

```python
    anti_slope, anti_intercept, _, _, _ = linregress(N_inv_sqrt, anti_errors)
    anti_fitted_errors = anti_slope * N_inv_sqrt + anti_intercept

    moment_slope, moment_intercept, _, _, _ = linregress(N_inv_sqrt, moment_errors)
    moment_fitted_errors = moment_slope * N_inv_sqrt + moment_intercept

    halton_slope, halton_intercept, _, _, _ = linregress(N_inv_sqrt, halton_errors)
    halton_fitted_errors = halton_slope * N_inv_sqrt + halton_intercept

    plt.figure(figsize=(10, 6))
    plt.plot(N_inv_sqrt, monte_errors, 'o', label="Monte Carlo Errors")
    plt.plot(N_inv_sqrt, monte_fitted_errors, 'r', label=f"Fitted Line (y = {monte_slope:.2f}x + {monte_intercept:.2f})")

    plt.plot(N_inv_sqrt, anti_errors, 'o', label="Antithetic Errors")
    plt.plot(N_inv_sqrt, anti_fitted_errors, 'r', label=f"Fitted Line (y = {anti_slope:.2f}x + {anti_intercept:.2f})")

    plt.plot(N_inv_sqrt, moment_errors, 'o', label="Moment Matching Errors")
    plt.plot(N_inv_sqrt, moment_fitted_errors, 'r', label=f"Fitted Line (y = {moment_slope:.2f}x + {moment_intercept:.2f})")

    plt.plot(N_inv_sqrt, halton_errors, 'o', label="Halton Sequence Errors")
    plt.plot(N_inv_sqrt, halton_fitted_errors, 'r', label=f"Fitted Line (y = {halton_slope:.2f}x + {halton_intercept:.2f})")

    plt.xlabel("$N^{-1/2}$")
    plt.ylabel("Error")
    plt.title("Monte Carlo Error vs. $N^{-1/2}$")
    plt.legend()
    plt.grid(True)
    plt.show()

def time_analysis(N):
    script = f"monte_carlo({N})"
    codeRuns = 100
    timeSimulate = timeit( script,number=codeRuns,globals=globals() )

    print("Time taken to run ",codeRuns," simulations with", N, "paths is", timeSimulate," seconds.")

    script = f"monte_carlo_antithetic({N})"
    timeSimulate = timeit( script,number=codeRuns,globals=globals() )

    print("Time taken to run ",codeRuns," simulations with", N, "paths is", timeSimulate," seconds.")

    script = f"monte_carlo_moment_matching({N})"
    timeSimulate = timeit( script,number=codeRuns,globals=globals() )

    print("Time taken to run ",codeRuns," simulations with", N, "paths is", timeSimulate," seconds.")

    script = f"monte_carlo_halton(2, 3, {N})"
```

```
    timeSimulate = timeit( script,number=codeRuns,globals=globals() )

    print("Time taken to run ",codeRuns," simulations with", N, "paths is", timeSimulate," seconds.")

def derivative_analysis(n_alpha):
    derivative_values = np.zeros(n_alpha)
    da_values = np.zeros(n_alpha)

    for i in range(n_alpha):
        da = 0.001 + 0.01/n_alpha * i
        da_values[i] = da
        derivative_values[i] = path_dependent_derivative(100000, 30, da)

    da_slope, da_intercept, _, _, _ = linregress(da_values, derivative_values)
    da_fitted_errors = da_slope * da_values + da_intercept
    print(da_intercept)

    plt.plot(da_values, derivative_values, 'o')
    plt.plot(da_values, da_fitted_errors, 'r')
    plt.xlabel("da")
    plt.ylabel("Derivative")
    plt.title("Path Dependent Derivative vs. da")
    plt.grid(True)
    plt.show()


if __name__ == "__main__":

    T= 2
    analytical_value = calculate_analytical_value()
    N_anaylsis(analytical_value)

    for i in range(1, 5):
        N = i*10000
        time_analysis(N)

    T= 0.5
    path_dep_N_analysis()
    derivative_analysis(10)
    print(path_dependent_option(1000000, 30)[0])
    print(path_dependent_option(1000000, 30)[1])
    print(path_dependent_derivative(100000, 30, 0.0001))
```