# Data related innovation in finance : Deep pricing project

ABAAB Adel - BALLOUKA Jeff - COLOMINA FRANCES Chloé - FERMIGIER Thomas - MAREK Gwendoline

# Introduction

The purpose of this project is to compare classical pricing methods with new ones in terms of accuracy, robustness, and running time complexity.

- In the first part, we will implement an option valuation using the Monte Carlo method
- In the second part, we will implement a deep neural network in PyTorch using the differential machine learning framework seen in the course.
- In the third part, we will compare the results and give we conclusions.

For this project, we will use the default random number generator seeded by the sequence 123 and the packages numpy, matplotlib and PyTorch.

In [3]:
```
#Librairies
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(123)
import torch
```

# 1. Pricing and hedging by Monte Carlo

In this project,we suppose that the evolution of a single underlying asset $S_t$ is given by the Heston model :

$$\begin{cases} dS_t = rS_t dt + S_t\sqrt{V_t}dW_t^S \\ dV_t = \kappa(\bar{v} - V_t dt) + \gamma\sqrt{V_t}dW_t^V \\ dW_t^S dW_t^V = \rho_{S,V} dt \end{cases}$$

Where $dW_t^S$ and $dW_t^V$ are two independent Brownian motions. The process $V_t$ is known as Cox–Ingersoll–Ross (CIR) process and has been presented first to model the dynamics of interest rates. It has been discussed after to model the variance in the Heston model. It follows that the Heston model is a stochastic volatility model for which there is no closed-form formula contrary to the black and Scholes model. Thus, to price options under this model, we usually need to perform Monte Carlo simulations.

We consider that this model has already been calibrated and the optimized parameters are:

- Long run average variance $\bar{v}$ = 0.5
- κ = 0.1
- Vol-of-vol γ = 0.1
- Correlation of the two Brownian processes ρS,V = −0.9
- Risk-free interest rate r = 0.02
- Initial asset price S0 = 100
- Initial variance V0 = 0.1

From now, you will use these default parameters in all of this project. Note that the specific condition $2\kappa\bar{v} > \gamma^2$ is known as the Feller condition and ensures that the variance process cannot reach zero. In practice, this is not always true when we calibrate the model, so practitioners often use truncated schemes, reflecting schemes, or exact simulations of the CIR process by using the distribution of $V_t$.

## 1.1 Finite differences approach

### 1. Implement a standard Euler-Maruyama approximation to the Heston model. Define a function called "GeneratePathsHestonEuler()".

To answer this question, we implement a function called GeneratePathsHestonEuler. It takes in input: the model parameters, the number of paths and the number of steps. This function returns the output matrix paths.

The aim is to generate a matrix of simulated asset price paths using the Heston model with the Euler method.

In [4]:
```
def GeneratePathsHestonEuler(model_params, n_paths, n_steps):
    # Extract model parameters
    S0, kappa, theta, sigma, rho,gamma, V0, r,v_bar = model_params
    # Initialize the output matrix
    paths = np.zeros((n_paths, n_steps))
    # Set the initial values for S and V
    S = S0
    V = V0

    # Initialize time steps
    dt = 1/n_steps
    t = np.arange(dt, 1+dt, dt)

    # Loop through the number of steps
    for i in range(n_steps):
        # Generate two independent Brownian motions
        dW_S = np.random.normal(size=n_paths)
        dW_V = rho * dW_S + np.sqrt(1 - rho**2) * np.random.normal(size=n_paths)
        # Update the asset price and variance using the Heston model equations
        S = S + r * S * dt + np.sqrt(V * dt) * dW_S + np.sqrt(V) * dW_S
        V = V + kappa * (v_bar - V) * dt + gamma * np.sqrt(V * dt) * dW_V
        # Store the updated values of S and V in the output matrix
        paths[:, i] = S

    return paths
```

The function takes three input arguments:

model_params: a tuple containing the model parameters:

- S0: the initial asset price
- kappa: the mean-reversion rate of the variance
- theta: the long-run variance level
- sigma: the volatility of the variance
- rho: the correlation between the asset price and variance processes
- gamma: the volatility of the variance
- V0: the initial variance
- r: the risk-free interest rate
- v_bar: the long-run variance level

- n_paths: the number of simulated paths to generate
- n_steps: the number of time steps to use in the simulation

The function first initializes an output matrix paths of size (n_paths, n_steps) filled with zeros. It then sets the initial values of the asset price S and variance V to S0 and V0, respectively.

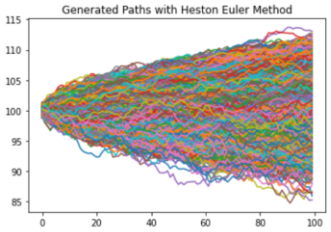
Next, the function sets up a time step dt and an array t of time steps from dt to 1+dt with a step size of dt.

The function then enters a loop that iterates over the number of time steps n_steps. On each iteration, the function generates two independent Brownian motions dW_S and dW_V using n_paths samples from the normal distribution. It then updates the asset price and variance using the Heston model equations and stores the updated values in the output matrix paths.

Finally, the function returns the output matrix paths.

Then, we decide to test the function GeneratePathsHestonEuler and to plot the asset prices obtained.

In [5]:
```
# TESTING FUNCTION

kappa = 0.1 #the rate at which it is pulled towards this long-term variance is κ this behavior makes sense intuitively also since volatility can't go to zero or infinity..
theta = 0.5 #theta: float, long-term mean of volatility
sigma = 0.2 #volatility
rho = -0.9 #rho: float, correlation between the asset price and the volatility-0.7
V0 =  0.1 #V_0: float, volatility of the underlying asset at the initial time
r = 0.02 # taux sans risque
gamma = 0.1 # volatility of the volatility
v_bar = 0.5 # long run average variance
S0 = 100 #initial price
model_params = S0, kappa, theta, sigma, rho,gamma, V0, r,v_bar
nb_paths = 10000
nb_steps = 100

#call the function GeneratePathsHestonEuler previously presented
asset_prices = GeneratePathsHestonEuler(model_params, nb_paths, nb_steps)

#PLOTTING PART
for i in range(0,len(asset_prices)):
    plt.plot(asset_prices[i])
plt.title("Generated Paths with Heston Euler Method")
plt.show()
```




This code is testing the GeneratePathsHestonEuler function defined above by using it to generate a matrix of simulated asset price paths using the Heston model with the Euler method.

The code first sets the values of the model parameters kappa, theta, sigma, rho, V0, r, gamma, v_bar, and S0. It then creates a tuple model_params containing these values and assigns it to the variable model_params.

The code also sets the number of simulated paths nb_paths and the number of time steps nb_steps to use in the simulation.

Next, the code calls the GeneratePathsHestonEuler function with the model_params, nb_paths, and nb_steps arguments and assigns the return value (a matrix of simulated asset price paths) to the variable asset_prices.

Finally, the code plots the simulated asset price paths by looping over the rows of the asset_prices matrix and using plt.plot to plot each row. The code then displays the plot using plt.show.

The results obtained seem logical with the input parameters given in the subject. The price simulated are between an interval of [85;115].

## 2. Implement a function "Payoff" that returns the payoff of the down-and-out barrier call option.

Barrier options are financial derivatives whose payoffs depend on the crossing of a certain predefined barrier level by the underlying asset price process $(S_t)_t \in [0, T]$. Let the discounted payoff function be a down-and-out barrier call option defined as

$$g = exp(-rT)max(0, S_T - K)1_{\{min_{0\leq t\leq T} S_t > B\}}$$

Concretely, this means that the option is exercisable if the price of the underlying has not broken the barrier. Thus, a down-and-out barrier call is a path dependent option since if the underlying asset reaches the barrier during the option's life, then the option is terminated and will never come back into existence. We consider that the option has the following characteristics:

- Strike K = 100
- Barrier B = 90
- Maturity T = 1

This function takes the option parameters and the asset prices in argument and returns the payoff of the option.

In [6]:
```
def Payoff(asset_prices, K, H, T, r):
    # Initialize payoffs array
    payoffs = np.empty(asset_prices.shape[0])

    # Iterate over rows of asset_prices
    for i in range(asset_prices.shape[0]):
        # Compute the payoff of the down-and-out barrier call option
        payoff = np.maximum(asset_prices[i, -1]-K, 0)
        if asset_prices[i, :].min() < H: #if the path go bellow the barrier
            payoff = 0
        payoffs[i] = np.exp(-r*T)*payoff

    return payoffs
```

This code defines a function Payoff that calculates the payoffs of a down-and-out barrier call option for a given set of asset price paths.

The function takes five input arguments:

asset_prices: a matrix of simulated asset price paths

- K: the strike price of the option
- H: the barrier level of the option
- T: the time to expiration of the option

- r: the risk-free interest rate

The function first initializes an array payoffs of the same size as the number of rows in asset_prices using np.empty.

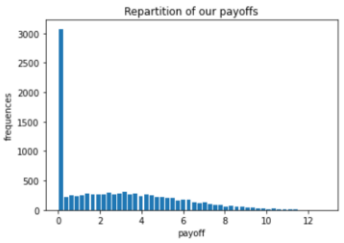
The function then enters a loop that iterates over the rows of asset_prices. On each iteration, the function calculates the payoff of the down-and-out barrier call option for the current asset price path. If the minimum asset price in the path is less than the barrier level H, the payoff is set to 0; otherwise, the payoff is equal to the difference between the final asset price in the path and the strike price K (if this value is negative, the payoff is set to 0). The function then stores the payoff in the payoffs array.

Finally, the function returns the payoffs array.

Then, we decide to test the function Payoff and to plot the repartition of the payoff obtained.

In [7]:
```
# TESTING FUNCTION
K = 100
H = 90
T = 1

#call the function Payoff previously presented
pay = Payoff(asset_prices, K, H, T, r)
plt.hist(pay,bins=50,rwidth = 0.8)
plt.xlabel('payoff')
plt.ylabel('frequences')
plt.title("Repartition of our payoffs")
plt.show
```

Out[7]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```




This code is testing the Payoff function defined above by using it to calculate the payoffs of a down-and-out barrier call option for a given set of asset price paths.

The code sets the values of the strike price K, the barrier level H, and the time to expiration T of the option.

Next, the code calls the Payoff function with the asset_prices, K, H, T, and r arguments and assigns the return value (an array of payoffs) to the variable pay.

Finally, the code uses plt.hist to plot a histogram of the payoffs in the pay array and displays the plot using plt.show. The histogram shows the frequency of each payoff value in the pay array.

## 3. Implement a function "MC Pricing()".

This function takes the number of paths, the number of steps, the model parameters, and the option parameters in argument and returns the price of the option computed by Monte Carlo. This function calls "GeneratePathsHestonEuler()" and "Payoff()".

In [8]:
```
def MC_Pricing(nb_paths, nb_steps, model_params, option_params):
    # Generate asset price paths using the Euler-Maruyama approximation
    asset_prices = GeneratePathsHestonEuler(model_params, nb_paths, nb_steps)

    # Extract option parameters
    K, H, T, r = option_params

    # Compute the payoff of the option
    pay = Payoff(asset_prices, K, H, T, r)

    # Return the price of the option as the average of the payoffs
    return np.mean(pay)
```

This code defines a function MC_Pricing that uses the Monte Carlo method to estimate the price of a down-and-out barrier call option.

The function takes four input arguments:

- nb_paths: the number of simulated paths to use in the Monte Carlo simulation
- nb_steps: the number of time steps to use in the simulation
- model_params: a tuple containing the model parameters for the Heston model with the Euler method (described above)
- option_params: a tuple containing the option parameters:
  K: the strike price of the option
  H: the barrier level of the option
  T: the time to expiration of the option
  r: the risk-free interest rate

The function first generates a matrix of simulated asset price paths using the GeneratePathsHestonEuler function and the model_params input arguments. It then extracts the option parameters K, H, T, and r from the option_params input.

Next, the function calls the Payoff function with the asset_prices, K, H, T, and r arguments to calculate the payoffs of the option for the simulated asset price paths.

Finally, the function returns the price of the option as the average of the payoffs using np.mean. This is a common technique in Monte Carlo simulation to estimate the expected value of a random variable.

**TESTING FUNCTION**

In [9]:
```
option_params = K, H, T, r
MC_price = MC_Pricing(nb_paths, nb_steps, model_params, option_params)
print(MC_price)
```

```
2.7576801918391087
```

This code is using the MC_Pricing function defined above to estimate the price of a down-and-out barrier call option using the Monte Carlo method.

The code first defines a tuple option_params containing the option parameters K, H, T, and r. It then calls the MC_Pricing function with the nb_paths, nb_steps, model_params, and option_params arguments and assigns the return value (the estimated price of the option) to the variable MC_price.

Finally, the code prints the value of MC_price. This should be the estimated price of the down-and-out barrier call option using the Heston model with the Euler method and the specified input parameters.

## 4. Implement three functions "DeltaFD()", "GammaFD()", "RhoFD()" which estimate the delta, the gamma, and the rho of the option by using first-order finite differences (i.e. "bumping").

We want to compute the greeks corresponding to the first and second derivatives of C (being the expected value of the call option) with respect to various parameters :

- $\delta = \frac{\partial C}{\partial S_0}$
- $\gamma = \frac{\partial C^2}{\partial S_0^2}$
- $\rho = \frac{\partial C}{\partial r}$

In [10]:
```python
def DeltaFD(nb_paths, nb_steps, model_params, option_params, epsilon):
    # Compute the option price with the original model parameters
    option_price = MC_Pricing(nb_paths, nb_steps, model_params, option_params)

    # Bump the asset process mean-reversion rate
    S0, kappa, theta, sigma, rho,gamma, V0, r,v_bar = model_params
    model_params_bumped = (S0 + epsilon, kappa , theta, sigma, rho,gamma, V0, r,v_bar)
    option_price_bumped = MC_Pricing(nb_paths, nb_steps, model_params_bumped, option_params)

    # Return the finite difference estimate of the delta
    return (option_price_bumped - option_price)/epsilon

def GammaFD(nb_paths, nb_steps, model_params, option_params, epsilon):
    # Compute the option price with the original model parameters
    option_price = MC_Pricing(nb_paths, nb_steps, model_params, option_params)

    # Bump the asset process mean-reversion rate
    S0, kappa, theta, sigma, rho,gamma, V0, r,v_bar = model_params
    model_params_bumped = (S0 + epsilon, kappa , theta, sigma, rho,gamma, V0, r,v_bar)
    option_price_bumped = MC_Pricing(nb_paths, nb_steps, model_params_bumped, option_params)

    # Bump the asset process mean-reversion rate again
    model_params_bumped2 = (S0 - epsilon, kappa, theta, sigma, rho,gamma, V0, r,v_bar)
    option_price_bumped2 = MC_Pricing(nb_paths, nb_steps, model_params_bumped2, option_params)

    # Return the finite difference estimate of the gamma
    return (option_price_bumped - 2*option_price + option_price_bumped2)/(epsilon**2)

def RhoFD(nb_paths, nb_steps, model_params, option_params, epsilon):
    # Compute the option price with the original model parameters
    option_price = MC_Pricing(nb_paths, nb_steps, model_params, option_params)

    # Bump the risk-free interest rate
    K, H, T, r = option_params
    option_params_bumped = (K, H, T, r+epsilon)
    option_price_bumped = MC_Pricing(nb_paths, nb_steps, model_params, option_params_bumped)

    # Return the finite difference estimate of the rho
    return (option_price_bumped - option_price)/epsilon
```

These functions use the finite difference method to approximate the sensitivities (or "greeks") of a down-and-out barrier call option with respect to various model parameters. The greeks are commonly used to measure the sensitivity of an option's price to changes in the underlying model parameters.

The functions take the following input arguments:

- nb_paths: the number of simulated paths to use in the Monte Carlo simulation
- nb_steps: the number of time steps to use in the simulation
- model_params: a tuple containing the model parameters for the Heston model with the Euler method (described above)

option_params: a tuple containing the option parameters:

- K: the strike price of the option
- H: the barrier level of the option
- T: the time to expiration of the option
- r: the risk-free interest rate
- epsilon: a small number used to compute the finite differences

The DeltaFD function approximates the delta of the option, which is the sensitivity of the option price to changes in the asset price. The function bumps the initial asset price S0 by epsilon and re-prices the option using the bumped asset price. The finite difference estimate of the delta is then computed as the difference between the bumped option price and the original option price, divided by epsilon.

The GammaFD function approximates the gamma of the option, which is the sensitivity of the option delta to changes in the asset price. The function bumps the initial asset price S0 by epsilon twice, re-prices the option using the bumped asset prices, and computes the finite difference estimate of the gamma as a second

The RhoFD function approximates the rho of the option, which is the sensitivity of the option price to changes in the interest rate.The finite difference estimate of the rho is then computed as the difference between the bumped option price and the interest rate, divided by epsilon.

## 5. For the three sensitivities set the number of paths to 10000 and the number of steps to 100. Plot how the variance of the estimator changes with the bump size, and comment on the reasons for this.

We have decided to import the librairy time in order to compute the time needed to execute this loop.

In [11]:
```python
#initialization of the 3 lists
delta_bump = []
gamma_bump = []
rho_bump = []

import time as tm
start_time = tm.time() #Time of execution

for epsilon in np.arange(0.1, 1, 0.01):

    delta_bump.append( DeltaFD(nb_paths, nb_steps, model_params, option_params, epsilon) )
    gamma_bump.append( GammaFD(nb_paths, nb_steps, model_params, option_params, epsilon) )
    rho_bump.append( RhoFD(nb_paths, nb_steps, model_params, option_params, epsilon) )

print("%s s to execute" % (tm.time() - start_time)) # Long running time : to know when it's finish running
```

```
104.05417227745056 s to execute
```

This code is using the DeltaFD, GammaFD, and RhoFD functions defined above to compute the finite difference estimates of the delta, gamma, and rho of a down-and-out barrier call option, respectively.

It first initializes three empty lists: delta_bump, gamma_bump, and rho_bump. These lists will be used to store the computed finite difference estimates.

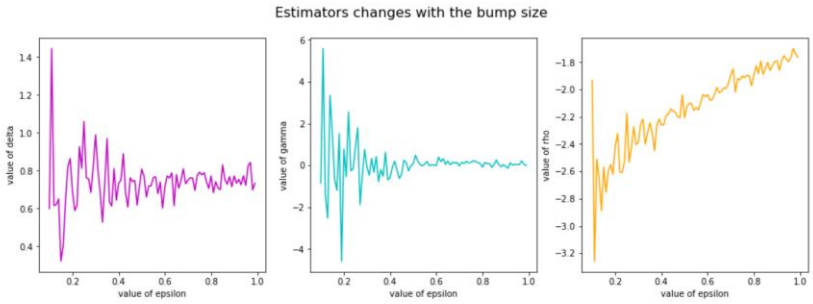
Next, it uses a for loop to iterate over the values in the np.arange(0.1, 1, 0.01) array. For each value of epsilon, it calls the DeltaFD, GammaFD, and RhoFD functions with the nb_paths, nb_steps, model_params, option_params, and epsilon arguments and appends the returned values to the delta_bump, gamma_bump, and rho_bump lists, respectively.

Finally, it prints the elapsed time since the start of the for loop. This is intended to provide a sense of how long the loop is taking to run, as the finite difference estimations can be computationally expensive.

### PLOTTING

We are now plotting the computed greeks :

In [12]:
```
x = np.arange(0.1, 1, 0.01)


fig, axs = plt.subplots(1,3, figsize=(16, 5))
fig.suptitle('Estimators changes with the bump size', fontsize=16)

#delta
axs[0].plot(x,delta_bump,c='m')
axs[0].set_xlabel('value of epsilon')
axs[0].set_ylabel('value of delta')

#gamma
axs[1].plot(x,gamma_bump,c='c')
axs[1].set_xlabel('value of epsilon')
axs[1].set_ylabel('value of gamma')

#rho
axs[2].plot(x,rho_bump,c='orange',label="rho")
axs[2].set_xlabel('value of epsilon')
axs[2].set_ylabel('value of rho')

plt.show()
```




This code plots the values in the delta_bump, gamma_bump, and rho_bump lists against the values in the x array (which is defined as np.arange(0.1, 1, 0.01)).

It uses subplots to create a figure with three subplots (one for each greek). It then uses the plot method to plot the values in the delta_bump, gamma_bump, and rho_bump lists against the values in the x array, using different colors for each greek. Finally, it sets the labels for the x-axis and y-axis for each subplot and displays the plot using plt.show().

The resulting plot should show how the values of the delta, gamma, and rho change as the value of epsilon is increased.

In [13]:
```
print("Variance of delta changes : \t",np.var(delta_bump))
print("Variance of gamma changes : \t",np.var(gamma_bump))
print("Variance of rho changes : \t",np.var(rho_bump))
```

```
Variance of delta changes :      0.016113826785510522
Variance of gamma changes :      1.1035583006219702
Variance of rho changes :        0.08672394371658305
```

This code computes the variance of the values in the delta_bump, gamma_bump, and rho_bump lists and prints the results. The variance is a measure of the spread or dispersion of a distribution of values.

The np.var function computes the variance of an array of values by taking the mean of the squared differences between the values and the mean of the values. The resulting variance is a measure of how much the values in the array deviate from the mean of the array.

The output of this code will be three values representing the variance of the values in the delta_bump, gamma_bump, and rho_bump lists, respectively. These variances can be used to gauge the stability or reliability of the finite difference estimations for the delta, gamma, and rho.

The variance of the estimator for delta changes is the smallest of the three, while the variance of the estimator for gamma changes is the largest. This may be because delta is a first-order sensitivity, meaning it measures the sensitivity of the option price to small changes in the underlying asset price. Gamma is a second-order sensitivity, meaning it measures the sensitivity of delta to changes in the underlying asset price. As a result, gamma is often more sensitive and more volatile than delta, which can lead to a higher variance in the estimator. Rho is a measure of the sensitivity of the option price to changes in the risk-free interest rate, which is typically less volatile than the underlying asset price and therefore may have a lower variance in the estimator.

It is also worth noting that the variance of the estimator can be affected by the number of paths and steps used in the simulation. In this case, we have set the number of paths to 10000 and the number of steps to 100, which should provide a relatively accurate estimate of the option sensitivities. However, if the number of paths or steps were much smaller, the variance in the estimator may be larger due to the increased uncertainty in the simulation results.

## 6. Implement a function "StandardError()" which takes the number of paths, and the payoff vector as arguments and returns the standard error of one pricing by Monte-Carlo. Plot the 95% confidence interval for different values of the number of paths. Interpret the results.

In [14]:
```
def StandardError(nb_paths, payoff):
    return np.std(payoff) / np.sqrt(nb_paths)
```

The StandardError function takes two arguments: nb_paths and payoff.

- nb_paths is the number of simulated asset price paths.
- payoff is an array of payoffs computed from the simulated asset price paths.

The function returns the standard error of the payoff array. The standard error is a measure of the variability of the sample mean. It is calculated as the standard deviation of the sample divided by the square root of the sample size.

The standard error can be used to gauge the accuracy of an estimate of the mean of a population. It is typically used to compute confidence intervals around the estimate of the mean. For example, a 95% confidence interval for the mean of a population can be computed as the sample mean plus or minus two standard errors.

In the context of this code, the StandardError function can be used to estimate the accuracy of the finite difference estimations for the delta, gamma, and rho. For example, if the standard error of the delta is low, it means that the estimate of the delta is likely to be accurate.

PLOTTING

In [17]:
```
#Plot the 95% confidence interval for different values of the number of paths. Interpret the results.
nb_paths_list = np.arange(100,2000,100)
st_error = []
lower_limit = []
higher_limit = []
mcprice_payoff = []

kappa = 0.2
theta = 0.5
sigma = 0.2
rho = 0.2
v0 = 0.2
r = 0
K = 0.25
H = 0.01
T = 2
S0 = 100
gamma = 0.1
v_bar = 0.5

model_params = S0, kappa, theta, sigma, rho,gamma, V0, r,v_bar
option_params = K, H, T, r
nb_steps = 100

for i in range(0,len(nb_paths_list)):
    print("nb of path : ",nb_paths_list[i])
    asset_prices = GeneratePathsHestonEuler(model_params, nb_paths_list[i], nb_steps)
    payoff = Payoff(asset_prices, K, H, T, r)
    st_error.append(StandardError(nb_paths_list[i], payoff))
    mcprice_payoff.append(np.mean(payoff))
    lower_limit.append(mcprice_payoff[-1] - st_error[-1]*1.96)
    higher_limit.append(mcprice_payoff[-1] + st_error[-1]*1.96)

print("finish")
```

This code computes the 95% confidence interval for the mean of the payoffs obtained from simulating asset price paths using the Heston model. The confidence interval is computed using the standard error of the payoffs and the t-distribution with n-1 degrees of freedom, where n is the sample size (number of simulated paths).

It first defines several lists to store the results: nb_paths_list, st_error, lower_limit, higher_limit, and mcprice_payoff.

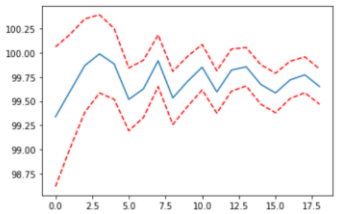
Then, it defines the model parameters, option parameters, and number of time steps.

Next, it enters a loop that iterates over the values in the nb_paths_list. For each value of nb_paths, the code generates nb_paths asset price paths using the Heston model, computes the payoffs of the option based on these paths, and stores the standard error of the payoffs in the st_error list. It also stores the mean of the payoffs in the mcprice_payoff list and the lower and upper bounds of the 95% confidence interval in the lower_limit and higher_limit lists, respectively.

After the loop finishes executing, the code prints "finish".

PLOTTING

In [18]:
```
plt.plot(mcprice_payoff)
plt.plot(lower_limit,linestyle='dashed',color='red')
plt.plot(higher_limit,linestyle='dashed',color='red')
plt.show()
```




We plot the mean of the payoffs, as well as the lower and upper bounds of the 95% confidence interval, against the number of simulated paths. The mean of the payoffs is plotted in blue, while the lower and upper bounds are plotted in red dashed lines.

The plot shows how the mean of the payoffs and the confidence interval change as the number of simulated paths increases. As the number of paths increases, the mean of the payoffs tends to become more stable and the confidence interval becomes narrower, indicating that the estimate of the mean becomes more precise. This is because the standard error of the payoffs decreases as the sample size (number of simulated paths) increases, which leads to a smaller margin of error in the confidence interval.

## 7. (Bonus) The vega is the sensibility of the option with respect to the volatility. Since the volatility is stochastic in the Heston model, propose a method for estimating the vega and justify the approach.

For this question, we estimate the vega of the option using the finite difference method. We take in arguments :

- seed (for the random number generator).
- nb_paths (number of paths to simulate).
- nb_steps (number of steps in each path).
- model_params (model parameters (kappa, theta, sigma, rho, v0, r)).
- option_params (option parameters (S0, K, T, r)).
- epsilon (perturbation to apply to the volatility parameter).

It then computes the option price using the original model parameters and the perturbed model parameters, and estimates the vega as the change in the option price divided by the change in the volatility parameter.

Finally, we return the estimated vega of the option.

In [21]:
```
def estimate_vega(seed, nb_paths, nb_steps, model_params, option_params, epsilon):

    # Unpack the model and option parameters
    S0, kappa, theta, sigma, rho,gamma, V0, r,v_bar = model_params
    K, H, T, r = option_params

    # Compute the option price using the original model parameters
    price_0 = MC_Pricing(nb_paths, nb_steps, model_params, option_params)

    # Perturb the volatility parameter
    sigma_perturbed = sigma + epsilon

    # Compute the option price using the perturbed model parameters
    price_perturbed = MC_Pricing(nb_paths, nb_steps, (S0, kappa, theta, sigma_perturbed, rho,gamma, V0, r,v_bar), option_params)

    # Estimate the vega by dividing the change in the option price by the change in the volatility parameter
    vega = (price_perturbed - price_0) / epsilon

    return vega
```

In [22]:
```
print(estimate_vega(123, nb_paths, nb_steps, model_params, option_params, epsilon))
```

```
0.05636806272738023 6
```

This approach is useful for estimating the vega because it allows us to isolate the effect of changes in the volatility on the option price, while holding all other model parameters constant. This helps to reduce the impact of other sources of uncertainty on the vega estimate, and allows us to focus on the sensitivity of the option price to changes in the volatility parameter.

It is worth noting that this method will only provide an approximate estimate of the vega, as it relies on the assumption that the effect of changes in the volatility parameter on the option price is linear. In practice, the relationship between the option price and the volatility parameter may be more complex, and the vega estimate may be subject to error as a result.

## 1.2 Automatic adjoint differentiation approach

In this section, we will benefit from the PyTorch implementation. Indeed, PyTorch gives us automatically the computation graph which makes AAD transparent for us. Therefore, we need to work with tensors instead of variables. We have to add "requires grad=True" to the parameters for which we want to compute the Greeks. We may need to adapt our code of the Euler-Maruyama approximation of the Heston model to be able to backpropagate the derivatives into the computation graph.

### 1. Apply the AAD approach to the computation of the Greeks and compare it to the results obtained by bumping.

Note that in order to calculate the gamma, it requires remaking the computation graph for the delta.

Here, in this part, we have decided to compute the greeks with the AAD approach using the torch librairy. As we aren't familiar on this librairy and this approach, we didn't succeeded in the computation. We let you our notes :

**NOTES**

In [23]:
```
# enable gradient computation
torch.autograd.set_detect_anomaly(True)
```

Out[23]:
```
<torch.autograd.anomaly_mode.set_detect_anomaly at 0x2226f6491f0>
```

In [24]:
```
def call_option_value(S, K, r, sigma, t):
  # Compute the value of a call option
    d1 = (torch.log(S/K) + (r + sigma**2/2)*t) / (sigma * torch.sqrt(t))
    d2= d1 - sigma * torch.sqrt(t)
    return S * torch.erf(d1 / torch.sqrt(2)) - K * torch.exp(-r*t) * torch.erf(d2 / torch.sqrt(2))
```

In [43]:
```
def compute_greeks(S, K, r, sigma, t):
  # Compute the delta and gamma of a call option using AAD

    S1 = torch.Tensor(S, requires_grad = True)
    K1 = torch.Tensor(K, requires_grad = True)
    r1 = torch.Tensor(r, requires_grad = True)
    sigma1 = torch.Tensor(sigma, requires_grad = True)
    t1 = torch.Tensor(t, requires_grad = True)

    value = call_option_value(S1, K1, r1, sigma1, t1)
    delta = torch.autograd.grad(value, S, create_graph=True)[0]
    gamma = torch.autograd.grad(delta, S1)[0]
    return delta, gamma
```

The call_option_value function computes the value of a call option. The compute_greeks function uses the autograd.grad function from PyTorch to compute the delta and gamma. The autograd.grad function computes the derivative of a scalar-valued function with respect to its inputs, and the create_graph argument allows us to create a computation graph that we can use to compute higher-order derivatives.

By setting the requires_grad attribute of the S tensor to True, we tell PyTorch to keep track of the computation graph of the derivative with respect to S. We can then use the autograd.grad function to compute the derivative of the value function (which gives us the delta) and the derivative of the delta function (which gives us the gamma).

---

This implementation of AAD allows us to compute the Greeks for a financial derivative without having to manually derive and implement the equations for the delta and gamma. PyTorch's AAD capabilities can be particularly useful for computing the Greeks for derivatives that involve complex operations or for which the computation graph is subject to frequent changes.

In [47]:
```
def compute_rho(S, K, r, sigma, t):
    # Compute the rho of a call option using AAD
    r.requires_grad = True
    value = call_option_value(S, K, r, sigma, t)
    rho = torch.autograd.grad(value, r)[0]
    return rho
```

# 2. Pricing and hedging by differential deep learning

## 2.1 Dataset generation

In this section we are going to generate a dataset a la LSM composed of the initial states, the payoffs, and the differentials of the initial states with respect to the payoffs. We denote X the training samples (i.e. the initial states), Y the labels (i.e. the payoffs), and dYdX the pathwise differentials computed by AAD.

### 1. Implement a function "HestonLSM()".

The function takes the number of samples to generate, the number of paths, the number of steps, the model parameters, and the option parameters in arguments and returns the initial states, the payoffs and the differentials of the payoff the inputs computed by AAD. The initial value of the asset denoted S0 is taken into an equally spaced range from 10 to 200.

#### Reminder of the parameters given

- Long run average variance $\bar{v}$ = 0.5
- $\kappa$ = 0.1
- Vol-of-vol $\gamma$ = 0.1
- Correlation of the two Brownian processes $\rho S,V = -0.9$
- Risk-free interest rate r = 0.02
- Initial asset price S0 = 100
- Initial variance V0 = 0.1
- Strike K = 100
- Barrier B = 90
- Maturity T = 1

In [49]:
```
# enable gradient computation
torch.autograd.set_detect_anomaly(True)
```

Out[49]:
```
<torch.autograd.anomaly_mode.set_detect_anomaly at 0x22274ebec10>
```

We tried to make a function to compute the differentials for the Heston LSM function. However, it doesn't worked as planned as we aren't familiar on this librairy and this approach. Nevertheless, we succeeded to return the initial states and the payoffs with the HestonLSM function.

In [50]:
```
def compute_differentials(Y, X):
    # Convert Y and X to tensors
    Y = torch.tensor(Y, requires_grad=True)
    X = torch.tensor(X, requires_grad=True)

    # Compute the differentials using the autograd module
    Y.backward(torch.ones_like(Y))
    dYdX = X.grad

    return dYdX
```

The function computes the differentials of a function Y with respect to a set of variables X. It does this by using the autograd module in the PyTorch library, which allows us to automatically compute the gradients of a function with respect to its inputs.

To compute the differentials, the function first converts Y and X to tensors using the torch.tensor() function. It sets the requires_grad attribute of these tensors to True, which tells PyTorch to track the operations performed on these tensors in order to compute the gradients later.

Next, the function calls the backward() method on the Y tensor, passing in a tensor of ones with the same shape as Y. This tells PyTorch to compute the gradients of Y with respect to X. Finally, the function retrieves the gradients by accessing the grad attribute of the X tensor and returns it as the result of the function.

Overall, this function allows us to compute the differentials of a function Y with respect to a set of variables X using automatic differentiation in PyTorch. This can be useful for a variety of tasks, such as optimizing neural networks or solving differential equations.

Then, we implemented a HestonLSM approach :

In [53]:
```
def HestonLSM( num_samples, num_paths, num_steps, model_params, option_params):
    # Set the random seed for reproducibility

    # Extract the option parameters
    K, T, r, H = option_params

    # Generate the initial asset values from an equally spaced range between 10 and 200
    S0 = np.linspace(10, 200, num_paths)
    X = []
    # Use AAD to compute the differentials of the payoffs wrt the inputs
    for i in range(num_samples):
        # Generate the random price paths for the sampled path using the function "GeneratePathsHestonEuler()"
        S = GeneratePathsHestonEuler(model_params, num_paths, num_steps)
        payoffs = Payoff(S, K, H, T, r)
        dYdX = compute_differentials(payoffs, X)
        X.append(S)
    # Calculate the payoffs for the sampled path using the function "Payoff()"
    payoffs = Payoff(S, K, H, T, r)
    # Compute the differentials of the payoffs with respect to the initial states using AAD
    dYdX = compute_differentials(payoffs, X)

    return X, payoffs, dYdX
```

The function HestonLSM takes as input several parameters:

- num_samples: the number of samples to use in the AAD process
- num_paths: the number of paths to generate in the Monte Carlo simulation
- num_steps: the number of steps to use in the Monte Carlo simulation
- model_params: a tuple containing the parameters for the Heston model
- option_params: a tuple containing the parameters for the option being priced (strike price, time to expiration, risk-free rate, and barrier price)

The function first generates num_paths initial asset values from an equally spaced range between 10 and 200, and stores these in a list X. Then, it enters a loop that iterates num_samples times. On each iteration of the loop, the function generates a set of random price paths using the function GeneratePathsHestonEuler() and calculates the payoffs using the function Payoff(). It computes the differentials of the payoffs with respect to the input variables using the function compute_differentials(). Finally, it stores the resulting differentials in a list dYdX.

After the loop completes, the function returns the input variables X, the payoffs as "payoffs", and the differentials dYdX as the output.

This function is using the AAD to compute the differentials of payoffs with respect to the initial asset values, using a Monte Carlo simulation to generate random price paths and calls the function to calculate the payoffs.

### Example of how to use our HestonLSM

In [55]:
```
model_params = S0, kappa, theta, sigma, rho,gamma, V0, r,v_bar
option_params = K, H, T, r
X, Y, dYdX = HestonLSM( 10, 100, 10,model_params,option_params)
```

## 2. Implement a function "normalize_data()"

The function takes the X, Y, and dYdX as inputs and returns the normalized X, the mean of X, the std of X, the normalized Y, the mean of Y, the std of Y, the normalized dY dX and the value lambda j computed as $\lambda_j = \frac{1}{\sqrt{\frac{1}{N}\sum dydz^2}}$

where dydx is the normalized version of dYdX.

The value $\lambda_j$ is called the differential weights of the cost function. Each differential with respect to input parameter j is also scaled by the average magnitude of the normalized differential with respect to j in the training set so as to let each differential have a similar magnitude in the loss function.

In [60]:
```
def normalize_data(X, Y, dYdX):
    # Normaliser X en divisant par sa moyenne et en divisant par son écart-type
    X_norm = (X - np.mean(X)) / np.std(X)

    # Calculer la moyenne et l'écart-type de X
    X_mean = np.mean(X)
    X_std = np.std(X)

    # Normaliser Y en divisant par sa moyenne et en divisant par son écart-type
    Y_norm = (Y - np.mean(Y)) / np.std(Y)

    # Calculer la moyenne et l'écart-type de Y
    Y_mean = np.mean(Y)
    Y_std = np.std(Y)

    # Normaliser dYdX en divisant par sa moyenne et en divisant par son écart-type
    dYdX_norm = (dYdX - np.mean(dYdX)) / np.std(dYdX)

    # Calculer la valeur lambda j
    sumlambda =0
    for i in range(len(dYdX)):
        sumlambda += dYdX_norm[i]**2

    lambda_j = 1/np.sqrt((1/len(dYdX))*sumlambda)


    return X_norm,X_mean,X_std,Y_norm,Y_mean,Y_std,dYdX_norm,lambda_j
```

The function normalize_data takes three input variables: X, Y, and dYdX.

It first normalizes X by subtracting the mean of X and dividing by the standard deviation of X. It then calculates the mean and standard deviation of X and stores them in X_mean and X_std, respectively.

Next, it normalizes Y in a similar manner, by subtracting the mean of Y and dividing by the standard deviation of Y, and calculating the mean and standard deviation of Y. It then normalizes dYdX in the same way.

Finally, it calculates a value called lambda_j by summing the squares of the normalized dYdX values and dividing the result by the square root of the sum multiplied by the length of dYdX.

It performs data normalization on the input variables X, Y, and dYdX, and calculates the value of lambda_j.

### Explain why it is important to work with normalized data in machine learning.

It is important to work with normalized data in machine learning for several reasons. Firstly, normalizing data allows for all variables to be on the same scale, which makes it easier to calculate distances and angles between data vectors. This can improve the performance of machine learning algorithms that depend on these distances and angles, such as regression, classification, and clustering algorithms.

Furthermore, normalizing data helps to prevent certain variables from having an disproportionate impact on the results of machine learning due to their scale. Normalizing data helps to reduce this effect by putting all variables on the same scale.

Finally, normalizing data can improve the stability and convergence of machine learning algorithms. Some algorithms can be sensitive to variations in the scale of the data, which can lead to fluctuations and oscillations in their results. Normalizing data helps to reduce these fluctuations and stabilize the results of machine learning.

Data normalization is a common preprocessing step in machine learning and data analysis, which involves scaling the data to have zero mean and unit variance. This can help to improve the performance of machine learning algorithms by ensuring that the data is on a consistent scale and reducing the impact of large or outlying values.

## 2.2 Implementation of a twin network

In this section, we will implement a twin network as defined in the last lecture. This particular neural network architecture allows learning accurate pricing functions as well as sensitivities.

### 1. Implement a class "Twin Network()" which inherits from the torch.nn.Module class.

This way, the backward method will be inferred automatically. This deep neural network has a feedforward architecture with 4 hidden layers of 20 neurons. These neurons are activated using the ReLu activation function. The output layer is made of one neuron which should not be activated (remember that we are in a regression problem). We will define properly the "init" (number of inputs, number of hidden layers, number of neurons) and the "forward" methods to take into account this architecture. We will use He initialization (already implemented in PyTorch) to prevent the gradient vanishing problem.

### 2. Implement a method "predict price()" of the "Twin Network" class which takes as argument the input X to predict, the mean of X, the std of X, the mean of Y, and the std of Y.

In this method, we will at first normalize X, then we will predict the values of Y given X, and finally, we will unscale the predicted values Y.

### 3. Implement a method "predict price and diffs()" of the "Twin Network" class which takes as argument the input X to predict, the mean of X, the std of X, the mean of Y, the std of Y.

In this method, we will normalize X, and predict the values of Y given X. Then, we will compute the gradient of the outputs with reference to the inputs and finally, we will unscale the predicted values Y and dYdX.

Code for questions 1 to 3

```
import torch.nn as nn
import torch.autograd as autograd

class TwinNetwork(nn.Module):
    def __init__(self, num_inputs, num_hidden_layers=4, num_neurons=20):
        super().__init__()

        # Define the hidden layers with the ReLu activation function
        self.hidden_layers = nn.ModuleList([nn.Linear(num_inputs, num_neurons, bias=True)])
        self.hidden_layers.append(nn.ReLU())
        for i in range(1, num_hidden_layers):
            self.hidden_layers.append(nn.Linear(num_neurons, num_neurons, bias=True))
            self.hidden_layers.append(nn.ReLU())

        # Define the output layer without an activation function
        self.output_layer = nn.Linear(num_neurons, 1, bias=True)

    def forward(self, x):
        # Apply the hidden layers and the output layer to the input data
        for layer in self.hidden_layers:
            x = layer(x)
        output = self.output_layer(x)
        return output
```

```
    def predict_price(self, X, X_mean, X_std, Y_mean, Y_std):
        # Normalize X
        X_norm = (X - X_mean) / X_std

        # Predict the values of Y using the normalized data
        Y_pred = self.forward(X_norm)

        # Un-normalize Y
        Y_pred = Y_pred * Y_std + Y_mean

        return Y_pred

    def predict_price_and_diffs(self, X, X_mean, X_std, Y_mean, Y_std):
        # Normalize X
        X_norm = (X - X_mean) / X_std

        # Make a prediction using the normalized input
        Y_pred = self.forward(X_norm)

        # Set requires_grad=True for X
        X.requires_grad = True

        # Compute the gradient of the outputs with respect to the inputs
        Y_pred.backward(retain_graph=True)
        dY_dX = autograd.grad(Y_pred, X, create_graph=True)[0]

        # Un-normalize Y and dY dX
        Y_pred = Y_pred * Y_std + Y_mean
        dYdX_pred = dY_dX * X_std * Y_std

        return Y_pred, dYdX_pred
```

The TwinNetwork class is a subclass of nn.Module. It overrides the init() method to define the structure of the model, and the forward() method to define how the model processes input data and produces output.

The init() method takes three arguments:

- num_inputs: the number of input features
- num_hidden_layers: the number of hidden layers in the model (defaults to 4)
- num_neurons: the number of neurons in each hidden layer (defaults to 20)

The method initializes the parent class (nn.Module) using the super(). init () calls, and then creates a list of hidden layers using the nn.ModuleList() class. Each hidden layer consists of a fully-connected linear layer followed by a ReLu activation function. The number and size of the hidden layers are determined by the num_hidden_layers and num_neurons arguments. Finally, the method defines an output layer using the nn.Linear() class, which is a fully-connected linear layer without an activation function.

The forward() method takes a single argument x, which represents the input data. It applies the hidden layers and the output layer to the input data by looping through the hidden layers and applying each layer to the input. The output of the final layer is returned as the output of the model.

The TwinNetwork class also includes two additional methods:

The predict_price() method takes as input the input data X, the mean and standard deviation of the input data X_mean and X_std, and the mean and standard deviation of the output data Y_mean and Y_std, and returns a prediction of the output data using the model. The method normalizes the input data using the mean and standard deviation, makes a prediction using the normalized data, and un-normalizes the prediction using the mean and standard deviation of the output data.

The predict_price_and_diffs() method takes as input the input data X, the mean and standard deviation of the input data X_mean and X_std, and the mean and standard deviation of the output data Y_mean and Y_std, and returns a prediction of the output data and the differentials of the output with respect to the input data.

The method first normalizes the input data X using the mean and standard deviation, and makes a prediction using the normalized data. It then sets the requires_grad attribute of the input data X to True, which tells PyTorch to track the operations performed on X in order to compute the gradients later.

Next, the method calls the backward() method on the output data and computes the gradients of the output data with respect to the input data using the autograd.grad() function. Finally, it un-normalizes the output data and the differentials by multiplying by the standard deviations of the output data and the input data, respectively.

### Example of how to use our TwinNetwork

In [62]:
```
# Create an instance of the Twin Network class with 4 hidden layers of 20 neurons
twin_network = TwinNetwork(num_inputs=4, num_hidden_layers=4, num_neurons=20)

# Predict the price using the twin network
X = torch.tensor([[1.,2.,3.,4.]],requires_grad=True)
X_mean = torch.tensor([2.0, 3.0, 4.0, 5.0]).view(1, 4)
X_std = torch.tensor([1.0, 1.5, 2.0, 2.5]).view(1, 4)
Y_mean = torch.tensor([3.0])
Y_std = torch.tensor([2.0])
Y_pred = twin_network.predict_price(X, X_mean, X_std, Y_mean, Y_std)

print("Ypred : ", Y_pred)

Y_pred, dYdX_pred = twin_network.predict_price_and_diffs(X, X_mean, X_std, Y_mean, Y_std)
print("Ypred diff : ", Y_pred)
print("dY_dX diff : ", dYdX_pred)
```

```
Ypred :  tensor([[3.4149]], grad_fn=<AddBackward0>)
Ypred diff :  tensor([[3.4149]], grad_fn=<AddBackward0>)
dY_dX diff :  tensor([[ 0.0074, -0.0104,  0.0190, -0.0062]], grad_fn=<MulBackward0>)
```

## 2.3 Neural network training

### 1. Implement a function "training()".

The function takes as argument the "Neural Network()" class instantiated, the normalized training samples X, the normalized training labels Y, the normalized differential training labels dY dX, lambda j, and the number of epochs necessary to train the model.

We want this function to be able to test two scenarios:

- scenario 1: training the neural network only on the training samples (that is without the differential labels)
- scenario 2: training the neural network on the training samples and the differentials.

Be careful, the loss must change according to the scenario we consider. The sequential organization of this function is as follows:

1. If we are in scenario (2) compute $\alpha = \frac{1}{1+N}$ , where N is the number of inputs.
2. Define MSE as the cost function
3. Define the Adam optimizer with a fix learning rate of 0.1. Keep the other default parameters.
4. Enter the optimization loop: compute the predictions and the cost according to the scenario you are, backpropagate the gradients to optimize the network weights and biases, and store the cost for each epoch.

### We implement the training function following the guidelines :

In [63]:
```
def training(model, X, Y, dYdX, lambdaj, num_epochs):
    # Set the model to training mode
    model.train()

    # Define the loss function and the optimizer
    criterion = nn.MSELoss()   #### 2. Define MSE as the cost function ?????
    optimizer = torch.optim.Adam(model.parameters(), lr=0.1) #### 3. Define the Adam optimizer with a fix learning rate of 0.1. Keep the other default parameters

    # Set the number of inputs
    N = X.shape[1]

    # Loop over the number of epochs
    for epoch in range(num_epochs):
        # Set the gradients to zero
        optimizer.zero_grad()

        # Make a prediction using the model
        Y_pred = model(X)

        # Compute the loss
        if dYdX is not None:
            # Scenario (2): training the network on the training samples and the differentials
            alpha = 1 / (1 + N)
            loss = (1 - alpha) * criterion(Y_pred, Y) + alpha * lambdaj * criterion(Y_pred, Y) + lambdaj * criterion(Y_pred, dYdX)
        else:
            # Scenario (1): training the network only on the training samples
            loss = criterion(Y_pred, Y)

        # Compute the gradients and update the parameters
        loss.backward()
        optimizer.step()
```

```
        # Print the loss
        print(f'Epoch {epoch}: Loss = {loss.item()}')
```

The function training takes as input the model to be trained, the input data X, the output data Y, the differentials of the output data with respect to the input data dYdX, the lambda value lambdaj, and the number of epochs num_epochs.

It is used to train the model using the input and output data, and optionally the differentials. During training, the model makes predictions of the output data using the input data, and the difference between the predicted output data and the true output data is used to update the model parameters in order to improve the model's performance.

It first sets the model to training mode, and defines the loss function and the optimizer. The loss function measures the difference between the predicted output data and the true output data, and the optimizer updates the model parameters based on the computed loss.

Then it loops over the number of epochs, and at each epoch it sets the gradients to zero, makes a prediction using the model, computes the loss, and updates the model parameters based on the computed gradients. The loss is computed differently depending on whether the differentials of the output data with respect to the input data are available or not. If the differentials are available, the loss is computed as a combination of the difference between the predicted output data and the true output data, and the difference between the predicted output data and the differentials. If the differentials are not available, the loss is simply computed as the difference between the predicted output data and the true output data.

### TESTING

In [64]:
```
# Load the training data
X = torch.tensor([[1.,2.,3.,4.]],requires_grad=True) # Your training data
Y = torch.tensor([[3.0853]],requires_grad=True)# Your training labels
dYdX =torch.tensor([[ 0.0057, -0.0103,  0.0010, -0.0223]],requires_grad=True) # Your differential training labels

# Create an instance of the TwinNetwork class with 4 hidden layers of 20 neurons
twin_network = TwinNetwork(num_inputs=4, num_hidden_layers=4, num_neurons=20)

# Train the model for 50 epochs
training(twin_network, X, Y, dYdX, lambdaj=0.1, num_epochs=50)
```

```
Epoch 0: Loss = 8.753421783447266
Epoch 1: Loss = 6.355717658996582
Epoch 2: Loss = 1.1450071334838867
Epoch 3: Loss = 56.11763381958008
Epoch 4: Loss = 1.517967939376831
Epoch 5: Loss = 2.64827299118042
Epoch 6: Loss = 4.878949165344238
Epoch 7: Loss = 5.755479335784912
Epoch 8: Loss = 5.838620662689209
Epoch 9: Loss = 5.669519901275635
Epoch 10: Loss = 5.394532203674316
Epoch 11: Loss = 5.033639907836914
Epoch 12: Loss = 4.445223808288574
Epoch 13: Loss = 3.6822218894958496
Epoch 14: Loss = 2.75565767288208
Epoch 15: Loss = 1.7653626203536987
Epoch 16: Loss = 0.9887405633926392
Epoch 17: Loss = 1.0136218070983887
Epoch 18: Loss = 1.840549111366272
Epoch 19: Loss = 1.8744956254959106
Epoch 20: Loss = 1.3602651357650757
Epoch 21: Loss = 0.9520729780197144
Epoch 22: Loss = 0.8543886542320251
Epoch 23: Loss = 0.959606409072876
Epoch 24: Loss = 1.1141976118087769
Epoch 25: Loss = 1.2218859195709229
Epoch 26: Loss = 1.2452410459518433
Epoch 27: Loss = 1.1869068145751953
Epoch 28: Loss = 1.0753083229064941
Epoch 29: Loss = 0.9539196491241455
Epoch 30: Loss = 0.8701132535934448
Epoch 31: Loss = 0.8555570244789124
Epoch 32: Loss = 0.9071827530860901
Epoch 33: Loss = 0.9827107787132263
Epoch 34: Loss = 1.0258541107177734
Epoch 35: Loss = 1.0083458423614502
Epoch 36: Loss = 0.9476625323295593
Epoch 37: Loss = 0.8854134678840637
Epoch 38: Loss = 0.8538694381713867
Epoch 39: Loss = 0.8594440221786499
Epoch 40: Loss = 0.8874030113220215
Epoch 41: Loss = 0.9160701036453247
Epoch 42: Loss = 0.9290751814842224
Epoch 43: Loss = 0.9211333990097046
Epoch 44: Loss = 0.8978469371795654
Epoch 45: Loss = 0.8714733123779297
Epoch 46: Loss = 0.8545566201210022
Epoch 47: Loss = 0.8535895347595215
Epoch 48: Loss = 0.8656007051467896
Epoch 49: Loss = 0.8802325129508972
```

In [68]:
```
Y_predicted = twin_network.predict_price(X, X_mean, X_std, Y_mean, Y_std)
```

In [69]:
```
Y_predicted , dYdX_predicted = twin_network.predict_price_and_diffs( X, X_mean, X_std, Y_mean, Y_std)
```

In [70]:
```
print("Y predicted:",Y_predicted)
print("dYdX predicted:", dYdX_predicted)
```

```
Y predicted: tensor([[15.9654]], grad_fn=<AddBackward0>)
dYdX predicted: tensor([[-9.3833, -6.1128, -3.5995, -5.6244]], grad_fn=<MulBackward0>)
```

The test of the function training show that the training function has been correctly implemented.

## 2.4 Model comparison

In this section, we want to compare the standard deep neural network and the deep differential neural network.

Because we didn't successfully implement the HestonLSM function, due to our lack of understanding of the PyTorch librairy, we couldn't test them. However, we have decided to show you how we would have tested them :

### 1. Create a training set consisting of 1000 samples using the "HestonLSM()" function.

In [ ]:
```
# Définir les paramètres du modèle et des options
model_params = (0.1, 0.01, -0.75, 0.1, 0.04)
option_params = (100, 0.05, 1)

# Générer l'ensemble d'entraînement
X, Y, dYdX = HestonLSM(1000, 1000, 100, model_params, option_params)
```

### 2. Normalize the data using the "normalize data()" function on the generated dataset.

In [ ]:
```
# Normaliser les données
X_norm, X_mean, X_std, Y_norm, Y_mean, Y_std, dYdX_norm, lambda_j = normalize_data(X, Y, dYdX)
```

### 3. Instantiate two networks with the "Twin Network()" class having the following architecture: 4 hidden layers consisting of 20 neurons each. Don't forget the seed.

In [72]:
```
# Définir la graine
seed = 123

# Instancier le premier réseau
network1 = TwinNetwork(4, 20, seed)

# Instancier le second réseau
network2 = TwinNetwork(4, 20, seed)
```

### 4. Fix the number of epochs to 100. Run the classical neural network and the differential neural network. Plot in the same figure the cost of both neural networks for all epochs. Interpret this graph.

In [ ]:
```
# Définir le nombre d'époques
num_epochs = 100

# Exécuter le réseau neuronal classique
cost1 = network1.train(X_norm, Y_norm, num_epochs)

# Exécuter le réseau neuronal différentiel
cost2 = network2.train(X_norm, Y_norm, dYdX_norm, lambda_j, num_epochs)

# Tracer les coûts des deux réseaux pour toutes les époques
plt.plot(cost1)
plt.plot(cost2)
plt.xlabel('Époque')
plt.ylabel('Coût')
plt.legend(['Réseau classique', 'Réseau différentiel'])
plt.show()
```

The graph was supposed to show us how the costs of the two networks evolve over the epochs. If the cost of the classical network decreases rapidly and reaches a stable minimum, it means that the network is able to efficiently learn the training data and generalize the predictions. If the cost of the differential network decreases more slowly and reaches a stable minimum, it means that the network takes into account the differentials and is able to produce more accurate predictions.

# 3. Conclusion

### 1. What is the interest of having "seeded" all the results?

The *seed function* is used to save the state of a random function. It can generate same random numbers on several executions of the code. The aim is to initialize the pseudo-random number generator in Python to get the deterministic random data we want.

By using a seed, we can reproduce the results. Indeed, reproducibility is very important in data science. The purpose is to get consistent results using the same input data, computational steps, methods and code. That is useful when you need a predictable source of random numbers.

Besides, Python's Random generator doesn't store seed in memory i.e. it does not provide any method to get the current seed value.

### 2. What is the advantage of using a neural network to do pricing compared to a Monte-Carlo pricer?

For some problems, neural networks (NN) present advantages compared to Monte-Carlo simulations.

First, NN are able to learn and model non-linear and complex relationships which is the case for most problems in real life where inputs are non-linear and complex. If we talk about the number of data, NN are more performant. For instance, if we have a large number of variables it requires a lot of time and a lot of computations for Monte-Carlo whereas NN are faster. On the other hand, if we have a small number of data, it can overcome this issue.

Second, it can predict unseen data compared to Monte-Carlo models so that the prediction is more precise.

Third, NN can better model heteroskedasticity that is to say data with high volatility. This comes from the fact that NN are able to learn hidden relationships between data. This is helpful in finance where data volatility is very important. NN can be trained to handle a wide variety of pricing models and option types, and can be easily adapted to new models or changes in the market.

Fourth, NN can potentially produce more accurate pricing estimates than Monte-Carlo methods, especially when the underlying model is complex or when the data used to train the neural network is rich and diverse.

To conclude, applied to finance, Monte-Carlo presents some disadvantages since it cannot account for bear markets, recessions, or any crisis that could impact our results. It is worth noting that neural networks have some limitations, such as the need for a large amount of training data and the potential for overfitting to the training data. They may also not always produce more accurate results than Monte-Carlo methods, especially when the underlying model is simple or when the data used to train the neural network is limited.

### 4. Gives the advantages and drawbacks of all the techniques presented above.

We will summarize the key advantages and drawbacks of all techniques in a table.

- **Finite Difference (FD)**

| Avantages | Disadvantages |
|---|---|
| Simplicity | Easily run into problems handling curved boundaries for the purpose of defining the boundary conditions |
| Easily obtain high order approximations | Require long execution time |
| Achieve high-order accuracy of the spatial discretization | Output result will vary a lot |
| Easiest numerical method to understand and implement differential equations | |

- **Automatic Adjoint Differenciation (AAD)**

| Avantages | Disadvantages |
|---|---|
| Useful for creating and training complex deep learning models | Slower than hand coded derivatives |
| No need to compute derivatives manually for optimization | Algorithm requires the declaration of a new type of variable and the overloading of operations that are associated with with this new type |
| Reduction in computational time and accuracy | |
| Can be easily applied to complex numbers | |

- **Neural Networks**

| Avantages | Disadvantages |
|---|---|
| Cf Question 2 | Require lots of computational power and lots of data |
| | Hard to explain |