Ecole Supérieure d'Ingénieurs
Léonard-de-Vinci

Simulation Method Project

Author:
Mathis Raimbault

Supervisor:
Vincent Lambert

April 3, 2024

Contents

1 Ratio of Uniforms

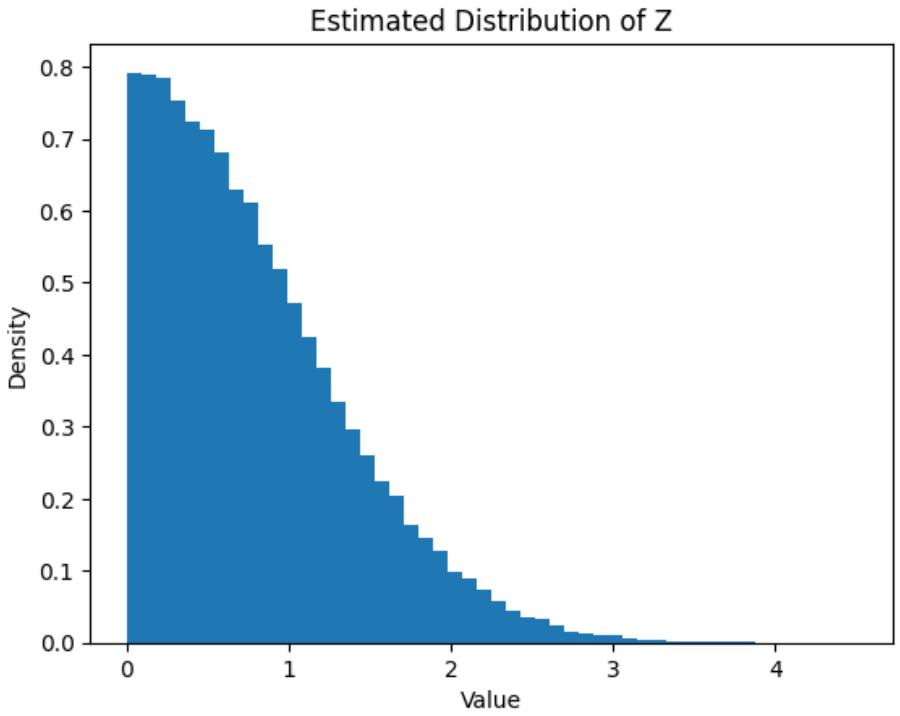
- Do :
  - generate $(U_1, U_2) \sim (\mathcal{U}([0,1]))^2$
  - While $U_1 > \exp\left(-\frac{1}{4}\left(\frac{U_2}{U_1}\right)^2\right)$
- Return $Z = \frac{U_2}{U_1}$

1.1 Implement the algorithm, what does the distribution of Z look like?

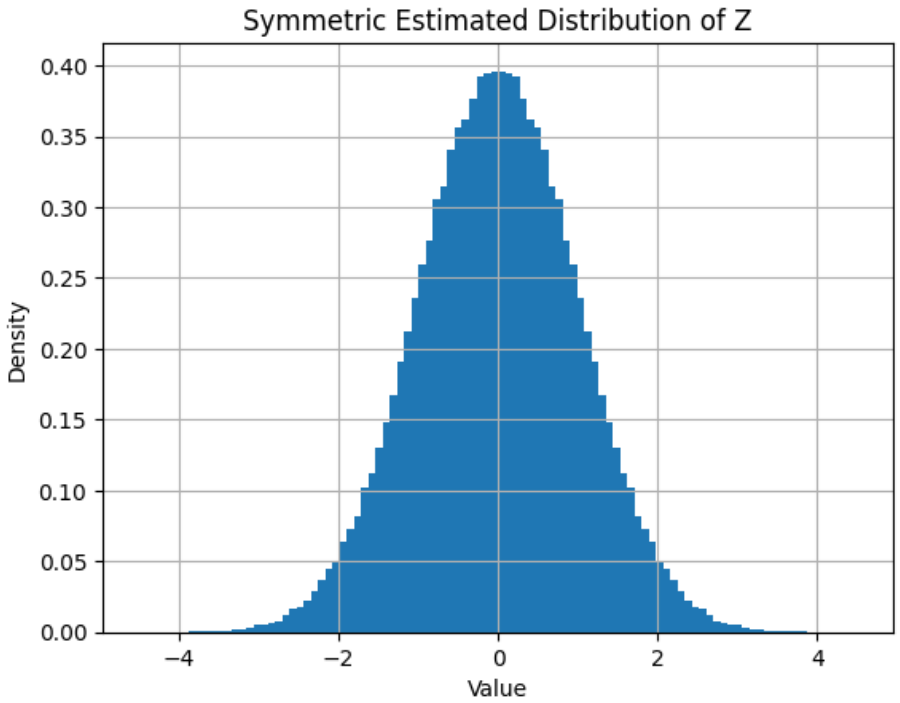
```
def ratio_of_uniforms():
    while True:
        U1, U2 = np.random.uniform(0, 1), np.random.uniform(0, 1)
        if U1 > np.exp(-0.25 * (U2 / U1)**2):
            continue
        else:
            Z = U2 / U1
            return Z

sample_size = 100000
samples = np.array([ratio_of_uniforms() for _ in range(sample_size)
    ])

plt.hist(samples, bins=50, density=True)
plt.title('Estimated Distribution of Z')
plt.xlabel('Value')
plt.ylabel('Density')
plt.show()
```

The distribution comes very close to the positive part of the Normal law. It is therefore sufficient to duplicate it symmetrically to obtain an approximation of the Normal law.

```
symmetric_samples = np.concatenate((samples, -samples))

plt.hist(symmetric_samples, bins=100, density=True)
plt.title('Symmetric Estimated Distribution of Z')
plt.xlabel('Value')
plt.ylabel('Density')
plt.grid(True)
plt.show()
```



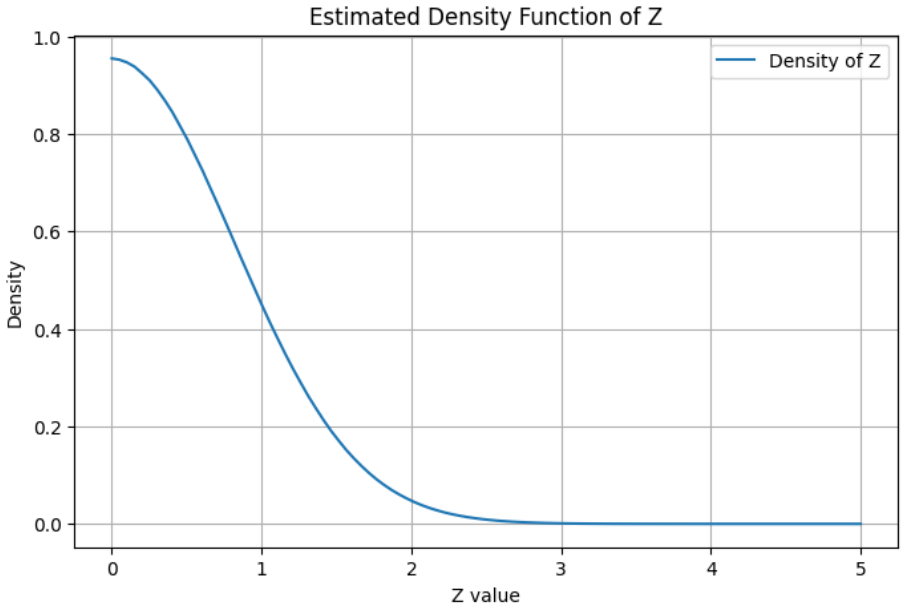
The distribution comes very close to the positive part of the Normal law.
It is therefore sufficient to duplicate it symmetrically to obtain an approximation of the Normal law.

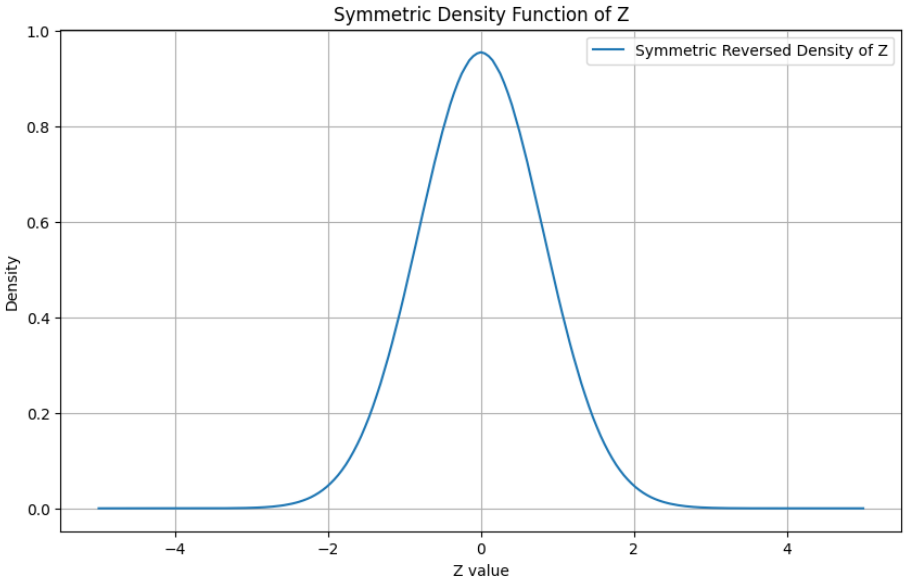


1.2 Find the distribution of Z by using the change of variables $U_1 = X$, $U_2 = XZ$

```
def joint_density(u1, u2):
    condition = u1 <= np.exp(-0.25 * (u2 / u1)**2)
    return np.where(condition, np.exp(-0.25 * (u2 / u1)**2), 0)

def density_Z(z, delta_x=0.001, max_x=10):
    x_values = np.arange(delta_x, max_x, delta_x)
    density = joint_density(x_values, x_values * z)
    return np.sum(density * x_values) * delta_x

z_values = np.linspace(0, 5, 100)
density_values = np.array([density_Z(z) for z in z_values])

density_values /= np.sum(density_values) * (z_values[1] - z_values
    [0])

plt.figure(figsize=(8, 5))
plt.plot(z_values, density_values, label='Density of Z')
plt.title('Estimated Density Function of Z')
plt.xlabel('Z value')
plt.ylabel('Density')
plt.legend()
plt.grid(True)
plt.show()

symmetric_reversed_density = np.concatenate((density_values[::-1],
     density_values))
```

```
symmetric_z_values = np.concatenate((-z_values[::-1], z_values))

plt.figure(figsize=(10, 6))
plt.plot(symmetric_z_values, symmetric_reversed_density, label='
    Symmetric Reversed Density of Z')
plt.title('Symmetric Density Function of Z')
plt.xlabel('Z value')
plt.ylabel('Density')
plt.legend()
plt.grid(True)
plt.show()
```





1.3 What is the theoretical acceptance rate? What is the theoretical average number of samples of (U1, U2) needed to generate a sample Z?

```
def simulate_ratio_of_uniforms(num_iterations=1000000):
    count_accepted = 0

    for _ in range(num_iterations):
        U1, U2 = np.random.uniform(0, 1), np.random.uniform(0, 1)
        if U1 <= np.exp(-0.25 * (U2 / U1)**2):
            count_accepted += 1

    acceptance_rate = count_accepted / num_iterations
    average_num_samples_needed = 1 / acceptance_rate

    return acceptance_rate, average_num_samples_needed

acceptance_rate, average_num_samples_needed =
    simulate_ratio_of_uniforms()

print("\nTheorical acceptance rate : ",acceptance_rate)
print("\nTheorical average number of samples of (U1,U2) needed to
    generate a sample Z : ", average_num_samples_needed)
```

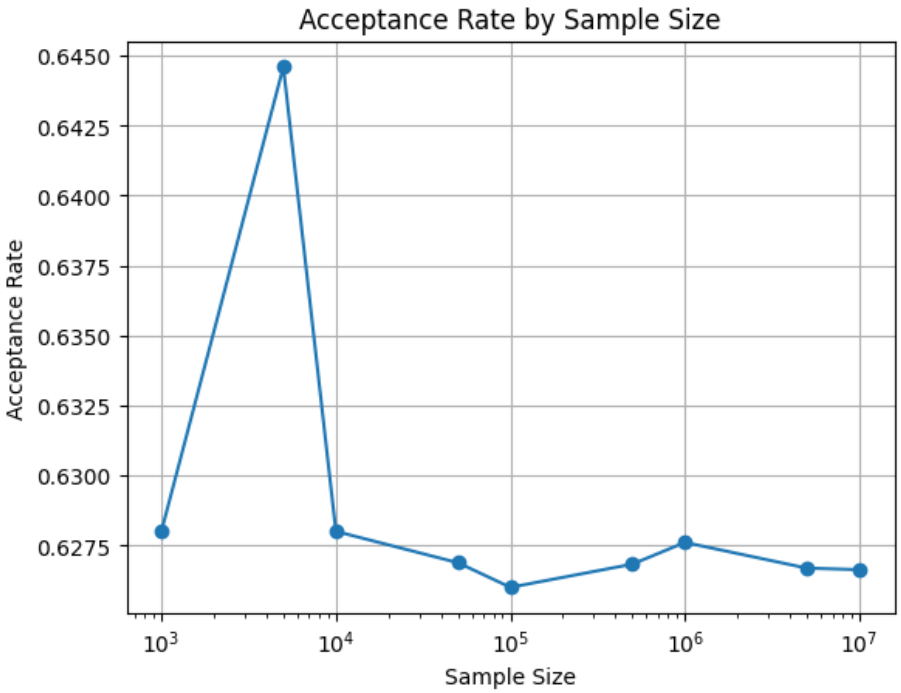
Theorical acceptance rate : 0.626999
Theorical average number of samples of (U1,U2) needed to generate a sample Z : 1.594898875436803

```
def simulate_ratio_of_uniforms(num_iterations):
    count_accepted = 0

    for _ in range(num_iterations):
        U1, U2 = np.random.uniform(0, 1), np.random.uniform(0, 1)
        if U1 <= np.exp(-0.25 * (U2 / U1)**2):
```

```
            count_accepted += 1

    acceptance_rate = count_accepted / num_iterations
    average_num_samples_needed = 1 / acceptance_rate

    return acceptance_rate, average_num_samples_needed

sizes_to_simulate = [1000, 5000 ,10000, 50000, 100000, 500000,
    1000000, 5000000, 10000000]
results = {}

for size in sizes_to_simulate:
    results[size] = simulate_ratio_of_uniforms(size)

print("Size\t\tAcceptance Rate\tAverage Number of Samples Needed")
for size, (rate, average) in results.items():
    print(f"{size}\t\t{rate:.4f}\t\t{average:.2f}")

acceptance_rates = [rate for size, (rate, average) in results.items
    ()]
plt.plot(sizes_to_simulate, acceptance_rates, marker='o')
plt.xlabel('Sample Size')
plt.ylabel('Acceptance Rate')
plt.title('Acceptance Rate by Sample Size')
plt.grid(True)
plt.xscale('log')
plt.show()
```

Size	Acceptance Rate	Average Number of Samples Needed
1000	0.6280	1.59
5000	0.6446	1.55
10000	0.6280	1.59
50000	0.6269	1.60
100000	0.6260	1.60
500000	0.6268	1.60
1000000	0.6276	1.59
5000000	0.6267	1.60
10000000	0.6266	1.60

Table 1: Acceptance rates and average number of samples needed at various sample sizes.


Acceptance Rate by Sample Size
Acceptance Rate
0.6450
0.6425
0.6400
0.6375
0.6350
0.6325
0.6300
0.6275
10^3
10^4
10^5
10^6
10^7
Sample Size

2 Importance sampling

We consider an asset (S_t) with Black-Scholes dynamics with the following parameters:

- $S_0 = \$50$
- $r = 1\%$ per year
- $\sigma = 17\%$ per square root of year

We would like to compute how many European digital Call options are worth a European Call option, with maturity $T = 1$ year and strike $K = \$600000$.

2.1 Write the solution using closed formula (Black-Scholes). What happens when you evaluate this formula?

We calculate here the price of a European call option and the price of a digital call with the desired parameters. We had restrictions on the use of certain libraries. Thus, to calculate N(d1) and N(d2), we calculated the area under the curve of the distribution function of the N(0,1) law.

```
import numpy as np

S0 = 50
K = 600000
r = 0.01
sigma = 0.17
T = 1

d1 = (np.log(S0 / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.
    sqrt(T))
d2 = d1 - sigma * np.sqrt(T)

def normal_pdf(z):
    return (1 / np.sqrt(2 * np.pi)) * np.exp(-(z ** 2) / 2)

def area_under(x):
    sum = 0
    i = -10
    step = 0.00001
    while i <= x:
        sum += normal_pdf(i)*step
        i += step
    return sum

call = S0 * area_under(d1) - K * np.exp(-r * T) * area_under(d2)

print("The price of the European option is", call)
```

The price of the European option is: 0.0

```
def box_muller():
    u1, u2 = np.random.uniform(low=0.0001, high=0.9999, size=2)
    z0 = np.sqrt(-2 * np.log(u1)) * np.cos(2 * np.pi * u2)
    return z0

num_simulations = 10000000

z = np.array([box_muller() for _ in range(num_simulations)])

ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T + sigma * np.sqrt(T) *
    z)

payoff = np.where(ST > K, 1, 0)
price_digital = np.exp(-r * T) * np.mean(payoff)

print("The price of the European digital option is:", price_digital
    )
```

The price of the European digital option is : 0.0

We can see that with a strike of 600,000$, the price of the European Call Option and Digital European Call Option are 0. However, it's not possible that an option is free. We have to find a way to express the two prices in order to compute the ratio of them. We will use the importance sampling hypothesis in the next question to solve this problem.

```
ratio = call / price_digital

print("Here is the vanilla ratio / digital : ", ratio)
```

Here is the vanilla ratio / digital : nan

2.2 Apply the principle of importance sampling to estimate the ratio. Explain how the trick below is used.

Let's demonstrate the trick given in the exercise. We want to show that $\forall\, C \in \mathbb{R}$:

$$\frac{\sum_i \exp(l_i) f_i}{\sum_i \exp(l_i) g_i} = \frac{\sum_i \exp(l_i + C) f_i}{\sum_i \exp(l_i + C) g_i}$$

Let's use the property of the exponential: $e^{a+b} = e^a e^b$.
So in our ratio we have $\forall\ i \in [1, n]\ with\ n \in \mathbb{N}$:

$$\begin{aligned}
\frac{\sum_i e^{l_i} f_i}{\sum_i e^{l_i} g_i} &= \frac{e^{l_1} f_1 + e^{l_2} f_2 + \cdots + e^{l_n} f_n}{e^{l_1} g_1 + e^{l_2} g_2 + \cdots + e^{l_n} g_n} \\
&= \frac{e^{l_1} f_1 + e^{l_2} f_2 + \cdots + e^{l_n} f_n}{e^{l_1} g_1 + e^{l_2} g_2 + \cdots + e^{l_n} g_n} \cdot \frac{e^C}{e^C} \\
&= \frac{e^{l_1+C} f_1 + e^{l_2+C} f_2 + \cdots + e^{l_n+C} f_n}{e^{l_1+C} g_1 + e^{l_2+C} g_2 + \cdots + e^{l_n+C} g_n} \\
&= \frac{\sum_i e^{l_i+C} f_i}{\sum_i e^{l_i+C} g_i}
\end{aligned}$$

```
C = 9

num_simulations = 100000


adjusted_mean = (r - 0.5 * sigma**2) * T + C
Z = np.array([box_muller() for _ in range(num_simulations)])
ST = S0 * np.exp(adjusted_mean + sigma * np.sqrt(T) * Z)


payoff_call = np.maximum(ST - K, 0)
payoff_digital_call = np.where(ST > K, 1, 0)


likelihood_ratios = np.exp(-C * Z - 0.5 * C**2 * T)


price_call = np.exp(-r * T) * np.mean(payoff_call *
    likelihood_ratios)
price_digital_call = np.exp(-r * T) * np.mean(payoff_digital_call *
     likelihood_ratios)

#price ratio calculation
ratio_sampling = price_call / price_digital_call

print("Ratio sampling (call/digital call):", ratio_sampling)
```

Ratio des prix (call/digital call): 8679.928835538174

We adjust the simulations to a new probability measure, employing importance sampling to increase estimation efficiency. We simulate the final asset value, calculate payoffs for each option, adjust these payoffs with weights derived from importance sampling, and derive discounted option prices. Then, we could estimate the ratio between the call option and the digital call option.

Note that for a C $<$ 9, the ratio is None. Thus, we have arbitrarily taken C=9. It would be interesting to find a method to define the optimal C to obtain

the best ratio.

2.3 Is it realistic? why?

According to the result, one European Call Option is equal to 8679 European Digital Call Option.

The scenario with a strike price of 600,000 versus a current price of the underlying asset of 50 is not realistic, as it implies an unrealistic expectation that the underlying asset will increase more than 1,000 times its current value within the one-year period, which is extremely unlikely under normal market conditions, thus rendering the value of any call option or digital call option, almost zero.

The extremely high ratio of 8679 between the price of a classic call and that of a digital call is unrealistic, as it suggests a massive disproportion in the valuation of two financial products whose values should be significantly closer.

3 Acceptance-Rejection

3.1 Show that the ratio in the previous exercise can be expressed as an conditional expectation $\mathbb{E}[\Phi(X)|X \geq A]$ where $A \in \mathbb{R}$.
To show that the ratio in the previous exercise can be expressed as a conditional expectation $\mathbb{E}[\Phi(X)|X \geq A]$, where $A \in \mathbb{R}$, we can rewrite the ratio in terms of conditional probabilities. Let X be a random variable and $\Phi(X)$ be a function of X. We want to express the ratio as $\mathbb{E}[\Phi(X)|X \geq A]$.
Using the definition of conditional expectation, we have:

$$\mathbb{E}[\Phi(X)|X \geq A] = \int \Phi(x) * \mathbb{P}(X \geq A|X = X)dx \tag{1}$$

Now, let's express the ratio in terms of conditional probabilities:

$$Ratio = \frac{\Sigma_i exp(l_i) * f_i}{\Sigma_i exp(l_i) * g_i} \tag{2}$$

We can rewrite the numerator and denominator as conditional probabilities:

$$Numerator = \Sigma_i exp(l_i) * f_i = \Sigma_i exp(l_i) * \mathbb{P}(X = x_i|X \geq A) \tag{3}$$

$$Denominator = \Sigma_i exp(l_i) * g_i = \Sigma_i exp(l_i) * \mathbb{P}(X = x_i|X \geq A) \tag{4}$$

Substituting the numerator and denominator in the ratio expression, we get:

$$Ratio = \frac{\Sigma_i exp(l_i) * \mathbb{P}(X = x_i|X \geq A)}{\Sigma_i exp(l_i) * \mathbb{P}(X = x_i|X \geq A)} \tag{5}$$

Now, we can see that the ratio is expressed as the conditional expectation $\mathbb{E}[\Phi(X)|X \geq A]$. Therefore, the ratio in the previous exercise can be expressed as $\mathbb{E}[\Phi(X)|X \geq A]$, where $A \in R$.

3.2 Let $A > 0$. If $Y \sim \mathcal{E}(\lambda)$ with $\lambda > 0$, what is the conditional distribution of $Y - A$ given $Y \geq A$? Name it.
To find the distribution of $Y - A|Y \geq A$, we can use conditional probability. Let's denote $Z = Y - A$. We want to find the distribution of Z given that $Y \geq A$.
Using conditional probability, we have:

$$P(Z = z|Y \geq A) = \frac{P(Z = z, Y \geq A)}{P(Y \geq A)} \tag{6}$$

Since we know that Y follows an exponential distribution with parameter λ, i.e. $Y \sim \mathcal{E}(\lambda)$. The probability density function (PDF) of an exponential distribution is given by:

$$f_Y(y) = \lambda e^{-\lambda y}$$

To find $P(Z = z, Y \geq A)$, we can integrate the joint PDF of Z and Y over the region where $Z = z$ and $Y \geq A$:

$$P(Z = z, Y \geq A) = \int_{A+z}^{\infty} f_Y(y)dy$$

Substituting the PDF of the exponential distribution, we have:

$$P(Z = z, Y \geq A) = \int_{A+z}^{\infty} \lambda e^{-\lambda y}dy$$

Simplifying the integral, we get:

$$P(Z = z, Y \geq A) = e^{-\lambda(A+z)}$$

Similarly, we can find $P(Y \geq A)$ by integrating the PDF of Y over the region where $Y \geq A$:

$$P(Y \geq A) = \int_{A}^{\infty} f_Y(y)dy$$

Substituting the PDF of the exponential distribution, we have:

$$P(Y \geq A) = \int_{A}^{\infty} \lambda e^{-\lambda y}dy$$

Simplifying the integral, we get:

$$P(Y \geq A) = e^{-\lambda A}$$

Now, we can substitute these probabilities back into the conditional probability formula:

$$P(Z = z|Y \geq A) = \frac{P(Z = z, Y \geq A)}{P(Y \geq A)} = \frac{e^{-\lambda(A+z)}}{e^{-\lambda A}}$$

Simplifying the expression, we have:

$$P(Z = z|Y \geq A) = e^{-\lambda z}$$

Therefore, the distribution of $Y - A|Y \geq A$ is an exponential distribution with parameter λ.

3.3 We denote by g the density of $Y|Y \geq A$. Give the expression of g.
To find the expression of the density function g of $Y|Y > A$, we can use conditional probability.
Let's denote $Z = Y - A$. We want to find the density function of Z given that $Y > A$.
Using conditional probability, we have:

$$g(z) = \frac{f_Z(z)}{P(Y > A)}$$

Since $Y \approx (\lambda)$, we know that Y follows an exponential distribution with parameter λ. The probability density function (PDF) of an exponential distribution is given by:

$$f_Y(y) = \lambda e^{-\lambda y}$$

To find the density function $f_Z(z)$, we can use the change of variables method. Let $h(y) = y - A$, then $y = h^{-1}(z) = z + A$. The density function $f_Z(z)$ can be obtained by substituting $y = z + A$ into the PDF of Y:

$$f_Z(z) = f_Y(z + A) = \lambda e^{-\lambda(z+A)}$$

Substituting the PDF of Z and $Y > A$ into the expression of $g(z)$, we have:

$$g(z) = \frac{\lambda e^{-\lambda(z+A)}}{P(Y > A)}$$

To find $P(Y > A)$, we can integrate the PDF of Y over the region where $Y > A$:

$$P(Y > A) = \int_A^\infty f_Y(y)dy$$

Substituting the PDF of the exponential distribution, we have:

$$P(Y > A) = \int_A^\infty \lambda e^{-\lambda y}dy$$

Simplifying the integral, we get:

$$P(Y > A) = e^{-\lambda A}$$

Now, we can substitute these expressions back into the expression of $g(z)$:

$$g(z) = \frac{\lambda e^{-\lambda(z+A)}}{e^{-\lambda A}}$$

Simplifying the expression, we have:

$$g(z) = \lambda e^{-\lambda z}$$

Therefore, the density function g of $Y|Y > A$ is an exponential distribution with parameter λ.

3.4 Propose an algorithm to generate $Y|Y \geq A$.

To generate $Y|Y > A$, we can use the inverse transform method for exponential distribution.

Here's the algorithm:

- 1. Generate a random number U from a uniform distribution between 0 and 1.
- 2. Compute $Y = -\frac{1}{\lambda}\ln(U)$.
- 3. If $Y > A$, return Y.
- 4. If $Y \leq A$, go back to step 1 and generate a new random number.

This algorithm generates random values of Y that follow an exponential distribution with parameter λ, conditioned on $Y > A$.

```
def generate_Y_knowing_YsupA(num_iterations=1000000, A=0.5):
    Y = []
    for _ in range(num_iterations):
        U1, U2 = np.random.uniform(0, 1), np.random.uniform(0, 1)
        if U1 > np.exp(-0.25 * (U2 / U1)**2):
            Y.append(U1)
    return Y

Y = generate_Y_knowing_YsupA(A=0.5)
print("Mean of Y knowing Y > A : " + str(np.mean(Y)))
```

Mean of Y knowing $Y > A$: 0.42504972792515217

3.5 Let $X \sim \mathcal{N}(0,1)$. Write the expression of $f_{X|X>A}$ up to a normalizing constant (the simplest one), let us denote this function by f. To write the expression of $f_{X|X>A}$ up to a normalizing constant, we can use the conditional probability formula. Let's denote $Z = X - A$. We want to find the conditional density function of Z given that $X > A$.
Using conditional probability, we have:

$$f_Z(z|X > A) = \frac{f_X(x)}{P(X > A)}$$

Since $X \approx \mathcal{N}(0,1)$, we know that X follows a normal distribution with mean 0 and standard deviation 1. The probability density function (PDF) of a normal distribution is given by:

$$f_X(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

To find the density function $f_Z(z)$, we can use the change of variables method. Let $h(x) = x - A$, then $x = h^{-1}(z) = z + A$. The density function $f_Z(z)$ can be obtained by substituting $x = z + A$ into the PDF of X:

$$f_Z(z) = f_X(z + A) = \frac{1}{\sqrt{2\pi}} e^{-\frac{(z+A)^2}{2}}$$

To find $P(X > A)$, we can integrate the PDF of X over the region where $X > A$:

$$P(X > A) = \int_A^\infty f_X(x)dx$$

Substituting the PDF of the normal distribution, we have:

$$P(X > A) = \int_A^\infty \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx$$

Unfortunately, there is no closed-form solution for this integral. However, we can use numerical methods or approximation techniques to estimate the value of $P(X > A)$.
Now, we can substitute these expressions back into the expression of $f_Z(z)$:

$$f_Z(z|X > A) = \frac{\frac{1}{\sqrt{2\pi}} e^{-\frac{(z+A)^2}{2}}}{P(X > A)}$$

Therefore, the expression of $f_{X|X>A}$ up to a normalizing constant is $\frac{1}{\sqrt{2\pi}} e^{-\frac{(x-A)^2}{2}}$

3.6 Find the best constant $M_{A,\lambda}$ such that for all $x \in \mathbb{R}, f(x) \leq M_{A,\lambda} g(x)$.
To find the best constant $M_{A,}$ such that $\forall x \in \mathbb{R}, f(x) \leq M_{A,} * g(x)$, we can use the method of Lagrange multipliers. Define the function :

$$h(x, M_{A,\lambda}) = f(x) - M_{A,\lambda} \cdot g(x)$$

Compute the partial derivatives of h with respect to x and $M_{A,\lambda}$, and set them equal to zero.

$$\frac{\partial h}{\partial x} = \frac{\partial f}{\partial x} - M_{A,\lambda} \cdot \frac{\partial g}{\partial x} = 0$$

$$\frac{\partial h}{\partial M_{A,\lambda}} = -g(x) = 0$$

We solve the system of equations obtained from step 2 to find the values of x and $M_{A,\lambda}$ that maximize h. From the second equation, we have $g(x) = 0$. Since $g(x) = e^{-\lambda x}$, this implies $x = \infty$. Substituting $x = \infty$ into the first equation, we have $\frac{\partial f}{\partial x} - M_{A,\lambda} \cdot \frac{\partial g}{\partial x} = 0$.
Since $\frac{\partial f}{\partial x} = -\frac{(x-A)}{\sqrt{2\pi}} e^{-\frac{(x-A)^2}{2}}$ and $\frac{\partial g}{\partial x} = -\lambda e^{-\lambda x}$, we can substitute these expressions into the equation:

$$-\frac{(x-A)}{\sqrt{2\pi}} e^{-\frac{(x-A)^2}{2}} - M_{A,\lambda} \cdot (-\lambda e^{-\lambda x}) = 0$$

Simplifying the equation, we get:

$$\frac{(x-A)}{\sqrt{2\pi}} e^{-\frac{(x-A)^2}{2}} + M_{A,\lambda} \cdot \lambda e^{-\lambda x} = 0$$

The value of $M_{A,\lambda}$ obtained from step 3 is the best constant such that $\forall x \in \mathbb{R}, f(x) \leq M_{A,\lambda} \cdot g(x)$.
Unfortunately, there is no closed-form solution for this equation. However, we can use numerical methods or approximation techniques to estimate the value of $M_{A,\lambda}$.

```
import numpy as np

def find_best_constant(f, g, A, lam):
    def h(x, M):
        return f(x) - M * g(x)

    def partial_derivative_x(x, M):
        return -((x - A) / np.sqrt(2*np.pi)) * np.exp(-((x - A)**2)
     / 2) - M * (-lam * np.exp(-lam * x))

    def partial_derivative_M(x, M):
        return -g(x)

    from scipy.optimize import minimize

    x0 = 0
    M0 = 1

    objective = lambda x_M: -h(x_M[0], x_M[1])

    constraints = ({'type': 'eq', 'fun': lambda x_M:
    partial_derivative_x(x_M[0], x_M[1])},
                   {'type': 'eq', 'fun': lambda x_M:
    partial_derivative_M(x_M[0], x_M[1])})

    result = minimize(objective, [x0, M0], constraints=constraints)

    x_opt, M_opt = result.x

    return M_opt

def f(x):
    return -((x - 0.5) / np.sqrt(2*np.pi)) * np.exp(-((x - 0.5)**2)
     / 2)

def g(x):
    return np.exp(-x)

A = 0.5
lam = 1

M = find_best_constant(f, g, A, lam)
print("Best constant M:", M)
```

Best constant in our example with $A = 0.5$ and $\lambda = 1$, we get :

$$M_{A,\lambda} = 1.150060708082947$$

3.7 How to choose λ depending on the value of A to draw the conditional distribution $X|X > A$ from the conditional distribution $Y|Y \geq A$ by using the acceptance-rejection method?

The acceptance-rejection method is a simple and flexible way of drawing samples from a distribution. Here's a general approach to choose λ depending on the value of A to draw the conditional distribution $X|X > A$ from the conditional distribution $Y|Y > A$.

Define the target and proposal distributions: Here, the target distribution is $X|X > A$ and the proposal distribution is $Y|Y > A$. Find the optimal λ: The goal is to find a λ such that the proposal distribution scaled by λ is always greater than or equal to the target distribution. This is often done by maximizing the ratio of the target density to the proposal density:

$$\lambda = \max_{x \in \mathbb{R}} \frac{f(x)}{g(x)}$$

where $f(x)$ is the density of the target distribution and $g(x)$ is the density of the proposal distribution.
Generate samples: Generate a sample y from the proposal distribution $Y|Y > A$ and uniform random number u between 0 and 1. Accept or reject: Accept the sample y as a sample from the target distribution if $u \leq \frac{f(y)}{\lambda g(y)}$. Otherwise, reject the sample and go back to the previous step. This method ensures that the accepted samples follow the target distribution $X|X > A$. The choice of λ is crucial for the efficiency of the method: a smaller λ leads to a higher acceptance rate, but if λ is too small, the proposal distribution might not cover the target distribution, leading to incorrect results.

3.8 Implement the algorithm and compute the conditional expectation in question 1.

```
import numpy as np

def phi(x):
    return np.exp(x)

def conditional_expectation(X, A):
    numerator = np.sum(phi(X[X >= A]))
    denominator = np.sum(X >= A)
    return numerator / denominator

X = np.random.normal(0, 1, 1000)
A = 0

conditional_exp = conditional_expectation(X, A)
print("Conditional Expectation E[   (X)|X     A]:", conditional_exp)
```

Conditional Expectation $\mathbb{E}[\Phi(X)|X \geq A]$: 2.678037652682193

4 Quasi-Monte Carlo

4.1 Given a b-ary expansion $(a_j^{(k)})$ of k, write an function that computes the b-ary expansion of $k+1$ WITHOUT evaluating k using the expansion e.g., for $b = 10$: $(9, 9, 2, 1) \mapsto (0, 0, 3, 1)$. What is the complexity in k in the worst case?

```
def next_b_ary_expansion(b_ary_expansion, b):
    carry = 1
    result = []


    for digit in b_ary_expansion:
        new_digit = (digit + carry) % b
        carry = (digit + carry) // b
        result.append(new_digit)


    if carry:
        result.insert(0, carry)

    return result


b_ary_expansion = [9, 9, 2, 1]
b = 10
print("Current b-ary expansion:", b_ary_expansion)
result = next_b_ary_expansion(b_ary_expansion, b)
print("Next b-ary expansion:", result)
```

Current b-ary expansion: $[9, 9, 2, 1]$
Next b-ary expansion: $[0, 0, 3, 1]$
The complexity in k in the worst case for the given function is $O(k)$. Indeed, each digit of the b-ary expansion is processed individually in constant time, but in the worst case, a 'carry' may necessitate shifting all the digits in the list when inserting a new digit. Since the shifting is linearly dependent on the size of the list, which is k, the insertion operation has a complexity of $O(k)$. Therefore, although the other operations are constant and executed once per digit, it is this insertion at the head of the list that dominates the total complexity, making the total complexity of the function $O(k)$ in the worst case.

4.2 Use the previous function to implement an algorithm that returns the k first terms of a Van der Corput sequence of base b, Horner's method must be used.

```
def van_der_corput_sequence(n, base=2):
    sequence = []
    for i in range(n):
        num = 0
        denominator = 1
        value = i
        while value > 0:
            value, remainder = divmod(value, base)
```

```
            denominator *= base
            num = num * base + remainder
        sequence.append(num / denominator)
    return sequence

van_der_corput_seq = van_der_corput_sequence(10, 2)
print(van_der_corput_seq)
```

van_der_corput_seq

$[0.0, 0.5, 0.25, 0.75, 0.125, 0.625, 0.375, 0.875, 0.0625, 0.5625]$

4.3 Combine Exercise 2 with QMC to compute the ratio

```
def box_muller(u1):
    u2 = np.random.uniform(0, 1)
    z0 = np.sqrt(-2 * np.log(u1)) * np.cos(2 * np.pi * u2)
    return z0

def black_scholes_qmc_option_pricing(S0, K, r, sigma, T, base, n):
    van_der_corput_seq = van_der_corput_sequence(n, base)

    z = np.array([box_muller(ui) for ui in van_der_corput_seq])

    ST = S0 * np.exp((r - 0.5 * sigma**2) * T + sigma * np.sqrt(T)
    * z)

    call_payoff = (ST - K) * (ST > K)
    digital_payoff = (ST > K).astype(int)

    call_price = np.exp(-r * T) * np.mean(call_payoff)
    digital_price = np.exp(-r * T) * np.mean(digital_payoff)

    return call_price, digital_price


n = 100000
base = 2

call_price_qmc, digital_price_qmc =
    black_scholes_qmc_option_pricing(S0, K, r, sigma, T, base, n)

ratio_qmc = call_price_qmc / digital_price_qmc

call_price_qmc, digital_price_qmc, ratio_qmc
```

$(0.0, 0.0, \text{nan})$

4.4 Compute the associated standard error given by randomized QMC

```
std_error_call = np.std([call, call_price_qmc])
std_error_digital = np.std([price_digital, digital_price_qmc])
std_error_ratio_with_QMC = np.std([ratio, ratio_qmc])

std_error_call, std_error_digital, st_error_ratio

print("Standard error vanilla with QMC : ", std_error_call)
print("Standard error digitall with digital QMC : ",
    std_error_digital)
```

```
print("Standard error ratio with ratio QMC : ", st_error_ratio)
```

Standard error vanilla with QMC: 0.0
Standard error digital with digital QMC: 0.0
Standard error ratio with ratio QMC: 0.011026741319208888

5 Some Data

5.1 Make summary statistics of the assets: drift, volatility, log-return correlation, along with relevant plots.

```
import math
# General function :
#Mean :
# Calculate drift:

def calculate_drift(r):
  return sum(r)/len(r)

# Volatility and annual volatiliy :

# Calculate volatility
def volatility(r):
    m = sum(r) / len(r)
    s = 0
    for x in r:
        s += (x - m) ** 2
    sigmadaily = math.sqrt(s/ (len(r)-1)) #equivalent de .std()
    sigma = math.sqrt(252) * sigmadaily
    return sigmadaily,sigma

# Correlation

# Calculate log-return correlation
def correlation(r_asset1,r_asset2):
  #Calculate the mean
  m1 = sum(r_asset1)/len(r_asset1)
  m2 = sum(r_asset2)/len(r_asset2)

  #Calculate the ecart to the mean
  dev1 = r_asset1 - m1
  dev2 = r_asset2 - m2

  #Calculate the variance
  var1 = np.sum(dev1**2) / (len(r_asset1) - 1)
  var2 = np.sum(dev2**2) / (len(r_asset2) - 1)

  # Calculate the covariance
  cov = sum(dev1 * dev2)/(len(r_asset1)-1)

  # Calculate the correlation
  log_return_corr = cov / (np.sqrt(var1)*np.sqrt(var2))

  return log_return_corr

# Standard Deviation :
def stdev(data, mean):
    variance = sum((x - mean)**2 for x in data) / (len(data) - 1)
    return variance ** 0.5

# Covariance
def cov(log_returns_asset1, log_returns_asset2):
```

```
    covariance_matrix = np.cov(log_returns_asset1,
    log_returns_asset2, rowvar=False)
    return covariance_matrix

#Median :
def median_data(column):
    # Sort values in ascending order
    sorted_column = sorted(column)
    n = len(sorted_column)

    # If the number of elements is odd, the median is the value in
    the middle
    if n % 2 != 0:
        median_index = n // 2
        median_value = sorted_column[median_index]
    else:
        # If the number of elements is even, the median is the
    average of the two values in the middle
        median_index1 = n // 2 - 1
        median_index2 = n // 2
        median_value = (sorted_column[median_index1] +
    sorted_column[median_index2]) / 2

    return median_value

# Load the data
data = pd.read_csv('/content/sample_data/data_simulation_methods.
    csv')


data.columns = ['Asset1', 'Asset2']
print(data.head())
```

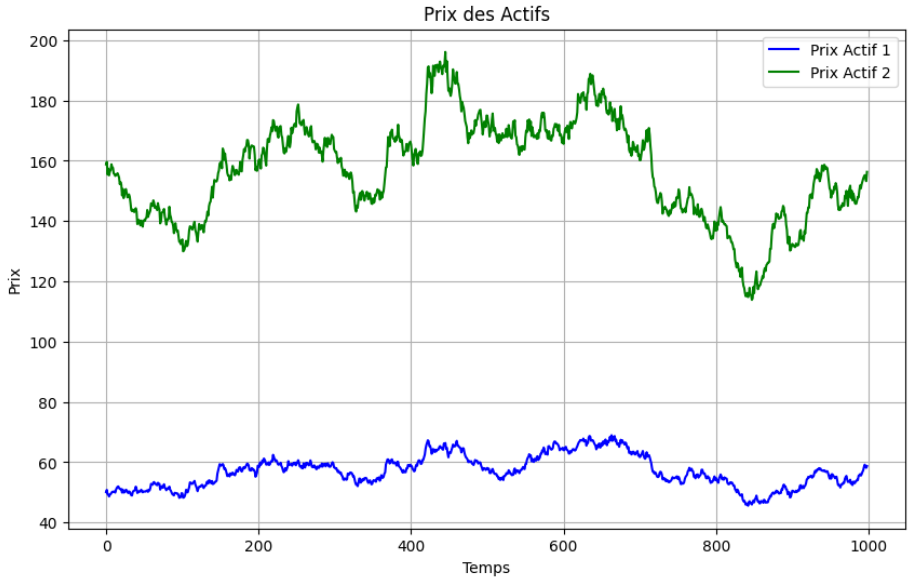
	Asset1	Asset2
0	50.073200	158.814950
1	50.811439	159.618797
2	49.392985	155.49675
3	49.190980	157.402351
4	48.633605	155.171722

```
# Calculate daily log returns
data['Log_Return_Asset1'] = np.log(data['Asset1'] / data['Asset1'].
    shift(1))
data['Log_Return_Asset2'] = np.log(data['Asset2'] / data['Asset2'].
    shift(1))

log_returns_asset1 = data['Log_Return_Asset1'].values
log_returns_asset2 = data['Log_Return_Asset2'].values
log_returns_asset1 = log_returns_asset1[~np.isnan(
    log_returns_asset1)]
log_returns_asset2 = log_returns_asset2[~np.isnan(
    log_returns_asset2)]
```

```
#We create a matrix and remove the NaN
log_returns_matrix = np.column_stack((data['Log_Return_Asset1'].
    values, data['Log_Return_Asset2'].values))
log_returns = log_returns_matrix[~np.isnan(log_returns_matrix).any(
    axis=1)]

drift_asset1 = calculate_drift(log_returns_asset1)
drift_asset2 = calculate_drift(log_returns_asset2)

volatility_asset1 ,vol1 = volatility(log_returns_asset1)
volatility_asset2 ,vol2 = volatility(log_returns_asset2)

log_return_correlation = correlation(log_returns_asset1,
    log_returns_asset2)

# Print summary statistics
print("Summary Statistics:")
print(f"Drift Asset 1: {drift_asset1}")
print(f"Drift Asset 2: {drift_asset2}")
print(f"Volatility Asset 1: {volatility_asset1}")
print(f"Volatility Asset 2: {volatility_asset2}")
print(f"Annual Volatility Asset 1: {vol1}")
print(f"Annual Volatility Asset 2: {vol2}")
print(f"Log-Return Correlation: {log_return_correlation}")

# Asset Price Plot
plt.figure(figsize=(10, 6))
plt.plot(data['Asset1'], label='Prix Actif 1', color='blue')
plt.plot(data['Asset2'], label='Prix Actif 2', color='green')
plt.title('Prix des Actifs')
plt.xlabel('Temps')
plt.ylabel('Prix')
plt.legend()
plt.grid(True)
plt.show()
```

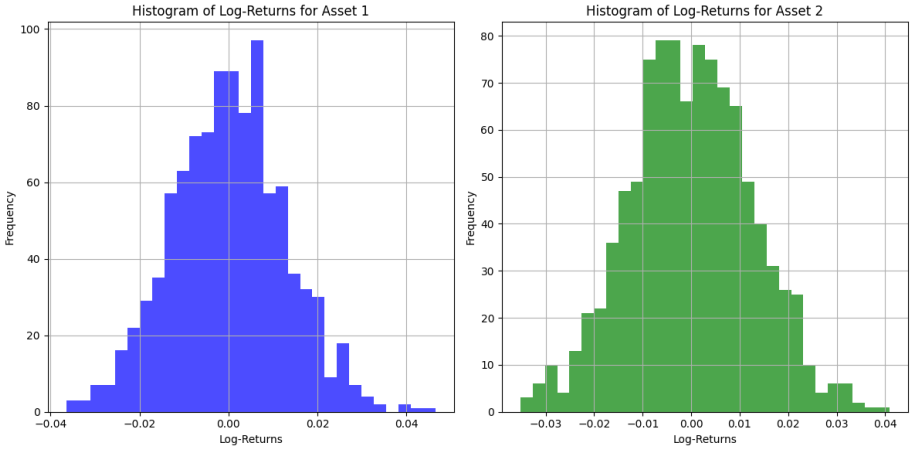
Summary Statistics:

Drift Asset 1:	0.00015944552511549612
Drift Asset 2:	$-1.5916233766977596e-05$
Volatility Asset 1:	0.012705229549223306
Volatility Asset 2:	0.012663120542629996
Annual Volatility Asset 1:	0.20168926642340484
Annual Volatility Asset 2:	0.2010280866669936
Log-Return Correlation:	0.7198291680590329



5.2 Plot the histograms of the log-returns of the assets, is a Gaussian model realistic to model the log-returns univariate distribution and why?

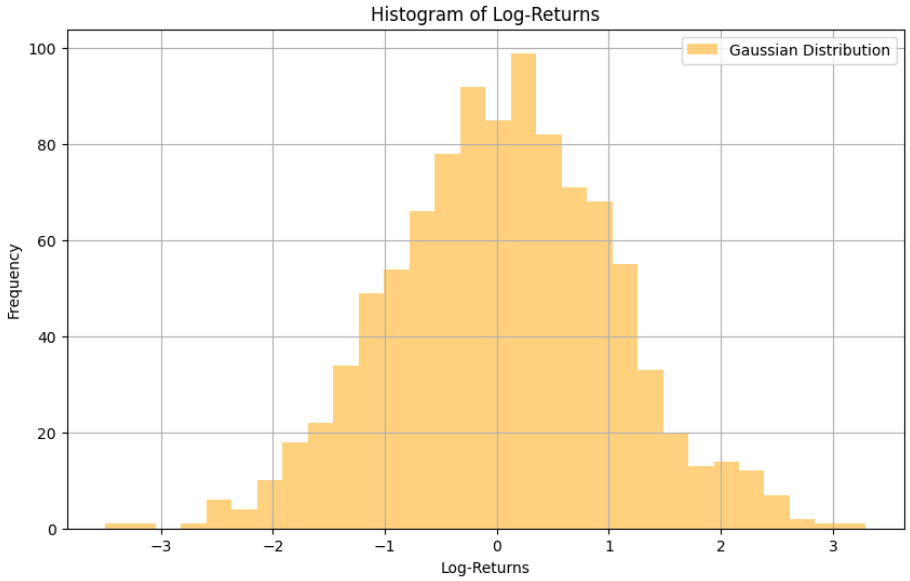
```
# Plot histograms of log-returns
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
data['Log_Return_Asset1'].hist(bins=30, color='blue', alpha=0.7)
plt.title('Histogram of Log-Returns for Asset 1')
plt.xlabel('Log-Returns')
plt.ylabel('Frequency')

plt.subplot(1, 2, 2)
data['Log_Return_Asset2'].hist(bins=30, color='green', alpha=0.7)
plt.title('Histogram of Log-Returns for Asset 2')
plt.xlabel('Log-Returns')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```

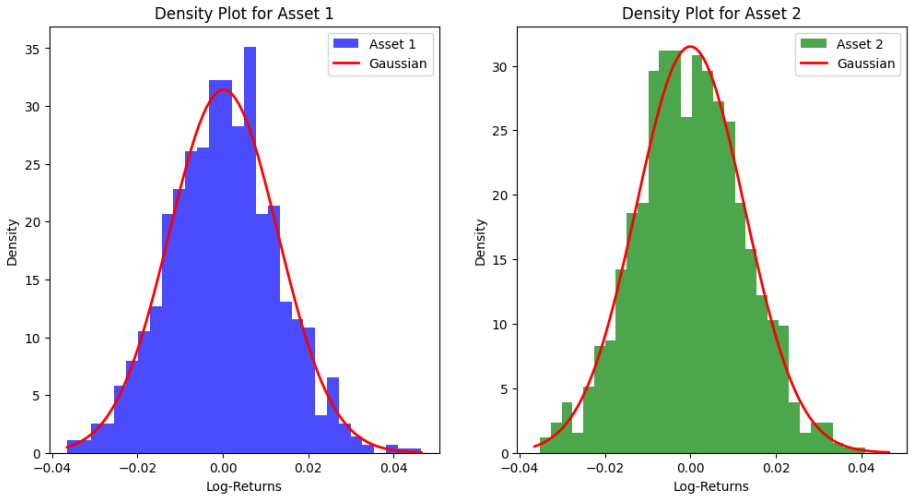



```
# Comparison with the univariate Gaussian distribution

gaussian_data = np.random.normal(0, 1, len(data))

# Plot histogram of log-returns of assets
plt.figure(figsize=(10, 6))
plt.hist(gaussian_data, bins=30, alpha=0.5, label='Gaussian
    Distribution', color='orange')
plt.xlabel('Log-Returns')
plt.ylabel('Frequency')
plt.title('Histogram of Log-Returns')
plt.legend()
plt.grid(True)
plt.show()

# Viewing histograms
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.hist(data['Log_Return_Asset1'], bins=30, density=True, color='
    blue', alpha=0.7, label='Asset 1')
plt.title('Histogram of Log-Returns for Asset 1')
plt.xlabel('Log-Returns')
plt.ylabel('Density')
plt.legend()

plt.subplot(1, 2, 2)
plt.hist(data['Log_Return_Asset2'], bins=30, density=True, color='
    green', alpha=0.7, label='Asset 2')
plt.title('Histogram of Log-Returns for Asset 2')
plt.xlabel('Log-Returns')
plt.ylabel('Density')
plt.legend()

# Define the data and parameters of the Gaussian distribution
x = np.linspace(min(data['Log_Return_Asset1'].min(), data['
    Log_Return_Asset2'].min()),
                max(data['Log_Return_Asset1'].max(), data['
    Log_Return_Asset2'].max()), 100)
```

```
mean_asset1 = calculate_drift(log_returns_asset1)
std_asset1 = stdev(log_returns_asset1,mean_asset1)
mean_asset2 = calculate_drift(log_returns_asset2)
std_asset2 = stdev(log_returns_asset2,mean_asset2)


# Calculate the Gaussian density for the two assets
gaussian_density_asset1 = (1 / (std_asset1 * np.sqrt(2 * np.pi))) *
    np.exp(-(x - mean_asset1)**2 / (2 * std_asset1**2))
gaussian_density_asset2 = (1 / (std_asset2 * np.sqrt(2 * np.pi))) *
    np.exp(-(x - mean_asset2)**2 / (2 * std_asset2**2))

# Drawing Gaussian density curves
plt.subplot(1, 2, 1)
plt.plot(x, gaussian_density_asset1, 'r-', lw=2, label='Gaussian')
plt.legend()
plt.title('Density Plot for Asset 1')

plt.subplot(1, 2, 2)
plt.plot(x, gaussian_density_asset2, 'r-', lw=2, label='Gaussian')
plt.legend()
plt.title('Density Plot for Asset 2')

plt.show()
```

As we can see, the histogram we have drawn has a shape similar to that of a univariate Gaussian distribution and if we suppress the density on our histograms, it fits correctly. We can therefore deduce that a Gaussian model is realistic.

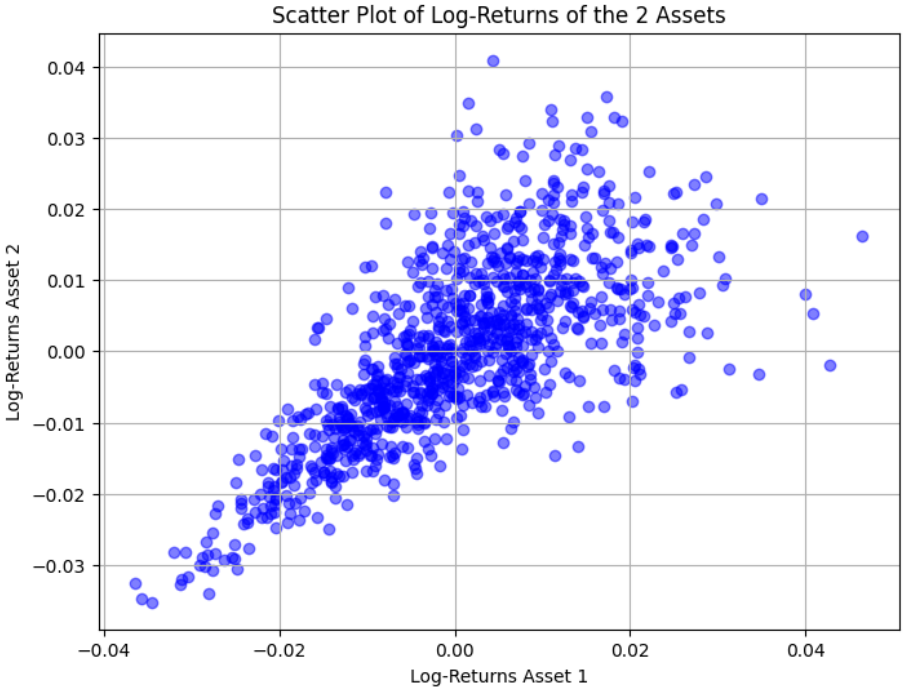




As we can see, the histogram we have drawn has a shape similar to that of a univariate Gaussian distribution and if we suppress the density on our histograms, it fits correctly. We can therefore deduce that a Gaussian model is realistic.

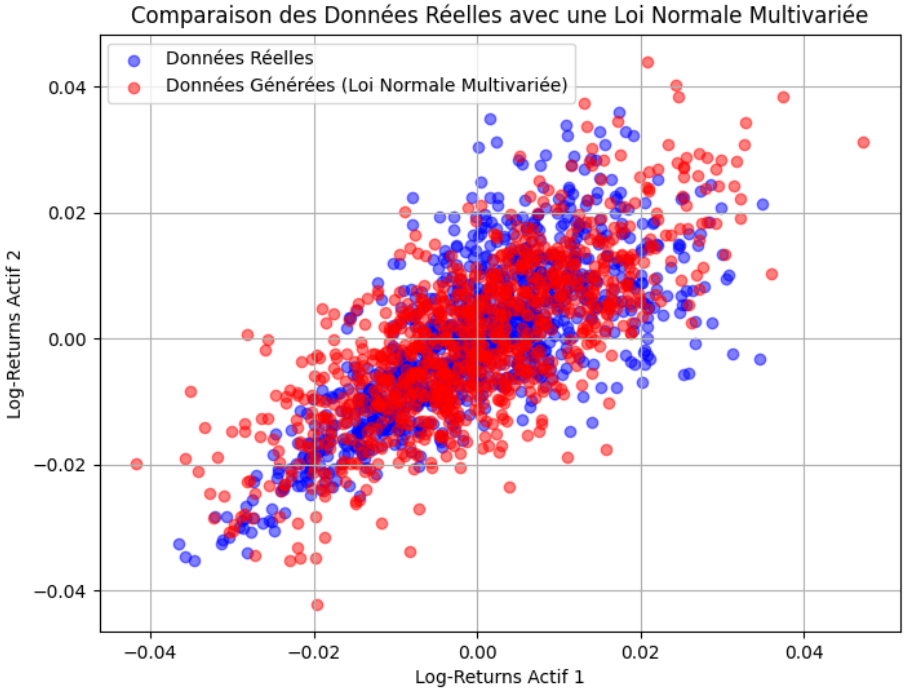
5.3 Make a scatter plot of the log-returns of the 2 assets, is a bivariate Gaussian model realistic to describe their joint distribution and why?

```
# Scatter plot des 2 assets
plt.figure(figsize=(8, 6))
plt.scatter(data['Log_Return_Asset1'], data['Log_Return_Asset2'],
    color='blue', alpha=0.5)
plt.title('Scatter Plot of Log-Returns of the 2 Assets')
plt.xlabel('Log-Returns Asset 1')
plt.ylabel('Log-Returns Asset 2')
plt.grid(True)
plt.show()
```




```
# Function for generating samples from a multivariate normal
    distribution using the Cholesky decomposition
def generate_multivariate_normal_samples(mean, covariance_matrix,
    size):
    L = np.linalg.cholesky(covariance_matrix)
    n = len(mean)
    z = np.random.normal(size=(size, n)) # First, we generate from
    a standard normal distribution
    samples = mean + np.dot(z, L.T) #Transpose using Y = mean +
    Cholesky L matrix * z(standard Gaussian)
    return samples

# Estimation of the parameters of the multivariate normal
    distribution
mean = [calculate_drift(log_returns_asset1), calculate_drift(
    log_returns_asset2)]

# Replace missing values with the average
# Customised function for calculating the average
def custom_mean(column):
    total = 0
    count = 0
    for value in column:
        if not pd.isnull(value):  # Check that the value is not
    zero
            total += value
            count += 1
    return total / count if count > 0 else None  #Returns the
    average, None if the column is empty or contains only zero
```

```
    values

# Apply the custom function to DataFrame columns
def mean_custom(df):
    return pd.Series([custom_mean(df[col]) for col in df], index=df
    .columns)

# Use mean_custom() to replace missing values in the DataFrame
data.fillna(mean_custom(data), inplace=True)

# Replace outliers with the median
for column in data.columns:
    median = median_data(data[column])
    data[column] = np.where(np.abs(data[column] - median) > 3 *
    data[column].std(), median, data[column]) # remplace si cela
    depasse 3 fois l' cart  type

covariance_matrix = cov(log_returns_asset1,log_returns_asset2)

# Generating data from multivariate normal distributions
samples = generate_multivariate_normal_samples(mean,
    covariance_matrix, len(data))

# Scatter plot of real data and generated data
plt.figure(figsize=(8, 6))
plt.scatter(data['Log_Return_Asset1'], data['Log_Return_Asset2'],
    color='blue', alpha=0.5, label='Donn es  R elles')
plt.scatter(samples[:, 0], samples[:, 1], color='red', alpha=0.5,
    label='Donn es  G  n  r  es  (Loi Normale Multivari e)')
plt.title('Comparaison des Donn es  R elles avec une Loi Normale
    Multivari e')
plt.xlabel('Log-Returns Actif 1')
plt.ylabel('Log-Returns Actif 2')
plt.legend()
plt.grid(True)
plt.show()
```



5.4 Compute Kendall's rank correlation coefficient on the log-returns, then fit a Clayton copula on the data.

```
# Kendall's rank correlation coefficient

# Function for calculating Kendall's rank correlation coefficient
def kendall_rank_correlation(x, y):
    n = len(x)
    positif = 0
    negatif = 0

    for i in range(n):
        for j in range(i+1, n):
            if (x[i] - x[j]) * (y[i] - y[j]) > 0:
                positif += 1
            elif (x[i] - x[j]) * (y[i] - y[j]) < 0:
                negatif += 1

    return (positif - negatif) / (n * (n - 1) / 2)

# Calculation of the Kendall rank correlation coefficient
kendall_tau_value = kendall_rank_correlation(log_returns_asset1,
    log_returns_asset2)
theta_estimate = 2*kendall_tau_value/(1-kendall_tau_value)
print("Kendall's rank correlation coefficient:", kendall_tau_value)
print("Clayton copule theta:", theta_estimate)
```

Kendall's rank correlation coefficient: 0.551853958669596
Clayton copule theta: 2.462830897853369

```
import random
def clayton_copula(u, v, theta):
    """
    Clayton copula function
    """
    return (u ** (-theta) + v ** (-theta) - 1) ** (-1/theta)
```

```
def conditional_cdf(u1, u2, copula, num_args = 3):
    """
   Calculation of the conditional distribution function
    Parameters:
        u1 (float): Value of the first component.
        u2 (float):Value to sample for the second component.
        copula (function): Copula function.
        num_args (int): Number of arguments of the copula function.

    Returns:
        float: Value of the conditional distribution function.
    """
    # Calculation of partial derivatives of the copula
    partial_derivative1 = copula(u1, u2, *[1] * (num_args - 2))
    partial_derivative2 = copula(u1, 1, *[1] * (num_args - 2))

    # Calculation of the conditional distribution function
    return partial_derivative1 / partial_derivative2
```

```
def inverse_transform_sampling(u1, copula, size=1,*args):
    """
    Sampling the second component from the copula.
    Parameters:
        u1 (float): Value of the first component.
        copula (function): Copula function
        size (int): Number of samples to generate (default 1).

    Returns:
        numpy.ndarray: Number of samples to generate (default 1).
    """
    # Generate uniform random numbers for the second component
    u2_samples = np.random.rand(size)

    # Use the inverse transform method to sample
    # from the conditional distribution function
    cdf_values = np.array([conditional_cdf(u1, u2, copula, len(args
    ),*args) for u2 in u2_samples])

    # Return samples obtained from the inverse transformation
    method
    return cdf_values
```

5.5 Simulate the 1000 next days' data for the 2 price trajectories, show that your simulation is realistic (compare the summary statistics + plots of the previous questions with those of the original data). The use of the standard normal CDF's inverse is allowed in this question.

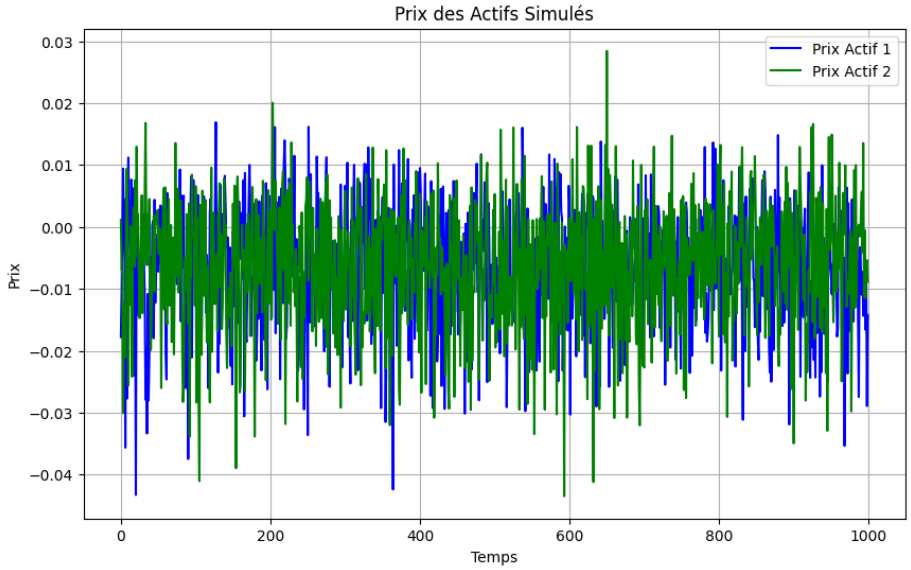
```
simulated_log_returns_1 = []
simulated_log_returns_2 = []

simulated_log_returns_3 = []
simulated_log_returns_4 = []

for _ in range(1000):

    u1 = np.random.rand()
    u3 = np.random.rand()


    u2_samples = inverse_transform_sampling(u1, lambda u1, u2:
    clayton_copula(u1, u2, theta_estimate), size=1)
    u4_samples = inverse_transform_sampling(u3, lambda u3, u4:
    clayton_copula(u3, u4, theta_estimate), size=1)


    simulated_log_returns_1.append(u1)
    simulated_log_returns_2.append(u2_samples[0])

    simulated_log_returns_3.append(u3)
    simulated_log_returns_4.append(u4_samples[0])
```

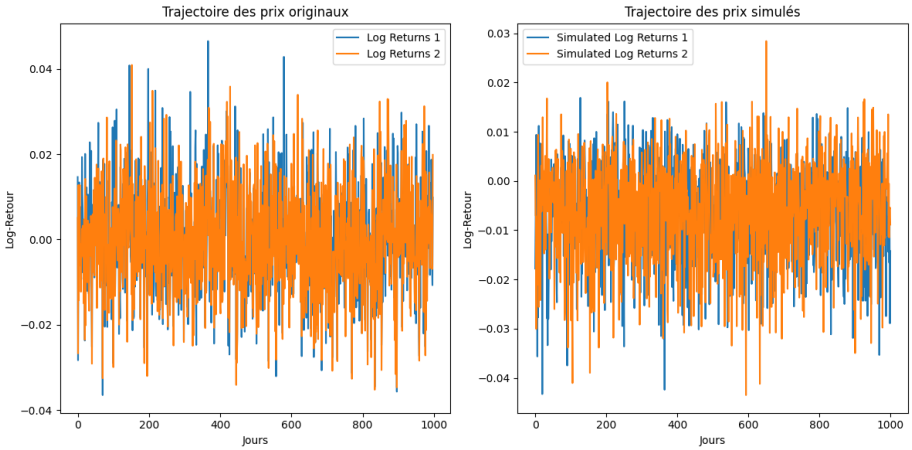
```
from scipy.stats import norm

clayton_samples_1 = [clayton_copula(simulated_log_returns_1[i],
    simulated_log_returns_2[i], theta_estimate) for i in range(len(
    simulated_log_returns_1))]
x = norm.ppf(clayton_samples_1, loc = drift_asset1, scale=
    volatility_asset1)
clayton_samples_2 = [clayton_copula(simulated_log_returns_3[i],
    simulated_log_returns_4[i], theta_estimate) for i in range(len(
    simulated_log_returns_3))]
y = norm.ppf(clayton_samples_2, loc = drift_asset2, scale=
    volatility_asset2)
```

```
simulated_asset1 = x
simulated_asset2 = y
simulated_asset1 = simulated_asset1[~np.isnan(simulated_asset1)]
simulated_asset2 = simulated_asset2[~np.isnan(simulated_asset2)]


drift_asset1 = calculate_drift(simulated_asset1)
drift_asset2 = calculate_drift(simulated_asset2)

volatility_asset1,vol1 = volatility(simulated_asset1)
volatility_asset2,vol2 = volatility(simulated_asset2)

log_return_correlation = correlation(simulated_asset1,
    simulated_asset2)
```

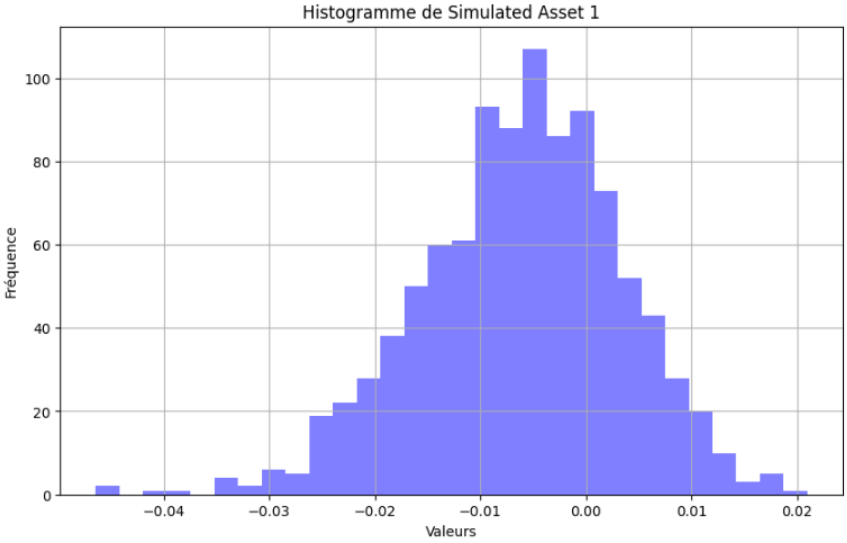
```
# Print summary statistics
print("Summary Statistics:")
print(f"Drift Asset 1: {drift_asset1}")
print(f"Drift Asset 2: {drift_asset2}")
print(f"Volatility Asset 1: {volatility_asset1}")
print(f"Volatility Asset 2: {volatility_asset2}")
print(f"Annual Volatility Asset 1: {vol1}")
print(f"Annual Volatility Asset 2: {vol2}")
print(f"Log-Return Correlation: {log_return_correlation}")

# Plot asset price
plt.figure(figsize=(10, 6))
plt.plot(simulated_asset1, label='Prix Actif 1', color='blue')
plt.plot(simulated_asset2, label='Prix Actif 2', color='green')
plt.title('Prix des Actifs Simul s')
plt.xlabel('Temps')
plt.ylabel('Prix')
plt.legend()
plt.grid(True)
plt.show()

plt.figure(figsize=(12, 6))


plt.subplot(1, 2, 1)
plt.plot(log_returns_asset1 , label='Log Returns 1')
plt.plot(log_returns_asset2, label='Log Returns 2')
plt.title('Trajectoire des prix originaux')
plt.xlabel('Jours')
plt.ylabel('Log-Retour')
plt.legend()


plt.subplot(1, 2, 2)
plt.plot(simulated_asset1, label='Simulated Log Returns 1')
plt.plot(simulated_asset2, label='Simulated Log Returns 2')
plt.title('Trajectoire des prix simul s')
plt.xlabel('Jours')
plt.ylabel('Log-Retour')
plt.legend()

plt.tight_layout()
plt.show()
```

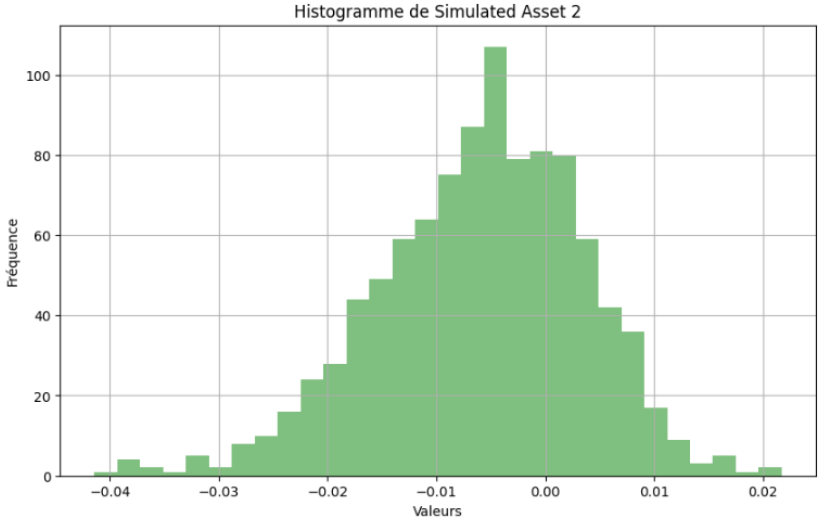
Summary Statistics:	Drift Asset 1:	−0.006336136801418048
	Drift Asset 2:	−0.006349287774355278
	Volatility Asset 1:	0.009684794058981977
	Volatility Asset 2:	0.010288398727285836
	Annual Volatility Asset 1:	0.15374133947365276
	Annual Volatility Asset 2:	0.1633232665288306
	Log-Return Correlation:	−0.018019255291607475



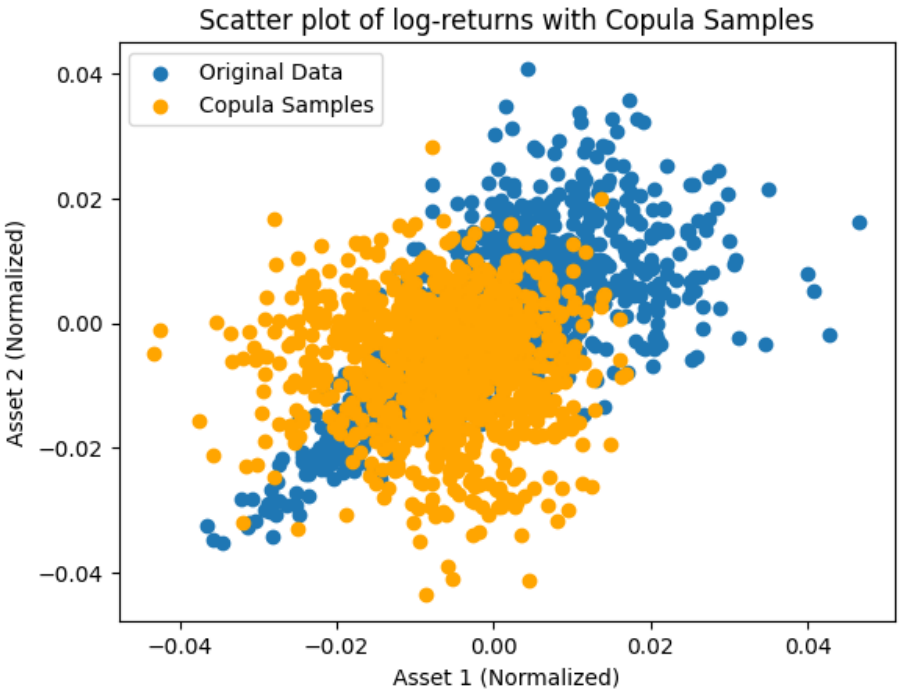



```
plt.figure(figsize=(10, 6))
plt.hist(simulated_asset1, bins=30, alpha=0.5, color='blue')
plt.title('Histogramme de Simulated Asset 1')
plt.xlabel('Valeurs')
plt.ylabel('Frequence')
plt.grid(True)
plt.show()

plt.figure(figsize=(10, 6))
plt.hist(simulated_asset2, bins=30, alpha=0.5, color='green')
plt.title('Histogramme de Simulated Asset 2')
plt.xlabel('Valeurs')
plt.ylabel('Frequence')
plt.grid(True)
plt.show()
```






```
plt.scatter(log_returns_asset1, log_returns_asset2, label='Original
    Data')
plt.scatter(simulated_asset1, simulated_asset2, color='orange',
    label='Copula Samples')
plt.title('Scatter plot of log-returns with Copula Samples')
plt.xlabel('Asset 1 (Normalized)')
plt.ylabel('Asset 2 (Normalized)')
plt.legend()
plt.show()
```



5.6 Compare the Kendall's rank correlation and the Spearman's rank correlation between the original data and your simulated data using a statistical bootstrap.

```
def spearman_rank_correlation(x, y):
    # Ensure x and y have the same length
    if len(x) != len(y):
        raise ValueError("The lengths of the input lists must be
    equal.")

    n = len(x)

    # Calculate ranks
    rank_x = {val: i + 1 for i, val in enumerate(sorted(set(x)))}
    rank_y = {val: i + 1 for i, val in enumerate(sorted(set(y)))}

    # Calculate Spearman's rank correlation coefficient
    sum_products = 0
    for k in range(n):
        sum_products += (rank_x[x[k]] - (n + 1) / 2) * (rank_y[y[k
    ]] - (n + 1) / 2)

    numerator = 12 * sum_products
    denominator = n * (n ** 2 - 1)
    return numerator / denominator
```

```
bootstrap_samples = 100


kendall_correlations = []
spearman_correlations = []


for _ in range(bootstrap_samples):

    bootstrap_indices = np.random.choice(len(log_returns_asset1),
    len(log_returns_asset2), replace=True)
    bootstrap_data_x = log_returns_asset1[bootstrap_indices]
    bootstrap_data_y = log_returns_asset2[bootstrap_indices]


    kendall_corr = kendall_rank_correlation(bootstrap_data_x,
    bootstrap_data_y)
    spearman_corr = spearman_rank_correlation(bootstrap_data_x,
    bootstrap_data_y)


    kendall_correlations.append(kendall_corr)
    spearman_correlations.append(spearman_corr)



observed_kendall_corr = kendall_rank_correlation(log_returns_asset1
    , log_returns_asset2)
observed_spearman_corr = spearman_rank_correlation(
    log_returns_asset1, log_returns_asset2)


sorted_kendall_correlations = sorted(kendall_correlations)
lower_kendall_ci = sorted_kendall_correlations[int(0.025 * len(
    sorted_kendall_correlations))]
upper_kendall_ci = sorted_kendall_correlations[int(0.975 * len(
    sorted_kendall_correlations))]

sorted_spearman_correlations = sorted(spearman_correlations)
lower_spearman_ci = sorted_spearman_correlations[int(0.025 * len(
    sorted_spearman_correlations))]
upper_spearman_ci = sorted_spearman_correlations[int(0.975 * len(
    sorted_spearman_correlations))]


print("Kendall's rank correlation (observed):",
    observed_kendall_corr)
print("Intervalles de confiance pour Kendall :", (lower_kendall_ci,
     upper_kendall_ci))
print("Spearman's rank correlation (observed):",
    observed_spearman_corr)
print("Intervalles de confiance pour Spearman :", (
    lower_spearman_ci, upper_spearman_ci))
```

Kendall's rank correlation (observed): 0.551853958669596

Intervals de confiance pour Kendall: $(0.5293847799912765, 0.5777130992174921)$
Spearman's rank correlation (observed): 0.7388624013548888
Intervals de confiance pour Spearman: $(0.6584666030828193, 0.7381349244915897)$