MTH5530: Assignment 2

Zheshi Dong / Patapongrat Lertbuaban
ID: 33768420 / 33471797

Due date: May 5, 2024; 11:55pm

1. Introduction

The Heston stochastic volatility model has been widely used in the field of quantitative finance and option pricing. This model assumes that the volatility of the process follows a random process. In the Heston stochastic volatility model, we have:

$$dS_t = rS_t d_t + \sqrt{V_t} S_t dW_t^1$$
$$dV_t = -\kappa(V_t - \theta)d_t + \omega\sqrt{V_t} dW_t^2$$
$$dW_t^1 dW_t^2 = \rho d_t$$

Satisfaction with the below Feller condition ensures the positivity of the variance process:

$$\kappa\theta \geq \frac{1}{2}\omega^2$$

When the Feller condition is not satisfied, it is possible that the variance process is not all-time positive. In this case, the four schemes from the research paper by Lord et al. (2010) can be applied in Euler discretization to cure the non-positivity problem, namely: the absorption, reflection, partial truncation and full truncation schemes.

2. Euler Discretization and Fixing Schemes

Euler discretization for V reads:

$$V(t + \Delta t) = (1 - \kappa\Delta t)V(t) + \kappa\theta\Delta t + \omega\sqrt{V(t)}\Delta W_v(t)$$

With $\Delta W_v(t) = W_v(t + \Delta t) - W_v(t)$

To cure the problem of running into negative V - hence the invalidity of $\sqrt{V(t)}$ - four schemes can be implemented through the below general framework and their respective formulations in Table 1:

$$\tilde{V}(t + \Delta t) = f_1(\tilde{V}(t)) - \kappa\Delta t(f_2(\tilde{V}(t) - \theta) + \omega\sqrt{f_3(\tilde{V}(t))}\Delta W_v(t)$$
$$V(t + \Delta t) = f_3(\tilde{V}(t + \Delta t))$$

Where $\tilde{V}(0) = V(0)$

Table 1: Formulations of Fixing Schemes

Scheme	$f_1(x)$	$f_2(x)$	$f_3(x)$
Absorption	x^+	x^+	x^+
Reflection	$\|x\|$	$\|x\|$	$\|x\|$
Partial truncation	x	x	x^+
Full truncation	x	x^+	x^+

As such, we have a general formula for the stock price process:

$$lnS(t+\Delta t) = lnS(t) + (\mu - \frac{1}{2}\lambda^2 S(t)^{2(\beta-1)}V(t))\Delta t + \lambda S(t)^{\beta-1}\sqrt{V(t)}\Delta W_s(t)$$

In the Heston model, $\mu = r$, $\lambda = 1$, $\beta = 1$, hence the stock price process in the Heston model is discretized as:

$$lnS(t+\Delta t) = lnS(t) + (r - \frac{1}{2}V(t))\Delta t + \sqrt{V(t)}\Delta W_s(t)$$

where $\Delta W_s(t) = \rho\Delta W_v(t) + \sqrt{1-\rho^2}\Delta Z(t)$ with $Z(t)$ independent of $W_v(t)$.

3. Comparison of Fixing Schemes

In order to compare each Euler scheme, the European call option is priced with the following parameters:

$$S_0 = 100, K = 100, T = 5, r = 0.05, V_0 = 0.09, \theta = 0.09, \kappa = 2, \omega = 1, \rho = -0.3$$

Note that the Feller condition is violated:

$$2 * 0.09 \not\geq \frac{1}{2}(1)^2$$

Hence, the Euler schemes are needed to cure the non-positivity of the variance process.
The true value for the European call option with the above parameters is 34.9998. We will apply the Euler discretization with all four Euler schemes and use the following measures to assess the pricing error.
Firstly, the root mean square error (RMSE) is used, which is defined as:

$$RMSE(\hat{x}) = \sqrt{\mathbb{E}[(\hat{x}-x)^2]}$$

where $\hat{x}$ is an estimator of the true value x that is to be estimated.
For Monte Carlo simulation, the standard error $s(\hat{x})$ is used to measure the deviation to the expected value, which is defined as:

$$s(\hat{x}) = \sqrt{\mathbb{E}[(\hat{x}-\mathbb{E}[\hat{x}])^2]}$$

The bias can also be measured by:

$$bias(\hat{x}) = |\mathbb{E}[\hat{x}] - x|$$

where the following relation holds: $RMSE(\hat{x}) = \sqrt{bias(\hat{x})^2 + s(\hat{x})^2}$

The Monte Carlo simulation is implemented with 10,000, 40,000 and 160,000 simulated paths and with 20, 40, and 80 steps per year, respectively. Each scheme will be repeated 100 times to estimate

the expected option price and calculate the pricing error.
The results from the simulation are shown in Table 2 below.

Table 2: Comparison of Fixing Scheme ($\omega = 1$)

Methods	**Number of Paths**	**10,000**	**40,000**	**160,000**
	Steps/year	**20**	**40**	**80**
Absorption Scheme	Bias	2.1406	1.5722	1.1829
	Standard Error	0.6086	0.3312	0.1400
	RMSE	2.2254	1.6067	1.1911
	Run Time (sec)	4.58	48.40	495.75
Reflection Scheme	Bias	4.5069	3.1605	2.3549
	Standard Error	0.8486	0.3370	0.1678
	RMSE	4.5861	3.1784	2.3608
	Run Time (sec)	4.29	42.49	490.17
Partial Truncation Scheme	Bias	0.4310	0.1738	0.0829
	Standard Error	0.5958	0.2593	0.1436
	RMSE	0.7354	0.3121	0.1658
	Run Time (sec)	3.99	42.95	500.26
Full Truncation Scheme	Bias	0.0502	0.0142	0.0236
	Standard Error	0.5810	0.2960	0.1428
	RMSE	0.5832	0.2963	0.1447
	Run Time (sec)	4.12	41.44	506.75

Based on the results presented above, the full truncation scheme displayed the lowest bias, standard error, and RMSE, making it the best-performing scheme. Increasing the number of paths and steps per year reduced the bias, standard error, and RMSE in all schemes, but it also increased the run time. It is worth noting that the run time is similar for all schemes, regardless of the method used, as it is determined by the number of paths and steps per year.

Additionally, the algorithm's runtime depends on the computational power. Running it on a MacBook Pro M3 with 12 threads is approximately 3 times faster than on a MacBook Pro Intel Core i5 with 8 threads.

4. Calculation of Delta and Gamma

Delta and gamma are part of the Greeks useful for hedging and risk management. Delta measures the sensitivity of option price to changes in underlying asset price. Gamma measures the sensitivity of delta to changes in underlying asset price. Applying central difference approximation, their formula is as follows:

$$\Delta_{Heston} = \frac{\partial C}{\partial S} \approx \frac{C(S_0 + \Delta S) - C(S_0 - \Delta S)}{2\Delta S}$$

$$\Gamma_{Heston} = \frac{\partial^2 C}{\partial S^2} \approx \frac{C(S_0 + \Delta S) - 2C(S_0) + C(S_0 - \Delta S)}{(\Delta S)^2}$$

Assuming $\Delta S = 0.01 S_0$, 100,000 simulations and 50 time steps/year, we calculated the delta and gamma of the above European call option under the Heston model with full truncation scheme:

$$\Delta_{Heston} = 0.7964$$

$$\Gamma_{Heston} = 0.0047$$

Comparing to the delta and gamma under Black-Scholes model:

$$\Delta_{BS} = 0.7606$$

$$\Gamma_{BS} = 0.0046$$

We found that, albeit those of Heston / full truncation are slighter higher, the results overall are very similar under both models. With the unrealistic assumption of constant volatility under Black-Scholes model, the preference is for the Heston model with a full truncation scheme.

5. Comparison of Fixing Schemes ($\omega = 0.3$)

When ω is changed to 0.3 instead of 1, and all other parameters are unchanged, the Feller condition is not violated:

$$2 * 0.09 \geq \frac{1}{2}(0.3)^2$$

The simulation results are summarized in Table 3 below.

Table 3: Comparison of Fixing Scheme ($\omega = 0.3$)

Methods	**Number of Paths**	**10,000**	**40,000**	**160,000**
	Steps/year	**20**	**40**	**80**
Absorption Scheme	Bias	0.8636	0.9320	0.8729
	Standard Error	0.5916	0.3241	0.1426
	RMSE	1.0468	0.9868	0.8845
	Run Time (sec)	4.16	43.65	503.85
Reflection Scheme	Bias	0.8108	0.8124	0.8710
	Standard Error	0.6067	0.3652	0.1589
	RMSE	1.0126	0.8907	0.8853
	Run Time (sec)	3.95	39.03	496.06
Partial Truncation Scheme	Bias	0.8491	0.8921	0.8627
	Standard Error	0.6628	0.2854	0.1622
	RMSE	1.0772	0.9366	0.8778
	Run Time (sec)	4.07	42.54	498.11
Full Truncation Scheme	Bias	0.8872	0.8689	0.8617
	Standard Error	0.6411	0.2975	0.1683
	RMSE	1.0945	0.9185	0.8780
	Run Time (sec)	3.87	40.71	497.06

As can be seen, when the Feller condition is not violated, for a particular combination of the number of paths and steps per year, the four schemes generated relatively similar bias, standard error and RMSE. This is because, when the Feller condition is not violated, the variance process is non-negative, rendering any adjustment schemes for tackling the negativity issue redundant; that is, the effect of the adjustment by each scheme is negligible, hence they all generate similar simulation performance.

6. American Put Option Pricing

We applied the Longstaff-Schwartz method for pricing the American put option. Specifically, we first simulated the variance and price processes of the underlying asset under the Heston model with a full truncation scheme using the stipulated parameters. We then worked backward in time to evaluate at each time step whether it is more profitable to exercise at that very time step or to continue, in which least squares Monte Carlo is applied for the discounted expected continuation value that's approximated by a quadratic function of asset price S and variance V. The cash flow matrix is then updated based on the decision to exercise or continue. By repeating the above steps and moving back in time, the final cash flow matrix is obtained which is then discounted and averaged over all paths to get the American put option price at $t = 0$:

$$P_{American} = 16.1353$$

Under the European option, we simulate variance and price processes of the underlying asset, and evaluate the option payoff at maturity which is then discounted to arrive at the option price at $t = 0$:

$$P_{European} = 12.7146$$

We can see that the American put option price is higher than the European put option price. That is because, whilst a European option only allows exercise at maturity, an American option allows exercise at and before maturity. Given the higher probability of being in the money (the entire path under the American option vs terminal only under the European option), an American option usually has a higher premium (price) than the corresponding European option with the same parameters.

References

Calculating the Greeks with Finite Difference and Monte Carlo Methods in C++. Quantstart. https://www.quantstart.com/articles/Calculating-the-Greeks-with-Finite-Difference-and-Monte-Carlo-Methods-in-C/

Lord, R., Koekkoek, R., Dijk, D. V. (2010). A comparison of biased simulation schemes for stochastic volatility models. *Quantitative Finance*, 10(2), 177-194. http://dx.doi.org/10.1080/14697680802392496.

Tian. (2024). *MTH5530: Computational Finance*. Monash University.

Appendix: Codes

Importing Libraries

```
import numpy as np
import time
from scipy.stats import norm
from sklearn.linear_model import LinearRegression
```

Euler Scheme Function

```
def get_truncation_functions(scheme):
        if scheme == 1:  # Absorption scheme
            return (lambda x: np.maximum(x, 0),
                    lambda x: np.maximum(x, 0),
                    lambda x: np.maximum(x, 0))
        elif scheme == 2:  # Reflection scheme
            return (np.abs, np.abs, np.abs)
        elif scheme == 3:  # Partial truncation scheme
            return (lambda x: x, lambda x: x, lambda x: np.maximum(0, x))
        elif scheme == 4:  # Full truncation scheme
            return (lambda x: x, lambda x: np.maximum(0, x), lambda x: np.maximum(0, x))
```

Function for Heston Model

```
def european_call_heston_model(S_0, K, T, r, V_0, theta, kappa, omega, rho, steps_per_year, num_paths, scheme):

    # Get the truncation functions
    f_1, f_2, f_3 = get_truncation_functions(scheme)

    # Calculate dt
    dt = 1 / steps_per_year

    # Initialize the matrices for V and ln(S)
    V_tilde = np.zeros((num_paths, int(T/dt)+1))
    V = np.zeros((num_paths, int(T/dt)+1))
    ln_S_current = np.zeros((num_paths, int(T/dt)+1))

    # Fill in the initial value
    V_tilde[:, 0] = V_0
    V[:, 0] = V_0
    ln_S_current[:, 0] = np.log(S_0)

    # Simulate dW_V
    dW_V = np.random.normal(0, np.sqrt(dt), (num_paths, int(T/dt)))

    # Simulate dZ
    dZ = np.random.normal(0, np.sqrt(dt), (num_paths, int(T/dt)))

    # Simulate Correlated Brownian Motion
    dW_S = rho * dW_V + np.sqrt(1 - rho**2) * dZ

    # Loop through each step and calculate V and ln(S) according to the Heston framework using Euler scheme
    for j in range(int(T/dt)):
        V_tilde[:, j+1] = (f_1(V_tilde[:, j])
                           - kappa * dt * (f_2(V_tilde[:, j]) - theta)
                           + omega * np.sqrt(f_3(V_tilde[:, j])) * dW_V[:, j])
        V[:, j+1] = f_3(V_tilde[:, j+1])
        ln_S_current[:, j+1] = (ln_S_current[:, j] + (r - 0.5 * V[:, j]) * dt
                           + np.sqrt(V[:, j]) * dW_S[:, j])

    # Get the stock price at time T by converting the ln(S) process to S
    S_T_current = np.exp(ln_S_current[:, -1])

    # Calculate the option price
    call_price_T_current = np.exp(-r * T) * np.mean(np.maximum(S_T_current - K, 0))

    return call_price_T_current
```

Initializing the Variables

```
# Initialize the variables
S_0 = 100   # Spot price
K = 100     # Strike price
T = 5       # Time period (years)
r = 0.05    # Risk-neutral drift of the asset price
V_0 = 0.09  # Initial variance
theta = 0.09  # Long-term average variance
kappa = 2   # Speed of mean-reversion of the variance
omega = 1   # Volatility of volatility
rho = -0.3  # Instantaneous correlation coefficient
x_true = 34.9998 # True value of the call option
```

Pricing Error Measures

```
# Pricing error measures
def calc_bias(x):
    return np.abs(np.mean(x) - x_true)

def calc_std(x):
    return np.sqrt(np.mean((x - np.mean(x))**2))

def calc_RMSE(x):
    return np.sqrt(calc_bias(x)**2 + calc_std(x)**2)
```

Scheme Comparison

```python
# Create the list of num_path
num_paths_list = [10000, 40000, 160000]

# Create the list of steps/year
steps_per_year_list = [20, 40, 80]

# Perform the monte-carlo simulation
num_iters = 100 # repeat each scheme 100 times
scheme_names = ['Absorption Scheme', 'Reflection Scheme', 'Partial Truncation Scheme', 'Full Truncation Scheme']

# Loop through every scheme
for scheme in range(1, 5):
    # Match the scheme name for printing out the results
    scheme_name = scheme_names[scheme-1]
    for i in range(3):
        num_paths = num_paths_list[i]
        steps_per_year = steps_per_year_list[i]
        print('-----------------------------------------------------')
        print(f'Method: {scheme_name} | Number of Paths: {num_paths} | Steps/year: {steps_per_year}')

        # Initialize the list of values
        call_price_T_list = np.zeros(num_iters)

        # Start the time
        start_time = time.time()

        # Repeat each scheme 100 times
        for j in range(num_iters):
            # Calculate the European call option prices
            call_price_T_list[j] = european_call_heston_model(
                S_0, K, T, r, V_0, theta, kappa, omega, rho, steps_per_year, num_paths, scheme)

        # Stop the time after repeating each scheme for 100 times
        elapsed_time = time.time() - start_time

        # Print out the elapsed time
        print(f'Time elapsed: {elapsed_time:.2f} seconds')

        # Print out the error measures and option price
        print(f'Bias: {calc_bias(call_price_T_list):.4f} | Std: {calc_std(call_price_T_list):.4f} | RMSE: {calc_RMSE(call_price_T_list):.4f}')
        print(f'Call Option Price: {np.mean(call_price_T_list):.4f} | True Value: {x_true}')
```

```
-----------------------------------------------------
Method: Absorption Scheme | Number of Paths: 10000 | Steps/year: 20
Time elapsed: 4.58 seconds
Bias: 2.1406 | Std: 0.6086 | RMSE: 2.2254
Call Option Price: 37.1404 | True Value: 34.9998
-----------------------------------------------------
Method: Absorption Scheme | Number of Paths: 40000 | Steps/year: 40
Time elapsed: 48.40 seconds
Bias: 1.5722 | Std: 0.3312 | RMSE: 1.6067
Call Option Price: 36.5720 | True Value: 34.9998
-----------------------------------------------------
Method: Absorption Scheme | Number of Paths: 160000 | Steps/year: 80
Time elapsed: 495.75 seconds
Bias: 1.1829 | Std: 0.1400 | RMSE: 1.1911
Call Option Price: 36.1827 | True Value: 34.9998
-----------------------------------------------------
Method: Reflection Scheme | Number of Paths: 10000 | Steps/year: 20
Time elapsed: 4.29 seconds
Bias: 4.5069 | Std: 0.8486 | RMSE: 4.5861
Call Option Price: 39.5067 | True Value: 34.9998
-----------------------------------------------------
Method: Reflection Scheme | Number of Paths: 40000 | Steps/year: 40
Time elapsed: 42.49 seconds
Bias: 3.1605 | Std: 0.3370 | RMSE: 3.1784
Call Option Price: 38.1603 | True Value: 34.9998
-----------------------------------------------------
Method: Reflection Scheme | Number of Paths: 160000 | Steps/year: 80
Time elapsed: 490.17 seconds
Bias: 2.3549 | Std: 0.1678 | RMSE: 2.3608
Call Option Price: 37.3547 | True Value: 34.9998
-----------------------------------------------------
Method: Partial Truncation Scheme | Number of Paths: 10000 | Steps/year: 20
Time elapsed: 3.99 seconds
Bias: 0.4310 | Std: 0.5958 | RMSE: 0.7354
Call Option Price: 35.4308 | True Value: 34.9998
-----------------------------------------------------
Method: Partial Truncation Scheme | Number of Paths: 40000 | Steps/year: 40
Time elapsed: 42.95 seconds
Bias: 0.1738 | Std: 0.2593 | RMSE: 0.3121
Call Option Price: 35.1736 | True Value: 34.9998
-----------------------------------------------------
Method: Partial Truncation Scheme | Number of Paths: 160000 | Steps/year: 80
Time elapsed: 500.26 seconds
Bias: 0.0829 | Std: 0.1436 | RMSE: 0.1658
Call Option Price: 35.0827 | True Value: 34.9998
-----------------------------------------------------
Method: Full Truncation Scheme | Number of Paths: 10000 | Steps/year: 20
Time elapsed: 4.12 seconds
Bias: 0.0502 | Std: 0.5810 | RMSE: 0.5832
Call Option Price: 34.9496 | True Value: 34.9998
```

```
------------------------------------------------------
Method: Full Truncation Scheme | Number of Paths: 40000 | Steps/year: 40
Time elapsed: 41.44 seconds
Bias: 0.0142 | Std: 0.2960 | RMSE: 0.2963
Call Option Price: 34.9856 | True Value: 34.9998
------------------------------------------------------
Method: Full Truncation Scheme | Number of Paths: 160000 | Steps/year: 80
Time elapsed: 506.75 seconds
Bias: 0.0236 | Std: 0.1428 | RMSE: 0.1447
Call Option Price: 35.0234 | True Value: 34.9998
```

Delta and Gamma Function

```python
def delta_gamma_calc(S_0, K, T, r, V_0, theta, kappa, omega, rho, steps_per_year, num_paths, scheme):

    # Get the truncation functions
    f_1, f_2, f_3 = get_truncation_functions(scheme)

    # Calculate dt
    dt = 1 / steps_per_year

    # Calculate dS
    dS = 0.01 * S_0

    # Initialize the matrices for V and ln(S)
    V_tilde = np.zeros((num_paths, int(T/dt)+1))
    V = np.zeros((num_paths, int(T/dt)+1))
    ln_S_current = np.zeros((num_paths, int(T/dt)+1))
    ln_S_before = np.zeros((num_paths, int(T/dt)+1))
    ln_S_after = np.zeros((num_paths, int(T/dt)+1))

    # Fill in the initial value
    V_tilde[:, 0] = V_0
    V[:, 0] = V_0
    ln_S_current[:, 0] = np.log(S_0)
    ln_S_before[:, 0] = np.log(S_0 - dS)
    ln_S_after[:, 0] = np.log(S_0 + dS)

    # Simulate dW_V
    dW_V = np.random.normal(0, np.sqrt(dt), (num_paths, int(T/dt)))

    # Simulate dZ
    dZ = np.random.normal(0, np.sqrt(dt), (num_paths, int(T/dt)))

    # Simulate Correlated Brownian Motion
    dW_S = rho * dW_V + np.sqrt(1 - rho**2) * dZ
```

```python
    # Loop through each step and calculate V and ln(S) according to the Heston framework using Euler scheme
    for j in range(int(T/dt)):
        V_tilde[:, j+1] = (f_1(V_tilde[:, j])
                           - kappa * dt * (f_2(V_tilde[:, j]) - theta)
                           + omega * np.sqrt(f_3(V_tilde[:, j])) * dW_V[:, j])
        V[:, j+1] = f_3(V_tilde[:, j+1])
        ln_S_current[:, j+1] = (ln_S_current[:, j] + (r - 0.5 * V[:, j]) * dt
                                + np.sqrt(V[:, j]) * dW_S[:, j])
        ln_S_before[:, j+1] = (ln_S_before[:, j] + (r - 0.5 * V[:, j]) * dt
                               + np.sqrt(V[:, j]) * dW_S[:, j])
        ln_S_after[:, j+1] = (ln_S_after[:, j] + (r - 0.5 * V[:, j]) * dt
                              + np.sqrt(V[:, j]) * dW_S[:, j])

    # Get the stock prices process at time T by converting the ln(S) process to S
    S_T_current = np.exp(ln_S_current[:, -1])
    S_T_before = np.exp(ln_S_before[:, -1])
    S_T_after = np.exp(ln_S_after[:, -1])

    # Calculate the option prices
    call_price_T_currrent = np.exp(-r * T) * np.mean(np.maximum(S_T_current - K, 0))
    call_price_T_before = np.exp(-r * T) * np.mean(np.maximum(S_T_before - K, 0))
    call_price_T_after = np.exp(-r * T) * np.mean(np.maximum(S_T_after - K, 0))

    # Calculate delta and gamma using finite difference method
    delta = (call_price_T_after - call_price_T_before) / (2 * dS)
    gamma = (call_price_T_after - 2 * call_price_T_currrent + call_price_T_before) / (dS ** 2)

    return call_price_T_currrent, delta, gamma
```

Initializing Variables

```python
# Initialize the variables
# The previous variables are the same
sigma = 0.3 # Sigma
num_paths = 100000 # 100000 path
steps_per_year = 50 # 50 steps/year
scheme = 4 # Full truncation scheme
```

Calculating Delta and Gamma using the Black-Scholes model

```python
# Calculate Delta and Gamma by using the Black-Scholes model
def black_scholes_greeks(S, K, r, T, sigma):
    d1 = (np.log(S / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
    delta = norm.cdf(d1)
    gamma = norm.pdf(d1) / (S * sigma * np.sqrt(T))
    return delta, gamma
```

Calculating Delta and Gamma using Finite Difference Method

```
# Print out the Delta and Gamma from the Black-Scholes model
Delta_BS, Gamma_BS = black_scholes_greeks(S_0, K, r, T, sigma)
print(f'Delta_BS: {Delta_BS:.4f}')
print(f'Gamma_BS: {Gamma_BS:.4f}')

# Perform the monte-carlo simulation
num_iters = 100 # repeat each scheme 100 times

scheme_name = 'Full Truncation Scheme'
print('-------------------------------------------------')
print(f'Method: {scheme_name} | Number of Paths: {num_paths} | Steps/year: {steps_per_year}')

# Initialize the list of values
call_price_T_list = np.zeros(num_iters)
Delta_list = np.zeros(num_iters)
Gamma_list = np.zeros(num_iters)

# Start the time
start_time = time.time()

# Repeat each scheme 100 times
for j in range(num_iters):
    # Calculate the European call option prices, delta, and gamma
    call_price_T_list[j], Delta_list[j], Gamma_list[j] = delta_gamma_calc(
        S_0, K, T, r, V_0, theta, kappa, omega, rho, steps_per_year, num_paths, scheme)

# Stop the time after repeating each scheme for 100 times
elapsed_time = time.time() - start_time

# Print out the elapsed time
print(f'Time elapsed: {elapsed_time:.2f} seconds')

# Print out the error measures and option price
print(f'Bias: {calc_bias(call_price_T_list):.4f} | Std: {calc_std(call_price_T_list):.4f} | RMSE: {calc_RMSE(call_price_T_list):.4f}')
print(f'Call Option Price: {np.mean(call_price_T_list):.4f} | True Value: {x_true} | Delta: {np.mean(Delta_list):.4f} | Gamma: {np.mean(Gamma_list):.4f}')
```

```
Delta_BS: 0.7606
Gamma_BS: 0.0046
-------------------------------------------------
Method: Full Truncation Scheme | Number of Paths: 100000 | Steps/year: 50
Time elapsed: 275.69 seconds
Bias: 0.0386 | Std: 0.1839 | RMSE: 0.1879
Call Option Price: 35.0384 | True Value: 34.9998 | Delta: 0.7964 | Gamma: 0.0047
```

Scheme Comparison (Omega=0.3)

```
# Change the omega to 0.3
omega = 0.3

# Create the list of num_path
num_paths_list = [10000, 40000, 160000]

# Create the list of steps/year
steps_per_year_list = [20, 40, 80]

# Perform the monte-carlo simulation
num_iters = 100 # repeat each scheme 100 times
scheme_names = ['Absorption Scheme', 'Reflection Scheme', 'Partial Truncation Scheme', 'Full Truncation Scheme']

# Loop through every scheme
for scheme in range(1, 5):
    # Match the scheme name for printing out the results
    scheme_name = scheme_names[scheme-1]
    for i in range(3):
        num_paths = num_paths_list[i]
        steps_per_year = steps_per_year_list[i]
        print('-------------------------------------------------')
        print(f'Method: {scheme_name} | Number of Paths: {num_paths} | Steps/year: {steps_per_year}')

        # Initialize the list of values
        call_price_T_list = np.zeros(num_iters)

        # Start the time
        start_time = time.time()

        # Repeat each scheme 100 times
        for j in range(num_iters):
            # Calculate the European call option prices
            call_price_T_list[j] = european_call_heston_model(
                S_0, K, T, r, V_0, theta, kappa, omega, rho, steps_per_year, num_paths, scheme)

        # Stop the time after repeating each scheme for 100 times
        elapsed_time = time.time() - start_time

        # Print out the elapsed time
        print(f'Time elapsed: {elapsed_time:.2f} seconds')

        # Print out the error measures and option price
        print(f'Bias: {calc_bias(call_price_T_list):.4f} | Std: {calc_std(call_price_T_list):.4f} | RMSE: {calc_RMSE(call_price_T_list):.4f}')
        print(f'Call Option Price: {np.mean(call_price_T_list):.4f} | True Value: {x_true}')
```

```
--------------------------------------------------
Method: Absorption Scheme | Number of Paths: 10000 | Steps/year: 20
Time elapsed: 4.16 seconds
Bias: 0.8636 | Std: 0.5916 | RMSE: 1.0468
Call Option Price: 35.8634 | True Value: 34.9998
--------------------------------------------------
Method: Absorption Scheme | Number of Paths: 40000 | Steps/year: 40
Time elapsed: 43.65 seconds
Bias: 0.9320 | Std: 0.3241 | RMSE: 0.9868
Call Option Price: 35.9318 | True Value: 34.9998
--------------------------------------------------
Method: Absorption Scheme | Number of Paths: 160000 | Steps/year: 80
Time elapsed: 503.85 seconds
Bias: 0.8729 | Std: 0.1426 | RMSE: 0.8845
Call Option Price: 35.8727 | True Value: 34.9998
--------------------------------------------------
Method: Reflection Scheme | Number of Paths: 10000 | Steps/year: 20
Time elapsed: 3.95 seconds
Bias: 0.8108 | Std: 0.6067 | RMSE: 1.0126
Call Option Price: 35.8106 | True Value: 34.9998
--------------------------------------------------
Method: Reflection Scheme | Number of Paths: 40000 | Steps/year: 40
Time elapsed: 39.03 seconds
Bias: 0.8124 | Std: 0.3652 | RMSE: 0.8907
Call Option Price: 35.8122 | True Value: 34.9998
--------------------------------------------------
Method: Reflection Scheme | Number of Paths: 160000 | Steps/year: 80
Time elapsed: 496.06 seconds
Bias: 0.8710 | Std: 0.1589 | RMSE: 0.8853
Call Option Price: 35.8708 | True Value: 34.9998
--------------------------------------------------
Method: Partial Truncation Scheme | Number of Paths: 10000 | Steps/year: 20
Time elapsed: 4.07 seconds
Bias: 0.8491 | Std: 0.6628 | RMSE: 1.0772
Call Option Price: 35.8489 | True Value: 34.9998
--------------------------------------------------
Method: Partial Truncation Scheme | Number of Paths: 40000 | Steps/year: 40
Time elapsed: 42.54 seconds
Bias: 0.8921 | Std: 0.2854 | RMSE: 0.9366
Call Option Price: 35.8919 | True Value: 34.9998
--------------------------------------------------
Method: Partial Truncation Scheme | Number of Paths: 160000 | Steps/year: 80
Time elapsed: 498.11 seconds
Bias: 0.8627 | Std: 0.1622 | RMSE: 0.8778
Call Option Price: 35.8625 | True Value: 34.9998
--------------------------------------------------
Method: Full Truncation Scheme | Number of Paths: 10000 | Steps/year: 20
Time elapsed: 3.87 seconds
Bias: 0.8872 | Std: 0.6411 | RMSE: 1.0945
Call Option Price: 35.8870 | True Value: 34.9998
```

```
--------------------------------------------------
Method: Full Truncation Scheme | Number of Paths: 40000 | Steps/year: 40
Time elapsed: 40.71 seconds
Bias: 0.8689 | Std: 0.2975 | RMSE: 0.9185
Call Option Price: 35.8687 | True Value: 34.9998
--------------------------------------------------
Method: Full Truncation Scheme | Number of Paths: 160000 | Steps/year: 80
Time elapsed: 497.06 seconds
Bias: 0.8617 | Std: 0.1683 | RMSE: 0.8780
Call Option Price: 35.8615 | True Value: 34.9998
```

Calculating European Put Option with Full Truncation under Heston Model

```python
def european_put_heston_model(S_0, K, T, r, V_0, theta, kappa, omega, rho, steps_per_year, num_paths, scheme):

    # Get the truncation functions
    f_1, f_2, f_3 = get_truncation_functions(scheme)

    # Calculate dt
    dt = 1 / steps_per_year

    # Initialize the matrices for V and ln(S)
    V_tilde = np.zeros((num_paths, int(T/dt)+1))
    V = np.zeros((num_paths, int(T/dt)+1))
    ln_S_current = np.zeros((num_paths, int(T/dt)+1))

    # Fill in the initial value
    V_tilde[:, 0] = V_0
    V[:, 0] = V_0
    ln_S_current[:, 0] = np.log(S_0)

    # Simulate dW_V
    dW_V = np.random.normal(0, np.sqrt(dt), (num_paths, int(T/dt)))

    # Simulate dZ
    dZ = np.random.normal(0, np.sqrt(dt), (num_paths, int(T/dt)))

    # Simulate Correlated Brownian Motion
    dW_S = rho * dW_V + np.sqrt(1 - rho**2) * dZ

    # Loop through each step and calculate V and ln(S) according to the Heston framework using Euler scheme
    for j in range(int(T/dt)):
        V_tilde[:, j+1] = (f_1(V_tilde[:, j])
                           - kappa * dt * (f_2(V_tilde[:, j]) - theta)
                           + omega * np.sqrt(f_3(V_tilde[:, j])) * dW_V[:, j])
        V[:, j+1] = f_3(V_tilde[:, j+1])
        ln_S_current[:, j+1] = (ln_S_current[:, j] + (r - 0.5 * V[:, j]) * dt
                           + np.sqrt(V[:, j]) * dW_S[:, j])
```

```
    # Get the stock price at time T by converting the ln(S) process to S
    S_T_current = np.exp(ln_S_current[:, -1])

    # Calculate the option price
    put_price_T_current = np.exp(-r * T) * np.mean(np.maximum(K - S_T_current, 0))

    return put_price_T_current
```

```
# Initialize the variables
S_0 = 100   # Spot price
K = 100     # Strike price
T = 5       # Time period (years)
r = 0.05    # Risk-neutral drift of the asset price
V_0 = 0.09  # Initial variance
theta = 0.09  # Long-term average variance
kappa = 2   # Speed of mean-reversion of the variance
omega = 1   # Volatility of volatility
rho = -0.3  # Instantaneous correlation coefficient
steps_per_year = 50 # 50 steps/year
num_paths = 10000 # 10000 simulated paths

print(f'The European put option price using Full Truncation scheme under Heston Model is \
{european_put_heston_model(S_0, K, T, r, V_0, theta, kappa, omega, rho, steps_per_year, num_paths, scheme=4)}')
```

```
The European put option price using Full Truncation scheme under Heston Model is 12.714589758735851
```

Calculating American Put Option (LSMC) with Full Truncation under Heston Model

```
def american_put_heston_model(S_0, K, T, r, V_0, theta, kappa, omega, rho, steps_per_year, num_paths, scheme):

    # Calculate number of steps
    num_steps = int(T * steps_per_year)

    # Get the truncation functions
    f_1, f_2, f_3 = get_truncation_functions(scheme)

    # Calculate dt
    dt = 1 / steps_per_year

    # Initialize the matrices for V and ln(S)
    V_tilde = np.zeros((num_paths, int(T/dt)+1))
    V = np.zeros((num_paths, int(T/dt)+1))
    ln_S_current = np.zeros((num_paths, int(T/dt)+1))

    # Fill in the initial value
    V_tilde[:, 0] = V_0
    V[:, 0] = V_0
    ln_S_current[:, 0] = np.log(S_0)

    # Simulate dW_V
    dW_V = np.random.normal(0, np.sqrt(dt), (num_paths, int(T/dt)))

    # Simulate dZ
    dZ = np.random.normal(0, np.sqrt(dt), (num_paths, int(T/dt)))

    # Simulate Correlated Brownian Motion
    dW_S = rho * dW_V + np.sqrt(1 - rho**2) * dZ

    # Loop through each step and calculate V and ln(S) according to the Heston framework using Euler scheme
    for j in range(int(T/dt)):
        V_tilde[:, j+1] = (f_1(V_tilde[:, j])
                           - kappa * dt * (f_2(V_tilde[:, j]) - theta)
                           + omega * np.sqrt(f_3(V_tilde[:, j])) * dW_V[:, j])
        V[:, j+1] = f_3(V_tilde[:, j+1])
        ln_S_current[:, j+1] = (ln_S_current[:, j] + (r - 0.5 * V[:, j]) * dt
                           + np.sqrt(V[:, j]) * dW_S[:, j])
```

```
    # Get the stock prices by converting the ln(S) process to S
    S_T_current = np.exp(ln_S_current)

    # Calculate the payoff
    payoff = np.maximum(K - S_T_current, 0)

    # Initialize the value matrix as the payoff matrix
    value = np.copy(payoff)

    # Loop through steps
    for t in range(num_steps - 1, 0, -1):

        # Get the stock prices and variances at time t
        S_t = S_T_current[:, t]
        V_t = V[:, t]

        # Fit the values
        features = np.column_stack((np.ones_like(S_t), S_t, S_t**2, V_t, V_t**2, S_t * V_t))
        model = LinearRegression().fit(features, np.exp(-r * dt) * value[:, t+1])

        # Predict the continuation value
        continuation_value = model.predict(features)

        # Compare the continuation value to the payoff
        value[:, t] = np.where(payoff[:, t] > continuation_value, payoff[:, t], np.exp(-r * dt) * value[:, t+1])

    # Calculate the American put option price
    price = np.mean(np.exp(-r * dt) * value)
    return price
```

```
# Initialize the variables
S_0 = 100   # Spot price
K = 100     # Strike price
T = 5       # Time period (years)
r = 0.05    # Risk-neutral drift of the asset price
V_0 = 0.09  # Initial variance
theta = 0.09  # Long-term average variance
kappa = 2   # Speed of mean-reversion of the variance
omega = 1   # Volatility of volatility
rho = -0.3  # Instantaneous correlation coefficient
steps_per_year = 50 # 50 steps/year
num_paths = 10000 # 10000 simulated paths

print(f'The American put option price using LSMC with Full Truncation under Heston Model is \
      {american_put_heston_model(S_0, K, T, r, V_0, theta, kappa, omega, rho, steps_per_year, num_paths, scheme=4)}')
```

```
The American put option price using LSMC with Full Truncation under Heston Model is 16.13530526717778
```