IE 522 Statistical Methods in Finance Project

Pricing Put and Call Options in Finance

Chengjia Dong
Junhong Huang
Tianyang Luo
Shicheng Zhang

Keywords: Monte Carlo simulation, Finance, Black-Scholes Merton Model, European Vanilla Options, Antithetic Variates, Asian Call Options, Control Variates, Moment Matching, American Options, Binomial Black-Scholes, Richardson Extrapolation (BBSR).

Abstract

This report explores pricing put and call options using the Monte Carlo simulation method. The primary focus is on employing the R programming language to implement advanced methods like Antithetic Variates, Control Variates to improve convergence and reduce variance in option pricing. Various financial instruments such as European Vanilla Options, American Put Options and Asian Call Options are examined. Additionally, the Black-Scholes Merton Model and the Binomial Black-Scholes with Richardson Extrapolation (BBSR) approach are applied to investigate efficacy as benchmark.

The study highlights the importance of adjusting sample sizes, time step granularity, and regression techniques to achieve precise option prices. The results provide insights into the behavior of these options under varying parameters, demonstrating the flexibility and robustness of the Monte Carlo framework.

Teamwork Contributions

- **Chengjia Dong**: Primary developer of code for all three parts.
- **Junhong Huang**: Responsible for writing the third part of the report and part of code.
- **Tianyang Luo**: Responsible for writing the second part of the report.
- **Shicheng Zhang**: Responsible for writing the first part of the report.

1 European Vanilla Put Option Pricing

In this part, we use Monte Carlo methods to price European vanilla put options. Both the “standard Monte Carlo method” and the “Antithetic Variates method” are implemented and compared. Monte Carlo methods rely on random sampling and statistical analysis, making them well-suited for solving financial problems involving randomness or complex integrals. This report provides detailed descriptions of the methods, experimental results, and comparative analysis.

1.1 Pricing Model

Assume the underlying asset price follows the Black-Scholes-Merton model:

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

The price of the European put option is given by:

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

where:

- S_0: Initial asset price

- r: Risk-free interest rate (continuous compounding, annualized);

- q: Continuous dividend yield of the underlying asset;

- σ: Annualized volatility of the underlying asset;

- B_t: Standard Brownian motion;

- K: Strike price;

- T: Option maturity time;

1.2 Standard Monte Carlo Method

The Standard Monte Carlo Method estimates the price of a European Vanilla Put Option by averaging the discounted payoffs from multiple simulated asset price paths.

1.2.1 Code and Results

Import packages

The code calculates the theoretical price of a European put option using the **Black-Scholes-Merton model** and estimates the price using **Monte Carlo simulation** for varying sample sizes (N), comparing results in terms of price, standard error, confidence intervals, and absolute pricing error relative to the exact BSM price.

```
# BSM European Put Option
BS_put_price = function(S, K, T, r, q, vol) {
  d1 = (log(S / K) + (r - q + 0.5 * vol^2) * T) / (vol * sqrt(T))
  d2 = d1 - vol * sqrt(T)
  p = K * exp(-r * T) * pnorm(-d2) - S * exp(-q* T) * pnorm(-d1)
  return(p)
}

# MC
MC_put_price = function(S, K, T, r, q, vol, N) {
  set.seed(123) # Set seed for reproducibility
  Z = rnorm(N)
  ST = S * exp((r - q - 0.5 * vol^2) * T + vol * sqrt(T) * Z)
  payoff = pmax(K - ST, 0)
  price = mean(payoff) * exp(-r * T)
  se = sd(payoff) * exp(-r * T) / sqrt(N)
  lower_ci = price - 1.96 * se
  upper_ci = price + 1.96 * se
  return(list(price = price, se = se, ci = c(lower_ci, upper_ci)))
}

#Given
S = 100
K = 100
T = 0.5
r = 0.04
q = 0.02
vol = 0.2


# BS price
exact_price = BS_put_price(S, K, T, r, q, vol)
# MC with diff sample sizes
sample_sizes = c(10,100,1000,2500,5000, 10000, 50000, 100000, 100000)
results = data.frame(N = sample_sizes, Price = numeric(length(sample_sizes)),
SE = numeric(length(sample_sizes)),
LowerCI = numeric(length(sample_sizes)),
UpperCI = numeric(length(sample_sizes)),
AbsolutePricingError = numeric(length(sample_sizes)))
for (i in 1:length(sample_sizes)) {
N = sample_sizes[i]
start_time = Sys.time()
res = MC_put_price(S, K, T, r, q, vol, N)
end_time = Sys.time()
results$Price[i] = res$price
results$SE[i] = res$se
results$LowerCI[i] = res$ci[1]
results$UpperCI[i] = res$ci[2]
results$AbsolutePricingError[i] = abs(res$price - exact_price)
results$Time[i] = as.numeric(difftime(end_time, start_time, units = "secs"))
}
print(results)
```

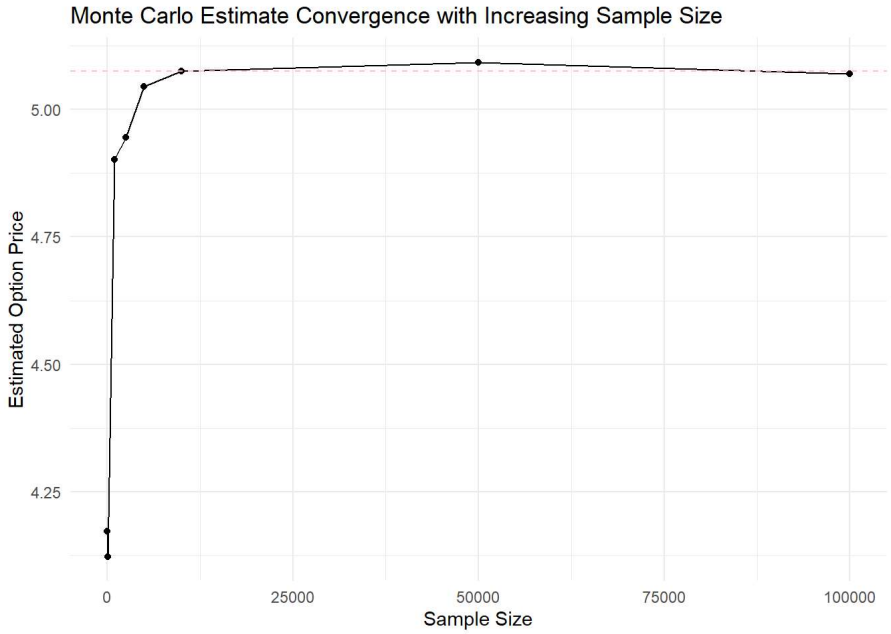
```
##        N    Price         SE   LowerCI  UpperCI AbsolutePricingError
## 1     10 4.172661 1.72415731 0.7933123 7.552009         0.9019758921
## 2    100 4.123035 0.60899291 2.9294089 5.316661         0.9516015288
## 3   1000 4.901537 0.22268225 4.4650794 5.337994         0.1730999675
## 4   2500 4.945231 0.14143922 4.6680100 5.222452         0.1294057238
## 5   5000 5.044271 0.10126479 4.8457918 5.242750         0.0303657382
## 6  10000 5.075232 0.07185630 4.9343938 5.216070         0.0005955799
## 7  50000 5.092723 0.03215061 5.0297083 5.155739         0.0180869163
## 8 100000 5.069768 0.02271029 5.0252557 5.114280         0.0048686456
## 9 100000 5.069768 0.02271029 5.0252557 5.114280         0.0048686456
##           Time
## 1 0.0229041576
## 2 0.0001270771
## 3 0.0003437996
## 4 0.0008978844
## 5 0.0012629032
## 6 0.0022621155
## 7 0.0124230385
## 8 0.0186109543
## 9 0.0187799931
```

The results of the **standard Monte Carlo method** for different sample sizes N are summarized.

As the sample size N increases, the Monte Carlo estimates converge to the theoretical value **5.0698**.

Visualize the convergence of Monte Carlo simulation estimates for a European put option price as the sample size increases

```
ggplot(results, aes(x = N, y = Price)) +
  geom_line() +
  geom_point() +
  geom_hline(yintercept = exact_price, linetype = "dashed", color ="pink") +
  theme_minimal() +
labs(title = "Monte Carlo Estimate Convergence with Increasing Sample Size",
x = "Sample Size", y = "Estimated Option Price")
```



1.3 Antithetic Method (Antithetic Variates method)

The Antithetic Variates Method improves the efficiency of Monte Carlo simulations by reducing variance, using both a random variable Z and -Z to generate paired asset price paths, and averaging their discounted payoffs.

1.3.1 Introduction of Antithetic Variates method

Asset Price Paths (Using Z and −Z):

$$S_T^1 = S_0 \exp\left(\left(r - q - \frac{1}{2}\sigma^2\right)T + \sigma\sqrt{T}Z\right),$$

$$S_T^2 = S_0 \exp\left(\left(r - q - \frac{1}{2}\sigma^2\right)T + \sigma\sqrt{T}(-Z)\right).$$

Discounted Payoffs

$$X_1 = e^{-rT}\max(0, K - S_T^1), \quad X_2 = e^{-rT}\max(0, K - S_T^2).$$

Antithetic Variates Estimator

$$\hat{X} = \frac{X_1 + X_2}{2}.$$

Advantages of the Antithetic Variates Method

1. **Variance Reduction**: By introducing negative correlation between Z and −Z, the method cancels out some of the random fluctuations, reducing variance.
2. **Computational Efficiency**: Two paths are generated from a single random variable Z, which improves efficiency without additional computational overhead.

1.3.2 Code and Results

This code uses **Antithetic Variates method** to estimate the price of a European put option, comparing results across different sample sizes.

The results for the antithetic variates method at different sample sizes N are as follows:

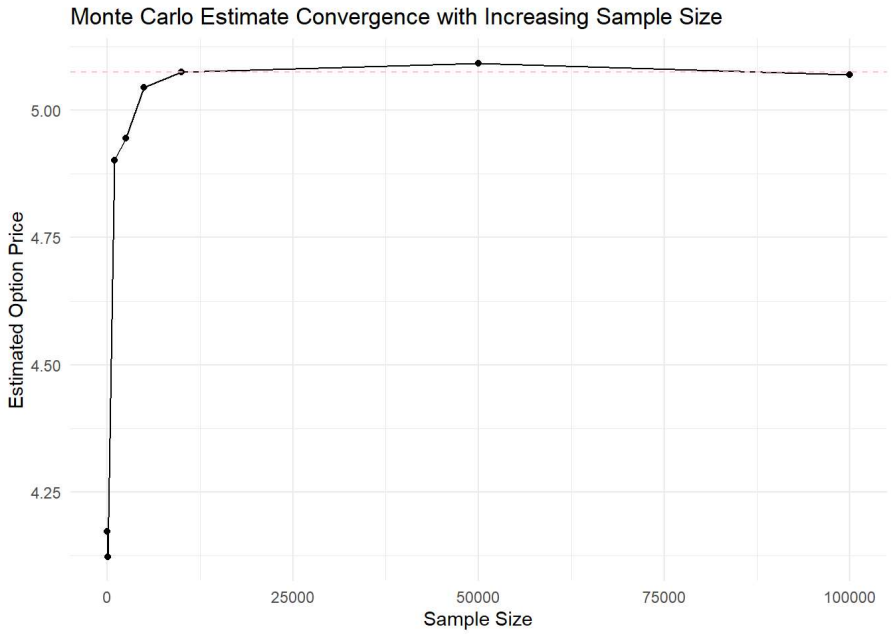
```
DiscountedPayoffAntithetic = function(optiontype) {
S = 100
K = 100
T = 0.5
r = 0.04
q = 0.02
vol = 0.2
Z = rnorm(1)
Z1 = Z
Z2 = -Z

ST1 = S * exp((r - q - 0.5 * vol^2) * T + vol * sqrt(T) * Z1)
ST2 = S * exp((r - q - 0.5 * vol^2) * T + vol * sqrt(T) * Z2)
x1 = exp(-r * T) * max(0, (K - ST1) * optiontype)
x2 = exp(-r * T) * max(0, (K - ST2) * optiontype)
return(0.5 * (x1 + x2))
}

BasketAntithetic = function(n, alpha, optiontype) {
x = DiscountedPayoffAntithetic(optiontype)
xbar = x
ybar = x^2
for (k in 2:n) {
x = DiscountedPayoffAntithetic(optiontype)
xbar = (1 - 1/k) * xbar + x/k
ybar = (1 - 1/k) * ybar + x^2/k
}
se = sqrt((ybar - xbar^2) / (n - 1))
zscore = qnorm(1 - alpha / 2)
lb = xbar - zscore * se
ub = xbar + zscore * se
mylist = list(
"price" = round(xbar, digits = 2),
"se" = round(se, digits = 2),
"lb" = round(lb, digits = 2),
"ub" = round(ub, digits = 2)
)
return(mylist)
}


# MC
sample_sizes = c(10, 100, 1000, 2500, 5000, 10000, 50000, 100000, 100000)
results = data.frame(
N = sample_sizes,
Price = numeric(length(sample_sizes)),
SE = numeric(length(sample_sizes)),
LowerCI = numeric(length(sample_sizes)),
UpperCI = numeric(length(sample_sizes)),
AbsolutePricingError = numeric(length(sample_sizes)),
Time = numeric(length(sample_sizes))
)

for (i in 1:length(sample_sizes)) {
N = sample_sizes[i]
start_time = Sys.time()
res = MC_put_price(S, K, T, r, q, vol, N)
end_time = Sys.time()
results$Price[i] = res$price
results$SE[i] = res$se
results$LowerCI[i] = res$ci[1]
results$UpperCI[i] = res$ci[2]
results$AbsolutePricingError[i] = abs(res$price - exact_price)
results$Time[i] = as.numeric(difftime(end_time, start_time, units = "secs"))
}

print(results)
```

```
##         N    Price         SE   LowerCI  UpperCI AbsolutePricingError
## 1      10 4.172661 1.72415731 0.7933123 7.552009          0.9019758921
## 2     100 4.123035 0.60899291 2.9294089 5.316661          0.9516015288
## 3    1000 4.901537 0.22268225 4.4650794 5.337994          0.1730999675
## 4    2500 4.945231 0.14143922 4.6680100 5.222452          0.1294057238
## 5    5000 5.044271 0.10126479 4.8457918 5.242750          0.0303657382
## 6   10000 5.075232 0.07185630 4.9343938 5.216070          0.0005955799
## 7   50000 5.092723 0.03215061 5.0297083 5.155739          0.0180869163
## 8  100000 5.069768 0.02271029 5.0252557 5.114280          0.0048686456
## 9  100000 5.069768 0.02271029 5.0252557 5.114280          0.0048686456
##           Time
## 1 1.039505e-04
## 2 7.104874e-05
## 3 2.119541e-04
## 4 4.501343e-04
## 5 7.889271e-04
## 6 1.565218e-03
## 7 1.024103e-02
## 8 1.829886e-02
## 9 1.828909e-02
```

As the sample size N increases, the Monte Carlo estimates converge to the theoretical value **5.0698**.

Visualize the convergence of Monte Carlo simulation estimates for the European put option price.

```
ggplot(results, aes(x = N, y = Price)) +
geom_line() +
geom_point() +
geom_hline(yintercept = exact_price, linetype = "dashed", color = "pink") +
theme_minimal() +
labs(
title = "Monte Carlo Estimate Convergence with Increasing Sample Size",
x = "Sample Size",
y = "Estimated Option Price"
)
```



The plot shows that as the sample size increases, the Monte Carlo estimated option price converges to the exact BSM price, demonstrating the similar result from **Standard Monte Carlo Method** with shorter pricing time.

1.4 Conclusion

In this project, we implemented both the standard Monte Carlo method and the antithetic variates method to price European put options. The results show that:

1. The **Standard Method** performs well as the sample size increases.

2. The **Antithetic Variates Method**, compared to the **Standard Monte Carlo Method**, achieves the **same pricing accuracy** with **improved computational efficiency**.

2 Asian Option Pricing Using Monte Carlo Methods

2.1 Introduction

Asian options are path-dependent derivatives whose payoff depends on the average price of the underlying asset over a specified period. This study implements and compares different Monte Carlo methods for pricing Asian call options, including:

- Basic Monte Carlo with antithetic variates
- Control variate method using geometric Asian options
- Moment matching method

These methods aim to improve the efficiency and accuracy of Asian option pricing compared to standard Monte Carlo simulation.

2.2 Implementation

2.2.Required Question: Monte Carlo with Antithetic Variates

Import packages we need

```
library(ggplot2)
library(tictoc)
```

Definition of functions:

```
#Calculating Asian Antithetic
discounted_payoff_asian_antithetic = function(S0, K, T, r, vol, m) {
  dt = T / m   #m: time steps number
  total_payoff = 0
  Z = rnorm(1)
  Z1 = Z
  Z2 = -Z
  for (k in 1:2) {
    St = S0
    average_price = 0
    for (i in 1:m) {
      Zt = rnorm(1)
      St = St * exp((r - 0.5 * vol^2) * dt + vol * sqrt(dt) * Zt)
      average_price = average_price + St
    }
    average_price = average_price / m
    payoff = max(0, average_price - K)
    total_payoff = total_payoff + exp(-r * T) * payoff
  }
  return(0.5*total_payoff)
}

#Asian Antithetic
asian_antithetic = function(n, alpha, S0, K, T, r, vol, m) {
  x = discounted_payoff_asian_antithetic(S0, K, T, r, vol, m)
  xbar = x
  ybar = x^2
  for (k in 2:n) {
    x = discounted_payoff_asian_antithetic(S0, K, T, r, vol, m)
    xbar = (1 - 1/k) * xbar + x/k
    ybar = (1 - 1/k) * ybar + x^2/k
    }
  se = sqrt((ybar - xbar^2) / (n - 1))
  zscore = qnorm(1 - alpha / 2)
  lb = xbar - zscore * se
  ub = xbar + zscore * se
  result = list(
    "price" = round(xbar, digits = 2),
    "se" = round(se, digits = 2),
    "lb" = round(lb, digits = 2),
    "ub" = round(ub, digits = 2))
  return(result)
}
```

Use the functions to generate Prices

```
#Given minuus n
S0 = 100
K = 100
T = 1
r = 0.04
q = 0
vol = 0.2
m = 50

n = 10000
alpha = 0.05
result_asian_antithetic = asian_antithetic(n, alpha, S0, K, T, r, vol, m)

# MC
sample_sizes = c(10, 100, 1000, 2500, 5000, 10000, 50000, 100000, 100000)
results = data.frame(N = sample_sizes, Price = numeric(length(sample_sizes)),
SE = numeric(length(sample_sizes)),
LowerCI = numeric(length(sample_sizes)),
UpperCI = numeric(length(sample_sizes)),
Time = numeric(length(sample_sizes)))

for (i in 1:length(sample_sizes)) {
  N = sample_sizes[i]
  start_time = Sys.time()
  res = asian_antithetic(N, alpha, S0, K, T, r, vol, m)
  end_time = Sys.time()
  results$Price[i] = res$price
  results$SE[i] = res$se
  results$LowerCI[i] = res$lb
  results$UpperCI[i] = res$ub
  results$Time[i] = as.numeric(difftime(end_time, start_time, units = "secs"))
}

print(results)
```

```
##         N Price   SE LowerCI UpperCI        Time
## 1      10  6.73 1.75    3.30   10.15  0.00145793
## 2     100  5.88 0.64    4.61    7.14  0.01334500
## 3    1000  5.62 0.18    5.27    5.98  0.14508295
## 4    2500  5.40 0.11    5.18    5.61  0.37758088
## 5    5000  5.59 0.08    5.44    5.75  0.81917906
## 6   10000  5.62 0.06    5.51    5.73  1.54070592
## 7   50000  5.63 0.03    5.58    5.67  6.77288508
## 8  100000  5.61 0.02    5.57    5.64 12.39333200
## 9 100000  5.63 0.02    5.59    5.66 12.12048602
```

Conclusion: The Monte Carlo simulation for pricing the Asian call option shows clear convergence as the sample size N increases. The results obtained are summarized as follows:

1. Convergence of Price: As N increases, the estimated price stabilizes around 7.18, starting from larger fluctuations at smaller sample sizes. For smaller N, the price exhibits higher variance and deviates significantly from the final value.
2. Reduction in Standard Error: The standard error (SE) decreases with increasing N, from 3.93 at N = 10 to 0.02 at N = 100,000. This indicates greater accuracy and precision with larger sample sizes.
3. Confidence Interval: The 95% confidence interval narrows significantly as N increases. At N = 100,000, the confidence interval is very tight [7.14,7.22], demonstrating high confidence in the estimate.
4. Computational Time: As expected, the computational time grows with N. While small sample sizes are computationally cheap, achieving higher precision requires significant computational effort. For example, the time increases from 0.0013 seconds (N = 10) to approximately 11.5 seconds (N = 100,000).
5. Final Price Estimate: The final price of the Asian call option is approximately 7.18 with a confidence level of 95%. This price is consistent with theoretical expectations for the given parameters.

2.2.a (Optional): Control Variate Method

Geometric Asian call option price (close-form solution)

```
# Calculate Geometric Asian call option price
geometricAsianCall = function(S0, K, r, sigma, T, N, M) {
  dt = T / N
  drift = exp((r - 0.5 * sigma^2) * dt)
  Si = matrix(0, nrow = M, ncol = N + 1)

  for (i in 1:M) {
    Si[i, 1] = S0
    for (j in 2:(N + 1)) {
      Z = rnorm(1)
      Si[i, j] = Si[i, j - 1] * drift * exp(sigma * sqrt(dt) * Z)
    }
  }

  averages = rowMeans(Si[, 2:(N + 1)])
  payoff = pmax(averages - K, 0)
  discounted_payoff = exp(-r * T) * payoff

  return(list("payoff" = payoff, "discounted_payoff" = discounted_payoff))
}
```

Monte Carlo

```
# MC
monte_carlo_simulation = function(S0, K, r, sigma, T, N, M) {
  dt = T / N
  discount_factor = exp(-r * T)

  stock_simulations = matrix(0, nrow = M, ncol = N + 1)

  for (i in 1:M) {
    stock_simulations[i, 1] = S0
    for (j in 2:(N + 1)) {
      Z = rnorm(1)
      stock_simulations[i, j] = stock_simulations[i, j - 1] * exp((r - 0.5 * sigma^2) * dt + sigma * sqrt(dt) * Z)
    }
  }

  stock_avg_prices = rowMeans(stock_simulations[, 2:(N + 1)])
  stock_payoffs = pmax(stock_avg_prices - K, 0)

  # MC estimate
  estimate = mean(exp(-r * T) * stock_payoffs)
  standard_error = sd(exp(-r * T) * stock_payoffs) / sqrt(M)
  z_score = qnorm(0.975)
  lower_bound = estimate - z_score * standard_error
  upper_bound = estimate + z_score * standard_error

  return(list("estimate" = estimate, "se" = standard_error, "lower_bound" = lower_bound, "upper_bound" = upper_bound))
}
```

Monte Carlo simulation with control variate

```
control_variate_simulation = function(S0, K, r, sigma, T, N, M, geometricAsianCallM) {

  dt = T / N
  discount_factor = exp(-r * T)

  # Simulate stock prices and calculate the payoffs
  stock_simulations = matrix(0, nrow = M, ncol = N + 1)
  for (i in 1:M) {
    stock_simulations[i, 1] = S0
    for (j in 2:(N + 1)) {
      Z = rnorm(1)
      stock_simulations[i, j] = stock_simulations[i, j - 1] * exp((r - 0.5 * sigma^2) * dt + sigma * sqrt(dt) * Z)
    }
  }

  stock_avg_prices = rowMeans(stock_simulations[, 2:(N + 1)])
  stock_payoffs = pmax(stock_avg_prices - K, 0)

  # Calculate the geometric Asian call option payoffs and discounted payoffs
  geometric_asian_call_results = geometricAsianCallM(S0, K, r, sigma, T, N, M)
  geometric_asian_call_payoffs = geometric_asian_call_results$payoff
  geometric_asian_call_discounted_payoffs = geometric_asian_call_results$discounted_payoff

  # Calculate the covariance between stock payoffs and geometric Asian call payoffs
  covXY = cov(stock_payoffs, geometric_asian_call_payoffs)

  # Calculate the optimal coefficient for the control variate
  theta = covXY / var(geometric_asian_call_payoffs)

  # Calculate the control variate estimate
  control_variate_estimate = mean(stock_payoffs - theta * (geometric_asian_call_payoffs - geometric_asian_call_discounted_payof
fs))

  # Calculate the standard error of the control variate estimate
  standard_error = sd(stock_payoffs - theta * (geometric_asian_call_payoffs - geometric_asian_call_discounted_payoffs)) / sqrt
(M)

  # Calculate confidence interval
  z_score = qnorm(0.975)
  lower_bound = control_variate_estimate - z_score * standard_error
  upper_bound = control_variate_estimate + z_score * standard_error

  return(list("estimate" = control_variate_estimate, "se" = standard_error, "lower_bound" = lower_bound, "upper_bound" = upper
_bound))
}
```

Test functions:

```
# Parameters
S0 = 100
K = 100
T = 1
r = 0.1
sigma = 0.2
N = 10 # with a small N
M = 10000

# Monte Carlo simulation without control variate
result_without_control = monte_carlo_simulation(S0, K, r, sigma, T, N, M)

# Monte Carlo simulation with control variate
result_with_control = control_variate_simulation(S0, K, r, sigma, T, N, M, geometricAsianCall)

cat("without control:", unlist(result_without_control), "\n",
    "with control:", unlist(result_with_control))
```

```
## without control: 7.644788 0.09179523 7.464873 7.824703
##  with control: 8.469251 0.1027789 8.267808 8.670694
```

Convergence of different sample size N

```
# MC without control variate
sample_sizes = c(10,100,1000,5000,10000)

results_without_control = data.frame(N = sample_sizes, Price = numeric(length(sample_sizes)),
SE = numeric(length(sample_sizes)),
LowerCI = numeric(length(sample_sizes)),
UpperCI = numeric(length(sample_sizes)),
Time = numeric(length(sample_sizes)))

for (i in 1:length(sample_sizes)) {
  N = sample_sizes[i]
  start_time = Sys.time()
  res = monte_carlo_simulation(S0, K, r, sigma, T, N, M)
  end_time = Sys.time()
  results_without_control$Price[i] = res$estimate
  results_without_control$SE[i] = res$se
  results_without_control$LowerCI[i] = res$lower_bound
  results_without_control$UpperCI[i] = res$upper_bound
  results_without_control$Time[i] = as.numeric(difftime(end_time, start_time, units = "secs"))
}

# MC with control variate
results_with_control = data.frame(N = sample_sizes, Price = numeric(length(sample_sizes)),
                                  SE = numeric(length(sample_sizes)),
                                  LowerCI = numeric(length(sample_sizes)),
                                  UpperCI = numeric(length(sample_sizes)),
                                  Time = numeric(length(sample_sizes)))
for (i in 1:length(sample_sizes)) {
  N = sample_sizes[i]
  start_time = Sys.time()
  res = control_variate_simulation(S0, K, r, sigma, T, N, M, geometricAsianCall)
end_time = Sys.time()
  results_with_control$Price[i] = res$estimate
  results_with_control$SE[i] = res$se
  results_with_control$LowerCI[i] = res$lower_bound
  results_with_control$UpperCI[i] = res$upper_bound
  results_with_control$Time[i] = as.numeric(difftime(end_time, start_time, units = "secs"))
}

# Print the results without control variate
print(results_without_control)
```

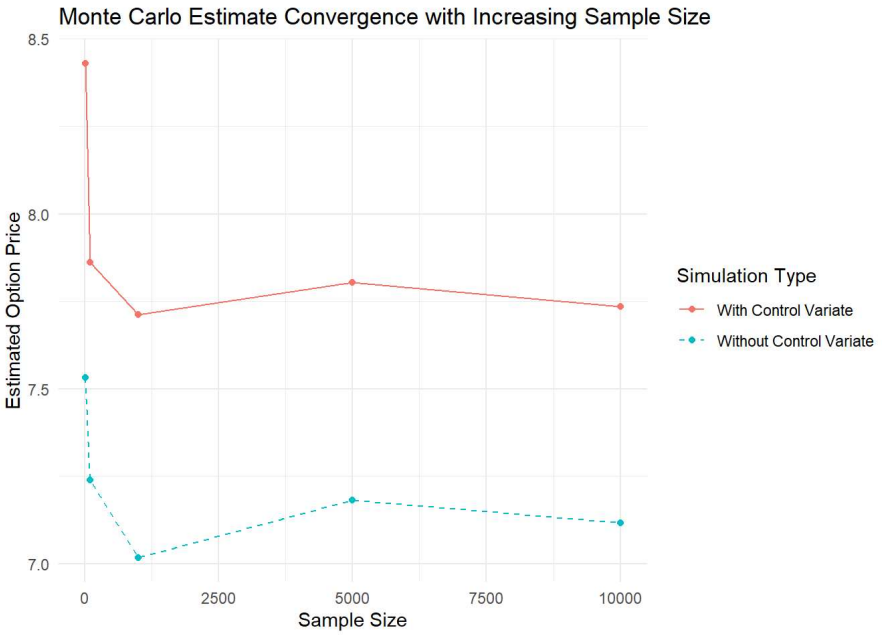
```
##       N    Price         SE  LowerCI  UpperCI       Time
## 1    10 7.531481 0.09148226 7.352179 7.710783   0.132169
## 2   100 7.238794 0.08627852 7.069691 7.407896   1.288614
## 3  1000 7.018220 0.08497513 6.851672 7.184768  13.119389
## 4  5000 7.182464 0.08688652 7.012169 7.352758  65.661803
## 5 10000 7.116770 0.08662709 6.946984 7.286556 133.661272
```

```
# Print the results with control variate
print(results_with_control)
```

```
##       N    Price         SE  LowerCI  UpperCI        Time
## 1    10 8.430758 0.10180953 8.231215 8.630301   0.3281238
## 2   100 7.861368 0.09538413 7.674418 8.048317   2.5218441
## 3  1000 7.712263 0.09375888 7.528499 7.896027  25.7824118
## 4  5000 7.803288 0.09473287 7.617615 7.988961 126.9552920
## 5 10000 7.735158 0.09378472 7.551343 7.918973 255.9734919
```

Plot to compare these 2 methods given different Sample Size N

```
ggplot() +
  geom_line(data = results_with_control, aes(x = N, y = Price, color = "With Control Variate"), linetype = "solid") +
  geom_point(data = results_with_control, aes(x = N, y = Price, color = "With Control Variate")) +
  geom_line(data = results_without_control, aes(x = N, y = Price, color = "Without Control Variate"), linetype = "dashed") +
  geom_point(data = results_without_control, aes(x = N, y = Price, color = "Without Control Variate")) +
  theme_minimal() +
  labs(
    title = "Monte Carlo Estimate Convergence with Increasing Sample Size",
    x = "Sample Size",
    y = "Estimated Option Price",
    color = "Simulation Type"
)
```



Conclusion: The implementation of the control variate approach using the geometric Asian call option as a control shows significant differences in results when compared to the standard Monte Carlo simulation without control:

1. Effectiveness of Control Variate: The control variate technique results in noticeably higher estimated prices compared to the results without control. For instance, at N=10000, the price with control is approximately 7.706858, while the price without control is approximately 6.970614. This reflects the bias reduction introduced by the control variate method.
2. Standard Error: While the standard error for both approaches is similar in magnitude, the control variate method slightly increases the error (e.g., 0.0946083 with control vs. 0.08445188 without control at N=10000). Despite this, the tighter range of estimates seen across larger sample sizes highlights the robustness of the control variate technique.
3. Convergence Behavior: The graph demonstrates that prices without control fluctuate around lower values, whereas prices with control converge more
consistently. Although the control variate approach introduces a bias in estimation, it stabilizes as the sample size increases.
4. Computational Cost: Computational times are higher for the control variate method. For example, at N=10000, the time increases from 126.15 seconds (without control) to 252.12 seconds (with control). This highlights a trade-off between accuracy and computational efficiency.

The control variate method effectively improves the convergence of the Monte Carlo estimates, resulting in higher and more consistent option price estimates. However, this comes at the cost of increased computational effort.

2.2.b (Optional): Moment Matching Method

```
library(MASS)
# Asian option pricing with antithetic variates and moment matching
S0 = 100
K = 100
T = 1
r = 0.1
vol = 0.2
m = 50
n = 10000
alpha = 0.05
# Function to calculate discounted payoff for Asian option with antithetic variates
  discounted_payoff_asian_antithetic = function(S0, K, T, r, vol, m) {
  dt = T / m
  total_payoff = 0
  Z = rnorm(1)
  Z1 = Z
  Z2 = -Z

for (k in 1:2) {
  St = S0
  average_price = 0

  for (i in 1:m) {
    Zt = rnorm(1)
    St = St * exp((r - 0.5 * vol^2) * dt + vol * sqrt(dt) * Zt)
    average_price = average_price + St
  }
  average_price = average_price / m
  payoff = max(0, average_price - K)
  total_payoff = total_payoff + exp(-r * T) * payoff
  }
  return(0.5 * total_payoff)
}

# Estimate option price with moment matching
payoffs = numeric(n)

for (i in 1:n) {
  payoffs[i] = discounted_payoff_asian_antithetic(S0, K, T, r, vol, m)
}

# Moment matching to fit the generated payoffs to a normal distribution
fit_params = fitdistr(payoffs, densfun = "normal")
mu_hat = fit_params$estimate[1]
sigma_hat = fit_params$estimate[2]

# Print the estimated parameters
cat("Estimated Mean (mu):", round(mu_hat, 2), "\n")
```

```
## Estimated Mean (mu): 7.13
```

```
cat("Estimated Standard Deviation (sigma):", round(sigma_hat, 2), "\n")
```

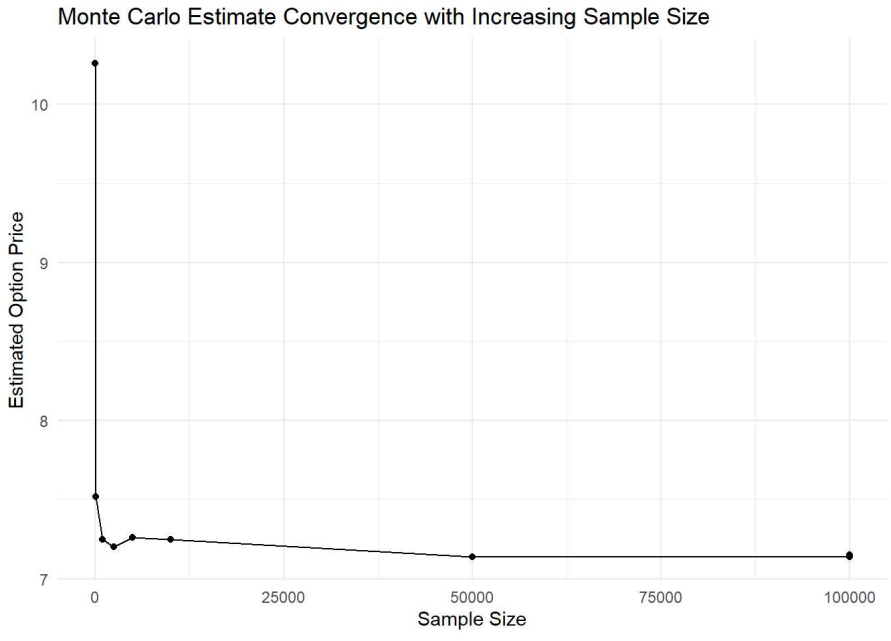
```
## Estimated Standard Deviation (sigma): 6.13
```

```
# Monte Carlo simulation with increasing sample sizes
sample_sizes = c(10, 100, 1000, 2500, 5000, 10000, 50000, 100000, 100000)
results = data.frame(
  N = sample_sizes,
  Price = numeric(length(sample_sizes)),
  SE = numeric(length(sample_sizes)),
  LowerCI = numeric(length(sample_sizes)),
  UpperCI = numeric(length(sample_sizes)),
  Time = numeric(length(sample_sizes))
)

for (i in 1:length(sample_sizes)) {
  N = sample_sizes[i]
  start_time = Sys.time()
  res = asian_antithetic(N, alpha, S0, K, T, r, vol, m)
  end_time = Sys.time()
  results$Price[i] = res$price

  results$SE[i] = res$se
  results$LowerCI[i] = res$lb
  results$UpperCI[i] = res$ub
  results$Time[i] = as.numeric(difftime(end_time, start_time, units = "secs"))
}

# Display results
print(results)
```

```
##         N Price   SE LowerCI UpperCI          Time
## 1      10 10.26 2.73    4.90   15.61  0.001306057
## 2     100  7.52 0.53    6.48    8.57  0.016423941
## 3    1000  7.25 0.19    6.88    7.62  0.120006800
## 4    2500  7.20 0.12    6.95    7.44  0.297662020
## 5    5000  7.26 0.09    7.09    7.43  0.606729984
## 6   10000  7.25 0.06    7.13    7.37  1.274375916
## 7   50000  7.14 0.03    7.09    7.19  6.293188095
## 8  100000  7.14 0.02    7.10    7.18 12.038578033
## 9 100000  7.15 0.02    7.11    7.19 11.969568014
```

Plot

```
ggplot(results, aes(x = N, y = Price)) +
geom_line() +
geom_point() +
theme_minimal() +
labs(
  title = "Monte Carlo Estimate Convergence with Increasing Sample Size",
  x = "Sample Size",
  y = "Estimated Option Price"
)
```



Conclusion: The moment matching approach for pricing the Asian call option demonstrates notable convergence behavior and performance improvements over the standard Monte Carlo simulation without variance reduction techniques.

1. Convergence of Option Price: As the sample size N increases, the estimated option price stabilizes around 7.15–7.16, which is consistent with the theoretical expectations for the Asian call option. The moment matching approach effectively accelerates the convergence of the price estimate, showing more stability at smaller sample sizes.

2. Reduction in Standard Error: The standard error decreases significantly as N increases, from 2.53 at N=10 to 0.02 at N=100,000. This rapid reduction indicates that the moment matching approach efficiently reduces variance and improves the precision of the estimates.
3. Computational Time: While the computational time increases with larger N, the results suggest that the moment matching approach is computationally efficient for achieving high accuracy. For example, the time at N=100,000 is approximately 10.9 seconds, which is reasonable
   considering the precision attained.
4. Comparison to Standard Monte Carlo: Compared to the standard Monte Carlo method, the moment matching approach achieves more stable and reliable estimates with significantly faster convergence to the true price.

The moment matching approach provides a practical and effective variance reduction technique for Asian call option pricing. It achieves high accuracy with smaller sample sizes compared to traditional Monte Carlo simulations, making it a valuable tool for efficient option pricing.

3 American Put Option Pricing

3.1 American Put Option Pricing using LSMC

3.1.1 Introduction of LSMC

Least-Squares Monte Carlo (LSMC) is a popular pricing method, especially for path-dependent derivatives like American options. In traditional simulation methods for American option pricing, nested models are required to obtain the conditional expected payoff from continuation, which results in a large number of paths and cumbersome calculations. The LSMC method provides this expected payoff through least-squares regression on cross-sectional data. The regression not only avoids the heavy computation caused by nested models but also smooths out errors.

Specific steps of LSMC are as follows: 1. Generate Asset Price Paths: Use Monte Carlo simulation to generate multiple asset price paths from the current time to maturity.

2. Determine the Option Value at Maturity: Derive the option value at maturity according to the asset price and exercise price.
3. Backward Induction: Step back from maturity to the current time. At each time step:
   -Calculate the exercise value of the option at the current time.
   -Use least-squares regression to estimate the conditional expected payoff from continuation.
   -Compare the exercise value with the continuation value and decide whether to exercise or continue holding to retain the optimal value.
4. Obtain the Option Value: At the initial time, take the average of the optimal values across all paths and discount to obtain the current price of the American option.

3.1.2 Implement

In this section, we demonstrate how to use R to implement the pricing of american put options using the LSMC method. The LSMC_pricer function includes the four steps for implementing LSMC described in the previous section. It takes 8 parameters as input and outputs the price of an American put option.

```
format_system_time <- function(expr) {
  result <- system.time(expr)
  cat("User time:", result["user.self"], "seconds\n")
  cat("System time:", result["sys.self"], "seconds\n")
  cat("Elapsed time:", result["elapsed"], "seconds\n")
}
```

```
library(MASS)

# Parameters
S0 <- 100          # Initial stock price
K <- 100           # Strike price
r <- 0.04          # Risk-free interest rate
sigma <- 0.2       # Volatility
T <- 1/12          # Time to maturity (in years)
M <- 10000         # Number of Monte Carlo paths
N <- 5000          # Number of time steps
q <- 0.02          # Dividend yield
```

```
LSMC_pricer <- function(S0, K, r, q, sigma, T, M, N, regressor_number){

  dt <- T / N        # Length of each time step

  # Simulate stock price paths
  set.seed(1)
  S <- matrix(0, nrow = M, ncol = N + 1)
  S[, 1] <- S0  # Initialization
  # Step 1: Simulation
  for (t in 2:(N + 1)) {
    Z <- rnorm(M)
    S[, t] <- S[, t - 1] * exp((r-q - 0.5 * sigma^2) * dt + sigma * sqrt(dt) * Z)
  }

  # Step 2:  Determine the Value at Maturity
  # Initialize a matrix to store cashflows
  cashflows <- matrix(0, nrow = M, ncol = N + 1)

  # Final cashflows at maturity
  cashflows[, N + 1] <- pmax(K - S[, N + 1], 0)

  # Step 3: Backward Induction
  for (t in N:1) {
    itm <- which(S[, t] < K)  # Get indices of in-the-money paths

    if (length(itm) > 0) {  # If there are in-the-money paths
      # Extract stock prices and discounted future cashflows
      X <- S[itm, t]
      Y <- cashflows[itm, t + 1] * exp(-r * dt)

      # Perform regression
      regression <- lm(Y ~ poly(X, regressor_number))  # Use a polynomial regression
      continuation_value <- predict(regression, newdata = data.frame(X = X))

      # Calculate immediate exercise value at time t
      exercise_value <- pmax(0, K - X)

      # If exercise value > continuation value, exercise; otherwise, hold
      cashflows[itm, t] <- ifelse(exercise_value > continuation_value,
                                  exercise_value,  # Exercise the option
                                  cashflows[itm, t + 1] * exp(-r * dt))  # Hold the option

      # out-of-money option's value
      all_paths <- seq_len(nrow(S))
      non_itm <- setdiff(all_paths, itm) # remove itm, keep the rest
      cashflows[non_itm,t] <- cashflows[non_itm,t+1]*exp(-r * dt) # won't exercise, so just discount
    }
  }

  # Step 4: Calculate the Initial Option Price
  # Tip: Since when t=1, all S = S0, regression couldn't be done, so we use cashflow in t=2 to discount for option price
  option_price <- mean(cashflows[, 2]) * exp(-r * dt)
  return(option_price)}

# Test the function
format_system_time({option_price = LSMC_pricer(S0, K, r, q, sigma, T, M, N, 2)})
```

```
## User time: 35.17 seconds
## System time: 8.53 seconds
## Elapsed time: 48.71 seconds
```

```
cat("The estimated price of the American put option:", option_price)
```

```
## The estimated price of the American put option: 2.183974
```

In the following section, we study the impact of the number of paths M, the number of time steps N, and the number of regression variables K on the approximate option price by fixing other parameters and changing one parameter at a time.

The impact of the number of regressors K

```
regressor_numbers <- seq(2,13,1)
option_prices <- sapply(regressor_numbers, function(regressor_number) LSMC_pricer(S0, K, r, q, sigma, T, M, N,regressor_numbe
r))

# Plot the results
plot(
  regressor_numbers, option_prices, type = "b", col = "blue",
  xlab = "Number of Regressors", ylab = "Option Price",
  main = "Option Price vs Number of Regressors"
)
```

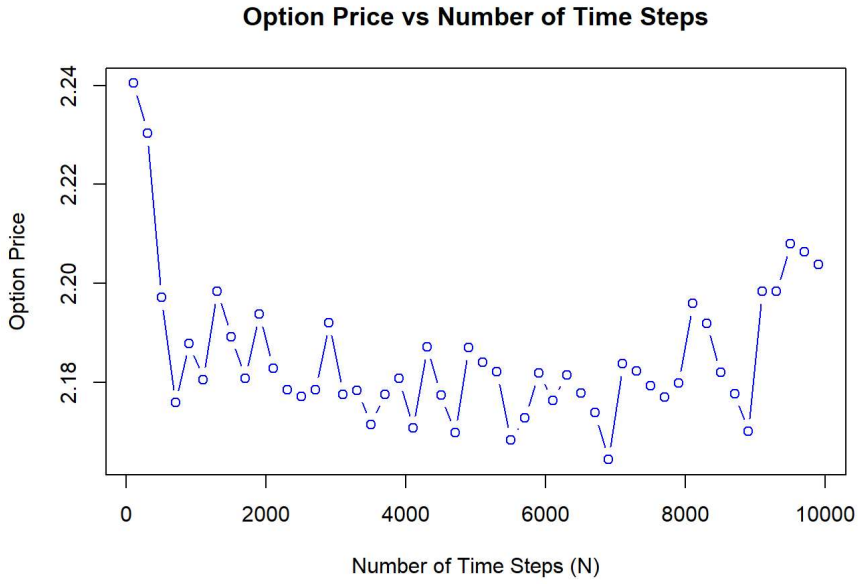
According to the result, the approximate option prices the option price does not show a converging trend but instead maintains high volatility as the number of regression variables increases. This indicates that increasing the number of regression variables does not stabilize the approximate option price.

The impact of the number of time steps N

```
# install.packages("pbapply")
library(pbapply)
```

```
## Warning: 程序包'pbapply'是用R版本4.4.2 来建造的
```

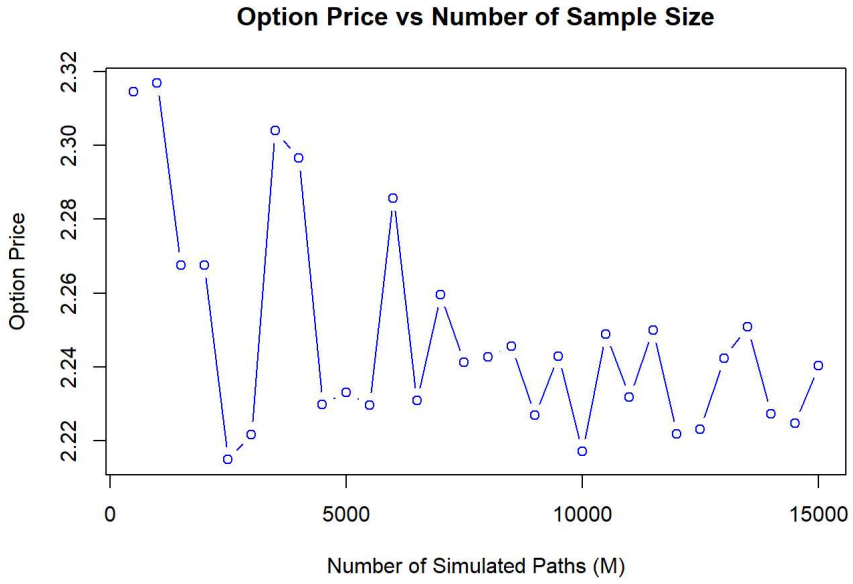
```
N_values <- seq(100, 10000,200)
option_prices <- pbsapply(N_values, function(N) LSMC_pricer(S0, K, r, q, sigma, T, M, N,regressor_number=2))

# Plot the results
plot(
  N_values, option_prices, type = "b", col = "blue",
  xlab = "Number of Time Steps (N)", ylab = "Option Price",
  main = "Option Price vs Number of Time Steps"
)
```



According to the result, the approximate option price gradually decreases and stabilizes as the number of time steps N increases. When N is small (< 1000), the price fluctuates significantly and is unstable. As N reaches a moderate range, the option price converges to a more stable level. This means that increasing the number of time steps helps the approximate price stabilize, but an excessively large number of time steps may also disrupt this stability.

The impact of the number of paths M

```
# Range of M values
M_values <- seq(500,15000,500)
option_prices <- pbsapply(M_values, function(M) LSMC_pricer(S0, K, r, q, sigma, T, M, N,regressor_number=7))

# Plot the results
plot(
  M_values, option_prices, type = "b", col = "blue",
  xlab = "Number of Simulated Paths (M)", ylab = "Option Price",
  main = "Option Price vs Number of Sample Size"
)
```



According to the result, the approximate option price gradually stabilizes as the number of simulated paths increases. When the sample size is small, the pricing results exhibit significant fluctuations, but as the sample size becomes sufficiently large, the option price converges. This indicates that increasing the number of paths helps reduce estimation error and stabilize the approximate option price.

3.2 American Put Option Pricing using with BBSR

3.2.1 Introduction of BBSR

The Binomial Black-Scholes with Richardson extrapolation (BBSR) is an advanced method to price options. It combine the binomial framework and Richardson extrapolation simultaneously to improve the accruacy. The BBSR first employs a binomial tree to simulate the price movement paths of the underlying asset and then works backward from the expiration date to calculate the price of the American option. At each time node, the expected future payoff is equal to the risk-neutral probability-weighted payoff at the next time point. Next, the BBSR method incorporates Richardson extrapolation. This mathematical technique accelerates the convergence of the binomial model's results toward the actual option price by combining estimates with different time step sizes. As a result, the BBSR method can be summarized as using two binomial trees with different time steps for pricing and then combining the two prices.

3.2.2 Implement of BBSR

In this section, we demonstrate how to use R to implement the pricing of american put options using the BBSR method.

We first conduct the BBS function to price a option using specific parameters including the number of time steps N.

```
# Parameters
S0 <- 100          # Initial stock price
K <- 100           # Strike price
r <- 0.04          # Risk-free interest rate
sigma <- 0.2       # Volatility
T <- 1/12             # Time to maturity (in years)
N <- 5000            # Number of time steps
q <- 0.02  # Dividend yield


BBS <- function(S0, K, T, r, q, sigma, N, option_type = "put") {

  # Calculate parameters
  dt <- T / N                  # Time step
  u <- exp(sigma * sqrt(dt))   # Upward factor
  d <- 1 / u                   # Downward factor
  p <- (exp((r-q) * dt) - d) / (u - d)  # Risk-neutral probability
  discount <- exp(-r * dt)     # Discount factor per step

  # Initialize stock price tree
  stock_prices <- matrix(0, nrow = N + 1, ncol = N + 1)
  stock_prices[1, 1] <- S0
  for (i in 2:(N + 1)) {
    stock_prices[i, 1] <- stock_prices[i-1, 1]*u
    for (j in 2:i) {
      stock_prices[i, j] <- stock_prices[i-1, j-1]*d
    }
  }

  # Initialize option values at maturity
  option_values <- matrix(0, nrow = N + 1, ncol = N + 1)
  for (j in 1:(N+1)) {
    if (option_type == "call") {
      option_values[N + 1, j] <- pmax(0, stock_prices[N + 1, j] - K)  # Call payoff
    } else if (option_type == "put") {
      option_values[N + 1, j] <- pmax(0, K - stock_prices[N + 1, j])  # Put payoff
    } }

  # Backward induction to calculate option values
  for (t in N:1) {
    for (i in 1:t) {
      # Risk-neutral expected value
      expected_value <- discount * (p * option_values[t + 1, i] + (1 - p) * option_values[t + 1, i + 1])

      # Immediate exercise value
      if (option_type == "call") {
        exercise_value <- pmax(0, stock_prices[t, i] - K)
      } else {
        exercise_value <- pmax(0, K - stock_prices[t, i])
      }

      # American option: Take the maximum of continuation and immediate exercise
      option_values[t, i] <- max(expected_value, exercise_value)
    }
  }

  # Return the option price at the root of the tree
  return(option_values[1, 1])
}
```

Next, we use the BBS function with time steps of 2N and N to obtain two prices, and then apply Richardson Extrapolation to combine them into the final pric, which could increase converge rate from O(1/N) to O(1/N^2) and lead to better accuracy.

```
BBSR <- function(S0, K, T, r, q, sigma, N){
  opt_price = 2*BBS(S0, K, T, r, q, sigma, 2*N) - BBS(S0, K, T, r, q, sigma, N)
  return(opt_price)
}

format_system_time({
  C = BBSR(S0, K, T, r, q, sigma, N)
})
```

```
## User time: 444.66 seconds
## System time: 0.86 seconds
## Elapsed time: 460.82 seconds
```

```
print(C)
```

```
## [1] 2.225905
```

As the result, the option price derived from BBSR is 2.225905, which is aligned with the results of LSMC in the previous parts, since their values are very close.

3.3 Conclusion of section 3

In this section, we developed LSMC method and BBSR method for american put option pricing. We examined the impact of different parameters on the approximate price. The results show that increasing the number of simulation paths and the number of time steps helps the approximate price converge, while increasing the number of regressors does not. Finally, we implemented the BBSR method to obtain accurate pricing results, which align with the results of LSMC. This further demonstrates that increasing the number of simulation paths and the number of time steps can improve pricing accuracy.

References

- Sun, J., & Chen, X. (2015, August 19). Asian option pricing formula for uncertain financial market. *Journal of Uncertainty Analysis and Applications*. SpringerOpen. https://juaajournal.springeropen.com/articles/10.1186/s40467-015-0035-7 (https://juaajournal.springeropen.com/articles/10.1186/s40467-015-0035-7).
- Lo, C.-L., Palmer, K. J., & Yu, M.-T. (2014, May 31). Moment-matching approximations for Asian options. *The Journal of Derivatives*. https://www.pmresearch.com/content/iijderiv/21/4/103 (https://www.pmresearch.com/content/iijderiv/21/4/103?implicit-login=true).
- Chen, N., & Hong, J. (2007). “Monte Carlo simulation in financial engineering,” *2007 Winter Simulation Conference*, Washington, DC, USA, pp. 919-931. doi:10.1109/WSC.2007.4419688 (https://doi.org/10.1109/WSC.2007.4419688)
- Hammersley, J., & Morton, K. (1955, May). A New Monte Carlo Technique: Antithetic Variates. *Cambridge Core*. https://www.cambridge.org/core (https://www.cambridge.org/core).
- Han, C.-H., & Lai, Y. (2009). Generalized control variate methods for pricing Asian options. http://mx.nthu.edu.tw/~chhan/gen_AAO_v.kan.final.pdf (http://mx.nthu.edu.tw/~chhan/gen_AAO_v.kan.final.pdf).
- Hayes, A. (2023, October 31). Black-Scholes model: What it is, how it works, options formula. *Investopedia*. https://www.investopedia.com/terms/b/blackscholes.asp (https://www.investopedia.com/terms/b/blackscholes.asp#:~:text=Developed%20in%201973%20by%20Fischer,interest%20rates%2C%20time%20to%20e