```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm, skew, kurtosis
```

Candidate number: 42416

In this project we will apply various numerical and computational methods in a quantitative financial context. In a first, we will cover sampling from the inverse transform method. In a second part, implementing and discuss several sampling techniques for stochastic processes and in a final part, the monte-carlo pricing theory along variance reduction. All numerical analysis is based on the lecture notes and seminar sheets.

1. Sampling from the inverse transform

Let X be a random variable with corresponding cdf

$$F(x)=\begin{cases} \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{y^2}{2}\right)dy, & \text{if } x<0, \\ \frac{3}{4}, & \text{if } 0\le x<3, \\ 1, & \text{if } x\ge 3. \end{cases}$$

This cdf is mixed distribution combining continuous and discrete components. We can generate a sample from F using the inverse transform method. In our case, F is piecewise and has point masses, so it is not strictly increasing everywhere. However, we can still determine $X=F^{-1}(U)$ given $U\sim U(0,1)$ by considering the different sections of the distribution separately. In order to generate samples, we use the following:

1. For $0\le U<0.5$, we have $F(x)=\Phi(x)$, where Φ is the standard normal CDF. Setting $\Phi(x)=U$, we can solve for x and get: $ X = \Phi^{-1}(U). $
2. For $0.5\le U<0.75$, $F(x)=0.75$ within the interval ¿. Since there is a jump at $x=0$ from $F(0^-)=0.5$ to $F(0)=0.75$, we have a point mass at $x=0$. Thus: $ X = 0. $
3. For $0.75\le U\le 1$, $F(x)=1$ for $x\ge 3$. Thus: $ X = 3. $

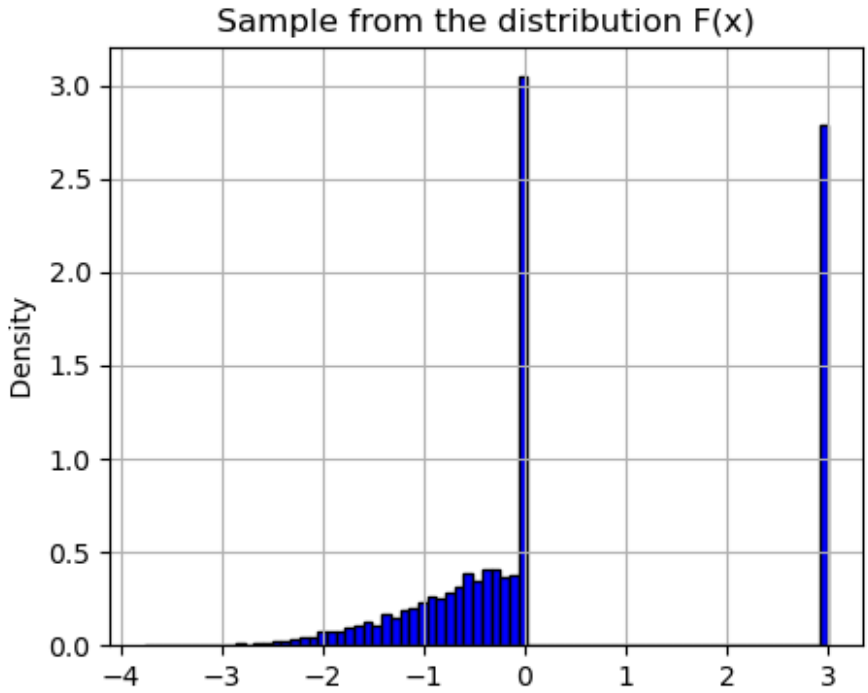
Hence, if $U_1, \ldots, U_n $ are i.i.d. $U(0,1)$ random variables, then the sample $X_1,\ldots,X_n$ defined as above is distributed according to F.

Therefore, to generate n independent samples $X_1,\ldots,X_n$ from F:

1. Generate n independent uniform random variables $U_1,\ldots,U_n\sim U(0,1)$
2. For each U_i, compute $X_i=F^{-1}(U_i)$ using the piecewise definition above

The following code outputs a histogram of the sampled values.

```
rng = np.random.default_rng(1234567)

def generate_sample(rng, samplesize):
    # We generate n ind. random variables U
    U = np.random.uniform(size=samplesize)
    normal_cdf_at_0 = norm.cdf(0)  # This is 0.5
    sample = []
    # For each U generated, we consider the conditions and append the
associated result
    for u in U:
        if u < normal_cdf_at_0:
            sample.append(norm.ppf(u))
        elif normal_cdf_at_0 <= u < 0.75:
            sample.append(0)
        else:
            sample.append(3)
    return sample

#Histogram Plot
fig , ax = plt . subplots ( nrows =1 , ncols =1 , figsize =(5 , 4) )
# Normalising the data so the overall density ==1
ax . hist ( generate_sample(rng, samplesize=10000) , bins =75,color =
"blue", density = True, edgecolor = "black")
ax . set_title (" Sample from the distribution F(x) ")
ax . set_ylabel (" Density ")
plt.grid()
plt . show ()
```



NB: we see a difference between the jumps at 0 and 3 due to our plotting behaviour. By using bins and collecting values around the plotted numbers, we interperet some of the normal distribution as belonging to the plot at 3

2. Analysis of SDE's

2.1 Generating sample paths for Stochastic Processes

We consider a stochastic process $\left(X_t\right)_{t\geq 0}$ defined by the stochastic differential equation (SDE):

$$d\,X_t=\mu\,X_t\,d\,t+\left(X_t-\gamma\,e^{\mu t}\right)\sigma\,d\,W_t,\,X_0=10,$$

where $\mu \in R$, $\sigma>0$, $\gamma \in (0,10)$, and $\left(W_t\right)_{t\geq 0}$ is a standard one-dimensional Brownian motion.

Our goal is to simulate a sample path of $\left(X_t\right)_{t\geq 0}$ on a discrete time grid from time 0 up to T.

In our algorithm we first define the time grid of the form:

$$0<h<2\,h<\ldots<N\,h=T,$$

where $h=\frac{T}{N}$ for $N \in N$, with N the number of steps in our simulation. Hence, $N\,h=T$.

We then introduce randomness. For $i=1,\ldots,n$, we denote by $Z_j^{(i)}$ by the j-th draw from the $N(0,1)$ distribution along the i-th path. We assume that the $Z_j^{(i)}$, $j=1,\ldots,N$, $i=1,\ldots,n$ are mutually independent random variables with $N(0,1)$ distribution. These independent standard normal variables, along with our step size is what define the random path of the Brownian Motion of our process of interest.

In order to simulate effectively the process we need to dicretise it by using some numerical schemes. We consequently introduce two different methods:

The First-Order Euler Scheme:

For $i=1,\ldots,n$, set $X_0^{\text{Euler},(i)}=X_0$ and define for all $j=0,\ldots,N-1$

$$X_{(j+1)h}^{(i)}=X_{jh}^{(i)}+\mu\,X_{jh}^{(i)}\,h+\sigma\left(X_{jh}^{(i)}-\gamma\,e^{\mu\,jh}\right)\sqrt{h}\,Z_{j+1}^{(i)}+\frac{1}{2}\sigma^2\left(X_{jh}^{(i)}-\gamma\,e^{\mu\,jh}\right)\left(\left(Z_{j+1}^{(i)}\right)^2-1\right)$$

The Milstein Scheme:

For $i=1,\ldots,n$, set $X_0^{\text{Milstein},(i)}=X_0$ and define for all $j=0,\ldots,N-1$

$$X_{(j+1)h}^{(i)}=X_{jh}^{(i)}+\mu\,X_{jh}^{(i)}\,h+\sigma\left(X_{jh}^{(i)}-\gamma\,e^{\mu\,jh}\right)\sqrt{h}\,Z_{j+1}^{(i)}+\frac{1}{2}\sigma^2\left(X_{jh}^{(i)}-\gamma\,e^{\mu\,jh}\right)\left(\left(Z_{j+1}^{(i)}\right)^2-1\right)$$

For these two schemes, we have different terms that drives the process, namely:

1. The drift term $f\left(X_t,t\right)=\mu\,X_{jh}^{(i)}h$

This term is present for both schemes and describes how the process changes over our very small time interval. It also gives the direction (positive or negative) and the rate of the process movement without randomness.

2. The diffusion term $g(X_t, t) = \sigma\left(X_{jh}^{(i)} - \gamma e^{\mu jh}\right)\sqrt{h}Z_{j+1}^{(i)}$

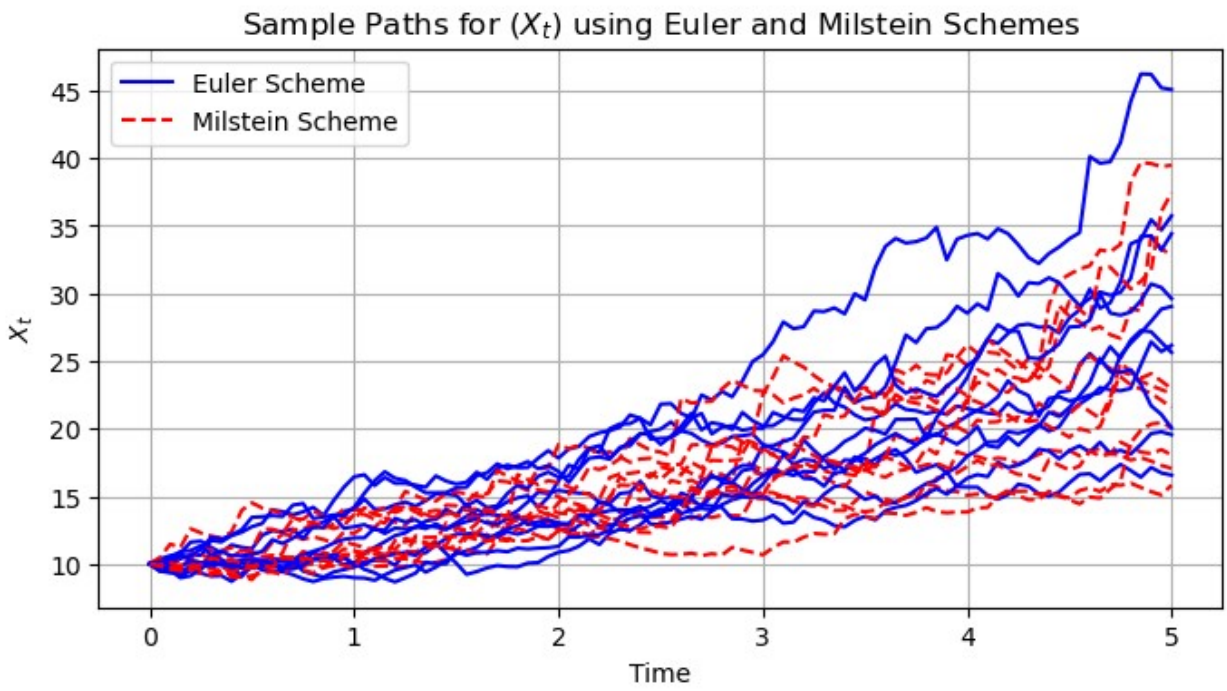
   This term represents the random component of the process and thus models deviations caused by uncertainty. It introduces some noise which is responsible for the deviation from the deterministc behavior of the drift. Our diffusion term is composed of σ (the volatility) which scales the intensity of the stochastic behavior and $\left(X_{jh}^{(i)} - \gamma e^{\mu jh}\right)$, with the time dependent factor $\gamma e^{\mu jh}$, which makes our diffusion evolve in terms of both the current state of the process and time.

3. The Milstein term $g'(X_t, t) = \sigma^2\left(X_{jh}^{\text{Milstein},(i)} - \gamma e^{\mu jh}\right)\left(\left(Z_{j+1}^{(i)}\right)^2 - 1\right)$

   It improves the strong convergence of the numerical solution by incorporating information about the non-linearity of the diffusion term. It represents a certain refinement from the first order Euler-scheme.

```
def Euler_samplepath(rng, s0, mu, sigma, gamma, T, nbsteps, n):

    """
    Simulates n sample paths of the SDE using the Euler scheme,
described above
    """

    h = T/nbsteps
    mynormal = rng.standard_normal(size = (n,nbsteps))
    samplepath = np.zeros((n, nbsteps +1))
    samplepath[:,0] = s0
    for i in range(nbsteps):
        samplepath[:, i+1] = (samplepath[:,i] + mu*samplepath[:,i] * h
+
                              (samplepath[:,i] - gamma*np.exp(mu*i*h)) *
                              sigma*mynormal[:,i]*np.sqrt(h))
    return samplepath

def Milstein_samplepath(rng, s0, mu, sigma, gamma, T, nbsteps, n):
    """
    this function simulates n sample paths of the SDE using the
Milstein scheme
    """
    h=T/nbsteps
    mynormal=rng.standard_normal(size = (n,nbsteps))
    # We create a numpy array of zeros and fix the initial condition
    samplepath= np.zeros((n, nbsteps+1))
    samplepath[:,0] = s0
```

```
    for k in range(nbsteps):
        tmp1 = samplepath[:,k] + mu*samplepath[:,k] * h  # drift term
        # computation of the diffusion term
        tmp2 = (samplepath[:,k] - gamma * np.exp(mu * k * h)) * sigma
* mynormal[:,k] * np.sqrt(h)
        # computation of the milstein correction term
        tmp3 = 0.5*(samplepath[:,k]  - gamma * np.exp(mu *k * h)) *
(sigma**2) * (mynormal[:,k]**2 - 1)
        #We update the sample path with its corresponding value within
the simulation
        samplepath[:,k+1]=  tmp1 + tmp2 + tmp3

    return samplepath

s0 = 10   #data
mu = 0.20
sigma = 0.2
T = 5
nbsteps = 100
n = 10
gamma = 2

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(8, 4))  #plot of
both of our schemes
time_grid = np.linspace(0, T, nbsteps + 1)

for _ in range(n):
    path1 = Euler_samplepath(rng, s0, mu, sigma, gamma, T, nbsteps, 1)
    path2 = Milstein_samplepath(rng, s0, mu, sigma, gamma, T, nbsteps,
1)
    ax.plot(time_grid, path1.T, label="Euler Path", color = "blue")
    ax.plot(time_grid, path2.T, label="Milstein Path", linestyle="--",
color= "red")

ax.set_xlabel("Time")
ax.set_ylabel("$X_t$")
ax.set_title("Sample Paths for ($X_t$) using Euler and Milstein
Schemes")
ax.legend(["Euler Scheme", "Milstein Scheme"], loc="upper left")
ax.grid()
plt.show()
```



From this plot of the sample paths generated by both schemes, we can see that the Milstein scheme shows more stability, generally staying within a lower range. This is the effect of the correction term. As time goes, the Euler Scheme shows more extreme values and signicant fluctuations. We can conclude that the Milstein scheme provides a more stable aproximation.

2.2 What about the Exact Simulation Scheme ?

In order to implement an exact scheme, we need an explicit solution. However the non-linearity of the process caused by the ¿) factor on the diffusion term prevents us from deriving the solution directly. Instead we need to substitute this factor and use Itô's Lemma, as follows:

The given SDE is:

$$d X_t = \mu X_t d t + \left(X_t - \gamma e^{\mu t} \right) \sigma d W_t, X_0 = 10.$$

Let

$$Z_t = X_t - \gamma e^{\mu t}$$

and consequently we get:

$$d Z_t = d X_t - \gamma e^{\mu t} d t$$

We then substitute $d X_t$, from the above equation, back into $d Z_t$ to get:

$$d Z_t = \mu X_t d t + \sigma X_t d W_t - \sigma \gamma e^{\mu t} d W_t - \gamma \mu e^{\mu t} d t$$

Replacing $X_t = Z_t + \gamma e^{\mu t}$:

$$dZ_t = \mu\left(Z_t + \gamma e^{\mu t}\right)dt + \sigma\left(Z_t + \gamma e^{\mu t}\right)dW_t - \sigma\gamma e^{\mu t}dW_t - \gamma\mu e^{\mu t}dt$$

And after cancelling all the terms involving $\gamma e^{\mu t}$, we finally get:

$$dZ_t = \mu Z_t dt + \sigma Z_t dW_t$$

which is a geometric brownian motion with general solution:

$$Z_t = Z_0 \exp\left(\left(\mu - \frac{\sigma^2}{2}\right)t + \sigma W_t\right)$$

From this we already know that $Z_0 = X_0 - \gamma e^{\mu \cdot 0} = 10 - \gamma$ and that $X_t = Z_t + \gamma e^{\mu t}$.

Substituting both of this result we get the explicit solution of the SDE for the process X_t:

$$X_t = (10 - \gamma)\exp\left(\left(\mu - \frac{\sigma^2}{2}\right)t + \sigma W_t\right) + \gamma e^{\mu t}$$

2.3.a Mathematical Analysis

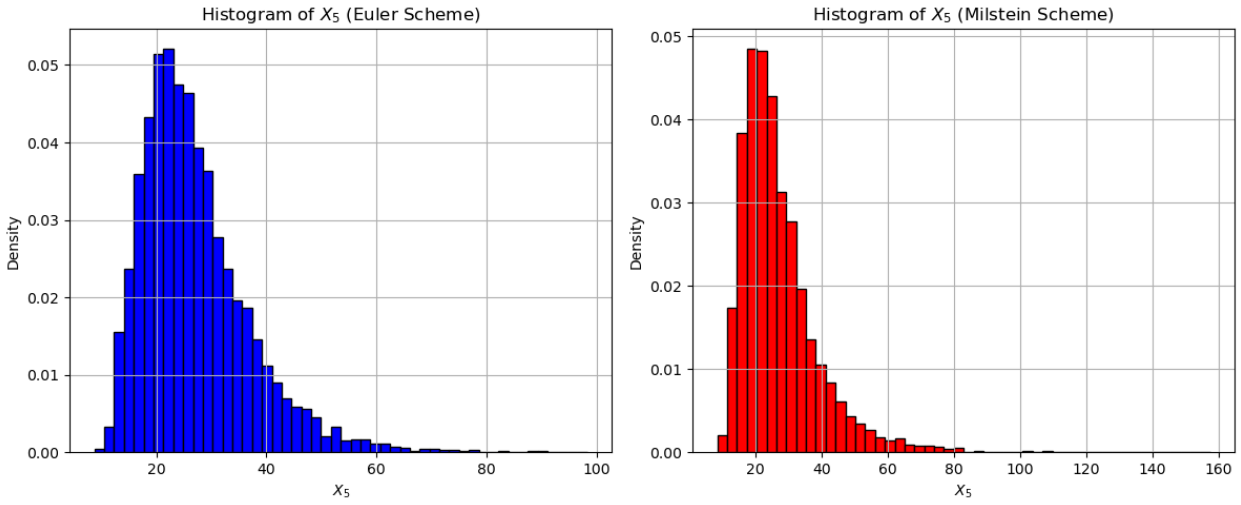
At time $t = 5$, the distribution of X_5 depends on how the drift μ, volatility σ, and parameter γ interact. A positive μ tends to push values of X_t upward. This is creating exponential growth in expectation. A negative μ brings the process down. The volatility σ introduces randomness that increases the spread of possible outcomes. In turn, it makes the distribution wider and potentially more skewed. The parameter γ sets a time-dependent component $\gamma e^{\mu t}$ around which the stochastic fluctuations happen. When X_t is close to $\gamma e^{\mu t}$, the diffusion is small and variability decreases, while larger deviations lead to increased randomness.

Together, these factors shape the distribution at $t = 5$. Higher σ and smaller μ create a more spread and highly skewed distribution. On the other hand, a lower σ and larger μ have a more concentrated distribution. This distribution grows closer to a deterministic exponential curve.

2.3.b Numerical Analysis

```
# We first simulate some paths
n = 10000
X_terminal_euler = Euler_samplepath(rng, s0, mu, sigma, gamma, T,
nbsteps, n)[:, -1]
X_terminal_milstein = Milstein_samplepath(rng, s0, mu, sigma, gamma,
T, nbsteps, n)[:, -1]


fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# we analyse the distribution of theEuler Scheme
axes[0].hist(X_terminal_euler, bins=50, color="blue", density=True,
edgecolor="black")
axes[0].set_title("Histogram of $X_5$ (Euler Scheme)")
axes[0].set_xlabel("$X_5$")
```

```
axes[0].set_ylabel("Density")
axes[0].grid()

# We analyse the distribution of the Milstein scheme
axes[1].hist(X_terminal_milstein, bins=50, color="red", density=True,
edgecolor="black")
axes[1].set_title("Histogram of $X_5$ (Milstein Scheme)")
axes[1].set_xlabel("$X_5$")
axes[1].set_ylabel("Density")
axes[1].grid()


plt.tight_layout()
plt.show()

# we compute statistics for Euler Scheme
print("Euler Scheme:")
print(f"Mean: {np.mean(X_terminal_euler):.2f}, Variance:
{np.var(X_terminal_euler):.2f}")
print(f"Skewness: {skew(X_terminal_euler):.2f}, Kurtosis:
{kurtosis(X_terminal_euler):.2f}")

# we compute statistics for Milstein Scheme
print("Milstein Scheme:")
print(f"Mean: {np.mean(X_terminal_milstein):.2f}, Variance:
{np.var(X_terminal_milstein):.2f}")
print(f"Skewness: {skew(X_terminal_milstein):.2f}, Kurtosis:
{kurtosis(X_terminal_milstein):.2f}")
```




```
Euler Scheme:
Mean: 27.04, Variance: 97.69
Skewness: 1.42, Kurtosis: 3.40
Milstein Scheme:
```

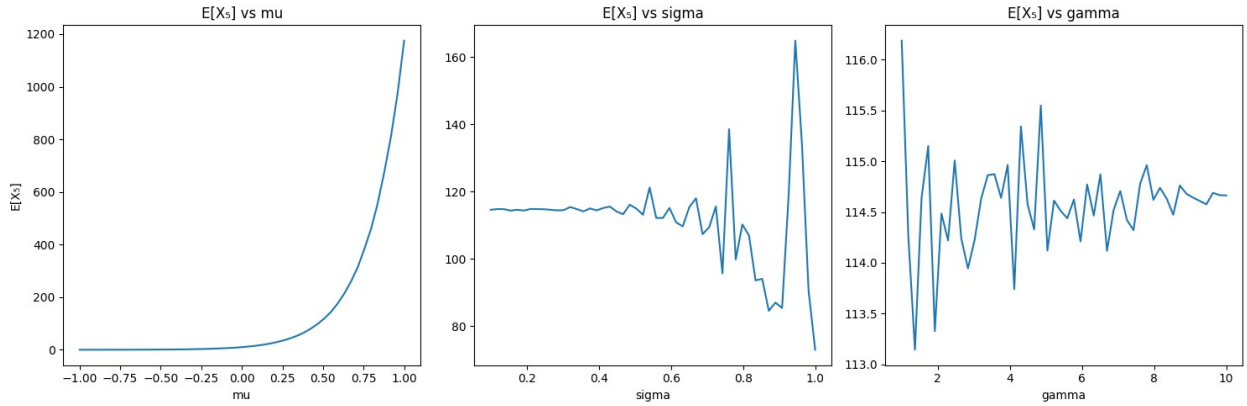
```
Mean: 26.98, Variance: 145.80
Skewness: 2.14, Kurtosis: 8.61
```

As we can observe in these graphs, the distribution of (X_5) looks like a shifted lognormal distribution because of the strong skewness, with mean centered around approximately 26. This shift happens because the process can be decomposed into a lognormal random component plus a deterministic component $\gamma\, e^{\mu t}$.

In the following, the Milstein scheme accounts for more precision (as shown above) and has thus been selected against the Euler scheme to study the evolution of $E[X_5]$ against a change in (γ, μ, σ)

```
# These plots aim to show the variability of the expected value of the
process at time 5
# when changes in parameters (mu, sigma, gamma) occure
# Firstly we choose our fixed parameters
s0, T = 10, 5
nbrsteps, n_samples = 50, 10000

plt.figure(figsize=(15, 5))

# We first plot E[X_5] vs mu
plt.subplot(131)
mu_range = np.linspace(-1, 1, 50)
mean_values_mu = []
for mu in mu_range:
    paths = Milstein_samplepath(rng, s0, mu, 0.3, 5, T, nbrsteps,
n_samples)
    mean_values_mu.append(np.mean(paths[:, -1]))
plt.plot(mu_range, mean_values_mu)
plt.title('E[X₅] vs mu')
plt.xlabel('mu')
plt.ylabel('E[X₅]')

# we now plot E[X_5] vs sigma
plt.subplot(132)
sigma_range = np.linspace(0.1, 1, 50)
mean_values_sigma = []
for sigma in sigma_range:
    paths = Milstein_samplepath(rng, s0, 0.5, sigma, 5, T, nbrsteps,
n_samples)
    mean_values_sigma.append(np.mean(paths[:, -1]))
plt.plot(sigma_range, mean_values_sigma)
plt.title('E[X₅] vs sigma')
plt.xlabel('sigma')

# We now plot E[X_5] vs gamma
plt.subplot(133)
gamma_range = np.linspace(1, 10, 50)
```

```
mean_values_gamma = []
for gamma in gamma_range:
    paths = Milstein_samplepath(rng, s0, 0.5, 0.3, gamma, T, nbrsteps,
n_samples)
    mean_values_gamma.append(np.mean(paths[:, -1]))
plt.plot(gamma_range, mean_values_gamma)
plt.title('E[X₅] vs gamma')
plt.xlabel('gamma')

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



In the first plot we can observe that μ has an exponential relationship with $E\left[X_5\right)$, specifically for μ > O.5. This aligns with the theoritical expectation $E\left[X_t\right)=s_0 \exp \mu t$. It is the term that drives the expected payoff the most In the second plot, we can see that the expected value is first stable but has an oscilatory response to changes in σ, specifically for σ >0.6. This indicates that the more the volatility increases the more it interacts with the current state dependent term $\gamma\, e^{\mu t}$. Finally, in the third plot, we can see that the change in γ only changes at most by 3.5 in the expected value and tends to 115 as γ goes to 10. This stabilisation tells us that, for large values of γ the process becomes more and more centered around $\gamma\, e^{\mu t}$.

3. Monte Carlo pricing and sensitivity analysis for an exotic derivative

3.1 Monte Carlo Estimation

Pricing Formula

The time-0 price of the derivative is the discounted expected payoff under the risk-neutral measure:

$$V_0=e^{-rT} E^Q ¿$$

Monte Carlo Estimator

Using N independent samples of the Brownian motion, we approximate V_0 as follows:

$$\widehat{V}_0 = e^{-rT}\frac{1}{N}\sum_{i=1}^{N}\left(K - \frac{1}{2}\left(S_{T/2,i}^{(1)} + S_{T,i}^{(1)}\right)\right)^{+} 1\{S_{T/2,i}^{(2)} < L\},$$

where $S_{T/2,i}^{(1)}$, $S_{T,i}^{(1)}$, and $S_{T/2,i}^{(2)}$ are simulated asset prices for the i-th path.

Confidence Interval

The asymptotic 99.5% confidence interval is:

$$\widehat{V}_0 \pm z_{0.995} \cdot \frac{\hat{\sigma}}{\sqrt{N}},$$

where $z_{0.995} \approx 2.576$ is the 99.5% quantile of the standard normal distribution, and $\hat{\sigma}$ is the sample standard deviation of the discounted payoff across the N simulations.

Simulation of Asset Prices

Discretisation

Using the geometric Brownian motion solution, the simulated prices are:

$$S_t^{(1)} = S_0^{(1)} \exp\left(\left(r - \frac{1}{2}\sigma_1^2\right)t + \sigma_1 W_t^{(1)}\right),$$

$$S_t^{(2)} = S_0^{(2)} \exp\left(\left(r - \frac{1}{2}\sigma_2^2\right)t + \sigma_2\left(\rho W_t^{(1)} + \sqrt{1-\rho^2}\, W_t^{(2)}\right)\right).$$

To simulate the random numbers:

- Simulate two independent standard normal variables $Z^{(1)}$ and $Z^{(2)}$ for each timestep.
- Construct correlated normal variables:

$$W_t^{(1)} = Z^{(1)}\sqrt{t}, W_t^{(2)} = \rho Z^{(1)}\sqrt{t} + \sqrt{1-\rho^2}\, Z^{(2)}\sqrt{t}.$$

Steps

1. Simulate $S_{T/2}^{(1)}$, $S_T^{(1)}$, and $S_{T/2}^{(2)}$ using the above formulas.
2. Compute the payoff for each path:

$$max\left(K - \frac{1}{2}\left(S_{T/2}^{(1)} + S_T^{(1)}\right), 0\right)^{+} 1\{S_{T/2}^{(2)} < L\}.$$

1. Average the discounted payoff to estimate V_0:

$$V_0 = e^{-rT}\frac{1}{N}\sum_{i=1}^{N}\text{Payoff}_i.$$

```
def generate_ST1_ST2(rng, s0_1, s0_2, r, t, sigma1, sigma2, rho,
samplesize):
    """
    This function allows us to simulate the paths of the risky assets
S1_T and S2_T
    which are essential for the computation of the time-0 price of the
derivative
    """
    # We first Simulate Brownian Motions for the first Asset
    mynormal1 = rng.standard_normal(size = samplesize)
    W1_T = np.sqrt(t)*mynormal1
    W1_T_over_2 = np.sqrt(t/2)*mynormal1

    # Then, we simulate the terminal values of S1_T and S1_(T/2)
    tmp1 = (r-0.5*sigma1**2)*t + sigma1*W1_T
    tmp2 = (r -0.5 * sigma1**2)*t/2 + sigma1 * W1_T_over_2
    S1_T = s0_1*np.exp(tmp1)
    S1_T_over_2 = s0_1*np.exp(tmp2)

    # Simulation of Brownian Motions for second Asset
    mynormal2 = rng.standard_normal(size = samplesize)
    W2_T = rho * mynormal1 + np.sqrt(1 - rho**2)*mynormal2*np.sqrt(t)
    W2_T_over_2 = rho * mynormal1 + np.sqrt(1 -
rho**2)*mynormal2*np.sqrt(t/2)

    # simulation of terminal values of S2_T and S2_(T/2)
    tmp3 = (r-0.5*sigma2**2)*t + sigma2*W2_T
    tmp4 = (r -0.5 * sigma2**2)*t/2 + sigma2 * W2_T_over_2
    S2_T = s0_2*np.exp(tmp3)
    S2_T_over_2 = s0_2*np.exp(tmp4)


    return S1_T, S1_T_over_2, S2_T, S2_T_over_2


def derivative_mcest(rng, s0_1, s0_2, r, t, sigma1, sigma2, rho,
samplesize, myepsilon, K, L):
    """
    This function uses the terminal values estimated by the function
above and returns
    the Monte-carlo simulated price of the derivative along with its
standard deviation and
    confidence interval.

    """
    # We first compute the payoff using outputs of the function above
and discount it
    myderivativesprices = generate_ST1_ST2(rng, s0_1, s0_2, r, t,
sigma1, sigma2, rho, samplesize)
    payoff = np.maximum(K - 0.5*(myderivativesprices[1]
```

```
                                    + myderivativesprices[0]), 0) *
(myderivativesprices[3] < L)
    discountedpayoff = np.exp(-r*t)*payoff
    # We then use the formula in step 2 to simulate the price of the
derivative
    price = np.mean(discountedpayoff)
    # And compute the standard deviation of the estimation with the
confidence interval
    standard_dev = np.std(discountedpayoff, ddof = 1)
    standard_dev_mc = standard_dev / np.sqrt(samplesize)
    aepsilon = norm.ppf(1 - myepsilon*0.5)
    ci_left = price - aepsilon * standard_dev_mc
    ci_right = price + aepsilon * standard_dev_mc
    return price, standard_dev_mc, ci_left, ci_right


s0_1 = 100
s0_2 = 90
r = 0.04
t = 1
sigma1 = 0.2
sigma2 = 0.5
rho = 0.75
samplesize = 100000
myepsilon = 0.005
K = 100
L = 80

# Compute price and confidence interval
price, std_dev_mc, ci_left, ci_right = derivative_mcest(rng, s0_1,
s0_2, r, t, sigma1, sigma2, rho, samplesize, myepsilon, K, L)
print(f"Price: {price:.4f}")
print(f"Standard Deviation (MC): {std_dev_mc:.4f}")
print(f"99.5% Confidence Interval: [{ci_left:.4f}, {ci_right:.4f}]")
```

```
Price: 4.7603
Standard Deviation (MC): 0.0252
99.5% Confidence Interval: [4.6896, 4.8310]
```

3.2. Variance reduction using control variate

The goal of the control variate method is to reduce the variance of our Monte-Carlo estimation of the derivative at t = 0. For this we need to define a control variate, a random variable X^i, that has a high correlation with the payoff of the derivative Y. It is defined as follow:

$$\overline{Y}_n(b) := \overline{Y}_n - b\left(\overline{X}_n - E[X]\right) = \frac{1}{n}\sum_{i=1}^{n}\left[Y_i - \hat{b}\left(X_i - E[X]\right)\right] = \frac{1}{n}\sum_{i=1}^{n} Y_i(b).$$

Consequently the choice of our control variate is crucial. It needs to be both, correlated with Y^i and have a known expected value. For the purpose of the question and taking into account our derivative, we have chosen three respective control variates, namely:

$$S_t^{(1)}, \frac{1}{2}\left(S_{T/2}^{(1)} + S_t^{(1)}\right), S_{T/2}^{(1)}$$

with which, we will return (for each estimator) the new estimated price, the confidence interval for that price and their standard deviation. All three of these estimators seems like good choices as they have a direct relationship with the payoff of the derivative. Additionally, under the risk neutral measure and because S_T follows a geometric brownian mottion, their respective expected value are known, namely:

$$E\left[\left(S_{T/2}^{(1)}\right)\right] = \exp(r\,t/2)\,S_0^1$$

$$E\left[\left(S_T^{(1)}\right)\right] = \exp(r\,t)\,S_0^1$$

$$E\left[\frac{1}{2}\left(S_{T/2}^{(1)} + S_T^{(1)}\right)\right] = \frac{1}{2}\left(\exp(r\,t/2) + \exp(r\,t)\right)S_0^1$$

We will then compare their respective variance and decide on which choice of X^i is the most efficient.

```
# problem 3c
# Control variates

def derivative_cv(rng, s0_1, s0_2, k, t, r, sigma1, sigma2, rho,
samplesize, myepsilon):
    """
        Control variate estimation for our derivative taking S1_T as
our control variate
    """
    # Generate terminal stock prices.
    myderivativesprices = generate_ST1_ST2(rng, s0_1, s0_2, r, t,
sigma1, sigma2, rho, samplesize)
    # Compute payoffs.
    payoffs = np.maximum(K - 0.5*(myderivativesprices[1]
                                  + myderivativesprices[0]), 0) *
(myderivativesprices[3] < L)
    # Discount payoffs
    discountedpayoffs = np.exp(- r * t)*payoffs
    # Use S1_T stock as control
    xs = np.exp(- r * t) * myderivativesprices[0]
    # Compute sample version of bhat
    bhat = np.cov(xs, discountedpayoffs, ddof=1)[0, 1] / np.var(xs,
ddof=1)
    # Define z= Y(bhat)
    z = discountedpayoffs - bhat * (xs - np.exp(r*t)*s0_1)
    # Compute MC price
```

```
    price = np.mean(z)
    # Compute confidence interval next
    standarddev_rv = np.std(z, ddof=1)
    standarddev_cvest = standarddev_rv / np.sqrt(samplesize)
    aepsilon = norm.ppf(1.0 - myepsilon * 0.5)
    ci_left = price - aepsilon * standarddev_cvest
    ci_right = price + aepsilon * standarddev_cvest

    return price, standarddev_cvest, ci_left, ci_right

CVresults = derivative_cv(rng, s0_1, s0_2, K, t, r, sigma1, sigma2,
rho, samplesize, myepsilon)
print('For Control Variate S1_T:')
print('CV price: {:.4f}; stdev of CV est: {:.4f}'.format(CVresults[0],
CVresults[1]))
print('CI based on CV is ({:.4f}, {:.4f})'.format(CVresults[2],
CVresults[3]))
```

```
For Control Variate S1_T:
CV price: 3.5532; stdev of CV est: 0.0173
CI based on CV is (3.5046, 3.6019)
```

```
def derivative_cv2(rng, s0_1, s0_2, k, t, r, sigma1, sigma2, rho,
samplesize, myepsilon):
    """
    Control variate estimation for our derivative taking (1/2)*(S1_T +
S1_T_over_2) as our control variate
    """
    # We first generate terminal stock prices.
    myderivativesprices = generate_ST1_ST2(rng, s0_1, s0_2, r, t,
sigma1, sigma2, rho, samplesize)
    # Compute payoffs.
    payoffs = np.maximum(K - 0.5*(myderivativesprices[1] +
myderivativesprices[0]), 0) * (myderivativesprices[3] < L)
    # Discount payoffs
    discountedpayoffs = np.exp(- r * t)*payoffs
    # We use (1/2)*(S1_T + S1_T_over_2) as control
    ys = 0.5*(myderivativesprices[0] + myderivativesprices[1])
    bhat = np.cov(ys, discountedpayoffs, ddof=1)[0, 1] / np.var(ys,
ddof=1)
    # Define z= Y(bhat)
    z = discountedpayoffs - bhat * (ys - 0.5*(np.exp(r*t) +
np.exp(r*t/2))*s0_1)
    # Compute MC price
    price = np.mean(z)
    standarddev_rv = np.std(z, ddof=1)
    standarddev_cvest = standarddev_rv / np.sqrt(samplesize)
    aepsilon = norm.ppf(1.0 - myepsilon * 0.5)
    # Left & Right boundary of CI
    ci_left = price - aepsilon * standarddev_cvest
```

```
    ci_right = price + aepsilon * standarddev_cvest
    return price, standarddev_cvest, ci_left, ci_right

CVresults2 = derivative_cv2(rng, s0_1, s0_2, K, t, r, sigma1, sigma2,
rho, samplesize, myepsilon)
print('For Control Variate (1/2)*(S1_T + S1_T_over_2):')
print('CV price: {:.4f}; stdev of CV est:
{:.4f}'.format(CVresults2[0], CVresults2[1]))
print('CI based on CV is ({:.4f}, {:.4f})'.format(CVresults2[2],
CVresults2[3]))
```

```
For Control Variate (1/2)*(S1_T + S1_T_over_2):
CV price: 4.6799; stdev of CV est: 0.0171
CI based on CV is (4.6319, 4.7279)
```

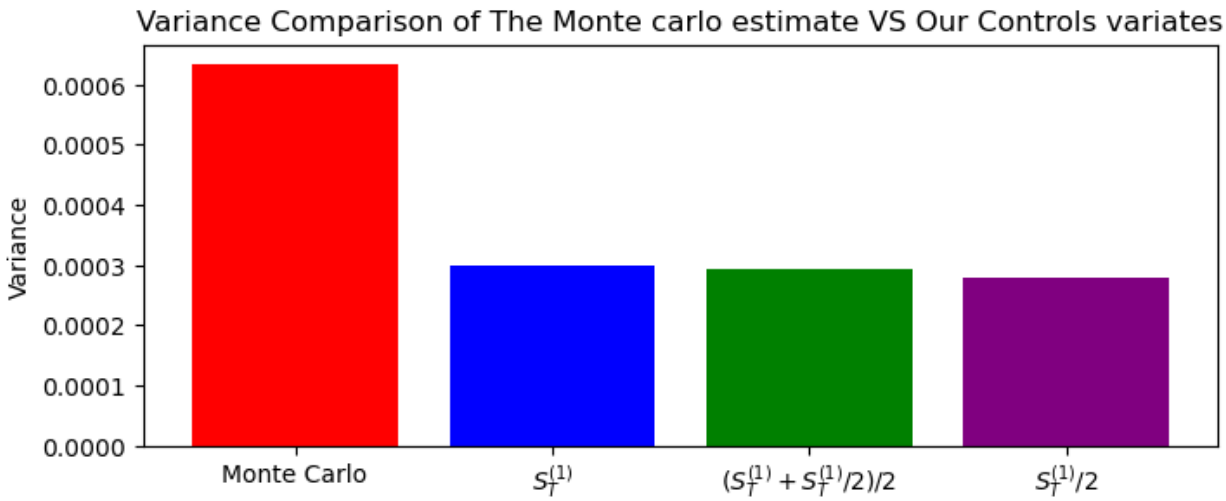
```
def derivative_cv3(rng, s0_1, s0_2, k, t, r, sigma1, sigma2, rho,
samplesize, myepsilon):
    """
    Control variate estimation for our derivative taking (S1_T_over_2)
as our control variate
    """
    # Generate terminal stock prices.
    myderivativesprices = generate_ST1_ST2(rng, s0_1, s0_2, r, t,
sigma1, sigma2, rho, samplesize)
    # Compute payoffs.
    payoffs = np.maximum(K - 0.5*(myderivativesprices[1]
                                  + myderivativesprices[0]), 0) *
(myderivativesprices[3] < L)
    # Discount payoffs
    discountedpayoffs = np.exp(- r * t)*payoffs
    # Use (S1_T_over_2) as control
    qs = (myderivativesprices[1])
    bhat = np.cov(qs, discountedpayoffs, ddof=1)[0, 1] / np.var(qs,
ddof=1)
    # Define z= Y(bhat)
    z = discountedpayoffs - bhat * (qs - np.exp(r*(t/2))*s0_1)
    # Compute MC price
    price = np.mean(z)
    standarddev_rv = np.std(z, ddof=1)
    standarddev_cvest = standarddev_rv / np.sqrt(samplesize)
    aepsilon = norm.ppf(1.0 - myepsilon * 0.5)
    # Left & right boundary of CI
    ci_left = price - aepsilon * standarddev_cvest
    ci_right = price + aepsilon * standarddev_cvest

    return price, standarddev_cvest, ci_left, ci_right

CVresults3 = derivative_cv3(rng, s0_1, s0_2, K, t, r, sigma1, sigma2,
rho, samplesize, myepsilon)
print('For Control Variate (S1_T_over_2):')
```

```
print('CV price: {:.4f}; stdev of CV est:
{:.4f}'.format(CVresults3[0], CVresults3[1]))
print('CI based on CV is ({:.4f}, {:.4f})'.format(CVresults3[2],
CVresults3[3]))
```

```
For Control Variate (S1_T_over_2):
CV price: 4.6884; stdev of CV est: 0.0167
CI based on CV is (4.6414, 4.7354)
```

```
# This is the compputation variances from standard deviations
var_mc = std_dev_mc**2
var_cv1 = CVresults[1]**2
var_cv2 = CVresults2[1]**2
var_cv3 = CVresults3[1]**2

# Here we print the comparison as well as the reduction ratio
print("\nVariance Comparison:")
print(f"Monte Carlo variance: {var_mc:.8f}")
print(f"CV method for S1_T: {var_cv1:.8f}")
print(f"CV method for 1/2(S1_T + S1_T/2): {var_cv2:.8f}")
print(f"CV method for S1_T/2: {var_cv3:.8f}")
print("\nVariance Reduction Ratios (compared to MC):")
print(f"CV method for S1_T: {var_mc/var_cv1:.2f}x")
print(f"CV method for 1/2(S1_T + S1_T/2): {var_mc/var_cv2:.2f}x")
print(f"CV method for S1_T/2:: {var_mc/var_cv3:.2f}x")

# The bar plot
methods = ['Monte Carlo', r'$S_T^{(1)}$',r'$(S_T^{(1)} +
S_T^{(1)}/2)/2$',r'$S_T^{(1)}/2$']
variances = [var_mc, var_cv1, var_cv2, var_cv3]
colors = ['red', 'blue', 'green', 'purple']

plt.figure(figsize=(8, 3))
plt.bar(methods, variances, color = colors)
plt.title('Variance Comparison of The Monte carlo estimate VS Our
Controls variates')
plt.ylabel('Variance')

plt.show()
```

```
Variance Comparison:
Monte Carlo variance: 0.00063427
CV method for S1_T: 0.00030052
CV method for 1/2(S1_T + S1_T/2): 0.00029244
CV method for S1_T/2: 0.00028046

Variance Reduction Ratios (compared to MC):
CV method for S1_T: 2.11x
CV method for 1/2(S1_T + S1_T/2): 2.17x
CV method for S1_T/2:: 2.26x
```



As shown in the bar graph and printed statistics, all of our choices of control variate have performed well in reducing the variance compared to the Monte-Carlo estimation. However $S_T^1/2$ (shown in purple) seems to be the most efficient within the tests. It reuced the variance by ≈ 174%. We can conclude that $S_T^1/2$ leads to the most precise estimation with the same computational effort.

2.3 Sensitivity Analysis

To approximate the sensitivity of the derivative's price $g(\rho)$, with respect to the correlation parameter ρ, we use the finite difference methods. By simulating the derivative's price at ρ , $\rho+h$ and $\rho-h$, we can approximate the first derivative $g'(\rho)$. without needing an analytical expression. Namely we will be implementing the two following formulas for values of ρ in the range [-1;1] :

The Forward difference approximation

$$g'(\rho) \approx \frac{g(\rho+h)-g(\rho)}{h}$$

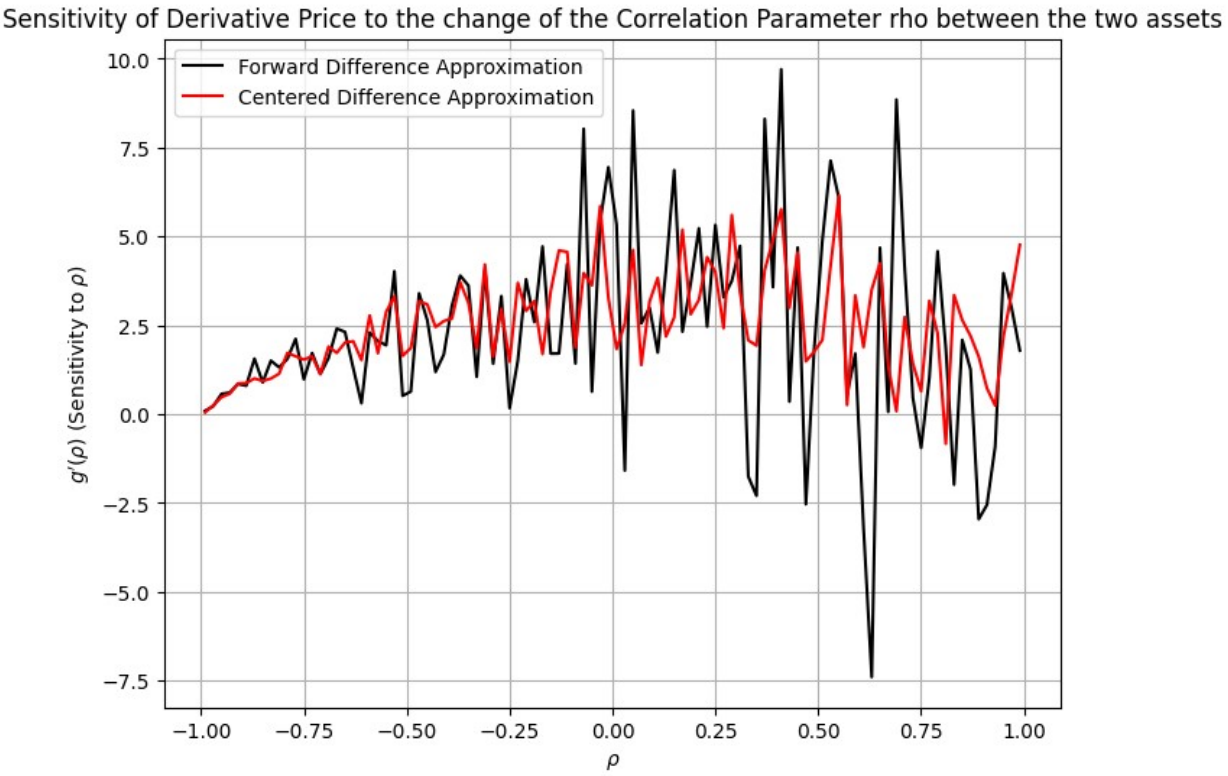
And the centered difference approximation

$$g'(\rho) \approx \frac{g(\rho+h)-g(\rho-h)}{2h}$$

Where h > 0, is the step size.

```
def sensitivity_rho_FD(rng, s0_1, s0_2, r, t, sigma1, sigma2, rho,
samplesize, myepsilon, K, L, h):
    # We first assign our derivative prices for the modified rhos
    rho_centre, _, _, _ = derivative_mcest(rng, s0_1, s0_2, r, t,
sigma1, sigma2, rho, samplesize, myepsilon, K, L)
    rho_right, _, _, _ = derivative_mcest(rng, s0_1, s0_2, r, t,
sigma1, sigma2, rho + h, samplesize, myepsilon, K, L)
```

```
    rho_left, _, _, _ = derivative_mcest(rng, s0_1, s0_2, r, t,
sigma1, sigma2, rho - h, samplesize, myepsilon, K, L)

    # Forward and centered finite differences
    sensitivity_forward = (rho_right - rho_centre) / h
    sensitivity_centered = (rho_right - rho_left) / (2*h)

    return sensitivity_forward, sensitivity_centered

rhos = np.linspace(-0.99, 0.99, 100)  # 100 points for rho within the
interval [-1, 1]
sensitivity_forward = []
sensitivity_centered = []

for rho in rhos:
    forward, centered = sensitivity_rho_FD(rng, s0_1, s0_2, r, t,
sigma1, sigma2, rho, samplesize, myepsilon, K, L, h=0.01)
    sensitivity_forward.append(forward)
    sensitivity_centered.append(centered)

sensitivity_forward = np.array(sensitivity_forward)
sensitivity_centered = np.array(sensitivity_centered)


plt.figure(figsize=(4, 3), dpi =100)
plt.plot(rhos, sensitivity_forward, label="Forward Difference
Approximation", color="black")
plt.plot(rhos, sensitivity_centered, label="Centered Difference
Approximation", color="red")
plt.xlabel(r"$\rho$")
plt.ylabel(r"$g'(\rho)$ (Sensitivity to $\rho$)")
plt.title("Sensitivity of Derivative Price to the change of the
Correlation Parameter rho between the two assets")
plt.legend(loc="best")

plt.grid()
plt.show()


# Question: this method relies on didvind the difference by the
stepsize h
```



From the above plot, we can see that the centered difference approximation oscilate with a smaller magnitude than the forward approximation as ρ changes between [-1 ; 1]. This indicates that the centered difference approach provides a more stable and reliable estimation of the derivative's sensitivity to ρ even though it shows a lot of rapid fluctuations.