import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from scipy.stats import norm

#Part a) Combine GBM and Feller's logic to derive Heston:

# Parameters taken from table 1 in Cui et al. (2017):

T = 1 # maturity

S0 = 1 # spot price

K = 1.1 # strike price

r = 0.02 # risk-free interest rate

q = 0 # dividend rate

v0 = 0.08 # initial variance

rho = -0.8 # correlation between Brownian motions

kappa = 3 # mean reversion rate

theta = 0.1 # Long term mean of variance

sigma = 0.25 # volatility of volatility

n\_steps = 200 # number of time steps

n\_paths = 200000 # number of paths

dt = T/n\_steps # time step

t=np.arange(0,T,dt)

# Using Multivariate Normal to generate Correlated normal random variables:

W1, W2 = np.random.multivariate\_normal([0,0], [[1, rho], [rho, 1]], (n\_steps, n\_paths)).T

# Creating an empty array for the variance:

v = np.zeros((n\_steps,n\_paths)).T

# indexing the initial variance to the first point in every path:

v[:,0] = v0

# Creating an empty array for the asset prices:

S = np.zeros((n\_steps,n\_paths)).T

# indexing the initial spot price to the first point in every path:

S[:,0] = S0

# Compute the each step and path: Euler-Maruyama:

a = (sigma\*\*2)/kappa\*(np.exp(-kappa\*dt)-np.exp(-2\*kappa\*dt)) #with analytic moments

b = theta\*(sigma\*\*2)/(2\*kappa)\*(1-np.exp(-kappa\*dt))\*\*2 #with analytic moments

for i in range(n\_steps-1): #compute and accumulate the increments

#v[:,i+1] = v[:,i] + kappa\*(theta - v[:,i])\*dt + sigma\*np.sqrt(v[:,i]\*dt)\*W1[:,i] #plain

v[:,i+1] = theta + (v[:,i]-theta)\*np.exp(-kappa\*dt) + np.sqrt((a\*v[:,i]+b))\*W1[:,i] #with analytic moments

muABM = r - q - 0.5\*v[:,i] #similar to GBM's script, we can define this calculation as muABM

S[:,i+1] = S[:,i] \* np.exp(muABM\*dt + np.sqrt(v[:,i]\*dt)\*W2[:,i])

EX= S0 \*np.exp(r-q-0.5\*t\*v0) #the expected path would be similar to GBM but taking muABM as mu.

df=pd.concat([pd.Series(x) for x in S],axis=1) #convert the array in a dataframe

mean\_path=df.apply(lambda row: np.mean(row),axis=1) #use lambda to generate the mean of each timestep

for k in range(100): #apply a for loop to plot all paths but limit it to 100 paths (faster)

plt.plot(t,S[k])

plt.plot(t,mean\_path,'k',t,EX,':k') #plot both the expected path and the mean path, there is some divergence.

plt.legend(['Derived Paths','Mean Path','Expected Path'])

plt.ylabel('Asset Prices')

plt.xlabel('Timesteps')

plt.title('Heston derived model via Feller and GBM')

plt.show()

nbins=100

try:

fig, axs = plt.subplots(3,sharex=True)

fig.suptitle('Generated Heston PDF at different times using numerical data')

axs[0].hist(S[:,20],nbins,density=True)

axs[1].hist(S[:,80],bins=nbins,density=True)

axs[2].hist(S[:,120],bins=nbins,density=True)

axs[3].hist(S[:,180],bins=nbins,density=True)

except IndexError:

print('It will chart the first 3 suplots')

# Part b) Add code to price call and put European options using the model above: IT WILL TAKE AROUND 50-60 SECONDS TO LOAD DUE TO NUMBER BLOCKS:

# Pricing:

nblocks = 2000 # number of blocks

npaths = 1000 # number of paths

nsteps = 200 # number of steps

# Create an empty array for both options:

VcMCb = np.zeros(nblocks) # call array

VpMCb = np.zeros(nblocks) # put array

for j in range(nblocks):

# Using Multivariate Normal to generate Correlated normal random variables

X1, X2 = np.random.multivariate\_normal([0,0], [[1, rho], [rho, 1]], (nsteps, npaths)).T

# Creating an empty array for the variance

v = np.zeros((nsteps, npaths)).T

# indexing the initial variance to the first point in every path

v[:,0] = v0

# Creating an empty array for the asset price:

S = np.zeros((nsteps, npaths)).T

# indexing the initial spot price to the first point in every path

S[:,0] = S0

# Compute the each step and path: Euler-Maruyama:

a = (sigma\*\*2)/kappa\*(np.exp(-kappa\*dt)-np.exp(-2\*kappa\*dt)) #with analytic moments

b = theta\*(sigma\*\*2)/(2\*kappa)\*(1-np.exp(-kappa\*dt))\*\*2 #with analytic moments

for l in range(nsteps-1):

#v[:,l+1] = v[:,l] + kappa\*(theta - v[:,l])\*dt + sigma\*np.sqrt(v[:,l]\*dt)\*X1[:,l] #plain

v[:,l+1] = np.abs(theta + (v[:,l]-theta)\*np.exp(-kappa\*dt) + np.sqrt((a\*v[:,l]+b))\*X1[:,l]) #with analytic moments

muABM = r - q - 0.5\*v[:,l]

S[:,l+1] = S[:,l] \* np.exp((muABM)\*dt + np.sqrt(v[:,l])\*np.sqrt(dt)\*X2[:,l])

# Calculate the discounted option price for each nblock

VcMCb[j] = np.exp(-r\*T)\*np.mean(np.maximum(S[:,-1] - K, 0))

VpMCb[j] = np.exp(-r\*T)\*np.mean(np.maximum(K - S[:,-1], 0))

# Derive the final option price (its the mean of all payoffs in each block)

VcMC = np.mean(VcMCb)

VpMC = np.mean(VpMCb)

#Compare the code above to the Analytical solution:

S\_0=1

muABM=r-q-0.5\*sigma\*\*2

d2=(np.log(S\_0/K)+muABM\*T)/(sigma\*np.sqrt(T))

d1=d2+sigma\*np.sqrt(T)

Vca=S\_0\*np.exp(-q\*T)\*norm.cdf(d1,0,1)-K\*np.exp(-r\*T)\*norm.cdf(d2,0,1)

Vpa=(K\*np.exp(-r\*T)\*norm.cdf(-d2,0,1))-(S\_0\*np.exp(-q\*T)\*norm.cdf(-d1,0,1))

put\_call\_parity=Vca+K\*np.exp(-1\*(r-q)\*T)-S\_0

deviation=((put\_call\_parity/Vpa)-1)\*100

print('Heston call without calibration option price : ' + str(round(VcMC,4)))

print('Analytical put option price : ' + str(round(Vca,4)))

print('Deviation of call across methods % '+str(round(((VcMC/Vca)-1),6)\*100))

print('Heston out without calibration option price : ' + str(round(VpMC,4)))

print('Analytical put option price : ' + str(round(Vpa,4)))

print('Deviation of put across methods % '+str(round(((VpMC/Vpa)-1),6)\*100))