Group Work Project 3

```
#Libraries Loaded and Universal Inputs
import numpy as np
import pandas as pd
import math
import warnings
#Set seed for the simulation
np.random.seed(3892)
#Universal Inputs used as model parameters
S0=80
r = 0.055
sigma = 0.35
T = 3/12 #3 months over 1 year
```

Part 1 - Question 5-7: Heston Model

Team Member A: Stochastic Volatility Modeler

Pricing an ATM European Call and Put, with a correlation of -0.30 via the Heston model:

$$S_t = S_{t-1} e^{\left(r - \frac{\nu_t}{2}\right) dt + \sigma \sqrt{\nu_t} dZ_1}$$
$$\nu_t = \nu_{t-1} + \kappa \left(\theta - \nu_{t-1}\right) dt + \sigma \sqrt{\nu_{t-1}} dZ_2$$

```
#Set seed for simulation process:
np.random.seed(3892)
# Heston model inputs (modified from codes of Module 7 Lesson Note 1):
v0 = 0.032
kappa = 1.85
theta = 0.045
rho = -0.3  # Serving for Question 5 with correlation of -0.3

M0 = 50  # Number of time steps in a year
M = int(M0 * T)  # Total time steps
Ite = 10000  # Number of simulations
dt = T / M  # Length of time step

# Heston model - Stochastic Voalitility Function:
def SDE_vol(v0, kappa, theta, sigma, T, M, Ite, rand, row, cho_matrix):
    dt = T / M  # T = maturity, M = number of time steps
    v = np.zeros((M + 1, Ite), dtype=np.float)
    v[0] = v0
    sdt = np.sqrt(dt)  # Sqrt of dt
```

✓ 0s completed at 9:10 AM

```
        ran = np.dot(cho_matrix, rand[:, t])
        v[t] = np.maximum(
            0,
            v[t - 1]
            + kappa * (theta - v[t - 1]) * dt
            + np.sqrt(v[t - 1]) * sigma * ran[row] * sdt,
        )
    return v

# Heston model - Stochastic equation for the underlying asset price evolution:
def Heston_paths(S0, r, v, row, cho_matrix, rand):
    S = np.zeros((M + 1, Ite), dtype=float)
    S[0] = S0
    sdt = np.sqrt(dt)
    for t in range(1, M + 1, 1):
        ran = np.dot(cho_matrix, rand[:, t])
        S[t] = S[t - 1] * np.exp((r - 0.5 * v[t]) * dt + np.sqrt(v[t]) * ran[row] * sdt

    return S

def heston_simulation(v0, kappa, theta, sigma, rho, S0, r, T, M, Ite):
  # generate random numbers
  rand = np.random.standard_normal((2, M + 1, Ite))

  # Covariance Matrix
  covariance_matrix = np.zeros((2, 2), dtype=np.float)
  covariance_matrix[0] = [1.0, rho]
  covariance_matrix[1] = [rho, 1.0]
  cho_matrix = np.linalg.cholesky(covariance_matrix)

  # Volatility process paths
  V = SDE_vol(v0, kappa, theta, sigma, T, M, Ite, rand, 1, cho_matrix)

  # Underlying price process paths
  S = Heston_paths(S0, r, V, 0, cho_matrix, rand)

  return S


# Option Pricing under Heston with Monte Carlo Methods (Dev for both Call and Put Optio
def heston_option_mc(S, K, r, T, t, optype):
    if optype == 'C':
      payoff = np.maximum(0, S[-1, :] - K)
    elif optype == 'P':
      payoff = np.maximum(0, K - S[-1, :])
    else:
      raise ValueError()
    average = np.mean(payoff)
    return np.exp(-r * (T - t)) * average
```

```
np.random.seed(3892)
#Generate price_shock with 1% increase in price of S0
price_shock = S0 * 0.01
rho_1 = -0.3  # Serving for Question 5 with correlation of -0.3

#Pricing the base price of Options (Call and Put) with Heston dynamics:
S_heston = heston_simulation(v0, kappa, theta, sigma, rho_1, S0, r, T, M, Ite)
heston_eu_call_price = heston_option_mc(S_heston, S0, r, T, dt, optype='C')
heston_eu_put_price = heston_option_mc(S_heston, S0, r, T, dt, optype='P')

#Pricing the 101% price_shock price of Options (Call and Put) with Heston dynamics:
S_heston_higher = heston_simulation(v0, kappa, theta, sigma, rho_1, S0 + price_shock, r
heston_call_price_higher = heston_option_mc(S_heston_higher, S0 + price_shock, r, T, dt
heston_put_price_higher = heston_option_mc(S_heston_higher, S0 + price_shock, r, T, dt,

#Pricing the 99% price_shock price of Options (Call and Put) with Heston dynamics:
S_heston_lower = heston_simulation(v0, kappa, theta, sigma, rho_1, S0 - price_shock, r,
heston_call_price_lower = heston_option_mc(S_heston_lower, S0 - price_shock, r, T, dt,
heston_put_price_lower = heston_option_mc(S_heston_lower, S0 - price_shock, r, T, dt, o

#Measuring Option Greek Delta:
heston_call_delta = (heston_call_price_higher - heston_eu_call_price) / price_shock
heston_put_delta = (heston_put_price_higher - heston_eu_put_price) / price_shock

#Measuring Greek Gamma:
heston_call_gamma = (heston_call_price_higher - 2*heston_eu_call_price + heston_call_pr
heston_put_gamma = (heston_put_price_higher - 2*heston_eu_put_price + heston_put_price_

pd.DataFrame(
    {"Heston European Call": [rho_1, heston_eu_call_price, heston_call_delta, heston_ca
   index=["Correlation Coefficicent","Option Price", "Greek Delta", 'Gamma']
).round(2)
```

```
<ipython-input-2-658fc93db2ce>:46: DeprecationWarning: `np.float` is a deprecated
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs
  covariance_matrix = np.zeros((2, 2), dtype=np.float)
<ipython-input-2-658fc93db2ce>:17: DeprecationWarning: `np.float` is a deprecated
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs
  v = np.zeros((M + 1, Ite), dtype=np.float)
```

	Heston European Call	Heston European Put
Correlation Coefficicent	-0.30	-0.30
Option Price	2.98	2.79
Greek Delta	-0.17	0.06
Gamma	-0.52	0.08

Question 6: Using the Heston Model, price an ATM European call and put, using a correlation value of -0.70.

```
np.random.seed(3892)
rho_2 = -0.7 # Change the correlation coefficicent to -0.70
#Generate price_shock with 1% increase in price of S0
price_shock = S0 * 0.01

#Pricing the base price of Options (Call and Put) with Heston dynamics:
S_heston = heston_simulation(v0, kappa, theta, sigma, rho_2, S0, r, T, M, Ite)
heston_call_price = heston_option_mc(S_heston, S0, r, T, dt, optype='C')
heston_put_price = heston_option_mc(S_heston, S0, r, T, dt, optype='P')

#Pricing the 101% price_shock price of Options (Call and Put) with Heston dynamics:
S_heston_higher = heston_simulation(v0, kappa, theta, sigma, rho_2, S0 + price_shock, r
heston_call_price_higher = heston_option_mc(S_heston_higher, S0 + price_shock, r, T, dt
heston_put_price_higher = heston_option_mc(S_heston_higher, S0 + price_shock, r, T, dt,

#Pricing the 99% price_shock price of Options (Call and Put) with Heston dynamics:
S_heston_lower = heston_simulation(v0, kappa, theta, sigma, rho_2, S0 - price_shock, r,
heston_call_price_lower = heston_option_mc(S_heston_lower, S0 - price_shock, r, T, dt,
heston_put_price_lower = heston_option_mc(S_heston_lower, S0 - price_shock, r, T, dt, o

#Measuring Option Greek Delta:
heston_call_delta = (heston_call_price_higher - heston_call_price) / price_shock
heston_put_delta = (heston_put_price_higher - heston_put_price) / price_shock

#Measuring Greek Gamma:
heston_call_gamma = (heston_call_price_higher - 2*heston_call_price + heston_call_price
heston_put_gamma = (heston_put_price_higher - 2*heston_put_price + heston_put_price_low

pd.DataFrame(
    {"Heston European Call": [rho_2, heston_call_price, heston_call_delta, heston_call_
   index=["Correlation Coefficicent","Option Price", "Greek Delta", 'Gamma']
).round(2)
```

```
<ipython-input-2-658fc93db2ce>:46: DeprecationWarning: `np.float` is a deprecated
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs
  covariance_matrix = np.zeros((2, 2), dtype=np.float)
<ipython-input-2-658fc93db2ce>:17: DeprecationWarning: `np.float` is a deprecated
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs
  v = np.zeros((M + 1, Ite), dtype=np.float)
```

	Heston European Call	Heston European Put
Correlation Coefficicent	-0.70	-0.70
Option Price	2.17	3.42
Greek Delta	-0.13	0.08

Gamma	-0.39	0.11

Question 7: Delta and Gamma for each of the options in Questions 5 and 6 have been integrated in the previous codes.

Part 2 - Question 8-10: Jump Diffusion Model - Merton

Team Member B: Jump Modeler

With μ = −0.5; δ = 0.22

The Merton model could be discretized as follows:

$$dS_t = (r - r_j)\, S_t dt + \sigma S_t dZ_t + J_t S_t dN_t$$
$$S_t = S_{t-1} * (e^{(r - r_j - \frac{\sigma^2}{2})dt + \sigma\sqrt{dt} z_t^1} + (e^{u_j + \delta z_t^2 - 1}) y_t)$$
$$Where: r_j = \lambda(e^{\mu_j + \frac{\delta^2}{2}}) - 1$$

```
np.random.seed(3892)
# Merton Modeler Inputs
S0 = 80
r = 0.055
sigma = 0.35
T = 3/12

mu = -0.5
delta = 0.22
M  = 50 # Total time steps
Ite = 100000 # Number of simulations

#Codes is modified from Module 7 Lesson Note 4:
def merton_mc(S0, sigma, mu, delta, lamb, M, Ite):
  SM = np.zeros((M + 1, Ite))
  SM[0] = S0
  dt = (T/M)

  # rj
  rj = lamb * (np.exp(mu + 0.5 * delta**2) - 1)

  # Random numbers
  np.random.seed(3892)
  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]
      )
      SM[t] = np.maximum(
          SM[t], 0.00001
      )  # To ensure that the price never goes below zero!
  return SM

def Merton_option_price(S, K, r, T, t, Opt):
  if Opt == "C":
    payoff = np.maximum(0, S[-1, :] - K)

  elif Opt == "P":
    payoff = np.maximum(0, K - S[-1, :])

  average = np.mean(payoff)

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

Question 8: Using the Merton Model, price an ATM European call and put with jump intensity parameter equal to 0.75.

```
np.random.seed(3892)
#Generate price_shock with 1% increase in price of S0
price_shock = S0 * 0.01

jump_inten_1 = 0.75  # Jump Intensitiy Parameter 0.75 - Served for Question 8

#Pricing the base price of Options (Call and Put) with Merton via Monte Carlo Methods:
S_Merton = merton_mc(S0, sigma, mu, delta, jump_inten_1, M, Ite)
Merton_eu_call_price = Merton_option_price(S_Merton, S0, r, T, 0, "C")
Merton_eu_put_price = Merton_option_price(S_Merton, S0, r, T, 0, "P")

#Pricing the 101% price_shock price of Options (Call and Put) with Merton via Monte Car
S_Merton_higher = merton_mc(S0 + price_shock, sigma, mu, delta, jump_inten_1, M, Ite)
Merton_call_price_higher = Merton_option_price(S_Merton_higher, S0 + price_shock, r, T,
Merton_put_price_higher = Merton_option_price(S_Merton_higher, S0 + price_shock, r, T,

#Pricing the 99% price_shock price of Options (Call and Put) with Merton via Monte Carl
S_Merton_lower = merton_mc(S0 - price_shock, sigma, mu, delta, jump_inten_1, M, Ite)
Merton_call_price_lower = Merton_option_price(S_Merton_lower, S0 - price_shock, r, T, 0
Merton_put_price_lower = Merton_option_price(S_Merton_lower, S0 - price_shock, r, T, 0,

#Measuring Option Greek Delta:
Call_Delta_Merton = (Merton_call_price_higher - Merton_eu_call_price) / price_shock
Put_Delta_Merton = (Merton_put_price_higher - Merton_eu_put_price) / price_shock
```

```
#Measuring Greek Gamma:
Call_Gamma_Merton = (Merton_call_price_higher - 2*Merton_eu_call_price + Merton_call_pr
Put_Gamma_Merton = (Merton_put_price_higher - 2*Merton_eu_put_price + Merton_put_price_

pd.DataFrame(
    {
      "Merton European Call": [jump_inten_1, Merton_eu_call_price, Call_Delta_Merton, C
      "Merton European Put": [jump_inten_1, Merton_eu_put_price, Put_Delta_Merton, Put_
   },
   index=["Jump Intensity","Option Price", "Greek Delta", "Gamma"]
  ).round(2)
```

	Merton European Call	Merton European Put
Jump Intensity	0.75	0.75
Option Price	8.32	7.26
Greek Delta	0.10	0.09
Gamma	0.00	0.00

Question 9: Using the Merton Model, price an ATM European call and put with jump intensity parameter equal to 0.25.

```
np.random.seed(3892)
#Generate price_shock with 1% increase in price of S0
price_shock = S0 * 0.01

jump_inten_2 = 0.25  # Jump Intensitiy Parameter 0.25 - Served for Question 9

#Pricing the base price of Options (Call and Put) with Merton via Monte Carlo Methods:
S_Merton = merton_mc(S0, sigma, mu, delta, jump_inten_2, M, Ite)
Merton_call_price = Merton_option_price(S_Merton, S0, r, T, 0, "C")
Merton_put_price = Merton_option_price(S_Merton, S0, r, T, 0, "P")

#Pricing the 101% price_shock price of Options (Call and Put) with Merton via Monte Car
S_Merton_higher = merton_mc(S0 + price_shock, sigma, mu, delta, jump_inten_2, M, Ite)
Merton_call_price_higher = Merton_option_price(S_Merton_higher, S0 + price_shock, r, T,
Merton_put_price_higher = Merton_option_price(S_Merton_higher, S0 + price_shock, r, T,

#Pricing the 99% price_shock price of Options (Call and Put) with Merton via Monte Carl
S_Merton_lower = merton_mc(S0 - price_shock, sigma, mu, delta, jump_inten_2, M, Ite)
Merton_call_price_lower = Merton_option_price(S_Merton_lower, S0 - price_shock, r, T, 0
Merton_put_price_lower = Merton_option_price(S_Merton_lower, S0 - price_shock, r, T, 0,

#Measuring Option Greek Delta:
Call_Delta_Merton = (Merton_call_price_higher - Merton_call_price) / price_shock
Put_Delta_Merton = (Merton_put_price_higher - Merton_put_price) / price_shock
```

```
#Measuring Greek Gamma:
Call_Gamma_Merton = (Merton_call_price_higher - 2*Merton_call_price + Merton_call_price
Put_Gamma_Merton = (Merton_put_price_higher - 2*Merton_put_price + Merton_put_price_low

pd.DataFrame(
    {
      "Merton European Call": [jump_inten_2, Merton_call_price, Call_Delta_Merton, Call
      "Merton European Put": [jump_inten_2, Merton_put_price, Put_Delta_Merton, Put_Gam
   },
   index=["Jump Intensity","Option Price", "Greek Delta", "Gamma"]
  ).round(2)
```

	Merton European Call	Merton European Put
Jump Intensity	0.25	0.25
Option Price	6.84	5.79
Greek Delta	0.09	0.07
Gamma	0.00	0.00

Question 10: Delta and Gamma for each of the options in Questions 8 and 9 have been integrated in the previous codes.

Step 2: All team members

Question 13: Repeat Questions 5 and 8 for the case of an American call option. Comment on the differences you observe from original Questions 5 and 8.

Heston model pricing for the case of an Amercian Call Option

```
np.random.seed(3892)
def american_payoff(S, K, optype):
    if optype == 'C':
      payoffs = np.maximum(0, S - K)
    elif optype == 'P':
      payoffs = np.maximum(0, K - S)
    else:
      raise ValueError()

    return payoffs
```

```
def heston_american_option_mc(S, K, r, T, dt, optype):
    payoffs = american_payoff(S[-1], K, optype)
    discounted_expected_payoff = np.zeros_like(S)
    discounted_expected_payoff[-1] = payoffs

    for i in range(discounted_expected_payoff.shape[0]-2, -1, -1):
      expected_payoff_hold = np.mean(discounted_expected_payoff[i+1] * np.exp(-r * dt))
      expected_payoff_exercise = np.mean(american_payoff(S[i], K, optype) * np.exp(-r *
      discounted_expected_payoff[i] = np.maximum(expected_payoff_exercise, expected_pay

    # calculate the Monte Carlo price of the American option
    monte_carlo_price = np.mean(discounted_expected_payoff[0] * np.exp(-r * dt))

    return monte_carlo_price

heston_ame_call_price = heston_american_option_mc(S_heston, S0, r, T, dt, optype='C')
heston_ame_put_price = heston_american_option_mc(S_heston, S0, r, T, dt, optype='P')

pd.DataFrame(
    {
      "Heston Amercian Call": [heston_ame_call_price], "Heston European Call": [heston_
   },
   index=["Option Price"]
).round(2)
```

Merton model pricing for the case of an Amercian Call Option

```
np.random.seed(3892)
def american_payoff(S, K, optype):
    if optype == 'C':
      payoffs = np.maximum(0, S - K)
    elif optype == 'P':
      payoffs = np.maximum(0, K - S)
    else:
      raise ValueError()

    return payoffs

def merton_american_option_mc(S, K, r, T, dt, optype):
    payoffs = american_payoff(S[-1], K, optype)
    discounted_expected_payoff = np.zeros_like(S)
    discounted_expected_payoff[-1] = payoffs
```

```
    for i in range(discounted_expected_payoff.shape[0]-2, -1, -1):
      expected_payoff_hold = np.mean(discounted_expected_payoff[i+1] * np.exp(-r * dt))
      expected_payoff_exercise = np.mean(american_payoff(S[i], K, optype) * np.exp(-r *
      discounted_expected_payoff[i] = np.maximum(expected_payoff_exercise, expected_pay

    # calculate the Monte Carlo price of the American option
    monte_carlo_price = np.mean(discounted_expected_payoff[0] * np.exp(-r * dt))

    return monte_carlo_price

merton_call_price = merton_american_option_mc(S_Merton, S0, r, T, 0, optype='C')
merton_put_price = merton_american_option_mc(S_Merton, S0, r, T, dt, optype='P')

pd.DataFrame(
    {
      "Merton Amercian Call": [merton_call_price], "Merton European Call " : [Merton_eu
   },
   index=["Option Price"]
).round(2)
```

Question 14: Using Heston model data from Question 6, price a European up-and-in call option (UAI) with a barrier level of 95 and a strike price of 95 as well. This UAI option becomes alive only if the stock price reaches (at some point before maturity) the barrier level (even if it ends below it).

```
#Set seed for the simulation
np.random.seed(3892)

UAI_barrier = 95.0 # Barrier level of the Eureopean UAI Call
K_barrier = 95.0 # Barrier of the strike price.

isin = np.ones([M])

# Generate the Heston model for the UAI Call
def uai_european_heston(S, K, r, T, t, M, uai_barrier_value, Opt):
  isin = np.ones([M+1])

  # find out if the stock price path stays below the barrier level
  isin = (S > uai_barrier_value).astype(float)

  # stock prices at the end of the paths and the payoffs
  if Opt == 'C':
    payoff = np.maximum(0, S[-1, :] - K)
```

```
      payoff = np.maximum(0, S[-1, :] - K)
    elif Opt == 'P':
      payoff = np.maximum(0, K - S[-1, :])

    # the option price at time 0 is the present value of the avg option price at expirati
    option_price0 = np.exp(-r * T) * payoff

    return np.mean(option_price0)

S_heston_barrier = heston_simulation(v0, kappa, theta, sigma, -0.7, S0, r, T, M, Ite) #
# print(S_heston_barrier)
European_Call_Simple = heston_option_mc(S_heston, K_barrier, r, T, dt, 'C')

European_Call_UAI = uai_european_heston(S_heston_barrier, K_barrier, r, T, dt, M, UAI_b

pd.DataFrame(
    {
      "Simple European Call": [European_Call_Simple],
      "Up-and-In European Call": [European_Call_UAI],
   },
index=["Option Price"]
).round(2)
```

Question 15: Using Merton model data from Question 8, price a European down-and-in put option (DAI) with a barrier level of 65 and a strike price of 65 as well. This UAO option becomes alive only if the stock price reaches (at some point before maturity) the barrier level (even if it ends above it).

```
np.random.seed(3892)
# Merton Modeler Inputs
S0 = 80
r = 0.055
sigma = 0.35
T = 3/12
```

```
mu = -0.5
delta = 0.22
M  = 50 # Total time steps
Ite = 100000 # Number of simulations
#Set seed for the simulation
np.random.seed(3892)

DAI_barrier = 65.0 # Barrier level of the Eureopean DAI Put
K_barrier = 65.0 # Barrier of the strike price.

isin = np.ones([M])

# Generate the Merton model for the DAI Put
def dai_european_merton(S, K, r, T, t, M, dai_barrier_value, Opt):
  isin = np.ones([M+1])

  # find out if the stock price path stays below the barrier level
  isin = (S < dai_barrier_value).astype(float)

  # stock prices at the end of the paths and the payoffs
  if Opt == 'C':
    payoff = np.maximum(0, S[-1, :] - K)
  elif Opt == 'P':
    payoff = np.maximum(0, K - S[-1, :])

  # the option price at time 0 is the present value of the avg option price at expirati
  option_price0 = np.exp(-r * T) * payoff

  return np.mean(option_price0)

S_merton_barrier = merton_mc(S0, sigma, mu, delta, 0.75, M, Ite) # With jump intensity
# print(S_merton_barrier)
European_Put_Simple = Merton_option_price(S_heston, K_barrier, r, T, dt, 'P')

European_Put_DAI = dai_european_merton(S_merton_barrier, K_barrier, r, T, dt, M, DAI_ba

pd.DataFrame(
    {
      "Simple European Put": [European_Put_Simple],
      "Down-and-In European Put": [European_Put_DAI],
   },
index=["Option Price"]
).round(2)
```