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

Step 1

Stochastic Volatility Modelling

Heston Model Implementation

General Parameters

```
#Initialize parameters
import numpy as np
from scipy.stats import norm

S0 = 80  # Initial stock price
r = 0.055  # Risk-free rate
sigma = 0.35  # Initial volatility
T = 3/12  # Time to maturity (3 months)

# Heston model parameters
v0 = 0.032  # Initial variance
kappa = 1.85  # Mean reversion speed of variance
theta = 0.045  # Long-term variance
xi = 0.1  # Volatility of variance (typically between 0.1 and 0.5)

# Simulation parameters
num_simulations = 100000
num_steps = 90  # 30 steps per month
dt = T / num_steps


def heston_model_mc(S0, v0, r, T, kappa, theta, xi, rho, num_simulations, num_steps):
    dt = T / num_steps

    # Initialize arrays
    S = np.zeros((num_simulations, num_steps + 1))
    v = np.zeros((num_simulations, num_steps + 1))

    # Set initial values
    S[:, 0] = S0
    v[:, 0] = v0

    # Generate correlated random numbers
    z1 = np.random.normal(0, 1, (num_simulations, num_steps))
    z2 = rho * z1 + np.sqrt(1 - rho**2) * np.random.normal(0, 1, (num_simulations, num_steps))

    for i in range(1, num_steps + 1):
        S[:, i] = S[:, i-1] * np.exp((r - 0.5 * v[:, i-1]) * dt + np.sqrt(v[:, i-1] * dt) * z1[:, i-1])
        v[:, i] = np.maximum(v[:, i-1] + kappa * (theta - v[:, i-1]) * dt + xi * np.sqrt(v[:, i-1] * dt) * z2[:, i-1], 0)

    return S[:, -1]

def price_option(S, K, r, T, option_type):
    if option_type == 'call':
        payoff = np.maximum(S - K, 0)
    else:
        payoff = np.maximum(K - S, 0)
    return np.exp(-r * T) * np.mean(payoff)
```

5. Price ATM European call and put (ρ = -0.30)

```
rho = -0.30
K = S0  # ATM option

final_prices_30 = heston_model_mc(S0, v0, r, T, kappa, theta, xi, rho == -0.30, num_simulations, num_steps)
```

```
hs_call_price_30 = price_option(final_prices_30, K, r, T, 'call')
hs_put_price_30 = price_option(final_prices_30, K, r, T, 'put')

print(f"ATM Heston European Call Price (rho = -0.30): ${hs_call_price_30:.2f}")
print(f"ATM Heston European Put Price (rho = -0.30): ${hs_put_price_30:.2f}")
```

```
ATM Heston European Call Price (rho = -0.30): $3.49
ATM Heston European Put Price (rho = -0.30): $2.42
```

6. Price ATM European call and put (ρ = -0.70)

```
rho = -0.70
K = S0  # ATM option

final_prices_70 = heston_model_mc(S0, v0, r, T, kappa, theta, xi, rho == -0.70, num_simulations, num_steps)

hs_call_price_70 = price_option(final_prices_70, K, r, T, 'call')
hs_put_price_70 = price_option(final_prices_70, K, r, T, 'put')

print(f"ATM Heston European Call Price (rho = -0.70): ${hs_call_price_70:.2f}")
print(f"ATM Heston European Put Price (rho = -0.70): ${hs_put_price_70:.2f}")
```

```
ATM Heston European Call Price (rho = -0.70): $3.49
ATM Heston European Put Price (rho = -0.70): $2.42
```

7. Heston's Delta and Gamma

```
def calculate_greeks(S0, v0, r, T, kappa, theta, xi, rho, K, num_simulations, num_steps, option_type, h=0.01): # h = small pertubation in va
    # Base price
    final_prices = heston_model_mc(S0, v0, r, T, kappa, theta, xi, rho, num_simulations, num_steps)
    base_price = price_option(final_prices, K, r, T, option_type)

    # Up price
    final_prices_up = heston_model_mc(S0 + h, v0, r, T, kappa, theta, xi, rho, num_simulations, num_steps)
    up_price = price_option(final_prices_up, K, r, T, option_type)

    # Down price
    final_prices_down = heston_model_mc(S0 - h, v0, r, T, kappa, theta, xi, rho, num_simulations, num_steps)
    down_price = price_option(final_prices_down, K, r, T, option_type)

    # Calculate delta
    delta = (up_price - down_price) / (2 * h)

    # Calculate gamma
    gamma = (up_price - 2 * base_price + down_price) / (h ** 2)

    return delta, gamma

# Calculate greeks for rho = -0.30
rho = -0.30
hs_call_delta, hs_call_gamma = calculate_greeks(S0, v0, r, T, kappa, theta, xi, rho, K, num_simulations, num_steps, 'call')
hs_put_delta, hs_put_gamma = calculate_greeks(S0, v0, r, T, kappa, theta, xi, rho, K, num_simulations, num_steps, 'put')

print(f" Heston Call Delta (rho = -0.30): {hs_call_delta:.4f}")
print(f" Heston Call Gamma (rho = -0.30): {hs_call_gamma:.4f}")
print(f"Heston Put Delta (rho = -0.30): {hs_put_delta:.4f}")
print(f"Heston Put Gamma (rho = -0.30): {hs_put_gamma:.4f}")

# Calculate greeks for rho = -0.70
rho = -0.70
hs_call_delta, hs_call_gamma = calculate_greeks(S0, v0, r, T, kappa, theta, xi, rho, K, num_simulations, num_steps, 'call')
hs_put_delta, hs_put_gamma = calculate_greeks(S0, v0, r, T, kappa, theta, xi, rho, K, num_simulations, num_steps, 'put')

print(f"Heston Call Delta (rho = -0.70): {hs_call_delta:.4f}")
print(f"Heston Call Gamma (rho = -0.70): {hs_call_gamma:.4f}")
print(f"Heston Put Delta (rho = -0.70): {hs_put_delta:.4f}")
print(f"Heston Put Gamma (rho = -0.70): {hs_put_gamma:.4f}")
```

```
 Heston Call Delta (rho = -0.30): 1.6282
 Heston Call Gamma (rho = -0.30): -519.4442
Heston Put Delta (rho = -0.30): 0.0080
Heston Put Gamma (rho = -0.30): -12.6955
Heston Call Delta (rho = -0.70): 1.2880
Heston Call Gamma (rho = -0.70): -500.0661
Heston Put Delta (rho = -0.70): 0.3710
```

```
    Heston Put Gamma (rho = -0.70): -396.9052
```

Jump Modelling

Merton Model

8. Price ATM European call and put (λ = 0.75)

```
## Initialize parameters

mu = -0.5  # average jump size
delta = 0.22  # Delta
r = 0.055  # Risk-free rate
sigma = 0.35  # Volatility
T = 3/12  # Maturity/time period (in years)
S0 = 80  # Current Stock Price
K = S0 # ATM strike

Ite = 100000  # Number of simulations (paths)
M = 90  # Number of steps (in 3 months)
dt = T / M  # Time-step
t = 0


SM = np.zeros((M + 1, Ite))
SM[0] = S0
def merton_call_mc(SM, K, r, lamb, T, t,sigma, delta):
    # rj
    rj = lamb * (np.exp(mu + 0.5 * delta**2) - 1)

    # Random numbers
    z1 = np.random.standard_normal((M + 1, Ite))
    z2 = np.random.standard_normal((M + 1, Ite))
    y = np.random.poisson(lamb * dt, (M + 1, Ite))


    for t in range(1, M + 1):
        SM[t] = SM[t - 1,:] * (
            np.exp((r - rj - 0.5 * sigma**2) * dt + sigma * np.sqrt(dt) * z1[t])
            + (np.exp(mu + delta * z2[t]) - 1) * y[t]
            )
    return np.maximum(SM[t], 0.00001)  # To ensure that the price never goes below zero!

def option_price (SM, K, option_type):
    if option_type == 'call':
        payoff = np.maximum(0, SM - K)
    else:
        payoff = np.maximum(0, K - SM )

    average = np.mean(payoff)

    return np.exp(-r * (T - t)) * average


final_stock_price_75 = merton_call_mc(SM, K, r, 0.75, T, 0, sigma, delta)

mt_call_price_75 = option_price(final_stock_price_75, K, 'call')
mt_put_price_75 = option_price(final_stock_price_75,  K, 'put')

print(f"European Call Price under Merton (lambda = 0.75) : {mt_call_price_75:.2f}")
print(f"European Put Price under Merton  (lambda = 0.75) : {mt_put_price_75:.2f}")
```

```
European Call Price under Merton (lambda = 0.75) : 8.34
European Put Price under Merton  (lambda = 0.75) : 7.20
```

9. Price ATM European call and put (λ = 0.25)

```
final_stock_price_25 = merton_call_mc(SM, K, r, 0.25, T, 0, sigma, delta)

mt_call_price_25 = option_price(final_stock_price_25, K, 'call')
mt_put_price_25 = option_price(final_stock_price_25,  K, 'put')
```

```
print(f"European Call Price under Merton (lambda = 0.25) : {mt_call_price_25:.2f}")
print(f"European Put Price under Merton  (lambda = 0.25) : {mt_put_price_25:.2f}")
```

```
European Call Price under Merton (lambda = 0.25) : 6.83
European Put Price under Merton  (lambda = 0.25) : 5.74
```

10. Merton's Delta and Gamma

```
# Initialize parameters

mu = -0.5  # average jump size
delta = 0.22  # Delta
r = 0.055  # Risk-free rate
sigma = 0.35  # Volatility
T = 3/12  # Maturity/time period (in years)
S0 = 80  # Current Stock Price
K = S0 # ATM strike

Ite = 100000  # Number of simulations (paths)
M = 90  # Number of steps (in 3 months)
dt = T / M  # Time-step
t = 0 # Start time


def merton_greeks(SM, r, T, lamb, K, Ite, M, option_type, h=0.01): # h = small pertubation in variable
    final_stock_price = merton_call_mc(SM, K, r, lamb, T, 0, sigma, delta)
    # Base price
    price = option_price(final_stock_price, K, option_type)

    # Up price
    price_up = option_price(final_stock_price + h, K, option_type)

    # Down price
    price_down = option_price(final_stock_price - h, K, option_type)

    # Calculate delta
    mt_delta = (price_up - price_down) / (2 * h)

    # Calculate gamma
    mt_gamma = (price_up - 2 * price + price_down) / (h ** 2)

    return mt_delta, mt_gamma

# Calculate greeks for lambda = 0.75
mt_call_delta, mt_call_gamma =  merton_greeks(SM, r, T, 0.75, K, Ite, M, 'call' , h=0.01)
mt_put_delta, mt_put_gamma =  merton_greeks(SM, r, T, 0.75, K, Ite, M, 'put' , h=0.01)

print(f" Merton Call Delta (lambda = 0.75): {mt_call_delta:.4f}")
print(f" Merton Call Gamma (lambda = 0.75): {mt_call_gamma:.4f}")
print(f"Merton Put Delta (lambda = 0.75): {mt_put_delta:.4f}")
print(f"Merton Put Gamma (lambda = 0.75): {mt_put_gamma:.4f}")

# Calculate greeks for lamba = 0.25
mt_call_delta, mt_call_gamma =  merton_greeks(SM, r, T, 0.25, K, Ite, M, 'call' , h=0.01)
mt_put_delta, mt_put_gamma =  merton_greeks(SM, r, T,0.25, K, Ite, M, 'put' , h=0.01)

print(f" Merton Call Delta (lambda = 0.25): {mt_call_delta:.4f}")
print(f" Merton Call Gamma (lambda = 0.25): {mt_call_gamma:.4f}")
print(f"Merton Put Delta (lambda = 0.25): {mt_put_delta:.4f}")
print(f"Merton Put Gamma (lambda = 0.25): {mt_put_gamma:.4f}")
```

```
 Merton Call Delta (lambda = 0.75): 0.5433
 Merton Call Gamma (lambda = 0.75): 0.0195
Merton Put Delta (lambda = 0.75): -0.4421
Merton Put Gamma (lambda = 0.75): 0.0222
 Merton Call Delta (lambda = 0.25): 0.5142
 Merton Call Gamma (lambda = 0.25): 0.0300
Merton Put Delta (lambda = 0.25): -0.4754
Merton Put Gamma (lambda = 0.25): 0.0261
```

Model Validation

11. Put-Call Parity (Heston-Merton Comparison)

```
# Put-Call Parity Difference for Heston Model (when rho = -0.30)
hs_parity_difference_30 = hs_call_price_30 - hs_put_price_30 - (S0 - K * np.exp(-r * T))
print(f"Heston Put-Call Parity Difference (rho = -0.30): {hs_parity_difference_30:.4f}")
```

```
Heston Put-Call Parity Difference (rho = -0.30): -0.0239
```

```
# Put-Call Parity Difference for Heston Model (when rho = -0.70)
hs_parity_difference_70 = hs_call_price_70 - hs_put_price_70 - (S0 - K * np.exp(-r * T))
print(f"Heston Put-Call Parity Difference (rho = -0.70): {hs_parity_difference_70:.4f}")
```

```
Heston Put-Call Parity Difference (rho = -0.70): -0.0253
```

```
# Put-Call Parity Difference for Merton Model (when lamda = 0.75)
mt_parity_difference_75 = mt_call_price_75 - mt_put_price_75 - (S0 - K * np.exp(-r * T))
print(f"Merton Put-Call Parity Difference (lamda = 0.75): {mt_parity_difference_75:.4f}")
```

```
Merton Put-Call Parity Difference (lamda = 0.75): 0.0542
```

```
# Put-Call Parity Difference for Merton Model (when lamda = 0.25)
mt_parity_difference_25 = mt_call_price_25 - mt_put_price_25 - (S0 - K * np.exp(-r * T))
print(f"Merton Put-Call Parity Difference (lamda = 0.25): {mt_parity_difference_25:.4f}")
```

```
Merton Put-Call Parity Difference (lamda = 0.25): 0.0004
```

Put-Call Parity is satisfied under both the Heston and Merton models. While the parity difference for both models are within a tolerance range very close to zero, it suffice to say that The averaging of option prices across a large number of simulations tend give varying values but significantly low deviation from 0.

12. Heston & Merton Option Prices with Different Strikes

S0 = 80

For Call Option:

- For Deep ITM:
- i. we will consider moneyness (K/S0_) of **85%**. (i.e **K = 68**)
- ii. we will consider moneyness (K/S0_) of **90%**. (i.e **K = 72**)
- For ITM :
- we will consider moneyness (K/S0) of **95%** (i.e **K = 76**)
- ATM: K = S0 = **80** (moneyness (K/S0) of **100%**)
- For OTM:
- - we will consider moneyness (K/S0) of **105%**. (i.e **K = 84**)
- For Deep OTM:
- i. we will consider moneyness (K/S0) of **110%** (i.e **K = 88**)
- ii. we will consider moneyness (K/S0) of **115%** (i.e **K = 92**)

For Put Option:

- For Deep OTM:
- i. we will consider moneyness (K/S0_) of **85%**. (i.e **K = 68**)
- ii. we will consider moneyness (K/S0_) of **90%**. (i.e **K = 72**)
- For OTM :
- - we will consider moneyness (K/S0) of **95%** (i.e **K = 76**)
- ATM: K = S0 = **80** (moneyness (K/S0) of **100%**)
- For ITM:
- - we will consider moneyness (K/S0) of **105%**. (i.e **K = 84**)
- For Deep ITM:
- i. we will consider moneyness (K/S0) of **110%** (i.e **K = 88**)

- ii. we will consider moneyness (K/S0) of **115%** (i.e **K = 92**)

Heston European Calls & Puts

```
# Heston European Calls for different Strike Prices (rho = -0.30)
hs_european_call_30_array = []
for K in [68, 72, 76, 80, 84, 88, 92]:
    hs_european_call_price_30 = price_option(final_prices_30, K, r, T, 'call')
    hs_european_call_30_array.append(hs_european_call_price_30)
    print("With K = {:3d}, Heston Call price (rho = -0.30) is {:.2f}".format(K, hs_european_call_price_30))
```

```
With K =  68, Heston Call price (rho = -0.30) is 12.94
With K =  72, Heston Call price (rho = -0.30) is 9.20
With K =  76, Heston Call price (rho = -0.30) is 5.96
With K =  80, Heston Call price (rho = -0.30) is 3.49
With K =  84, Heston Call price (rho = -0.30) is 1.86
With K =  88, Heston Call price (rho = -0.30) is 0.92
With K =  92, Heston Call price (rho = -0.30) is 0.42
```

```
# Heston European Calls for different Strike Prices (rho = -0.70)
hs_european_call_70_array = []
for K in [68, 72, 76, 80, 84, 88, 92]:
    hs_european_call_price_70 = price_option(final_prices_70, K, r, T, 'call')
    hs_european_call_70_array.append(hs_european_call_price_70)
    print("With K = {:3d}, Heston Call price (rho = -0.70) is {:.2f}".format(K, hs_european_call_price_70))
```

```
With K =  68, Heston Call price (rho = -0.70) is 12.94
With K =  72, Heston Call price (rho = -0.70) is 9.20
With K =  76, Heston Call price (rho = -0.70) is 5.96
With K =  80, Heston Call price (rho = -0.70) is 3.49
With K =  84, Heston Call price (rho = -0.70) is 1.86
With K =  88, Heston Call price (rho = -0.70) is 0.91
With K =  92, Heston Call price (rho = -0.70) is 0.42
```

```
# Heston European Puts for different Strike Prices (rho = -0.30)
hs_european_put_30_array = []
for K in [68, 72, 76, 80, 84, 88, 92]:
    hs_european_put_price_30 = price_option(final_prices_30, K, r, T, 'put')
    hs_european_put_30_array.append(hs_european_put_price_30)
    print("With K = {:3d}, Heston Put price (rho = -0.30) is {:.2f}".format(K, hs_european_put_price_30))
```

```
With K =  68, Heston Put price (rho = -0.30) is 0.03
With K =  72, Heston Put price (rho = -0.30) is 0.25
With K =  76, Heston Put price (rho = -0.30) is 0.94
With K =  80, Heston Put price (rho = -0.30) is 2.42
With K =  84, Heston Put price (rho = -0.30) is 4.74
With K =  88, Heston Put price (rho = -0.30) is 7.74
With K =  92, Heston Put price (rho = -0.30) is 11.19
```

```
# Heston European Puts for different Strike Prices (rho = -0.70)
hs_european_put_70_array = []
for K in [68, 72, 76, 80, 84, 88, 92]:
    hs_european_put_price_70 = price_option(final_prices_70, K, r, T, 'put')
    hs_european_put_70_array.append(hs_european_put_price_70)
    print("With K = {:3d}, Heston Put price (rho = -0.70) is {:.2f}".format(K, hs_european_put_price_70))
```

```
With K =  68, Heston Put price (rho = -0.70) is 0.03
With K =  72, Heston Put price (rho = -0.70) is 0.24
With K =  76, Heston Put price (rho = -0.70) is 0.95
With K =  80, Heston Put price (rho = -0.70) is 2.42
With K =  84, Heston Put price (rho = -0.70) is 4.74
With K =  88, Heston Put price (rho = -0.70) is 7.73
With K =  92, Heston Put price (rho = -0.70) is 11.18
```

```
# compile list of arrays
strike_prices = np.array([68, 72, 76, 80, 84, 88, 92]) # for 3 ITMs, ATM, & 3 OTMs
heston_euro_raw_data = pd.DataFrame([strike_prices, hs_european_call_30_array, hs_european_put_30_array, hs_european_call_70_array, hs_europe
heston_euro_data = heston_euro_raw_data.transpose()
heston_euro_data.columns = ["Strike_prices", "Heston_Call_30", "Heston_Put_30","Heston_Call_70", "Heston_Put_70" ]
heston_euro_data.round(2)
```

	Strike_prices	Heston_Call_30	Heston_Put_30	Heston_Call_70	Heston_Put_70
0	68.0	12.94	0.03	12.94	0.03
1	72.0	9.20	0.25	9.20	0.24
2	76.0	5.96	0.94	5.96	0.95
3	80.0	3.49	2.42	3.49	2.42
4	84.0	1.86	4.74	1.86	4.74
5	88.0	0.92	7.74	0.91	7.73
6	92.0	0.42	11.19	0.42	11.18

Merton European Calls & Puts

```
# Merton European Calls for different Strike Prices (lambda = 0.75)
mt_european_call_75_array = []
for K in [68, 72, 76, 80, 84, 88, 92]:
    mt_european_call_price_75 = option_price(final_stock_price_75, K, 'call')
    mt_european_call_75_array.append(mt_european_call_price_75)
    print("With K = {:3d}, Merton Call price (lambda = 0.75) is {:.2f}".format(K, mt_european_call_price_75))
```

```
With K =  68, Merton Call price (lambda = 0.75) is 16.35
With K =  72, Merton Call price (lambda = 0.75) is 13.39
With K =  76, Merton Call price (lambda = 0.75) is 10.71
With K =  80, Merton Call price (lambda = 0.75) is 8.34
With K =  84, Merton Call price (lambda = 0.75) is 6.34
With K =  88, Merton Call price (lambda = 0.75) is 4.70
With K =  92, Merton Call price (lambda = 0.75) is 3.40
```

```
# Merton European Calls for different Strike Prices (lambda = 0.25)
mt_european_call_25_array = []
for K in [68, 72, 76, 80, 84, 88, 92]:
    mt_european_call_price_25 = option_price(final_stock_price_25, K, 'call')
    mt_european_call_25_array.append(mt_european_call_price_25)
    print("With K = {:3d}, Merton Call price (lambda = 0.25) is {:.2f}".format(K, mt_european_call_price_25))
```

```
With K =  68, Merton Call price (lambda = 0.25) is 14.81
With K =  72, Merton Call price (lambda = 0.25) is 11.77
With K =  76, Merton Call price (lambda = 0.25) is 9.09
With K =  80, Merton Call price (lambda = 0.25) is 6.83
With K =  84, Merton Call price (lambda = 0.25) is 4.99
With K =  88, Merton Call price (lambda = 0.25) is 3.54
With K =  92, Merton Call price (lambda = 0.25) is 2.46
```

```
# Merton European Puts for different Strike Prices (lambda = 0.75)
mt_european_put_75_array = []
for K in [68, 72, 76, 80, 84, 88, 92]:
    mt_european_put_price_75 = option_price(final_stock_price_75, K, 'put')
    mt_european_put_75_array.append(mt_european_put_price_75)
    print("With K = {:3d}, Merton Put price (lambda = 0.75) is {:.2f}".format(K, mt_european_put_price_75))
```

```
With K =  68, Merton Put price (lambda = 0.75) is 3.37
With K =  72, Merton Put price (lambda = 0.75) is 4.35
With K =  76, Merton Put price (lambda = 0.75) is 5.61
With K =  80, Merton Put price (lambda = 0.75) is 7.20
With K =  84, Merton Put price (lambda = 0.75) is 9.14
With K =  88, Merton Put price (lambda = 0.75) is 11.44
With K =  92, Merton Put price (lambda = 0.75) is 14.09
```

```
# Merton European Puts for different Strike Prices (lambda = 0.25)
mt_european_put_25_array = []
for K in [68, 72, 76, 80, 84, 88, 92]:
    mt_european_put_price_25 = option_price(final_stock_price_25, K, 'put')
    mt_european_put_25_array.append(mt_european_put_price_25)
    print("With K = {:3d}, Merton Put price (lambda = 0.25) is {:.2f}".format(K, mt_european_put_price_25))
```

```
With K =  68, Merton Put price (lambda = 0.25) is 1.88
With K =  72, Merton Put price (lambda = 0.25) is 2.78
With K =  76, Merton Put price (lambda = 0.25) is 4.06
With K =  80, Merton Put price (lambda = 0.25) is 5.74
With K =  84, Merton Put price (lambda = 0.25) is 7.84
With K =  88, Merton Put price (lambda = 0.25) is 10.34
With K =  92, Merton Put price (lambda = 0.25) is 13.20
```

```
# compile list of arrays
strike_prices = np.array([68, 72, 76, 80, 84, 88, 92]) # for 3 ITMs, ATM, & 3 OTMs
merton_euro_raw_data = pd.DataFrame([strike_prices, mt_european_call_75_array, mt_european_put_75_array, mt_european_call_25_array, mt_europ
merton_euro_data = merton_euro_raw_data.transpose()
merton_euro_data.columns = ["Strike_prices", "Merton_Call_75", "Merton_Put_75","Merton_Call_25", "Merton_Put_25" ]
merton_euro_data.round(2)
```

	Strike_prices	Merton_Call_75	Merton_Put_75	Merton_Call_25	Merton_Put_25
0	68.0	16.35	3.37	14.81	1.88
1	72.0	13.39	4.35	11.77	2.78
2	76.0	10.71	5.61	9.09	4.06
3	80.0	8.34	7.20	6.83	5.74
4	84.0	6.34	9.14	4.99	7.84
5	88.0	4.70	11.44	3.54	10.34
6	92.0	3.40	14.09	2.46	13.20

Step 2

13. Pricing American Call options Using Heston & Merton models

Heston American Call (ρ = -0.30)

```
# Initialize paramters
S0 = 80  # Initial stock price
K = 80   # Strike price (ATM)
T = 3/12 # Time to maturity (3 months)
r = 0.055 # Risk-free rate
sigma = 0.35 # Volatility
num_sim = 100000
num_steps = 90

# Heston model parameters
v0 = 0.032  # Initial variance
kappa = 1.85  # Mean reversion speed of variance
theta = 0.045  # Long-term variance
xi = 0.1  # Volatility of variance (typically between 0.1 and 0.5)
rho = -0.30


# Heston Model
def american_option_heston(S0, K, T, r, v0, kappa, theta, sigma, rho, num_sim, num_steps):
    dt = T / num_steps
    S = np.zeros((num_sim, num_steps + 1))
    V = np.zeros((num_sim, num_steps + 1))
    S[:, 0] = S0
    V[:, 0] = v0

    for t in range(1, num_steps + 1):
        z1 = np.random.normal(size=num_sim)
        z2 = np.random.normal(size=num_sim)
        W1 = z1
        W2 = rho * z1 + np.sqrt(1 - rho**2) * z2

        V[:, t] = np.maximum(
            V[:, t - 1] +
            kappa * (theta - V[:, t - 1]) * dt +
            sigma * np.sqrt(V[:, t - 1] * dt) * W2, 0)

        S[:, t] = S[:, t - 1] * np.exp((r - 0.5 * V[:, t - 1]) * dt + np.sqrt(V[:, t - 1] * dt) * W1)

    return S, V


# Monte-carlo simualation for American Call
def ls_american_call_mc(S, K, r, dt):
    n_sim, n_steps = S.shape
    n_steps -= 1  # Adjust for initial condition
    payoff = np.maximum(S[:, -1] - K, 0)
    cashflows = payoff.copy()

    for t in range(n_steps - 1, 0, -1):

        itm_indices = S[:, t] > K  # In-the-money condition
        if not np.any(itm_indices):
                continue

        X = S[itm_indices, t]
        Y = cashflows[itm_indices] * np.exp(-r * dt)

        # Regression to estimate continuation value
        regression = np.polyfit(X, Y, deg=2)
        continuation_value = np.polyval(regression, X)

        exercise_value = X - K
        exercise = exercise_value > continuation_value

        cashflows[itm_indices] = np.where(exercise, exercise_value, cashflows[itm_indices] * np.exp(-r * dt))


    price = np.mean(cashflows * np.exp(-r * dt))
    return price

S, V = american_option_heston(S0, K, r, T, v0, kappa, theta, sigma, rho, num_sim, num_steps)
american_call_heston = ls_american_call_mc(S, K, r, dt)

print(f"American Call Option Price (Heston Model): {american_call_heston:.2f}")
```

```
American Call Option Price (Heston Model): 1.96
```

Merton American Call (λ = 0.75)

```
# Merton model parameters
lambda_ = 0.75 # Jump intensity
mu_j = -0.5 # Jump size mean
sigma_j = 0.22 # Jump size standard deviation


def price_american_call_merton(S0, K, T, r, sigma, lambda_, mu_j, sigma_j, num_simulations, num_steps, option_type):
    dt = T / num_steps

    # Generate stock price paths using Merton model
    S = np.zeros((num_simulations, num_steps + 1))
    S[:, 0] = S0

    for t in range(1, num_steps + 1):
        Z = np.random.normal(0, 1, num_simulations)
        N = np.random.poisson(lambda_ * dt, num_simulations)
        J = np.random.normal(mu_j, sigma_j, num_simulations)

        S[:, t] = S[:, t-1] * np.exp((r - lambda_ * (np.exp(mu_j + 0.5 * sigma_j**2) - 1) - 0.5 * sigma**2) * dt
                                     + sigma * np.sqrt(dt) * Z
                                     + N * J)

    # Initialize option values at maturity
    option_values = np.maximum(S[:, -1] - K, 0)

    # Backward induction for early exercise
    for t in range(num_steps - 1, 0, -1):
        in_the_money = S[:, t] > K
        X = S[in_the_money, t]
        Y = option_values[in_the_money] * np.exp(-r * dt)

        # Fit a polynomial regression
        A = np.vstack([X**0, X**1, X**2]).T
        beta = np.linalg.lstsq(A, Y, rcond=None)[0]

        # Calculate continuation values
        continuation_values = np.dot(A, beta)

        # Update option values
        immediate_exercise = S[in_the_money, t] - K
        option_values[in_the_money] = np.maximum(immediate_exercise, continuation_values)
        option_values[~in_the_money] *= np.exp(-r * dt)

    # Estimate option price
    american_call_price = np.mean(option_values) * np.exp(-r * dt)

    return american_call_price


american_call_price_merton = price_american_call_merton(S0, K, T, r, sigma, lambda_, mu_j, sigma_j, num_simulations, num_steps, 'call')
print(f"American Call Option Price (Merton model): ${american_call_price_merton:.2f}")
```

```
American Call Option Price (Merton model): $8.32
```

Differences between Heston American Call Option Price and Merton American Call Price

The results show that a merton american call price is significantly larger than that of the Heston American Call Option price. This is as a result of the following:

- the Merton's call is a reflection of jumps in the volatility unlike volatility of the Heston's model which is stochastic and mean reverting. Hence, call option price for the former tends to higher for an early exercise american option
- option premiums for heston call is sensitive low-value paramters like kappa, theta and sigma whereas the option premium fpr merton's call is sensitve to extreme jump intensity value (lamnda). The affects the call option price results
- Merton's Call Option price is skew-driven while heston's call prices is a reflection of volatility smile.

14. Pricing a European Up-and-In Call option using the Heston model

```
# UAI parameters:
H = 95 # Barrier price
K = 95
rho = -0.70


def heston_model_mc(S0, v0, r, T, kappa, theta, sigma, rho, num_sim, num_steps):
    dt = T / num_steps
    S = np.zeros((num_sim, num_steps + 1))
    v = np.zeros((num_sim, num_steps + 1))
    S[:, 0] = S0
    v[:, 0] = v0

    for i in range(1, num_steps + 1):
        z1 = np.random.normal(0, 1, num_sim)
        z2 = rho * z1 + np.sqrt(1 - rho**2) * np.random.normal(0, 1, num_simulations)
        S[:, i] = S[:, i-1] * np.exp((r - 0.5 * v[:, i-1]) * dt + np.sqrt(v[:, i-1] * dt) * z1)
        v[:, i] = np.maximum(v[:, i-1] + kappa * (theta - v[:, i-1]) * dt + sigma * np.sqrt(v[:, i-1] * dt) * z2, 0)

    return S
def price_uai_call_heston(S0, K, H, T, r, v0, kappa, theta, sigma, rho, num_sim, num_steps):
    S = heston_model_mc(S0, v0, r, T, kappa, theta, sigma, rho, num_sim, num_steps)

    barrier_hit = np.any(S >= H, axis=1)
    payoffs = np.where(barrier_hit, np.maximum(S[:, -1] - K, 0), 0)

    option_price = np.exp(-r * T) * np.mean(payoffs)
    return option_price

# Price UAI Call Option
uai_call_price = price_uai_call_heston(S0, K, H, T, r, v0, kappa, theta, sigma, rho, num_sim, num_steps)

# Price simple European Call Option (for comparison)
simple_call_price = hs_call_price_70  # from prior python code

print(f"UAI Call Price: ${uai_call_price:.2f}")
print(f"Simple European Call Price: ${simple_call_price:.2f}")
```

```
UAI Call Price: $0.03
Simple European Call Price: $3.49
```

Comparison between UAI Heston Call Option Price and Simple Heston European Call Price

The results show that a simple european heston call is significantly larger than the up-and-in- barrier call option. This is as a result of the following:

- the call option is only active when the stock exceeds the barrier price, resulting in lower (cheaper) call option price at expiration

- the probability of exercise is higher for simple european call since there is no activation barrier
- a simple european call is easier to hedge because of non-existence of barrier price.
- simple european call has vanilla payoff whereas the UAI option only pay off only when stock price hit barrier price

15. Pricing a European Down-and-In Put option using the Merton model

```
#DAI parameters:
H = 65 # Barrier price
K = 65 # Strike
lam = 0.75 #jump intensity


def merton_jump_path(S0, r, sigma, T, lam, mu_j, sigma_j, steps, num_paths):
    dt = T / steps
    t = np.linspace(0, T, steps+1)
    S = np.zeros((num_paths, steps+1))
    S[:, 0] = S0

    for i in range(1, steps+1):
        poisson = np.random.poisson(lam*dt, num_paths)
        jump = np.random.normal(mu_j, sigma_j, num_paths) * poisson
        drift = (r - lam*(np.exp(mu_j + 0.5*sigma_j**2) - 1) - 0.5*sigma**2) * dt
        diffusion = sigma * np.sqrt(dt) * np.random.standard_normal(num_paths)
        S[:, i] = S[:, i-1] * np.exp(drift + diffusion + jump)

    return S

def price_dai_put_merton(S0, K, H, T, r, sigma, lam, mu_j, sigma_j, num_simulations, num_steps):
    S = merton_jump_path(S0, r, sigma, T, lam, mu_j, sigma_j, num_steps, num_simulations)

    barrier_hit = np.any(S <= H, axis=1)
    payoffs = np.where(barrier_hit, np.maximum(K - S[:, -1], 0), 0)

    option_price = np.exp(-r * T) * np.mean(payoffs)
    return option_price


# Price DAI Put Option
dai_put_price = price_dai_put_merton(S0, K, H, T, r, sigma, lam, mu_j, sigma_j, num_simulations, num_steps)

# Price simple European Put Option (for comparison)
simple_put_price = mt_put_price_75 # from prior merton python code

print(f"DAI Put Price: ${dai_put_price:.2f}")
print(f"Simple European Put Price: ${simple_put_price:.2f}")
```

```
DAI Put Price: $2.78
Simple European Put Price: $7.20
```

Comparison between DAI Merton Put Option Price and Simple Merton European Put Price

The results show that a simple merton european put is significantly larger than the down-and-in- barrier put option. This is as a result of the following:

- the put option is only active when it stock falls below the barrier price, resulting in lower (cheaper) put option price at expiration.
- the probability of exercise is higher for simple european put since there is no activation barrier.
- a simple european call is easier to hedge because of non-existence of barrier price.
- simple european call has vanilla payoff whereas the UAI option only pay off only when stock price falls below barrier price.
- The DAI put option is mainly used for special hedging cases where costs is major concern.