Monte Carlo Methods with Financial Applications Assignment 2

Viktor Ågren

January 2022

Problem 1.

Generate a three-dimensional Halton sequence with bases 2, 3 and 5 respectively. Use it to determine quasi-Monte Carlo approximations to the integral

$$\int_0^1 \int_0^1 \int_0^1 \frac{xy}{\sqrt{x^2+y^2+z^2}} dxdydz$$

for different sequence lengths. Compare the convergence to the true numerical value 0.2332969032. . . and to classical Monte Carlo estimates

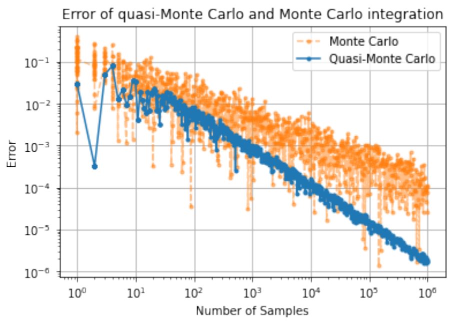
Implementation The `halton` function computes the Halton sequence for a given number of elements and base. The `f` function is the function that will be integrated using the quasi-Monte Carlo method. The `qmc` function computes the quasi-Monte Carlo integration of $f(x, y, z)$ using a given number of samples. The function takes one argument, n, which specifies the number of samples to use. It then uses the `halton` function to generate three sequences of n elements each, using bases 2, 3, and 5. The function then computes the sum of $f(x, y, z)$ evaluated at these points and returns the average value of this sum. This average value is an approximation of the integral of $f(x, y, z)$ over the domain $[0, 1]$ in all three dimensions. The qmc function can be represented as

$$qmc(n) = \frac{1}{n}\sum_{i=1}^{n} f(x_i, y_i, z_i) \qquad = \frac{1}{n}\sum_{i=1}^{n} \frac{x_i y_i}{\sqrt{x_i^2+y_i^2+z_i^2}}$$

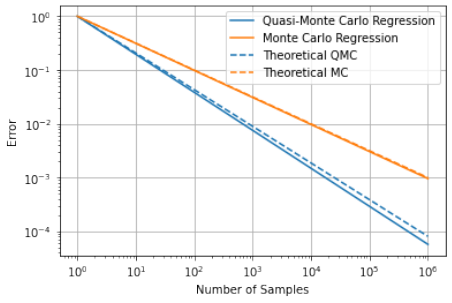
where (x_i, y_i, z_i) is the i'th sample of the quasi-Monte Carlo sequence generated using the `halton` function.

Figure 1a presents a comparison of the error approximations of the Monte Carlo and quasi-Monte Carlo methods to the true numerical value. The quasi-Monte Carlo method, which incorporates the Halton sequence, demonstrates a lower variance over the range of the simulation and a more accurate approximation to the true numerical value. This indicates that the quasi-Monte Carlo method is more reliable for estimating the true numerical value, as it exhibits a higher degree of stability compared to the Monte Carlo method.

In Figure 1b, we plot the linear regression of the individual methods and compare them to their respective theoretical convergence rates, which are expressed in terms of Big O notation. The Monte Carlo method has a theoretical estimation error order of $O(n^{-\frac{1}{2}})$, which is consistent with the simulated convergence observed in the plot. The quasi-Monte Carlo method, on the other hand, is expected to have an estimation error order of $O(\frac{(\log n)^d}{n})$. In this particular case, the simulated convergence of the quasi-Monte Carlo method is actually slightly better than the theoretical expectation. This suggests that the quasi-Monte Carlo method may be more efficient than expected at estimating the true numerical value in this particular scenario. More likely, the estimation was better because of the number of samples utilized and this is more of a result of *luck*.



(a) Error analysis



(b) Linear regression

Figure 1: Comparison of Monte Carlo error and quasi-Monte Carlo error

Problem 2.

Use the basic Metropolis-Hastings algorithm with a random walk sampler to simulate a sample of 10000 values from the distribution given by the probability density function f/C where $f(x) = x^3 \sin(\pi x) on[-1, 1]$ and $C = 2/\pi - 12/\pi^3$.

To generate the candidate values, use a density function $g(y|x)$ that is the density of a normal distribution with mean x and fixed variance σ^2. Consider different choices of σ and describe the effect.

Implementation The Metropolis-Hastings algorithm with a a random walk sampler is a Markov Chain Monte Carlo (MCMC) method for generating samples from a target distribution. The general algorithm works by proposing new candidate samples from the current sample using a proposal distribution $q(x|y)$, and then accepting or rejecting the proposal based on the acceptance probability defined as:

$$\alpha(x, y) = \min\left(1, \frac{f(y)q(x|y)}{f(x)q(y|x)}\right)$$

Where $f(x)$ is the target distribution and $q(x|y)$ is the proposal distribution [1]. The idea is that the samples generated by the algorithm will eventually converge to samples from the target distribution. The specific implementation of the proposal distribution in this case is a normal distribution with mean x and standard deviation σ, where σ is an input parameter.

In the code, the target distribution is defined as $f(x) = x^3 \sin(\pi x)$, and the `metropolis_hastings` function uses the acceptance probability to generate a sequence of samples using the target and proposal distributions. The number of samples to generate, and the standard deviation of the normal distribution, are passed as input arguments.

The figures 2 and 3 generated by the code are the simulation of the Metropolis-Hastings algorithm using different values of σ. The first figure shows the results for each σ value and provide a sense of the movement of the samples through the parameter space during the simulation. The second figure shows the histograms of the samples generated by the simulation for each σ this can help to see if the samples are converging to the target distribution.

Larger values of σ are interpreted as larger movements in the random walk. As the step size represented by σ increases, the sampling is able to explore a larger region of the parameter space with each step. However, as the variance of the walk increases, the samples generated by the algorithm are less likely to be concentrated around the peak of the target distribution, making the algorithm less efficient. As a result, for small σ values, the theoretical simulation shows less movement and a more concentrated distribution, while as σ increases the variance of the walk increases and the distribution becomes less concentrated.

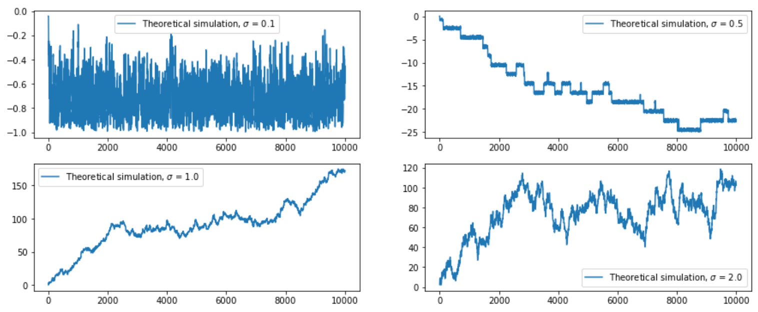


Figure 2: The Metropolis-Hastings algorithm with a random walk sampler for different choice of σ.

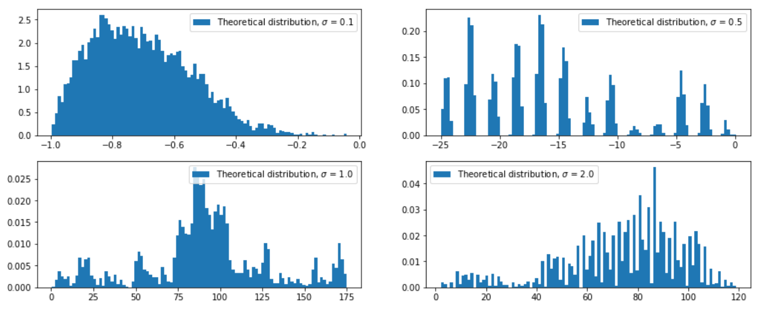


Figure 3: Probability density function for the Metropolis-Hastings algorithm with a random walk sampler for different choice of σ.

Problem 3.

Apply Monte Carlo simulation to estimate numerically delta and vega (derivatives of the option price with respect to initial stock price and volatility, respectively) for the following three types of options, all expiring in six months:

(a) the European put options with strike prices K equal to 120, 130, 140;

(b) the arithmetic average Asian put options with monitoring dates at the end of each month (thus six dates in total) and the same strike prices as in (a);

(c) the digital option with payoff

$$\begin{cases} 10 & S(T) < K, \\ 0 & \text{otherwise,} \end{cases}$$

with the same values of K as in Part (a).

Implementation (a) This first part simulates the prices of European put options with different strike prices $K = 120, 130, 140$ at expiration. The simulation is based on the geometric Brownian motion model, where the stock price at expiration ($S(T)$) is determined by the Euler-Maruyama method.

In particular, the initial stock price $S(0) = 130$, the risk-free interest rate $r = 2\%$, volatility $\sigma = 0.2$, time to expiration $T = 0.5$ (6 months) and a standard normal random variable Z are used in the simulation. The Euler-Maruyama method provides an approximation of the stock price at expiration using the following formula:

$$S(T) = S(0)exp((r - \frac{1}{2}\sigma^2) \times T + \sigma\sqrt{T}Z),$$

then the code calculates the European put option prices at expiration for each sample path.

Furthermore, the code estimates the Δ and ν (vega) of the options using the Monte Carlo simulated option prices. The delta is the derivative of the option price with respect to the stock price, and the vega is the derivative of the option price with respect to the volatility. The delta and vega are estimated by averaging the partial derivative of the option price with respect to the stock price and the volatility over $M = 1000000$ simulated stock prices. Since the European put option is not time-dependent, the delta and vega are computed using the following

$$\frac{dY}{d\sigma} = e^{-rT}(-\sigma T + \sqrt{T}z)S(T)\mathbf{1}\{S(T) > K\}$$

and

$$\frac{dY}{dS(0)} = e^{-rT}\frac{S(T)}{S(0)}\mathbf{1}\{S(T) > K\}$$

Lastly, we then average the Greeks by computing the mean over all simulations. The results are shown in Table 1.

Strike	**Estimated Price**	**Estimated Delta**	**Estimated Vega**
$K = 120$	$p_1 = 2.817429$	$\Delta_1 = -0.239712$	$\nu_1 = 28.566528$
$K = 130$	$p_2 = 6.730186$	$\Delta_2 = -0.443533$	$\nu_2 = 36.303493$
$K = 140$	$p_3 = 12.762675$	$\Delta_3 = -0.648515$	$\nu_3 = 34.090829$

Table 1: European Call

Implementation (b) Following the same parameters and assumptions as the European put from (a), we now want to simulate the arithmetic average Asian put. The payoff for this option is given by the following formula:

$$Y = \max\left(K - \frac{1}{T}\int_0^T S(t)dt, 0\right).$$

Since the final time is 6 months, and the option can be exercised early at the end of each month, then we have 6 monitoring dates in total. We can compute the average of the underlying asset prices at these monitoring dates to get the average stock price. And use this average stock price in the final payoff calculation.

The computation for the option delta and vega also differs from the European put option as we now need to consider the pathwise estimators. The delta of the option is defined as the derivative of the option price with respect to the stock price. For the arithmetic average Asian put option the delta is given by the following formula:

$$\frac{dY}{dS(0)} = e^{-rT}\mathbf{1}\{\bar{S} > K\}\frac{\bar{S}}{S(0)}$$

where e^{-rT} is the discount factor, $\bar{S}$ is the arithmetic average of the underlying asset prices at the monitoring dates, S(0) is the initial stock price.

The vega is defined as the derivative of the option price with respect to volatility. The vega of the arithmetic average Asian put option is given by the following formula:

$$e^{-rT}\sum_{i=1}^{N}\frac{dS(t_i)}{d\sigma}\mathbf{1}\{\bar{S} > K\}$$

where N is the number of monitoring dates and $S(t_i)$ is the underlying price at the i-th monitoring date. The results from this option is shown in Table 2.

Strike	**Estimated Price**	**Estimated Delta**	**Estimated Vega**
$K = 120$	$p_1 = 1.069633$	$\Delta_1 = -0.162836$	$\nu_1 = 11.815700$
$K = 130$	$p_2 = 4.406525$	$\Delta_2 = -0.457106$	$\nu_2 = 18.339488$
$K = 140$	$p_3 = 10.906453$	$\Delta_3 = -0.754900$	$\nu_3 = 12.855311$

Table 2: Arithmetic average Asian Put

Implementation (c) This section simulates digital option prices and calculates digital option deltas and vegas. For this, we assume the same parameters and models as before. The code first simulates the stock price at expiration (ST). It then calculates the digital option prices by checking if the strike price (K) is greater than the simulated stock price (ST). If the strike price is greater than the simulated stock price, the digital option price is 10, otherwise, it is 0. Since the payoff is time-dependent, we follow the calculations for Δ and ν as in (b). The results from this option are shown in Table 3.

Strike	**Estimated Price**	**Estimated Delta**	**Estimated Vega**
$K = 120$	$p_1 = 2.859980$	$\Delta_1 = -0.239898$	$\nu_1 = 28.586623$
$K = 130$	$p_2 = 5.000080$	$\Delta_2 = -0.443731$	$\nu_2 = 36.332276$
$K = 140$	$p_3 = 6.997680$	$\Delta_3 = -0.648864$	$\nu_3 = 34.095768$

Table 3: Digital option

Problem 4.

Apply the least-squares approach to approximate numerically the price of a Bermuda call option with the above stock as the underlying instrument. The strike price is $K = 135$, and the option can be exercised at the end of each month. You can use either the approach outlined in class or the closely related approach from Section 8.6 in the textbook. Investigate how different values of the initial price of the underlying stock influence the price of this option (consider $S(0) \in \{120, 125, 130, 135, 140\}$).

Implementation The Least-Squares Monte Carlo (LSMC) method is a technique used to price options such as Bermuda or American style options. The method approximates the expected future option value, $V(S_t, t)$, at each **discrete** time step, t, by using a polynomial function of the underlying asset price, S_t, of degree n (`poly_degree` in the code). The parameters of this polynomial function, denoted by λ, are determined by minimizing the least squares error between the polynomial function and the discounted future option values, $V(S_t, t+1)e^{-r(T-t)}$, which are simulated using the Monte Carlo method[1].

First, generate M independent stock paths, $\{S(t)^i\}_{i=1}^{M}$, where the underlying stock is modeled by a geometric Brownian motion. Then at each discrete time step, calculate the discounted option values, $V(S_t, t+1)e^{-r(T-t)}$ for each simulation by computing the option at the final time step with discounting. Then fit a polynomial function of degree n to the set of pairs

$$\{S(t)^i, V(S_t, t+1)e^{-r(T-t)}\}$$

using least-square regression, to obtain coefficients λ. For each simulation, the method uses the polynomial function and the current asset price $S(t)^i$ to calculate the future option value,

$$h(S(t)^i, t) = \lambda_0 + \lambda_1 S(t)^i + \cdots + \lambda_n (S(t)^i)^n.$$

Then updates the option value for

$$V(S_t, t) = \max(V(S_t, t+1)e^{-r(T-t)}, \max(S_t - K, 0)).$$

The final estimate of the option price is the average of the option value at $t = 0$.

The function `LSMC_Bermuda_Call(S0, K, r, T, sigma, M, N, poly_degree)` takes as input the initial price of the underlying asset ($S(0)$), the strike price of the option (K), the risk-free interest rate (r), the time to expiration of the option (T), the volatility of the underlying asset (σ), the number of Monte Carlo simulations to run (M), the number of time steps to divide the simulation into (N number of monitoring dates), and the degree of the polynomial (`poly_degree`). The results for an increasing initial stock price $S(0) = \{120, \ldots 140\}$ are shown in Figure 4. As expected the option price increases for the call option as the stock price increases. If the number of points increase in the figure we can expect a less linear plot.

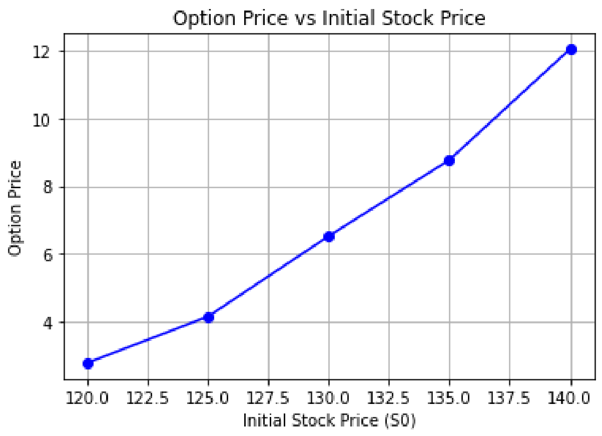


Figure 4: Bermuda style call option approximated by the LSMC

References

[1] P. Glasserman, *Monte Carlo Methods in Financial Engineering*. Applications of mathematics : stochastic modelling and applied probability, Springer, 2004.

A Code

A.1 Question 1

```
# initialize packages
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import tplquad
from scipy.stats import norm
from scipy.optimize import minimize
from tqdm import tqdm

def halton(n,b):
    """
    Compute the Halton sequence for the given number of elements and base.
    Args:
        n: The number of elements in the sequence.
        b: The base to use for the Halton sequence.
    Returns:
        The computed Halton sequence of length n.
    """
    # Create an array of zeros with length n and store it in the variable h
    h = np.zeros(n)
    p = 1./b
    # Loop through n elements
    for i in range(n):
        h[i] = 0
        f = p
        j = i+1
        while j > 0:
            h[i] += f*(j%b)
            j = np.floor(j/b)
            f = f/b
    return h

def f(x,y,z):
    """
    The function to integrate.
    Args:
        x: The x coordinate.
        y: The y coordinate.
        z: The z coordinate.
    Returns:
        The value of the function at the given coordinates.
    """
    return x*y/np.sqrt(x**2 + y**2 + z**2)
```

```
def qmc(n):
    """
    Compute the quasi-Monte Carlo integration of the function f.
    Args:
        n: The number of samples to use for the integration.
    Returns:
        The computed value of the integration.
    """
    # Halton sequence with bases 2, 3 and 5 respectively.
    x = halton(n,2)
    y = halton(n,3)
    z = halton(n,5)
    return np.sum(f(x,y,z))/n

def mc(n):
    """
    Compute the Monte Carlo integration of the function f.
    Args:
        n: The number of samples to use for the integration.
    Returns:
        The computed value of the integration.
    """
    x = np.random.uniform(0,1,n)
    y = np.random.uniform(0,1,n)
    z = np.random.uniform(0,1,n)
    return np.sum(f(x,y,z))/n

n = np.logspace(0,6,1000, dtype='int')
qmc_err = []
mc_err = []
for i in tqdm(n):
    qmc_err.append(np.abs(qmc(i)-tplquad(f,0,1,0,1,0,1)[0]))
    mc_err.append(np.abs(mc(i)-tplquad(f,0,1,0,1,0,1)[0]))

plt.figure()
plt.loglog(n,mc_err,label='Monte Carlo', color='C1', linestyle='--',
alpha=0.5,marker='o', markersize=3)
plt.loglog(n,qmc_err,label='Quasi-Monte Carlo', color='C0', marker='o', markersize=3)
plt.xlabel('Number of Samples')
plt.ylabel('Error')
plt.legend()

# Add a title and a grid to the figure.
plt.title('Error of quasi-Monte Carlo and Monte Carlo integration')
plt.grid(True)
plt.show()

# Polynomial curve fitting of degree 1
qmc_fit = np.polyfit(np.log(n), np.log(qmc_err), 1)
mc_fit = np.polyfit(np.log(n), np.log(mc_err), 1)

qmc_slope = qmc_fit[0]
qmc_intercept = qmc_fit[1]

mc_slope = mc_fit[0]
mc_intercept = mc_fit[1]

# Convergence rate comparison
plt.loglog(n,np.power(n, qmc_slope),color='C0', label='Quasi-Monte Carlo Regression')
plt.loglog(n,np.power(n, mc_slope),color='C1', label='Monte Carlo Regression')
plt.loglog(n,np.power(n, -0.68191),'--',color='C0', label='Theoretical QMC')
plt.loglog(n,np.power(n, -1/2),'--',color='C1', label='Theoretical MC')
plt.xlabel('Number of Samples')
plt.ylabel('Error')
plt.legend()
plt.grid(True)

plt.show()
```

A.2 Question 2

```
def target(x):
    """
    Evaluate the target distribution at the given point x.
    Args:
        x: The point at which to evaluate the target distribution.
    Returns:
        The value of the target distribution at the given point x.
    """
    return x**3 * np.sin(np.pi * x)/ (2 / np.pi - 12 / np.pi**3)

def proposal(x, y, sigma):
    """
    Generate a proposal point y using a normal distribution centered at x.
    Args:
        x: The center of the normal distribution.
        y: The point at which to evaluate the proposal distribution.
        sigma: The standard deviation of the normal distribution.
    Returns:
        A proposal point y generated using the given normal distribution.
    """
    return 1 / np.sqrt(2 * np.pi * sigma**2) * np.exp(-(y - x)**2 / (2 * sigma**2))
def metropolis_hastings(n_samples, sigma):
    """
    Generate a sequence of samples using the Metropolis-Hastings algorithm.
    args:
    n_samples: The number of samples to generate.
    sigma: The standard deviation of the normal distribution used as the proposal
    distribution.
    Returns:
    samples: A 1-D array of samples.
    """
    # Initialize an empty list to store the samples
    samples = []
    x = 0 # start from x = 0
    # Calculate the constant C
    C = 2 * np.pi - 1 / 2 * np.pi**3
    # Loop through the number of samples
    for i in range(n_samples):
        # Generate a proposal point y using a normal distribution centered at x with
        standard deviation sigma
        y = np.random.normal(x, sigma)
        # Calculate the acceptance probability
        acceptance_probability = target(y) / (target(x) + 1e-10) * proposal(x, y, sigma)
        / (proposal(y, x, sigma)+ 1e-10)
        # If the acceptance probability is greater than or equal to 1, accept the
        proposal point y
        if acceptance_probability >= 1:
            x = y
            samples.append(x)
        # If the acceptance probability is less than 1, use a random uniform value to
        determine whether to accept or reject the proposal point y
        else:
            u = np.random.uniform()
            if u < acceptance_probability:
                x = y
                samples.append(x)
            else:
                samples.append(x)
     # Return the samples
    return np.array(samples)
# Set the number of samples and the variance of the normal distribution.
n_samples = 10000
random_seed = 1
# Set the values of sigma to use.
sigmas = [0.1, 0.5, 1.0, 2.0]
```

```
fig, axs = plt.subplots(2,2, figsize=(15, 6), facecolor='w', edgecolor='k')
fig1, axs1 = plt.subplots(2,2, figsize=(15, 6), facecolor='w', edgecolor='k')
axs = axs.ravel()
axs1 = axs1.ravel()
j=0
# Generate a sample for each value of sigma, and plot the resulting histogram.
for sigma in sigmas:
    # Initialize the sample array and the starting point.
    sample = np.zeros(n_samples)
    # Generate the sample using the Metropolis-Hastings algorithm.
    sample = metropolis_hastings(n_samples, sigma)
    # Plot the simulation of the sample.
    axs[j].plot(range(len(sample)),sample)
    axs[j].legend(['Theoretical simulation, $\sigma$ = '+ str(sigma)])
    axs1[j].hist(sample, bins=100, density=True)
    axs1[j].legend(['Theoretical distribution, $\sigma$ = '+ str(sigma)])
    j+=1
```

A.3 Question 3

A.3.1 (a)

```
# parameters
S0 = 130
r = 0.02
sigma = 0.2
T = 0.5
M = 1000000

# simulate stock price at expiration
z = np.random.standard_normal(M)
ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T + sigma * np.sqrt(T) * z)

# calculate European put option prices
Ks = [120, 130, 140]
put_price = np.zeros((M, len(Ks)))
for i in range(M):
    for j, k in enumerate(Ks):
        put_price[i, j] = max(k - ST[i], 0)

# estimate delta and vega using Monte Carlo simulation
deltas = np.zeros((M, len(Ks)))
vegas = np.zeros((M, len(Ks)))
for i in range(M):
    for j, k in enumerate(Ks):
        if ST[i]<k:
            vegas[i, j] = -np.exp(-r*T)*(-sigma*T+np.sqrt(T)*z[i])*ST[i]
            deltas[i, j] = -np.exp(-r*T)*ST[i]/S0
vegas = vegas.mean(axis=0)
deltas = deltas.mean(axis=0)

# output
print("European puy option prices:")
for i, k in enumerate(Ks):
    print("K = %d: %.6f" % (k, put_price[:,i].mean()))

print("\nEuropean put option deltas:")
for i, k in enumerate(Ks):
    print("K = %d: %.6f" % (k, deltas[i]))

print("\nEuropean put option vegas:")
for i, k in enumerate(Ks):
    print("K = %d: %.6f" % (k, vegas[i]))
```

A.3.2 (b)

```
# Parameters
S0 = 130
r = 0.02
sigma = 0.2
T = 0.5
M = 1000000
monitoring_dates = 6
dt = T / monitoring_dates

S = np.zeros((monitoring_dates+1,M))
S[0,:] = S0
# Simulation
z = np.random.standard_normal((monitoring_dates+1,M))
for i in range(0,monitoring_dates):
    S[i+1] = S[i] * np.exp((r - 0.5 * sigma ** 2) * T + sigma * np.sqrt(dt) * z[i])
# Calculate average stock price over the six months
S_avg = S[1:,:].mean(axis=0) # Removing S0
dSdSigma = S * np.log(S / S0 - (r + 0.5 * sigma ** 2) * dt) / sigma
# Calculate option prices
Ks = [120, 130, 140]
put_price = np.zeros((M, len(Ks)))
for i in range(M):
    for j, k in enumerate(Ks):
        put_price[i, j] = max(k - S_avg[i], 0)
deltas = np.zeros((M, len(Ks)))
```

```
vegas = np.zeros((M, len(Ks)))
for i in range(M):
    for j, k in enumerate(Ks):
        if S_avg[i]<k:
            vegas[i, j] = -np.exp(-r*T)*np.mean(dSdSigma[:,i])
            deltas[i, j] = -np.exp(-r*T)*S_avg[i]/S0
vegas = vegas.mean(axis=0)
deltas = deltas.mean(axis=0)

# output
print("Arithmetic Asian put prices:")
for i, k in enumerate(Ks):
    print("K = %d: %.6f" % (k, put_price[:,i].mean()))

print("Arithmetic Asian put option deltas:")
for i, k in enumerate(Ks):
    print("K = %d: %.6f" % (k, deltas[i]))

print("\nArithmetic Asian put option vegas:")
for i, k in enumerate(Ks):
    print("K = %d: %.6f" % (k, vegas[i]))
```

A.3.3 (c)

```
# parameters
S0 = 130
r = 0.02
sigma = 0.2
T = 0.5
M = 1000000

# simulate stock price at expiration
z = np.random.standard_normal(M)
ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T + sigma * np.sqrt(T) * z)

# calculate digital option prices
Ks = [120, 130, 140]
digital_price = np.zeros((M, len(Ks)))
for i in range(M):
    for j, k in enumerate(Ks):
        digital_price[i, j] = 10 if k - ST[i] > 0 else 0

# estimate delta and vega using Monte Carlo simulation
deltas = np.zeros((M, len(Ks)))
vegas = np.zeros((M, len(Ks)))
for i in range(M):
    for j, k in enumerate(Ks):
        if ST[i]<k:
            vegas[i, j] = -np.exp(-r*T)*(-sigma*T+np.sqrt(T)*z[i])*ST[i]
            deltas[i, j] = -np.exp(-r*T)*ST[i]/S0
vegas = vegas.mean(axis=0)
deltas = deltas.mean(axis=0)

# output
print("Digital option prices:")
for i, k in enumerate(Ks):
    print("K = %d: %.6f" % (k, digital_price[:,i].mean()))

print("\nDigital option deltas:")
for i, k in enumerate(Ks):
    print("K = %d: %.6f" % (k, deltas[i]))

print("\nDigital option vegas:")
for i, k in enumerate(Ks):
    print("K = %d: %.6f" % (k, vegas[i]))
```

A.4 Question 4

```
def LSMC_Bermuda_Call(S0, K, r, T, sigma, M, N, poly_degree):
    """
    This function calculates the price of a Bermuda call option
    using the Least-Squares Monte Carlo (LSMC) method.
    Args:
    S0 : The initial price of the underlying asset.
    K : The strike price of the option.
    r : The risk-free interest rate.
    T : The time to expiration of the option, in years.
    sigma : The volatility of the underlying asset.
    M : The number of Monte Carlo simulations to run.
    N : The number of time steps to divide the simulation into.
    poly_degree The degree of the polynomial used in the LSMC method.

    Returns:
    price : The estimated price of the Bermuda call option.
    """
    #np.random.seed(random_seed)
    # Calculate the time step size
    dt = T / N
    # Generate matrix of brownian motions
    dW = np.random.standard_normal((M, N + 1))*np.sqrt(dt)
    # Calculate asset price at each time step using the Euler-Maruyama method
    S = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt + sigma * dW,axis=1))
    S[:, 0] = S0
    # Initialize matrix to store option values at each time step
    V = np.zeros((M,N+1))
    # Calculate option values at maturity
    C = np.maximum(S - K, 0)
    V[:, -1] = C[:, -1]
    # Iterate backwards through time steps
    for t in range(N - 1, 0, -1):
        # Fit polynomial to asset prices and discounted future option values
        Lambdas = np.polyfit(S[:, t], V[:, t + 1] * np.exp(-r * dt), poly_degree)
        # Calculate expected future option values using fitted polynomial
        h = np.zeros(M)
        for i in range(M):
            h[i] = 0
            for j in range(poly_degree+1):
                h[i] += Lambdas[j]*S[i, t]**(poly_degree-j)
        # Update option values using optimal exercise decision
        for i in range(M):
            if C[i, t] > h[i]:
                V[i, t] = C[i, t]
            else:
                V[i, t] = V[i, t + 1] * np.exp(-r * dt)
    # Calculate average option value at time t=0
    price = np.mean(V[:, 1] * np.exp(-r * dt))
    return price

random_seed = 1
S0 = [120,125,130,135,140]
K = 135
r = 0.02
sigma = 0.2
T = 0.5
N = int(12 * 0.5)
M = 10000
dt = T / N
poly_degree=5
price = np.zeros(len(S0))
for i in range(len(S0)):
    price[i] = LSMC_Bermuda_Call(S0[i], K, r, T, sigma, M, N, poly_degree)
# Add x and y labels and a title
plt.xlabel('Initial Stock Price (S0)')
plt.ylabel('Option Price')
plt.title('Option Price vs Initial Stock Price')

# Add a grid
plt.grid(True)
```

```
# Plot the data
plt.plot(S0, price, 'o-', color='blue')

# Show the plot
plt.show()
```