IE522 Project Report

Group Members: Yixuan Huo, Yiwen Zhang, Chayapon Lee-Isranukul, Michael Wattelet

1. Introduction

Accuracy in pricing options is critical as inaccurate prices may lead to inefficiencies in the market. Inaccurate option prices can create arbitrage opportunities or deter investors, resulting in illiquid markets. Monte Carlo simulation plays a key roll in option pricing as it provides a method for prcing complex derivatives when an analytical solution is not available. Monte Carlo simulation estimates the value of a financial instrument by simulating an array of potential price trajectories. The standard Monte Carlo approach uses random sampling to better model uncertanties and then computes the expected payoff based on these modeled uncertanties. This expected payoff is then discounted to present time to obtain the present value of the price trajectory. This process is iteratively simulated many times, where the payoff of each price trajectory is calculated. In each iteration, the sample mean and updated and stored, reducing excess memory usage. The sample mean and squared sample mean are sufficient to calculate the final option price and its variance or standard error.

This report uses Monte Carlo simulation to price three types of options: European vanilla put options, Asian call options, and American put options. The European vanilla put option prices are calculated in the Black-Scholes-Merton model and using the standard Monte Carlo approach. In addition, the antithetic approach is also used to reduce variance and improve efficiency. This is completed by generating pairs of negatively correlated random variables, allowing for more accurate results with fewer simulations. The Quasi-Monte Carlo approach (QMC) is also implemented to improve efficiency with faster convergence. This is completed by using low-discrepancy sequences instead of random sampling for the generation of sampling points. In pricing the European vanilla put option, the sample size, option price, estimated standard error, 95% confidence interval, and the absolute pricing error are reported. Convergence between the different methods is also investigated.

To price the Asian call option in the Black-Scholes-Merton model, the standard Monte Carlo approach is used. In addition, to reduce variance, the control variate approach is used with geometric Asian call as a control. The geometric Asian call has a closed form solution with known expected value and is correlated with the price of the true Asian call option. The use of this additional control variate variable allows for final values to be more precise with smaller samples. In addition, the moment method approach is utilized to increase computational efficiency by using indirect simulation methods, reducing the need for direct simulation. In pricing the Asian call options, he sample size, option price, estimated standard error, 95% confidence interval, and computational time is reported for each Monte Carlo simulation. Also, the convergence of the Monte Carlo methods is investigated as the number of samples, N, increases.

To price the American vanilla option, the Longstaff-Schwartz method is employed using Monte Carlo simulation. This method uses regression to estimate early exercise strategy, with the value of the option being the average of the many simulated price trajectories. Different regressors, sample sizes, and time steps are investigated to see how they effect the option price. The Binomial Black-Scholes with Richardson Extrapolation (BBSR) method is also implemented and used as a high-accuracy benchmark for comparison. The BBSR method refines the binomial tree approach, improving accuracy with fewer time steps.

2. European Vanilla Put Option

2.1 Standard Monte Carlo Approach

To price a European Vanilla put option, Monte Carlo methods can be employed. The Black-Scholes-Merton (BSM) model is assumed for the underlying asset price and is shown in the below equation.

$$S_t = S_0 \exp\left(\left(r - q - \frac{1}{2}\sigma^2\right)t + \sigma B_t\right), \quad 0 \leq t \leq T$$

Table 2.1: Parameter Descriptions

Parameter	Description
S_t	The price of the asset at time t
S_0	Initial asset price
r	Risk-free interest rate per year with continuous compounding
q	Continuous yield of the asset
σ	Volatility per year for the underlying asset
B_t	Standard Brownian motion
t	Time, with $0 \leq t \leq T$
T	The maturity time

The price of the European vanilla put option with strike price K and maturity T is given in the below equation. Because a European vanilla put option is being priced, the value of the put option will only be positive if the underlying asset price S_t is less than the strike price K at maturity.

$$p = e^{-rT}\mathbb{E}\left[\max(0, K - S_T)\right]$$

Both equations 1 and 2 are used in the following Monte Carlo simulation. This Monte Carlo simulation outputs European vanilla put option prices, the estimated standard error of the put option prices, the 95% confidence interval, and the absolute pricing error, which compares the simulated option price to the exact option price. Each simulation is run for an inputted sample size (N) and the corresponding computation time is calculated for each sample size. For this problem, the parameters are assumed to have the values shown in Table 2.

Table 2.2: Parameters Used in the BSM Model

Parameter	Value
S_0	100
K	100
T	0.5
r	0.04
q	0.02
σ	0.2

```
EuropeanPut <- function(n, alpha = 0.05){
  S0 = 100 # stock price
  K = 100 # strike price
  t = 0.5 # time to maturity
  r = 0.04 # risk free rate
  q = 0.02 # dividend yield
  sigma = 0.2 # implied volatility

  start_time <- Sys.time()
```

```
  df = exp(-r*t)
  Z = rnorm(n) #calculate n standard random normal variables, store in vector

  x_bar = 0
  y_bar = 0

  for (k in 1:n){
    S_T = S0*exp((r - q - 1/2*sigma^2)*t + sigma*sqrt(t)*Z[k])
    p = df * max(0, K - S_T)

    x_bar = (1 - 1/k)*x_bar + 1/k*p
    y_bar = (1 - 1/k)*y_bar + 1/k*p^2
  }

  end_time <- Sys.time()

  se <- sqrt((y_bar - x_bar^2) / (n - 1))

  z_alpha <- qnorm(1 - alpha/2)
  lower_bound <- x_bar - z_alpha * se
  upper_bound <- x_bar + z_alpha * se

  d1 <- (log(S0/K) + (r - q + 1/2*sigma^2)*t) / (sigma * sqrt(t))
  d2 <- (log(S0/K) + (r - q - 1/2*sigma^2)*t) / (sigma * sqrt(t))
  exact_put_price <- K * exp(-r*t) * pnorm(-d2) - S0 * exp(-q*t) * pnorm(-d1)

  absolute_price_error = abs(x_bar - exact_put_price)


  comp_time = as.numeric(difftime(end_time, start_time, units = "secs"))

  results <- data.frame(
    "Sample Size" = n,
    "Option Price" = x_bar,
    "Standard Error" = se,
    "Lower Bound CI" = lower_bound,
    "Upper Bound CI" = upper_bound,
    "Exact Put Price" = exact_put_price,
    "Absolute Error" = absolute_price_error,
    "Time (seconds)" = comp_time
  )
  return(results)
}
```

In the above code, the function "EuropeanPut" runs a Monte Carlo simulation to price a European vanilla put option, with input sample size N. In the function, the parameters are initialized, and the computation start time is stored in the "start_time" variable. "Z" is a vector of size n that stores n standard random normal generated values. It should be noted that standard random normal values are used because Brownian motion is a stochastic process that exhibits normally distributed increments with mean equal to zero and variance that is proportional to the time interval. For each small time step, it is thus sufficient to generate random samples from the standard normal distribution and scale them by the square root of the time step

to simulate Brownian motion. In the above code, the equation for S_t is thus altered from the first equation.

$$S_t = S_0 \exp\left(\left(r - q - \frac{1}{2}\sigma^2\right)t + \sigma\sqrt{t}Z_t\right)$$

The value S_t is generated using the first value from Z and inputted into equation 2 to generate the put option price. This price is stored as variable p. The put option price, p, is then used in the calculation for x_bar, which keeps track of the mean of the put option price across all simulations. Additionally, the square of the put option price, p^2, is used to calculate y_bar, which tracks the squared mean of the put option price. This process constitutes one iteration of the Monte Carlo simulation, and is iterated N times. It is important to note that through Monte Carlo simulation, the means can be computed without the need to store all previous values of the put option price. The stored means are updated with each iteration, reducing memory usage and computation time. x_bar and y_bar are updated iteratively in the loop as shown below.

$$\bar{x} = \left(1 - \frac{1}{k}\right)\bar{x} + \frac{1}{k}p$$

$$\bar{y} = \left(1 - \frac{1}{k}\right)\bar{y} + \frac{1}{k}p^2$$

It is important to note that the square of the put option price, y_bar, is computed in order to easily compute the standard error of the simulated put option price. This standard error is computed as shown below, after all iterations are complete.

$$\text{se} = \sqrt{\frac{\bar{y} - \bar{x}^2}{n-1}}$$

Following the calculation of the standard error, the lower bound and upper bounds of the 95% confidence interval are computed.

$$\text{lower\_bound} = \bar{x} - z_\alpha \cdot \text{se}$$

$$\text{upper\_bound} = \bar{x} + z_\alpha \cdot \text{se}$$

Next, the absolute pricing error is computed, which is the absolute value of the difference between the simulated put option price and the exact option price. The price of a European put option has a closed-form solution under the Black-Scholes model, allowing for a straightforward calculation with provided input parameters.

$$p_{\text{exact}} = K\exp(-rt)\Phi(-d_2) - S_0\exp(-qt)\Phi(-d_1)$$

$$d_1 = \frac{\log(S_0/K) + \left(r - q + \frac{1}{2}\sigma^2\right)t}{\sigma\sqrt{t}}$$

$$d_2 = \frac{\log(S_0/K) + \left(r - q - \frac{1}{2}\sigma^2\right)t}{\sigma\sqrt{t}}$$

The calculation of the absolute pricing error is shown below.

$$\text{absolute\_price\_error} = |\bar{x} - p_{\text{exact}}|$$

The computation end time is also recorded after the simulation is complete and the total computation time for the Monte Carlo Simulation for a given sample size, N, is calculated. All outputs are stored in a data frame, and the results for Monte Carlo Simulations with different sample sizes are shown below.

```
result_table <- function(sample_sizes, alpha=0.05){
  results <- data.frame(matrix(ncol = 8, nrow = 0))
  colnames(results) <- c("N", "Option Price", "Std. Error", "LB CI", "Upper Bound CI",
                         "Exact Price", "Absolute Error", "Time (seconds)")

  for (N in sample_sizes){
    result <- EuropeanPut(N, alpha=0.05)
```

```
    results <- rbind(results, result)
  }
  results
}

sample_sizes <- c(1000, 4000, 16000, 64000, 256000, 1024000)
results_standard <- result_table(sample_sizes, alpha=0.05)

write.csv(results_standard, file = "ResultsStandard.csv", row.names = FALSE)
```

Table 2.3: Standard Monte Carlo Simulation Results

Sample.Size	Option.Price	Standard.Error	Lower.Bound.CI	Upper.Bound.CI	Exact.Put.Price	Absolute.Error	Time..seconds.
1000	4.802281	0.2095881	4.391495	5.213066	5.074637	0.2723560	0.0008960
4000	5.064871	0.1147066	4.840050	5.289692	5.074637	0.0097654	0.0033052
16000	5.201631	0.0576645	5.088610	5.314651	5.074637	0.1269940	0.0139370
64000	5.090333	0.0284555	5.034562	5.146105	5.074637	0.0156970	0.0690742
256000	5.080220	0.0142115	5.052366	5.108074	5.074637	0.0055831	0.2887089
1024000	5.080600	0.0070984	5.066687	5.094512	5.074637	0.0059630	0.7177570

The exact put option price from the Black-Scholes-Merton model is \$5.074637. The put option price generated from the Monte Carlo simulation is within \$0.01 the exact put option price when the sample size is around 256,000 samples. This means, for this problem, to be extremely accurate, one would need to run at least 256,000 iterations to be extremely accurate. Further discussion on the results of the standard Monte Carlo approach is included in the discussion section. At 1,024,000 samples, the estimated option price is \$5.081, whith an absolute price error of \$0.00596. A sample size of 1,024,000 also has a standard error of 0.007, leading to a very precise result.

2.2. Antithetic Monte Carlo Approach

To reduce the variance in the estimated European put option price, the antithetic approach is utilized. By reducing variance, the antithetic approach can achieve the same accuracy as the standard Monte Carlo approach with a smaller sample size. This in turn leads to faster computational efficiency. The antithetic Monte Carlo approach starts by generating a standard normal random value and its "antithetic" counterpart, which is the negative value of the generated standard normal random value.

$$Z_{i1} = \text{a random variable such that } Z_{i1} \sim N(0,1)$$

$$Z_{i2} = -Z_{i1}$$

It should be noted that the antithetic approach has the same sample mean as the standard Monte Carlo approach. Z_{i1} and its "antithetic" counterpart Z_{i2}, are symmetrically distributed with the expectation of Z_{i1} being equal to the expectation of Z_{i2}.

$$\mathbb{E}[f(Z_1)] = \mathbb{E}[f(-Z_1)]$$

By averaging f(Z1) and f(Z2), the expected value of the function remains unbiased, ensuring the expected value of the function in the antithetic approach is equal to expected value of the function in the standard approach.

$$\mathbb{E}\left[\frac{f(Z_1) + f(Z_2)}{2}\right] = \mathbb{E}[f(Z)]$$

The variance in the antithetic approach is less than the standard approach due to the fact that f(Z1) and f(Z2) are negatively correlated. The expression for the standard approach variance is shown in the following expression.

$$\text{Var}(\hat{\mu}) = \frac{1}{n}\text{Var}(f(Z))$$

The variance of the antithetic approach is shown in the following expression.

$$\text{Var}(\hat{\mu}) = \frac{1}{4n}\left(\text{Var}(f(Z1)) + \text{Var}(f(Z2)) + 2\text{Cov}(f(Z1), f(Z2))\right)$$

The negative correlation between f(Z1) and f(Z2) leads to the antithetic approach having the lower variance. The code block below shows the antithetic approach used to price the European put option.

```
EuropeanPut_antithetic <- function(n, alpha) {
  S0 = 100  # stock price
  K = 100   # strike price
  t = 0.5   # time to maturity
  r = 0.04  # risk-free rate
  q = 0.02  # dividend yield
  sigma = 0.2  # implied volatility

  start_time <- Sys.time()

  Z1 = rnorm(n/2)
  Z2 = -Z1
  df = exp(-r*t)

  x_bar = 0
  y_bar = 0

  for (k in 1:(n/2)) {
    S_T1 = S0*exp((r - q - 1/2*sigma^2)*t + sigma*sqrt(t)*Z1[k])
    S_T2 = S0*exp((r - q - 1/2*sigma^2)*t + sigma*sqrt(t)*Z2[k])

    p1 = df * max(0, K-S_T1)
    p2 = df * max(0, K-S_T2)

    x = (p1 + p2) / 2

    x_bar = (1 - 1/k)*x_bar + 1/k*x
    y_bar = (1 - 1/k)*y_bar + 1/k*x^2
  }

  end_time <- Sys.time()

  se <- sqrt((y_bar - x_bar^2) / (n/2 - 1))

  z_alpha <- qnorm(1 - alpha/2)
  lower_bound <- x_bar - z_alpha * se
  upper_bound <- x_bar + z_alpha * se

  d1 <- (log(S0/K) + (r - q + 1/2*sigma^2)*t) / (sigma * sqrt(t))
  d2 <- (log(S0/K) + (r - q - 1/2*sigma^2)*t) / (sigma * sqrt(t))
  exact_put_price <- K * exp(-r*t) * pnorm(-d2) - S0 * exp(-q*t) * pnorm(-d1)

  absolute_price_error = abs(x_bar - exact_put_price)


  comp_time = as.numeric(difftime(end_time, start_time, units = "secs"))
```

```
  result_row <- data.frame(
    "Sample Size (n)" = n,
    "Anithetic Sample Size (n/2)" = n/2,
    "Option Price" = x_bar,
    "Standard Error" = se,
    "Lower CI" = lower_bound,
    "Upper CI" = upper_bound,
    "Exact Put Price" = exact_put_price,
    "Absolute Error" = absolute_price_error,
    "Time (seconds)" = comp_time
  )
  return(result_row)
}
```

Because the antithetic approach involves generating Z1 and then computing the negative of Z1 to get Z2, only n/2 random values are generated to create n samples. Thus, The for loop: for (k in 1:(n/2)) needs to be iterated only n/2 times as each iteration processes a pair of samples (Z1 and Z2). For each iteration, a stock price, S_{T1} and S_{T2}, is generated for both Z1 and Z2. The payoff, p1 and p2, for each stock price is also computed and the average of the payoffs is stored in x. Similarly to the standard approach, running average of the mean payoff x_{bar} is computed and stored along with the square of the mean payoff x_{bar} as shown in the equations below.

$$\bar{x} = \left(1 - \frac{1}{k}\right)\bar{x} + \frac{1}{k}x$$

$$\bar{y} = \left(1 - \frac{1}{k}\right)\bar{y} + \frac{1}{k}x^2$$

The standard error, confidence intervals, and absolute price error are computed as shown in the standard approach, with the correct sample size (n/2) used for the calculation of the antithetic standard error. The results for the antithetic approach are shown below.

```
convergence_table <- function(sample_sizes, alpha){
  results <- data.frame(matrix(ncol = 9, nrow = 0))
  colnames(results) <- c("Sample Size (n)", "Antithetic Sample Size (n/2)", "Option Price",
                         "Standard Error", "Lower CI", "Upper CI",
                         "Exact Put Price", "Absolute Error", "Time (seconds)")

  for (N in sample_sizes){
    result <- EuropeanPut_antithetic(N, alpha)
    results <- rbind(results, result)
  }
  return(results)
}

# Define N values and alpha
sample_sizes <- c(1000, 4000, 16000, 64000, 256000, 1024000)
alpha <- 0.05

# Generate the table
results_antithetic <- convergence_table(sample_sizes, alpha)
write.csv(results_antithetic, file = "ResultsAntithetic.csv", row.names = FALSE)
```

Table 2.4: Antithetic Monte Carlo Simulation Results

Sample.Size..n.	Anithetic.Sample.Size..n.2.	Option.Price	Standard.Error	Lower.CI	Upper.CI	Exact.Put.Price	Absolute.Error	Time..seconds.
1000	500	5.345560	0.1688253	5.014669	5.676452	5.074637	0.2709237	0.0007942
4000	2000	5.068155	0.0812799	4.908849	5.227461	5.074637	0.0064814	0.0034051
16000	8000	5.037366	0.0397200	4.959516	5.115216	5.074637	0.0372707	0.0136590
64000	32000	5.073101	0.0200428	5.033818	5.112384	5.074637	0.0015354	0.0620248
256000	128000	5.068351	0.0100614	5.048631	5.088071	5.074637	0.0062856	0.2079480
1024000	512000	5.073578	0.0050239	5.063731	5.083424	5.074637	0.0010588	0.6335568

Note the antithetic approach converges to within 1 cent of the exact European put price when the sample size is on average 64,000 (32,000 antithetic pairs generated). The absolute price error is 0.0015, and continues to be within 1 cent of the exact price as sample size increases. This convergence is quicker compared to the standard approach where the convergence to within 1 cent of the exact European put price took 256,000 samples (n). See discussion section for further discussion of results.

2.3: Quasi-Monte-Carlo Approach

To further increase the speed of the convergence, the Quasi-Monte-Carlo approach can be utilized. The Quasi-Monte-Carlo approach differs from the standard approach in how the samples are generated. The standard approach utilizes random sampling, while the Quasi-Monte-Carlo approach uses low discrepancy sequences, such as Sobol, to generate samples. The low discrepancy sequences are more uniformly spread across the sample space, leading to faster convergences when compared to random sampling. Note that Sobol sequences are deterministic, meaning for a given set of inputs, the same sequence of samples will be generated. This allows the convergence for Quasi-Monte-Carlo to be $O\left(\frac{1}{n}\right)$ compared to the standard approach with convergence $O\left(\frac{1}{\sqrt{n}}\right)$. The Quasi-Monte-Carlo approach is shown below.

```
EuropeanPutQMC <- function(n, alpha) {
  S0 = 100 # stock price
  K = 100 # strike price
  t = 0.5 # time to maturity
  r = 0.04 # risk-free rate
  q = 0.02 # dividend yield
  sigma = 0.2 # implied volatility

  start_time <- Sys.time()

  # Generate Sobol sequence
  Z = qnorm(suppressWarnings(sobol(n, dim = 1, scrambling = 1, seed = 42)))

  df = exp(-r * t)

  x_bar = 0
  y_bar = 0

  for (k in 1:n) {
    S_T = S0 * exp((r - q - 1/2 * sigma^2) * t + sigma * sqrt(t) * Z[k])
    p = df * max(0, K - S_T)

    x_bar = (1 - 1/k) * x_bar + 1/k * p
    y_bar = (1 - 1/k) * y_bar + 1/k * p^2
  }

  end_time <- Sys.time()

  se <- sqrt((y_bar - x_bar^2) / (n - 1))
  z_alpha <- qnorm(1 - alpha/2)
```

```r
  lower_bound <- x_bar - z_alpha * se
  upper_bound <- x_bar + z_alpha * se

  d1 <- (log(S0/K) + (r - q + 1/2 * sigma^2) * t) / (sigma * sqrt(t))
  d2 <- (log(S0/K) + (r - q - 1/2 * sigma^2) * t) / (sigma * sqrt(t))
  exact_put_price <- K * exp(-r * t) * pnorm(-d2) - S0 * exp(-q * t) * pnorm(-d1)

  absolute_price_error <- abs(x_bar - exact_put_price)


  comp_time = as.numeric(difftime(end_time, start_time, units = "secs"))

  result_row <- data.frame(
    "Sample Size" = n,
    "Option Price" = x_bar,
    "Standard Error" = se,
    "Lower CI" = lower_bound,
    "Upper CI" = upper_bound,
    "Exact Put Price" = exact_put_price,
    "Absolute Error" = absolute_price_error,
    "Time (seconds)" = comp_time
  )
  return(result_row)
}

sample_sizes <- c(1000, 4000, 16000, 64000, 256000, 1024000)
alpha <- 0.05

convergence_table <- function(sample_sizes, alpha){
  results <- data.frame(matrix(ncol = 8, nrow = 0))
  colnames(results) <- c("Sample Size", "Option Price", "Standard Error", "Lower CI",
                         "Upper CI", "Exact Put Price", "Absolute Error", "Time (seconds)")

  for (N in sample_sizes){
    result <- EuropeanPutQMC(N, alpha)
    results <- rbind(results, result)
  }
  return(results)
}

results_QMC <- convergence_table(sample_sizes, alpha)
write.csv(results_QMC, file = "ResultsQMC.csv", row.names = FALSE)
```

Table 2.5: Quasi-Monte-Carlo Simulation Results

Sample.Size	Option.Price	Standard.Error	Lower.CI	Upper.CI	Exact.Put.Price	Absolute.Error	Time..seconds.
1000	5.052071	0.2249903	4.611098	5.493044	5.074637	0.0225656	0.0013549
4000	5.071074	0.1133398	4.848932	5.293216	5.074637	0.0035625	0.0038731
16000	5.074067	0.0567705	4.962799	5.185335	5.074637	0.0005698	0.0104291
64000	5.074551	0.0283965	5.018895	5.130207	5.074637	0.0000852	0.0415630
256000	5.074626	0.0141995	5.046796	5.102457	5.074637	0.0000103	0.1855099
1024000	5.074636	0.0070999	5.060721	5.088552	5.074637	0.0000004	0.7902770

The total number of samples required to be within 1 cent of the exact option price is around 4000 samples. The absolute price error is 0.0035 and continues to decrease as sample size increases. Note this is much lower compared to the standard (256,000) and antithetic approaches (64,000 (32,000 antithetic pairs)). However, at 4000 samples the standard error is 11 cents, which is not as precise. If more precision is needed, sample size will need to be increased. See discussion section for further analysis of results.

2.4. Discussions

The below plot shows the estimated European Put option price for each method.

```
ggplot() +
  # Add results_standard
  geom_line(data = results_standard, aes(x = Sample.Size, y = Option.Price),
            color = "blue") +
  geom_point(data = results_standard, aes(x = Sample.Size, y = Option.Price),
             color = "blue") +

  # Add results_antithetic
  geom_line(data = results_antithetic, aes(x = Sample.Size..n., y = Option.Price),
            color = "red") +
  geom_point(data = results_antithetic, aes(x = Sample.Size..n., y = Option.Price),
             color = "red") +

  # Add results_QMC
  geom_line(data = results_QMC, aes(x = Sample.Size, y = Option.Price),
            color = "green") +
  geom_point(data = results_QMC, aes(x = Sample.Size, y = Option.Price),
             color = "green") +

  geom_line(data = results_standard, aes(x = Sample.Size, y = Exact.Put.Price),
            color = "black", linetype = "dashed") +

  scale_x_continuous(trans = "log10", labels = scales::comma) +

  labs(
    title = "Put Option Price Convergence for Different Methods vs Sample Size",
    x = "Sample Size (log scale)",
    y = "Option Price"
  ) +
  theme_minimal()
```

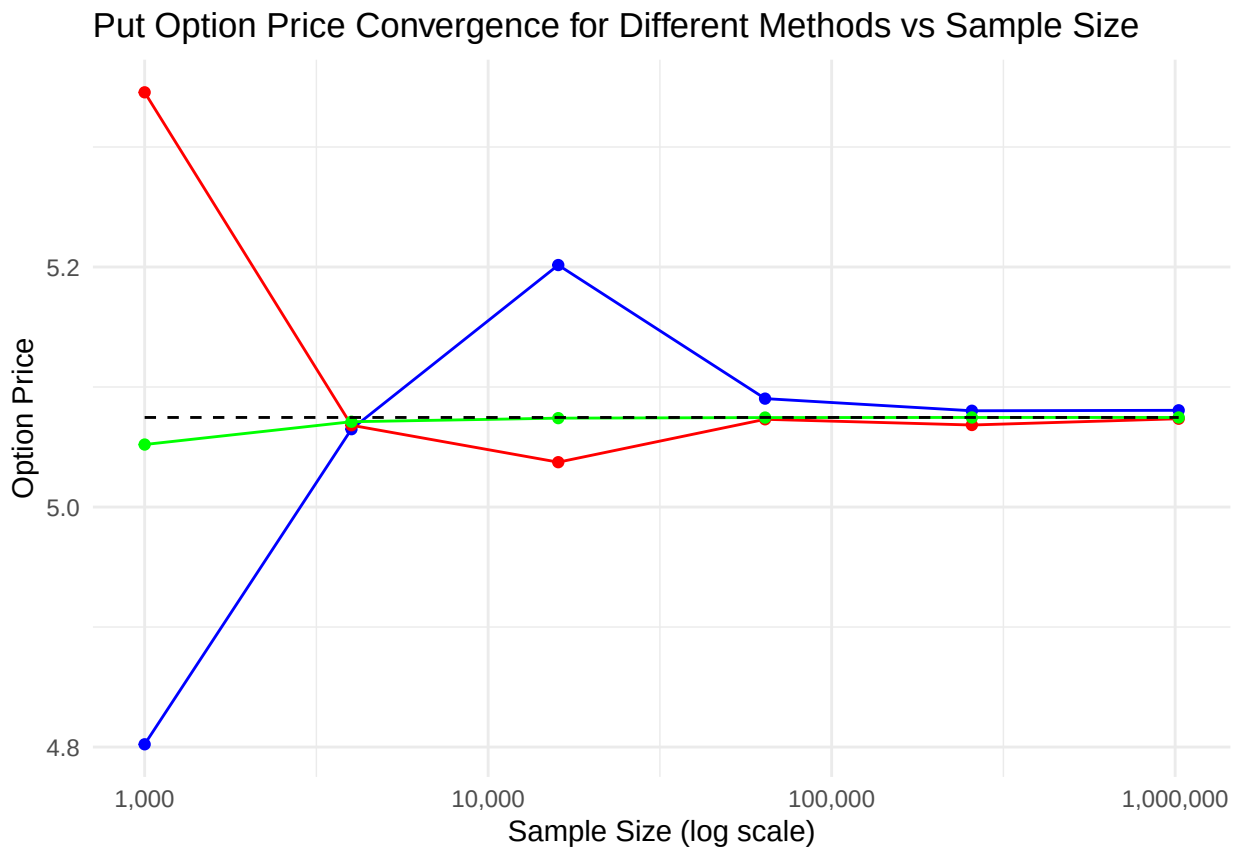


Figure 2.1: Put Option Convergence

It can be seen from the above plot that the Quasi-Monte-Carlo method had the fastest convergence to the exact price by sample size (4000), followed by the antithetic approach (64000, (32000 antithetic pairs)), and then the standard approach (256,000). The below graph shows the absolute price error of the three methods, giving more insight to the convergence. The QMC approach had the fastest convergence, followed by the antithetic and standard approaches. Note that QMC has the smallest absolute price error for all sample sizes, followed by on average the antithetic approach and the standard approach.

```
ggplot() +
  # Add results_standard
  geom_line(data = results_standard, aes(x = Sample.Size, y = Absolute.Error),
            color = "blue") +
  geom_point(data = results_standard, aes(x = Sample.Size, y = Absolute.Error),
             color = "blue") +

  # Add results_antithetic
  geom_line(data = results_antithetic, aes(x = Sample.Size..n., y = Absolute.Error),
            color = "red") +
  geom_point(data = results_antithetic, aes(x = Sample.Size..n., y = Absolute.Error),
             color = "red") +

  # Add results_QMC
  geom_line(data = results_QMC, aes(x = Sample.Size, y = Absolute.Error),
            color = "green") +
```

```
  geom_point(data = results_QMC, aes(x = Sample.Size, y = Absolute.Error),
             color = "green") +

  geom_line(data = results_standard, aes(x = Sample.Size, y = 0.01),
            color = "black", linetype = "dashed") +

  scale_x_continuous(trans = "log10", labels = scales::comma) +

  labs(
    title = "Abs. Price Error for Different Methods vs Sample Size",
    x = "Sample Size (log scale)",
    y = "Absolute Price Error"
  ) +
  theme_minimal()
```

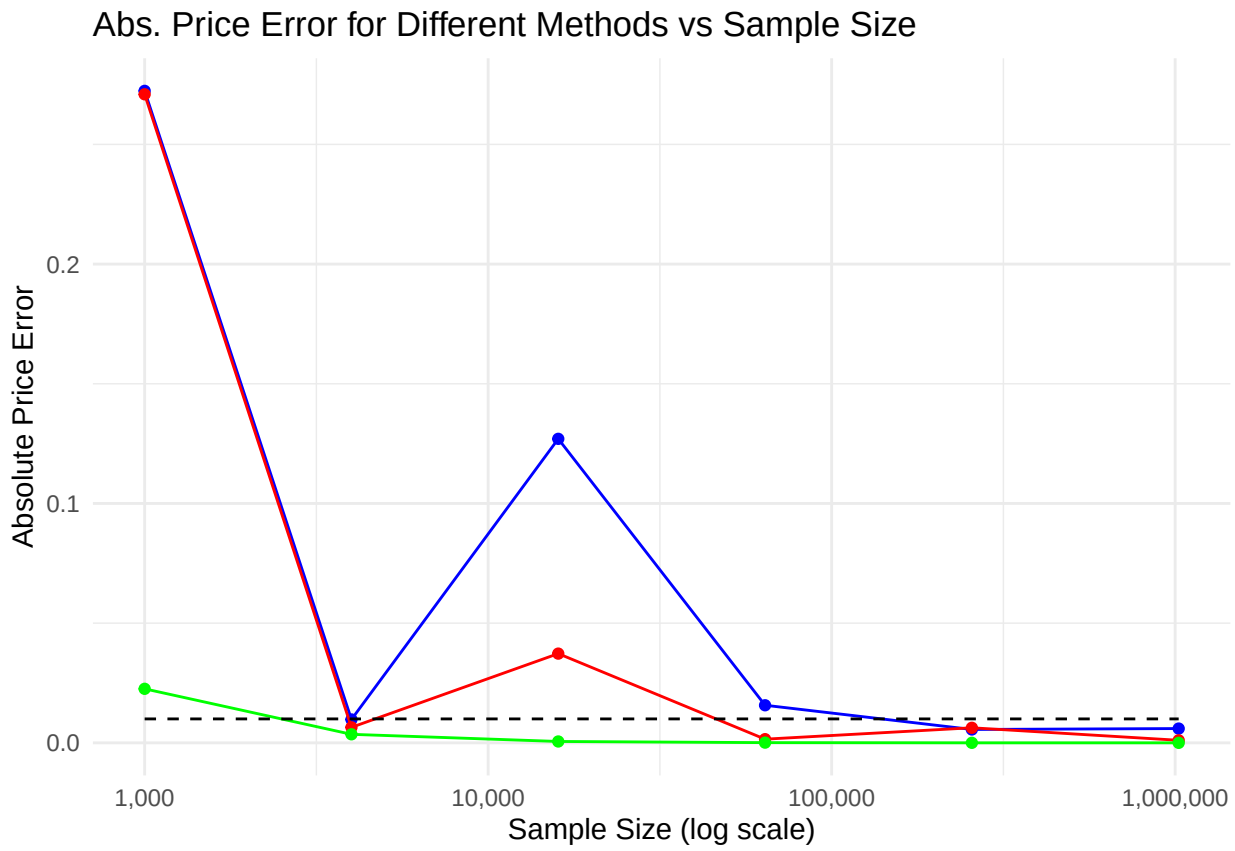


Figure 2.2: Absolute Price Error

The below graph shows the comparison between computation time between the three methods.

```
df_standard <- results_standard %>%
  select(Sample.Size, Time..seconds.) %>%
  mutate(Method = "Standard")

df_antithetic <- results_antithetic %>%
```

```r
  select(Sample.Size..n., Time..seconds.) %>%
  rename(Sample.Size = Sample.Size..n.) %>%  # Rename the column to Sample.Size
  mutate(Method = "Antithetic")

df_QMC <- results_QMC %>%
  select(Sample.Size, Time..seconds.) %>%
  mutate(Method = "QMC")

# Combine all methods into one data frame
df_all <- bind_rows(df_standard, df_antithetic, df_QMC)

ggplot(df_all, aes(x = factor(Sample.Size), y = Time..seconds., fill = Method)) +
  geom_bar(stat = "identity", position = "dodge") +
  facet_wrap(~ Method, scales = "free_x", ncol = 3) +  # Facet by Method, with free x-axis scale
  labs(
    title = "Computation Time for Different Methods vs Sample Sizes",
    x = "Sample Size",
    y = "Computation Time (seconds)"
  ) +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))  # Rotate x-axis labels
```

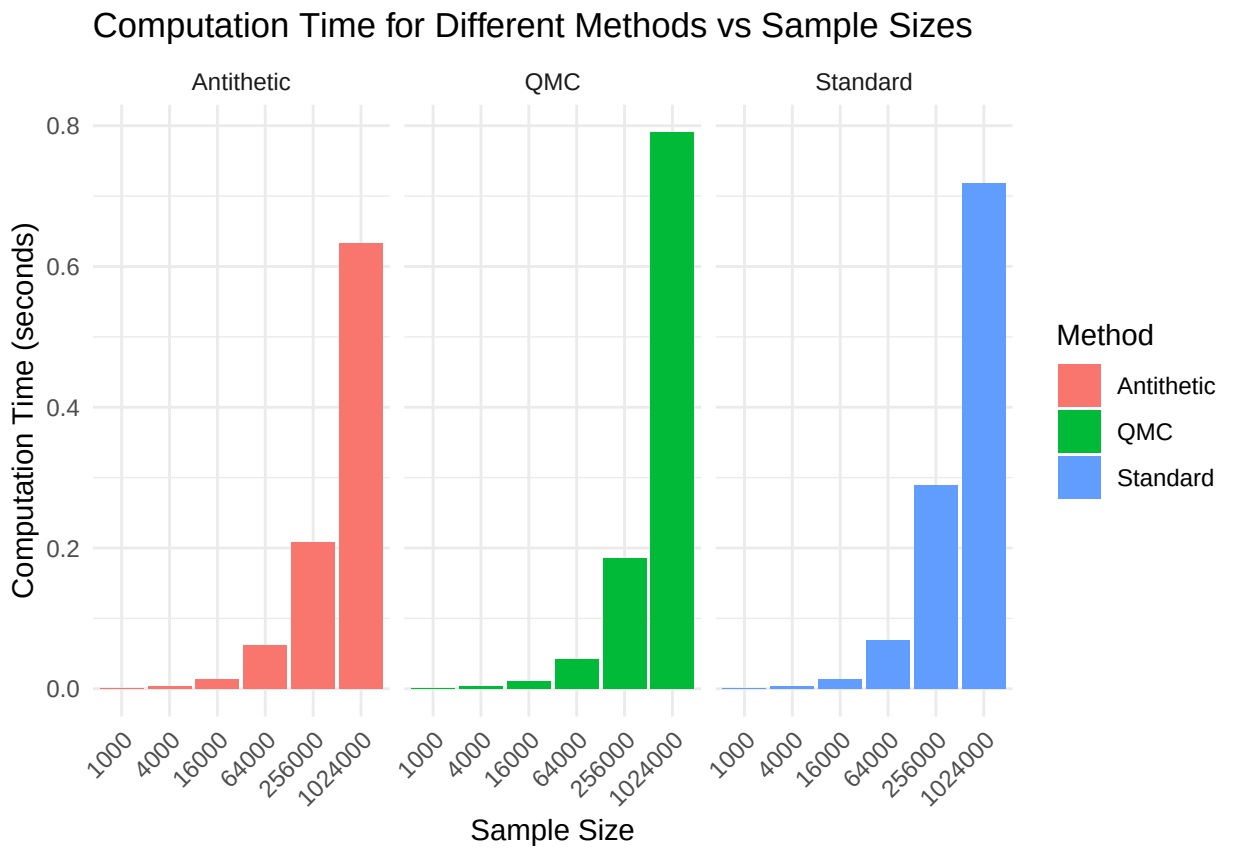


Figure 2.3: Computation Time

In this case, the Antithetic approach was the fastest computationally for running n independent samples mainly because the antithetic approach generates n/2 random independent samples for the standard Monte

Carlo n. This is due to the fact that the negative "antithetic" counterpart of the n/2 samples is also generated leading to a total of n samples. Although the Antithetic approach has the smallest computation time for a given number of samples, the Quasi-Monte-Carlo method is still faster at converging to the exact put option price as it requires fewer samples to do so. The Quasi-Monte-Carlo method had on average the computationally fastest convergence to a 1 cent error in option price (0.0038 seconds), compared to the antithetic approach (0.062 seconds), and the standard approach (0.2887 seconds). Because Monte Carlo Simulation is used, these values can slightly vary. Note that the standard Monte Carlo approach is the slowest method in convergence by sample size, leading to the slowest computational speed. This will not be the preferred method if the antithetic or QMC approaches are able to be used.

From all methods, it can be seen that the standard error decreases proportionally relative to the square root of the number of simulations.

$$SE \propto \frac{1}{\sqrt{n}}$$

This means in order to decrease the standard error by half, the sample size N would need to increase by 4. A plot of the standard error with respect to the number of simulations is shown in the below plot.

```
ggplot() +
  # Add results_standard
  geom_line(data = results_standard, aes(x = Sample.Size, y = Standard.Error),
            color = "blue") +
  geom_point(data = results_standard, aes(x = Sample.Size, y = Standard.Error),
             color = "blue") +

  # Add results_antithetic
  geom_line(data = results_antithetic, aes(x = Sample.Size..n., y = Standard.Error),
            color = "red") +
  geom_point(data = results_antithetic, aes(x = Sample.Size..n., y = Standard.Error),
             color = "red") +

  # Add results_QMC
  geom_line(data = results_QMC, aes(x = Sample.Size, y = Standard.Error),
            color = "green") +
  geom_point(data = results_QMC, aes(x = Sample.Size, y = Standard.Error),
             color = "green") +

  scale_x_continuous(trans = "log10", labels = scales::comma) +

  labs(
    title = "Standard Error for Different Methods vs Sample Size",
    x = "Sample Size (log scale)",
    y = "Standard Error"
  ) +
  theme_minimal()
```

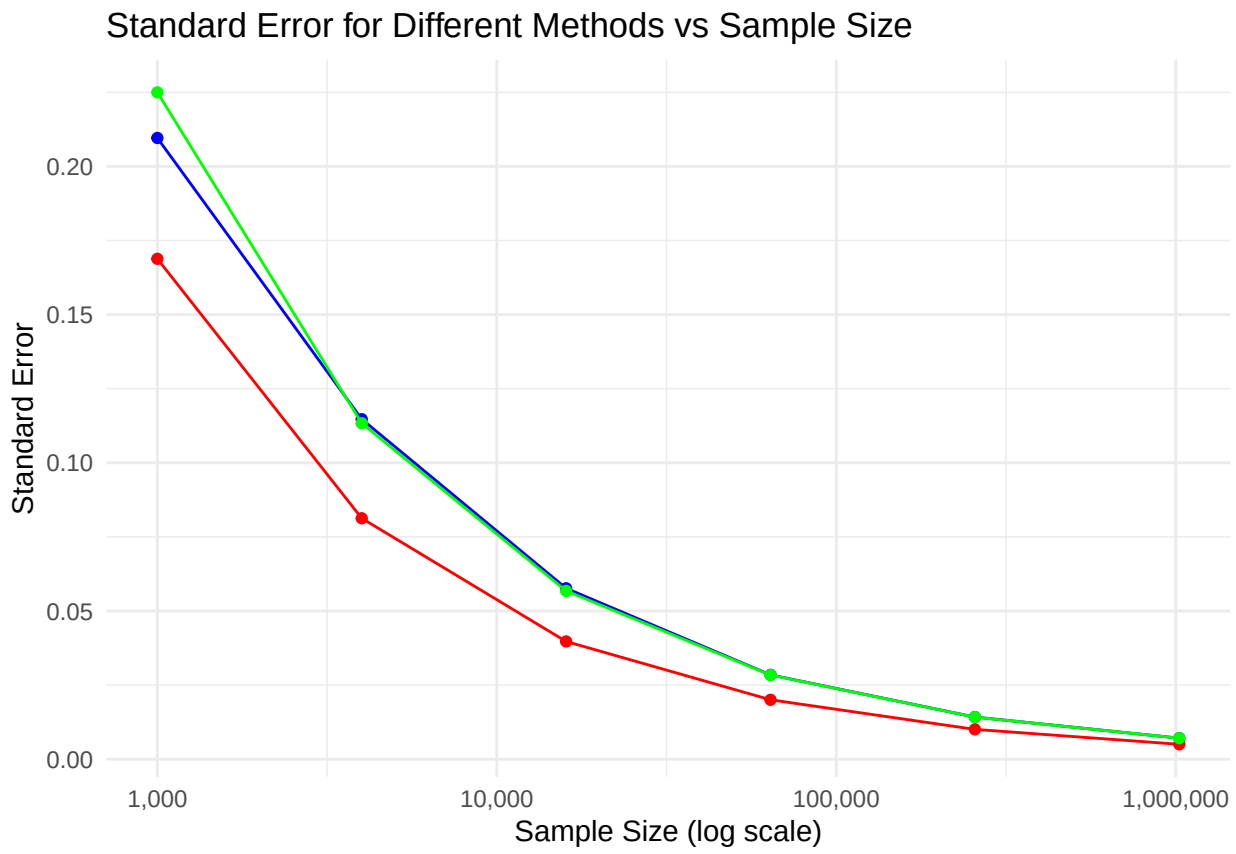


Figure 2.4: Standard Error

Note that the sample size is quadrupled between data points, therefore the standard error decreases by half between the data points. This is also shown in the confidence intervals as the confidence intervals decrease in size by half as the standard error decreases by half. Also note that the "Antithetic" standard error is much lower for each sample size as there is a negative correlation between the generated pairs.

It is determined that the most accurate method is the Quasi-Monte Carlo approach, followed by the antithetic approach and the standard approach.

2.5. Reduction in Computational Time

Computational time was first reduced by calculating the discount factor outside of the iterative for loop, instead of recalculated it inside the loop. The discount factor is able to be pulled outside of the loop due to the fact that the assumptions of the risk-free rate and time to maturity are constant. This discount factor is stored in the variable "df".

The second way that computational time was reduced was by storing n standard random normal generated values in vector "Z" instead of computing a standard random normal value in each iteration of the Monte Carlo simulation. This is more efficient as memory does not need to be repeatedly allocated inside the loop and the "rnorm" function does not need to be repeatedly called. This is especially important when large sample sizes are used for simulation.

These methods of computing the discount factor outside of the loop and storing the standard random normal generated values in vector Z (deterministic for QMC) were utilized in all three models. The antithetic approach and Quasi-Monte-Carlo approach had large improvements in computational time compared to the standard approach as faster convergence to the European put option price was observed. This is largely due to a reduction in variance from the antithetic approach and deterministic sampling from the Quasi-Monte-Carlo approach.

3. Monte Carlo Simulation for pricing Asian Call Option

3.1 Introduction

An Asian option is a type of exotic option where the payoff depends on the average price of the underlying asset over a certain period, rather than its price at a specific point in time. This average can be computed as either an arithmetic average or a geometric average. Asian options are particularly useful in markets where the underlying asset's price is highly volatile, as they reduce the impact of price manipulation or extreme price movements near expiration. For an Asian call or put option, the payoff formulas are listed as follows.

The payoff for a continuous arithmetic average Asian call or put option is:

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

The payoff for a continuous geometric average Asian call or put option is:

$$\Phi(S) = \max\left(e^{\frac{1}{T}\int_0^T \log S(t)dt} - K, 0\right) \quad \text{or} \quad \Phi(S) = \max\left(K - e^{\frac{1}{T}\int_0^T \log S(t)dt}, 0\right).$$

For discrete monitoring, the arithmetic average Asian call or put option has the following payoff:

$$\Phi(S) = \max\left(\frac{1}{m+1}\sum_{i=0}^{m} S\left(\frac{iT}{m}\right) - K, 0\right) \quad \text{or} \quad \Phi(S) = \max\left(K - \frac{1}{m+1}\sum_{i=0}^{m} S\left(\frac{iT}{m}\right), 0\right).$$

Similarly, for discrete monitoring, the geometric average Asian call or put option has the payoff:

$$\Phi(S) = \max\left(e^{\frac{1}{m+1}\sum_{i=0}^{m} \log S\left(\frac{iT}{m}\right)} - K, 0\right) \quad \text{or} \quad \Phi(S) = \max\left(K - e^{\frac{1}{m+1}\sum_{i=0}^{m} \log S\left(\frac{iT}{m}\right)}, 0\right).$$

Asian options are widely used in financial markets for energy and commodity trading. Their structure reduces the risk of price manipulation near expiration and smooths out price volatility through averaging.

In the following section, we will focus on three parts. The first is pricing an arithmetic average Asian call option using the Monte Carlo simulation method. Then, we will investigate how to enhance the efficiency of the Monte Carlo method by implementing variance reduction techniques: the Control Variate Method. Additionally, we will further explore variance reduction methods, such as combining control variates with stratified sampling and the brownian bridge.

3.2 Asian Option Pricing– Monte Carlo Method

Monte Carlo (MC) simulation is a widely used numerical method for pricing financial derivatives. The MC method involves generating a large number of simulated paths for the underlying asset price and computing the average payoff over these paths to estimate the option price. Its flexibility makes it suitable for pricing Asian options, as it can handle the arithmetic averaging of the underlying prices, which does not have a closed-form solution.

The key steps in pricing an Asian call option using the Monte Carlo method are as follows:

Simulate Asset Price Paths: The underlying asset price S_t is modeled as a geometric Brownian motion (GBM) under the risk-neutral measure: $dS_t = (r-q)S_t\,dt + \sigma S_t\,dW_t$, where: - r: Risk-free interest rate, - q: Dividend yield, - σ: Volatility of the asset, - W_t: Standard Brownian motion.

We apply a logarithmic transformation on S_t, and use Ito's Lemma to solve the stochastic differential equation. Integrating the log-transformed SDE, we get the stock price at time t under risk-neutral probability: $\log(S_t) = \log(S_0) + (r - q - 0.5\sigma^2)t + \sigma W_t$. To Simulate stock prices, we utilize a vectorized approach in the logarithmic domain to ensure both computational efficiency and numerical stability. We begin by generating

a matrix of standard normal random variables, $Z \sim \mathcal{N}(0,1)$, representing N paths over m discrete time steps. We calculate the drift term and diffusion term separately. The drift term is added to account for deterministic growth. A cumulative sum is applied to the diffusion term across time steps to approximate W_t. This approach leverages matrix operations to compute all paths simultaneously, eliminating the need for iterative updates and enhancing performance for large-scale simulations. Furthermore, working in the logarithmic domain mitigates numerical instabilities that may arise from compounding errors in the stock price domain, particularly for long time horizons or high volatility.

Calculate Payoff: For each simulated path, compute the arithmetic average of the asset prices: $\bar{S}_{\text{arith}} = \frac{1}{m}\sum_{i=1}^{m} S_{t_i}$. Then, calculate the payoff of the option for each path as: $\text{Payoff}_i = \max(\bar{S}_{\text{arith}} - K, 0)$, where K is the strike price.

Discount Payoffs and Estimate Option Price: Discount the payoff to present value using the risk-free rate: $\text{Discounted Payoff}_i = e^{-rT} \cdot \text{Payoff}_i$. The Monte Carlo estimate of the Asian option price is the average of the discounted payoffs: $\text{Option Price} = \frac{1}{N}\sum_{i=1}^{N} \text{Discounted Payoff}_i$, where N is the number of simulated paths.

Measure Uncertainty: The standard error of the Monte Carlo estimate is given by: $\text{Standard Error} = \frac{\text{Standard Deviation of Payoffs}}{\sqrt{N}}$.

Below is the R implementation of calculating Asian call option prices using monte carlo method:

```r
price_asian_call_MC <- function(S0, K, T, r, q, sigma, m, N) {
  start_time <- proc.time()
  delta_t <- T / m
  sqrt_delta_t <- sqrt(delta_t)

  # Generate N x m standard normal random variables
  Z <- matrix(rnorm(N * m), nrow = N, ncol = m)

  # Simulate log prices and then transform to prices
  drift <- (r - q - 0.5 * sigma^2) * delta_t
  diffusion <- sigma * sqrt_delta_t * Z
  W <- t(apply(diffusion, 1, cumsum))
  log_S <- log(S0) + outer(rep(1, N), drift * (1:m)) + W
  S <- exp(log_S)

  # Calculate arithmetic average and payoffs
  S_bar <- rowMeans(S)
  payoffs <- pmax(S_bar - K, 0)
  discounted_payoffs <- exp(-r * T) * payoffs

  # Estimate option price
  option_price <- mean(discounted_payoffs)
  std_error <- sd(discounted_payoffs) / sqrt(N)

  # Calculate 95% confidence interval
  CI_lower <- option_price - 1.96 * std_error
  CI_upper <- option_price + 1.96 * std_error

  end_time <- proc.time()
  comp_time <- (end_time - start_time)[["elapsed"]]

  return(list(
    N = N,
    Option_Price = round(option_price, 2),
```

```
    Standard_Error = round(std_error, 5),
    Confidence_Interval = c(round(CI_lower, 2), round(CI_upper, 2)),
    Computation_Time_sec = round(comp_time, 4)
  ))
}
```

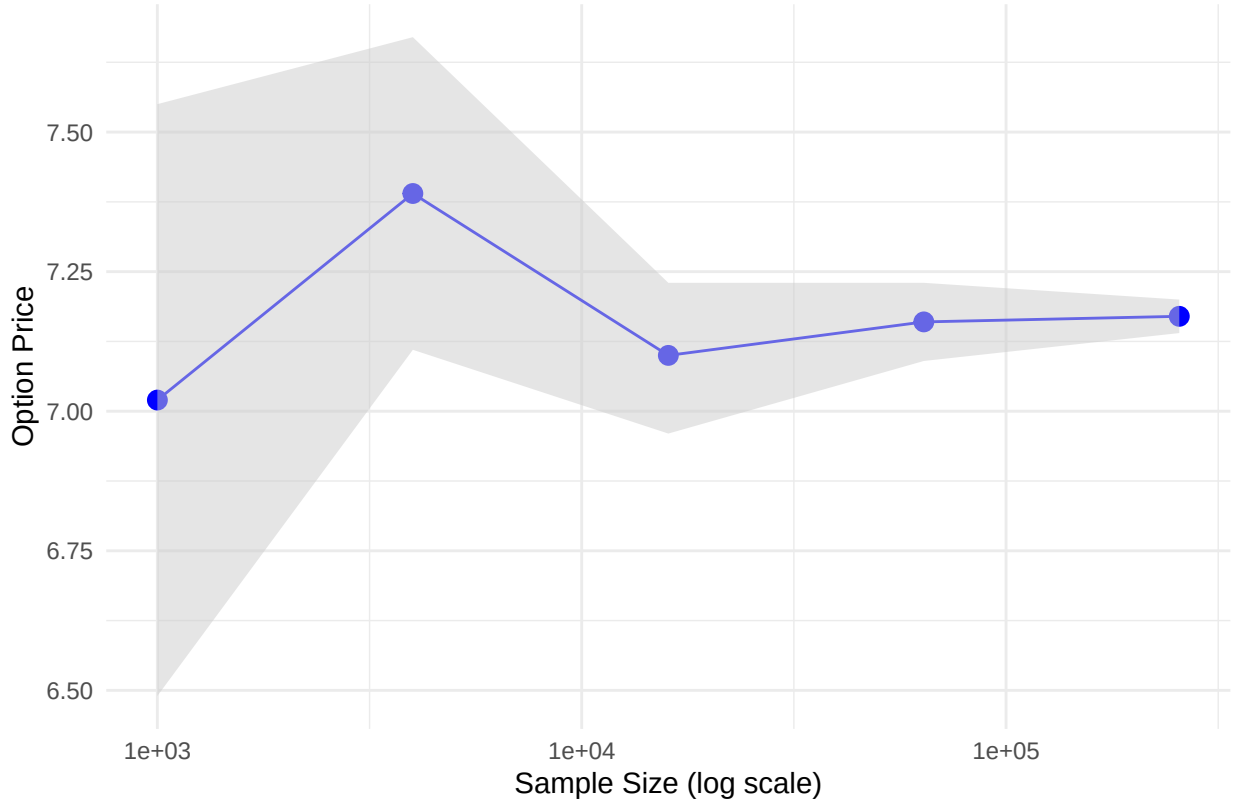
```
# Set parameters
S0 <- 100      # Initial asset price
K <- 100       # Strike price
T <- 1         # Time to maturity (in years)
r <- 0.10      # Risk-free rate
q <- 0         # Dividend yield
sigma <- 0.20  # Volatility
m <- 50        # Number of monitoring points

sample_sizes <- c(1000, 4000, 16000, 64000, 256000)

# Initialize a data frame to store results
standard_mc_results <- data.frame(
  Sample_Size = numeric(),
  Option_Price = numeric(),
  Standard_Error = numeric(),
  CI_Lower = numeric(),
  CI_Upper = numeric(),
  Computation_Time_sec = numeric(),
  stringsAsFactors = FALSE
)

# Set the seed once before the loop
set.seed(123)

# Loop through different sample sizes
for (N in sample_sizes) {
  result <- price_asian_call_MC(S0, K, T, r, q, sigma, m, N)
  standard_mc_results <- rbind(standard_mc_results, data.frame(
    Sample_Size = N,
    Option_Price = result$Option_Price,
    Standard_Error = result$Standard_Error,
    CI_Lower = result$Confidence_Interval[1],
    CI_Upper = result$Confidence_Interval[2],
    Computation_Time_sec = result$Computation_Time_sec
  ))
}

suppressWarnings(suppressPackageStartupMessages({
  library(ggplot2)
  library(knitr)
  library(dplyr)
}))
```

Table 3.1: Monte Carlo Simulation Results

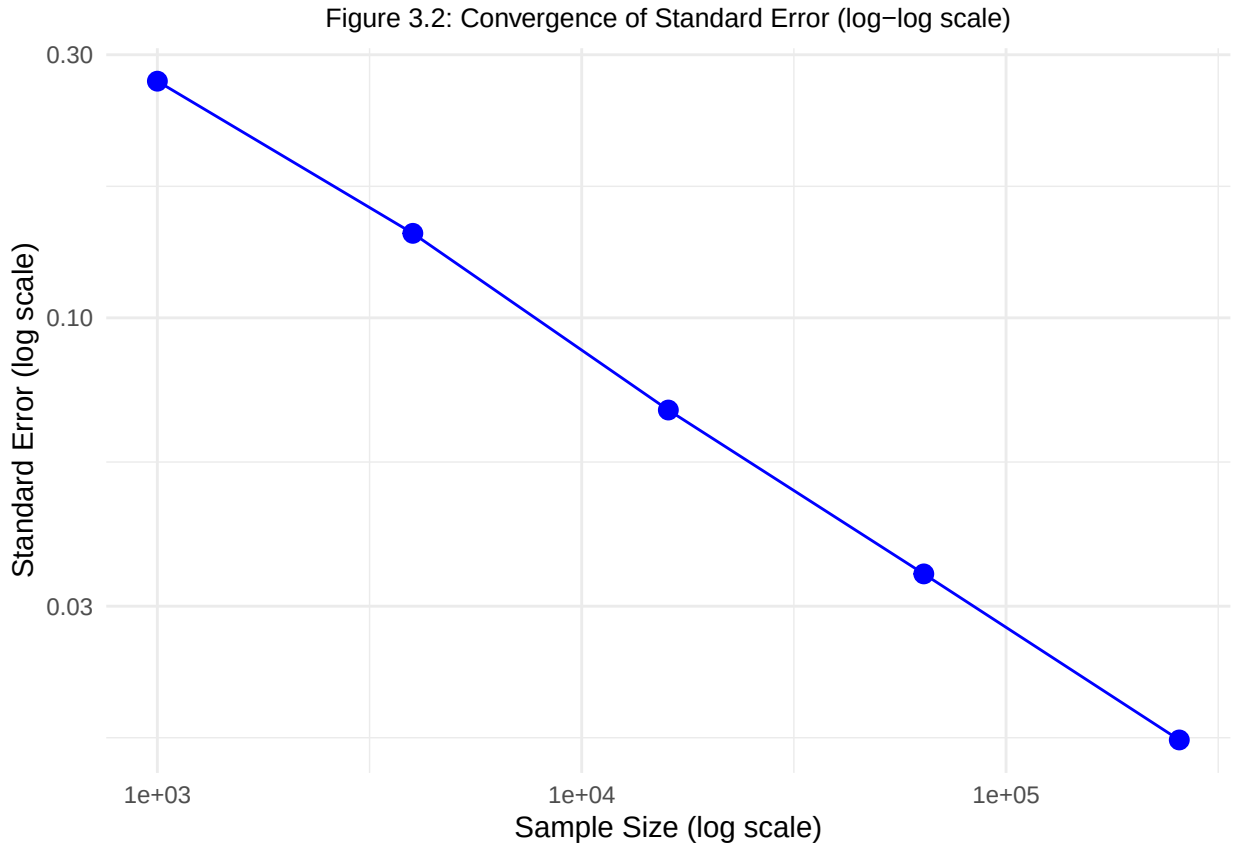
Sample_Size	Option_Price	Standard_Error	CI_Lower	CI_Upper	Computation_Time_sec
1000	7.02	0.26869	6.49	7.55	0.003
4000	7.39	0.14233	7.11	7.67	0.012
16000	7.10	0.06805	6.96	7.23	0.062
64000	7.16	0.03433	7.09	7.23	0.438
256000	7.17	0.01716	7.14	7.20	1.381

```r
# Plot the convergence of Option Price
ggplot(standard_mc_results, aes(x = Sample_Size, y = Option_Price)) +
  geom_line(color = "blue") +
  geom_point(size = 3, color = "blue") +
  geom_ribbon(aes(ymin = CI_Lower, ymax = CI_Upper), fill = "grey80", alpha = 0.5) +
  scale_x_log10() +
  labs(
    title = "Figure 3.1: Convergence of Option Price as Sample Size Increases",
    x = "Sample Size (log scale)",
    y = "Option Price"
  ) +
  theme_minimal()+
  theme(
    plot.title = element_text(
      size = 10,
      hjust = 0.5,
      vjust = 1.2
    )
  )
```

Figure 3.1: Convergence of Option Price as Sample Size Increases




```
# Plot the convergence of Standard Error on log-log scale
ggplot(standard_mc_results, aes(x = Sample_Size, y = Standard_Error)) +
  geom_line(color = "blue") +
  geom_point(size = 3, color = "blue") +
  scale_x_log10() +
  scale_y_log10() +
  labs(
    title = "Figure 3.2: Convergence of Standard Error (log-log scale)",
    x = "Sample Size (log scale)",
    y = "Standard Error (log scale)"
  ) +
  theme_minimal()+
  theme(
    plot.title = element_text(
      size = 10,
      hjust = 0.5,
      vjust = 1.2
    )
  )
```



Convergence of Option Price

The first plot highlights the behavior of the estimated option price as the sample size increases. For smaller sample sizes (e.g., $N = 1000$), the option price exhibits larger variability and a wider confidence interval, indicating less precision. As the sample size increases to $N = 256,000$, the option price converges to a stable value of approximately 7.17, with the 95% confidence interval narrowing significantly to a width of 0.06. This reflects the expected improvement in accuracy with larger sample sizes, consistent with Monte Carlo theory. The convergence is gradual, with the estimates stabilizing beyond $N = 64,000$.

Convergence of Standard Error

The second plot, presented on a log-log scale, demonstrates the relationship between the standard error and the sample size. The convergence follows the theoretical rate of:

$$\text{Standard Error} \propto \frac{1}{\sqrt{N}}$$

This means that as the sample size increases by a factor of 4, the standard error decreases approximately by half. The straight line in the log-log plot shows that doubling the sample size results in a consistent proportional reduction in the standard error, making it predictable and reliable. For example: at $N = 1000$, the standard error is 0.26869. At $N = 4000$ (4 times $N = 1000$), the standard error decreases to approximately 0.14233. Similarly, at $N = 16,000$ (another 4x increase), the standard error further decreases to 0.06805.

3.3 Variance Reduction Techniques– Control Variate Method

In this section, we will introduce a variance reduction technique: control variate method. Suppose we want to estimate the expected value $\theta = \mathbb{E}[Y]$, where $Y = g(X)$ is a function of some random variable X. If we can find another random variable Z, for which the expected value $\mathbb{E}[Z]$ is known, we can construct alternative estimators for θ. For example,

The standard Monte Carlo estimator:

$$\hat{\theta} = Y$$

The control variate estimator:

$$\hat{\theta}_c = Y + c \cdot (Z - \mathbb{E}[Z]),$$

where c is a constant.

It can be shown that the control variate estimator $\hat{\theta}_c$ is unbiased, as:

$$\mathbb{E}[\hat{\theta}_c] = \mathbb{E}[Y] + c \cdot (\mathbb{E}[Z] - \mathbb{E}[Z]) = \mathbb{E}[Y] = \theta.$$

To minimize the variance of $\hat{\theta}_c$, we start with its variance formula:

$$\mathrm{Var}(\hat{\theta}_c) = \mathrm{Var}(Y) + c^2 \cdot \mathrm{Var}(Z) + 2c \cdot \mathrm{Cov}(Y, Z).$$

Treating this as a function of c, we differentiate with respect to c to obtain:

$$f'(c) = 2c \cdot \mathrm{Var}(Z) + 2 \cdot \mathrm{Cov}(Y, Z).$$

Setting $f'(c) = 0$, we solve for c to find the critical point:

$$c_{\text{opt}} = -\frac{\mathrm{Cov}(Y, Z)}{\mathrm{Var}(Z)}.$$

To confirm that this value minimizes the variance, we compute the second derivative:

$$f''(c) = 2 \cdot \mathrm{Var}(Z).$$

Since $\mathrm{Var}(Z) > 0$, $f(c)$ is convex, and c_{opt} is the minimizer.

This demonstrates that the control variate method reduces variance by leveraging the correlation between Y and Z, particularly when they are highly correlated.

In this practice, we will use geometric Asian Call as a control variant. This method is efficient, since first, the correlation between the arithmetic average and the geometric average is very high; second, the counterpart geometric average Asian option price has a closed-form solution. We use the following formula to calculate geometric Asian call option price, serving as the expected value of the control variant in the control variate estimator:

$$\begin{aligned}
\sigma_z^2 &= \sigma^2 \cdot \frac{(m+1)(2m+1)}{6m^2} \\
\mu &= \left(r - q - \frac{1}{2}\sigma^2\right) \cdot \frac{m+1}{2m} + \frac{1}{2}\sigma_z^2 \\
d_1 &= \frac{\ln\left(\frac{S_0}{K}\right) + \left(\mu + \frac{1}{2}\sigma_z^2\right) T}{\sigma_z \sqrt{T}} \\
d_2 &= \frac{\ln\left(\frac{S_0}{K}\right) + \left(\mu - \frac{1}{2}\sigma_z^2\right) T}{\sigma_z \sqrt{T}} \\
\text{Geometric Asian Call Price} &= e^{-rT}\left[S_0 \cdot e^{\mu T} \cdot N(d_1) - K \cdot N(d_2)\right]
\end{aligned}$$

Here: σ_z^2: Effective volatility of the geometric average. μ: Drift term adjusted for the geometric average. d_1, d_2: Terms used in the Black-Scholes framework for the option price. $N(d)$: Cumulative distribution function of the standard normal distribution.

Below is the R implementation of calculating the analytical price of Geometric Asian Call Option:

```r
# function to calculate the analytical price of Geometric Asian Call Option
price_geometric_asian_call <- function(S0, K, T, r, q, sigma, m){
  delta_t <- T/m
  sigma_sq <- sigma^2
  sigma_z_sq <- (sigma_sq) * (m + 1) * (2 * m + 1) / (6 * m^2)
  drift <-(r - q - 0.5*sigma_sq) * (m + 1) / (2 * m) + 0.5 * sigma_z_sq
  sigma_z <- sqrt(sigma_z_sq)
  d1 <- (log(S0/K) + (drift + 0.5 * sigma_z_sq) * T) / (sigma_z * sqrt(T))
  d2 <- (log(S0/K) + (drift - 0.5 * sigma_z_sq) * T) / (sigma_z * sqrt(T))
  geo_asian_price <- exp(-r*T) * (S0* exp(drift * T) * pnorm(d1) - K * pnorm(d2))
  return(geo_asian_price)
}
```

To implement this method, we first simulate the stock price Y using standard Monte Carlo techniques. Next, we compute the geometric average of the stock prices and take its logarithm to derive the geometric option price, denoted as Z. The closed-form solution for the geometric option price allows us to calculate its expected value, $\mathbb{E}[Z]$. Using this information, we construct the control variate estimator:

$\hat{\theta}_c = Y + c \cdot (Z - \mathbb{E}[Z])$, where $c_{\text{opt}} = -\frac{\text{Cov}(Y,Z)}{\text{Var}(Z)}$ is the optimal control coefficient. Below is the R implementation of the Monte Carlo control variate method:

```r
# function to price asian call option using geometric asian call price as a control variate
price_asian_call_MC_control_variate <- function(S0, K, T, r, q, sigma, m, N) {
  start_time <- proc.time()

  delta_t <- T / m
  sqrt_delta_t <- sqrt(delta_t)

  Z <- matrix(rnorm(N * m), nrow = N, ncol = m)

  drift <- (r - q - 0.5 * sigma^2) * delta_t
  diffusion <- sigma * sqrt_delta_t * Z
  W <- t(apply(diffusion, 1, cumsum))
  log_S <- log(S0) + outer(rep(1, N), (r - q - 0.5 * sigma^2) * (delta_t * (1:m))) + W
  S <- exp(log_S)

  S_bar_arith <- rowMeans(S)
  S_bar_geo <- exp(rowMeans(log(S)))

  payoffs_arith <- pmax(S_bar_arith - K, 0)

  # Calculate payoffs for geometric Asian call
  payoffs_geo <- pmax(S_bar_geo - K, 0)

  # Discount payoffs to present value
  discounted_payoffs_arith <- exp(-r * T) * payoffs_arith
  discounted_payoffs_geo <- exp(-r * T) * payoffs_geo

  # Calculate analytical price of Geometric Asian Call
  geo_asian_price <- price_geometric_asian_call(S0, K, T, r, q, sigma, m)

  # Calculate covariance and variance for Control Variate
  cov_xy <- cov(discounted_payoffs_arith, discounted_payoffs_geo)
  var_y <- var(discounted_payoffs_geo)
```

```
  # Optimal coefficient
  theta <- cov_xy / var_y

  # Calculate Control Variate estimator
  control_variate_estimator <- discounted_payoffs_arith - theta * (discounted_payoffs_geo - geo_asian_pr

  # Calculate option price using Control Variate
  option_price_cv <- mean(control_variate_estimator)

  std_error_cv <- sd(control_variate_estimator) / sqrt(N)

  CI_lower_cv <- option_price_cv - 1.96 * std_error_cv
  CI_upper_cv <- option_price_cv + 1.96 * std_error_cv

  end_time <- proc.time()
  comp_time <- (end_time - start_time)[["elapsed"]]

  return(list(
    N = N,
    Option_Price = round(option_price_cv, 2),
    Standard_Error = round(std_error_cv, 5),
    Confidence_Interval = c(round(CI_lower_cv, 2), round(CI_upper_cv, 2)),
    Computation_Time_sec = round(comp_time, 4)
  ))
}
```

```
# Set parameters
S0 <- 100       # Initial asset price
K <- 100        # Strike price
T <- 1          # Time to maturity (in years)
r <- 0.10       # Risk-free rate
q <- 0          # Dividend yield
sigma <- 0.20   # Volatility
m <- 50         # Number of monitoring points

sample_sizes <- c(1000, 4000, 16000, 64000, 256000)
# Initialize the results data frame
results_cv <- data.frame(
  Sample_Size = numeric(),
  Option_Price = numeric(),
  Standard_Error = numeric(),
  CI_Lower = numeric(),
  CI_Upper = numeric(),
  Computation_Time_sec = numeric(),
  stringsAsFactors = FALSE
)
# Remove or comment out the next line to prevent printing the empty data frame
# results_cv

for (N in sample_sizes) {
  # Monte Carlo with Control Variate
  res_cv <- price_asian_call_MC_control_variate(S0, K, T, r, q, sigma, m, N)
  results_cv <- rbind(results_cv, data.frame(
```

```
    Sample_Size = N,
    Option_Price = res_cv$Option_Price,
    Standard_Error = res_cv$Standard_Error,
    CI_Lower = res_cv$Confidence_Interval[1],
    CI_Upper = res_cv$Confidence_Interval[2],
    Computation_Time_sec = res_cv$Computation_Time_sec
  ))
}
```

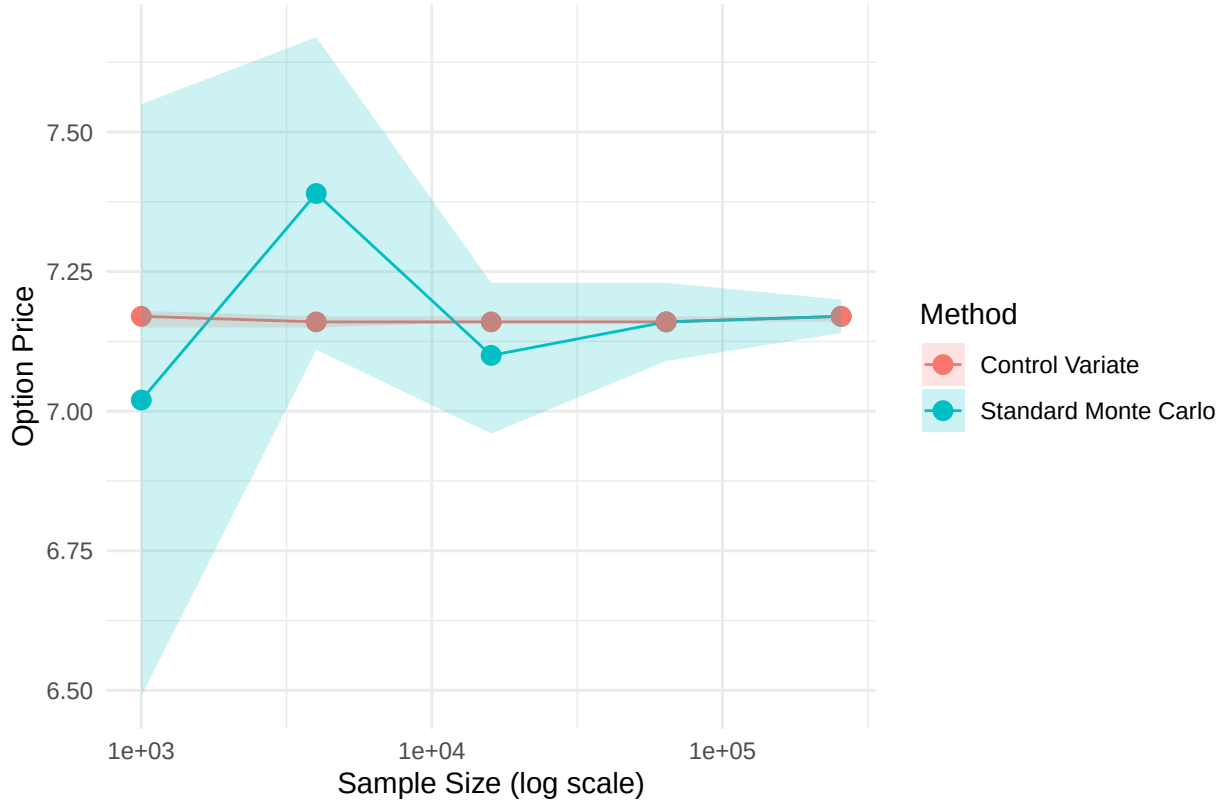
Table 3.2: Monte Carlo with Control Variate Simulation Results

Sample_Size	Option_Price	Standard_Error	CI_Lower	CI_Upper	Computation_Time_sec
1000	7.17	0.00888	7.15	7.18	0.013
4000	7.16	0.00401	7.15	7.17	0.012
16000	7.16	0.00205	7.16	7.17	0.057
64000	7.16	0.00102	7.16	7.17	0.226
256000	7.17	0.00051	7.16	7.17	1.226

```
# Combine data if not already done
standard_mc_results$Method <- "Standard Monte Carlo"
results_cv$Method <- "Control Variate"
comparison_results <- rbind(standard_mc_results, results_cv)
```

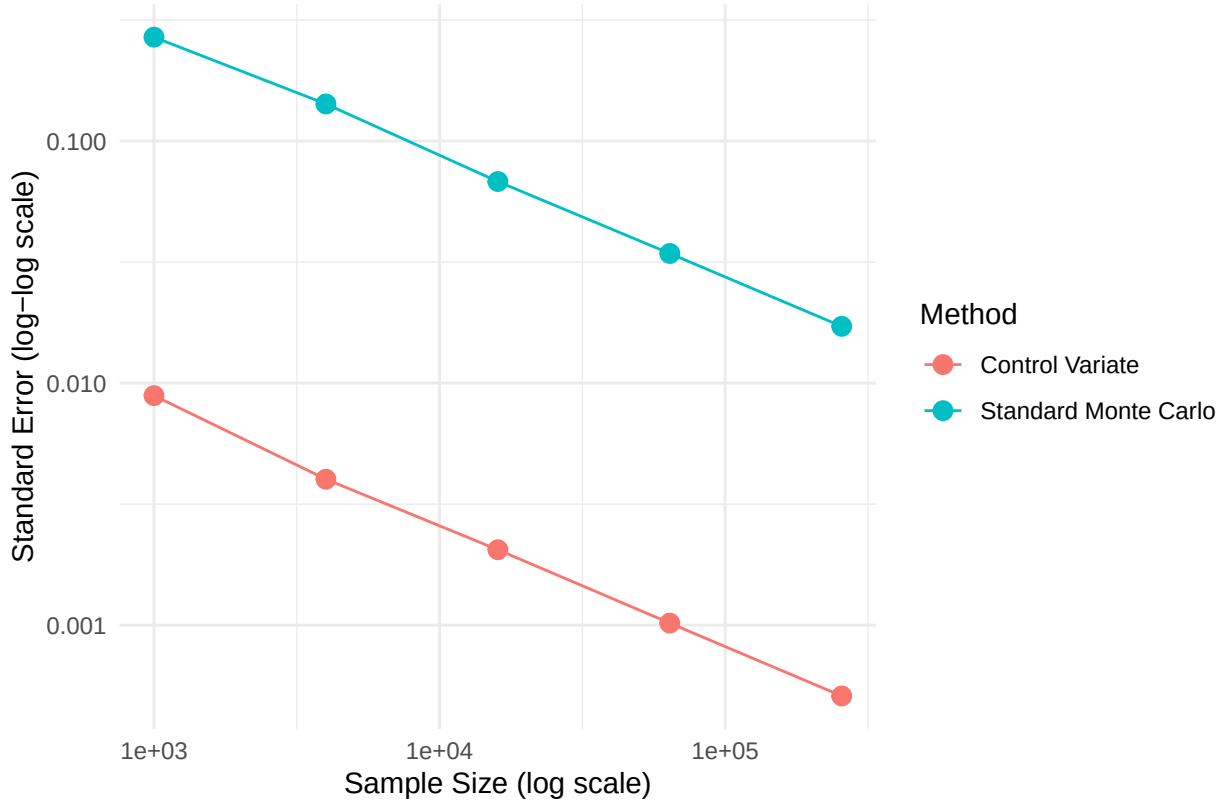
```
# Plot Option Price Convergence with 95% CI
ggplot(comparison_results, aes(x = Sample_Size, y = Option_Price, color = Method, group = Method)) +
  geom_line() +
  geom_point(size = 3) +
  geom_ribbon(
    aes(ymin = CI_Lower, ymax = CI_Upper, fill = Method),
    alpha = 0.2,
    color = NA
  ) +
  scale_x_log10() +
  labs(
    title = "Figure 3.3: Option Price Convergence with 95% CI: Standard MC vs Control Variate",
    x = "Sample Size (log scale)",
    y = "Option Price",
    color = "Method",
    fill = "Method"
  ) +
  theme_minimal() +
  theme(
    plot.title = element_text(
      size = 10,
      hjust = 0.5,
      vjust = 1.2
    )
  )
```

Figure 3.3: Option Price Convergence with 95% CI: Standard MC vs Control Variate



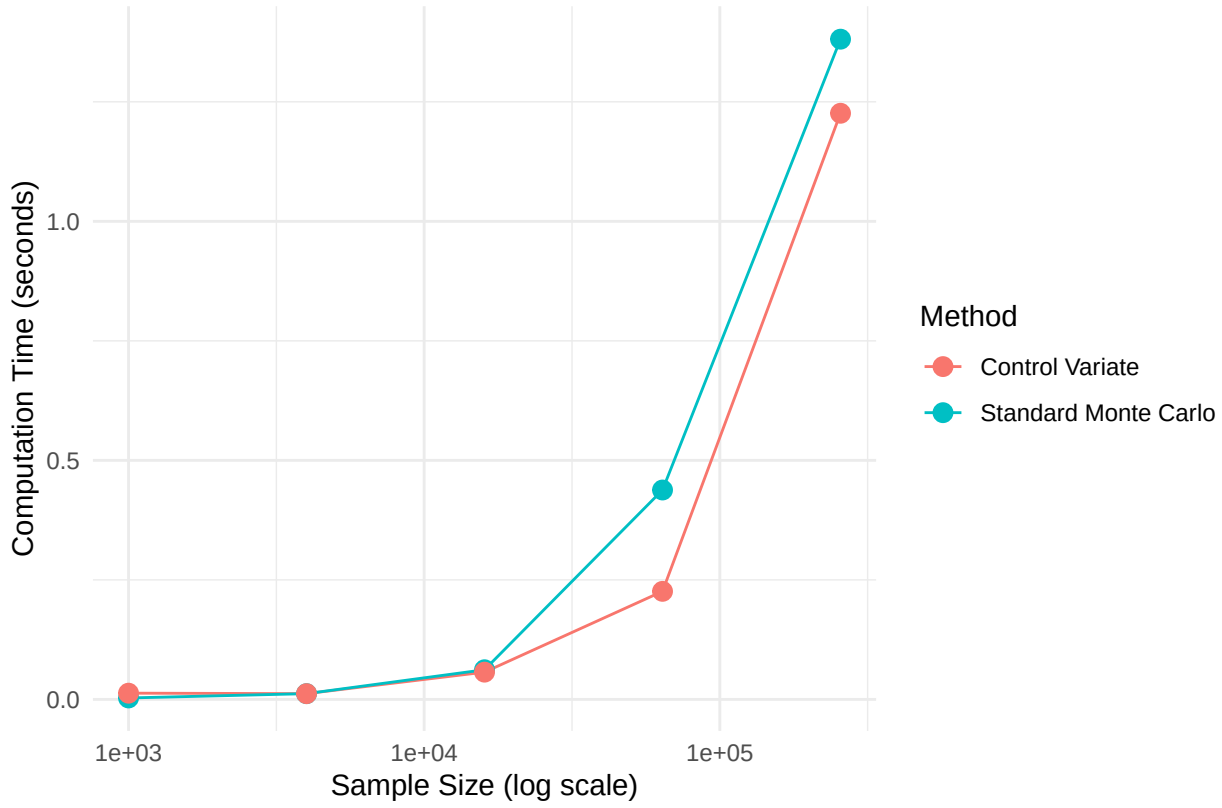

```
# Plot Standard Error Convergence
ggplot(comparison_results, aes(x = Sample_Size, y = Standard_Error, color = Method, group = Method)) +
  geom_line() +
  geom_point(size = 3) +
  scale_x_log10() +
  scale_y_log10() +
  labs(
    title = "Figure 3.4: Standard Error Convergence: Standard MC vs Control Variate",
    x = "Sample Size (log scale)",
    y = "Standard Error (log-log scale)",
    color = "Method"
  ) +
  theme_minimal() +
  theme(
    plot.title = element_text(
      size = 10,
      hjust = 0.5,
      vjust = 1.2
    )
  )
```

Figure 3.4: Standard Error Convergence: Standard MC vs Control Variate




```
# Plot Computation Time Comparison
ggplot(comparison_results, aes(x = Sample_Size, y = Computation_Time_sec, color = Method, group = Metho
  geom_line() +
  geom_point(size = 3) +
  scale_x_log10() +
  labs(
    title = "Figure 3.5: Computation Time: Standard MC vs Control Variate",
    x = "Sample Size (log scale)",
    y = "Computation Time (seconds)",
    color = "Method"
  ) +
  theme_minimal() +
  theme(
    plot.title = element_text(
      size = 10,
      hjust = 0.5,
      vjust = 1.2
    )
  )
```

Figure 3.5: Computation Time: Standard MC vs Control Variate



3.4 Analysis of Results: Comparing Standard Monte Carlo (MC) and Control Variate (CV)

The comparison of the Standard Monte Carlo method and the Control Variate method reveals clear advantages of the latter in terms of convergence rate, variance reduction, and overall efficiency.

1. Option Price Convergence

The Option Price Convergence plot highlights the convergence of option price estimates as the sample size increases. The control variate method demonstrates significantly tighter confidence intervals compared to the standard Monte Carlo method.

For the CV method, the option price estimate stabilizes more rapidly and converges to a consistent value, even at smaller sample sizes. In contrast, the standard MC method requires larger sample sizes to achieve comparable stability. Both methods ultimately converge to approximately the same option price as the sample size increases ($N = 256,000$), confirming the consistency of their final estimates.

The tighter confidence intervals produced by the CV method highlight its ability to improve precision, particularly when computational resources are limited or fewer samples are used.

2. Standard Error Convergence

The Standard Error Convergence plot examines the convergence of standard error on a log-log scale. Both methods exhibit the expected Monte Carlo convergence rate of $O(1/\sqrt{N})$, as evidenced by the linear trends in the plot. However, the CV method consistently achieves lower standard errors compared to the standard MC method at all sample sizes.

For instance: at $N = 1000$, the CV method achieves a standard error more than 10 times smaller than the standard MC method. Even as N increases to $256,000$, the CV method maintains a significant advantage in standard error reduction.

This reduction in standard error reflects the CV method's effectiveness in reducing variance, which is critical in applications requiring high precision. The CV technique allows for comparable accuracy with far fewer samples, making it computationally more efficient than standard MC.

3. Computation Time

The third plot compares the computation times for the two methods. As expected, the computation time increases linearly with sample size for both approaches. The CV method incurs slightly higher computational costs compared to standard MC, reflecting the additional calculations required for the control variate adjustment. For example, at $N = 256,000$, the computation time for CV is marginally higher than that of standard MC.

Despite this minor overhead, the significant reduction in variance achieved by the CV method more than compensates for the slightly higher computation cost. This trade-off makes the CV technique highly effective in scenarios where variance reduction is a priority.

3.5 Further Discussion on Variance Reduction Techniques

In addition to the control variate method, variance reduction can be further enhanced by combining it with other techniques such as stratified sampling and the Brownian bridge construction. These methods introduce additional refinements to the Monte Carlo simulation process, addressing specific sources of randomness and variability in the simulation.

Stratified sampling divides the sample space into non-overlapping strata and ensures that an equal number of samples are drawn from each stratum. This method reduces variance by minimizing randomness in the sampling process.

Mathematically, the variance of the stratified sampling estimator can be expressed as:

$$\mathrm{Var}(\hat{\theta}_{\text{stratified}}) = \sum_{k=1}^{L} \frac{w_k^2}{n_k} \mathrm{Var}(\theta_k),$$

where: - L is the number of strata, - w_k is the weight of the k-th stratum, - n_k is the number of samples in the k-th stratum, - $\mathrm{Var}(\theta_k)$ is the variance within the k-th stratum.

By ensuring that n_k is proportional to w_k, stratified sampling can achieve a lower variance compared to direct sampling. A common application of stratified sampling in Monte Carlo simulations is in the context of Brownian motion, where stratification is often applied to the first step of the simulation, Z_1, which introduces the most variability in the path. This step simplifies the implementation and ensures that the variability in the simulation is minimized from the outset. However, stratified sampling has its limitations. It is computationally more expensive than standard Monte Carlo sampling because it requires the sample space to be divided and the simulation to be carefully managed within each stratum.

The Brownian bridge construction is an effective technique for variance reduction that refines the simulation of Brownian paths by interpolating intermediate points based on the known values at the start and end of the process. By introducing conditional dependencies among simulated points, this method ensures consistency with the overall process, making it particularly useful when the terminal value of the asset significantly influences the option payoff. Moreover, the Brownian bridge is computationally efficient for high-dimensional problems, as it reduces the number of independently generated random variables and complements the control variate method by improving the accuracy of simulated paths.

Despite its advantages, the implementation of the Brownian bridge introduces added complexity. Interpolation of intermediate points and adjustments to the simulation framework require careful execution.

Additionally, the variance reduction achieved depends on the characteristics of the option and the underlying asset paths, and in some cases, the benefits may be marginal relative to the additional implementation effort.

3.6 Variance Reduction Techniques– Moment Matching

The Moment Matching (MM) method is a variance reduction technique used to improve the convergence and accuracy of Monte Carlo simulations. By adjusting the random samples to match the theoretical properties of the underlying distribution, MM ensures that the simulated paths conform more closely to the model assumptions.

Mathematical Formulation Let $Z_{ij} \sim \mathcal{N}(0,1)$ be the random samples used to simulate the asset price paths, where Z_{ij} represents the j-th time step in the i-th simulated path. For Z, we define:

1. Empirical mean:
$$\hat{\mu} = \frac{1}{N \cdot m} \sum_{i=1}^{N} \sum_{j=1}^{m} Z_{ij},$$
where N is the number of simulated paths, and m is the number of time steps.

2. Empirical variance:
$$\hat{\sigma}^2 = \frac{1}{N \cdot m - 1} \sum_{i=1}^{N} \sum_{j=1}^{m} (Z_{ij} - \hat{\mu})^2.$$

The goal of Moment Matching is to adjust Z such that:

$$\text{Mean of Adjusted } Z = 0, \quad \text{Variance of Adjusted } Z = 1.$$

Steps for Adjustment

1. **Centralization (Zero Mean)**: Subtract the empirical mean from each sample to centralize the data:
$$Z'_{ij} = Z_{ij} - \hat{\mu}.$$

2. **Normalization (Unit Variance)**: Divide each sample by the empirical standard deviation:
$$Z''_{ij} = \frac{Z'_{ij}}{\hat{\sigma}}.$$

After adjustment, the modified samples Z''_{ij} satisfy:

$$\text{Mean of } Z'' = 0, \quad \text{Variance of } Z'' = 1.$$

Why Moment Matching Works

1. **Variance Reduction**: By enforcing theoretical mean and variance, MM reduces the noise introduced by random sampling errors, leading to faster convergence.
2. **Improved Accuracy**: MM ensures that the simulated random variables match the assumed distribution properties more closely, improving the precision of Monte Carlo estimates.

Below is the implementation of the Moment Matching algorithm:

```r
# Moment Matching Function
moment_matching <- function(Z) {
  # Step 1: Centralize the samples (adjust mean to 0)
  Z_centered <- Z - rowMeans(Z)

  # Step 2: Normalize the samples (adjust variance to 1)
  Z_normalized <- Z_centered / apply(Z_centered, 1, sd)

  # Return the adjusted random samples
  return(Z_normalized)
}
```

The `moment_matching` function ensures that the random samples used in the simulation have properties consistent with a standard normal distribution. Specifically, it adjusts the samples so that each simulated path has a mean of 0 and a variance of 1.

The function works in two steps:

1. **Centralization**: Each row of the random matrix is adjusted by subtracting the row mean. This ensures that the average value of the samples is zero.
2. **Normalization**: After centralization, each row is scaled by its standard deviation. This ensures that the variance of the samples is equal to one.

These adjustments reduce noise introduced by sampling and help the Monte Carlo simulation converge more efficiently. By aligning the samples with the theoretical distribution, the results become more stable and accurate, especially for smaller sample sizes.

```r
# Asian Call Option Pricing with Moment Matching
price_asian_call_MM <- function(S0, K, T, r, q, sigma, m, N) {
  start_time <- proc.time()
  delta_t <- T / m
  sqrt_delta_t <- sqrt(delta_t)

  # Generate random numbers and apply Moment Matching
  Z <- matrix(rnorm(N * m), nrow = N, ncol = m)
  Z <- moment_matching(Z)

  # Simulate asset price paths
  drift <- (r - q - 0.5 * sigma^2) * delta_t
  diffusion <- sigma * sqrt_delta_t * Z
  W <- t(apply(diffusion, 1, cumsum))
  log_S <- log(S0) + outer(rep(1, N), drift * (1:m)) + W
  S <- exp(log_S)

  # Calculate option price
  S_bar <- rowMeans(S)
  payoffs <- pmax(S_bar - K, 0)
  discounted_payoffs <- exp(-r * T) * payoffs
  option_price <- mean(discounted_payoffs)
  std_error <- sd(discounted_payoffs) / sqrt(N)
  CI_lower <- option_price - 1.96 * std_error
  CI_upper <- option_price + 1.96 * std_error
```

```r
  comp_time <- (proc.time() - start_time)[["elapsed"]]

  return(list(
    N = N,
    Option_Price = round(option_price, 2),
    Standard_Error = round(std_error, 5),
    Confidence_Interval = c(round(CI_lower, 2), round(CI_upper, 2)),
    Computation_Time_sec = round(comp_time, 4)
  ))
}
```

The above `price_asian_call_MM` function implements the Monte Carlo simulation for pricing an Asian call option using the moment matching method to improve accuracy and efficiency. The function consists of three main steps: generating random numbers, simulating asset price paths, and calculating the option price.

Step 1: Generate Random Numbers with Moment Matching The function first generates a matrix of standard normal random numbers Z with dimensions $N \times m$, where N is the number of simulated paths, and m is the number of time steps. The `moment_matching` function is applied to adjust the random numbers so that:

$$\text{Mean of Adjusted } Z = 0, \quad \text{Variance of Adjusted } Z = 1.$$

This adjustment ensures that the simulated random samples align more closely with the theoretical properties of the standard normal distribution, reducing noise and improving convergence.

Step 2: Simulate Asset Price Paths The adjusted random numbers are used to simulate asset price paths under the Geometric Brownian Motion (GBM) model. The GBM model for the stock price S_t is given by:

$$S_t = S_0 \exp\left((r - q - 0.5\sigma^2)t + \sigma W_t\right),$$

where: - S_0: Initial stock price, - r: Risk-free interest rate, - q: Dividend yield, - σ: Volatility, - W_t: Standard Brownian motion.

The function uses the drift and diffusion terms to construct the log-transformed price:

$$\log(S_t) = \log(S_0) + (r - q - 0.5\sigma^2)t + \sigma W_t,$$

and then converts the log prices back to the original price domain:

$$S_t = \exp(\log(S_t)).$$

Step 3: Calculate Option Price For each simulated path, the arithmetic average of the asset prices is computed:

$$\bar{S} = \frac{1}{m}\sum_{i=1}^{m} S_{t_i}.$$

The payoff of the Asian call option is then calculated as:

$$\text{Payoff}_i = \max(\bar{S} - K, 0),$$

where K is the strike price. The present value of the payoff is obtained by discounting it at the risk-free rate:

$$\text{Discounted Payoff}_i = e^{-rT} \cdot \text{Payoff}_i.$$

The option price is estimated as the average of the discounted payoffs:

$$\text{Option Price} = \frac{1}{N}\sum_{i=1}^{N} \text{Discounted Payoff}_i.$$

Statistical Outputs To assess the accuracy of the estimate, the function computes the standard error:

$$\text{Standard Error} = \frac{\text{Standard Deviation of Payoffs}}{\sqrt{N}},$$

and constructs a 95% confidence interval for the option price:

$$\text{Confidence Interval} = [\text{Option Price} \pm 1.96 \times \text{Standard Error}]\,.$$

The function also measures the computation time to evaluate performance. By reducing the variability in the random samples, the moment matching method ensures faster convergence of the Monte Carlo simulation and provides more stable results, especially for small sample sizes.

Testing the Moment Matching Method

To evaluate the performance of the moment matching method, we test the `price_asian_call_MM` function using different sample sizes (N) and compare the results. The table below summarizes the option price, standard error, 95% confidence interval, and computation time for each sample size.

```r
# Parameters for the Asian call option
S0 <- 100      # Initial stock price
K <- 100       # Strike price
T <- 1         # Time to maturity (in years)
r <- 0.10      # Risk-free rate
q <- 0         # Dividend yield
sigma <- 0.20  # Volatility
m <- 50        # Number of monitoring points

# Define sample sizes for testing
sample_sizes <- c(1000, 4000, 16000, 64000, 256000)

# Initialize a data frame to store results
moment_matching_results <- data.frame(
  Sample_Size = numeric(),
  Option_Price = numeric(),
  Standard_Error = numeric(),
  CI_Lower = numeric(),
  CI_Upper = numeric(),
  Computation_Time_sec = numeric(),
  stringsAsFactors = FALSE
)

# Loop through different sample sizes and compute results
for (N in sample_sizes) {
  result <- price_asian_call_MM(S0, K, T, r, q, sigma, m, N)
  moment_matching_results <- rbind(moment_matching_results, data.frame(
    Sample_Size = N,
    Option_Price = result$Option_Price,
    Standard_Error = result$Standard_Error,
    CI_Lower = result$Confidence_Interval[1],
    CI_Upper = result$Confidence_Interval[2],
    Computation_Time_sec = result$Computation_Time_sec
  ))
}
```

Table 3.3: Moment Matching Simulation Results

Sample_Size	Option_Price	Standard_Error	CI_Lower	CI_Upper	Computation_Time_sec
1000	4.82	0.14381	4.54	5.10	0.007
4000	4.67	0.07066	4.53	4.80	0.033
16000	4.84	0.03611	4.77	4.91	0.127
64000	4.77	0.01795	4.74	4.81	0.520
256000	4.78	0.00899	4.76	4.80	2.358

The above code tests the performance of the moment matching method by evaluating the option price under different sample sizes N. The `price_asian_call_MM` function simulates the stock price paths based on the Geometric Brownian Motion model using moment matching to adjust the random samples. Specifically, the simulation begins with generating a matrix of standard normal random variables Z, with dimensions $N \times m$, where N is the number of paths and m is the number of time steps. Moment matching ensures the random samples have a mean of zero and variance of one to align with the standard normal distribution. These adjusted samples are then used to simulate the asset prices S_t at each time step. The arithmetic average of the asset prices for each path is computed, and the option payoff is calculated as $\max(\bar{S} - K, 0)$, where $\bar{S}$ is the average price and K is the strike price. The payoff is then discounted to the present value using e^{-rT}, where r is the risk-free interest rate and T is the time to maturity. The function outputs the average option price, standard error, 95% confidence interval, and computation time.

The results in the table demonstrate the convergence of the Monte Carlo simulation with moment matching as the sample size increases. At $N = 1000$, the option price estimate is 4.82 with a relatively large standard error of 0.14381, and the confidence interval is wide, ranging from 4.54 to 5.10. As the sample size increases to $N = 4000$, the option price drops to 4.67, and the standard error decreases to 0.07066, showing a noticeable improvement in precision. With $N = 16000$, the option price estimate stabilizes at 4.84, and the standard error further reduces to 0.03611, with a narrower confidence interval of [4.77, 4.91]. When the sample size increases to $N = 64000$, the option price is 4.77, and the standard error is significantly reduced to 0.01795, confirming the expected improvement in accuracy as N increases. Finally, for $N = 256000$, the option price converges to 4.78, with a very small standard error of 0.00899 and a confidence interval of [4.76, 4.80].

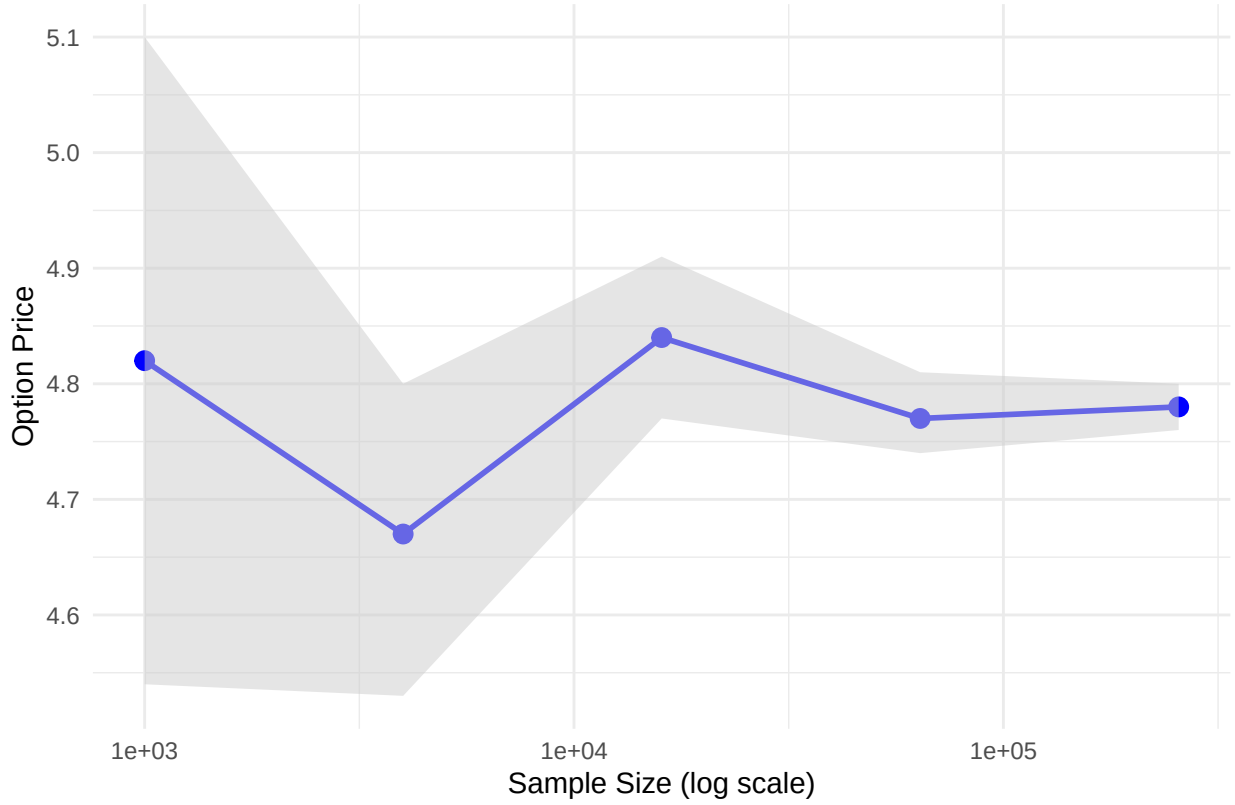
The computation time also increases with the sample size, from 0.008 seconds for $N = 1000$ to 2.416 seconds for $N = 256000$. This behavior is expected since larger sample sizes require more calculations. The results indicate that the moment matching method provides stable and accurate estimates of the Asian call option price. As the sample size increases, the standard error decreases approximately at the rate of $O(1/\sqrt{N})$, which is consistent with the theoretical behavior of the Monte Carlo method. The final estimate stabilizes around 4.78, suggesting good convergence of the simulation. Overall, the moment matching approach effectively improves the reliability of the Monte Carlo method by reducing sampling variability and enhancing the precision of the results.

To visualize the convergence of the option price as the sample size increases, we plot the option price estimates along with their 95% confidence intervals.

```
library(ggplot2)

# Plot Option Price Convergence
ggplot(moment_matching_results, aes(x = Sample_Size, y = Option_Price)) +
  geom_line(color = "blue", size = 1) +
  geom_point(size = 3, color = "blue") +
  geom_ribbon(aes(ymin = CI_Lower, ymax = CI_Upper), fill = "grey80", alpha = 0.5) +
  scale_x_log10() +
  labs(
    title = "Figure 3.6: Convergence of Option Price with Increasing Sample Size",
```

```
    x = "Sample Size (log scale)",
    y = "Option Price"
  ) +
  theme_minimal()+
  theme(
    plot.title = element_text(
      size = 10,
      hjust = 0.5,
      vjust = 1.2
    )
  )
```

Figure 3.6: Convergence of Option Price with Increasing Sample Size



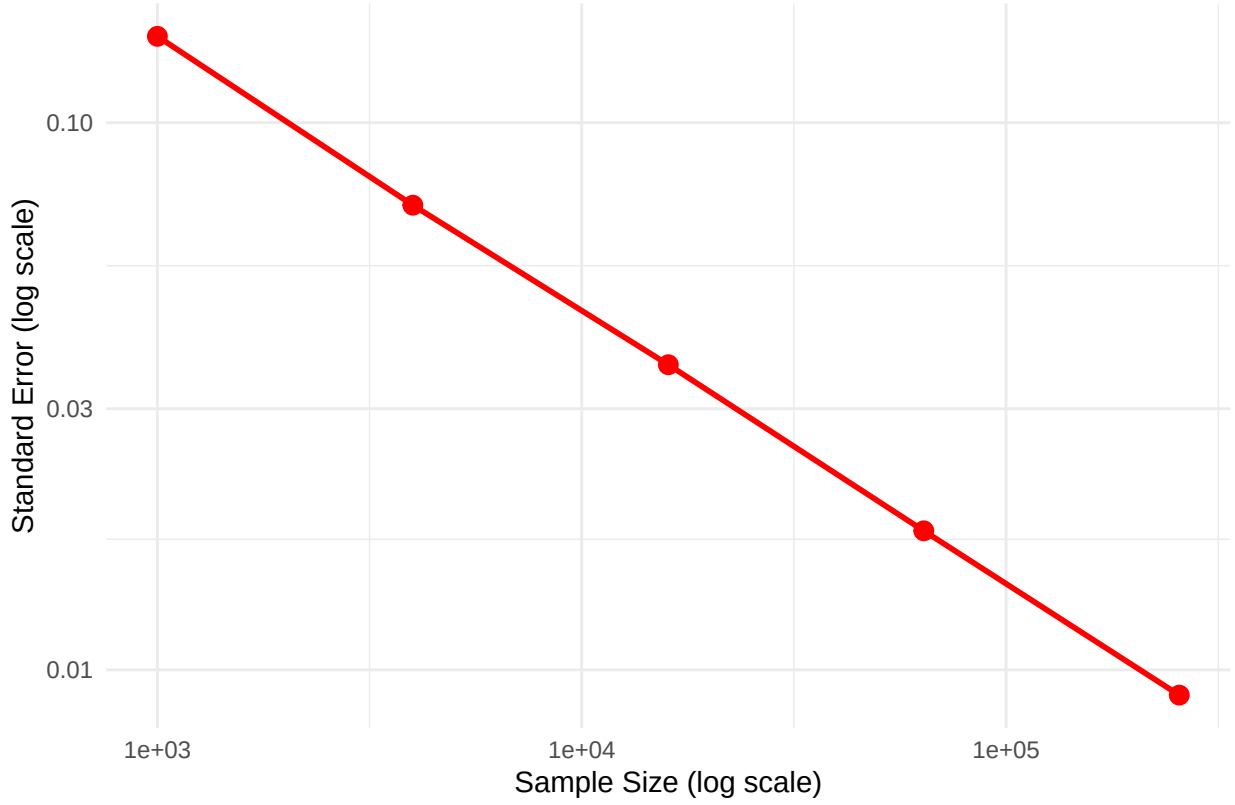
The plot illustrates the convergence of the Asian call option price as the sample size increases. At smaller sample sizes (e.g., $N = 1000$), the option price estimate shows noticeable variability, with a wider confidence interval. This reflects higher uncertainty due to insufficient paths in the Monte Carlo simulation. As the sample size increases to $N = 16000$ and beyond, the option price stabilizes around 4.78, and the confidence interval narrows significantly. This behavior is consistent with the theoretical property of Monte Carlo methods, where the standard error decreases at a rate of $O(1/\sqrt{N})$. The convergence of the option price and the reduction in uncertainty demonstrate the effectiveness of the moment matching method in improving the reliability of the simulation results.

To analyze the reduction in standard error as the sample size increases, we plot the standard error against the sample size on a log-log scale.

```
# Plot Standard Error Convergence
ggplot(moment_matching_results, aes(x = Sample_Size, y = Standard_Error)) +
```

```
  geom_line(color = "red", size = 1) +
  geom_point(size = 3, color = "red") +
  scale_x_log10() +
  scale_y_log10() +
  labs(
    title = "Figure 3.7: Convergence of Standard Error with Increasing Sample Size",
    x = "Sample Size (log scale)",
    y = "Standard Error (log scale)"
  ) +
  theme_minimal()+
  theme(
    plot.title = element_text(
      size = 10,
      hjust = 0.5,
      vjust = 1.2
    )
  )
```

Figure 3.7: Convergence of Standard Error with Increasing Sample Size



The plot shows the convergence of the standard error as the sample size increases, presented on a log-log scale. As expected, the standard error decreases approximately linearly with the logarithm of the sample size, confirming the theoretical rate of convergence for Monte Carlo simulations. Specifically, the standard error follows the relationship:

$$\text{Standard Error} \propto \frac{1}{\sqrt{N}},$$

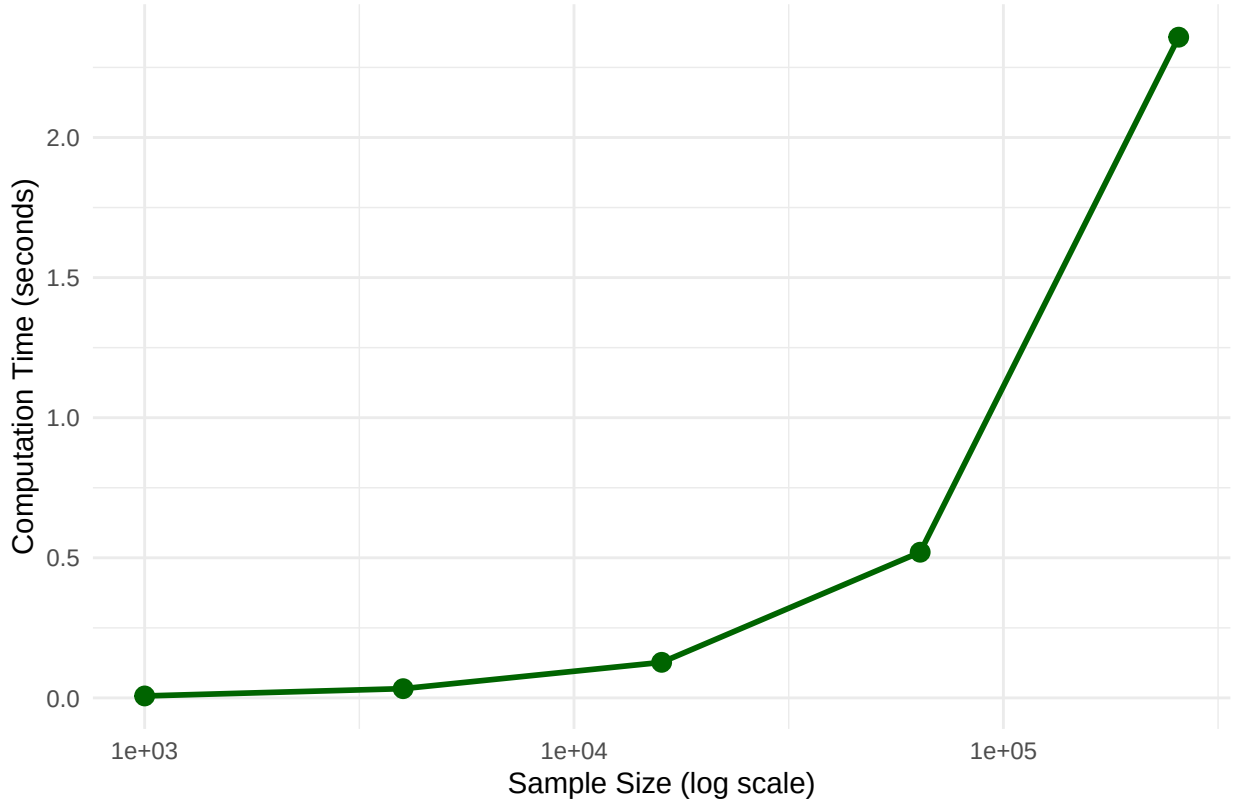
where N is the sample size. This behavior is evident in the graph, as the data points form a straight line on the log-log scale, indicating that the standard error decreases proportionally to $O(1/\sqrt{N})$.

At smaller sample sizes (e.g., $N = 1000$), the standard error is relatively high (above 0.1), reflecting greater variability in the option price estimates. As the sample size increases to $N = 256000$, the standard error reduces significantly to approximately 0.01, demonstrating improved precision. This result highlights the efficiency of increasing the sample size to achieve more accurate estimates, while also validating the effectiveness of the moment matching method in reducing variance in the Monte Carlo simulation.

To analyze the computational cost, we plot the computation time against the sample size to observe how the time scales with increasing sample size.

```
# Plot Computation Time vs Sample Size
ggplot(moment_matching_results, aes(x = Sample_Size, y = Computation_Time_sec)) +
  geom_line(color = "darkgreen", size = 1) +
  geom_point(size = 3, color = "darkgreen") +
  scale_x_log10() +
  labs(
    title = "Figure 3.8: Computation Time with Increasing Sample Size",
    x = "Sample Size (log scale)",
    y = "Computation Time (seconds)"
  ) +
  theme_minimal()+
  theme(
    plot.title = element_text(
      size = 10,
      hjust = 0.5,
      vjust = 1.2
    )
  )
```

Figure 3.8: Computation Time with Increasing Sample Size



The plot shows the computation time as a function of the sample size on a log scale. As expected, the computation time increases with the sample size. For small sample sizes, such as $N = 1000$, the computation time is negligible, close to 0.008 seconds. As the sample size increases to $N = 16000$, the computation time begins to rise gradually, reaching 0.122 seconds. For larger sample sizes, the computation time increases more significantly, reaching approximately 2.4 seconds for $N = 256000$. This behavior reflects the linear scaling of computation time with the sample size, as each additional path requires extra calculations for the asset price simulation and payoff evaluation.

The sharp increase in computation time at higher sample sizes highlights the trade-off between accuracy and computational cost in Monte Carlo simulations. While increasing the sample size reduces the standard error and improves precision, it also leads to higher computational expenses. The moment matching method efficiently reduces variance, allowing for accurate results even with moderate sample sizes, which helps balance precision and performance.

In this part of the project, we implemented the moment matching method to improve the accuracy and efficiency of Monte Carlo simulations for pricing an Asian call option. By adjusting the random samples to have a mean of zero and a variance of one, the moment matching method ensures that the simulated paths align more closely with the theoretical properties of the standard normal distribution. This adjustment reduces sampling variability and enhances the convergence rate of the simulation.

The results demonstrate that the option price estimates converge as the sample size increases, with the confidence intervals narrowing significantly. For smaller sample sizes, the estimates exhibit greater variability, as shown by wider confidence intervals. However, as the sample size grows, the option price stabilizes around 4.78, and the standard error decreases approximately at the expected rate of $O(1/\sqrt{N})$. This is consistent with the theoretical properties of Monte Carlo methods and confirms the effectiveness of moment matching in variance reduction.

The analysis of computation time reveals a linear relationship between the sample size and the time required for simulation. While larger sample sizes lead to more accurate results, they also result in higher computa-

tional costs. For instance, the computation time increases from 0.008 seconds for $N = 1000$ to 2.4 seconds for $N = 256000$. This trade-off highlights the importance of balancing precision and efficiency when selecting an appropriate sample size for practical applications.

Overall, the moment matching method proves to be a robust variance reduction technique, significantly improving the precision of Monte Carlo simulations without introducing excessive computational overhead. By reducing the standard error, it allows accurate estimates to be achieved with moderate sample sizes, making it an efficient tool for pricing complex financial derivatives like Asian options.

Future work could explore combining moment matching with other variance reduction techniques, such as the control variate method or quasi-Monte Carlo approaches, to further enhance simulation efficiency. Additionally, the applicability of moment matching to other exotic options or high-dimensional financial models could be investigated to broaden its practical use in quantitative finance.

4. Least-Squares Monte Carlo Algorithm (LSM) on Put Option

4.1 Objective and Motivation

In this section, we implement the Least Squares Monte Carlo (LSM) algorithm based on the method outlined in the paper "Valuing American Options by Simulation: A Simple Least-Squares Approach" by Longstaff and Schwartz. The LSM approach is highly effective in pricing American options, as it estimates the conditional expectation of continuation payoffs, which is crucial for determining the optimal exercise strategy. This aligns with the fundamental asset pricing theorem, where the conditional expectation under the risk-neutral measure ensures that the stock price paths $S_t^* = S_t e^{-rt}$ are modeled as martingales, reflecting no-arbitrage conditions. By leveraging the martingale property and performing cross-sectional regression on simulated paths, LSM accurately captures the fair price of the option's continuation payoff.

It is important to note that the goal of this report is not to summarize or restate the details of the LSM algorithm as presented by Longstaff and Schwartz. Rather, our primary objective is to apply the LSM method iteratively to calculate the expected value of a put option across various settings. We experiment with different combinations of parameters, such as the number of stock paths (n), the number of regressors (k), and the number of time steps (m). We will present our findings and explore how variations in these parameters affect the outcomes of the simulation. Throughout the report, we will also discuss the application of variance reduction techniques and our own data processing choices where appropriate.

One potential drawback of the LSM algorithm, if implemented naively, is its computational expense. As both the number of paths and time steps increase, the computational cost grows significantly due to the need for performing least-squares regression at each recursive step. We will also address these challenges by presenting our efforts to improve the computational efficiency of the algorithm and the resulting performance enhancements.

4.2 Generating Stock Path

Each stock path follows a geometric brownian motion under the risk-neutral measure, complying to no-arbitrage and ensuring the discounted stock S^* paths are martingales. We define our stock process for each discrete time step Δt as

$$\Delta S_t = (r - q)S_t\Delta t + \sigma S_t\sqrt{\Delta t} * \phi_t$$

where S_t is our stock value, r and q is our constant risk-free rate and continuous dividend rate, and ϕ is random standard normal variable.

Variance Reduction 1: We generate our stocks paths by creating an antithetic counterpart to ϕ_t^i for each ϕ_t^i generaated. Specifically for each time step, if we require n paths, we only need to generate $n/2$ values of $\phi_t^1, \phi_t^1, ...\phi_t^{n/2}$, and take the negative for the other half. By introducing negative correlation between paired outcomes, antithetic variates balance fluctuations in opposite directions, reducing variance and enabling the Monte Carlo estimator to converge more efficiently to the true value without increasing the sample size.

The implementation of the stock path simulation is shown in Snippet 4.1.

Snippet 4.1: Generating stock path matrix

```
simulate_S_paths <- function(S0,r,q,sigma,n,m,delta_t) {
  set.seed(3)
  S = rep(S0,n)
  S_mat = matrix(0, nrow = n, ncol = m)
  for (i in 1:m) {
    # + antithetic paths
    z_temp = rnorm(n/2)
    z = c(z_temp, -z_temp)
```

```
    delta_S =  S * (r-q) * delta_t + sigma * S * sqrt(delta_t) * z
    S = S + delta_S
    S_mat[,i] = S
  }
  return(S_mat)
}
```

4.3 Choice of Basis Function and Scaling Stock Prices

In our experiment, we ran our least squares method by representing our regressors as basis functions calculated from our underlying stock prices $S_t = S_t^1, S_t^2, ..., S_t^n$ for each time step t. Although our primary analysis will be based on using Chebyshev polynomials as our basis function, we did run simulations on Laguerre polynomials (as suggested in the paper) as well. We concluded that the converged put option values are very similar to each other, and the discrepancy between the two choices of basis functions is very small. Both polynomials are orthogonal, which helps reduce collinearity between the regressors. We omit the use of regular polynomials since they are prone to collinearity. Using Chebyshev and Laguerre polynomials has proven to offer numerical stability in performing least squares regression. For interested readers, we include our LSM results using Laguerre polynomials in Appendix 4.2.

Before applying Chebyshev polynomial conversions on the stock prices, we apply min-max scaling with boundary $[-1, 1]$ since Chebyshev polynomials are orthogonal between $[-1, 1]$ and minimize the approximation error within these boundaries. For Laguerre polynomials, we apply min-max scaling with boundary $[0, 1]$, since they are designed to take positive values $[0, \infty]$, but because they consist of exponential terms, we cap our normalized value within 1, ensuring numerical stability.

We demonstrate our implementations of our basis functions and scaling functions in Snippet 4.2.

Snippet 4.2: Basis Functions and min-max scaler

```
mm_scaler <- function(x) {

  xmin = min(x)
  xmax = max(x)
  a = 2 / (xmax - xmin)
  b = 1 - a * xmax
  return( a*x+b)
}

laguerre_basis <- function(k, x) {
  laguerre_pi <- function(i, x) {
    if (i == 0) return(exp(-x / 2))
    coeff = sapply(0:i, function(j) ((-1)^j * choose(i, j) * x^j) / factorial(j))
    return(exp(-x / 2) * rowSums(coeff))
  }
  # Use sapply to compute Laguerre polynomials and convert result to a matrix
  result = sapply(0:(k - 1), function(i) laguerre_pi(i, x))
  return(as.matrix(result))  # Convert to matrix
}

chebyshev_basis <- function(k, x) {

  n = length(x)
  X = matrix(0, nrow = n, ncol = k + 1)
  X[, 1] = 1
```

```
  if (k > 0) {
    X[, 2] = x
  }
  if (k > 1) {
    for (i in 3:(k + 1)) {
      X[, i] = 2 * x * X[, i - 1] - X[, i - 2]
    }
  }
  return(X[, -1])
}
```

4.4 Least Squares Monte Carlo Algorithm

In this section, we briefly go through our implementation of the LSM algorithm. We start by generating our n stock paths with m time steps and pre-calculate our payoff matrix. We apply backward recursive steps starting from the $m-1$th step. As for American options, we are concerned with stock paths that give us the opportunity to early exercise; paths that are in-the-money (ITM) at any time step. The idea is we want to compare the these ITM paths' payoffs with the conditional expectation of (discounted) continuation payoffs ($\hat{V}^t_{cont.}\mathbb{E}[V^t_{cont.}|X_{S_t}]$), and decide whether or not to exercise at at current time step t or at the future $t+a$ step, where $t+a$ will depend on our tracked stopping time for each stock path . We generate our conditional expectations by running least squares on our transformed S_t's in the form of Chebyshev Polynomials. The parameter k will determine the number of regressors, and hence will specify the order of the polynomials. Through out our report, we denote k as the number of regressors not including the constant term, so k specifically is the number of polynomial regressors. We also define a vector "exercise_times" to keep track of our stopping times for each stock path. At any time step and stock path, if the current payoff exceeds $\hat{V}^t_{cont.}$, then we update our stopping time by replacing stopping time $t+a$ with t. Once all time steps have been recursively traversed, we discount all the payoffs of every stock path with reference to our tracked stopping times and compute the following measures: the expectation and standard error of the discounted payoffs, percentage of stock paths that have been early exercised, and exercise times. The code snippet of our implementation can be viewed in Snippet 4.3.

Variance Reduction 2: Although we can use the entire n stock paths as our input to the least squares algorithm, we only used samples that are ITM. Using only ITM stock paths in LSM improves the accuracy of the regression by focusing on paths where the option holder might exercise. OTM paths contribute no useful information since their payoffs are always zero, adding unnecessary noise to the estimation. Limiting the regression to ITM paths reduces variance, leading to more precise estimates of the continuation value and a better exercise strategy. This approach also improves computational efficiency by decreasing the number of data points used in the regression.

Snippet 4.3: LSM Algorithm Implementation

```
LSM_put <- function(S0,K,r,q,sigma,t,n,m, k_regressors, basis_func) {
  run_ols <- function(X, Y) { # function to generate condtional expectation values using least squares
    beta <- solve(t(X) %*% X, t(X) %*% Y)
    return(X %*% beta)
  }
  # Function to run the Least-Squares Monte Carlo
  delta_t = t/m # time steps
  S_mat = simulate_S_paths(S0,r,q,sigma,n,m,delta_t)  # simulate stock paths
  exercise_times = rep(m,n) # intialize stopping times at expiration

  # create payoff dataframe for all discrete times
  payoff_mat = pmax(K-S_mat,0)

  # Recursively loop backwards to apply LSM
  for (i in (m-1):1) {
    itm_idx = payoff_mat[,i] > 0 # find ITM idx

    # get future payoffs according to current stopping times
    future_cashflows = payoff_mat[cbind(1:n, exercise_times)][itm_idx]

    # get times to discount according to current stopping times
    discount_times = delta_t * (exercise_times - i)

    # define target as present value of future payoffs
    Y = future_cashflows * exp(-r*discount_times[itm_idx])

    # filter only ITM underlying stock prices
    S_itm = S_mat[,i][itm_idx]

    # Create Laguerre polynomial regressors matrix
    X = basis_func(k_regressors, mm_scaler(S_itm))

    # Run OLS and calculate conditional expectation of Y|X
    cond_exp_Y = run_ols(cbind(1, X), Y)

    # If current payoff exceeds E[Y|X], then exercise now, if not in the future
    # To implement this logic, we update our stopping times
    current_itm_payoff = payoff_mat[,i][itm_idx]
    exercise_times[itm_idx] = ifelse(current_itm_payoff > cond_exp_Y, i, exercise_times[itm_idx])
  }
  # get future payoffs according to final stopping times, and discount them
  payoff_decisions = payoff_mat[cbind(1:n, exercise_times)]
  discount_times = delta_t * (exercise_times - i)
  option_path_values = payoff_decisions * exp(-r*discount_times)

  # option value is the mean of all option present values from each path
  option_value = mean(option_path_values)
  se = sd(option_path_values) / sqrt(n)

  # % of paths that are early exercised
  early_exercise_portion = mean(exercise_times < m)
  return(list(value = option_value, se=se, early_portion=early_exercise_portion,
```

```
                ee_times=exercise_times))}
```

4.5 Computational Efficiency Improvements

In this section, we explain our implementations aimed at increasing the computational efficiency and speed of our LSM algorithm. Our original implementation of the algorithm relied primarily on manipulating dataframe objects and passing our regressors' dataframe into R's "lm" least squares package. This method proved to be underwhelming in terms of computational speed and allocated unnecessary memory overhead in R (or, in fact, in any scientific programming language). For interested readers, the original implementation of the algorithm, including the basis function implementations, can be found in Appendix 4.1. However, we argue that starting the prototyping of the algorithm with dataframes is preferred, as it is easier to debug and more intuitive for the developer. For instance, it allows us to check the least squares results to verify whether numerical instability persists and whether the coefficients produced are stable.

After ensuring that our algorithm computations were stable and the results sound, we shifted our approach from working with dataframes to matrices, as seen in Snippets 4.2 and 4.3. In this updated approach, the entire code now consists solely of matrix computations. In particular, we discarded the "lm" package and implemented our own least squares function, solving for the coefficients using the R function "solve" and computing the dot product between the Chebyshev polynomials and coefficients. This change resulted in a significant improvement in computational efficiency.

We compare the computational speed between the "dataframe" and "matrix" implementations with three regressors in Table 4.1 and Figure 4.1 by using our most precise LSM pricer(n=256000, k=3).Computational speed is calculated as option prices per second by taking the inverse of the run time (sec). Table 4.1 shows the speed of in computing each options price as the number of m time steps increase. Figure 4.1 illustrates this in log10 scaling. As an example, for $m = 80$, the speed increased from 0.01 to 0.31 options prices per second. This means that for $n = 256000$, the corresponding runtime will have decreased from around 187.1 to 3.2 seconds. As we will see later during accuracy evaluation, for $n = 256000$, the standard error of the options price is very small. We compare the runtimes for $n = 256000$ and three regressors in Table 4.2.

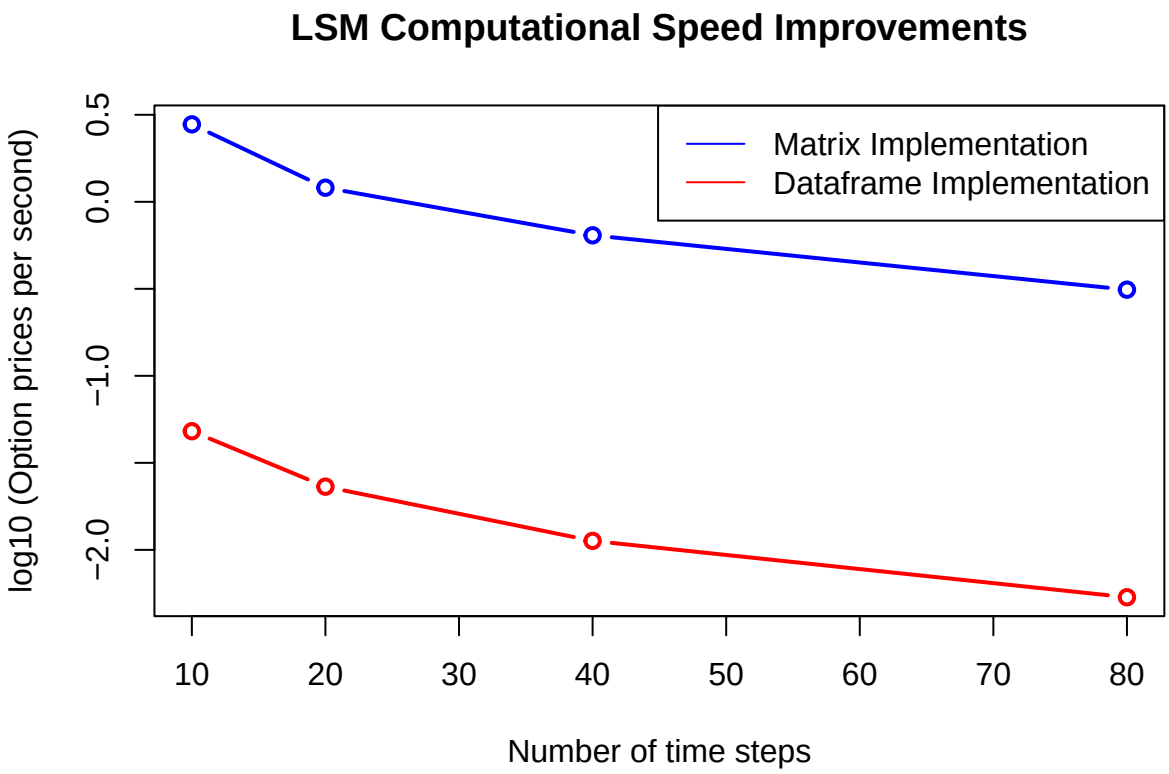
Table 4.1: Computational Speed Comparison (in units of prices/second) for n=256000 and k=3

	m=10	m=20	m=40	m=80
dataframe implementation	0.05	0.02	0.01	0.01
matrix implementation	2.79	1.21	0.64	0.31

Table 4.2: Runtime Comparison for n=256000 and k=3

	m=10	m=20	m=40	m=80
dataframe implementation	20.81	43.33	88.76	187.14
matrix implementation	0.36	0.83	1.56	3.20

Figure 4.1: Computational Speed Improvements



4.6 Experiment Implementation

We run our LSM simulation based on the following sets of n's, m's, and k's: $n : \{1000, 4000, 16000, 64000, 256000\}$, $m : \{10, 20, 40, 80\}$, $k : \{1, 2, 3\}$. For our stock parameters: $S_0 = K = 100$, $T = 1/12$, $r = 0.04$, $q = 0.02$, and $\sigma = 0.2$, corresponding to underlying stock price at $t = 0$, strike price, time to maturity, risk-free rate, and continuous dividend rate, respectively. Similar to results shown in the original paper, we also compute the early exercise value by taking the difference between the LSM American option and Black Scholes Merton closed-form (BSM) values. By definition any stock with dividends should lead to a positive early exercise value. As such we define our BSM function in Snippet 4.4. Our iterative implementation of LSM on different combinations of n's, m's, and k's is shown in Snippet 4.5.

Snippet 4.4: BSM Closed Form Function

```
# Closed form BSM solution for European put
european_put_BSM <- function(S,K,r,q,sigma,t) {

  d1 = (log(S / K) + (r - q + 0.5 * sigma^2) * t) / (sigma * sqrt(t))
  d2 = d1 - sigma * sqrt(t)

  put_price = K * exp(-r * t) * pnorm(-d2) - S0 * exp(-q * t) * pnorm(-d1)
  return(put_price)
}
```

Snippet 4.5: Experiment Implementation

```r
# main code to iterate LSM for list of m steps, n paths, and k regressors using Chebyshev Basis Functio
# output two dataframes: option value and se, early exercise value and % of early exercise paths.
# Early exercise value = American (LSM) value - European value
S0 = K = 100
t = 1/12
r = 0.04
q = 0.02
sigma = 0.2

european_put_value = european_put_BSM(S0,K,r,q,sigma,t)

m_list = c(10,20,40,80)
n_list = c(1000,1000*4, 1000*4**2, 1000*4**3, 1000*4**4)
k_regressors_list = c(1,2,3)

put_LSM_results = data.frame()
put_LSM_results2 = data.frame()

for (k_regressors in k_regressors_list) {
  temp_results = data.frame(matrix(ncol = 0, nrow = length(n_list)))
  temp_results2 = data.frame(matrix(ncol = 0, nrow = length(n_list)))

  for (m in m_list) {
    value_list = c()
    se_list = c()
    rt_list = c()

    early_portion_list = c()
    early_value_list = c()
    for (n in n_list) {
      start_time = Sys.time()
      option_LSM_res = LSM_put(S0,K,r,q,sigma,t,n,m, k_regressors, chebyshev_basis)
      end_time = Sys.time()
      runtime = as.numeric(end_time - start_time, units = "secs")
      value_list = c(value_list, round(option_LSM_res$value,4))
      se_list = c(se_list, round(option_LSM_res$se,4))
      rt_list = c(rt_list, round(runtime, 4))

      early_value_list = c(early_value_list, round(option_LSM_res$value - european_put_value,4))
      early_portion_list = c(early_portion_list, round(option_LSM_res$early_portion,3))

    }
    temp_results[[paste0("value_m",m)]] = value_list
    temp_results[[paste0("se_m",m)]] = se_list
    temp_results[[paste0("rt_m",m)]] = rt_list

    temp_results2[[paste0("EE_value_m",m)]] = early_value_list
    temp_results2[[paste0("Pct_EE_m",m)]] = early_portion_list
  }
  n_names = c()
  for (n in n_list) {n_names = c(n_names, paste0("k",k_regressors,",","n",n))}
  rownames(temp_results) = n_names
  rownames(temp_results2) = n_names
```

```
  put_LSM_results = rbind(put_LSM_results, temp_results)
  put_LSM_results2 = rbind(put_LSM_results2, temp_results2)
}

# write.csv(put_LSM_results, file = "put_LSM_cpf_res_20241127.csv")
# write.csv(put_LSM_results2, file = "put_LSM_cpf_res2_20241127.csv")
```

4.7 Result Analysis

Table 4.3 shows the LSM simulations results for every combination of n's, m's, and k's stated in section 4.6 and includes put option values (value), standard errors (se), and run times (rt) and Table 4.4 shows the results for early exercise values (EE_value) and percentage of options that were early exercised (Pct_EE). We observed that $\forall k$, the standard error of the put value decreases as we increase n. Since we choose n such that for every iteration n increases by 4 times, the standard error of decreases by 2 times, as standard error converges at a rate of order $\frac{1}{\sqrt{n}}$. Although not so obvious, as m increases, we observe that the standard error also decreases but by a very marginal amount. Run time for the simulations also increases as either one of the parameters m, n, k increases. In particular, $\forall n \in [4000, 16000]$, we were able to compute put values that have standard errors approximately between 4 to 2 cents, which is certainly admissible for American Options. Figure 4.2 shows the convergence path of the put values as n increases for difference m parameters fixing $k = 3$. We can see that for $n < 64000$ the convergence path seems to be oscillating quite significantly. For $n > 64000$, we get a very smooth converging path. For $m = 40, 80$, both path converges to a very similar value, which is in line with the fact that as m increases, we get a more accurate result. We will discuss about accuracy of our put value in section 4A where we also compute the put option value via Binomial Black-Schoels with Richardson Exprapolation.

Moving on to Table 4.4, we see the effects of including higher order polynomials (increasing k). All else equal, both the early exercise value and percentage of early exercise tends to increase. Increasing the order of polynomial basis in LSM improves the accuracy of continuation value estimation, allowing for better identification of early exercise opportunities and capturing complex, nonlinear relationships between stock price and payoffs. This leads to higher early exercise value and percentage of early exercised options. This is also inline with the fact that the early exercise value should positive for dividend paying stocks. Increasing m also increases the percentage of early exercise which comes from the increasing granularity of capturing accurate continuation payoffs. Fixing m and k, we see that increasing n reduces the variability in early exercise value and percentage of early exercise. The values stabilize and tend toward convergence as n becomes large.

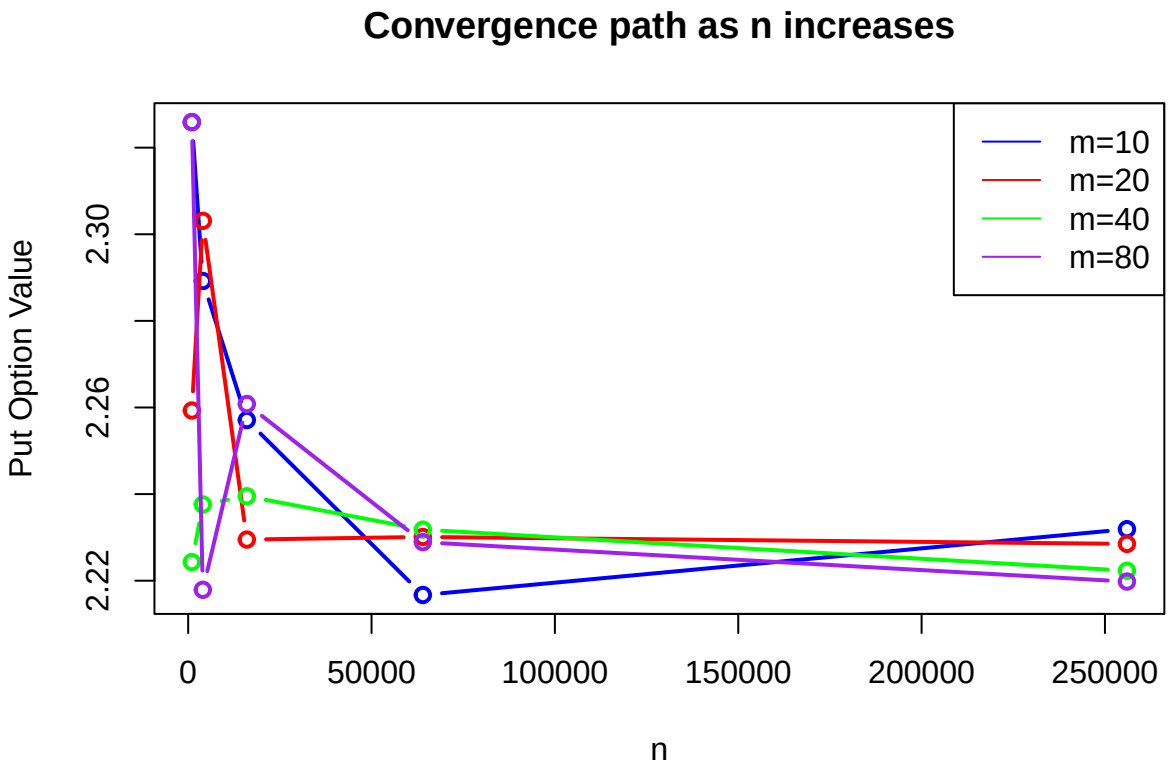
Table 4.3: Put Option Value, Standard Error, and Run time

	value_m10	se_m10	rt_m10	value_m20	se_m20	rt_m20	value_m40	se_m40	rt_m40	value_m80	se_m80	rt_m80
k1,n1000	2.2873	0.0914	0.0128	2.2247	0.0879	0.0024	2.1553	0.0842	0.0053	2.2102	0.0889	0.0103
k1,n4000	2.2846	0.0477	0.0046	2.2644	0.0451	0.0088	2.2332	0.0455	0.0234	2.2222	0.0437	0.0404
k1,n16000	2.2311	0.0233	0.0176	2.2169	0.0227	0.0909	2.2190	0.0228	0.0826	2.2327	0.0225	0.1683
k1,n64000	2.2149	0.0115	0.0839	2.2190	0.0114	0.1600	2.2125	0.0111	0.3388	2.2152	0.0111	0.7099
k1,n256000	2.2237	0.0058	0.3679	2.2205	0.0057	0.7610	2.2104	0.0056	1.5137	2.2083	0.0055	2.8193
k2,n1000	2.3399	0.0941	0.0013	2.2425	0.0858	0.0027	2.2603	0.0917	0.0054	2.3217	0.0989	0.0109
k2,n4000	2.2822	0.0474	0.0045	2.2750	0.0447	0.0100	2.2194	0.0422	0.0203	2.2445	0.0429	0.0405
k2,n16000	2.2566	0.0233	0.0194	2.2232	0.0220	0.0450	2.2291	0.0222	0.0812	2.2458	0.0216	0.1750
k2,n64000	2.2168	0.0111	0.0850	2.2249	0.0110	0.1678	2.2266	0.0109	0.3530	2.2253	0.0110	0.7242
k2,n256000	2.2291	0.0057	0.3400	2.2271	0.0055	0.7623	2.2190	0.0054	1.5714	2.2151	0.0053	3.0637
k3,n1000	2.3259	0.0927	0.0014	2.2593	0.0862	0.0030	2.2243	0.0868	0.0056	2.3259	0.0980	0.0115
k3,n4000	2.2892	0.0479	0.0049	2.3031	0.0457	0.0102	2.2376	0.0452	0.0211	2.2179	0.0403	0.0469
k3,n16000	2.2571	0.0231	0.0198	2.2295	0.0220	0.0410	2.2395	0.0225	0.0945	2.2608	0.0230	0.1856
k3,n64000	2.2167	0.0111	0.0902	2.2301	0.0113	0.1865	2.2318	0.0109	0.3511	2.2289	0.0112	0.7842
k3,n256000	2.2319	0.0057	0.3588	2.2285	0.0055	0.8297	2.2223	0.0055	1.5572	2.2198	0.0054	3.1976

Table 4.4: Early Exercise Value, Percentage of Early Exercise

	EE_value_m10	Pct_EE_m10	EE_value_m20	Pct_EE_m20	EE_value_m40	Pct_EE_m40	EE_value_m80	Pct_EE_m80
k1,n1000	0.0723	0.236	0.0096	0.235	-0.0598	0.197	-0.0048	0.515
k1,n4000	0.0695	0.142	0.0494	0.261	0.0181	0.191	0.0071	0.264
k1,n16000	0.0160	0.170	0.0018	0.176	0.0040	0.197	0.0177	0.206
k1,n64000	-0.0001	0.158	0.0040	0.169	-0.0026	0.199	0.0002	0.207
k1,n256000	0.0087	0.155	0.0054	0.182	-0.0046	0.201	-0.0068	0.209
k2,n1000	0.1249	0.276	0.0275	0.339	0.0453	0.255	0.1066	0.433
k2,n4000	0.0671	0.184	0.0600	0.334	0.0043	0.342	0.0295	0.338
k2,n16000	0.0415	0.264	0.0081	0.296	0.0140	0.281	0.0307	0.326
k2,n64000	0.0018	0.240	0.0099	0.264	0.0116	0.303	0.0102	0.297
k2,n256000	0.0140	0.240	0.0120	0.275	0.0040	0.301	0.0001	0.313
k3,n1000	0.1108	0.340	0.0443	0.359	0.0092	0.334	0.1108	0.500
k3,n4000	0.0741	0.178	0.0880	0.346	0.0226	0.371	0.0028	0.402
k3,n16000	0.0420	0.294	0.0145	0.333	0.0244	0.350	0.0458	0.367
k3,n64000	0.0017	0.279	0.0151	0.300	0.0168	0.354	0.0139	0.350
k3,n256000	0.0168	0.274	0.0135	0.319	0.0072	0.348	0.0047	0.359

Figure 4.2: Options Convergence path when k=3



4A. Put Option Pricing via Binomial Black-Scholes with Richardson Extrapolation

4A.1 Objective

In this section, we implement the Binomial Black-Scholes with Richardson Extrapolation (BBSR) to compute the put option value. This allows us to compare the accuracy of the put value between using BBSR and LSM. As suggested in the paper "Broadie and Detemple, 1996, American Option Valuation: New Bounds,

Approximations, and a Comparison of Existing Methods", we will also define our benchmark for the put option value using a standard Binomial Tree pricer with $m = 15000$ time steps. Through this section, we refer to this benchmark as the "true value". We will also compare computational speeds and convergence behavior with respect to the true put option value.

4A.2 BBSR Implementation

The BBSR algorithm derived from the standard Bimomial Tree algorithm but with some modifications. The first modification is on the $m-1$th time step, where the option continuation value at every node is replaced with the Black-Scholes option value. The binomial tree combined with the first modification is called the Binomial Black-Scholes (BBS). The second modification is applying Richardson extrapolation on the computed option value. In our implementation, we use "two-point" Richardson extrapolation. Referencing to the paper, we use the "two-point" Richardson extrapolation as the authors suggest that higher order extrapolation does not add value in terms of accuracy. Hence as an example, if were to compute the BBSR price V for $m = 40$, the Richardson extrapolation formula used to apply on the BBS price is $V = 2 \cdot V1 - V0$, where $V1$ and $V0$ are the BBS prices on $m_1, m_0 = 40, 20$, respectively. The BBSR code snippet can be found in Snippet 4.6. In computing the option put value, we use the same stock parameters as defined in section 3. We ran our algorithm using $m : \{10, 20, 40, 80, 160, 320\}$. We also record the run times for m cases. For interested readers, we include our implementation of the standard Binomial Tree algorithm with $m = 15000$ in Appendix 4.3.

Snippet 4.6: Experiment Implementation

```
# Binomial Black-Scholes Algorithm
BBS <- function(S0,r,q,sigma,m,t) {
  dt = t/m
  u = exp(sigma*sqrt(dt))
  d = 1/u
  p = (exp((r-q)*dt) - d)/ (u-d)
  df = exp(-r*dt)

  S = matrix(0, nrow=m+1, ncol=m+1)
  V = matrix(0, nrow=m+1, ncol=m+1)

  for (j in 1:(m+1)) {
    for (i in 1:j) {
      S[i,j] = S0 * u**(j-i) * d**(i-1)
    }
  }
  # discard option value at time step m+1, because we can calculate in from indexing at node m
  for (j in (m):1) {

    for (i in 1:j) {
      if (j == m) {
        v_continuation = european_put_BSM(S[i,j], K, r, q, sigma, dt)

        V[i, j] = max(v_continuation, max(K - S[i,j], 0))
      }

      else {
        v_continuation = df * (p* V[i,j+1] + (1-p) *V[i+1,j+1])
        if (j==1) {

          V[i,j] = v_continuation
        }
```

```
        else {

          V[i,j] = max(v_continuation, max(K - S[i,j], 0))
        }

      }
    }
  }
  return(V[1,1])
}

# Richardson Extrapolation Function
richardson_extrap <- function(S0,r,q,sigma,t,m) {

  step_ratio = 2
  V0 = BBS(S0, r, q, sigma, m/step_ratio, t)

  V1 = BBS(S0, r, q, sigma, m, t)
  V = 2*V1 - V0

  return(V)
}
S0 = K = 100
t = 1/12
r = 0.04
q = 0.02
sigma = 0.2
m_list = c(10,20,40,80)
V_bbsr = c()
V_bbsr_rt = c()
for (m in m_list) {
  start_time = Sys.time()
  V_bbsr = c(V_bbsr, c(richardson_extrap(S0,r,q,sigma,t,m)))
  end_time = Sys.time()
  run_time = as.numeric(end_time - start_time, units = "secs")
  V_bbsr_rt = c(V_bbsr_rt, round(run_time,4))
}
```

4A.3 Discussion on Accuracy, Path Convergence, and Computational Efficiency

Accuracy

We compare accuracy, convergence behavior and computational speed of LSM($k = 3, n = 64000$), LSM($k = 3, n = 256000$), and BBSR methods with varying m steps where $m : \{10, 20, 40, 80\}$. We purposely chose LSM($k = 3, n = 256000$) to reflect the most precise and method among all LSM runs and include LSM($k = 3, n = 64000$) to sacrifice some precision for speed. This will provide us with a more meaningful comparison once we include the BBSR method into the picture. We choose the absolute error metric as our basis for evaluating accuracy.

LSM price accuracy evaluation using BSSR price as benchmark

We now evaluate our LSM model's accuracy using BSSR model as the benchmark. The absolute error values are shown in Table 4.5, where we measure the difference between BSSR prices and LSM prices by varying m steps. Overall, LSM prices are considered to be accurate. For $m \geq 20$, we achieve absolute error values of under 0.01, which is quite remarkable. From our results, we can see that we do not require as much as

256000 stock paths to get an accurate price, as the LSM model with 64000 stock paths are enough to achieve sufficient accuracy.

Table 4.5: LSM price accuracy with BSSR price as benchmark

	BBSR	LSM(n=64000)	Abs. Error	LSM(n=256000)	Abs. Error
m=10	2.292439	2.2167	0.0757395	2.2319	0.0605395
m=20	2.227690	2.2301	0.0024101	2.2285	0.0008101
m=40	2.226110	2.2318	0.0056905	2.2223	0.0038095
m=80	2.225939	2.2289	0.0029614	2.2198	0.0061386

Compare LSM and BSSR prices with the True Value as benchmark

Here, we include the true value (from Binomial(n=15000)) in our comparison and use the true value as the benchmark for the BSSR and LSM prices. The results are shown in Table 4.6. We can see that the BBSR method is very accurate and superior to the LSM method at every m. At $m = 80$, the error of the BSSR value is small and negligible in practice. This is not to say that the LSM algorithm did a poor job, because we can see that the absolute error is of order 10^{-3}, which is certainly less than a cent.

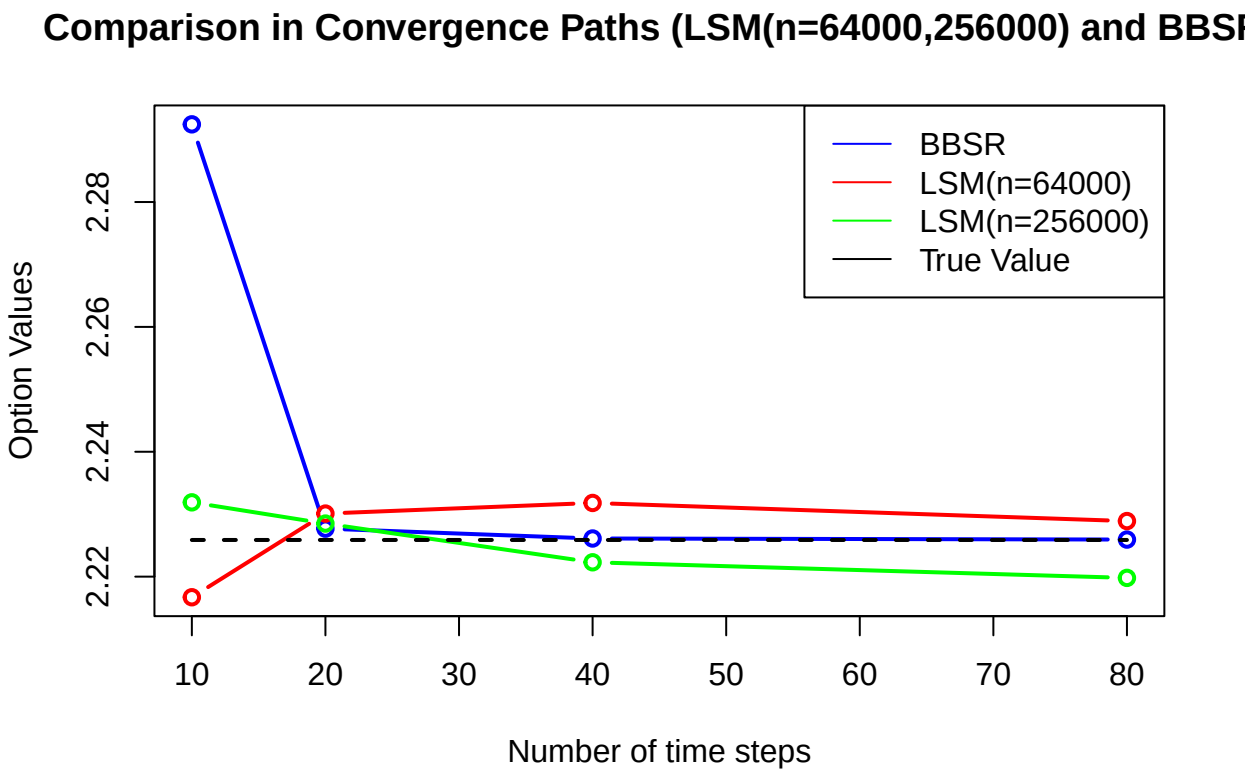
Table 4.6: Accuracy Comparison with True Value as benchmark

	True Value	LSM(n=64000)	Abs. Error	LSM(n=256000)	Abs. Error	BBSR	Abs. Error
m=10	2.225874	2.2167	0.009174	2.2319	0.006026	2.292439	0.0665655
m=20	2.225874	2.2301	0.004226	2.2285	0.002626	2.227690	0.0018159
m=40	2.225874	2.2318	0.005926	2.2223	0.003574	2.226110	0.0002355
m=80	2.225874	2.2289	0.003026	2.2198	0.006074	2.225939	0.0000646

Convergence Behavior

Here, we discuss the convergence behaviors of the algorithms and Figure 4.3 shows the plot each algorithm's convergence path. In terms of converging behavior, the LSM algorithm started off with a relatively low discrepancy compared to the true value and converges more smoothly compared to the BSSR. As comparison, the BSSR algorithm started of with a relatively high discrepancy and descends with a rapid rate and converges to the true value beautifully at around $m = 40$. This implies that the LSM algorithm converges with a slower rate than the BSSR and might require more time steps to produce a more accurate result. The result shows that the BSSR is very powerful in that it is able to converge to the true price by using a very low computational setting since it is already accurate for a few time steps (i.e. $m = 40$).

Figure 4.3: Options Convergence Path with True Value as benchmark



Computational Speed

Table 4.7 shows the run times and computational speed. Computational speed is calculated as option prices per second by taking the inverse of the run time (sec). In particular, we observed that the computational efficiency results for BSSR method is not linear in that for smaller time steps, it takes longer time to run the algorithm. The increased runtime for smaller time steps (m) in the implementation is likely due to fixed overheads in initializing and updating matrices, which dominate when m is small and the computations are less intensive. Nevertheless, if we compare the speeds and run time, BSSR is fast and efficient. At $m = 80$, using the observed speed of 344 prices per second, we can say that it is certainly viable in practice. Compared to LSM ($n = 64000$), we can compute only 3 prices per second.

Table 4.7: Run Times (seconds/price) and Speed Comparison (prices/sec)

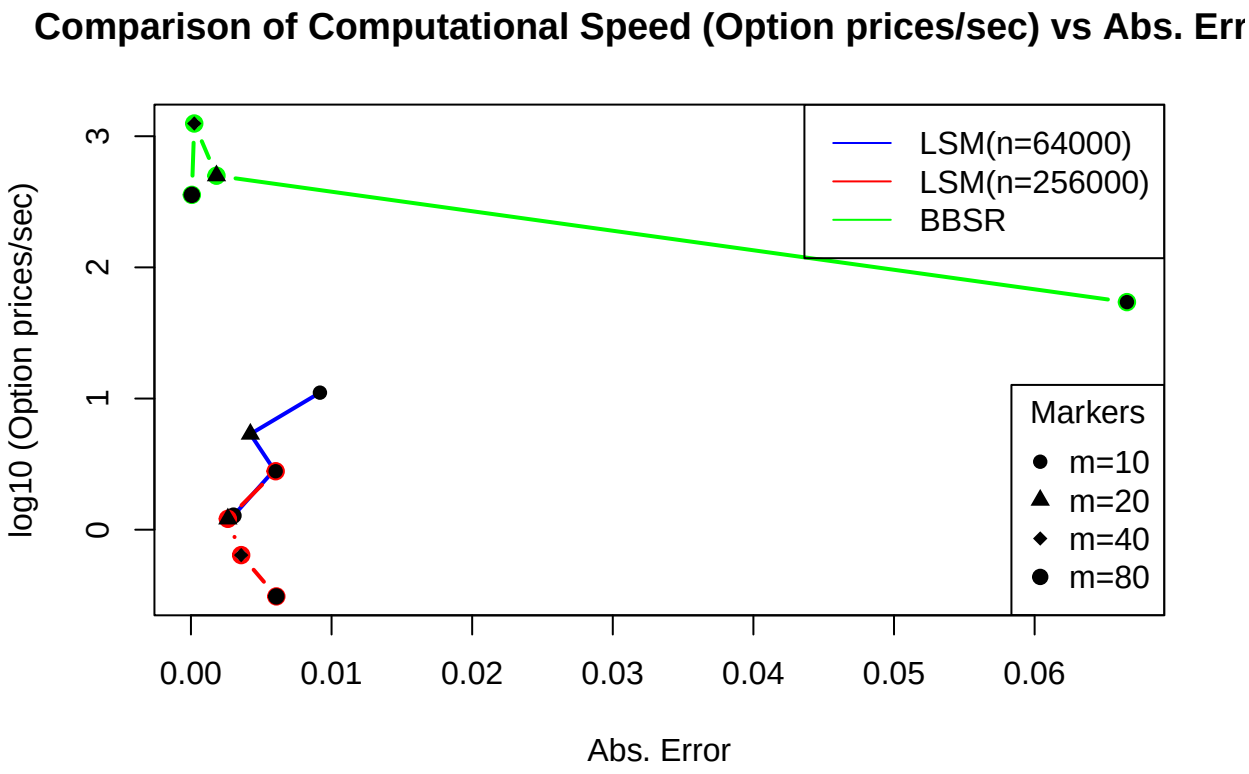
	LSM(n=64000) Run Time	LSM(n=64000) Speed	LSM(n=256000) Run Time	LSM(n=256000) Speed	BBSR Run Time	BBSR Speed
m=10	0.0902	11.09	0.3588	2.79	0.0184	54.35
m=20	0.1865	5.36	0.8297	1.21	0.0020	500.00
m=40	0.3511	2.85	1.5572	0.64	0.0008	1250.00
m=80	0.7842	1.28	3.1976	0.31	0.0028	357.14

4A.4 Accuracy vs Computational Speed

We conclude that in our specific case, the BSSR is superior to LSM in that it requires very small computational effort to compute a very accurate option price. BBSR wins in both the race for accuracy and speed. Figure 3.4 illustrates the plot that examines the absolute error and speed of each algorithm with varying time steps. The absolute error values are computed in comparison again the true value (based on Binomial($n = 15000$)). In the plot, we scale the speed by taking the log_{10} for better representation for comparison. In this plot, the preferred model would be at the top left area, and these models would exhibit relatively high speeds and low errors. Although the plot shows that the accuracy between BSSR and LSM

is very close and hence the LSM model is very accurate, the computational speed is the winning factor and we conclude that BSSR certainly practical, accurate, and relatively simple to implement.

Figure 4.4: Speed vs Abs. Error Plot Comparison



Appendix

4.1: Non-vectorized version of LSM Algorithm using DataFrames

```
simulate_S_paths0 <- function(S0,r,q,sigma,n,m,delta_t) {
  #Function for simulating discrete-time stock paths of m steps with n paths based on the
  # BSM risk neutral measure
  set.seed(3)
  S = rep(S0,n)
  S_df = data.frame(matrix(ncol = 0, nrow = n))
  for (i in 1:m) {
    # + antithetic paths
    z_temp = rnorm(n/2)
    z = c(z_temp, -z_temp)
    delta_S =  S * (r-q) * delta_t + sigma * S * sqrt(delta_t) * z
    S = S + delta_S
    S_df[[paste0("S",i)]] = S
  }
  return(S_df)
}

laguerre_basis0 <- function(k, x) {
  laguerre_pi <- function(i, x) {
    if (i == 0) return(exp(-x / 2))
    coeff <- sapply(0:i, function(j) ((-1)^j * choose(i, j) * x^j) / factorial(j))
    return(exp(-x / 2) * rowSums(coeff))
  }
  return(as.data.frame(sapply(0:(k-1), function(i) laguerre_pi(i, x))))
}

chebyshev_basis0 <- function(k,x) {
  X = data.frame(matrix(ncol = 0, nrow = length(x)))
  X[[paste0("CB",0)]] = rep(1, length(x))
  X[[paste0("CB",1)]] = x
  if (k > 1) {
    for (i in 2:k) {
      X[[paste0("CB",i)]] = 2 * x * X[[paste0("CB",i-1)]] - X[[paste0("CB",i-2)]]
    }
  }
  return(subset(X, select = -CB0))
}
```

```
LSM_put0 <- function(S0,K,r,q,sigma,t,n,m, k_regressors, basis_func) {
  # Function to run the Least-Squares Monte Carlo
  delta_t = t/m # time steps
  S_df = simulate_S_paths(S0,r,q,sigma,n,m,delta_t) # simulate stock paths
  exercise_times = rep(m,n) # intialize stopping times at expiration
  # create payoff dataframe for all discrete times
  payoff_df = data.frame(apply(K - S_df, c(1, 2), function(x) max(x, 0)))

  # scaling underlying stock prices to prevent numerical issues with polynomials
  S_scaled_df = S_df
  # Recursively loop backwards to apply LSM
```

```
  for (i in (m-1):1) {
    itm_idx = payoff_df[,i] > 0 # find ITM idx
    # get future payoffs according to current stopping times
    future_cashflows = mapply(function(row, col) payoff_df[row, col], row = 1:nrow(payoff_df), col = ex
    # get times to discount according to current stopping times
    discount_times = delta_t * (exercise_times - i)
    # define target as present value of future payoffs
    Y = future_cashflows * exp(-r*discount_times[itm_idx])
    # filter only ITM underlying stock prices
    S_itm = S_scaled_df[,i][itm_idx]
    # Create Laguerre polynomial regressors matrix
    X = basis_func(k_regressors, mm_scaler(S_itm))
    # Run OLS and calculate conditional expectation of Y|X
    model = lm(Y ~ ., data=X)
    cond_exp_Y = predict(model, newdata = data.frame(X))
    names(cond_exp_Y) = NULL
    # If current payoff exceeds E[Y|X], then exercise now, if not in the future
    # To implement this logic, we update our stopping times
    current_itm_payoff = payoff_df[,i][itm_idx]
    exercise_times[itm_idx] = ifelse(current_itm_payoff > cond_exp_Y, i, exercise_times[itm_idx])
  }
  # get future payoffs according to final stopping times, and discount them
  payoff_decisions = mapply(function(row, col) payoff_df[row, col], row = 1:nrow(payoff_df), col = exer
  discount_times = delta_t * (exercise_times - i)
  option_path_values = payoff_decisions * exp(-r*discount_times)
  # option value is the mean of all option present values from each path
  option_value = mean(option_path_values)
  se = sd(option_path_values) / sqrt(n)
  # % of paths that are early exercised
  early_exercise_portion = mean(exercise_times < m)
  return(list(value = option_value, se=se, early_portion=early_exercise_portion, ee_times=exercise_time
}
```

4.2: LSM Results using Laguerre Basis Functions

	value_m10	se_m10	rt_m10	value_m20	se_m20	rt_m20	value_m40	se_m40	rt_m40	value_m80	se_m80	rt_m80
k1,n1000	2.3032	0.0898	0.0098	2.1934	0.0810	0.0032	2.2247	0.0859	0.0054	2.1483	0.0717	0.0113
k1,n4000	2.2466	0.0438	0.0130	2.2793	0.0443	0.0111	2.2236	0.0410	0.0191	2.2447	0.0418	0.0399
k1,n16000	2.2483	0.0227	0.0235	2.2251	0.0214	0.0372	2.2137	0.0210	0.0865	2.2305	0.0202	0.1756
k1,n64000	2.2153	0.0108	0.1276	2.2201	0.0106	0.1755	2.2241	0.0105	0.3117	2.2041	0.0101	0.6608
k1,n256000	2.2301	0.0055	0.3479	2.2246	0.0053	0.7727	2.2151	0.0052	1.3900	2.2071	0.0051	2.8284
k2,n1000	2.3490	0.0943	0.0017	2.2395	0.0853	0.0032	2.2467	0.0903	0.0075	2.2967	0.0965	0.0171
k2,n4000	2.2861	0.0475	0.0054	2.2737	0.0449	0.0111	2.2332	0.0424	0.0234	2.2369	0.0424	0.0511
k2,n16000	2.2506	0.0230	0.0217	2.2254	0.0221	0.0466	2.2257	0.0221	0.0918	2.2491	0.0216	0.2209
k2,n64000	2.2194	0.0112	0.0881	2.2252	0.0110	0.1796	2.2275	0.0109	0.3977	2.2196	0.0108	0.7909
k2,n256000	2.2276	0.0056	0.3798	2.2269	0.0055	0.8221	2.2196	0.0054	1.5772	2.2155	0.0053	3.3583
k3,n1000	2.3351	0.0934	0.0020	2.2448	0.0858	0.0041	2.2155	0.0859	0.0080	2.3165	0.0994	0.0363
k3,n4000	2.2879	0.0480	0.0065	2.2800	0.0441	0.0126	2.2578	0.0451	0.0253	2.2592	0.0431	0.0522
k3,n16000	2.2540	0.0229	0.0249	2.2264	0.0219	0.0535	2.2362	0.0225	0.1077	2.2547	0.0225	0.2351
k3,n64000	2.2158	0.0111	0.0995	2.2264	0.0111	0.2162	2.2295	0.0109	0.4092	2.2252	0.0111	0.8643
k3,n256000	2.2316	0.0056	0.4085	2.2271	0.0056	0.8942	2.2206	0.0054	1.8166	2.2188	0.0054	3.7954

	EE_value_m10	Pct_EE_m10	EE_value_m20	Pct_EE_m20	EE_value_m40	Pct_EE_m40	EE_value_m80	Pct_EE_m80
k1,n1000	0.0882	0.317	-0.0217	0.377	0.0097	0.393	-0.0668	0.459
k1,n4000	0.0316	0.296	0.0643	0.365	0.0085	0.396	0.0297	0.399
k1,n16000	0.0333	0.304	0.0101	0.354	-0.0013	0.379	0.0154	0.410
k1,n64000	0.0002	0.306	0.0050	0.344	0.0090	0.378	-0.0110	0.401
k1,n256000	0.0151	0.299	0.0096	0.346	0.0001	0.380	-0.0079	0.397
k2,n1000	0.1339	0.274	0.0244	0.341	0.0316	0.255	0.0816	0.505
k2,n4000	0.0710	0.182	0.0586	0.338	0.0182	0.371	0.0218	0.374
k2,n16000	0.0355	0.266	0.0104	0.296	0.0106	0.313	0.0341	0.375
k2,n64000	0.0043	0.234	0.0101	0.263	0.0125	0.319	0.0045	0.349
k2,n256000	0.0126	0.237	0.0119	0.277	0.0045	0.319	0.0005	0.348
k3,n1000	0.1200	0.329	0.0297	0.357	0.0004	0.335	0.1014	0.493
k3,n4000	0.0729	0.176	0.0650	0.346	0.0428	0.372	0.0442	0.380
k3,n16000	0.0390	0.286	0.0113	0.319	0.0212	0.336	0.0397	0.365
k3,n64000	0.0007	0.272	0.0113	0.295	0.0145	0.347	0.0101	0.341
k3,n256000	0.0166	0.270	0.0120	0.307	0.0056	0.340	0.0037	0.350

4.3: Code Snippet of Binomial Model with n=15000 time steps

```
BinomTree <- function(S0,r,q,sigma,m,t) {
  dt = t/m
  u = exp(sigma*sqrt(dt))
  d = 1/u

  p = (exp((r-q)*dt) - d)/ (u-d)

  df = exp(-r*dt)


  S = matrix(0, nrow=m+1, ncol=m+1)
  V = matrix(0, nrow=m+1, ncol=m+1)

  for (j in 1:(m+1)) {

    for (i in 1:j) {
      S[i,j] = S0 * u**(j-i) * d**(i-1)

    }
  }
  # discard option value at time step m+1, only
  for (j in (m):1) {

    for (i in 1:j) {
      if (j == m) {
        v_continuation = df * (p*  max(K - S[i,j+1], 0) + (1-p) * max(K - S[i,j+1], 0))

        V[i, j] = max(v_continuation, max(K - S[i,j], 0))

      }

      else {

        if (j==1) {

          V[i,j] = v_continuation
        }
        else {

          V[i,j] = max(v_continuation, max(K - S[i,j], 0))
        }
      }

    }

  }
  return(V[1,1])
}
m=15000
V_binom = BinomTree(S0,r,q,sigma,m,t)
```

5. Conclusion

This report summarized how Monte Carlo simulation can be applied to pricing three types of options: European vanilla put options, Asian call options, and American vanilla put options. The results showed how to price complex options, especially when a closed formed solution was unavailable. Variance reduction techniques, such as the antithetic approach, the control variate approach, and the quasi-random sequence approach were used to improve the accuracy of the option price and the computational efficiency. In addition, the Longstaff-Schwartz method for American options and the Binomial Black-Scholes with Richardson Extrapolation method were used as benchmarks for early exercise strategies. This report showed how Monte Carlo simulations are able to precisely price complex options and investigated convergence speed and computational efficiency across the different methods.

In pricing the European vanilla put option with the following parameters, $S_0 = K = 100$, $T = 0.5$, $r = 0.04$, $q = 0.02$, $\sigma = 0.2$, the exact European put option price under Black-Scholes-Merton was calculated to be \$5.0746. The standard Monte Carlo approach was the slowest at converging to within \$0.01 of the exact price, with a sample size of around 256,000. The antithetic approach was the next fastest, converging to within \$0.01 of the exact price with a sample size of around 64,000 (32,000 anithetic pairs). The Quasi-Monte-Carlo approach was the fastest, converging to within \$0.01 of the exact price in around 4,000 samples.

In pricing the Asian call option under the Black-Scholes-Merton model, we applied the Monte Carlo simulation method combined with the moment matching technique to improve convergence and accuracy. The moment matching method adjusts the simulated standard normal samples to ensure they have a mean of zero and variance of one, aligning them more closely with the theoretical distribution. This adjustment reduces variance in the simulation, leading to faster and more stable convergence of the option price. The results demonstrated that the option price estimates stabilized as the sample size increased, converging to approximately \$4.78. The standard error decreased at the theoretical rate of $O(1/\sqrt{N})$, confirming the efficiency of the Monte Carlo method. For smaller sample sizes, the estimates exhibited greater variability, reflected in wider confidence intervals. However, as the sample size increased to $N = 256,000$, the confidence interval narrowed significantly, and the standard error reduced to 0.009, achieving high precision. The computation time scaled linearly with the sample size, increasing from a negligible 0.008 seconds for $N = 1000$ to 2.4 seconds for $N = 256,000$. This trade-off between computational cost and accuracy highlights the importance of selecting an appropriate sample size for practical applications. The moment matching method proved effective in reducing variance and improving simulation efficiency, making it a reliable tool for pricing exotic options like Asian call options when closed-form solutions are unavailable.

In implementing the Least-Squares Monte Carlo (LSM) algorithm based on Longstaff-Schwartz to price an American Option on a dividend-paying stock, we included variance reduction techniques by using antithetic samples and excluding non-in-the-money paths when running the backward recursive least-squares regression. In terms of computational efficiency, replacing the the dataframe implementation with matrix implementation have proven to have significant improvement in pricing speed. Using the LSM model with $n = 256000$ stock paths, $k = 3$ regressors, and $m = 80$ time steps, the option value converged to 2.2198 with a standard error of under a cent. The Binomial Black-Scholes with Richardson Extrapolation (BSSR) model was also implemented and used as a benchmark in evaluating the LSM model's accuracy. The resulting BSSR price with 80 steps was 2.225939, and with the LSM price of 2.2289 using $n = 64000$ stock paths, $k = 3$ regressors, and $m = 80$ time steps, an absolute error of around 0.003 was achieved. Hence, in this specific experiment the LSM model with 64000 stock paths represents an admissible trade off between speed and accuracy in pricing American options.