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 equation 1.

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

Table 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

As the underlying asset price S_t, is calculated in equation 1, the price of the European vanilla put option with strike price K and maturity T is given in equation 2. Because we are pricing a European put option, 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.

$$p = e^{-rT}\mathbb{E}\left[\max(0, K - S_T)\right] \tag{2}$$

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: **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 equation 1.

$$S_t = S_0 \exp\left(\left(r - q - \frac{1}{2}\sigma^2\right)t + \sigma\sqrt{t}Z_t\right) \tag{3}$$

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 in equations 4 and 5.

$$\bar{x} = \left(1 - \frac{1}{k}\right)\bar{x} + \frac{1}{k}p \tag{4}$$

$$\bar{y} = \left(1 - \frac{1}{k}\right)\bar{y} + \frac{1}{k}p^2 \tag{5}$$

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 in equation 6, after all iterations are complete.

$$\text{se} = \sqrt{\frac{\bar{y} - \bar{x}^2}{n-1}} \tag{6}$$

Following the calculation of the standard error, the lower bound and upper bounds of the 95% confidence interval are computed as shown in equations 7 and 8.

$$\text{lower\_bound} = \bar{x} - z_\alpha \cdot \text{se} \tag{7}$$

$$\text{upper\_bound} = \bar{x} + z_\alpha \cdot \text{se} \tag{8}$$

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, as shown in equation 9, with provided input parameters.

$$p_{\text{exact}} = K\exp(-rt)\Phi(-d_2) - S_0\exp(-qt)\Phi(-d_1) \tag{9}$$

$$d_1 = \frac{\log(S_0/K) + \left(r - q + \frac{1}{2}\sigma^2\right)t}{\sigma\sqrt{t}} \tag{10}$$

$$d_2 = \frac{\log(S_0/K) + \left(r - q - \frac{1}{2}\sigma^2\right)t}{\sigma\sqrt{t}} \tag{11}$$

Equation 12 shows the calculation of the absolute pricing error.

$$\text{absolute\_price\_error} = |\bar{x} - p_{\text{exact}}| \tag{12}$$

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)
```

Table 3: **Example Output: Monte Carlo Simulation Results for European Put Option Prices**

N	Option Price	Std. Error	LB CI	UB CI	Exact Price	Abs. Price Error	Comp. Time
1000	5.2603	0.2273	4.8147	5.7058	5.0746	0.1856	0.0009
4000	4.9925	0.1115	4.7740	5.2110	5.0746	0.0822	0.0034
16000	5.0265	0.0564	4.9159	5.1370	5.0746	0.0482	0.0135
64000	5.0977	0.0285	5.0419	5.1536	5.0746	0.0231	0.0491
256000	5.0650	0.0142	5.0322	5.0877	5.0746	0.0997	0.1874
1024000	5.0663	0.0071	5.0524	5.0802	5.0746	0.0084	0.7294

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 \$5.0663, which is and absolute price error of \$0.0084. A sample size of 1,024,000 also has a standard error of less than 1 cent, 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)
```

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). 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

Table 4: **Example Output: Monte Carlo Simulation Results with Antithetic Variates**

N	(N/2)	Option Price	Std. Error	LB CI	UB CI	Exact Price	Price Error	Comp. Time
1000	500	4.8683	0.1547	4.5651	5.1715	5.0746	0.2063	0.0007
4000	2000	5.0508	0.0806	4.8928	5.2087	5.0746	0.0239	0.0027
16000	8000	5.1566	0.0401	5.0780	5.2352	5.0746	0.0820	0.0108
64000	32000	5.0677	0.0200	5.0284	5.1070	5.0746	0.0069	0.0456
256000	128000	5.0688	0.0100	5.0491	5.0885	5.0746	0.0059	0.1881
1024000	512000	5.0681	0.0050	5.0583	5.0780	5.0746	0.0065	0.7244

samples (n). For an even more precise answer, 128,000 antithetic pairs can be generated to have an even smaller absolute price error and a standard error of around 1 cent. Discussions of results are included in the below discussion section.

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.

```
library(randtoolbox)
```

```
## Loading required package: rngWELL
```

```
## This is randtoolbox. For an overview, type 'help("randtoolbox")'.
```

```
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)
  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)
```

The total number of samples required to be within 1 cent of the exact option price is around 4000 samples. Note this is much lower compared to the standard (256,000) and antithetic approaches (64,000 (32,000 antithetic pairs)). However a standard error of around 10 cents is less precise, as if more precision is needed,

Table 5: **Exmaple Output: Monte Carlo Simulation Results with QMC**

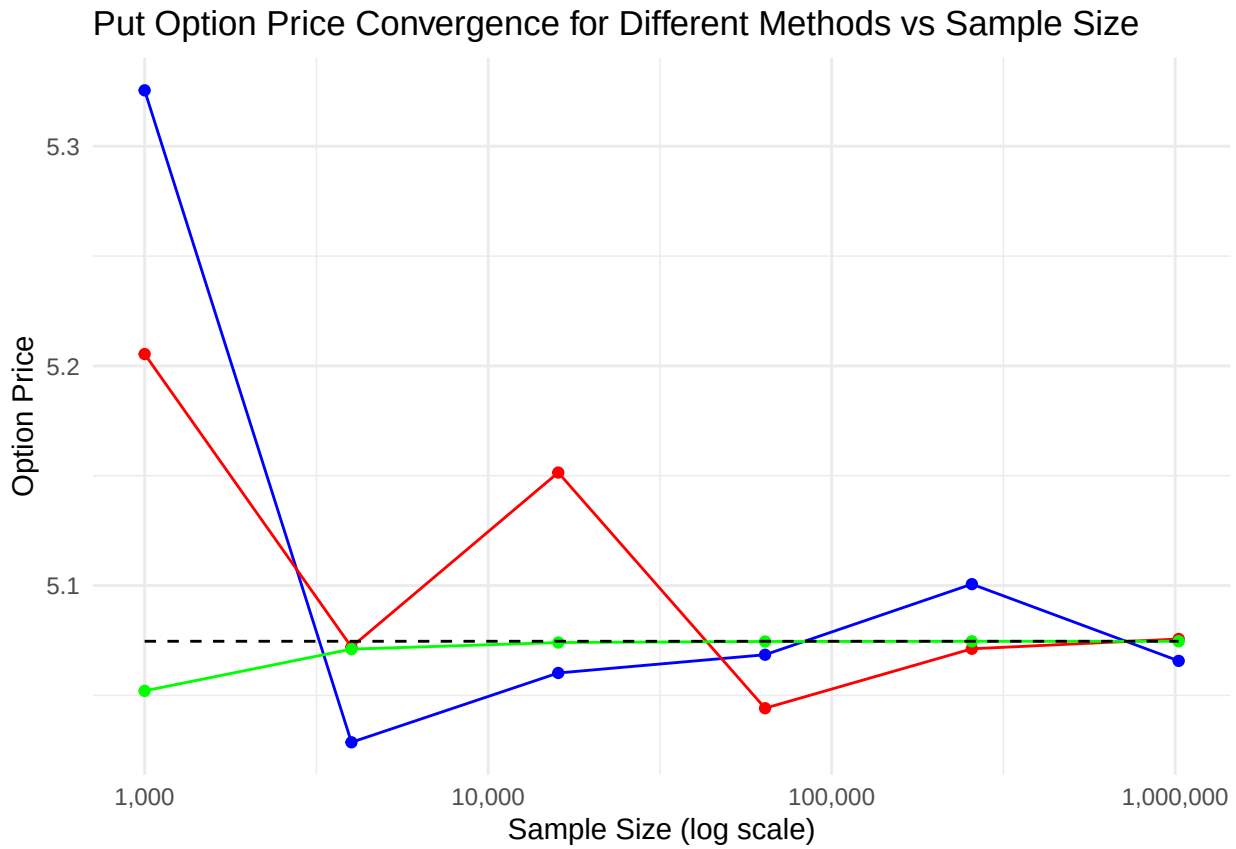
N	Option Price	Std. Error	LB CI	UB CI	Exact Price	Price Error	Comp. Time
1000	5.0521	0.22499	4.6111	5.4930	5.0746	0.0226	0.0014
4000	5.0711	0.11334	4.8489	5.2932	5.0746	0.0036	0.0030
16000	5.0741	0.05677	4.9628	5.1853	5.0746	0.0006	0.0133
64000	5.0746	0.02840	5.0189	5.1302	5.0746	0.0001	0.0594
256000	5.0746	0.01420	5.0468	5.1025	5.0746	0.00001	0.1943
1024000	5.0746	0.00710	5.0607	5.0886	5.0746	0.00000037	0.7538

sample size will need to be increased. See discussion sections for further analysis of results.

2.4. Discussions

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

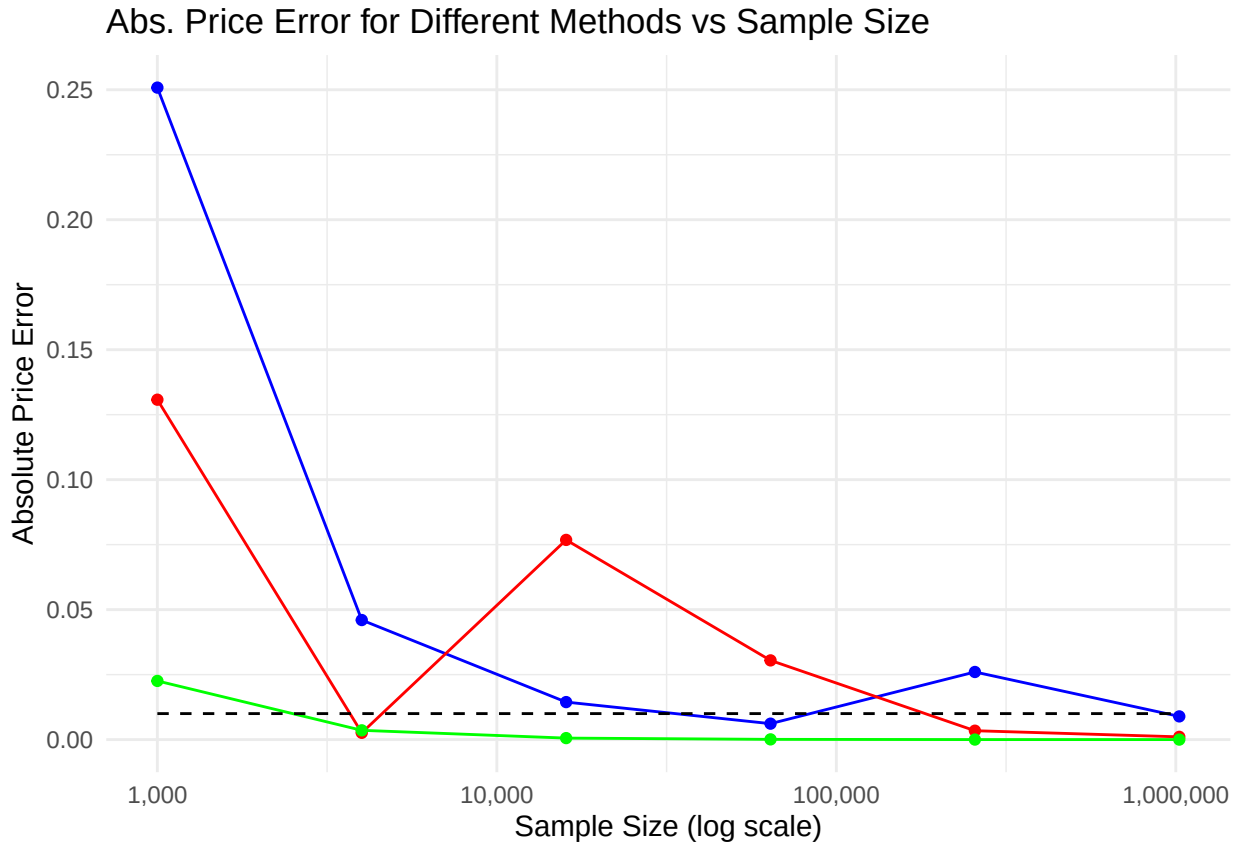
```
library(ggplot2)

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()
```



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, followed by the antithetic approach, and then the standard approach. This is also highlighted in the results section where the Quasi-Monte-Carlo method had on average the fastest convergence in sample size to a 1 cent error in option price. The below graph shows the absolute price error of the three methods, giving more insight to the convergence. 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()
```

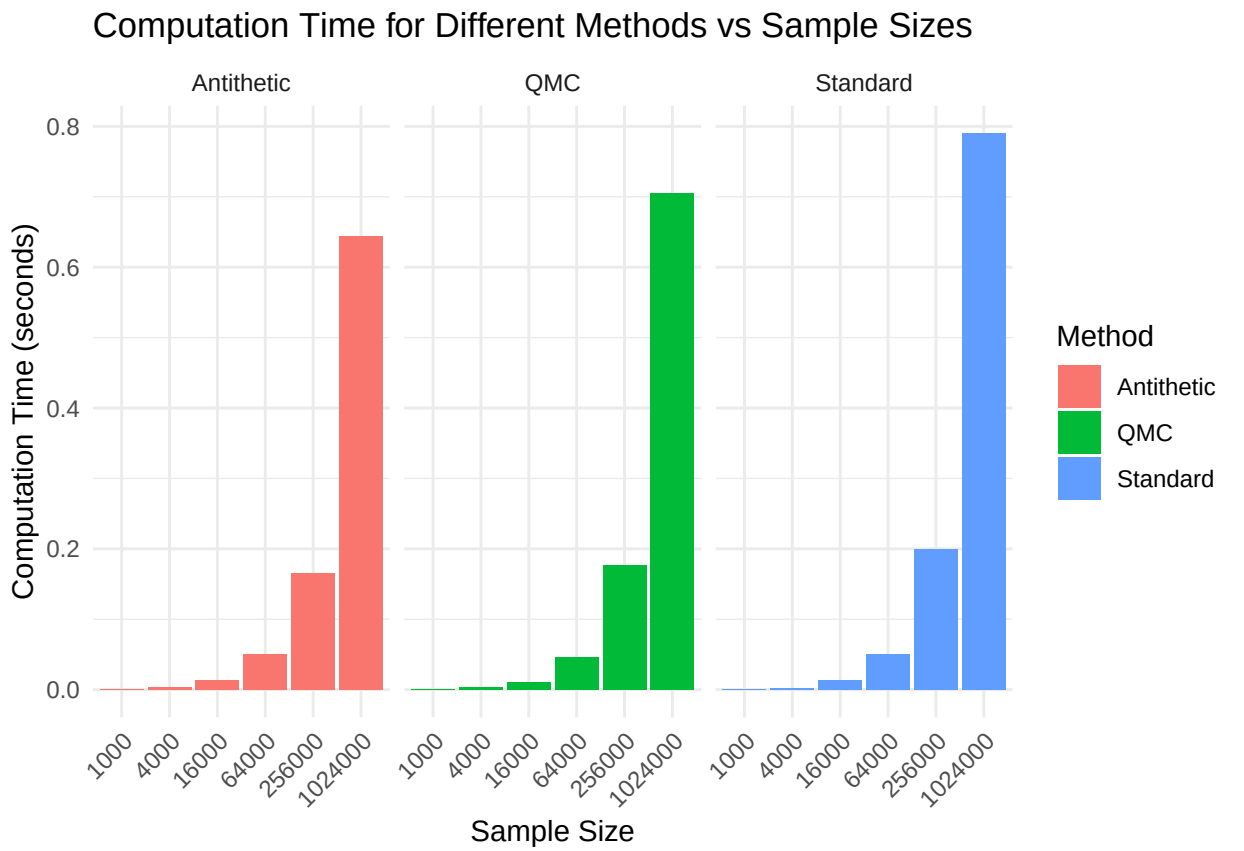


The below graph shows the comparison between computation time between the three methods.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

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

df_antithetic <- results_antithetic %>%
  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
```



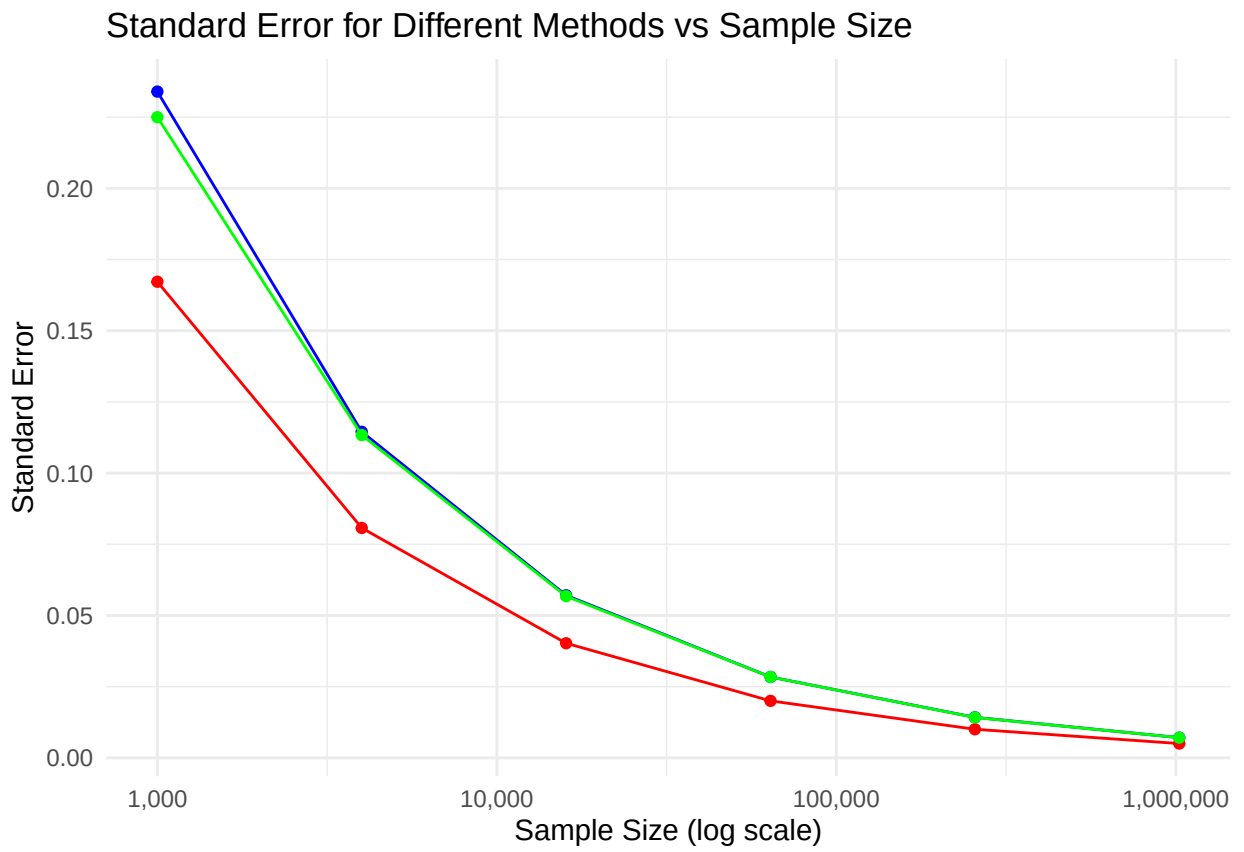
In this case, the Antithetic approach was the fastest computationally for running n independent samples mainly because the antithetic approach generates n/2 random independent samples for the standard Monte Carlo n. This is due to the fact that the negative "antithetic" counterpart of the n/2 samples 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. This can be seen by the fact that it takes roughly 0.0030 seconds for the QMC method to have an accuracy within 1 cent and 0.04 seconds for the antithetic approach to have an accuracy within 1 cent. The standard Monte Carlo approach took roughly 0.1881 seconds to have an accuracy of 0.01 cents. The computational time values are taken from the example outputs in sections 2.1, 2.2, and 2.3, with the graph showing another iteration. Because Monte Carlo Simulation is used, these values can vary. Note that the standard Monte Carlo approach is the slowest method and also converges to the exact put option price the slowest. This will not be the preferred method if the antithetic or QMC approaches can 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 as shown by equation 13.

$$SE \propto \frac{1}{\sqrt{n}} \tag{13}$$

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") +
```

```r
# 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()
```



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.