JUMP PROCESSES

FINAL PROJECT

Paul-Louis CABAL
Thomas JOUVE
Théo MIDAVAINE
Matthieu JULIEN

ESILV A5 IF 2023-2024

Contents

1 Question 1

Using the code for the Merton model as a blueprint, implement the Monte Carlo simulation of the Kou jump diffusion model.

We recall that in the Kou jump diffusion model, the jumps are identically and independently distributed double exponentials. Therefore, we are mixing two independent exponential random variables of parameters λ_+ and λ_- using a Bernouilli random variable of parameter p :

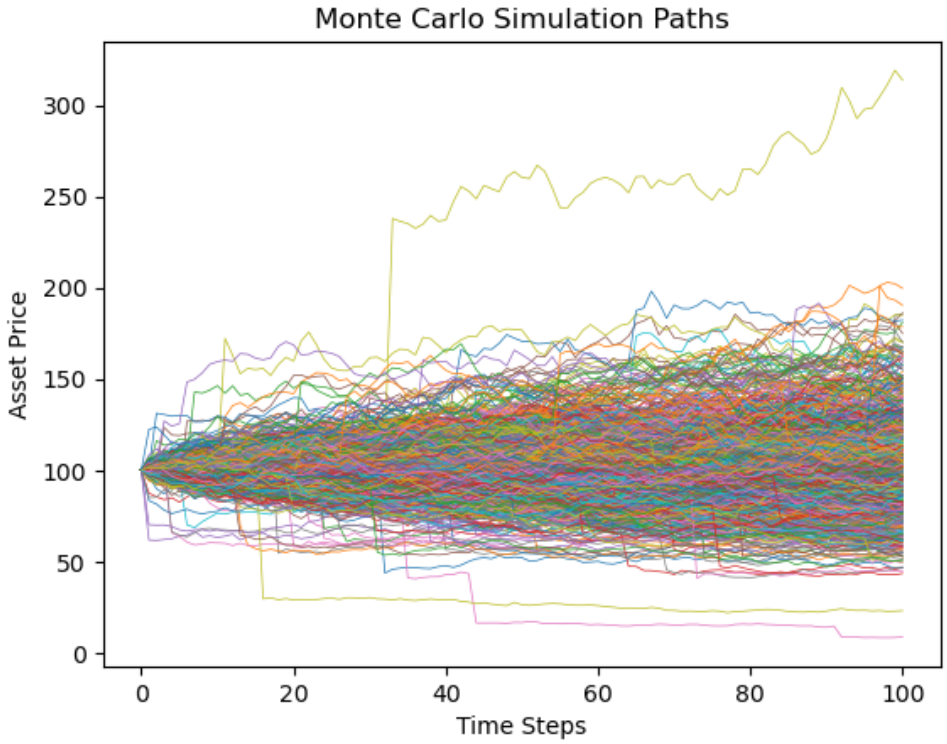
$$\xi_i = \begin{cases} X^+ & \text{with probability } p \\ -X^- & \text{with probability } 1-p \end{cases}$$

where X+ follows Exp(λ_+) and X- follows Exp(λ_-)

In order to simulate the jump size in every grid interval, we loop on the number of Poisson events:

```
def MonteCarloSimulation_Kou(simN, stepsN, S0, T, r, q, params):

    np.random.seed(123)
    sigma, lam, p, lam_plus, lam_minus = params
    dt = T / stepsN
    price = np.ones((simN, stepsN + 1)) * S0
    jump_compensator = (lam * (p * lam_plus / (lam_plus - 1) + (1 - p) * lam_minus / (1 + lam_minus) -
    ↪  1))
    drift = (r - q - sigma ** 2 / 2 - jump_compensator) * dt * np.ones((simN, stepsN))

    dWt = np.random.normal(0, 1, (simN, stepsN))
    diffusion = sigma * dWt * np.sqrt(dt)

    jump_N = si.stats.poisson.rvs(lam * dt, size=(simN, stepsN))
    jumps = np.zeros((simN, stepsN))

    for i in range(simN):
        for j in range(stepsN):
            for k in range(jump_N[i, j]):
                U = np.random.rand(1)
                jumps[i, j] += np.random.exponential(1 / lam_plus) * (U < p) - np.random.exponential(1 /
                ↪  lam_minus) * (U > p)

    price[:, 1:] = S0 * np.exp(np.cumsum(drift + diffusion + jumps, axis=1))

    mart = np.exp(-r * T) * np.mean(price[:, -1])

    return price
```

We can plot our paths :



Then, we implement the Lewis Integral option pricing formula for the model.
In order to do that, we first define our characteristic exponent :

```
def cgmy(params, z):
    C, G, M, Y = params
    return C * si.special.gamma(-Y) * ((M - 1j * z) ** Y - M ** Y + (G + 1j * z) ** Y - G ** Y)

def merton(params, z):
    sigma, lam, mu, delta = params
    return lam * (np.exp(1j * z * mu - 0.5 * delta**2 * z**2) - 1) -0.5 * (z**2) * sigma**2

def nig(params, z):
    kappa, sigma, mu = params
    return (1 - np.sqrt(1 + z**2 * sigma**2 * kappa - 2 * 1j * mu * z * kappa)) / kappa

def characteristic_exponent(model, params, z):

    functions = {
        'CGMY': cgmy,
        'MERTON': merton,
        'NIG': nig,
    }

    if model not in functions:
        raise ValueError('Characteristic exponent not available')
```

```
    return functions[model](params, z)
```

Then, the characteristic function :

```
def characteristic_function(z, S0, T, r, q, params, model):

    if model in {'NIG', 'MERTON', 'CGMY'}:
        ce = T * (characteristic_exponent(model, params, z) -
                  1j * z * characteristic_exponent(model, params, -1j))
        cf = np.exp(1j * z * (np.log(S0) + T * (r - q)) + ce)

    elif model == 'HESTON':
        v0, kappa, theta, eta, rho = params
        d = np.sqrt((rho * eta * 1j * z - kappa)**2 + eta**2 * (1j * z + z**2))
        g = (kappa - rho * eta * 1j * z - d) / (kappa - rho * eta * 1j * z + d)
        A = theta * kappa / (eta**2) * ((kappa - rho * eta * 1j * z - d) * T - 2 * np.log((1 - g *
        ↪  np.exp(-d * T)) / (1 - g)))
        B = v0 / eta**2 * (kappa - rho * eta * 1j * z - d) * (1 - np.exp(-d * T)) / (1 - g * np.exp(-d *
        ↪  T))
        cf = np.exp(1j * z * (np.log(S0) + (r - q) * T) + A + B)

    else:
        raise ValueError('Characteristic function not available')

    return cf
```

We can now calculate the Lewis integral to price a call option as follows:

```
def Lewis_Pricing(S0, K, T, r, q, params, trunc, im, model):

    def integrand(om, S0, K, T, r, q, params, im, model):
        i = 1j
        z = om + i * im
        return characteristic_function(-z, S0, T, r, q, params, model) * K ** (1 + i * z) / (i * z - z **
        ↪  2)

    call, _ = si.integrate.quad(integrand, -trunc, trunc, args=(S0, K, T, r, q, params, im, model))

    call *= 0.5 * np.exp(-r * T) / np.pi
    call = abs(call)

    return call
```

2 Question 2

Implement the Monte Carlo simulation of a Bilateral Gamma option pricing model (difference of two Gamma subordinators) and as a particular case reproduce the Variance Gamma model as originally introduced as the exponential of a subordinated Brownian motion.

First we solve to find the values for θ_+ and θ_-, such as :

$$(\theta_+\theta_-)^{-1} = \sigma^2\kappa/2, \qquad \theta_+^{-1} - \theta_-^{-1} = \kappa\mu.$$

For that, we set x = θ_+ and y = θ_- and develop the system, to get this following system to solve :

$$\begin{cases} x \cdot y = \dfrac{2}{\sigma^2 \cdot \kappa} \\ y - x = \dfrac{2 \cdot \mu}{\sigma^2} \end{cases} \tag{1}$$

We obtain :

$$\theta+ : \left(\frac{-\kappa\mu - \sqrt{\kappa(\kappa\mu^2 + 2\sigma^2)}}{\kappa\sigma^2}, \frac{\kappa\mu - \sqrt{\kappa(\kappa\mu^2 + 2\sigma^2)}}{\kappa\sigma^2} \right)$$

$$\theta- : \left(\frac{-\kappa\mu + \sqrt{\kappa(\kappa\mu^2 + 2\sigma^2)}}{\kappa\sigma^2}, \frac{\kappa\mu + \sqrt{\kappa(\kappa\mu^2 + 2\sigma^2)}}{\kappa\sigma^2} \right)$$

We test our obtained solutions, by calculating the following with random values of sigma, kappa and mu to expect have a result of 0 :

$$\begin{cases} x \cdot y - \dfrac{2}{\sigma^2 \cdot \kappa} \\ y - x - \dfrac{2 \cdot \mu}{\sigma^2} \end{cases} \tag{2}$$

We can test our set of solutions to see if we get 0

```
Entrée [144]: (solutions[1][1] - solutions[1][0] - 2*mu/sigma**2).subs({sigma: 0.2, kappa: 0.1, mu: 0.05})
  Out[144]: 0

Entrée [145]: (solutions[1][1] - solutions[1][0] - 2*mu/sigma**2).subs({sigma: 0.2, kappa: 0.1, mu: 0.05})
  Out[145]: 0

Entrée [146]: round((solutions[0][1] * solutions[0][0]- 2/(kappa*sigma**2)).subs({sigma: 0.2, kappa: 0.1, mu: 0.05}))
  Out[146]: 0

Entrée [147]: round((solutions[1][1] * solutions[1][0]- 2/(kappa*sigma**2)).subs({sigma: 0.2, kappa: 0.1, mu: 0.05}))
  Out[147]: 0
```

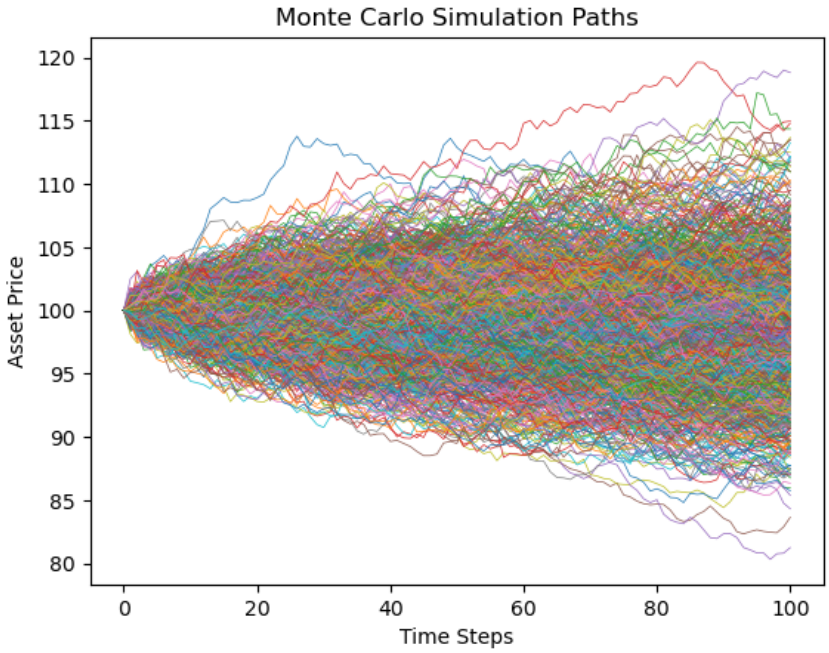
Thus, our set of solution for (x,y) are correct. Since we want positive values for theta we will be choosing the set of thetas accordingly. For sigma = 0.2, kappa = 1, mu = 0.05, we get :

$\theta_+ = 5.93070330817254$
$\theta_- = 8.43070330817254$

We implement the Monte Carlo simulation of a Bilateral Gamma option pricing model :

```
def valorisation_Monte_Carlo(S0, r, nb_simulations, nb_steps, strike, T, theta_plus, theta_minus):
    dt = T / nb_steps
    paths = np.zeros((nb_simulations, nb_steps + 1))
    paths[:,0] = S0

    for i in range(nb_steps):
        # Generate two Gamma processes for theta_plus and theta_minus
        gamma_plus = np.random.gamma(scale=dt/theta_plus, shape=theta_plus, size=nb_simulations)
        gamma_minus = np.random.gamma(scale=dt/theta_minus, shape=theta_minus, size=nb_simulations)

        #Variance-Gamma model can be written as the difference of two independent gamma processes:
        VG = gamma_plus - gamma_minus
        paths[:,i+1] = paths[:, i] * np.exp(VG - r*dt)


    option_payoffs = np.maximum(paths[:,-1] - strike, 0)
    option_value = np.exp(-r * T) * np.mean(option_payoffs)
    print(option_value)
    return option_value,paths
```

We can plot the paths :



3 Question 3

Using the acceptance-rejection code (AR) for simulating positive tempered stable random variables, implement an exact Monte Carlo simulator for the CGMY model with (0, 1). A CGMY process in that case will be just a difference of Tempered Stable subordinators. Follow the papers of Kawai and Masuda (2011) and Zhang and Zhang (2018). Beware of the many possible different parametrizations when using a stable random number generator.

The CGMY Model and Jump Processes

This model is characterized by four parameters that allow great flexibility in modeling the distribution of returns. The Y parameter controls the thickness of the distribution's tails, while the C, G, and M parameters adjust the intensity and direction of jumps.
The CGMY model is an example of a jump process, which allows modeling abrupt changes (or jumps) in the price of an asset, unlike diffusion models that assume a continuous evolution of prices. Jumps are often used to model rare but influential events, such as significant economic announcements or financial crises.

The code defines a function CGMY_Final_MC that simulates price paths of an asset under the CGMY model over a given period. Here is a detailed explanation of each part of the code:

Function Parameters:
simN: The number of simulations to perform.
S0: The initial price of the asset.
T: The total duration of the simulation.
r: The risk-free interest rate.
q: The dividend yield of the asset.
params: The CGMY model parameters (C, G, M, Y).
time_steps: The number of time steps in the simulation.

Calculation of Lambda and Compensator:

lambda_Y and compensator are calculated from the CGMY parameters. These values are used to adjust the jumps in the simulation and to compensate the mean so that the model is a martingale process.

Initialization of Price Paths:

Price paths are initialized in a NumPy array. The first element of each path is the initial price S0.

Simulation Loop:

For each time step, the code generates positive (jump_plus) and negative (jump_minus) jump sizes for each simulation. The generation of jumps uses the acceptance-rejection algorithm to simulate positive tempered stable random variables. This involves rejecting some generated values until a condition based on the M and G parameters is satisfied. The generated jumps are then used to update the price paths.

Updating Price Paths:

Price paths are updated by multiplying the price at the previous step by an exponential factor that accounts for the risk-free rate, the dividend yield, the compensator, and the difference between positive and negative jumps.

Output:

The function returns an array of the simulated price paths.

Here is the code

```
def CGMY_Final_MC(simN, S0, T, r, q, params, time_steps):
    C, G, M, Y = params

    # Calculate lambda and compensator
    lambda_Y = (C * np.math.gamma(1 - Y) / Y * np.cos(np.pi * Y / 2) * T) ** (1 / Y)
    compensator = C * np.math.gamma(-Y) * ((M - 1) ** Y - M ** Y + (G + 1) ** Y - G ** Y) * T

    # Initialize arrays
    price_paths = np.zeros((simN, time_steps))
    price_paths[:, 0] = S0

    dt = T / (time_steps - 1)

    np.random.seed(123)  # for reproducibility

    for i in range(1, time_steps):
        # Generate jump sizes
        jump_plus = np.zeros(simN)
        jump_minus = np.zeros(simN)

        for j in range(simN):
            accepted = False
            while not accepted:
                jump = lambda_Y * levy_stable.rvs(Y, 1, loc=0, scale=1, size=1)
                U = np.random.rand()
                if np.exp(-M * jump) > U:
                    jump_plus[j] = jump
                    accepted = True

            accepted = False
            while not accepted:
                jump = lambda_Y * levy_stable.rvs(Y, 1, loc=0, scale=1, size=1)
                V = np.random.rand()
                if np.exp(-G * jump) > V:
                    jump_minus[j] = jump
                    accepted = True

        # Debug output because had issues with jump values
        if i % 10 == 0:  # Print debug information every 10 steps
            print(f"Time step {i}, sample jump_plus: {jump_plus[:5]}, jump_minus: {jump_minus[:5]}")

        # Compute price paths
        price_paths[:, i] = price_paths[:, i-1] * np.exp((r - q - compensator) * dt + jump_plus -
        ↪  jump_minus)

        # Debug output to check for non-positive prices
        if (price_paths[:, i] <= 0).any():
            print(f"Non-positive prices found at time step {i}")
            break

    return price_paths
```

To see if it generates a coherent path and returns, we plot the result with basic parameters. But to obtain coherent path prices and to study the return, we must calibrate the model. For this, we need the historical log return (for example) to calibrate it. Now we will try to see with arbitrary values what we have and with the values given in the Zhang and Zhang paper.

```
final_prices = price_paths[:, -1]
# Exclude non-positive final prices before taking the logarithm
positive_final_prices = final_prices[final_prices > 0]
if len(positive_final_prices) == 0:
raise ValueError("All final prices are non-positive, check the model parameters and simulation.")

max_price_path_value = np.max(price_paths)
min_price_path_value = np.min(price_paths)

print(f"Maximum price path value: {max_price_path_value}")
print(f"Minimum price path value: {min_price_path_value}")
if max_price_path_value > 1e100 or min_price_path_value < 1e-100:
    print("Warning: Extreme values detected in price paths.")

# Plot a subset of paths
plt.figure(figsize=(10, 6))
for path in price_paths[:10, :]:  # Plot only the first 10 paths for clarity
    plt.plot(time_grid, path)
plt.yscale('log')  # Log scale for better visibility in case of extreme values
plt.xlabel('Time')
plt.ylabel('Price')
plt.title('CGMY Price Paths')
plt.show()

# Calculate log-prices with a filter for positive final prices
log_prices = np.log(final_prices[final_prices > 0] / S0)

# Plot histogram within a specified range to avoid extreme values
plt.hist(log_prices, bins=50, range=[np.percentile(log_prices, 1), np.percentile(log_prices, 99)],
↪  alpha=0.75)
plt.xlabel('Log-Price')
plt.ylabel('Frequency')
plt.title('Frequency Histogram of Log-Prices')
plt.show()

# Calculate log returns
log_returns = np.log(final_prices / S0)
plt.hist(log_returns, bins=50, density=True, alpha=0.75)
plt.title('Returns CGMY model T=1')
plt.xlabel('Log return')
plt.ylabel('Density')
plt.show()
```

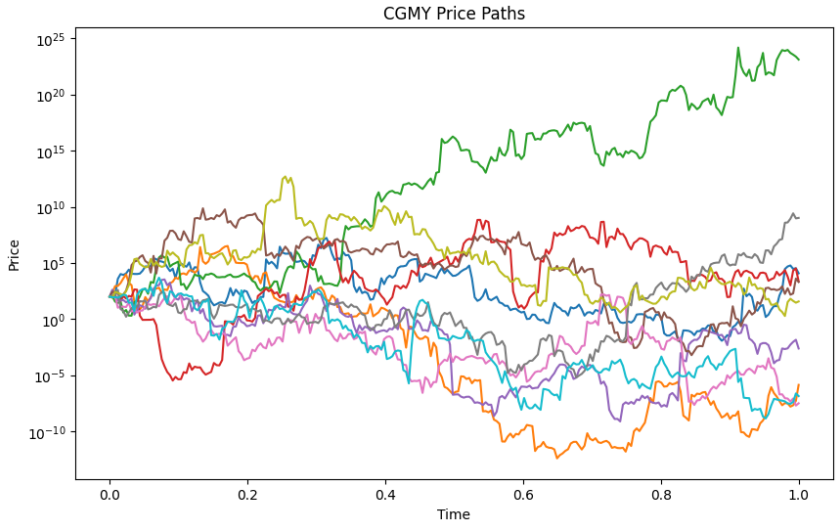
And here are the results with different parameters :

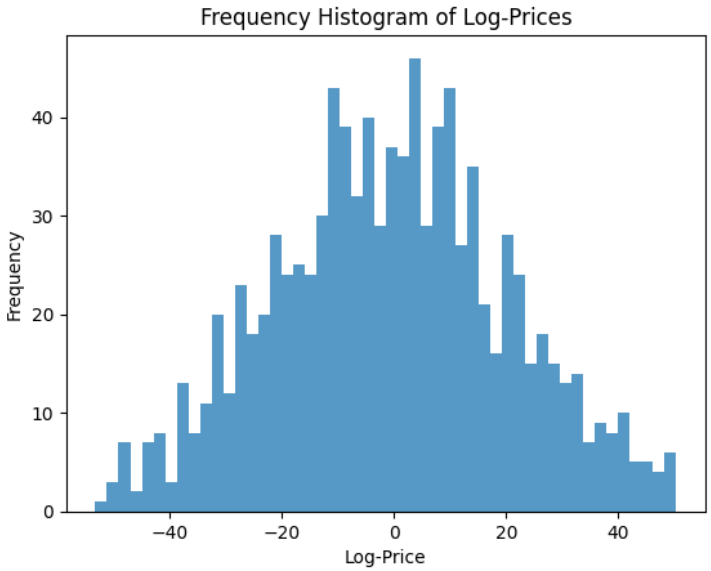
simN = 1000

S0 = 100

T = 1

r = 0.01
q = 0.02
time_steps = 252





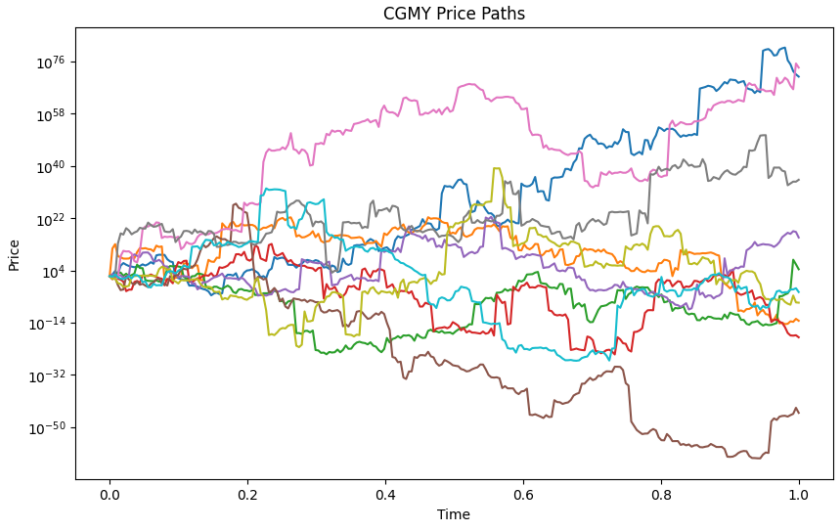
First Graph (Parameters: C=0.5, G=0.5, M=0.5, Y=0.5)

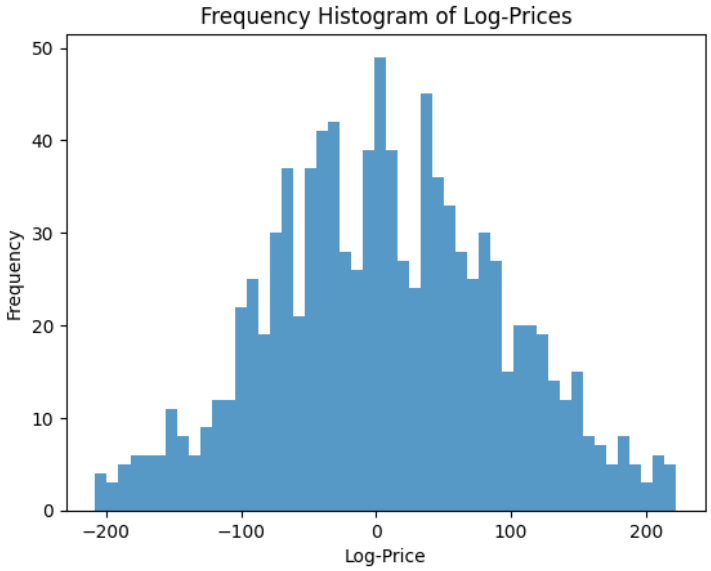
Volatility and Jumps: With all parameters set to 0.5, the price paths exhibit moderate levels of volatility and jump intensity. This is due to the relatively balanced effect of the G (negative jumps) and M (positive jumps) parameters, along with the modest impact of the C parameter, which scales the overall jump intensity.

Tail Behavior: Since the Y parameter, which controls the tail heaviness, is at 0.5, the tails of the distribution are not too heavy or light. This results in a distribution of jumps that is less likely to produce extreme values than in a distribution with heavier tails.

Overall Price Dynamics: The price paths show some spikes and drops, indicative of jumps, but remain relatively stable, suggesting the tempered nature of the stable distribution influenced by the parameters.

With Zhang Zhang's conclusion in the 6.2 part of their paper : C, G, M, Y = 0.42, 4.37, 191.2, 1.0102





Comparison and Interpretation

When comparing the two graphs, it is evident that the second graph shows a much more volatile and unpredictable set of price paths, with some paths reaching extremely high or low values. The first graph shows more controlled dynamics with less extreme variations in prices.

The CGMY model's flexibility in capturing various types of price behaviors is clearly demonstrated by these two sets of parameters. The first set results in a more stable price evolution, whereas the second set, with its extreme M value and slightly higher Y, showcases the model's ability to generate paths with significant jumps

and heavy tails, which could represent a market under stress or exhibiting high levels of speculation.

Trying the implementation of the double rejection algorithm of the Zhang Zhang's paper :

Explanation of the Double Rejection Algorithm:
The double rejection algorithm is an advanced simulation technique used for generating random variables from complex distributions that are difficult to sample from directly. It builds upon the standard acceptance-rejection method by introducing a second layer of rejection, hence the term "double rejection."

The primary advantage of the double rejection method is that it can improve the efficiency of the simulation process. In scenarios where the probability of accepting a sample in the standard rejection method is low, the double rejection method can be particularly beneficial. This method employs two different acceptance criteria, which together ensure that the generated samples conform more closely to the target distribution, thereby potentially reducing the number of rejected samples and improving computational efficiency.

Here is how we understand it and made a pseudo code :

```
Function GenerateCGMYIncrement:
repeat:
    repeat:
        Generate V, W from a uniform distribution over [0,1]
        Set U according to some criteria based on V, W, and given parameters
        Generate a new random variable W_uniform from a uniform distribution over [0,1]
        Calculate acceptance probabilities , , z, and  based on U and other parameters
    until U is less than  and W_uniform times  is less than or equal to 1

    Calculate variables a, m, , a1, a2, a3, and s based on the acceptance probabilities
    Generate another variable V_prime from a uniform distribution over [0,1]
    Determine X_prime based on V_prime and the calculated variables
    Generate an exponential random variable E
    until X_prime is greater than or equal to 0 and meets the acceptance condition based on a, m, , E

    return the generated value ^(1/Y) / X_prime^b
End Function
```

And we tried to implement it in Python but didn't succeed :

```
def A(u, Y):
    if u == 0:
        return 1
    else:
        return ((np.sin(Y * u)) ** Y * (np.sin((1 - Y) * u)) ** (1 - Y)) / (np.sin(u) ** Y)

def B(x, T, Y):

    if x == 0:
        return (Y - 1) ** Y * Y ** (-Y)
    else:
        return A(x, Y) * (x ** (1 - Y))
```

```
def double_rejection(C, G, M, Y, T):
    # Formula of the paper

    lambda_Y = C * T * ((M - 1) ** Y - M ** Y + (G + 1) ** Y - G ** Y)
    gamma = M ** Y + G ** Y - (M ** Y) * (G ** Y)
    y1 = -np.pi / 2 * Y
    y2 = np.pi / 2 * (1 - Y)
    u1 = y2 / (y1 + y2)
    u2 = y1 / (y1 + y2)
    u3 = np.pi
    b = (1 - Y) / Y
    c = np.pi ** (2 * Y) / np.sin(np.pi * Y)

    while True:
        # Step 1: Generate V and W' uniformly on [0, 1]
        V = np.random.uniform(0, 1)
        W_prime = np.random.uniform(0, 1)


        print(V, W_prime)
        x
        return None
C = 1.0
G = 5.0
M = 10.0
Y = 0.5
T = 1.0
sample = double_rejection(C, G, M, Y, T)
print(sample)
```

To conclude this question, we understand why the double rejection method can be usefull :

The double rejection method is advantageous because it can handle cases where the probability density function (PDF) of the desired distribution is complex and does not lend itself to simple analytical inversion or other straightforward simulation methods. By applying a second layer of rejection, the algorithm can more finely tune the sampling process to closely match the target distribution, thereby improving the quality of the samples and often reducing the computational time required to generate a set of samples that are representative of the desired distribution.

Moreover, the double rejection method can be particularly useful when dealing with heavy-tailed distributions like the CGMY model, where the tails of the distribution play a critical role in the behavior of the process being modeled, such as financial asset returns that exhibit jumps or sudden large changes in value.

4 Question 4

Explicitly calculate the characteristic function of the BNS model in the case of Z being a CPP process with exponential jumps.

```
    def characteristic_function_BNS2(z, S, T, r, q, params):
      alpha, beta, rho, _lambda, sigma = params
      f1 = 1j * z * rho - (1 / _lambda) * (z ** 2 + 1j * z) * (1 - np.exp(-_lambda * T)) / 2
      f2 = 1j * z * rho - (1 / _lambda) * (z ** 2 + 1j * z) / 2


      cf1 = 1j * z * (np.log(S) + (r - q - alpha * _lambda * rho * (1 / (beta - rho))) * T)
      cf2 = (1 / _lambda) * (z ** 2 + 1j * z) * (1 - np.exp(-_lambda * T)) * (sigma ** 2) / 2
      cf3 = alpha * (1 / (beta - f2)) * (beta * np.log((beta - f1) / (beta - 1j * z * rho)) + f2 *
      ↪  _lambda * (-T))

      temp = cf1 - cf2 + cf3
      cf_result = np.exp(temp)
      return cf_result
```

Use this formula to implement the Lewis integral option pricing formula for such model.

```
    def CallPricingLewis_eta(S0, K, T, r, q, params, trunc, im):
        def integrand(om, S0, K, T, r, q, params, im):
            i = 1j
            z = om + i * im
        return characteristic_function_BNS2(-z, S0, T, r, q, params) * (K ** (1 + i * z) / (i * z - z **
        ↪  2)) * np.exp(-1j * z * (np.log(S0 + (r - q)*T)))

    # Use quad for numerical integration
    call, _ = si2.quad(integrand, -trunc, trunc, args=(S0, K, T, r, q, params, im))

    call *= 0.5 * np.exp(-r * T) / np.pi
    call = abs(call)

    return call


params = [2.9, 0.2, 0.8, 0.08, 0.5]  #alpha, beta, rho, _lambda, sigma
trunc = np.inf  # Numerical truncation value
im = 0.5  # Imaginary part for the characteristic function
model = 'BNS'  # The model used for the characteristic function

# Spot price, strike price, time to maturity, risk-free rate, and dividend yield
S0 = 100
K = 100
T = 1
r = 0.02
q = 0.00

call_price_BNS = CallPricingLewis_eta(S0, K, T, r, q, params, trunc, im)
call_price_BNS
```

Change then to a Lévy subordinator Z of infinite activity, and implement the pricing integral for the corresponding BNS model.

To include a Lévy subordinator Z of infinite activity in the characteristic function for option pricing within

the Bates-Neumann-Shephard (BNS) model context in Python, we'll adapt the characteristic function. This adaptation will involve adding a term to represent the Lévy subordinator's effect, focusing on capturing the infinite activity aspect.

```
import numpy as np
import scipy.integrate as si

def characteristic_function_BNS2_with_Levy(z, S, T, r, q, params):
    alpha, beta, rho, _lambda, sigma = params
    f1 = 1j * z * rho - (1 / _lambda) * (z ** 2 + 1j * z) * (1 - np.exp(-_lambda * T)) / 2
    f2 = 1j * z * rho - (1 / _lambda) * (z ** 2 + 1j * z) / 2

    cf1 = 1j * z * (np.log(S) + (r - q - alpha * _lambda * rho * (1 / (beta - rho))) * T)
    cf2 = (1 / _lambda) * (z ** 2 + 1j * z) * (1 - np.exp(-_lambda * T)) * (sigma ** 2) / 2
    cf3 = alpha * (1 / (beta - f2)) * (beta * np.log((beta - f1) / (beta - 1j * z * rho)) + f2 * _lambda
    ↪  * (-T))

    gamma = 0.1  # Drift parameter for the Lévy subordinator
    delta = 0.2  # Volatility parameter for the continuous part of the Lévy subordinator

    levy_effect = T * (1j * z * gamma - 0.5 * delta ** 2 * z ** 2)

    temp = cf1 - cf2 + cf3 + levy_effect
    cf_result = np.exp(temp)
    return cf_result

def CallPricingLewis_eta_with_Levy(S0, K, T, r, q, params, trunc, im):
    def integrand(om, S0, K, T, r, q, params, im):
        i = 1j
        z = om + i * im
        return characteristic_function_BNS2_with_Levy(-z, S0, T, r, q, params) * (K ** (1 + i * z) / (i *
        ↪  z - z ** 2)) * np.exp(-1j * z * (np.log(S0 + (r - q)*T)))

    # Numerical integration over the specified range
    call, _ = si.quad(integrand, -trunc, trunc, args=(S0, K, T, r, q, params, im))

    # Final adjustment to the call price formula
    call *= 0.5 * np.exp(-r * T) / np.pi
    call = abs(call)  # Ensure the call price is non-negative
    return call

# Parameters and inputs for the call price calculation
params_levy_sub = [2.8, 0.2, 0.8, 0.08, 0.5]  # alpha, beta, rho, _lambda, sigma
trunc = 100  # Large truncation value for numerical integration
im = 0.5  # Imaginary part for the characteristic function adjustment
S0, K, T, r, q = 100, 100, 1, 0.05, 0.00  # Spot, strike, maturity, rate, and yield

# Calculate the call price using the modified characteristic function
call_price_with_levy = CallPricingLewis_eta_with_Levy(S0, K, T, r, q, params_levy_sub, trunc, im)
print(f"Call Price with Levy Subordinator: {call_price_with_levy}")
```

Result : Call Price with Levy Subordinator: 12.58

(Not asked) We also implemented a Variance Gamma Process to compare with our BNS Levy Subordinator.

The Variance Gamma (VG) process, with its capability to model financial asset returns incorporating jumps, skewness, and kurtosis, appears to us well-suited for scenarios involving infinite activity.

```
# Define a function to simulate asset paths using the Variance Gamma process
def simulate_vg_option_price(S0, K, T, r, q, sigma, theta, kappa, nu, num_paths, num_steps):
    dt = T / num_steps  # Time step
    # The Variance Gamma process requires simulating a Brownian motion (dW)
    # and a Gamma process (dG)
    dW = np.random.normal(0, np.sqrt(dt), size=(num_paths, num_steps))
    dG = np.random.gamma(dt / nu, nu, size=(num_paths, num_steps))

    # Adjust the drift and diffusion terms according to the VG model
    # The drift should be adjusted by subtracting the term theta * kappa
    # The diffusion term now includes the square root of dG, the gamma process
    log_S = np.log(S0) + np.cumsum((r - q - theta * kappa) * dt + theta * dG + sigma * np.sqrt(dG) * dW,
    ↪  axis=1)

    # Convert log prices back to regular prices
    S = np.exp(log_S)

    # Prepend the initial asset prices to the simulated paths
    S = np.hstack((S0 * np.ones((num_paths, 1)), S))

    # Calculate the payoff for each path at maturity for a call option
    payoffs = np.maximum(S[:, -1] - K, 0)

    # Calculate the option price as the present value of the expected payoff
    option_price = np.exp(-r * T) * np.mean(payoffs)

    return option_price
```

Theta (θ) The parameter θ represents the drift of the Variance Gamma process. In financial terms, drift can be thought of as the average trend or direction in which the price of the underlying asset is expected to move over a given period. A positive θ would indicate a general upward trend in the asset price, while a negative θ would indicate a downward trend.

In the VG model, θ adjusts the process so that it can better fit the asymmetries observed in the returns of real assets, allowing the process to lean towards either positive or negative returns. This helps to capture the market's skewness.

Kappa (κ) The parameter κ, sometimes referred to as the variance rate or scale parameter, is related to the volatility of the returns of the underlying asset in the VG model. Specifically, it controls the rate at which variance "arrives" in the process, since the VG process is a Lévy process used to model jumps in asset prices.

A smaller κ means that variance arrives at a faster pace, indicating more frequent jumps and thus a return distribution with heavier tails (high kurtosis). This is characteristic of a leptokurtic distribution, where extreme events are more likely than those predicted by a normal (Gaussian) distribution. Conversely, a larger κ implies a slower arrival of variance, with less frequent jumps and a return distribution closer to normal.

In our case, due to the lack of historical data, we will estimate these parameters to obtain a plausible option price. However, in reality, θ and κ are adjusted to fit historical asset price data or to match market prices of derivative instruments. They are crucial for capturing the unique characteristics of the underlying asset's return

distribution, thereby enabling a more accurate option valuation.

Change the MC simulator from the tutorials in a similar vein and compare option prices/implied volatilities.

```
def BNS_MC_CallPrice(simN, stepsN, S0, K, T, r, q, params):
    """
    Monte Carlo simulation for call option pricing under the BNS model.

    Parameters
    ----------
    simN : int
        Number of simulations.
    stepsN : int
        Number of time steps.
    S0 : float
        Spot price of the underlying asset.
    K : float
        Strike price of the call option.
    T : float
        Time to maturity (in years).
    r : float
        Risk-free interest rate.
    q : float
        Continuous dividend yield.
    params : list
        Model parameters: v0 (initial variance), alpha, rho, lambda_ (jump intensity), theta (jump
        ↪ size).

    Returns
    -------
    call_price : float
        Simulated call option price.
    """
    v0, alpha, rho, lambda_, theta = params

    dt = T / stepsN
    logPrice = np.log(S0) + np.zeros((simN, stepsN + 1))
    variance = v0 * np.ones((simN, stepsN + 1))

    dWt = np.random.normal(0, 1, size=(simN, stepsN))

    sqrdt = np.sqrt(dt)
    compensator = lambda_ * rho / (theta - rho)

    # Simulate variance and log price paths
    for i in range(1, stepsN + 1):
        # Jumps
        jumpN = np.random.poisson(lambda_ * dt, size=simN)
        jump = np.random.gamma(jumpN, 1.0 / theta, size=simN)

        variance[:, i] = variance[:, i - 1] - alpha * variance[:, i - 1] * dt + jump

        logPrice[:, i] = (
```

```
            logPrice[:, i - 1]
            + (r - q - 0.5 * variance[:, i - 1] - compensator) * dt
            + np.sqrt(variance[:, i - 1]) * dWt[:, i - 1] * sqrdt
            + rho * jump
        )

    # Calculate call option payoff at maturity for each simulation
    S_T = np.exp(logPrice[:, -1])  # Final stock prices
    payoffs = np.maximum(S_T - K, 0)  # Call option payoffs

    # Discount payoffs back to present and take average
    call_price = np.exp(-r * T) * np.mean(payoffs)

    return call_price
```

Comparison of the results

Prices of the options :

```
# Parameters

S0 = 100 #(Prix initial)
K = 100 #(Prix d'exercice)
T = 1 #(1 an jusqu'à la maturité)
r = 0.05 #(Taux d'intérêt de 5%)
q = 0.00    # Dividend yield
sigma = 0.2 #(Volatilité)
theta = 0.08 #pour VG (ajusté pour simuler un drift similaire)
nu = 0.2 #pour VG (Taux de variance)
kappa = 0.2   # Kappa from the VG model
params = [0.1, 0.7, 0.04, 0.01, 0.01] #pour MC v0 (initial variance), alpha, rho, lambda_ (jump
↪  intensity), theta (jump size)
num_paths = 100
num_steps = 10000

# Calculate the MC simulated European call option price
vg_option_price = simulate_vg_option_price(S0, K, T, r, q, sigma, theta, kappa, nu, num_paths, num_steps)
mc_call_price = BNS_MC_CallPrice(num_paths, num_steps, S0, K, T, r, q, params)
print('Price with MC :' , mc_call_price)
print('Price with VG : ', vg_option_price)
print('Price with classic BNS : ', call_price_BNS)
print('Price with BNS Levy Subordinator : ', call_price_with_levy)
```

Results :

Price with MC : 12.60

Price with VG : 11.48

Price with classic BNS : 11.24

Price with BNS Levy Subordinator : 12.58

To calculate the implied volatilities of options using the prices obtained from the simulate_vg_option_price, BNS_MC_CallPrice, and CallPricingLewis_eta_with_Levy functions, we will use a root-finding or optimization routine to invert the Black-Scholes formula and find the volatility that produces the simulated option price.

Implied volatility is the volatility that, when input into the Black-Scholes model, yields the option price observed in the market (or in our case, the simulated price). To do this, we will use a minimize function to find the implied volatility such that the difference black_scholes_call_price(S, K, T, r, sigma) - target_price is minimized as much as possible

```
from scipy.stats import norm
from scipy.optimize import minimize

def black_scholes_call_price(S, K, T, r, sigma):
    d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    return S * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)

def implied_volatility_minimize(target_price, S, K, T, r):
    def objective_function(sigma):
        return (black_scholes_call_price(S, K, T, r, sigma) - target_price)**2

    #result = minimize(objective_function, 0.009, bounds=[(1e-8, 10)], method='Nelder-Mead')
    result = minimize(objective_function, 0.009, tol = 0, method='Nelder-Mead')
    return result.x[0]  # Retourne la volatilité implicite optimisée

# fonctions de simulation pour obtenir le prix d'option
vg_option_price = simulate_vg_option_price(S0, K, T, r, q, sigma, theta, kappa, nu, num_paths, num_steps)
bns_Levy_Sub = CallPricingLewis_eta_with_Levy(S0, K, T, r, q, params_levy_sub, trunc, im)
mc_option_price = BNS_MC_CallPrice(num_paths, num_steps, S0, T, r, q, K, params)

# Calculez les volatilités implicites
vg_iv = implied_volatility_minimize(vg_option_price, S0, K, T, r)
mc_iv = implied_volatility_minimize(mc_option_price, S0, K, T, r)
bns_levy_sub_iv = implied_volatility_minimize(bns_Levy_Sub, S0, K, T, r)

print('Implied Vol with MC : ', mc_iv)
print('Implied Vol with VG : ', vg_iv)
print('Implied Vol with BNS with Levy Subordinator : ', bns_levy_sub_iv)
```

Results :
Implied Vol with MC : 0.0063
Implied Vol with VG : 0.2275
Implied Vol with BNS with Levy Subordinator : 0.2564

Implied Volatily with MC BNS : This very low implied volatility value suggests that, under the BNS model using a Monte Carlo simulation approach, the market perceives very little uncertainty or risk in the underlying asset's price movements over the option's life. This could indicate a market environment with minimal expected fluctuations. Implied Volatility with BNS and Levy Subordinator : This value, derived from the BNS model incorporating a Lévy subordinator, is the highest among the three, indicating the greatest level of perceived risk or uncertainty in the underlying asset's price movements. At approximately 25.64%, this implied volatility suggests that the inclusion of a Lévy subordinator, which models infinite activity or jumps in the asset price, adds to the complexity and unpredictability captured by the model. This could reflect a market view that anticipates more significant price movements, possibly due to the model's ability to capture extreme events or

heavy tails in the asset's return distribution.

5 Question 5

Implement the Lewis integral formula for the NIGSVOU, VGSVOU, CGMYSVOU Lévy models with stochastic volatility

First we need to determine the characteristic function of the Ornstein-Uhlenbeck's process

```
def char_func_OU(z,  a, sigma_OU, T, X0=1):
  return np.exp( 1j * X0 * np.exp(a*T) * z - ((sigma_OU ** 2) *(np.exp(2 * a * T ) - 1) * (z**2))/(4 *
  ↪  a))
```

NIGSVOU

Once done, we can define our characteristic function for the NIG-SV-OU model :

Recall of the NIG model :

NIG model

$$\exp\left(iz\log S_0 + (r - q - \Theta(-i))izT + \frac{T}{\kappa}\left(1 - \sqrt{1 + z^2\sigma^2\kappa - 2i\gamma z\kappa}\right)\right) \quad (3.61)$$

```
def unit_time_log_char_f_NIG(z, sigma, nu, theta):
  return sigma * ( nu/theta - np.sqrt( (nu**2 / theta ** 2) - 2 * ((theta * 1j * z) / sigma ** 2) + z**2)
  ↪  )
```

```
def char_f_log_price_NIGSVOU(z, S0, r, q, T, a, sigma_OU, sigma, nu, theta):
  return np.exp(1j * z * np.log(S0 * np.exp((r - q)*T))) * ( char_func_OU( -1j *
  ↪  unit_time_log_char_f_NIG(z, sigma, nu, theta) , a, sigma_OU, T) / (char_func_OU( -1j *
  ↪  unit_time_log_char_f_NIG(-1j, sigma, nu, theta) , a, sigma_OU, T)) ** (1j * z) )
```

We implement the corresponding Lewis Integral formula (with the right parameters) :

```
def CallPricingLewis_NIGSVOU(S, K, T, r, q, params, trunc, im):
    """
    Fourier inversion method using Lewis approach for call and put pricing
    """
    a, sigma_OU, sigma, nu, theta = params

    def integrand(om, S, K, T, r, q, params, im):
        i = 1j
        z = om + i * im
        return  char_f_log_price_NIGSVOU(-z, S0, r, q, T, a, sigma_OU, sigma, nu, theta) * (K ** (1 + i *
        ↪  z) / (i * z - z ** 2)) * np.exp(-1j * z * (np.log(S + (r -q)*T)))

    call, _ = si.integrate.quad(integrand, -trunc, trunc, args=(S, K, T, r, q, params, im))

    call *= 0.5 * np.exp(-r * T) / np.pi
    call = abs(call)
```

```
    return call

S = 100
trunc = 10
params_NIGSVOU = [0.05, 0.1, 0.2, 0.3, -0.06]
price_NIGSVOU = CallPricingLewis_NIGSVOU(S, K, T, r, q, params_NIGSVOU, trunc, im)
print('Price with NIGSVOU : ', price_NIGSVOU)
```

Results : Price with NIGSVOU : 11.82

VG-SV-OU

We apply the exact same process for the VG-SV-OU model. We recall the definition of the Variance Gamma Model :

Variance Gamma model

$$\phi_X(z) = \exp\left(iz \log S_0 + (r - q - \Theta(-i))izT\right)\left(1 - iz\kappa\theta + \frac{\sigma^2 z^2 \kappa}{2}\right)^{-T/\kappa} \quad (3.62)$$

```
def unit_time_log_char_f_VG(z, sigma, theta, nu=1):  #We take sigma = 1 so that C=1=y(0) (y(0) from OU
↪  process)
  C=1/nu
  G=1/(np.sqrt(((theta)**2 * (nu)**2)/4 + ((sigma)**2 * nu)/2)- (theta*nu)/2)
  M=1/(np.sqrt(((theta)**2 * (nu)**2)/4 + ((sigma)**2 * nu)/2) + (theta*nu)/2)

  return C * np.log(G*M / (G*M)+(M-G)*1j * z + z ** 2)
```

We found the following charateristic function :

```
def char_f_log_price_VGSVOU(z, S0, r, q, T, a, sigma_OU, sigma, theta):
  return np.exp(1j * z * np.log(S0 * np.exp((r - q)*T))) * (char_func_OU( -1j *
  ↪  unit_time_log_char_f_VG(z, sigma, theta) , a, sigma_OU, T) / (char_func_OU( -1j *
  ↪  unit_time_log_char_f_VG( -1j, sigma, theta) , a, sigma_OU, T)) ** (1j * z) )
```

And we defined the following Lewis Integral :

```
def CallPricingLewis_VGSVOU(S, K, T, r, q, params, trunc, im):
    """
    Fourier inversion method using Lewis approach for call and put pricing
    """

    a, sigma_OU, sigma, theta = params

    def integrand(om, S, K, T, r, q, params, im):
        i = 1j
        z = om + i * im
        return  char_f_log_price_VGSVOU(-z, S0, r, q, T, a, sigma_OU, sigma, theta) * (K ** (1 + i * z) /
        ↪  (i * z - z ** 2)) * np.exp(-1j * z * (np.log(S + (r -q)*T)))
```

```
    call, _ = si.integrate.quad(integrand, -trunc, trunc, args=(S, K, T, r, q, params, im))

    call *= 0.5 * np.exp(-r * T) / np.pi
    call = abs(call)

    return call

trunc = 10
params_VSGVOU = [0.05, 0.09, 0.2, -0.03]
price_VSGVOU = CallPricingLewis_VGSVOU(S, K, T, r, q, params_VSGVOU, trunc, im)
print('Price with VSGVOU : ', price_VSGVOU)
```

Results : Price with VSGVOU : 11.69

CGMY-SV-OU

Charateristic function :

```
def unit_time_log_char_f_CGMY(z, C_n, C_p, Y_p, Y_n, G, M) :
  xi = C_n / C_p
  return C_p * np.math.gamma( -Y_p) * ( (M - 1j*z) ** Y_p - M ** Y_p) + xi * np.math.gamma(- Y_n) * ( (G
  ↪  + 1j * z) ** Y_n - G ** Y_n)
```

```
def char_f_log_price_CGMYSVOU(z, S0, r, q, T, a, sigma_OU, C_n, C_p, Y_p, Y_n, G, M):
  return np.exp(1j * z * np.log(S0 * np.exp((r - q)*T))) * ( char_func_OU( -1j *
  ↪  unit_time_log_char_f_CGMY(z, C_n, C_p, Y_p, Y_n, G, M) , a, sigma_OU, T) / (char_func_OU( -1j *
  ↪  unit_time_log_char_f_CGMY(-1j, C_n, C_p, Y_p, Y_n, G, M) , a, sigma_OU, T)) ** (1j * z) )
```

Corresponding Lewis Integral :

```
def CallPricingLewis_CGMYSVOU(S, K, T, r, q, params, trunc, im):
    """
    Fourier inversion method using Lewis approach for call and put pricing
    """

    a, sigma_OU, C_n, C_p, Y_p, Y_n, G, M =params

    def integrand(om, S, K, T, r, q, params, im):
        i = 1j
        z = om + i * im
        return  char_f_log_price_CGMYSVOU(-z, S0, r, q, T, a, sigma_OU,  C_n, C_p, Y_p, Y_n, G, M) * (K
        ↪  ** (1 + i * z) / (i * z - z ** 2)) * np.exp(-1j * z * (np.log(S + (r -q)*T)))


    call, _ = si.integrate.quad(integrand, -trunc, trunc, args=(S, K, T, r, q, params, im))

    call *= 0.5 * np.exp(-r * T) / np.pi
    call = abs(call)

    return call
```

```
trunc = 10
params_CGMYSVOU = [3.8, 0.2, 0.09, 0.2, -0.037, 0.3, 0.8, 0.2]
price_CGMYSVOU = CallPricingLewis_CGMYSVOU(S, K, T, r, q, params_CGMYSVOU, trunc, im)
print('Price with CGMYSVOU : ', price_CGMYSVOU)
```

Results : Price with CGMYSVOU : 11.81

In the context of option pricing models such as CGMY with Stochastic Volatility and Jumps (CGMYSVOU), Variance Gamma with Stochastic Volatility and Jumps (VGSVOU), and NIG with Stochastic Volatility and Jumps (NIGSVOU), all parameters have been estimated to produce plausible option prices for a scenario where the underlying asset's spot price (S0) is 100, with a strike price of 100 and a maturity of one year. This estimation process involves selecting model parameters that reflect realistic market conditions and option pricing behaviors, ensuring that the calculated option prices are within reasonable bounds for the given setup.

However, it's important to note that while these parameter estimations are crafted to yield credible option prices, a more rigorous approach would involve calibrating the model parameters using historical market data. Calibration to historical data allows for the adjustment of the model's parameters to closely align with real market dynamics, capturing the historical volatility, jump frequencies, and other relevant market characteristics. This process enhances the model's predictive accuracy and reliability, as it grounds the model's assumptions and parameters in empirical evidence rather than theoretical or assumed market conditions.

6 Question 6

Modify the Benth and Benth (2008) commodity spot price model in order to accommodate other stochastic OU factors other than a single non-Gaussian OU process Y_t and a seasonality function.

We will chose : $log(S_t/S_0) = log\lambda_t + X_t + Y_t$

Or : $S(t) = S_0 * \lambda(t) * exp(X(t) + Y(t))$

X is a Gaussian (driven by a Brownian motion) OU process, and Y is non-Gaussian (driven by a Levy process) OU as in Benth and Benth model.

```
def benth_model(S0, Lambda, Xt, Yt):
    return S0 * Lambda * np.exp(Xt + Yt)

def X(T,mu,theta,sigma):
    dt = 0.01  # time step
    n = int(T / dt)  # number of time steps
    t = np.linspace(0.0, T, n)  # time grid

    # Volatility process parameters
    alpha = 0.2  # volatility of volatility
    beta = 0.1  # mean volatility

    # Initialize processes
    x = np.zeros(n)
```

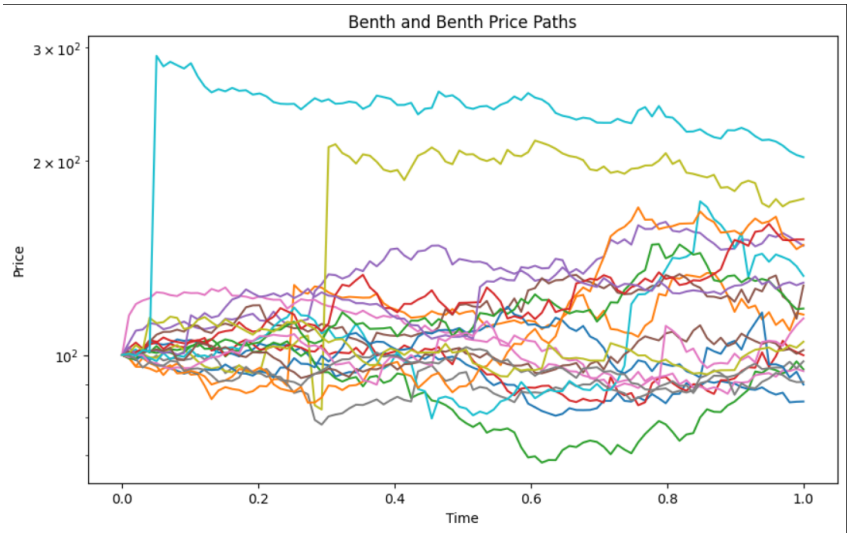
```
    v = np.zeros(n)

    # Initial conditions
    x[0] = 0.0
    v[0] = beta

    # Simulate processes
    for i in range(n - 1):
        dW_x = np.random.normal(0, np.sqrt(dt))
        dW_v = np.random.normal(0, np.sqrt(dt))
        x[i + 1] = x[i] + theta * (mu - x[i]) * dt + v[i] * dW_x
        v[i + 1] = v[i] + alpha * (beta - v[i]) * dt + sigma * dW_v
    return x

def Y(T,alpha,beta):

    dt = 0.01  # time step
    n = int(T / dt)  # number of time steps
    t = np.linspace(0.0, T, n)  # time grid

    # Ornstein-Uhlenbeck parameters
    mu = 0.1  # mean
    theta = 0.5  # speed of reversion to the mean
    sigma = 0.1  # volatility

    # Initialize process
    y = np.zeros(n)

    # Initial condition
    y[0] = 0.0

    # Simulate process
    for i in range(n - 1):
        dW = levy_stable.rvs(alpha, beta, loc=0, scale=np.sqrt(dt))  # Lévy noise
        y[i + 1] = y[i] + theta * (mu - y[i]) * dt + sigma * dW
    return y
```

Implement a Monte Carlo code

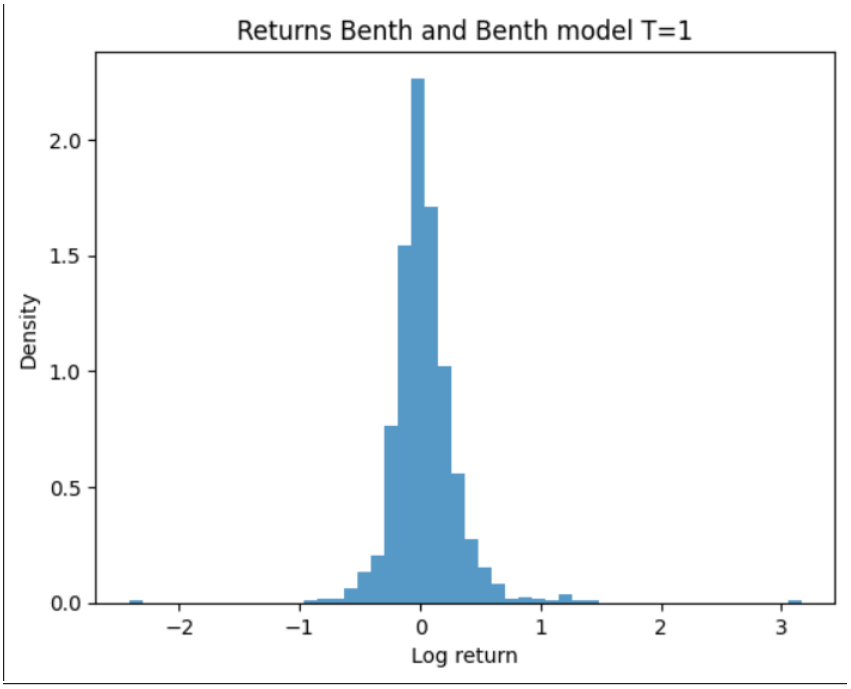
```
def MonteCarlo_BenthandBenth(S0,Lambda,T,mu,theta,sigma,alpha,beta,nb_simulations):
    # Initialize array to hold S(t) values
    S_t = np.zeros((nb_simulations, int(T / 0.01)))

    # Compute S(t) for all simulations
    for i in range(nb_simulations):
        Xt = X(T, mu, theta, sigma)
        Yt = Y(T, alpha, beta)
        S_t[i, :] = benth_model(S0, Lambda, Xt, Yt)
    #mean_S_t = np.mean(S_t, axis=0)
    return S_t
```

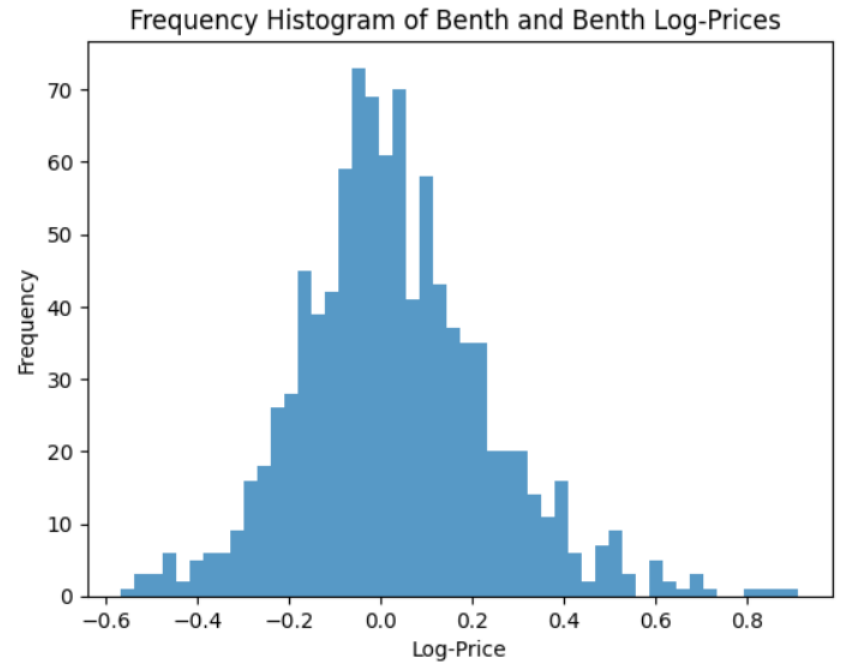
We plot some of our paths with theses parameters :

```
T = 1.0  # total time
mu = 0.0  # mean
theta = 0.5  # speed of reversion to the mean
sigma = 0.1  # volatility
alpha = 1.70  # stability parameter for Levi process
beta = 0.5  # skewness parameter for Levi process

# Parameters for benth_model
S0 = 100  # initial price
Lambda = 1.0  # risk-neutral drift

# Number of simulations
nb_simulations = 1000

S_t = MonteCarlo_BenthandBenth(S0,Lambda,T,mu,theta,sigma,alpha,beta,nb_simulations)
```



As we can see the the Levy component in the Benth and Benth model creates some jumps, which are smoothed by the number of simulations in the final price.
To see if it generates a coherent path and returns, we plot the result with basic parameters. But to obtain coherent path prices and to study the return, we must calibrate the model. For this, we need the historical log return to calibrate it.





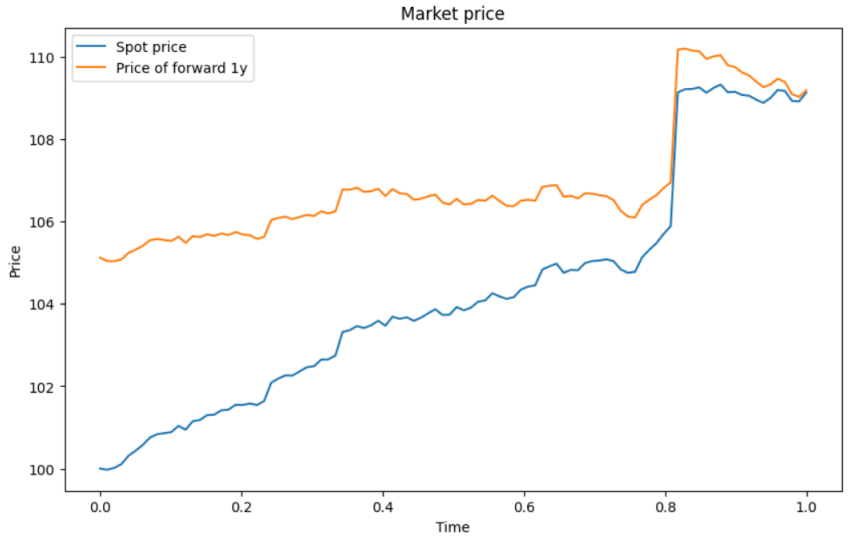
Our histograms seems to be acceptable with our parameters, we will continue with these.

Plot the forward curve, showing contango and backwardation.

```
def calculate_forward_prices(S, r, T):
    return S * np.exp(r * T)

spot_price = mean_S_t[0]
```

```
t = np.linspace(0.0, T, int(T / 0.01))
forward_prices = []
for i in range(0,int(T / 0.01)):
    forward_prices.append(calculate_future_prices(mean_S_t[i], 0.05, T-(i/int(T / 0.01))))
```

We then plot our forward prices and our spot price, so we can determine if it shows contango or backwardation

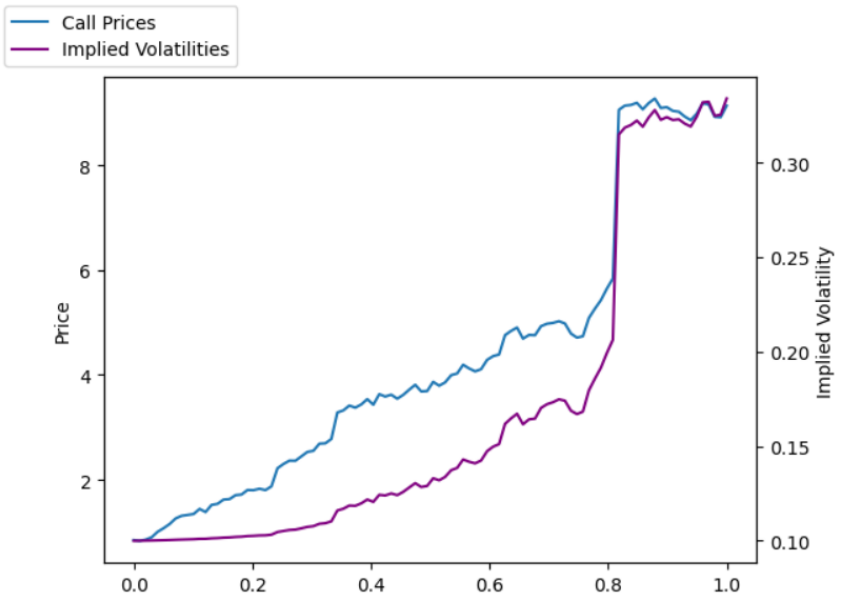


As we can see on the plot, the forward curve is in contango Calculate the price of options expirying at τ < T on futures F(t, T) and extract implied volatilities

```
# Black-Scholes formula for option pricing
def black_scholes_call(F, K, r, T, sigma):
    d1 = (np.log(F / K) + (0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    return np.exp(-r * T) * (F * norm.cdf(d1) - K * norm.cdf(d2))

BS_market = np.zeros_like(mean_S_t)
K = 100
r = 0.05

for i in range(len(mean_S_t)):
    BS_market[i] = (black_scholes_call(mean_S_t[i], K,  T-(i/int(T / 0.01)), r, sigma))

def implied_volatility(S, K, T, r, C_market):
    # Define a function that calculates the difference between the market price and the Black-Scholes
    ↪  price
    def f(sigma):
        #print(black_scholes_call(S, K, T, r, sigma))
        #print(C_market)
        return black_scholes_call(S, K, T, r, sigma) - C_market

    # Use the Newton-Raphson method to find the root
    sigma_initial_guess = 0.5
```

```
    sigma = newton(f, sigma_initial_guess)
    return sigma

# Extract the implied volatility
implied_volatilities = np.zeros_like(mean_S_t)

for i in range(len(implied_volatilities)):
    implied_volatilities[i] = implied_volatility(mean_S_t[i], K,  T, r, BS_market[i])

fig, ax1 = plt.subplots()
ax1.plot(t, BS_market, label='Call Prices')
ax1.set_ylabel('Price')
ax2 = ax1.twinx()
ax2.plot(t, implied_volatilities, label='Implied Volatilities', color='purple')
ax2.set_ylabel('Implied Volatility')
fig.legend(loc="upper left")
plt.show()
```

Call Price and its Implied volatility of a Call 1 year, K=100



7 Question 7

Try to calibrate to market one or more of the models discussed in the course and comment on the results. The choice of model and data is completely open. We will provide a set for reference, but it is not compulsory to use that.

In this part, we decided to calibrate the KOU model using the Nelder Mead algorithm. We only go for calibrating the model over one maturity.
To do that we took the already existing Kou MC

Then we defined the Objective Function objective_function_mnmz is the function that quantifies the difference between market prices and model prices.

It takes several arguments:
params: The Kou model parameters to be optimized.
market_prices: Observed option prices from the market.
strikes: Strike prices of the options.
KOU_MC_OPT: A Monte Carlo simulation function for the Kou model.
simN: Number of simulated paths.
stepsN: Number of time steps in each simulated path.
S0: Current underlying asset price.
T: Time to option maturity.
r: Risk-free interest rate.
q: Dividend yield.
The function iterates over each strike price, simulates the option prices using the Kou model, and calculates the sum of squared differences between the model prices and market prices. This sum is the error that the optimization process aims to minimize.

Loading Market Data The script loads market data from an Excel file containing information about SPX options. It then processes this data to calculate the mid-price for each option and prepares it for the calibration process.

Setting Initial Parameters and Bounds params_initial contains the initial guesses for the model parameters, which will be optimized. The Bounds object sets the permissible range for each parameter to ensure they stay within realistic and meaningful ranges during optimization.

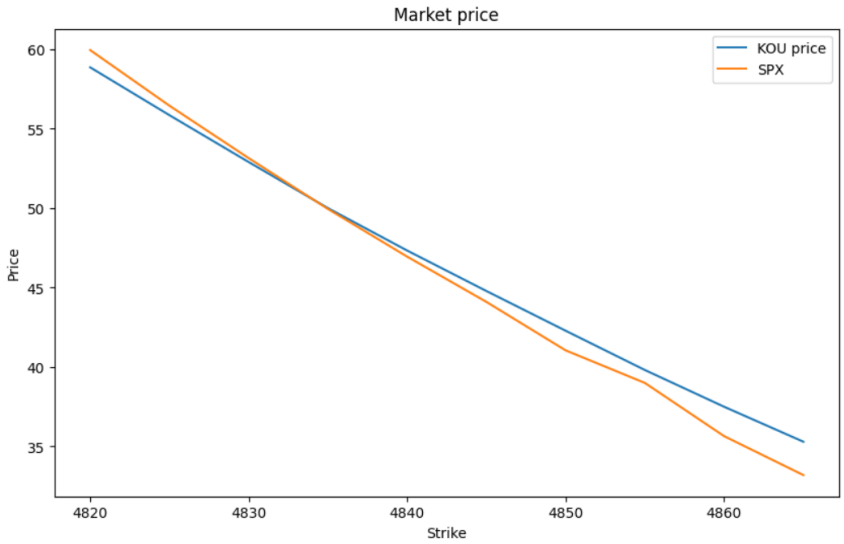
Optimization Process The minimize function is called with the objective function, initial parameters, additional arguments required for the objective function, bounds for the parameters, and the optimization method 'Nelder-Mead'. The result of this optimization process is stored in params_optimized, representing the best-fit parameters for the Kou model based on the market data.

Visualization Finally, the script plots the simulated option prices against the market prices for visual comparison. This helps in assessing the calibration visually, showing how closely the calibrated model prices match the observed market prices.

```
def KOU_MC_OPT(simN, stepsN, S0, K, T, r, q, params):
    """
    Monte Carlo for Kou model

    Parameters
    ----------
    simN : int
        numbers of simmulations
```

```
        stepsN : int
            numbers of steps
        S0 : float
            Spot price
        K : float
            Strike price
        T : float
            Time to maturity (in years)
        r : float
            Risk free rate
        q : float
            Continuous dividend yield
        params :
            list of parameters according to the choosen model

        Returns
        -------
        price : arrays
            options paths
        """

        # Merton model scheme in log coordinates

        # Parameters
        sigma, lam, p, lam_plus, lam_minus = params

        # Seeding the Mersenne Twister
        np.random.seed(123)

        dt = T / stepsN

        # Initialize arrays
        price = np.ones((simN, stepsN + 1)) * S0

        # Levy compensator
        jump_compensator = (lam * (p * lam_plus / (lam_plus - 1) + (1 - p) * lam_minus / (1 + lam_minus)
        ↪  - 1))  # *mu

        # Drift
        drift = (r - q - sigma ** 2 / 2 - jump_compensator) * dt * np.ones((simN, stepsN))

        # Brownian increments
        dWt = np.random.normal(0, 1, (simN, stepsN))
        diffusion = sigma * dWt * np.sqrt(dt)

        # Poisson process for jumps
        jump_N = si.stats.poisson.rvs(lam * dt, size=(simN, stepsN))

        jumps = np.zeros((simN, stepsN))

        for i in range(simN):
            for j in range(stepsN):
                for k in range(jump_N[i, j]):
                    U = np.random.rand(1)
```

```
                jumps[i, j] += np.random.exponential(1 / lam_plus) * (U < p) -
                ↪  np.random.exponential(1 / lam_minus) * (U > p)

    # Price evolution
    price[:, 1:] = S0 * np.exp(np.cumsum(drift + diffusion + jumps, axis=1))

    # Compute the payoff for each simulated price path
    payoff = np.maximum(price[:, -1] - K, 0)

    # Discount the payoffs to get the option price
    price_opt = np.exp(-r * T) * np.mean(payoff)

    return price_opt



def objective_function_mnmz(params, market_prices,strikes, KOU_MC, simN, stepsN, S0, T, r, q):

    for K in strikes:
        model_prices = KOU_MC(simN, stepsN, S0, K,T, r, q, params)
        #print(model_prices)
        error = np.sum((model_prices - market_prices)**2)
    return error



params_initial = [0.131400813, 0.100984056, 0.001259063, 0.001303577, 2.85494576]
bounds = Bounds([0.01, 0, 0, 0.01, 0.01], [2, 100, 1, 10, 10])

market_prices = np.array(df_data_spx['Price'])

result = minimize(objective_function_mnmz, params_initial, args=(market_prices,
↪  df_data_spx['Strike'],KOU_MC_OPT, 1000, 100, 4852.24, 0.035, 0, 0), bounds =
↪  bounds,method='Nelder-Mead')

params_optimized = result.x


model_prices = []
for i in df_data_spx['Strike']:
    model_prices.append(KOU_MC_OPT(100, 10, 4852.24, i, 0.035, 0, 0, params_optimized))
print(np.sum(model_prices-df_data_spx['Price']))


plt.figure(figsize=(10, 6))
plt.plot(df_data_spx['Strike'], model_prices, label='KOU price')
plt.plot(df_data_spx['Strike'], df_data_spx['Price'], label='Market Price')
plt.xlabel('Strike')
plt.ylabel('Price')
plt.title('Market price')
plt.legend()
```



The plot above illustrates the result of calibrating the Kou model to market data for a single maturity. The blue line represents the option prices as predicted by the Kou model, while the orange line depicts the actual market prices across various strike prices.

In conclusion, the calibration of the Kou model has been successfully executed, yielding a close match between the model and market prices for the options with the specified maturity. This is a promising indication of the model's ability to capture market behaviour, making it a valuable addition to the array of financial tools for option valuation and risk assessment. But we only did it for one maturity, and we didn't have enough time to perform the global optimization of the KOU model.