MONTE CARLO

INTRODUCTORY CONCEPTS

For a random variable Y if one wishes to calculate the expectation and one can produce a sequence of i.i.d random variables $Z_1, Z_2, \ldots$ from the distribution of Y then $\hat{Y}_n = \frac{1}{n}\sum_{i=1}^{n} Z_i$ is a consistent estimator of $\mu = E[Y]$.

If $\sigma^2 = Var[Y]$ then by the central limit theorem $Y_n - \mu \sim \frac{\sigma}{\sqrt{n}} N(0,1)$

Hence $\mu_n = \frac{1}{n}\sum_{i=1}^{n} Z_i$ and $s_n{}^2 = \frac{1}{n-1} \sum_{i=1}^{n}(Z_i - \mu_n)^2$ are respectively sample mean and variance.

Then an approximate 95% confidence interval for μ is $\mu_n \pm \{ 1.96 \frac{s_n}{\sqrt{n}}\}$

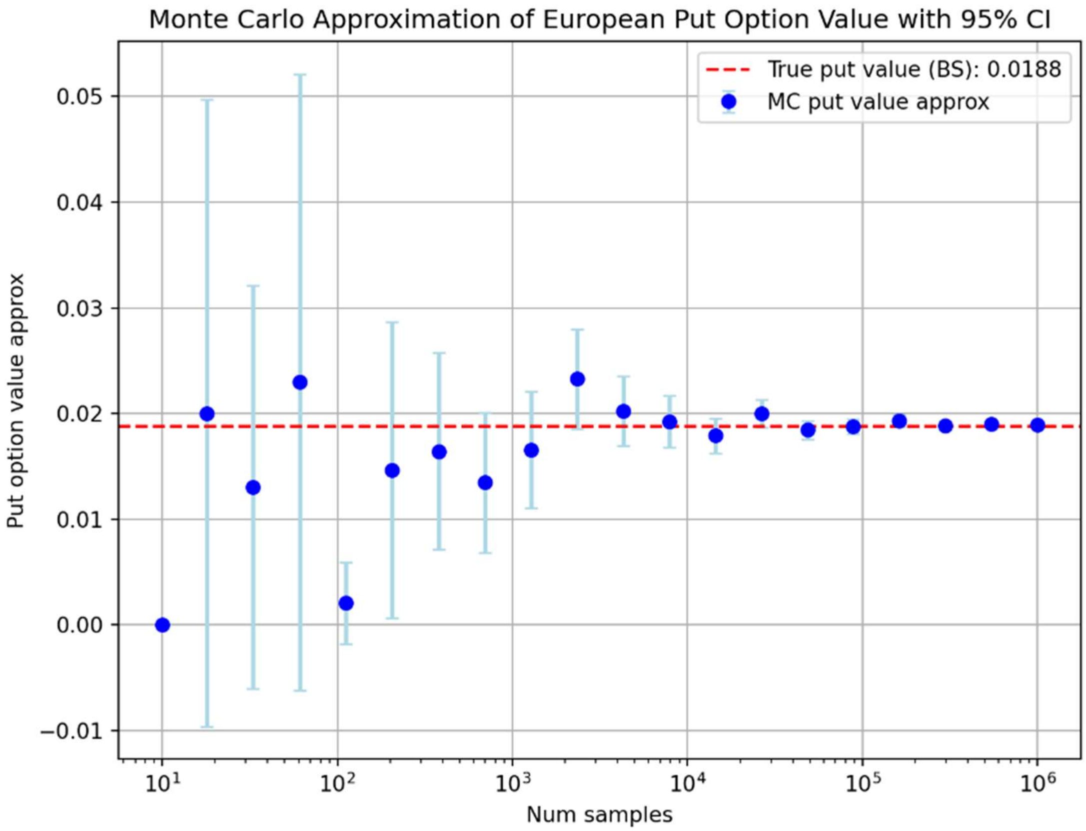
Recall, in the Black Scoles Model under the risk neutral measure
$dS_t = \mu S_t dt + \sigma S_t dW_t \quad P_0 = E^{\mathbb{Q}}[e^{-rT}(K - S_T)^+]$

Since
$S_T = S_0 \exp\left(\left(\mu - \frac{\sigma^2}{2}\right)T + \sigma W_T\right)$ we only have to sample from $N(0, T)$

We will simulate a put option under this model:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

np.random.seed(42)

# Parameters for the European put option

S0 = 10        # Initial stock price
K = 9          # Strike price
sigma = 0.1    # Volatility
r = 0.06       # Risk-free rate
T = 1          # Time to maturity

# Number of samples to use (logarithmic scale)
sample_sizes = np.logspace(1, 6, num=20, dtype=int)

# Function to calculate the payoff of a European put option
def put_payoff(S_T, K):
    return np.maximum(K - S_T, 0)

# Monte Carlo simulation of the European put option price

def monte_carlo_put_price(S0, K, r, sigma, T, num_samples):
    Z = np.random.randn(num_samples)  # Standard normal samples
    S_T = S0 * np.exp((r - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * Z)  # Sim
ulated stock price at T
```

```python
    payoff = put_payoff(S_T, K)
    discounted_payoff = np.exp(-r * T) * payoff
     return discounted_payoff

# Black-Scholes Formula for a European Put Option
def black_scholes_put(S0, K, r, sigma, T):
    d1 = (np.log(S0 / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T)
    d2 = d1 - sigma * np.sqrt(T)
    put_price = K * np.exp(-r * T) * norm.cdf(-d2) - S0 * norm.cdf(-d1)
    return put_price

# Calculate the true Black-Scholes put option price
true_put_value = black_scholes_put(S0, K, r, sigma, T)

# Store the option values, standard deviations, and confidence intervals
put_values = []
conf_intervals_lower = []
conf_intervals_upper = []

for N in sample_sizes:
    payoffs = monte_carlo_put_price(S0, K, r, sigma, T, N)
    # Sample mean (put option value)
    mean_put_price = np.mean(payoffs)
   # Sample variance
    variance = np.var(payoffs, ddof=1)
    # Sample standard deviation
    std_error = np.sqrt(variance / N)
   # 95% confidence interval
    conf_interval = 1.96 * std_error

    # Store values for plotting
    put_values.append(mean_put_price)
    conf_intervals_lower.append(mean_put_price - conf_interval)
    conf_intervals_upper.append(mean_put_price + conf_interval)

# Plotting the results
plt.figure(figsize=(8, 6))
# Plot the Monte Carlo estimates with error bars (95% confidence intervals)
plt.errorbar(sample_sizes, put_values,
             yerr=[np.array(put_values) - np.array(conf_intervals_lower),
                   np.array(conf_intervals_upper) - np.array(put_values)],
             fmt='o', color='blue', label='MC put value approx', ecolor='lightb
lue', elinewidth=2, capsize=3)
# Plot the Black-Scholes true value as a reference
plt.axhline(true_put_value, color='red', linestyle='--', label=f'True put value
(BS): {true_put_value:.4f}')

# Logarithmic scale on x-axis
plt.xscale('log')
plt.xlabel('Num samples')
plt.ylabel('Put option value approx')
plt.legend()
plt.title('Monte Carlo Approximation of European Put Option Value with 95% CI')
plt.grid(True)
plt.show()
```



Euler Discretization

To simulate the trajectories of the discussed models, we can discretize the stochastic differential equation (SDE) with respect to time. For the general SDE represented by:

$$S_t = S_0 + \int_0^t \mu S_u \, du + \int_0^t \sigma S_u \, dB_u$$

we can create a partition $0 = t_0 < t_1 < \cdots < t_N = T$ as before and suppose $\Delta t_j = \Delta t$ for all j. The Euler discretization for this SDE is given by:

$$X_{j+1} = X_j + \mu(t_j, X_j)\Delta t + \sigma(t_j, X_j)\Delta W_j$$

where ΔW_j represents the increments of a Wiener process. It can be demonstrated that, depending on the functions μ and σ, this discretization converges to the solution of the SDE as Δt approaches zero.

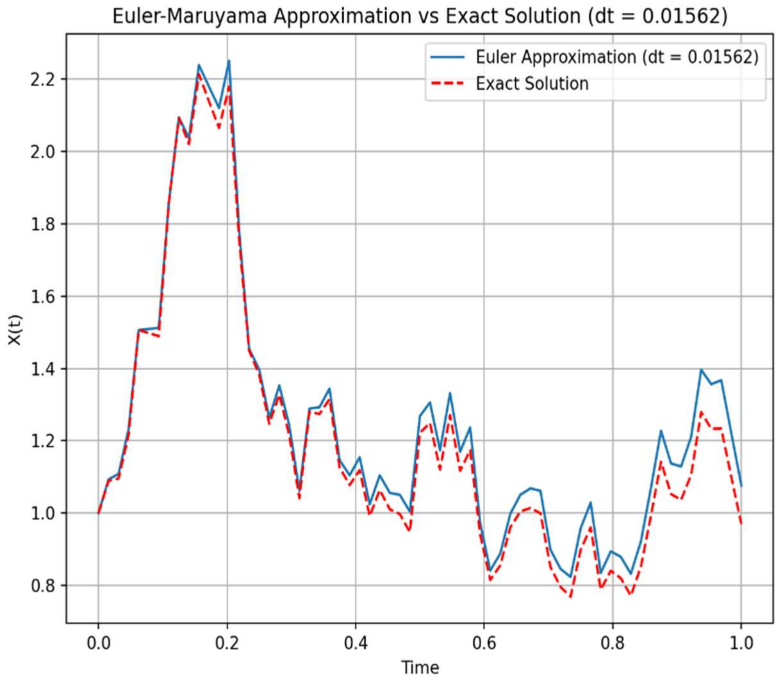
To analyze the discretization error, we can refer to the work of Higham (2001) by applying the method to the linear SDE $dX(t) = \lambda X(t)dt + \mu X(t)dW(t)$

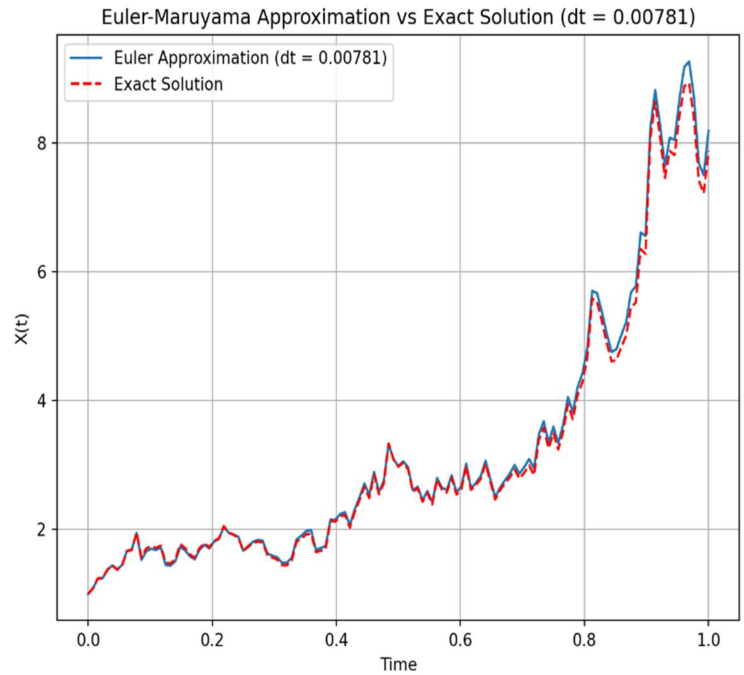
Using Itô's lemma, the exact solution is given by:

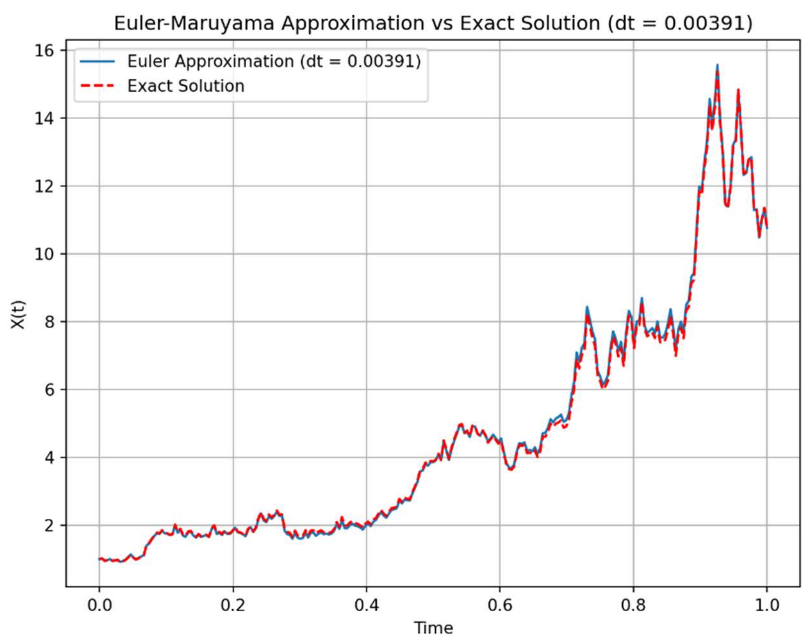
$$X(t) = X(0)\, exp\,((\lambda - 0.5\mu_n{}^2)t + \mu W(t))$$

For parameters $\lambda = 2, \mu = 1, X(0) = 1$, and $\Delta t = (4)2^{\wedge} - 8$ we can simulate a discretized Brownian path and compare the results of the Euler approximation with the true solution

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

# Set the random seed for reproducibility
np.random.seed(42)

# Parameters for the SDE
lambda_ = 2.0   # Drift term
mu = 1.0        # Volatility term
X0 = 1.0        # Initial condition
T = 1.0         # Time horizon

# Function to generate the exact solution at all time points
def exact_solution(lambda_, mu, X0, W, N):
    time_points = np.linspace(0, T, N + 1)
    return X0 * np.exp((lambda_ - 0.5 * mu**2) * time_points + mu * W)

# Function for Euler-Maruyama discretization
def euler_maruyama(lambda_, mu, X0, T, N):
    dt = T / N
    X = np.zeros(N + 1)
    X[0] = X0
    W = np.random.randn(N) * np.sqrt(dt)  # Brownian increments
    W_cumsum = np.concatenate(([0], W.cumsum()))  # Include W(0) = 0
    for j in range(N):
        X[j + 1] = X[j] + lambda_ * X[j] * dt + mu * X[j] * W[j]
    return X, W_cumsum  # Return cumulative sum for W(t)

# Time step sizes
time_steps = [4 * 2**(-8), 2 * 2**(-8), 2**(-8)]
errors = []

# Simulate for each time step size
for dt in time_steps:
    N = int(T / dt)  # Number of steps
    # Euler-Maruyama simulation
    X_euler, W_T = euler_maruyama(lambda_, mu, X0, T, N)
# Exact solution at all time points using the same num of time points as Euler
    X_exact = exact_solution(lambda_, mu, X0, W_T, N)

    # Error at the endpoint
    error = np.abs(X_exact[-1] - X_euler[-1])
    errors.append(error)

    # Create a time vector for plotting
    t = np.linspace(0, T, N + 1)
    plt.figure(figsize=(8, 6))
    plt.plot(t, X_euler, label=f'Euler Approximation (dt = {dt:.5f})')
    plt.plot(t, X_exact, color='red', linestyle='--', label='Exact Solution')
    plt.title(f'Euler-Maruyama Approximation vs Exact Solution (dt = {dt:.5f})'
)
    plt.xlabel('Time')
    plt.ylabel('X(t)')
    plt.grid(True)
    plt.legend()
    plt.show()
```








```python
# Print the errors at the endpoint for different time steps
for dt, error in zip(time_steps, errors):
    print(f"Error at the endpoint with dt = {dt:.10f}: {error:.10f}")
```

```
## Error at the endpoint with dt = 0.0156250000: 0.1070394568
## Error at the endpoint with dt = 0.0078125000: 0.3116458176
## Error at the endpoint with dt = 0.0039062500: 0.0100896752
```

Key Takeaway

In our Black-Scholes model, one of the key takeaways is that in order to apply Monte Carlo, we simulate the asset price based off the property of the increments of the Brownian motion and use that for the simulation of the contingent claim.

Variance Reduction

Antithetic Variables

To reduce variance in Monte Carlo estimators, we employ antithetic pairs of random variables that exhibit negative correlation. The objective is for the negative correlation to offset deviations from the target parameter θ\thetaθ within each pair, leading to a more accurate estimate.

Let n be an even integer and define $Y = h(X)$ with $E[Y] = \theta$. We generate n/2 independent and identically distributed (i.i.d.) pairs of random variables:

$(Y_1, \widetilde{Y_1})$, $(Y_{2.}, \widetilde{Y_2})$,..., $\left(Y_{\frac{n}{2}}, \widetilde{Y}_{\frac{n}{2}}\right)$ Each pair $(Y_i, \widetilde{Y_\iota})$ is constructed to be negatively correlated, while all pairs $(Y_i, \widetilde{Y_\iota})$ and $(Y_j, \widetilde{Y_J})$ are independent for $i \neq j$. The purpose of the negative correlation is to reduce variance by balancing out deviations in opposite directions.

The antithetic estimator is then given by:

$$\hat{\theta}_{AT} = \frac{1}{n/2} \cdot \sum_{i=1}^{n/2} \frac{Y_i + \tilde{Y}_i}{2}$$

$$Var[\hat{\theta}_{AT}] = \frac{\sigma^2}{n} + \frac{1}{n} COV[Y_i, \widetilde{Y_\iota}]$$

This estimator averages each pair and combines them to produce a more precise estimate of θ, leveraging the negative correlation to reduce the estimator's variance compared to using independent samples.

Antithetic Variables

To estimate the parameter:

$\theta = \int_0^1 f(u)\,du$ we can use the crude Monte Carlo estimator:

$\hat{\theta}_{CR} = \frac{1}{n}\sum_{i=1}^{n} f(U_i)$, where $\{U_1,\ldots, U_n\}$ are independent and identically distributed (i.i.d.) random variables drawn from a uniform distribution $UNIF(0,1)$.

Now, assume there exists a function $g(u)$ such that $f(U_i)$ and $g(U_i)$ are positively correlated. Additionally, we assume that we can compute $\theta_g = E[g(U)]$ exactly, without relying on simulation.

Control Variates Method

If the crude Monte Carlo estimator of $\hat{\theta}_g = \frac{1}{n}\sum_{i=1}^{n} g(U_i)$, and if $\hat{\theta}_g$ is larger than θ_g, then it is likely that $\hat{\theta}_{CR}$ is also overestimating θ. Similarly, if $\hat{\theta}_g$ is smaller than θ , we suspect $\hat{\theta}_{CR}$ is underestimating θ. Therefore, we adjust $\hat{\theta}_{CR}$ by subtracting or adding a correction based on the difference $\hat{\theta}_g$ - θ_g

Definition

The **control variate (CV) estimator** takes the form:

$\hat{\theta}_{CV} = \frac{1}{n}\sum_{i=1}^{n} f(U_i) + \beta(\theta_g - g(U_i)\ )$which can be rewritten as:

$$\hat{\theta}_{CV} = \frac{1}{n}\sum_{i=1}^{n} f(U_i) + \beta g(U_i)) - E[(U_i)\ ]$$

where β is a constant chosen to minimize the variance of $\hat{\theta}_{CV}$.

Optimal β

The optimal value of β\betaβ that minimizes the variance of $\hat{\theta}_{CV}$ is:

$$\beta^* = \frac{COV[f(U_i), g(U_i)]}{VAR[g(U_i)]}$$

In this case, $VAR[\hat{\theta}_{CV}] = (1-\rho^2)\,VAR[\hat{\theta}_{CR}]$

where ρ is the correlation between $f(U_i)$ and $g(U_i)$:

$\rho = \frac{COV[f(U_i),g(U_i)]}{\sqrt{VAR(f(U_i))VAR[g(U_i)]}}$ If ρ=±1, the estimator becomes perfect with zero variance.

Estimation of β^*

Since $COV[f(U_i), g(U_i)]$ is typically unknown, we estimate β^* by using the empirical covariance and variance from the sample:

$$\hat{\beta} = \frac{\sum_{i=1}^{n} f(U_i)g(U_i) - n\hat{\theta}_{CR}\hat{\theta}_g}{(n-1)S_g^2}$$

where S_g^2 is the sample variance of $g(U_i)$, given by:

$$S_g^2 = \frac{1}{n-1} \cdot \sum_{i=1}^{n} \left(g(U_i) - \hat{\theta}_g\right)^2$$

If the variance of $g(U_i)$ is known, it can replace S_g^2 in the formula for $\hat{\beta}$.

Example Usage: Arithmetic Asian Option

The payoff of a time-T expiry arithmetic average Asian put option with strike price K is

$$h_T = \left(K - \frac{1}{n}\sum_{i=1}^{r} S_{t_i}\right)^+$$ where:

S_{t_i} : the asset prices at equally spaced times t_i over the period $[0, T]$

$(x)^+$: defined as $max(0, x)$

Suppose that our model is such that the price of an asset at time t, S_t, follows the dynamics $dS_t = \mu S_t dt + \sigma S_t dW_t$ under the risk-neutral measure $\mathbb{Q}$.

Consider the partition $\pi = \{0 = t_0 < t_1 < \cdots < t_n = T\}$ of $[0, T]$ with equal spacing Δt.

Let us suppose further that $r = 0.0175$, $\sigma = 0.25$, $S_0 = 500$, $K = S_0$, $T = 1$, and $n = 52$.

We will write a program that will generate a path $(S_0, S_{t_1}, \ldots, S_{t_n})$ of the price asset using the solution of the geometric Brownian Motion.

```
import numpy as np
import pandas as pd

def S_path(S0, r, sigma, T, n):
    # Time increment
    dt = T / n
    # Initialize price array with zeros and set the first element to S0
    S = np.zeros(n + 1)
    S[0] = S0
```

```
    # Generate the price path
    for i in range(1, n + 1):
        Z = np.random.normal()  # Standard normal random variable
        X = (r - 0.5 * sigma ** 2) * dt + sigma * np.sqrt(dt) * Z
        S[i] = S[i - 1] * np.exp(X)

    return S
```

Now, we will attempt using the N-paths of the asset price to calculate the arithmetic average Asian put option prices at time zero using (crude) Monte-Carlo, and we will also show the confidence interval of this option using the following code.

```
# Parameters
S0 = 500  # Initial stock price
r = 0.0175  # Risk-free interest rate
sigma = 0.25  # Volatility
T = 1  # Time to maturity
n = 52  # Number of time steps (weekly)
K = S0  # Strike price
Ns = [10**4, 10**5]  # Different Monte Carlo sample sizes


def Arith_Avg_Crude_MC(S0, r, sigma, T, n, K, N):
    np.random.seed(42)  # Set seed for reproducibility
    temp = 0
    temp2 = 0
    for _ in range(N):
        # Generate a price path
        S = S_path(S0, r, sigma, T, n)
       # Calculate the arithmetic average of the path
        Aavg = sum(S[1:]) / n  # Sum from S[1] to S[n] and divide by n
        # Calculate the payoff for the put option
        if Aavg < K:
            disc_payoff = np.exp(-r * T) * (K - Aavg)
            temp += disc_payoff
            temp2 += disc_payoff ** 2
    # Monte Carlo estimates
    muhat = temp / N
    s2 = (temp2 / (N - 1)) - (N / (N - 1)) * muhat ** 2
    shat = np.sqrt(s2)
    MSE = s2 / N
    # 95% confidence interval
    ci_error = 1.96 * shat / np.sqrt(N)
    ci_l = muhat - ci_error
    ci_u = muhat + ci_error
    CI = (ci_l, ci_u)
    return {
        "N": N,
        "Estimated Price": muhat,
        "MSE": MSE,
        "Confidence Interval": CI
    }
```

```
# Run simulations for different values of N
results = [Arith_Avg_Crude_MC(S0, r, sigma, T, n, K, N) for N in Ns]
# Convert results to DataFrame
df_results = pd.DataFrame(results)
df_results.columns = ["Sample Size (N)", "Estimated Price", "MSE", "Con
fidence Interval"]
# Set pandas display option to show all columns
pd.set_option("display.max_columns", None)

print(df_results)
```

```
##          (N)   Estimated Price     MSE       Confidence Interval
## 0     10000    26.947076       0.139226   (26.215741072, 27.67841046)
## 1    100000    26.496398       0.013459   (26.26901314,  26.72378240)
```

Now we will apply the concept of Antithetic variables as we have explained above.

```
# Parameters
S0 = 500  # Initial stock price
r = 0.0175  # Risk-free interest rate
sigma = 0.25  # Volatility
T = 1  # Time to maturity
n = 52  # Number of time steps (weekly)
K = S0  # Strike price
Ns = [10**4, 10**5]  # Different Monte Carlo sample sizes
# Define functions

def Arith_Avg_Anti_Sums(S0, r, sigma, T, n):
    dt = T / n
    S, Sa = S0, S0
    tempSum, tempSuma = 0, 0
    for _ in range(n):
        Z = np.random.normal()
        X = (r - 0.5 * sigma ** 2) * dt + sigma * np.sqrt(dt) * Z
        Xa = (r - 0.5 * sigma ** 2) * dt - sigma * np.sqrt(dt) * Z
        S *= np.exp(X)
        Sa *= np.exp(Xa)
        tempSum += S
        tempSuma += Sa

    return (tempSum / n, tempSuma / n)


def Arith_crude_MC(S0, r, sigma, T, n, K, N):
    if N % 2 != 0:
        raise ValueError("Error: N must be even")
    M = N // 2
    np.random.seed(42)  # Set seed for reproducibility
    temp, temp2 = 0, 0

    for _ in range(M):
        Aavg1, Aavg2 = Arith_Avg_Anti_Sums(S0, r, sigma, T, n)
        # Calculate discounted payoffs for the put option
        disc_payoff1 = np.exp(-r * T) * max(K - Aavg1, 0)
        disc_payoff2 = np.exp(-r * T) * max(K - Aavg2, 0)
        disc_payoff_avg = (disc_payoff1 + disc_payoff2) / 2
```

```
        temp += disc_payoff_avg
        temp2 += disc_payoff_avg ** 2

    # Monte Carlo estimates
    muhat = temp / M
    s2 = (temp2 / (M - 1)) - (M / (M - 1)) * muhat ** 2
    shat = np.sqrt(s2)
    MSE = s2 / M

    # 95% confidence interval
    ci_error = 1.96 * shat / np.sqrt(M)
    ci_l = muhat - ci_error
    ci_u = muhat + ci_error
    CI = (ci_l, ci_u)

    return {
        "Sample Size (N)": N,
        "Estimated Price": muhat,
        "MSE": MSE,
        "Confidence Interval": CI
    }

# Run simulations for different values of N
results = [Arith_crude_MC(S0, r, sigma, T, n, K, N) for N in Ns]

# Convert results to DataFrame
df_results = pd.DataFrame(results)

# Set pandas display option to show all columns
pd.set_option("display.max_columns", None)

# Display the DataFrame
print(df_results)
```

```
##        (N)   Estimated Price      MSE      Confidence Interval
## 0    10000    26.425029      0.064996   (25.92533845, 26.924718647)
## 1   100000    26.580326      0.006499   (26.42231737, 26.738334127)
```

The implementation of antithetic variables in our Monte Carlo simulation demonstrated a notable impact on the confidence intervals. Specifically, for a sample size of $N=10^5$, the width of the confidence interval was approximately 0.3, indicating a relatively wide range of uncertainty in our estimates but a better one nevertheless than the crude Monte-Carlo estimate.

Moreover, the use of antithetic variates led to a significant improvement in estimation efficiency. Excluding considerations of computation time, the efficiency enhancement was calculated to be 107.09%, determined using the formula $Eff = \frac{1}{MSE}$

This substantial increase in efficiency underscores the benefits of employing variance reduction techniques, such as antithetic variates, in Monte Carlo simulations for option pricing.

To further reduce variance, we will explore the application of the control variates method in our analysis. This technique aims to leverage the known properties of related variables to enhance the precision of our estimators, potentially leading to even tighter confidence intervals and more reliable pricing estimates. We shall be using the geometric Asian put option as a control variate.

NOTE: more details on the geometric Asian put option will be posted at a later date. But for the sake of clearance we will use the fact that for the geometric Asian put is such that $P_{GA} = e^{-rT}\left(K\Phi\left(-\widehat{d_2}\right) - s_0 e^{\hat{\mu}T}\Phi\left(-\widehat{d_1}\right)\right)$ *where* $\hat{\mu} = \left(r \quad -\frac{1}{2} \quad \sigma^2\right)\frac{(n+1)}{2n} + \frac{1}{2}\hat{\sigma}^2$,

$\hat{\sigma}^2 = \frac{\sigma^2(n+1)(2n+1)}{6n^2}$, $\hat{d}_1 = \frac{log\left(\frac{S_0}{K}\right)+\left(\hat{\mu}+\frac{\hat{\sigma}^2}{2}\right)T}{\hat{\sigma}\sqrt{T}}$ *and* $\hat{d}_2 = \hat{d}_1 - \hat{\sigma}\sqrt{T}$

```
import numpy as np
import pandas as pd
from scipy.stats import norm

def arith_geo_avgs(S0, r, sigma, T, n):
    Dt = T / n
    S = S0
    temp_sum = 0
    temp_prod = 1

    for _ in range(n):
        Z = np.random.normal()
        X = (r - 0.5 * sigma ** 2) * Dt + sigma * np.sqrt(Dt) * Z
        S *= np.exp(X)
        temp_sum += S
        temp_prod *= S
    arith_mean = temp_sum / n
    geo_mean = temp_prod ** (1 / n)

    return arith_mean, geo_mean


def geo_avg_put(S0, r, sigma, T, n, K):
    """

    Calculates the time-zero price of a geometric average Asian put opt
ion.
    """
    # Mean and variance for the log of the geometric mean based on prov
ided context
    mean_geo = (r - 0.5 * sigma ** 2) * (n + 1) / (2 * n)
    variance_geo = (sigma ** 2) * ((n + 1) * (2 * n + 1)) / (6 * n ** 2
)
    mu_hat = mean_geo + 0.5 * variance_geo
    sigma_hat = np.sqrt(variance_geo)
    d1_hat = (np.log(S0 / K) + (mu_hat + 0.5 * sigma_hat ** 2) * T) / (
sigma_hat * np.sqrt(T))
    d2_hat = d1_hat - sigma_hat * np.sqrt(T)
    # Put option price using the adapted formula
    put_price = np.exp(-r * T) * (K * norm.cdf(-d2_hat) - S0 * np.exp(m
u_hat * T) * norm.cdf(-d1_hat))

    return put_price
```

```python
def asian_put_cv(S0, r, sigma, T, n, K, N, p):
    np.random.seed(42)
    m = int(p * N)
    M = N - m
    temp_muA = 0
    temp_muG = 0
    temp_s2G = 0
    disc_AG = 0

    # Pilot run to estimate optimal parameters
    for _ in range(m):
        Aavg, Gavg = arith_geo_avgs(S0, r, sigma, T, n)
        disc_payoffA = np.exp(-r * T) * max(K - Aavg, 0)  # Put option
        disc_payoffG = np.exp(-r * T) * max(K - Gavg, 0)  # Put option
        disc_AG += disc_payoffA * disc_payoffG
        temp_muA += disc_payoffA
        temp_muG += disc_payoffG
        temp_s2G += disc_payoffG ** 2

    muA = temp_muA / m
    muG = temp_muG / m
    s2G = (temp_s2G / (m - 1)) - (m / (m - 1)) * muG ** 2
    chat = (disc_AG - m * muA * muG) / ((m - 1) * s2G)
    # Main CV estimator
    C0_Geo_True = geo_avg_put(S0, r, sigma, T, n, K)
    temp_muCV = 0
    temp_s2CV = 0

    for _ in range(M):
        Aavg, Gavg = arith_geo_avgs(S0, r, sigma, T, n)
        disc_payoffA = np.exp(-r * T) * max(K - Aavg, 0)  # Put option
        disc_payoffG = np.exp(-r * T) * max(K - Gavg, 0)  # Put option
        temp_CV = disc_payoffA - chat * (disc_payoffG - C0_Geo_True)
        temp_muCV += temp_CV
        temp_s2CV += temp_CV ** 2

    muCV = temp_muCV / M
    s2CV = (temp_s2CV / (M - 1)) - (M / (M - 1)) * muCV ** 2
    sCV = np.sqrt(s2CV)
    MSE = s2CV / M
    ci_error = 1.96 * sCV / np.sqrt(M)
    ci_lower = muCV - ci_error
    ci_upper = muCV + ci_error
    CI = (ci_lower, ci_upper)

    return N, muCV, MSE, CI

  # Example usage
S0 = 500  # Initial stock price
r = 0.0175  # Risk-free interest rate
sigma = 0.25  # Volatility
T = 1  # Time to maturity
n = 52  # Number of time steps (weekly)
K = S0  # Strike price
N = 10**5

# Get results
result = asian_put_cv(S0, r, sigma, T, n, K, N, 0.5)
```

```
# Convert results to a DataFrame
results_df = pd.DataFrame({
    'N': [result[0]],
    'Estimated Value (muCV)': [result[1]],
    'Mean Squared Error (MSE)': [result[2]],
    'Confidence Interval Lower Bound (cil)': [result[3][0]],
    'Confidence Interval Upper Bound (ciu)': [result[3][1]]
})

# Display all columns in the DataFrame
pd.set_option('display.max_columns', None)  # Show all columns
print(results_df)
```

```
##        (N)   Estimated Price     MSE      Confidence Interval

## 0   100000    26.67342      0.000028   (26.663141, 26.683699)
```

The low MSE of 0.000028 indicates a high precision level in the estimated price. This precision translates into a remarkable efficiency improvement of 47,968%, disregarding any computational error. In this instance, the control variate technique significantly enhanced the Monte Carlo simulation process.

Example Usage: Arithmetic Asian Option

- Shreve, S. E. (2003). Stochastic Calculus for Finance I: The Binomial Asset Pricing Model. Springer.
- Van der Hoek, J., & Elliott, R. J. (2006). Binomial Models in Finance. Springer.
- Derman, E., & Kani, I. (1994). The volatility smile and its implied tree. Goldman Sachs Quantitative Strategies Research Notes.
- Hindman, C. MACF 402 (Mathematical and Computational Finance II): Implied Volatility Trees. [Slides].