

# Projet Calibration

Théo MIDAVAINE & Matthieu JULIEN

```
In [ ]: import math
import pandas as pd
import numpy as np
import random
import copy

from scipy.stats import norm

import matplotlib.pyplot as plt
```

## I – Densités risque neutre

1/ Sur les prix d'option données dans le tableau ci-après, calibrer une densité risque neutre en utilisant la formule de Breeden-Litzenberger et la technique de Shimko. Comparer avec une densité gaussienne.

```
In [ ]: data = pd.DataFrame()
data["strikes"] = [95, 96, 97, 98, 99, 100, 101, 102, 103, 104]
data["prices"] = [10.93, 9.55, 8.28, 7.40, 6.86, 6.58, 6.52, 6.49, 6.47, 6.46]

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

def newton_raphson_implied_vol(row, S, r, initial_guess=0.2, T=1):
    sigma = initial_guess
    K = row['strikes']
    price_market = row['prices']
    for _ in range(100):
        d1 = (np.log(S / K) + (r + sigma**2 / 2) * T) / (sigma * np.sqrt(T))
        d2 = d1 - sigma * np.sqrt(T)
        price_model = S * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
        vega = S * np.sqrt(T) * norm.pdf(d1)
        price_diff = price_market - price_model

        if np.abs(price_diff) < 1e-8:
            return sigma
        sigma += price_diff / vega # Mise à jour de sigma

        if sigma < 0:
            sigma = 0.001
    return sigma

def quadratic_interpolation(x_interpolated, x_original, y_original):
    y_interpolated = []
    for x in x_interpolated:
        for i in range(len(x_original) - 2):
```

```

        if x_original[i] <= x <= x_original[i + 2]:
            x1, x2, x3 = x_original[i], x_original[i + 1], x_original[i + 2]
            y1, y2, y3 = y_original[i], y_original[i + 1], y_original[i + 2]
            break

        A = np.array([[x1**2, x1, 1], [x2**2, x2, 1], [x3**2, x3, 1]])
        B = np.array([y1, y2, y3])
        a, b, c = np.linalg.solve(A, B)
        y = a*x**2 + b*x + c
        y_interpolated.append(y)

    return y_interpolated

```

Nelder-Mead function used as minimizing function that will be used throughout the project

```

In [ ]: def nelder_mead(cost_function, initial_guess,*args, tol=1e-8, max_iter=2000, alpha=0.1, gamma=1.0, beta=0.5):
    n = len(initial_guess)
    simplex = np.zeros((n + 1, n))
    simplex[0] = initial_guess

    for i in range(1, n + 1):
        simplex[i] = initial_guess + 0.1 * (i * np.eye(n)[i - 1])

    for _ in range(max_iter):
        values = np.array([cost_function(point,args) for point in simplex])

        order = np.argsort(values)
        simplex = simplex[order]
        values = values[order]
        centroid = np.mean(simplex[:-1], axis=0)

        reflection = centroid + alpha * (centroid - simplex[-1])
        reflection_value = cost_function(reflection,args)

        if values[0] <= reflection_value < values[-2]:
            simplex[-1] = reflection

        elif reflection_value < values[0]:
            expansion = centroid + gamma * (reflection - centroid)
            expansion_value = cost_function(expansion,args)

            if expansion_value < reflection_value:
                simplex[-1] = expansion
            else:
                simplex[-1] = reflection

        else:
            contraction = centroid + beta * (simplex[-1] - centroid)
            contraction_value = cost_function(contraction,args)

            if contraction_value < values[-1]:
                simplex[-1] = contraction
            else:
                simplex[1:] = simplex[0] + 0.5 * (simplex[1:] - simplex[0])

        if np.linalg.norm(simplex[0] - simplex[-1]) < tol:
            break

    result = {'x': simplex[0], 'fun': values[0]}
    return result

```

```
In [ ]: T = 1 # On considère des calls 1 an  
r = 0 # Risk free rate à 0  
S0 = 100. # Spot price à T0 = 100
```

```
In [ ]: data["implied_vol"] = data.apply(lambda row: newton_raphson_implied_vol(row, S0, r, data.T
```

Out[ ]:

	0	1	2	3	4	5	6	
<b>strikes</b>	95.000000	96.000000	97.000000	98.000000	99.000000	100.000000	101.000000	102.000000
<b>prices</b>	10.930000	9.550000	8.280000	7.400000	6.860000	6.580000	6.520000	6.490000
<b>implied_vol</b>	0.210948	0.189023	0.170031	0.160956	0.160079	0.165124	0.175031	0.181000

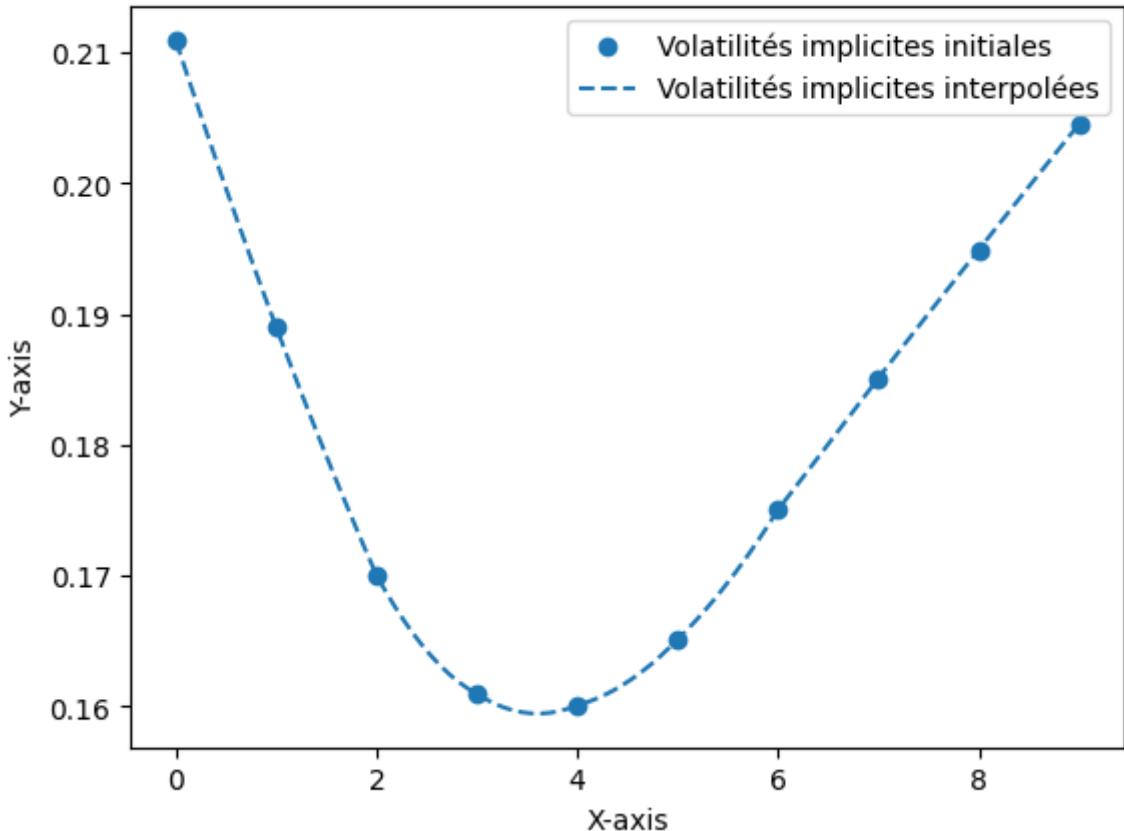
```
In [ ]: print('Prix théorique avec la volatilité implicite calculée par NewtonRaphson : ',  
print('Prix de marché :', round(data[data['strikes'] == 95]['prices'][0],2))
```

Prix théorique avec la volatilité implicite calculée par NewtonRaphson : 10.93  
Prix de marché : 10.93

Interpolation des volatilités implicites

```
In [ ]: x_original = np.array(data.index)  
y_original = np.array(data["implied_vol"])  
  
nb_pts = 100  
x_interpolated = np.linspace(x_original.min(), x_original.max(), nb_pts)  
vol_interpolated = quadratic_interpolation(x_interpolated, x_original, y_original)  
  
plt.scatter(x_original, y_original, label='Volatilités implicites initiales')  
plt.plot(x_interpolated, vol_interpolated, label='Volatilités implicites interpolées')  
plt.legend()  
plt.xlabel('X-axis')  
plt.ylabel('Y-axis')  
plt.title('Interpolation Quadratique de la Volatilité implicite')  
plt.show()
```

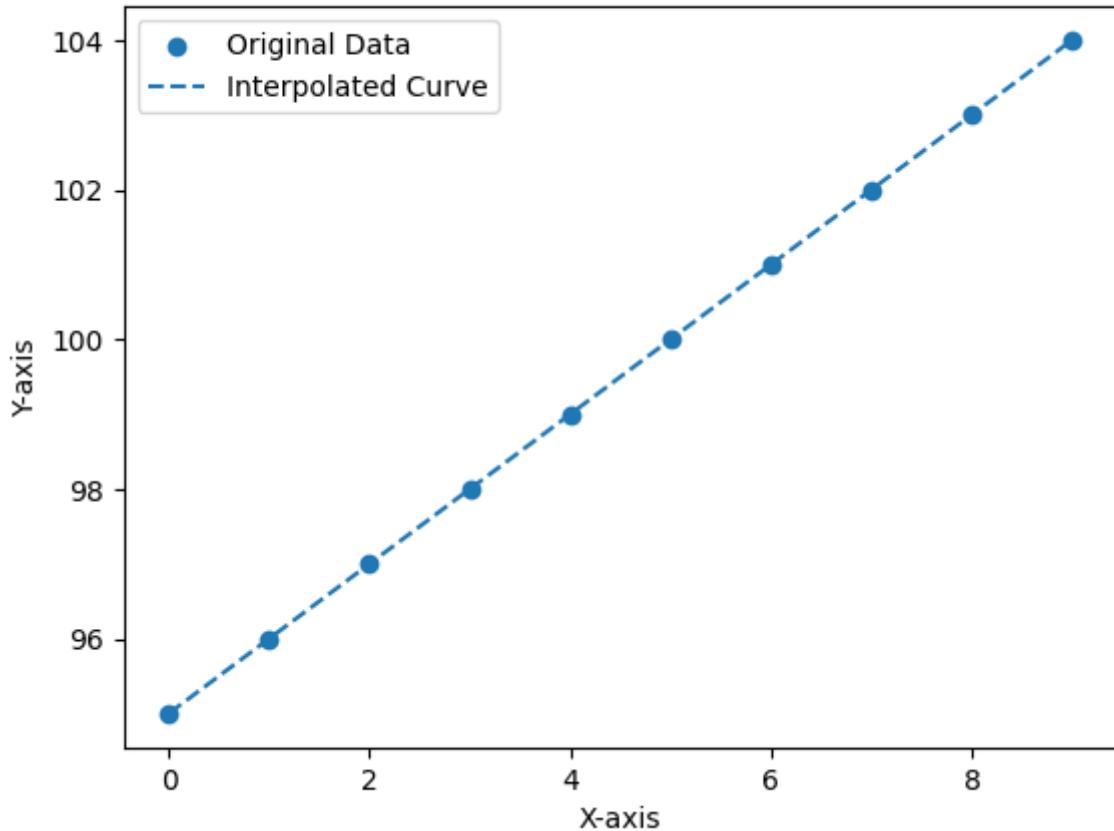
## Interpolation Quadratique de la Volatilité implicite



Interpolation des strikes

```
In [ ]: x_original=data.index  
y_original=data["strikes"]  
nb_pts=100  
x_interpolated=np.linspace(x_original.min(), x_original.max(), nb_pts)  
strike_interpolated = quadratic_interpolation(x_interpolated, x_original, y_original)  
  
plt.scatter(x_original, y_original, label='Original Data')  
plt.plot(x_interpolated, strike_interpolated, label='Interpolated Curve', linestyle='dashed')  
plt.legend()  
plt.xlabel('X-axis')  
plt.ylabel('Y-axis')  
plt.title('Interpolation quadratique du Strike')  
plt.show()
```

## Interpolation quadratique du Strike



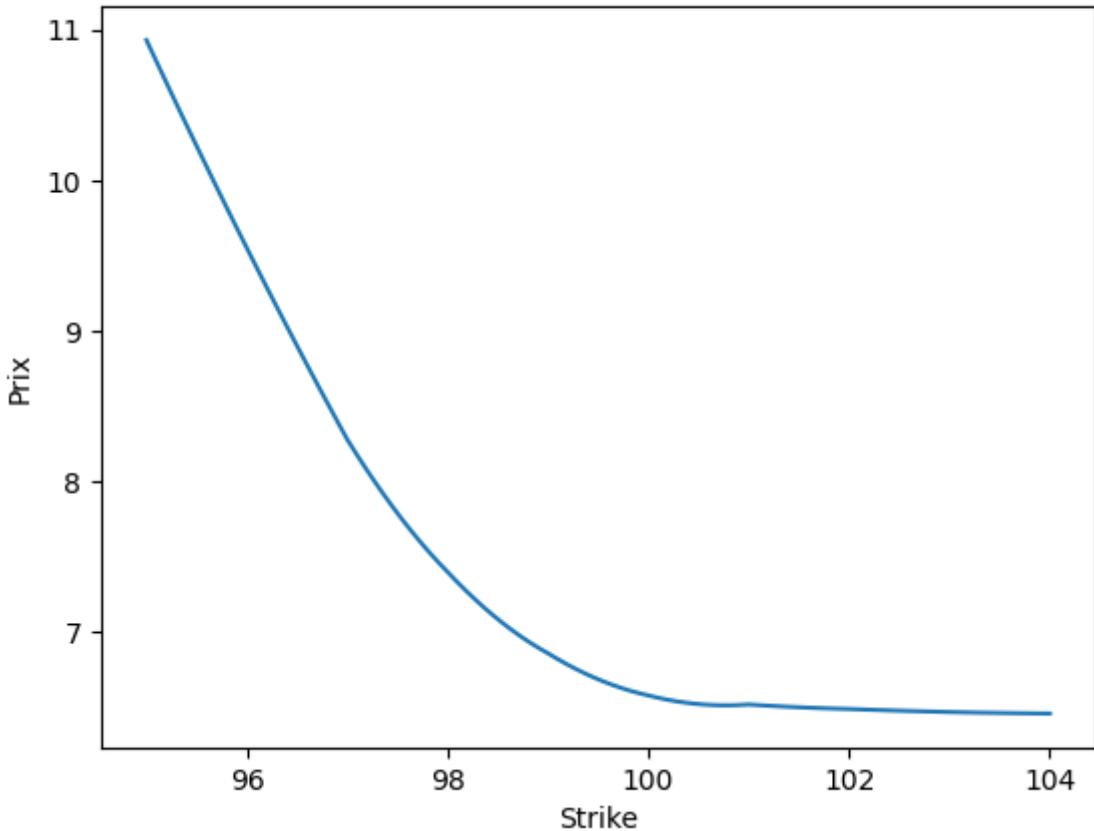
Calcul des prix avec Black-Scholes avec les vol et strike interpolés

```
In [ ]: prices=[]

for i,j in zip(strike_interpolated,vol_interpolated):
    prices.append(black_scholes_call_price(S0,i,T,r,j))

df=pd.DataFrame([prices,strike_interpolated],index=["Price","Strike"]).T
plt.plot(np.array(df["Strike"]),np.array(df["Price"]))
plt.title("Evolution du prix du call en fonction du strike ")
plt.ylabel("Prix")
plt.xlabel("Strike")
plt.show()
```

## Evolution du prix du call en fonction du strike



```
In [ ]: def neutral_risk_density(strikes , r , S0 , T , implied_vols ) :
    density = np.zeros(len(strikes))
    k=df["Strike"][-1]-df["Strike"][0]

    for i , K in enumerate (strikes) :
        C_plus_k = black_scholes_call_price (S0 , K + k , T , r , implied_vols[i] )
        C_minus_k = black_scholes_call_price (S0 , K - k , T , r , implied_vols[i])
        C_actuel = black_scholes_call_price (S0 , K , T , r , implied_vols[i] )

        # Formule de Breeden - Litzenberger
        density [ i ] = np.exp( r * T ) *(( C_plus_k - 2 * C_actuel + C_minus_k ) / (2 * k))

    return density

density_risque_neutre = neutral_risk_density (strike_interpolated , r , S0 , T , vols)
```

```
In [ ]: def gaussian(x, mu, sigma, ratio):
    return ratio * np.exp(-(x - mu)**2 / (2 * sigma**2)) / (sigma * np.sqrt(2 * np.pi))

def objective(params, args):
    mu, sigma, ratio = params
    return np.sum((gaussian(args[0], mu, sigma, ratio) - args[1])**2)

initial_guess = [98., 5., 2.]
result = nelder_mead(objective,np.array(initial_guess),df["Strike"],density_risque_neutre)

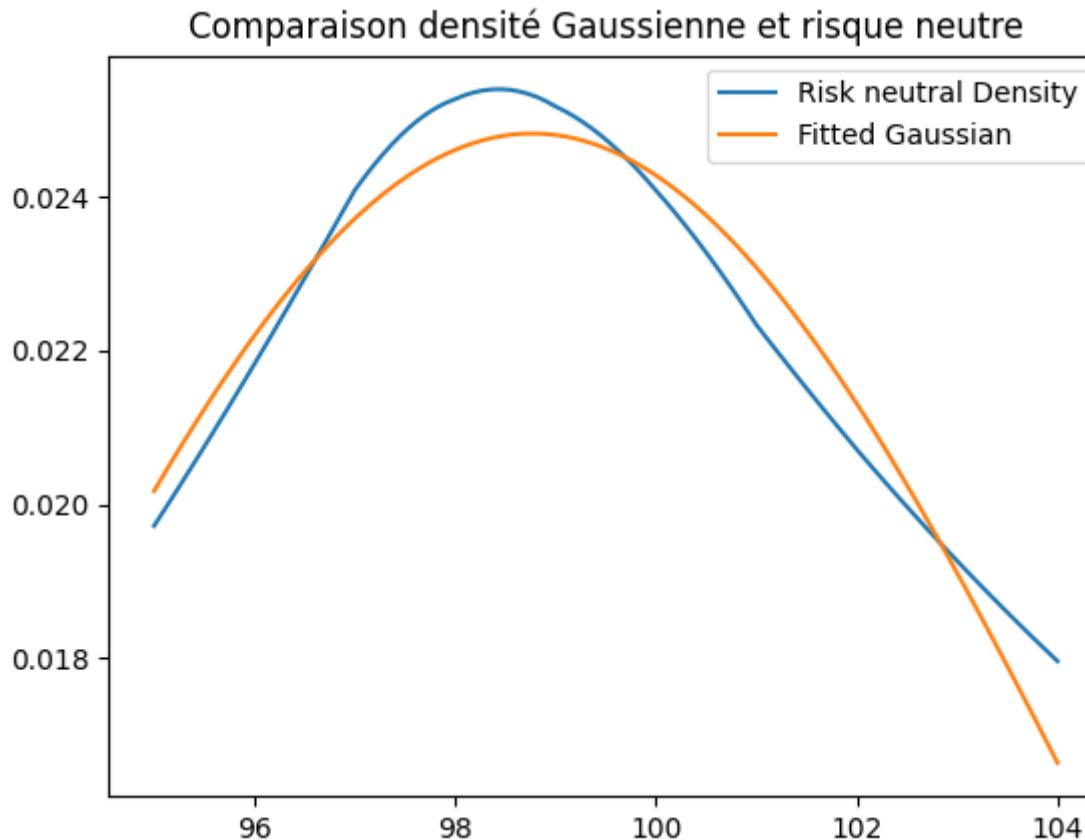
fitted_mu, fitted_sigma, fitted_ratio = result['x']
```

```
In [ ]: plt.plot(df["Strike"], density_risque_neutre, label='Risk neutral Density')
plt.plot(df["Strike"], gaussian(df["Strike"], fitted_mu, fitted_sigma, fitted_ratio),
         label='Gaussian Fit')
plt.title( "Comparaison densité Gaussienne et risque neutre")
plt.legend()
plt.show()
```

```

print("Fitted Mean:", fitted_mu)
print("Fitted Standard Deviation:", fitted_sigma)
print("Fitted Amplitude (Ratio):", fitted_ratio)

```



```

Fitted Mean: 98.7679361801859
Fitted Standard Deviation: 5.852217978227197
Fitted Amplitude (Ratio): 0.3640658898678022

```

**2/ Vérifier, en faisant des tirages dans cette loi implicite, si l'on trouve un prix de modèle proche du prix de marché pour toutes les options ci-dessus.**

```

In [ ]: # Simulation Monte Carlo pour valoriser une option
def valorisation_MC(S, K, T, r, pas, sigma,nb_simulations):
    dt=T/pas
    paths = np.zeros((nb_simulations, pas + 1)) #on crée une matrice de taille nb simulations par pas+1
    paths[:, 0] = S #on remplit la première valeur de chaque ligne avec le spot
    for i in range(pas):
        paths[:, i + 1] = paths[:, i] *np.exp(dt*(r - 0.5*sigma**2) + sigma*np.sqrt(dt))
    #pour chaque pas on calcule le prix future en utilisant le spot précédent
    option_payoffs = np.maximum(paths[:, -1] - K, 0) #on prend ensuite le max de chaque ligne
    option_value = np.exp(-r * T) * np.mean(option_payoffs) # on calcule la moyenne

    return option_value

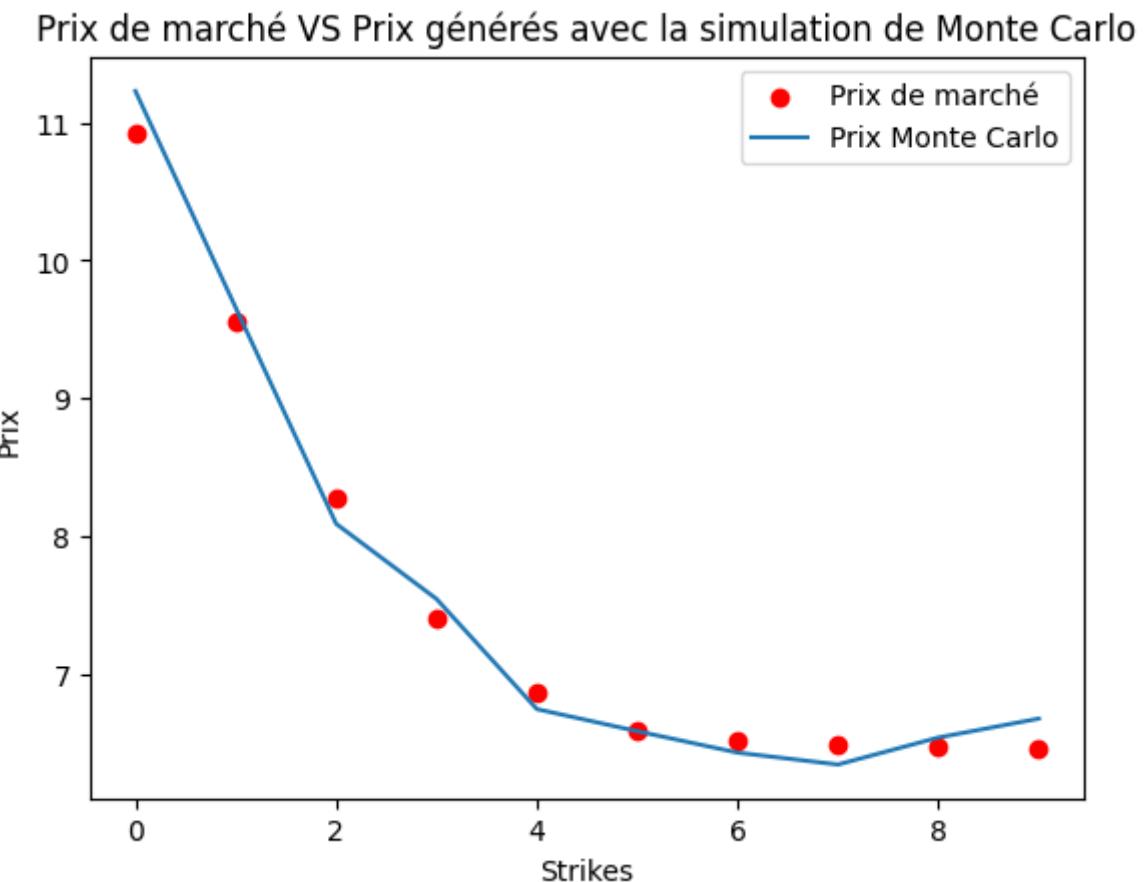
```

```

In [ ]: option_prices_mc = [valorisation_MC(S0, strike, T, r, 252, vol, 10000) for strike,
fig, ax = plt.subplots()
index = np.arange(len(data["strikes"]))
rects1 = ax.scatter(index, data["prices"], label='Prix de marché', color='red')
rects2 = ax.plot(index, option_prices_mc, label='Prix Monte Carlo')

```

```
ax.set_xlabel('Strikes')
ax.set_ylabel('Prix')
ax.set_title('Prix de marché VS Prix générés avec la simulation de Monte Carlo')
ax.legend()
plt.show()
```



```
In [ ]: for strike,vol,market_price,mc_price in zip(data["strikes"],data["implied_vol"],data["market_prices"]):
    print(f"Option : K = {strike}, Sigma = {vol:.2f}")
    print(f"Prix par simulation de Monte Carlo = {mc_price:.2f}")
    print(f"Market Price = {market_price:.2f}\nDifference = {abs(mc_price-market_price)}
```

```

Option : K = 95, Sigma = 0.21
Prix par simulation de Monte Carlo = 11.23
Market Price = 10.93
Difference = 0.30

Option : K = 96, Sigma = 0.19
Prix par simulation de Monte Carlo = 9.66
Market Price = 9.55
Difference = 0.11

Option : K = 97, Sigma = 0.17
Prix par simulation de Monte Carlo = 8.09
Market Price = 8.28
Difference = 0.19

Option : K = 98, Sigma = 0.16
Prix par simulation de Monte Carlo = 7.54
Market Price = 7.40
Difference = 0.14

Option : K = 99, Sigma = 0.16
Prix par simulation de Monte Carlo = 6.74
Market Price = 6.86
Difference = 0.12

Option : K = 100, Sigma = 0.17
Prix par simulation de Monte Carlo = 6.58
Market Price = 6.58
Difference = 0.00

Option : K = 101, Sigma = 0.18
Prix par simulation de Monte Carlo = 6.43
Market Price = 6.52
Difference = 0.09

Option : K = 102, Sigma = 0.19
Prix par simulation de Monte Carlo = 6.34
Market Price = 6.49
Difference = 0.15

Option : K = 103, Sigma = 0.19
Prix par simulation de Monte Carlo = 6.54
Market Price = 6.47
Difference = 0.07

Option : K = 104, Sigma = 0.20
Prix par simulation de Monte Carlo = 6.67
Market Price = 6.46
Difference = 0.21

```

## II – Interpolation et volatilité locale

En plus du tableau donné précédemment, on va utiliser pour cette partie les prix d'options suivants

On utilise la fonction d'interpolation quadratique, pour lisser la courbe

```
In [ ]: def interpolation(data,T):
    prices = []
    #interpolation of vol
    x_original = np.array(data.index)
    y_original = np.array(data["implied_vol"])
```

```

nb_pts = 100
x_interpolated = np.linspace(x_original.min(), x_original.max(), nb_pts)
y_interpolated = quadratic_interpolation(x_interpolated, x_original, y_original)

#interpolation of strikes
x_original=data.index
y_original=data["strikes"]
x_interpolated=np.linspace(x_original.min(), x_original.max(), nb_pts)
strike_interpolated = quadratic_interpolation(x_interpolated, x_original, y_ori

#Calcul des prix avec la formule de black scholes en fonction des paramètres interpolés
for i,j in zip(strike_interpolated,y_interpolated):
    prices.append(black_scholes_call_price(S0,i,T,r,j))

df=pd.DataFrame([prices,strike_interpolated,y_interpolated],index=["prices","strikes"])
return df

```

```

In [ ]: data_12_months = {'strikes': [95, 96, 97, 98, 99, 100, 101, 102, 103, 104],
                      'prices': [10.93, 9.55, 8.28, 7.40, 6.86, 6.58, 6.52, 6.49, 6.47, 6.45]}
data_9_months = {'strikes': [95, 96, 97, 98, 99, 100, 101, 102, 103, 104],
                 'prices':[11.79 ,8.95 ,8.07 ,7.03 ,6.18 ,6.04 ,5.76 ,5.50 ,5.50 ,5.50]}
data_6_months = {'strikes': [95, 96, 97, 98, 99, 100, 101, 102, 103, 104],
                  'prices': [10.71 ,8.28 ,6.91 ,6.36 ,5.29 ,5.07 ,4.76 ,4.47 ,4.37 ,4.37]}
data_3_months = {'strikes': [95, 96, 97, 98, 99, 100, 101, 102, 103, 104],
                  'prices': [8.67, 7.14, 5.98, 4.93, 4.09, 3.99, 3.43, 3.01, 2.72, 2.72]}

df_12_months = pd.DataFrame(data_12_months)
df_9_months = pd.DataFrame(data_9_months)
df_6_months = pd.DataFrame(data_6_months)
df_3_months = pd.DataFrame(data_3_months)

# On utilise l'algorithme de Newton Raphson pour trouver les volatilités implicites

df_12_months["implied_vol"] = df_12_months.apply(lambda row: newton_raphson_implied_vol(row))
df_9_months["implied_vol"] = df_9_months.apply(lambda row: newton_raphson_implied_vol(row))
df_6_months["implied_vol"] = df_6_months.apply(lambda row: newton_raphson_implied_vol(row))
df_3_months["implied_vol"] = df_3_months.apply(lambda row: newton_raphson_implied_vol(row))

df_12_months_interp = interpolation(df_12_months,1)
df_9_months_interp = interpolation(df_9_months,0.75)
df_6_months_interp = interpolation(df_6_months,0.5)
df_3_months_interp = interpolation(df_3_months,0.25)

df_12_months['maturity'] = [1]*10
df_9_months['maturity'] = [0.75]*10
df_6_months['maturity'] = [0.5]*10
df_3_months['maturity'] = [0.25]*10

df_12_months_interp['maturity'] = [1]*100
df_9_months_interp['maturity'] = [0.75]*100
df_6_months_interp['maturity'] = [0.5]*100
df_3_months_interp['maturity'] = [0.25]*100

df_concatenated = pd.concat([df_12_months,df_9_months, df_6_months, df_3_months])
df_concatenated_interp = pd.concat([df_12_months_interp,df_9_months_interp, df_6_months_interp, df_3_months_interp])

df_concatenated.reset_index(drop=True,inplace=True)
df_concatenated_interp.reset_index(drop=True,inplace=True)

```

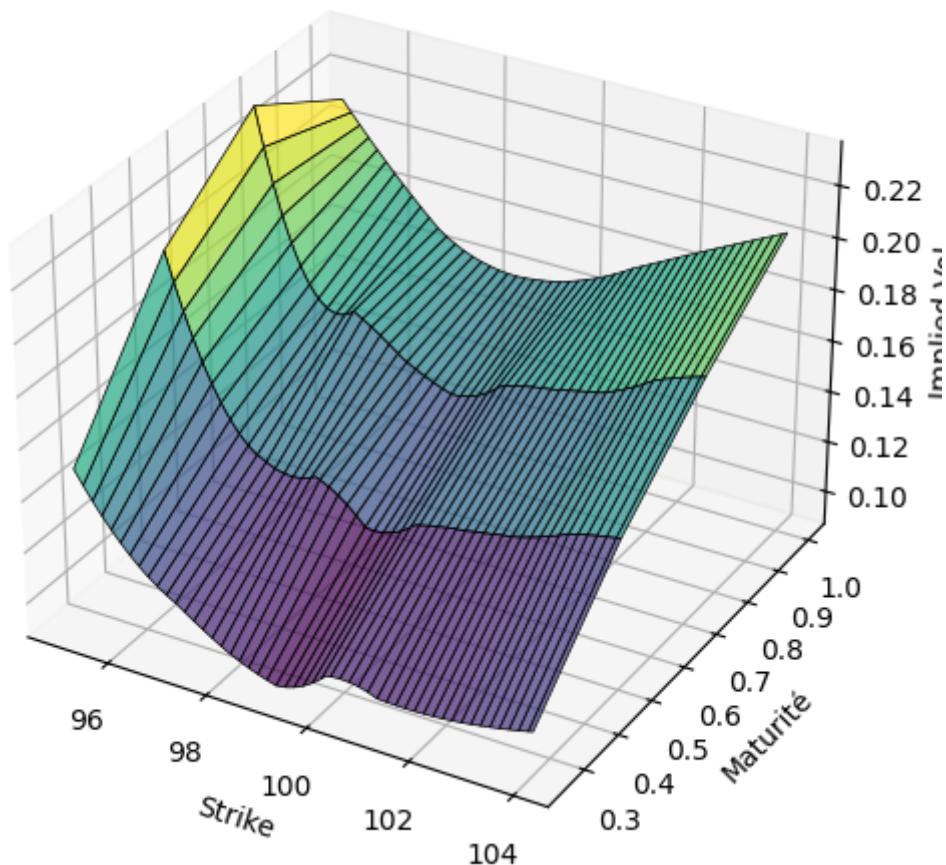
```
In [ ]: fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')

ax.plot_surface(df_concatenated_interp['strikes'].values.reshape(4, -1),
                df_concatenated_interp['maturity'].values.reshape(4, -1),
                df_concatenated_interp['implied_vol'].values.reshape(4, -1),
                cmap='viridis', edgecolor='k', linewidth=0.5, alpha=0.7)

ax.set_title("Nappe de volatilité")
ax.set_xlabel('Strike')
ax.set_ylabel('Maturité')
ax.set_zlabel('Implied Vol')

plt.show()
```

Nappe de volatilité



**3/ Déterminez la nappe de volatilité correspondant à ces 40 options. Proposer un prix, le plus juste possible, pour une option de strike 99.50 et de maturité 8 mois (justifier la méthode retenue).**

Proposer un prix, le plus juste possible, pour une option de strike 99.50 et de maturité 8 mois (justifier la méthode retenue).

```
In [ ]: def quadratic_interpolation(x_interpolated, x_original, y_original):
    y_interpolated = []
    for x in x_interpolated:
        for i in range(len(x_original) - 2):
            if x_original[i] <= x <= x_original[i + 2]:
                x1, x2, x3 = x_original[i], x_original[i + 1], x_original[i + 2]
```

```

        y1, y2, y3 = y_original[i], y_original[i + 1], y_original[i + 2]
        break

    A = np.array([[x1**2, x1, 1], [x2**2, x2, 1], [x3**2, x3, 1]])
    B = np.array([y1, y2, y3])
    a, b, c = np.linalg.solve(A, B)
    y = a*x**2 + b*x + c
    y_interpolated.append(y)

return y_interpolated

```

On test l'interpolation quadratique avec une valeur que l'on connaît pour estimer son erreur.  
Ici avec un Call K=99 T=1

```

In [ ]: volatility_interpolated = quadratic_interpolation([99], data["strikes"], data["imp"]

print('K = 99')
print(f"The interpolated volatility = {volatility_interpolated[0]:.4f}")
print(f"Implied vol = {float(data[data['strikes']==99]['implied_vol'].values):.4f}")

K = 99
The interpolated volatility = 0.1601
Implied vol = 0.1601

In [ ]: y_interpolated = quadratic_interpolation([8/12], [0.25, 0.5, 0.75, 1], list(df_conc

print('K = 99.50 and T = 8/12')
print(f"Implied vol interpolated : {y_interpolated[0]:.4f}")

K = 99.50 and T = 8/12
Implied vol interpolated : 0.1760

```

We will use interp2d as a benchmark to test our functions

```

In [ ]: from scipy.interpolate import interp2d
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)

strike_prices = list(df_concatenated_interp[df_concatenated_interp['maturity']==1][
maturities = [0.5, 0.75]

implied_volatilities = [
    list(df_6_months_interp['implied_vol'].values),
    list(df_9_months_interp['implied_vol'].values)
]

interp_func = interp2d(strike_prices, maturities, implied_volatilities, kind='linear')

x_target = 99.5
z_target = 8/12

implied_volatility_8_12 = interp_func(x_target, z_target)

print(f"Implied Volatility for maturity 8/12 and strike 99.5: {implied_volatility_8_12}")

```

Implied Volatility for maturity 8/12 and strike 99.5: 0.1371

We remark that the error is too big, and the quadratic interpolation function is not adapted to fit both interpolated strikes and maturity, so we will use the bilinear interpolation which is more adapted to interpolate 2 parameters

```
In [ ]: def bilinear_interpolation(x, y, points):
    # Voir sur https://fr.wikipedia.org/wiki/Interpolation_bilinéaire
    points = sorted(points)
    (x1, y1, q11), (x2, y2, q12), (x2, y1, q21), (x1, y2, q22) = points

    if x1 != x2 or x2 != x or y1 != y or y2 != y:
        raise ValueError("points do not form a rectangle")
    if not x1 <= x <= x2 or not y1 <= y <= y2:
        raise ValueError("(x, y) not within the rectangle")

    return (q11 * (x2 - x) * (y2 - y) +
            q21 * (x - x1) * (y2 - y) +
            q12 * (x2 - x) * (y - y1) +
            q22 * (x - x1) * (y - y1)) / ((x2 - x1) * (y2 - y1) + 0.0)
```

```
In [ ]: x_val = 99
y_val = 1
z_val = bilinear_interpolation(x_val,y_val,[(*98,0.75,float(df_9_months_interp[df_9_
(98,1,float(df_12_months_interp[df_12_n
(100,0.75,float(df_9_months_interp[df_9_mor
(100,1,float(df_12_months_interp[df_12_mont
```

```
In [ ]: true_z_val = float(df_12_months_interp[df_12_months_interp['strikes']==99]['implied
print(f'Implied vol from the dataframe : {true_z_val:.4f}')
print(f'Bilinear interpolated implied vol : {z_val:.4f}')
print(f'The error between bilinear interpolation and true value is : {abs(true_z_val - z_val)}')
```

Implied vol from the dataframe : 0.1601  
Bilinear interpolated implied vol : 0.1630  
The error between bilinear interpolation and true value is : 0.0030

```
In [ ]: t1 = float(df_concatenated_interp[(df_concatenated_interp['strikes'] == 99) & (df_concatenated_interp['maturity'] == '12M')])
print(f'Market Price: {t1:.4f}')
t2 = black_scholes_call_price(100,99,1,0,z_val)
print(f'BS Call price with bilinear interpolated implied vol : {t2:.4f}')
print(f'The error is {abs(t1-t2):.4f} or {abs((t1-t2)/t1):.4f}%')
```

Market Price: 6.8600  
BS Call price with bilinear interpolated implied vol : 6.9769  
The error is 0.1169 or 0.0170%

We consider the error is acceptable so we will go with this method

```
In [ ]: x_val = 99.5
y_val = 8/12
z_995_val = bilinear_interpolation(x_val,y_val,[(*99,0.75,float(df_9_months_interp[df_9_
(99,0.5,float(df_6_months_interp[df_6_month_
(100,0.75,float(df_9_months_interp[df_9_mor_
(100,0.5,float(df_6_months_interp[df_6_mont_
])
```

```
In [ ]: print(f'Implied vol from interp2d : {implied_volatility_8_12[0]:.4f}')
print(f'Bilinear interpolated implied vol : {z_995_val:.4f}')
print(f'The error between bilinear interpolation and true value is : {abs(implied_vo

prix_quad = black_scholes_call_price(100,99.5,8/12,0,y_interpolated[0])
print(f"Prix trouvé avec l'interpolation quadratique : {prix_quad:.4f}")
prix_bi = black_scholes_call_price(100,99.5,8/12,0,z_995_val)
print(f"Prix trouvé avec l'interpolation bilinéaire : {prix_bi:.4f}")
```

```

prix_interp2d = black_scholes_call_price(100,99.5,8/12,0,implied_volatility_8_12[0]
print("Prix théorique avec interp2d: {prix_interp2d:.4f}")

```

```

Implied vol from interp2d : 0.1371
Bilinear interpolated implied vol : 0.1446
The error between bilinear interpolation and true value is : 0.0075
Prix trouvé avec l'interpolation quadratique : 5.9683
Prix trouvé avec l'interpolation bilinéaire : 4.9494
Prix théorique avec interp2d: 4.7066

```

## 4/ Calibrer un modèle à volatilité locale de type SVI en suivant les étapes suivantes

- écrire l'algorithme de valorisation d'une option avec le modèle SVI, par EDP d'une part et par Monte Carlo d'autre part et comparer les résultats pour plusieurs jeux de paramètres

```

In [ ]: param = [0.1] * 5

def svi_model(x, param):
    a,b,rho,m,sigma = param
    variance = abs(a) + abs(b) * (rho * (x - m) + np.sqrt((x - m)**2 + sigma**2))
    return np.sqrt(variance)

def forward_price_calculation(S0, r, T):
    return S0 * np.exp(r*T)

def log_moneyness_calculation(strike, forward_price):
    return np.log(strike/forward_price)

```

```

In [ ]: def valorisation_EDP(S0,K,r,T,sigma,N,I,Smin,Smax):
    step=(Smax-Smin)/I
    dt=T/N
    S=np.arange(Smin,Smax+step,step)
    S_squared = np.array([x**2 for x in S])
    grille = np.arange(0,T+dt,dt)
    alpha=pow(sigma,2)/(2*pow(step,2))*S_squared
    beta=(r*S)/(2*step)
    option_prices = np.zeros((I,1))
    A = np.zeros((I,I))

    for i in range(I):
        option_prices[i] = max(S[i]-K,0)
    for i in range(1,len(A)+1):
        A[i-1][i-1]=2*alpha[i]+r
        if i>1:
            A[i-1][i-2]=-alpha[i]+beta[i]

        if i<I:
            A[i-1][i]=-alpha[i]-beta[i]

    y=np.zeros((I,1))
    alphaI=pow(sigma,2)/(2*pow(step,2))*S_squared[I]
    betaI=(r*S[I])/(2*step)
    y[0]=0
    alphaI=pow(sigma,2)/(2*pow(step,2))*S_squared[I]
    betaI=(r*S[I])/(2*step)

    for n in range(0, N):
        y[I-1]=(-alphaI-betaI)*(Smax-K*np.exp(-r*grille[n]))

```

```
option_prices = option_prices - dt * (A @ option_prices + y)
```

```
return pd.DataFrame(option_prices, index=S[:-1]).loc[S0]
```

```
In [ ]: for strike,market_price,maturity in zip(df_12_months["strikes"],df_12_months["price_forward_price"] = forward_price_calculation(S0, r, maturity)
log_moneyness = log_moneyness_calculation(strike, forward_price)
sigma = svi_model(log_moneyness, param)
option_price_EDP_SVI = valorisation_EDP(S0,strike,r,maturity,sigma,10000,200,0,
print(f"Prix par EDP pour K = {strike}: {option_price_EDP_SVI[0]:.2f}")
print(f"Market Price = {market_price:.2f}, Difference = {abs(option_price_EDP_SVI[0] - market_price)}")
```

```
Prix par EDP pour K = 95: 16.00
Market Price = 10.93, Difference = 5.07
Prix par EDP pour K = 96: 15.47
Market Price = 9.55, Difference = 5.92
Prix par EDP pour K = 97: 14.96
Market Price = 8.28, Difference = 6.68
Prix par EDP pour K = 98: 14.45
Market Price = 7.40, Difference = 7.05
Prix par EDP pour K = 99: 13.96
Market Price = 6.86, Difference = 7.10
Prix par EDP pour K = 100: 13.49
Market Price = 6.58, Difference = 6.91
Prix par EDP pour K = 101: 13.03
Market Price = 6.52, Difference = 6.51
Prix par EDP pour K = 102: 12.58
Market Price = 6.49, Difference = 6.09
Prix par EDP pour K = 103: 12.14
Market Price = 6.47, Difference = 5.67
Prix par EDP pour K = 104: 11.72
Market Price = 6.46, Difference = 5.26
```

```
In [ ]: def valorisation_MC_SVI(S0, param, r, nb_simulations, pas, strike, maturity):
dt = maturity / pas
paths = np.zeros((nb_simulations, pas + 1))
paths[:, 0] = S0

for i in range(pas):
forward_price = forward_price_calculation(S0, r, maturity)
log_moneyness = log_moneyness_calculation(strike, forward_price)
vol = svi_model(log_moneyness, param)
paths[:, i + 1] = paths[:, i] * np.exp((r - 0.5 * vol**2) * dt + np.sqrt(dt) * np.random.normal(0, 1, nb_simulations))

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

return option_value
```

- estimez les cinq paramètres permettant de réduire au plus l'erreur moyenne

```
In [ ]: for strike,market_price,maturity in zip(df_12_months["strikes"],df_12_months["price_option_price_mc_SVI = valorisation_MC_SVI(S0, param, r, 1000, 252, strike, maturity)
print(f"Prix par simulation de Monte Carlo pour K = {strike}: {option_price_mc_SVI[0]:.2f}")
print(f"Market Price = {market_price:.2f}, Difference = {abs(option_price_mc_SVI[0] - market_price)}")
```

```

Prix par simulation de Monte Carlo pour K = 95: 15.78
Market Price = 10.93, Difference = 4.85
Prix par simulation de Monte Carlo pour K = 96: 15.11
Market Price = 9.55, Difference = 5.56
Prix par simulation de Monte Carlo pour K = 97: 16.15
Market Price = 8.28, Difference = 7.87
Prix par simulation de Monte Carlo pour K = 98: 14.78
Market Price = 7.40, Difference = 7.38
Prix par simulation de Monte Carlo pour K = 99: 14.81
Market Price = 6.86, Difference = 7.95
Prix par simulation de Monte Carlo pour K = 100: 15.07
Market Price = 6.58, Difference = 8.49
Prix par simulation de Monte Carlo pour K = 101: 13.45
Market Price = 6.52, Difference = 6.93
Prix par simulation de Monte Carlo pour K = 102: 12.96
Market Price = 6.49, Difference = 6.47
Prix par simulation de Monte Carlo pour K = 103: 12.90
Market Price = 6.47, Difference = 6.43
Prix par simulation de Monte Carlo pour K = 104: 12.55
Market Price = 6.46, Difference = 6.09

```

```

In [ ]: def cost_function_SVI(param,args):
    strikes,vols_market,maturity=args
    vols_model = []
    for strike in strikes :
        forward_price = forward_price_calculation(S0, r, maturity)
        log_moneyness = log_moneyness_calculation(strike, forward_price)
        vols_model.append(svi_model(log_moneyness, param))
    return np.sum((vols_model - vols_market) ** 2)

In [ ]: def check_results_svi(results,data):
    calibrated_params = results
    pas=252
    diff_totale=0
    opt = []
    for strike,market_price,maturity in zip(data["strikes"],data["prices"], data["maturity"]):
        option_price_mc_SVI = valorisation_MC_SVI(S0, calibrated_params, r, 5000, p)
        print(f"Prix par simulation de Monte Carlo pour K = {strike}: {option_price_mc_SVI}")
        print(f"Market Price = {market_price:.2f}, Difference = {abs(option_price_mc_SVI - market_price)}")
        opt.append(option_price_mc_SVI)
        diff_totale+=abs(option_price_mc_SVI-market_price)
    print(diff_totale)
    fig, ax = plt.subplots()

    rect1 = ax.plot(data["strikes"], data["prices"],'-o',label='Prix de marché',color='red')
    rect2 = ax.plot(data["strikes"], opt, label='Prix modèle (SVI)')
    #rects3 = ax.plot(data["strikes"], prices,'-', label='Prix EDP (SVI)')

    ax.set_xlabel('Strikes')
    ax.set_ylabel('Prices')
    ax.set_title('Graphique des prix obtenus')
    ax.legend()
    plt.show()

In [ ]: result_svi_12m = nelder_mead(cost_function_SVI,np.array([0.1,0.1,0.1,0.1,0.1]),df_1)
check_results_svi(result_svi_12m,df_12_months)

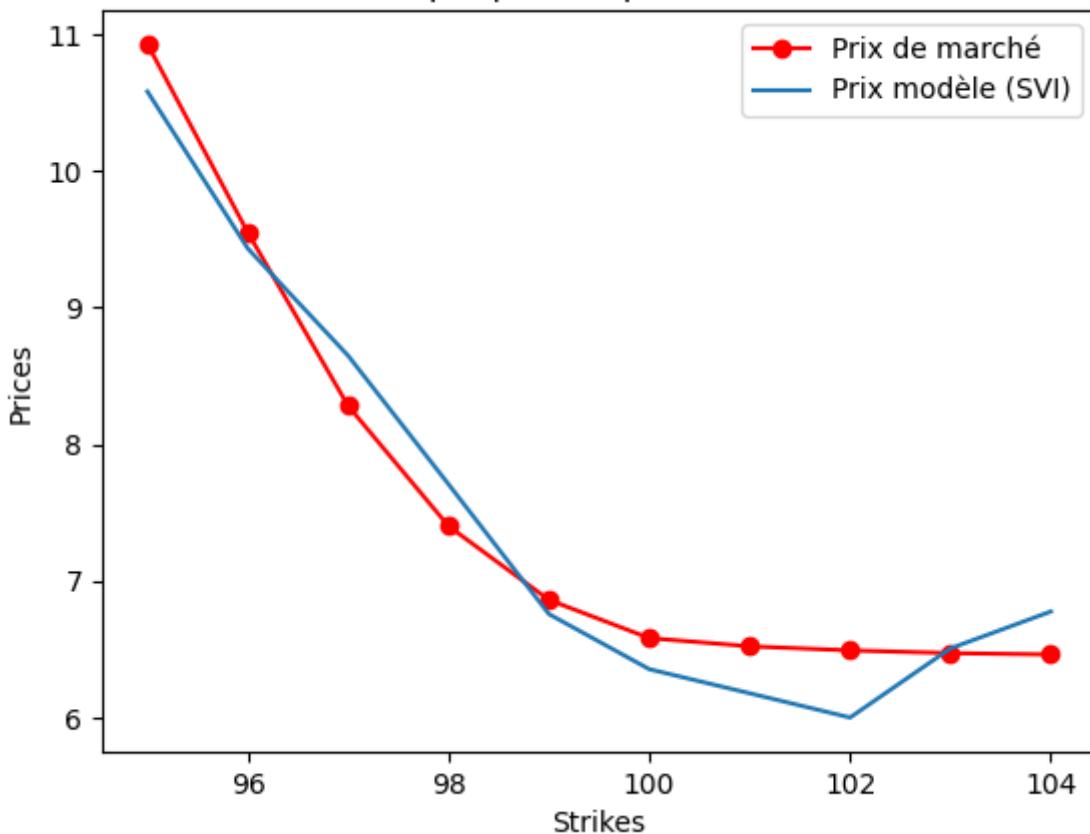
```

```

Prix par simulation de Monte Carlo pour K = 95: 10.58
Market Price = 10.93, Difference = 0.35
Prix par simulation de Monte Carlo pour K = 96: 9.43
Market Price = 9.55, Difference = 0.12
Prix par simulation de Monte Carlo pour K = 97: 8.65
Market Price = 8.28, Difference = 0.37
Prix par simulation de Monte Carlo pour K = 98: 7.71
Market Price = 7.40, Difference = 0.31
Prix par simulation de Monte Carlo pour K = 99: 6.76
Market Price = 6.86, Difference = 0.10
Prix par simulation de Monte Carlo pour K = 100: 6.35
Market Price = 6.58, Difference = 0.23
Prix par simulation de Monte Carlo pour K = 101: 6.18
Market Price = 6.52, Difference = 0.34
Prix par simulation de Monte Carlo pour K = 102: 6.00
Market Price = 6.49, Difference = 0.49
Prix par simulation de Monte Carlo pour K = 103: 6.50
Market Price = 6.47, Difference = 0.03
Prix par simulation de Monte Carlo pour K = 104: 6.77
Market Price = 6.46, Difference = 0.31
2.6557576577008586

```

Graphique des prix obtenus



```
In [ ]: result_svi_12m
```

```
Out[ ]: array([-4.11263928e-12,  7.73404374e-01,  2.38308840e-01,  3.79010349e-03,
   3.47985236e-02])
```

On teste notre EDP avec le modèle SVI calibré

```

In [ ]: prices_EDP = []
for strike,market_price,maturity in zip(df_12_months["strikes"],df_12_months["price"],
    forward_price = forward_price_calculation(S0, r, maturity)
    log_moneyness = log_moneyness_calculation(strike, forward_price)
    sigma = svi_model(log_moneyness, result_svi_12m)
    option_price_EDP_SVI = valorisation_EDP(S0,strike,r,maturity,sigma,1000,100,0,2
    prices_EDP.append(option_price_EDP_SVI[0])

```

```

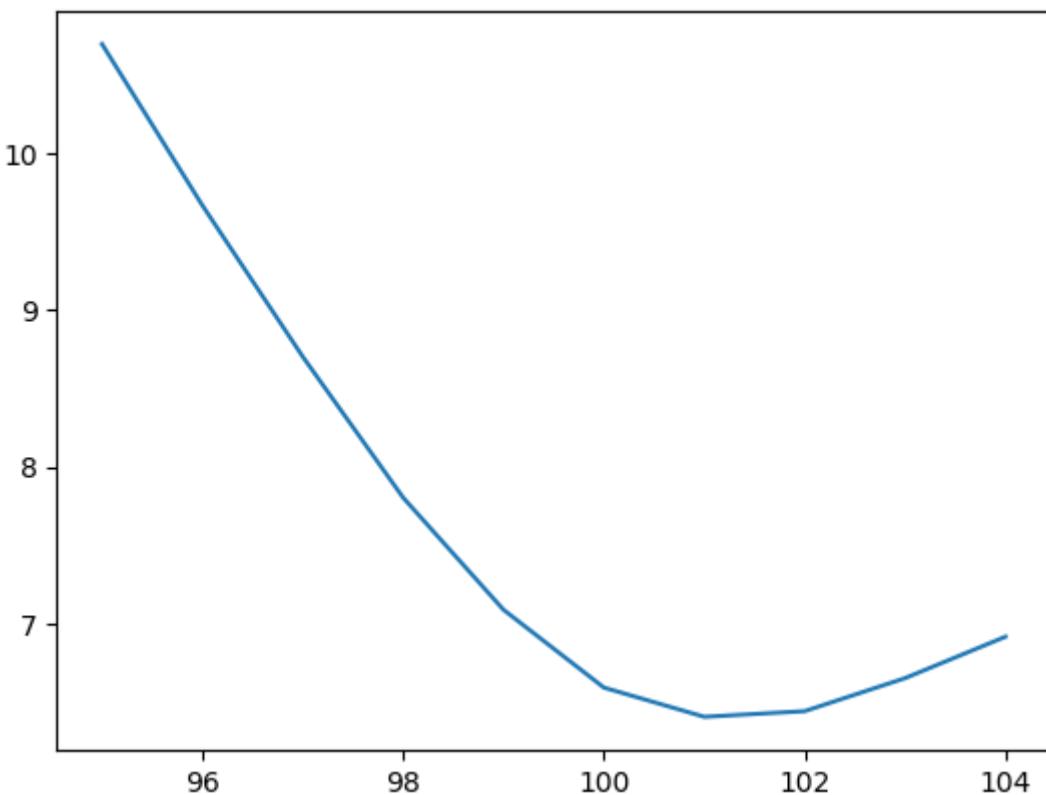
print(f"Prix avec EDP pour K = {strike}: {option_price_EDP_SVI[0]:.2f}")
print(f"Market Price = {market_price:.2f}, Difference = {abs(option_price_EDP_SVI[0] - market_price)}")

```

Prix avec EDP pour K = 95: 10.70  
 Market Price = 10.93, Difference = 0.23  
 Prix avec EDP pour K = 96: 9.67  
 Market Price = 9.55, Difference = 0.12  
 Prix avec EDP pour K = 97: 8.70  
 Market Price = 8.28, Difference = 0.42  
 Prix avec EDP pour K = 98: 7.80  
 Market Price = 7.40, Difference = 0.40  
 Prix avec EDP pour K = 99: 7.09  
 Market Price = 6.86, Difference = 0.23  
 Prix avec EDP pour K = 100: 6.59  
 Market Price = 6.58, Difference = 0.01  
 Prix avec EDP pour K = 101: 6.40  
 Market Price = 6.52, Difference = 0.12  
 Prix avec EDP pour K = 102: 6.44  
 Market Price = 6.49, Difference = 0.05  
 Prix avec EDP pour K = 103: 6.65  
 Market Price = 6.47, Difference = 0.18  
 Prix avec EDP pour K = 104: 6.91  
 Market Price = 6.46, Difference = 0.45

In [ ]: plt.plot(df\_12\_months["strikes"],prices\_EDP)

Out[ ]: [`<matplotlib.lines.Line2D at 0x158925b1c40>`]



On teste la valorisation des options 12 mois et comparons à nos vraies valeurs de marché.

Pour le set de paramètres nous utilisons celui déterminé dans la suite du projet, qui minimise l'erreur moyenne.

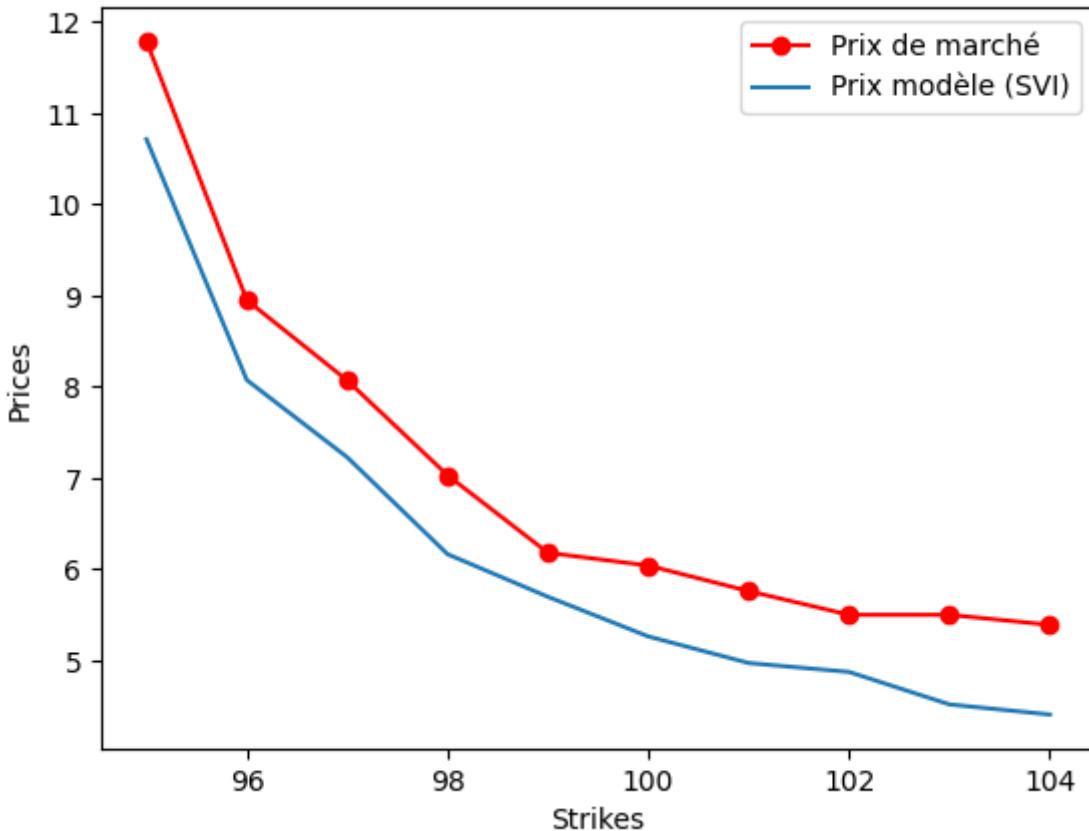
In [ ]: result\_svi\_9m = nelder\_mead(cost\_function\_SVI,np.array([0.1,0.1,0.1,0.1,0.1]),df\_9\_months)  
 check\_results\_svi(result\_svi\_9m,df\_9\_months)

```

Prix par simulation de Monte Carlo pour K = 95: 10.71
Market Price = 11.79, Difference = 1.08
Prix par simulation de Monte Carlo pour K = 96: 8.07
Market Price = 8.95, Difference = 0.88
Prix par simulation de Monte Carlo pour K = 97: 7.23
Market Price = 8.07, Difference = 0.84
Prix par simulation de Monte Carlo pour K = 98: 6.16
Market Price = 7.03, Difference = 0.87
Prix par simulation de Monte Carlo pour K = 99: 5.70
Market Price = 6.18, Difference = 0.48
Prix par simulation de Monte Carlo pour K = 100: 5.26
Market Price = 6.04, Difference = 0.78
Prix par simulation de Monte Carlo pour K = 101: 4.97
Market Price = 5.76, Difference = 0.79
Prix par simulation de Monte Carlo pour K = 102: 4.87
Market Price = 5.50, Difference = 0.63
Prix par simulation de Monte Carlo pour K = 103: 4.52
Market Price = 5.50, Difference = 0.98
Prix par simulation de Monte Carlo pour K = 104: 4.41
Market Price = 5.39, Difference = 0.98
8.299099180320598

```

Graphique des prix obtenus



Notre algorithme de valorisation par Monte Carlo nous donne des prix proches

```
In [ ]: result_svi_6m = nelder_mead(cost_function_SVI,np.array([0.1,0.1,0.1,0.1,0.1]),df_6m)
result_svi_3m = nelder_mead(cost_function_SVI,np.array([0.1,0.1,0.1,0.1,0.1]),df_3m)
```

```
In [ ]: result_NM = [result_svi_12m , result_svi_9m , result_svi_6m , result_svi_3m ]
NM=pd.DataFrame(result_NM,index=["12 months","9 months","6 months","3 months"],columns=NM)
```

Out[ ]:

	a	b	rho	m	sigma
<b>12 months</b>	-4.112639e-12	0.773404	0.238309	0.003790	0.034799
<b>9 months</b>	6.699056e-17	-5.461080	-0.950492	-0.055174	-0.013091
<b>6 months</b>	1.247204e-10	3.325039	-0.944192	-0.053479	-0.014539
<b>3 months</b>	-6.401172e-03	0.350345	-0.839731	-0.028842	-0.013455

In [ ]:

```

vols_market_12m = df_12_months["implied_vol"]
vols_market_9m = df_9_months["implied_vol"]
vols_market_6m = df_6_months["implied_vol"]
vols_market_3m = df_3_months["implied_vol"]

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(10,10))
fig.suptitle('SVI Calibrated model for each maturity')

strikes_plot = np.linspace(95, 104, 10)
vols_plot_12m = []
for strike in strikes_plot :
    forward_price = forward_price_calculation(S0, r, 1)
    log_moneyness = log_moneyness_calculation(strike, forward_price)
    vols_plot_12m.append(svi_model(log_moneyness, result_svi_12m))

ax1.plot(strikes_plot, vols_plot_12m, label='SVI Model')
ax1.scatter(strikes_plot, vols_market_12m, color='red', label='Market Data')
ax1.set_xlabel('Strike')
ax1.set_ylabel('Implied Volatility')
ax1.set_title("12-months option")
ax1.legend()

vols_plot_9m = []
for strike in strikes_plot :
    forward_price = forward_price_calculation(S0, r, 0.75)
    log_moneyness = log_moneyness_calculation(strike, forward_price)
    vols_plot_9m.append(svi_model(log_moneyness, result_svi_9m))

ax2.plot(strikes_plot, vols_plot_9m, label='SVI Model')
ax2.scatter(strikes_plot, vols_market_9m, color='red', label='Market Data')
ax2.set_xlabel('Strike')
ax2.set_ylabel('Implied Volatility')
ax2.set_title("9-months option")
ax2.legend()

vols_plot_6m = []
for strike in strikes_plot :
    forward_price = forward_price_calculation(S0, r, 0.5)
    log_moneyness = log_moneyness_calculation(strike, forward_price)
    vols_plot_6m.append(svi_model(log_moneyness, result_svi_6m))

ax3.plot(strikes_plot, vols_plot_6m, label='SVI Model')
ax3.scatter(strikes_plot, vols_market_6m, color='red', label='Market Data')
ax3.set_xlabel('Strike')
ax3.set_ylabel('Implied Volatility')
ax3.set_title("6-months option")
ax3.legend()

vols_plot_3m = []
for strike in strikes_plot :
    forward_price = forward_price_calculation(S0, r, 0.25)
    log_moneyness = log_moneyness_calculation(strike, forward_price)
    vols_plot_3m.append(svi_model(log_moneyness, result_svi_3m))

```

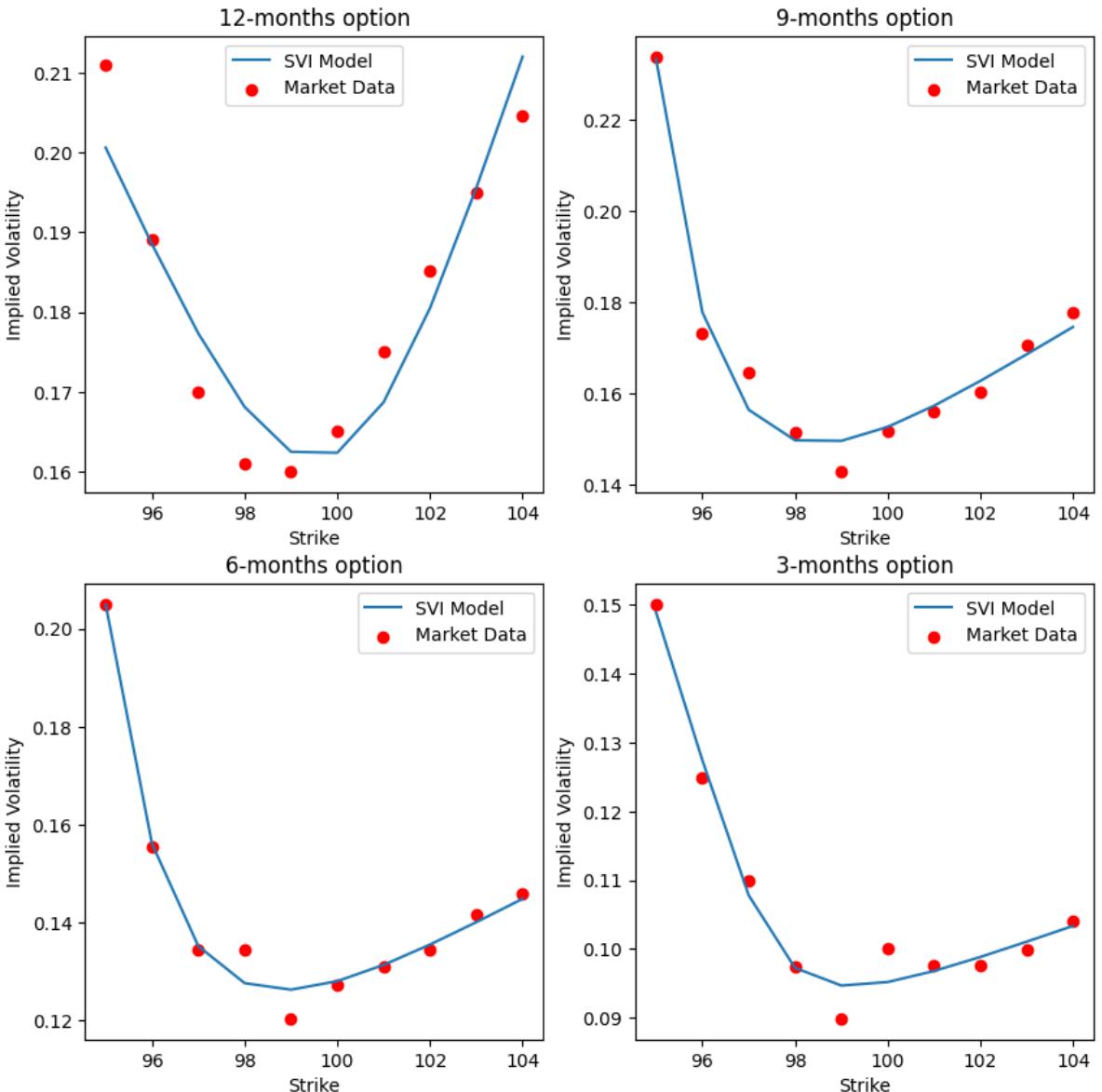
```

ax4.plot(strikes_plot, vols_plot_3m, label='SVI Model')
ax4.scatter(strikes_plot, vols_market_3m, color='red', label='Market Data')
ax4.set_xlabel('Strike')
ax4.set_ylabel('Implied Volatility')
ax4.set_title("3-months option")
ax4.legend()

plt.show()

```

SVI Calibrated model for each maturity



- on fixe tous les paramètres aux valeurs estimées ci-avant (sauf  $a$  et  $b$ ) et on va calibrer les deux paramètres  $a$  et  $b$  pour chaque option : vous obtiendrez donc une nappe pour chacun de ces deux paramètres et vous introduirez une contrainte de régularisation de ces deux nappes dans votre problème d'optimisation.

In [ ]: calibrated\_params\_svi = [result\_svi\_12m]+[result\_svi\_9m]+[result\_svi\_6m]+[result\_svi\_3m]

In [ ]: 

```
def svi_fixed(param, calibrated_params,x):
    a, b = param
    rho, m, sigma = calibrated_params[2:5]
    variance = abs(a) + abs(b) * (rho * (x - m) + np.sqrt((x - m)**2 + sigma**2))
```

```

    return np.sqrt(variance)

def cost_function_SVI_fixed(param, args):
    calibrated_params, strike, vols_market, maturity = args
    forward_price = forward_price_calculation(S0, r, maturity)
    log_moneyness = log_moneyness_calculation(strike, forward_price)
    vol = svi_fixed(param, calibrated_params, log_moneyness)
    return np.sum((vol - vols_market) ** 2)

```

```
In [ ]: a_values_12m = []
a_values_9m = []
a_values_6m = []
a_values_3m = []
b_values_12m = []
b_values_9m = []
b_values_6m = []
b_values_3m = []
```

```
In [ ]: for k,sigma in zip(strikes_plot, df_12_months['implied_vol']):
    res_12m = nelder_mead(cost_function_SVI_fixed, np.array([0.1, 0.6]), calibrated_
    a_12m, b_12m = res_12m
    a_values_12m.append(a_12m)
    b_values_12m.append(b_12m)
for k,sigma in zip(strikes_plot, df_9_months['implied_vol']):
    res_9m = nelder_mead(cost_function_SVI_fixed, np.array([0.1, 0.6]), calibrated_
    a_9m, b_9m = res_9m
    a_values_9m.append(a_9m)
    b_values_9m.append(b_9m)

for k,sigma in zip(strikes_plot, df_6_months['implied_vol']):
    res_6m = nelder_mead(cost_function_SVI_fixed, np.array([0.1, 0.6]), calibrated_
    a_6m, b_6m = res_6m
    a_values_6m.append(a_6m)
    b_values_6m.append(b_6m)

for k,sigma in zip(strikes_plot, df_3_months['implied_vol']):
    res_3m = nelder_mead(cost_function_SVI_fixed, np.array([0.1, 0.6]), calibrated_
    a_3m, b_3m = res_3m
    a_values_3m.append(a_3m)
    b_values_3m.append(b_3m)
```

```
In [ ]: a_values = a_values_12m+a_values_9m+a_values_6m+a_values_3m
b_values = b_values_12m+b_values_9m+b_values_6m+b_values_3m
strikes = df_concatenated["strikes"]
maturities = df_concatenated['maturity']
vol_market = df_concatenated['implied_vol']
```

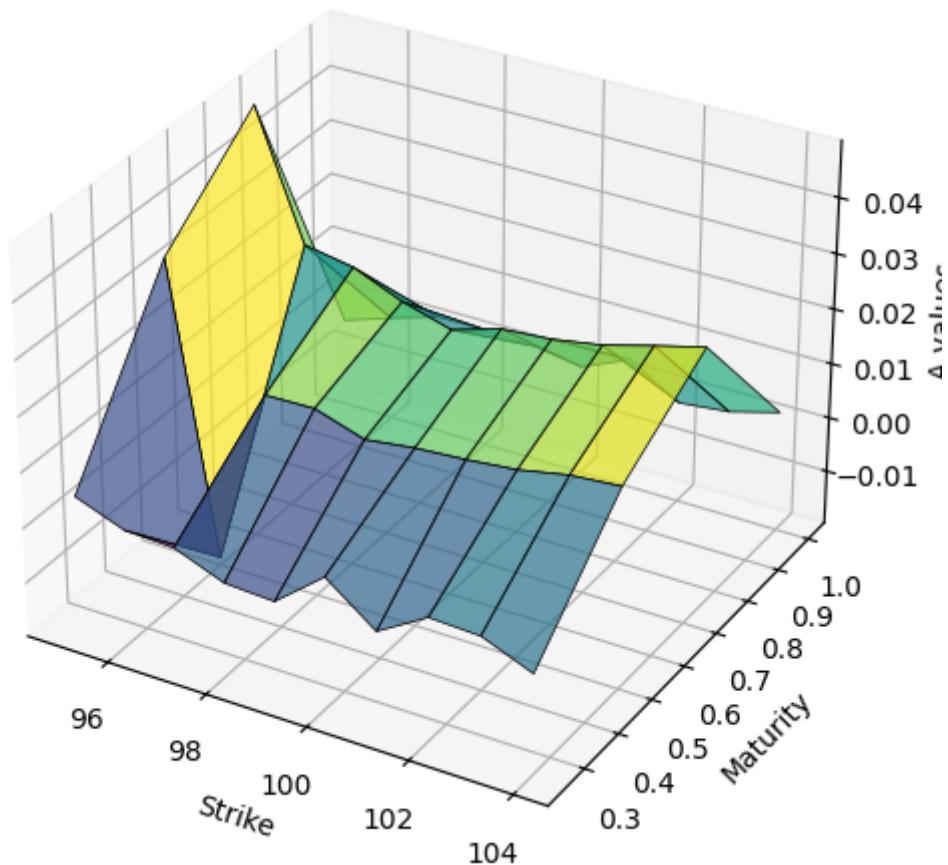
```
In [ ]: def surface(strikes,maturities,values):
    fig = plt.figure(figsize=(10, 6))
    ax = fig.add_subplot(111, projection='3d')

    ax.plot_surface(np.array(strikes).reshape(4,-1),
                    np.array(maturities).reshape(4,-1),
                    np.array(values).reshape(4,-1),
                    cmap='viridis', edgecolor='k', linewidth=0.5, alpha=0.7)

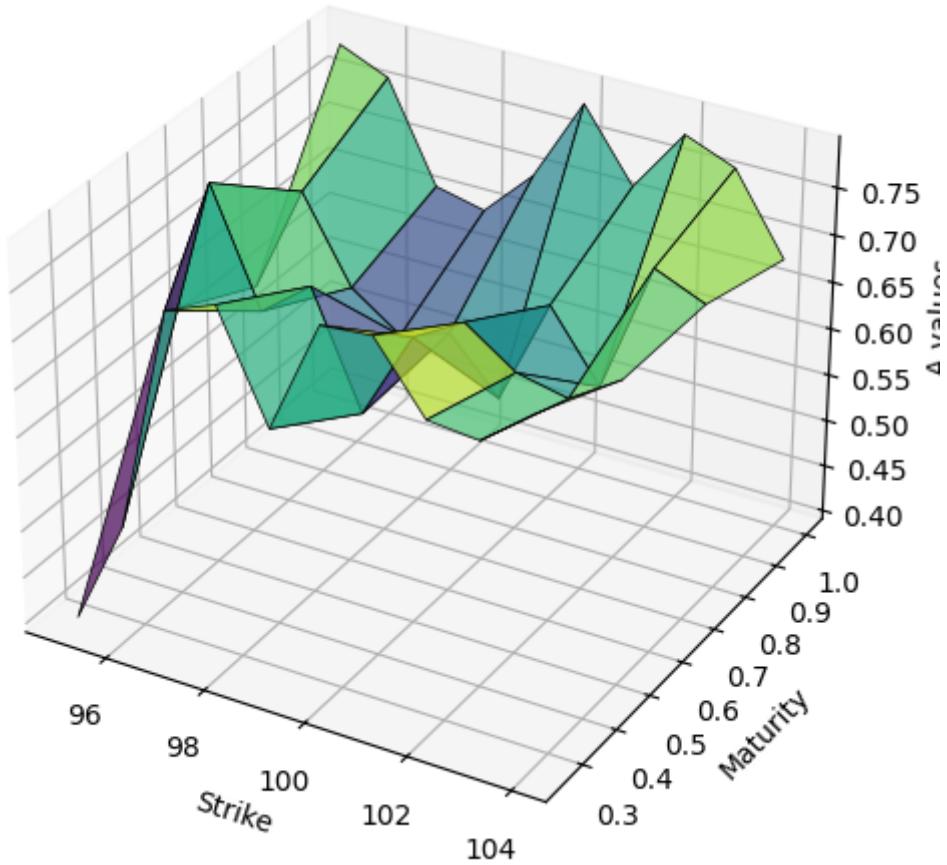
    ax.set_xlabel('Strike')
    ax.set_ylabel('Maturity')
    ax.set_zlabel('A values')

    plt.show()
```

```
In [ ]: surface(strikes,maturities,a_values)
```



```
In [ ]: surface(strikes,maturities,b_values)
```



```
In [ ]: fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(10,10))
fig.suptitle('SVI Calibrated model for each maturity')

strikes_plot = np.linspace(95, 104, 10)
vols_plot_12m = []
for strike,a,b in zip(strikes_plot,a_values_12m,b_values_12m):
    forward_price = forward_price_calculation(S0, r, 1)
    log_moneyness = log_moneyness_calculation(strike, forward_price)
    vols_plot_12m.append(svi_model(log_moneyness, list([a,b]) + list(result_svi_12m[2:])))

ax1.plot(strikes_plot, vols_plot_12m, label='SVI Model')
ax1.scatter(strikes_plot, vols_market_12m, color='red', label='Market Data')
ax1.set_xlabel('Strike')
ax1.set_ylabel('Implied Volatility')
ax1.set_title("12-months option")
ax1.legend()

vols_plot_9m = []
for strike,a,b in zip(strikes_plot,a_values_9m,b_values_9m) :
    forward_price = forward_price_calculation(S0, r, 0.75)
    log_moneyness = log_moneyness_calculation(strike, forward_price)
    vols_plot_9m.append(svi_model(log_moneyness,list([a,b]) + list(result_svi_9m[2:])))

ax2.plot(strikes_plot, vols_plot_9m, label='SVI Model')
ax2.scatter(strikes_plot, vols_market_9m, color='red', label='Market Data')
ax2.set_xlabel('Strike')
ax2.set_ylabel('Implied Volatility')
ax2.set_title("9-months option")
ax2.legend()

vols_plot_6m = []
for strike,a,b in zip(strikes_plot,a_values_6m,b_values_6m) :
    forward_price = forward_price_calculation(S0, r, 0.5)
    log_moneyness = log_moneyness_calculation(strike, forward_price)
    vols_plot_6m.append(svi_model(log_moneyness,list([a,b]) + list(result_svi_6m[2:])))

ax3.plot(strikes_plot, vols_plot_6m, label='SVI Model')
ax3.scatter(strikes_plot, vols_market_6m, color='red', label='Market Data')
ax3.set_xlabel('Strike')
ax3.set_ylabel('Implied Volatility')
ax3.set_title("6-months option")
ax3.legend()
```

```

log_moneyness = log_moneyness_calculation(strike, forward_price)
vols_plot_6m.append(svi_model(log_moneyness, list([a,b]) + list(result_svi_6m[2:])))

ax3.plot(strikes_plot, vols_plot_6m, label='SVI Model')
ax3.scatter(strikes_plot, vols_market_6m, color='red', label='Market Data')
ax3.set_xlabel('Strike')
ax3.set_ylabel('Implied Volatility')
ax3.set_title("6-months option")
ax3.legend()

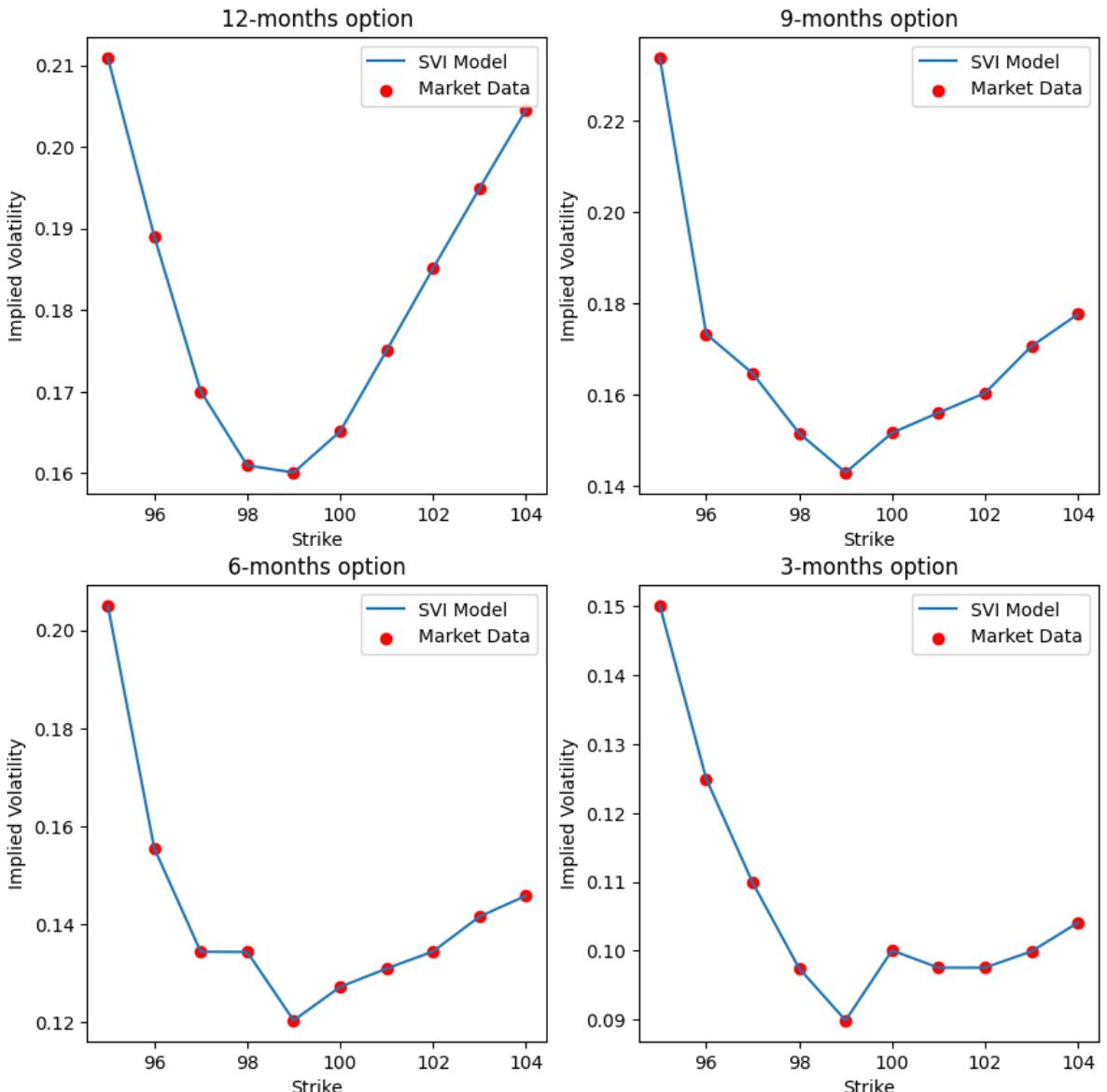
vols_plot_3m = []
for strike,a,b in zip(strikes_plot,a_values_3m,b_values_3m) :
    forward_price = forward_price_calculation(S0, r, 0.25)
    log_moneyness = log_moneyness_calculation(strike, forward_price)
    vols_plot_3m.append(svi_model(log_moneyness, list([a,b]) + list(result_svi_3m[2:])))

ax4.plot(strikes_plot, vols_plot_3m, label='SVI Model')
ax4.scatter(strikes_plot, vols_market_3m, color='red', label='Market Data')
ax4.set_xlabel('Strike')
ax4.set_ylabel('Implied Volatility')
ax4.set_title("3-months option")
ax4.legend()

plt.show()

```

### SVI Calibrated model for each maturity



```
In [ ]: def cost_function_reg_b(values,args):
    vol_market, lam, mu = args
    values_12m,values_9m,values_6m,values_3m = np.array_split(values,4)

    vol_model = []
    for strike,a,values in zip(strikes_plot,a_values_12m,values_12m) :
        forward_price = forward_price_calculation(S0, r, 1)
        log_moneyness = log_moneyness_calculation(strike, forward_price)
        vol_model.append(svi_model(log_moneyness, list([a,values]) + list(result_svi)))
    for strike,a,values in zip(strikes_plot,a_values_9m,values_9m) :
        forward_price = forward_price_calculation(S0, r, 0.75)
        log_moneyness = log_moneyness_calculation(strike, forward_price)
        vol_model.append(svi_model(log_moneyness, list([a,values]) + list(result_svi)))
    for strike,a,values in zip(strikes_plot,a_values_6m,values_6m) :
        forward_price = forward_price_calculation(S0, r, 0.5)
        log_moneyness = log_moneyness_calculation(strike, forward_price)
        vol_model.append(svi_model(log_moneyness, list([a,values]) + list(result_svi)))
    for strike,a,values in zip(strikes_plot,a_values_3m,values_3m) :
        forward_price = forward_price_calculation(S0, r, 0.25)
        log_moneyness = log_moneyness_calculation(strike, forward_price)
        vol_model.append(svi_model(log_moneyness, list([a,values]) + list(result_svi)))
```

```

error_vol = np.sum((vol_market-vol_model)**2)

values_surface = pd.DataFrame()
values_surface.index = strikes_plot
values_surface['12'] = values_12m
values_surface['9'] = values_9m
values_surface['6'] = values_6m
values_surface['3'] = values_3m

error_strike = 0
for i in range(1,values_surface.shape[0]):
    error_strike += (lam * np.sum(((values_surface.iloc[i]) - values_surface.i
error_maturity = 0
for m in range(3,15,3):
    error_maturity+=(mu * np.sum((values_surface[f'{m}']) - values_surface[f'{m}]

print(error_vol + error_strike + error_maturity)

return error_vol + error_strike + error_maturity

```

Ajouter dans le nelder mead la condition stop si < 0.001 pour b et 0.125 pour a

```
In [ ]: result_reg_b = nelder_mead(cost_function_reg_b,np.array(b_values),vol_market,1,1)[0]
```

```

In [ ]: def cost_function_reg_a(values,args):

    vol_market, lam, mu = args
    values_12m, values_9m, values_6m, values_3m = np.array_split(values,4)

    vol_model = []
    for strike,values,b in zip(strikes_plot,values_12m,b_values_12m) :
        forward_price = forward_price_calculation(S0, r, 1)
        log_moneyness = log_moneyness_calculation(strike, forward_price)
        vol_model.append(svi_model(log_moneyness, list([values,b]) + list(result_svi)))
    for strike,values,b in zip(strikes_plot,values_9m,b_values_9m) :
        forward_price = forward_price_calculation(S0, r, 0.75)
        log_moneyness = log_moneyness_calculation(strike, forward_price)
        vol_model.append(svi_model(log_moneyness, list([values,b]) + list(result_svi)))
    for strike,values,b in zip(strikes_plot,values_6m,b_values_6m) :
        forward_price = forward_price_calculation(S0, r, 0.5)
        log_moneyness = log_moneyness_calculation(strike, forward_price)
        vol_model.append(svi_model(log_moneyness, list([values,b]) + list(result_svi)))
    for strike,values,b in zip(strikes_plot,values_3m,b_values_3m) :
        forward_price = forward_price_calculation(S0, r, 0.25)
        log_moneyness = log_moneyness_calculation(strike, forward_price)
        vol_model.append(svi_model(log_moneyness, list([values,b]) + list(result_svi)))

    error_vol = np.sum((vol_market-vol_model)**2)

    values_surface = pd.DataFrame()
    values_surface.index = strikes_plot
    values_surface['12'] = values_12m
    values_surface['9'] = values_9m
    values_surface['6'] = values_6m
    values_surface['3'] = values_3m

    error_strike = 0
    for i in range(1,values_surface.shape[0]):
        error_strike += (lam * np.sum(((values_surface.iloc[i]) - values_surface.i
    error_maturity = 0
    for m in range(3,15,3):
        error_maturity+=(mu * np.sum((values_surface[f'{m}']) - values_surface[f'{m}'])


```

```

        print(error_vol + error_strike + error_maturity)

    return error_vol + error_strike + error_maturity

```

In [ ]: result\_reg\_a = nelder\_mead(cost\_function\_reg\_a,np.array(a\_values),vol\_market,1,1)[0]

In [ ]: surface(strikes,maturities,a\_values)

In [ ]: surface(strikes,maturities,result\_reg\_a)

In [ ]: surface(strikes,maturities,b\_values)

In [ ]: surface(strikes,maturities,result\_reg\_b)

**5/ En utilisant les nappes obtenues pour le modèle SVI (à la dernière étape), proposer encore un prix, le plus juste possible, pour une option de strike 99.50 et de maturité 8 mois (justifier la méthode retenue).**

```

In [ ]: df_concatenated['a_values'] = a_values
df_concatenated['b_values'] = b_values

In [ ]: x_val = 99
y_val = 1
z_val = bilinear_interpolation(x_val,y_val,[ (98,0.75,float(df_concatenated[(df_concatenated[(98,1,float(df_concatenated[(df_concatenated[(99,0.75,float(df_concatenated[(df_concatenated[(99,1,float(df_concatenated[(df_concatenated[
                )]
z2_val = bilinear_interpolation(x_val,y_val,[ (98,0.75,float(df_concatenated[(df_concatenated[(98,1,float(df_concatenated[(df_concatenated[(99,0.75,float(df_concatenated[(df_concatenated[(99,1,float(df_concatenated[(df_concatenated[
                )]

In [ ]: forward_price = forward_price_calculation(S0, r, 1)
log_moneyness = log_moneyness_calculation(99, forward_price)
test_imp = svi_model(log_moneyness, calibrated_params_svi[0])
price = black_scholes_call_price(100,99,1,0,test_imp)
price

Out[ ]: 6.956420019657095

In [ ]: float(df_concatenated[(df_concatenated['maturity'] == 1) & (df_concatenated['strike'] == 99.5)]
Out[ ]: 6.86

In [ ]: x_val = 99.5
y_val = 8/12
a_val = bilinear_interpolation(x_val,y_val,[ (99,0.5,float(df_concatenated[(df_concatenated[(99,0.75,float(df_concatenated[(df_concatenated[(100,0.5,float(df_concatenated[(df_concatenated[(100,0.75,float(df_concatenated[(df_concatenated[
                )]
b_val = bilinear_interpolation(x_val,y_val,[ (99,0.5,float(df_concatenated[(df_concatenated[(99,0.75,float(df_concatenated[(df_concatenated[(100,0.5,float(df_concatenated[(df_concatenated[
                )

```

```

        (99,0.75, float(df_concatenated[(df_concatenated
(100,0.5, float(df_concatenated[(df_concatenated
(100,0.75, float(df_concatenated[(df_concatenated
    )
)
```

In [ ]:

```
def linear_interpolation(x0, y0, x1, y1, x):
    slope = (y1 - y0) / (x1 - x0)
    y = y0 + slope * (x - x0)
    return y
```

In [ ]:

```
rho_8m = linear_interpolation(6,calibrated_params_svi[2][2],9,calibrated_params_svi
m_8m = linear_interpolation(6,calibrated_params_svi[2][3],9,calibrated_params_svi[1]
sigma_8m = linear_interpolation(6,calibrated_params_svi[2][4],9,calibrated_params_s
```

In [ ]:

```
forward_price = forward_price_calculation(S0, r, 1)
log_moneyness = log_moneyness_calculation(99.5, forward_price)
imp_vol_svi_8 = svi_fixed([a_val,b_val],calibrated_params_svi[3],log_moneyness)
price_svi = black_scholes_call_price(100,99.5,8/12,0,imp_vol_svi_8)
print(f"Prix trouvé avec l'interpolation bilinéaire et param pour 9 mois : {price_s

forward_price_int = forward_price_calculation(S0, r, 1)
log_moneyness_int = log_moneyness_calculation(99.5, forward_price_int)
imp_vol_svi_8_int = svi_fixed([a_val,b_val],[0,0,rho_8m,m_8m,sigma_8m],log_moneynes
price_svi_inter = black_scholes_call_price(100,99.5,8/12,0,imp_vol_svi_8_int)
print(f"Prix trouvé avec l'interpolation bilinéaire de a et b et autres params intere
```

Prix trouvé avec l'interpolation bilinéaire et param pour 9 mois : 4.9968  
Prix trouvé avec l'interpolation bilinéaire de a et b et autres params interpolés pour 8 mois : 4.7942

4.7066 vrai prix

## 6/ écrire l'algorithme de valorisation d'une option avec le modèle PDV, par Monte Carlo (les seuls rendements considérés seront ceux simulés entre la date courante et la maturité)

In [ ]:

```
def K1(t,param):
    return abs(param[3]) * np.exp(-param[3] * t)

def K2(t,param):
    return abs(param[3]) * np.exp(-param[4] * t)

def R(x):
    epsilon = 1e-7 # Une petite constante
    return (x[:,1] / (x[:,0] + epsilon)) - 1
def PDV_model(param,x,t):
    rt = R(x)
    sigma = param[0] + param[1]*(sum(([K1(T-i/t,param)*rt for i in range(t)]))) +
# print(sigma)
    return sigma
```

In [ ]:

```
def valorisation_MC_PDV(S, param,r, nb_simulations, pas, strike, T):
    dt=T/pas

    paths = np.zeros((nb_simulations, pas + 1))
    paths[:, 0:2] = S

    for i in range(1,pas):
        vol = PDV_model(param,paths[0:, i-1:i+1],pas) #A chaque pas on calcule not
```

```

        paths[0:, i + 1] = paths[0:, i] * np.exp((r-0.5 * vol **2) * dt + vol * np.
option_payoffs = np.maximum(paths[0:, -1] - strike, 0)
option_value = np.exp(-r * T) * np.mean(option_payoffs) #payoff actualisé
return option_value,paths

```

```

In [ ]: def cost_function_PDV(param,args):
    S0,r, nb_simulations, pas, strikes, maturity,market_prices = args
    dt=maturity/pas
    model_prices = []
    paths = np.zeros((nb_simulations, pas + 1))
    paths[:, 0:2] = S0
    for strike in strikes:
        for i in range(1,pas):
            vol = PDV_model(param,paths[0:, i-1:i+1],pas) #A chaque pas on calcule
            paths[0:, i + 1] = paths[0:, i] * np.exp((r-0.5 * vol **2) * dt + vol *
            option_payoffs = np.maximum(paths[0:, -1] - strike, 0)
            model_prices.append(np.exp(-r * maturity) * np.mean(option_payoffs)) #payoff
    print(param,np.sum((model_prices - market_prices)**2))

    return np.sum((model_prices - market_prices)**2)

```

On valorise nos options 1 an et les comparons à notre prix de marché

```

In [ ]: # Parameters for the PDV model
lambda_1 = 0.5 # Lambda 1 ne peut pas etre negatif
lambda_2 = 0.3
beta_0 = 0.1
beta_1 = 0.2
beta_2 = 0.3
initial_params_PDV = [
beta_0,
beta_1,
beta_2,
lambda_1,
lambda_2
]

In [ ]: def check_result_PDV(param,data,maturity):
    nb_simulations = 2000
    maturity=1
    pas=50
    opt = []
    total_error = 0
    for strike,price in zip(data["strikes"],data["prices"]):
        option_price,paths = valorisation_MC_PDV(S0, param,r, nb_simulations, pas,
        print(f"Prix par simulation de Monte Carlo pour K = {strike} : {option_price}")
        opt.append(option_price)
        print(f"Market Price = {price:.2f}, Difference = {abs(option_price-price)}")
        total_error+=abs(option_price-price)
    print(f'total error :{total_error:.2f}')
    fig, ax = plt.subplots()

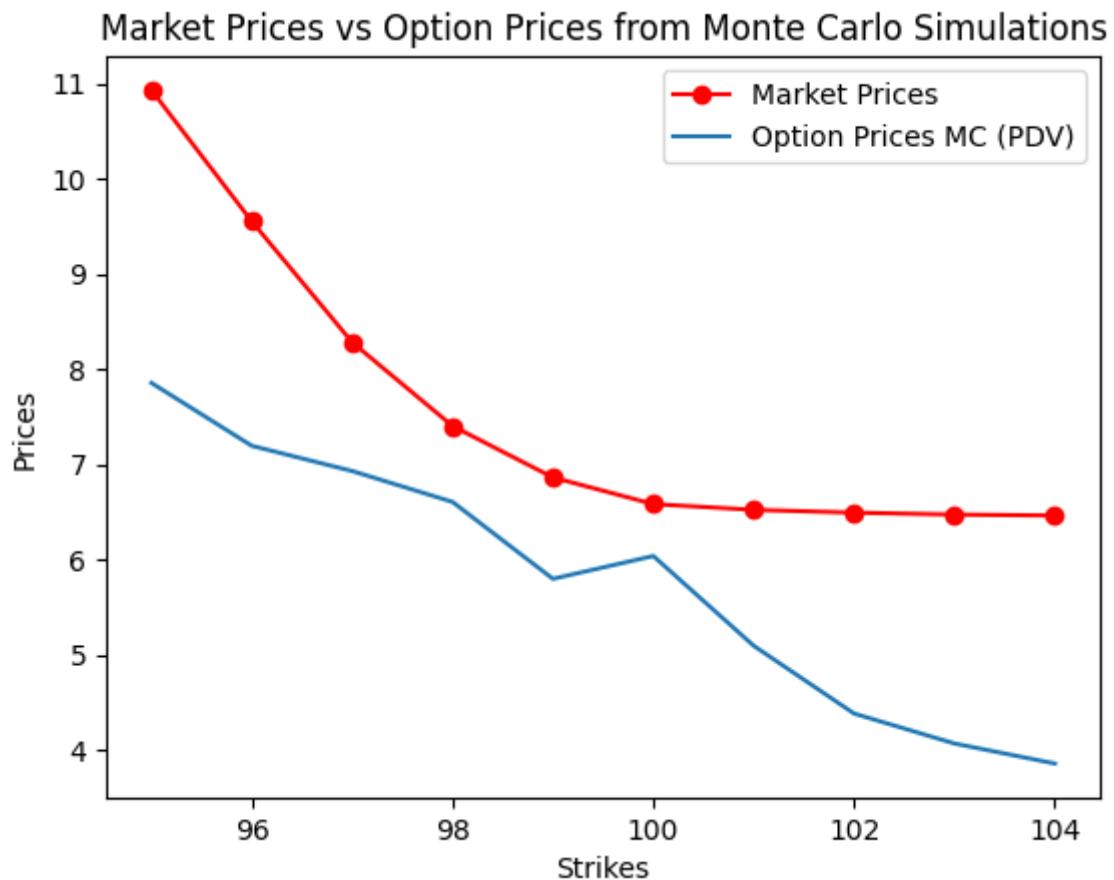
    rect1 = ax.plot(data["strikes"], data["prices"], '-o',label='Market Prices',co
    rect2 = ax.plot(data["strikes"], opt, label='Option Prices MC (PDV)')

    ax.set_xlabel('Strikes')
    ax.set_ylabel('Prices')
    ax.set_title('Market Prices vs Option Prices from Monte Carlo Simulations')
    ax.legend()
    plt.show()

```

```
In [ ]: check_result_PDV(initial_params_PDV,df_12_months,1)
```

Prix par simulation de Monte Carlo pour K = 95 : 7.85  
Market Price = 10.93, Difference = 3.08  
Prix par simulation de Monte Carlo pour K = 96 : 7.19  
Market Price = 9.55, Difference = 2.36  
Prix par simulation de Monte Carlo pour K = 97 : 6.93  
Market Price = 8.28, Difference = 1.35  
Prix par simulation de Monte Carlo pour K = 98 : 6.60  
Market Price = 7.40, Difference = 0.80  
Prix par simulation de Monte Carlo pour K = 99 : 5.79  
Market Price = 6.86, Difference = 1.07  
Prix par simulation de Monte Carlo pour K = 100 : 6.03  
Market Price = 6.58, Difference = 0.55  
Prix par simulation de Monte Carlo pour K = 101 : 5.09  
Market Price = 6.52, Difference = 1.43  
Prix par simulation de Monte Carlo pour K = 102 : 4.38  
Market Price = 6.49, Difference = 2.11  
Prix par simulation de Monte Carlo pour K = 103 : 4.06  
Market Price = 6.47, Difference = 2.41  
Prix par simulation de Monte Carlo pour K = 104 : 3.85  
Market Price = 6.46, Difference = 2.61  
total error :17.74



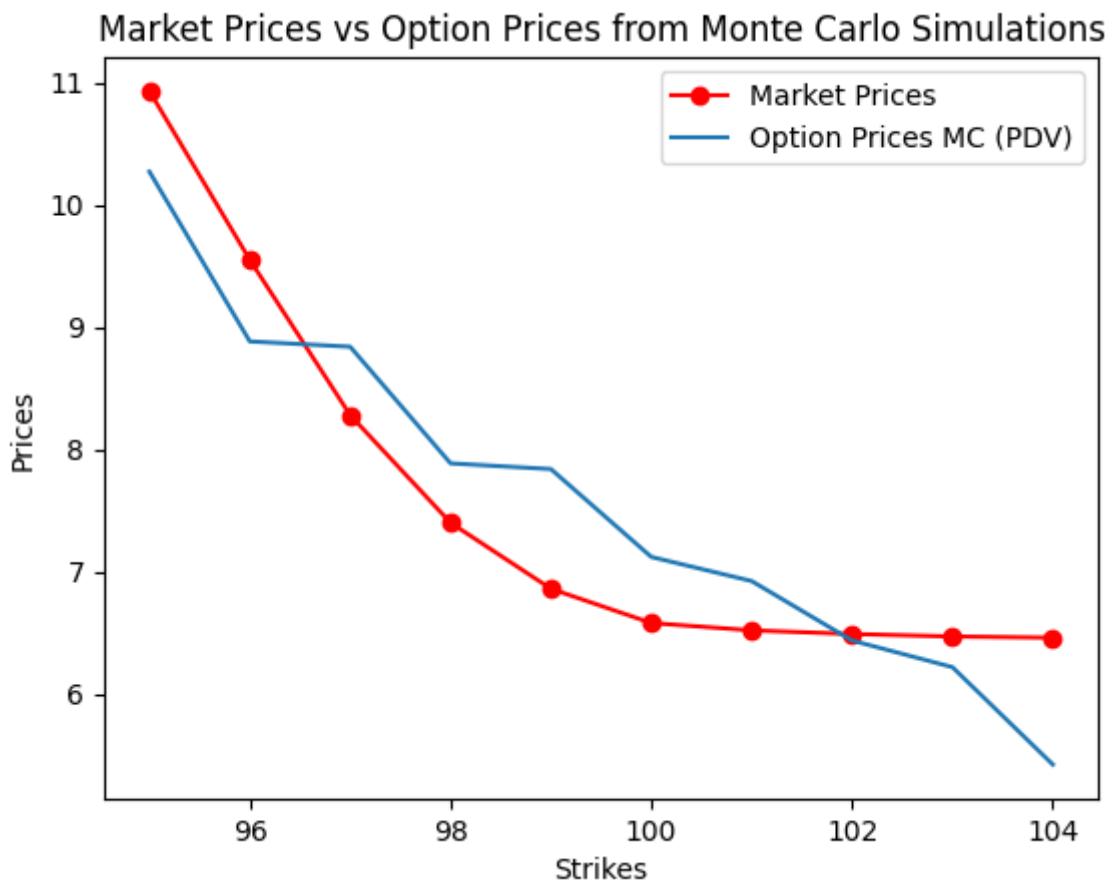
```
In [ ]: result_PDV_12 = nelder_mead(cost_function_PDV,np.array(initial_params_PDV),S0,r,500)
```

```
In [ ]: check_result_PDV(result_PDV_12,df_12_months,1)
```

```

Prix par simulation de Monte Carlo pour K = 95 : 10.27
Market Price = 10.93, Difference = 0.66
Prix par simulation de Monte Carlo pour K = 96 : 8.88
Market Price = 9.55, Difference = 0.67
Prix par simulation de Monte Carlo pour K = 97 : 8.84
Market Price = 8.28, Difference = 0.56
Prix par simulation de Monte Carlo pour K = 98 : 7.89
Market Price = 7.40, Difference = 0.49
Prix par simulation de Monte Carlo pour K = 99 : 7.84
Market Price = 6.86, Difference = 0.98
Prix par simulation de Monte Carlo pour K = 100 : 7.12
Market Price = 6.58, Difference = 0.54
Prix par simulation de Monte Carlo pour K = 101 : 6.92
Market Price = 6.52, Difference = 0.40
Prix par simulation de Monte Carlo pour K = 102 : 6.44
Market Price = 6.49, Difference = 0.05
Prix par simulation de Monte Carlo pour K = 103 : 6.22
Market Price = 6.47, Difference = 0.25
Prix par simulation de Monte Carlo pour K = 104 : 5.42
Market Price = 6.46, Difference = 1.04
total error : 5.63

```



```

In [ ]: result_PDV_9 = nelder_mead(cost_function_PDV,np.array(initial_params_PDV),S0,r,200)
In [ ]: result_PDV_6 = nelder_mead(cost_function_PDV,np.array(initial_params_PDV),S0,r, 10,
In [ ]: check_result_PDV(result_PDV_6,df_6_months,0.5)
In [ ]: result_PDV_3 = nelder_mead(cost_function_PDV,np.array(initial_params_PDV),S0,r, 200)
result_PDV_3
In [ ]: check_result_PDV(result_PDV_3,df_3_months,0.25)

```

```
In [ ]: result_PDV = [result_PDV_12 , result_PDV_9 , result_PDV_6 , result_PDV_3 ]
PDV=pd.DataFrame(result_PDV,index=["12 months","9 months","6 months","3 months"],co
```

```
Out[ ]:
```

	<b>beta_0</b>	<b>beta_1</b>	<b>beta_2</b>	<b>lambda_1</b>	<b>lambda_2</b>
<b>12 months</b>	0.186469	0.280632	-0.138468	0.052050	-0.146798
<b>9 months</b>	0.211102	0.031277	-0.729899	0.627062	-0.022394
<b>6 months</b>	0.052033	-0.497440	0.451744	0.334670	-0.170222
<b>3 months</b>	0.088400	0.156800	0.215200	0.323600	-0.025610

```
In [ ]: def PDV_model_fixed(param,calibrated_params,x,t):
    #param = [beta_1, beta_2]
    #calibrated_params = [beta_0,beta_1,beta_2,Lambda_1,Lambda_2]
    rt = R(x)
    sigma = calibrated_params[0] + param[0]*(sum(([K1(T-i/t,calibrated_params)*rt
    return sigma
def cost_function_PDV_fixed(param,args):
    calibrated_params,S0,r, nb_simulations, pas, strikes, maturity,market_price = args
    dt=maturity/pas

    paths = np.zeros((nb_simulations, pas + 1))
    paths[:, 0:2] = S0
    for i in range(1,pas):
        vol = PDV_model_fixed(param,calibrated_params,paths[0:, i-1:i+1],pas)
        paths[0:, i + 1] = paths[0:, i] * np.exp((r-0.5 * vol **2) * dt + vol * np.
    option_payoffs = np.maximum(paths[0:, -1] - strike, 0)
    model_price = (np.exp(-r * maturity) * np.mean(option_payoffs))
    #print(param,np.sum((model_price - market_price)**2))
    return np.sum((model_price - market_price)**2)
```

```
In [ ]: beta_1_values_12m = []
beta_1_values_9m = []
beta_1_values_6m = []
beta_1_values_3m = []
beta_2_values_12m = []
beta_2_values_9m = []
beta_2_values_6m = []
beta_2_values_3m = []
```

```
In [ ]: from tqdm import tqdm
```

```
In [ ]: for k,St in tqdm(zip(strikes_plot, df_12_months['prices'])):
    res_12m = nelder_mead(cost_function_PDV_fixed, np.array([0.1, 0.1]), result_PDV_
    beta_1_12m, beta_2_12m = res_12m
    beta_1_values_12m.append(beta_1_12m)
    beta_2_values_12m.append(beta_2_12m)
for k,St in tqdm(zip(strikes_plot, df_9_months['prices'])):
    res_9m = nelder_mead(cost_function_PDV_fixed, np.array([0.1, 0.1]), result_PDV_
    beta_1_9m, beta_2_9m = res_9m
    beta_1_values_9m.append(beta_1_9m)
    beta_2_values_9m.append(beta_2_9m)

for k,St in tqdm(zip(strikes_plot, df_6_months['prices'])):
    res_6m = nelder_mead(cost_function_PDV_fixed, np.array([0.1, 0.1]), result_PDV_
    beta_1_6m, beta_2_6m = res_6m
    beta_1_values_6m.append(beta_1_6m)
    beta_2_values_6m.append(beta_2_6m)
```

```

for k,St in tqdm(zip(strikes_plot, df_3_months['prices'])):
    res_3m = nelder_mead(cost_function_PDV_fixed, np.array([0.1, 0.1]), result_PDV)
    beta_1_3m, beta_2_3m = res_3m
    beta_1_values_3m.append(beta_1_3m)
    beta_2_values_3m.append(beta_2_3m)

```

```

10it [00:02, 3.39it/s]
10it [00:01, 7.43it/s]
10it [00:02, 4.51it/s]
10it [00:02, 3.65it/s]

```

```

In [ ]: fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(10, 10))
fig.suptitle('SVI Calibrated model for each maturity')

strikes_plot = np.linspace(95, 104, 10)
prices_plot_12m = []
for strike, beta_1, beta_2 in zip(strikes_plot, beta_1_values_12m, beta_2_values_12m):
    st, p = (valorisation_MC_PDV(S0, list([result_PDV_12[0]])) + list([beta_1, beta_2]))
    prices_plot_12m.append(st)
ax1.plot(strikes_plot, prices_plot_12m, label='PDV Model')
ax1.scatter(strikes_plot, df_12_months['prices'], color='red', label='Market Data')
ax1.set_xlabel('Strike')
ax1.set_ylabel('Prices')
ax1.set_title("12-months option")
ax1.legend()

prices_plot_9m = []
for strike, beta_1, beta_2 in zip(strikes_plot, beta_1_values_9m, beta_2_values_9m):
    st, p = (valorisation_MC_PDV(S0, list([result_PDV_9[0]])) + list([beta_1, beta_2]))
    prices_plot_9m.append(st)
ax2.plot(strikes_plot, prices_plot_9m, label='PDV Model')
ax2.scatter(strikes_plot, df_9_months['prices'], color='red', label='Market Data')
ax2.set_xlabel('Strike')
ax2.set_ylabel('Prices')
ax2.set_title("9-months option")
ax2.legend()

prices_plot_6m = []
for strike, beta_1, beta_2 in zip(strikes_plot, beta_1_values_6m, beta_2_values_6m):
    st, p = (valorisation_MC_PDV(S0, list([result_PDV_6[0]])) + list([beta_1, beta_2]))
    prices_plot_6m.append(st)
ax3.plot(strikes_plot, prices_plot_6m, label='PDV Model')
ax3.scatter(strikes_plot, df_6_months['prices'], color='red', label='Market Data')
ax3.set_xlabel('Strike')
ax3.set_ylabel('Prices')
ax3.set_title("6-months option")
ax3.legend()

prices_plot_3m = []
for strike, beta_1, beta_2 in zip(strikes_plot, beta_1_values_3m, beta_2_values_3m):
    st, p = (valorisation_MC_PDV(S0, list([result_PDV_3[0]])) + list([beta_1, beta_2]))
    prices_plot_3m.append(st)
ax4.plot(strikes_plot, prices_plot_3m, label='PDV Model')
ax4.scatter(strikes_plot, df_3_months['prices'], color='red', label='Market Data')
ax4.set_xlabel('Strike')
ax4.set_ylabel('Prices')
ax4.set_title("3-months option")
ax4.legend()

plt.show()

```

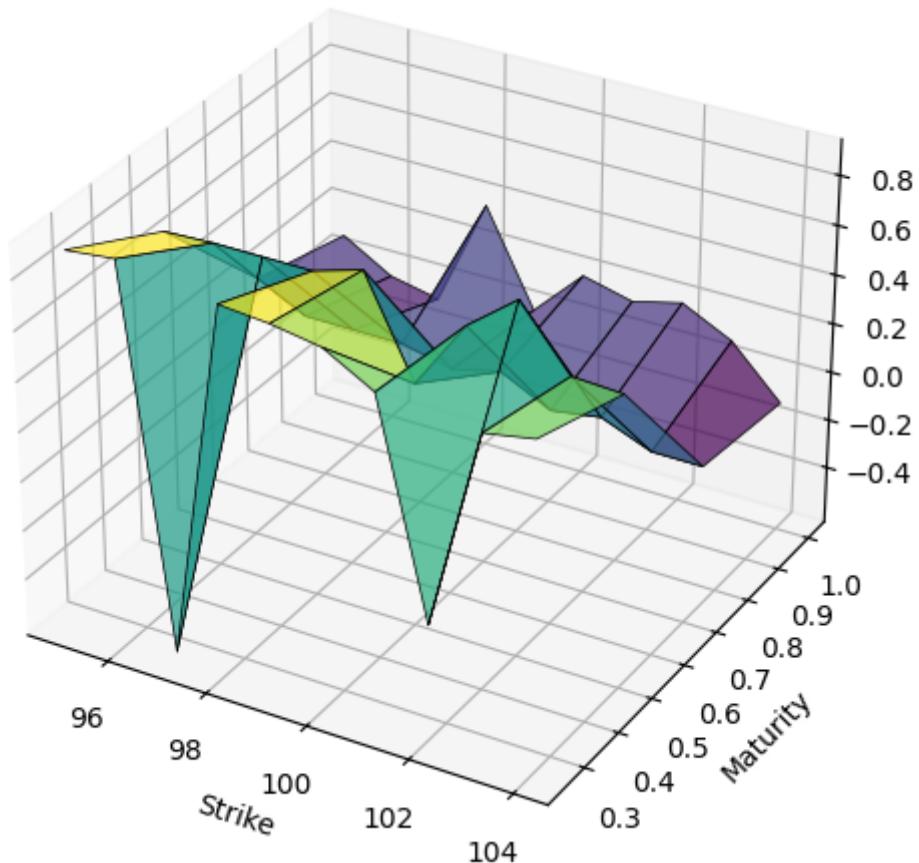
```

In [ ]: beta_1_values = beta_1_values_12m + beta_1_values_9m + beta_1_values_6m + beta_1_values_3m
beta_2_values = beta_2_values_12m + beta_2_values_9m + beta_2_values_6m + beta_2_values_3m

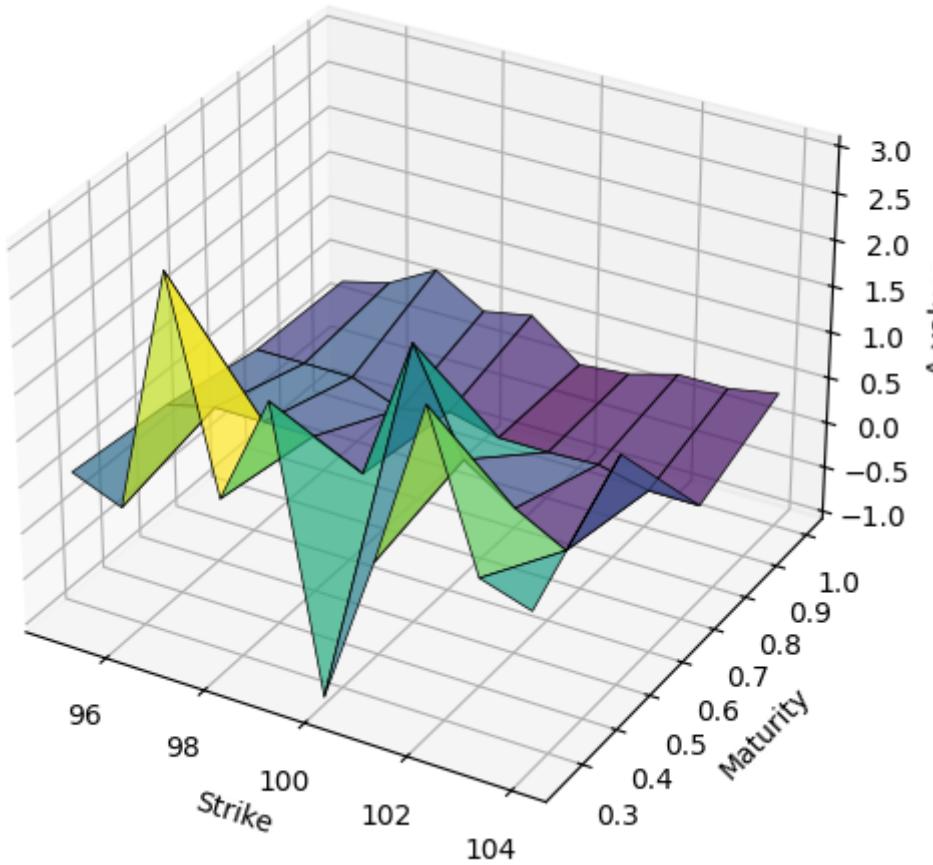
```

```
strikes = df_concatenated["strikes"]
maturities = df_concatenated['maturity']
vol_market = df_concatenated['implied_vol']
```

```
In [ ]: surface(strikes,maturities,beta_1_values)
```



```
In [ ]: surface(strikes,maturities,beta_2_values)
```



```
In [ ]: def cost_function_reg_beta_1(values,args):

    market_prices, lam, mu = args
    values_12m, values_9m, values_6m, values_3m = np.array_split(values,4)

    price_model = []
    for k,beta_1,beta_2 in zip(strikes_plot,values_12m,beta_2_values_12m):
        st,p = (valorisation_MC_PDV(S0,list([result_PDV_12[0]])) + list([beta_1,beta_2]))
        price_model.append(st)
    for k,beta_1,beta_2 in zip(strikes_plot,values_9m,beta_2_values_9m):
        st,p = (valorisation_MC_PDV(S0,list([result_PDV_9[0]])) + list([beta_1,beta_2]))
        price_model.append(st)
    for k,beta_1,beta_2 in zip(strikes_plot,values_6m,beta_2_values_6m):
        st,p = (valorisation_MC_PDV(S0,list([result_PDV_6[0]])) + list([beta_1,beta_2]))
        price_model.append(st)
    for k,beta_1,beta_2 in zip(strikes_plot,values_3m,beta_2_values_3m):
        st,p = (valorisation_MC_PDV(S0,list([result_PDV_3[0]])) + list([beta_1,beta_2]))
        price_model.append(st)

    error_price = np.sum((market_prices-price_model)**2)
    values_surface = pd.DataFrame()
    values_surface.index = strikes_plot
    values_surface['12'] = values_12m
    values_surface['9'] = values_9m
    values_surface['6'] = values_6m
    values_surface['3'] = values_3m

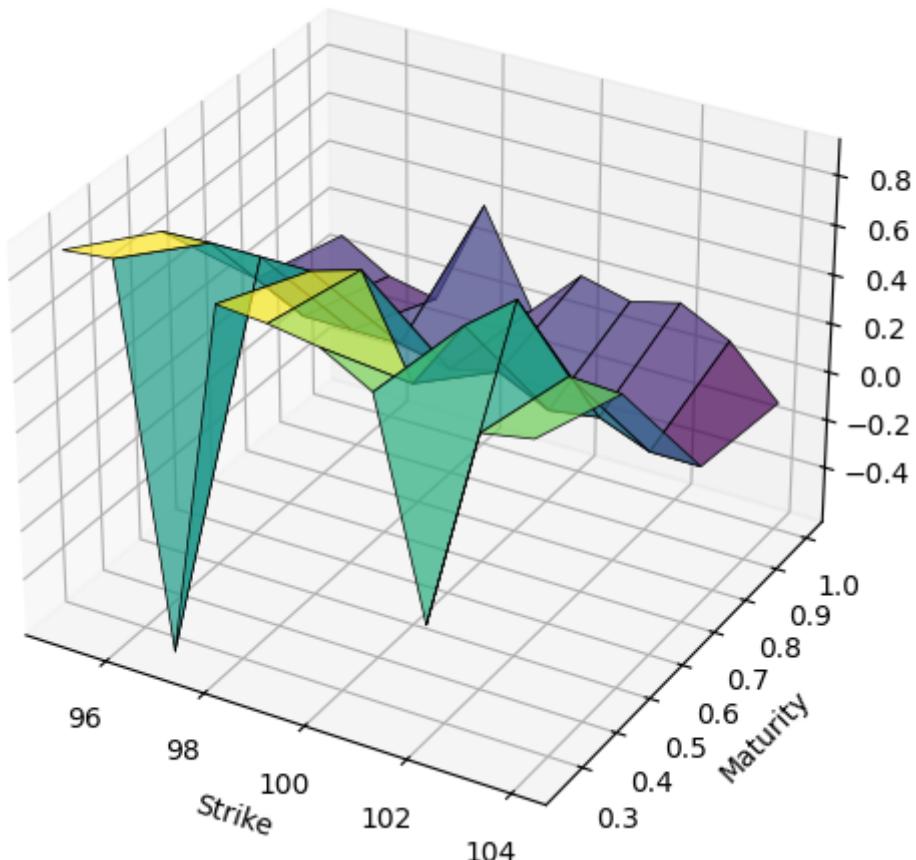
    error_strike = 0
    for i in range(1,values_surface.shape[0]):
        error_strike += (lam * np.sum(((values_surface.iloc[i]) - values_surface.i
    error_maturity = 0
    for m in range(3,15,3):
        error_maturity+=(mu * np.sum((values_surface[f'{m}'] - values_surface[f'{m}'])))
```

```
    print(error_price + error_strike + error_maturity)
```

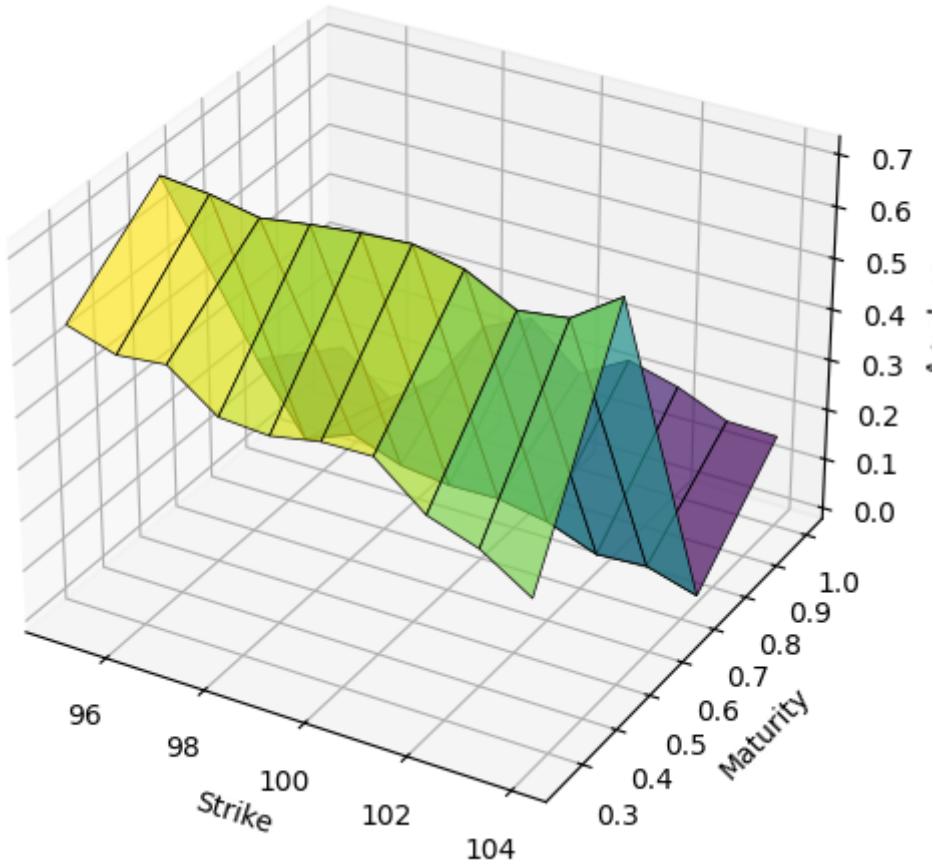
```
    return error_price + error_strike + error_maturity
```

```
In [ ]: result_reg_beta_1 = nelder_mead(cost_function_reg_beta_1,np.array(beta_1_values),df)
```

```
In [ ]: surface(strikes,maturities,beta_1_values)
```



```
In [ ]: surface(strikes,maturities,result_reg_beta_1['x'])
```



```
In [ ]: def cost_function_reg_beta_2(values,args):

    market_prices, lam, mu = args
    values_12m, values_9m, values_6m, values_3m = np.array_split(values,4)

    price_model = []
    for k,beta_1,beta_2 in zip(strikes_plot,beta_1_values_12m,values_12m):
        st,p = (valorisation_MC_PDV(S0,list([result_PDV_12[0]])) + list([beta_1,beta_2]))
        price_model.append(st)
    for k,beta_1,beta_2 in zip(strikes_plot,beta_1_values_9m,values_9m):
        st,p = (valorisation_MC_PDV(S0,list([result_PDV_9[0]])) + list([beta_1,beta_2]))
        price_model.append(st)
    for k,beta_1,beta_2 in zip(strikes_plot,beta_1_values_6m,values_6m):
        st,p = (valorisation_MC_PDV(S0,list([result_PDV_6[0]])) + list([beta_1,beta_2]))
        price_model.append(st)
    for k,beta_1,beta_2 in zip(strikes_plot,beta_1_values_3m,values_3m):
        st,p = (valorisation_MC_PDV(S0,list([result_PDV_3[0]])) + list([beta_1,beta_2]))
        price_model.append(st)

    error_price = np.sum((market_prices-price_model)**2)
    values_surface = pd.DataFrame()
    values_surface.index = strikes_plot
    values_surface['12'] = values_12m
    values_surface['9'] = values_9m
    values_surface['6'] = values_6m
    values_surface['3'] = values_3m

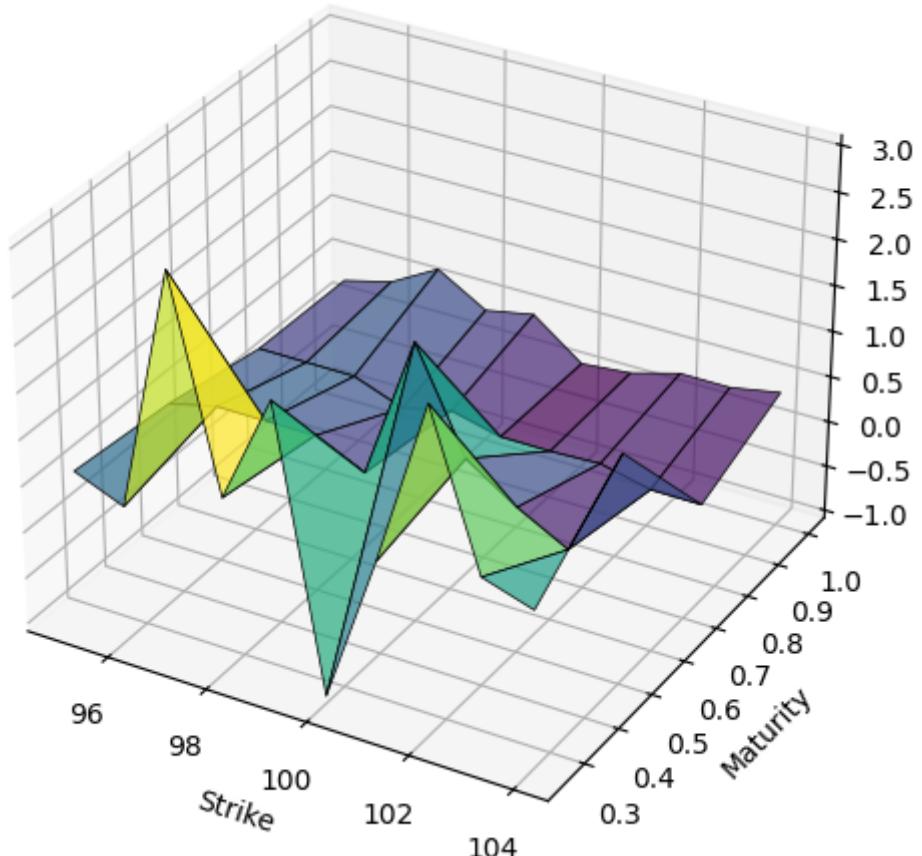
    error_strike = 0
    for i in range(1,values_surface.shape[0]):
        error_strike += (lam * np.sum(((values_surface.iloc[i]) - values_surface.i
    error_maturity = 0
    for m in range(3,15,3):
        error_maturity+=(mu * np.sum((values_surface[f'{m}'] - values_surface[f'{m}'])))
```

```
    print(error_price + error_strike + error_maturity)
```

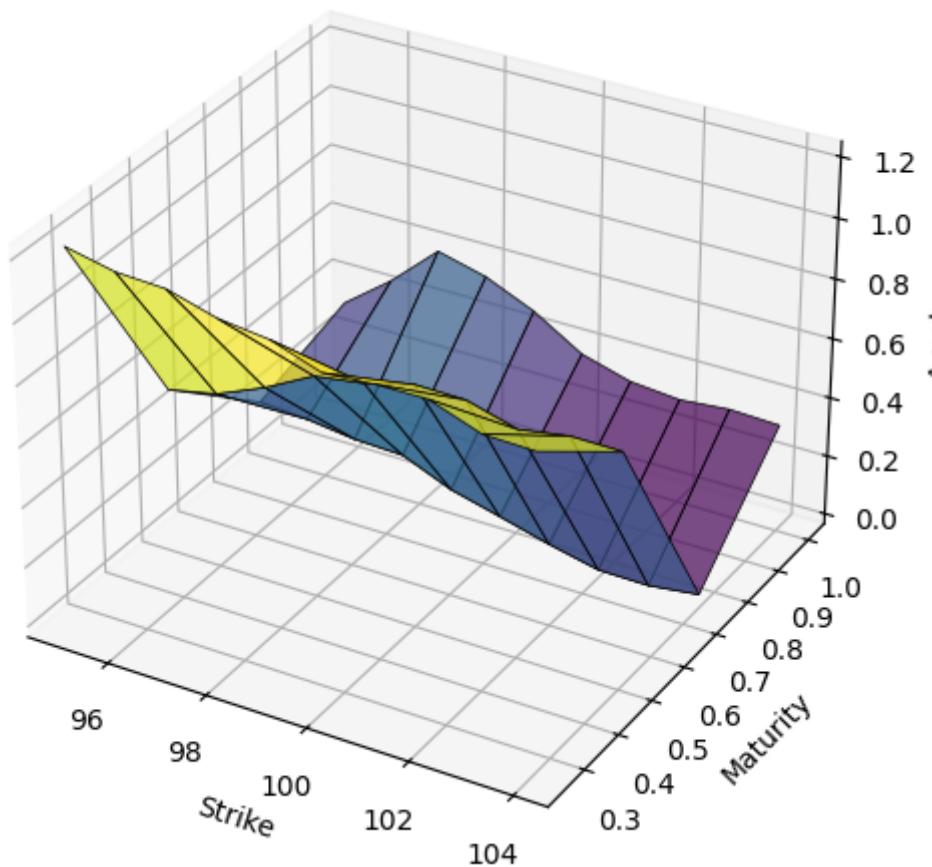
```
    return error_price + error_strike + error_maturity
```

```
In [ ]: result_reg_beta_2 = nelder_mead(cost_function_reg_beta_1,np.array(beta_2_values),df)
```

```
In [ ]: surface(strikes,maturities,beta_2_values)
```



```
In [ ]: surface(strikes,maturities,result_reg_beta_2['x'])
```



7/ Avec ce dernier modèle, proposer encore un prix, le plus juste possible, pour une option de strike 99.50 et de maturité 8 mois (justifier la méthode retenue)

In [ ]: df\_concatenated

Out[ ]:	<b>strikes</b>	<b>prices</b>	<b>implied_vol</b>	<b>maturity</b>	<b>a_values</b>	<b>b_values</b>	<b>beta_1</b>	<b>beta_2</b>
<b>0</b>	95	10.93	0.210948	1.00	-0.003590	0.786290	0.088506	0.327751
<b>1</b>	96	9.55	0.189023	1.00	-0.000636	0.763783	0.010153	0.450627
<b>2</b>	97	8.28	0.170031	1.00	0.002072	0.660010	0.079349	0.598432
<b>3</b>	98	7.40	0.160956	1.00	0.002224	0.647961	0.214094	0.555934
<b>4</b>	99	6.86	0.160079	1.00	0.001727	0.699773	0.257963	0.483449
<b>5</b>	100	6.58	0.165124	1.00	0.000333	0.789779	0.174446	0.382368
<b>6</b>	101	6.52	0.175031	1.00	0.004280	0.716055	0.228600	0.335066
<b>7</b>	102	6.49	0.185098	1.00	-0.001214	0.784568	0.202778	0.320407
<b>8</b>	103	6.47	0.194908	1.00	-0.000250	0.763311	0.162213	0.335590
<b>9</b>	104	6.46	0.204530	1.00	0.002237	0.681451	0.159859	0.329350
<b>10</b>	95	11.79	0.233804	0.75	0.048722	0.596259	0.209152	0.225585
<b>11</b>	96	8.95	0.173213	0.75	0.025876	0.713399	0.072894	0.214277
<b>12</b>	97	8.07	0.164578	0.75	0.024288	0.624987	0.112821	0.193438
<b>13</b>	98	7.03	0.151480	0.75	0.020534	0.588249	0.078483	0.187325
<b>14</b>	99	6.18	0.142861	0.75	0.017929	0.605721	0.066895	0.117717
<b>15</b>	100	6.04	0.151545	0.75	0.020618	0.550736	0.069341	0.074844
<b>16</b>	101	5.76	0.155976	0.75	0.021317	0.665338	0.052237	0.030310
<b>17</b>	102	5.50	0.160271	0.75	0.022808	0.594026	0.015834	-0.007002
<b>18</b>	103	5.50	0.170531	0.75	0.025267	0.733389	0.022424	-0.008820
<b>19</b>	104	5.39	0.177544	0.75	0.027558	0.710719	-0.006587	0.012355
<b>20</b>	95	10.71	0.205084	0.50	0.033890	0.646372	0.707847	0.517847
<b>21</b>	96	8.28	0.155474	0.50	-0.018344	0.795566	0.698167	0.551504
<b>22</b>	97	6.91	0.134345	0.50	0.014332	0.676749	0.679504	0.620511
<b>23</b>	98	6.36	0.134300	0.50	0.014524	0.717580	0.693408	0.706836
<b>24</b>	99	5.29	0.120325	0.50	0.011604	0.599572	0.703719	0.730472
<b>25</b>	100	5.07	0.127172	0.50	0.012750	0.694835	0.709605	0.773803
<b>26</b>	101	4.76	0.130909	0.50	0.013377	0.725199	0.689331	0.771792
<b>27</b>	102	4.47	0.134403	0.50	0.014265	0.688566	0.640268	0.725218
<b>28</b>	103	4.35	0.141521	0.50	0.016034	0.677329	0.653162	0.752737
<b>29</b>	104	4.14	0.145789	0.50	0.016764	0.711979	0.719793	0.751810
<b>30</b>	95	8.67	0.150113	0.25	0.004560	0.399191	0.564560	1.224780
<b>31</b>	96	7.14	0.124956	0.25	0.001199	0.513443	0.534648	1.189732
<b>32</b>	97	5.98	0.109874	0.25	0.000814	0.755099	0.544380	1.181970
<b>33</b>	98	4.93	0.097460	0.25	-0.002723	0.775640	0.474042	1.135147
<b>34</b>	99	4.09	0.089910	0.25	-0.003208	0.664942	0.467353	1.116122
<b>35</b>	100	3.99	0.100056	0.25	0.004040	0.785038	0.485276	1.059418

	strikes	prices	implied_vol	maturity	a_values	b_values	beta_1	beta_2
36	101	3.43	0.097553	0.25	-0.002814	0.789987	0.486934	1.071871
37	102	3.01	0.097559	0.25	0.002578	0.721125	0.406726	1.073197
38	103	2.72	0.099923	0.25	0.002186	0.716138	0.373734	1.013402
39	104	2.53	0.104078	0.25	-0.001559	0.758498	0.311137	1.009277

```
In [ ]: df_concatenated['beta_1'] = result_reg_beta_1['x']
df_concatenated['beta_2'] = result_reg_beta_2['x']
```

```
In [ ]: x_val = 99.5
y_val = 8/12
beta_1_val = bilinear_interpolation(x_val,y_val,[99,0.5,float(df_concatenated[(df_
(99,0.75,float(df_concatenated[(df_concatenated[(100,0.5,float(df_concatenated[(df_concatenated[(100,0.75,float(df_concatenated[(df_concatenated[(
)],

beta_2_val = bilinear_interpolation(x_val,y_val,[99,0.5,float(df_concatenated[(df_
(99,0.75,float(df_concatenated[(df_concatenated[(100,0.5,float(df_concatenated[(df_concatenated[(100,0.75,float(df_concatenated[(df_concatenated[
)],

)
```

```
In [ ]: beta_0_8m = linear_interpolation(6,result_PDV_6[0],9,result_PDV_9[0],8)
lambda_1_8m = linear_interpolation(6,result_PDV_6[3],9,result_PDV_9[3],8)
lambda_2_8m = linear_interpolation(6,result_PDV_6[4],9,result_PDV_9[4],8)
```

```
In [ ]: price_pdv_9912_8 = valorisation_MC_PDV(S0,[beta_0_8m,beta_1_val,beta_2_val,lambda_1
print(price_pdv_9912_8[0])
```

4.664352009663584

4.7066 vrai prix

### III – Test d’algorithmes d’optimisation

On veut comparer la performance de trois algorithmes d’optimisation : Nelder-Mead, le recuit simulé et l’essaim particulaire. Pour simplifier le problème on réduit sa dimension et on se place donc dans le cadre d’une estimation et non plus d’une calibration. Il s’agit d’estimer le modèle SVI (à cinq paramètres) qui minimise l’écart quadratique moyen entre prix de marché et prix de modèle pour les 40 options considérées. Trouver le couple de paramètres optimal avec Nelder-Mead puis avec recuit simulé et enfin avec essaim particulaire. Comparer les résultats obtenus : précision de l’estimation, rapidité de la convergence, stabilité de la solution.

```
In [ ]: def recuit_simule(f, x0,*args, q=0.2, c0=1e-2, alpha=0.99, max_iter=2000):
    # q = variance de la gaussienne
    # c0 = critère d'arrêt
```

```

# alpha = taux d'acceptation

for _ in range(max_iter):
    x_new = np.random.normal(x0, q) # Tirez un nouveau x à partir d'une distri

    delta_f = f(x_new,args) - f(x0,args)

    """
    La probabilité que xk +1 soit accepé est :
        1 si f (xk +1 ) ≤f (xk ) ;
        exp(-f (xk +1 )-f (xk ) / ck) sinon;
    """
    if delta_f <= 0 or np.random.rand() < np.exp(-delta_f / c0):
        x0 = x_new

    """
    Parametre de controle initialement grand
    (grand taux d'acceptation) puis
    décroissance lente :
    ck +1 = α * ck
    avec α = 0.99
    """
    c0 *= alpha

    result = {'x': x0, 'fun': f(x0,args)}
return result

```

```
In [ ]: result_svi_12m_recuit = recuit_simule(cost_function_SVI,np.array([0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1]),1000)
result_svi_9m_recuit = recuit_simule(cost_function_SVI,np.array([0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1]),1000)
result_svi_6m_recuit = recuit_simule(cost_function_SVI,np.array([0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1]),1000)
result_svi_3m_recuit = recuit_simule(cost_function_SVI,np.array([0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1]),1000)
```

```
In [ ]: result_concat_recuit= [result_svi_12m_recuit['x'] , result_svi_9m_recuit['x'] , result_svi_6m_recuit['x'] , result_svi_3m_recuit['x']]
res_recuit=pd.DataFrame(result_concat_recuit,index=["12 months","9 months","6 months","3 months"])
res_recuit
```

	a	b	rho	m	sigma
<b>12 months</b>	0.000312	-0.195806	0.064344	-0.062557	0.156394
<b>9 months</b>	0.004592	0.091084	-0.017651	0.233204	0.106941
<b>6 months</b>	-0.005482	-0.018410	-0.098095	0.540495	0.493437
<b>3 months</b>	0.004715	-0.043534	-0.860810	0.022316	0.137091

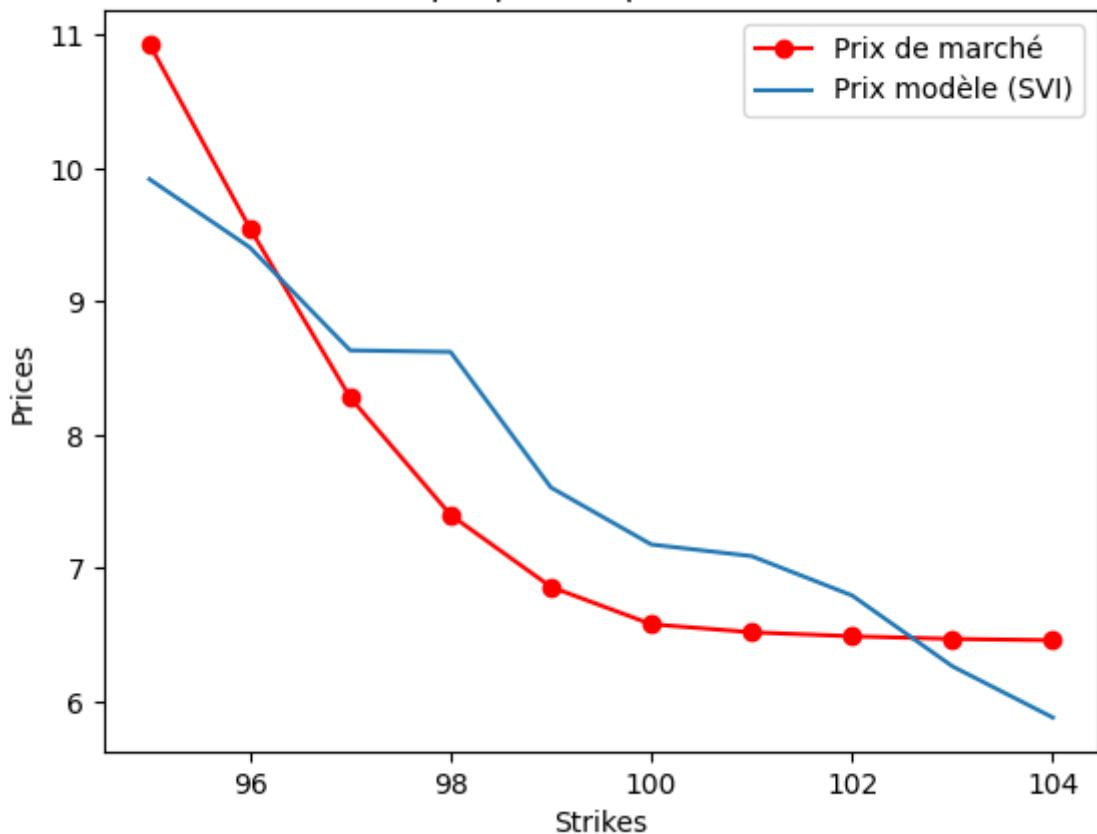
```
In [ ]: print(result_svi_12m_recuit)
print(result_svi_9m_recuit)
print(result_svi_6m_recuit)
print(result_svi_3m_recuit)

{'x': array([ 0.00031231, -0.19580635,   0.0643439 ,  -0.06255694,   0.15639389]), 'fun': 0.002912652684406095}
{'x': array([ 0.00459227,   0.09108445, -0.01765108,   0.23320426,   0.10694128]), 'fun': 0.005062087981608031}
{'x': array([-0.00548214, -0.01840983, -0.09809526,   0.54049495,   0.49343693]), 'fun': 0.004843206508760098}
{'x': array([ 0.00471495, -0.04353419, -0.86081034,   0.02231627,   0.13709122]), 'fun': 0.0018227832806760982}
```

```
In [ ]: check_results_svi(result_svi_12m_recuit['x'],df_12_months)
```

```
Prix par simulation de Monte Carlo pour K = 95: 9.92
Market Price = 10.93, Difference = 1.01
Prix par simulation de Monte Carlo pour K = 96: 9.41
Market Price = 9.55, Difference = 0.14
Prix par simulation de Monte Carlo pour K = 97: 8.63
Market Price = 8.28, Difference = 0.35
Prix par simulation de Monte Carlo pour K = 98: 8.62
Market Price = 7.40, Difference = 1.22
Prix par simulation de Monte Carlo pour K = 99: 7.61
Market Price = 6.86, Difference = 0.75
Prix par simulation de Monte Carlo pour K = 100: 7.18
Market Price = 6.58, Difference = 0.60
Prix par simulation de Monte Carlo pour K = 101: 7.09
Market Price = 6.52, Difference = 0.57
Prix par simulation de Monte Carlo pour K = 102: 6.80
Market Price = 6.49, Difference = 0.31
Prix par simulation de Monte Carlo pour K = 103: 6.26
Market Price = 6.47, Difference = 0.21
Prix par simulation de Monte Carlo pour K = 104: 5.88
Market Price = 6.46, Difference = 0.58
5.737266379468076
```

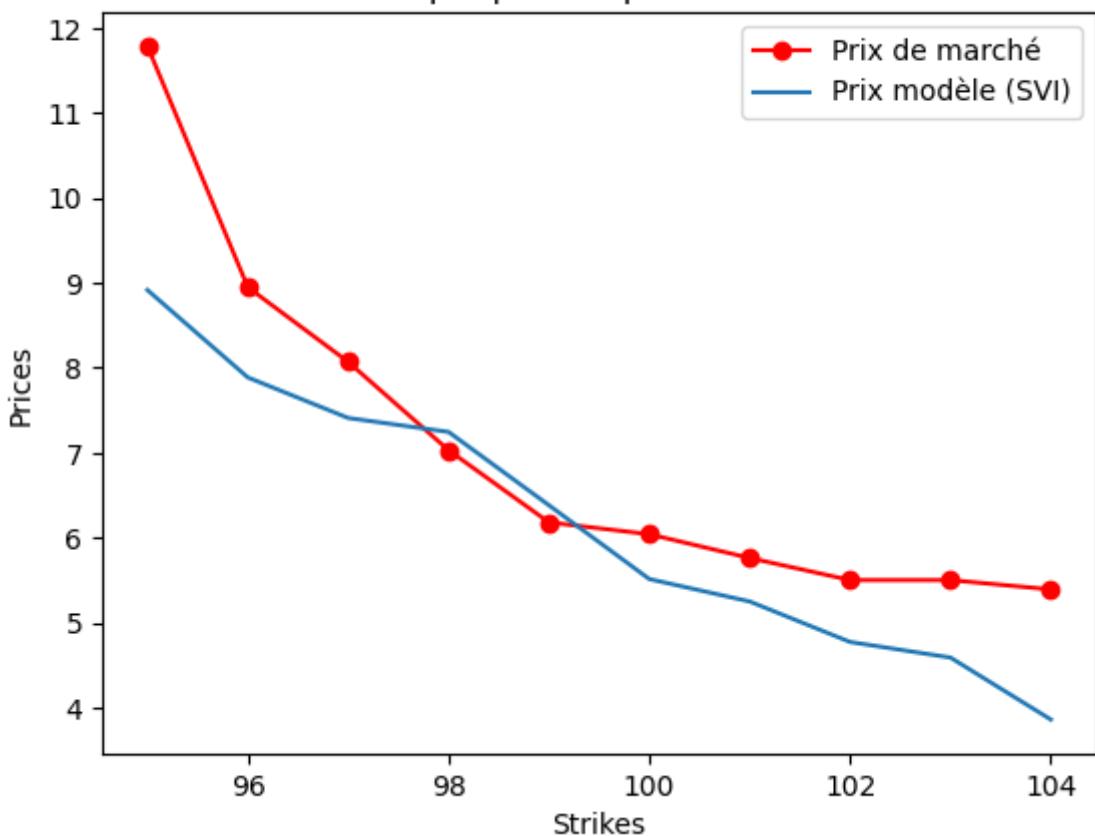
Graphique des prix obtenus



```
In [ ]: check_results_svi(result_svi_9m_recuit['x'],df_9_months)
```

```
Prix par simulation de Monte Carlo pour K = 95: 8.92
Market Price = 11.79, Difference = 2.87
Prix par simulation de Monte Carlo pour K = 96: 7.89
Market Price = 8.95, Difference = 1.06
Prix par simulation de Monte Carlo pour K = 97: 7.41
Market Price = 8.07, Difference = 0.66
Prix par simulation de Monte Carlo pour K = 98: 7.24
Market Price = 7.03, Difference = 0.21
Prix par simulation de Monte Carlo pour K = 99: 6.38
Market Price = 6.18, Difference = 0.20
Prix par simulation de Monte Carlo pour K = 100: 5.51
Market Price = 6.04, Difference = 0.53
Prix par simulation de Monte Carlo pour K = 101: 5.25
Market Price = 5.76, Difference = 0.51
Prix par simulation de Monte Carlo pour K = 102: 4.77
Market Price = 5.50, Difference = 0.73
Prix par simulation de Monte Carlo pour K = 103: 4.59
Market Price = 5.50, Difference = 0.91
Prix par simulation de Monte Carlo pour K = 104: 3.86
Market Price = 5.39, Difference = 1.53
9.22817075337001
```

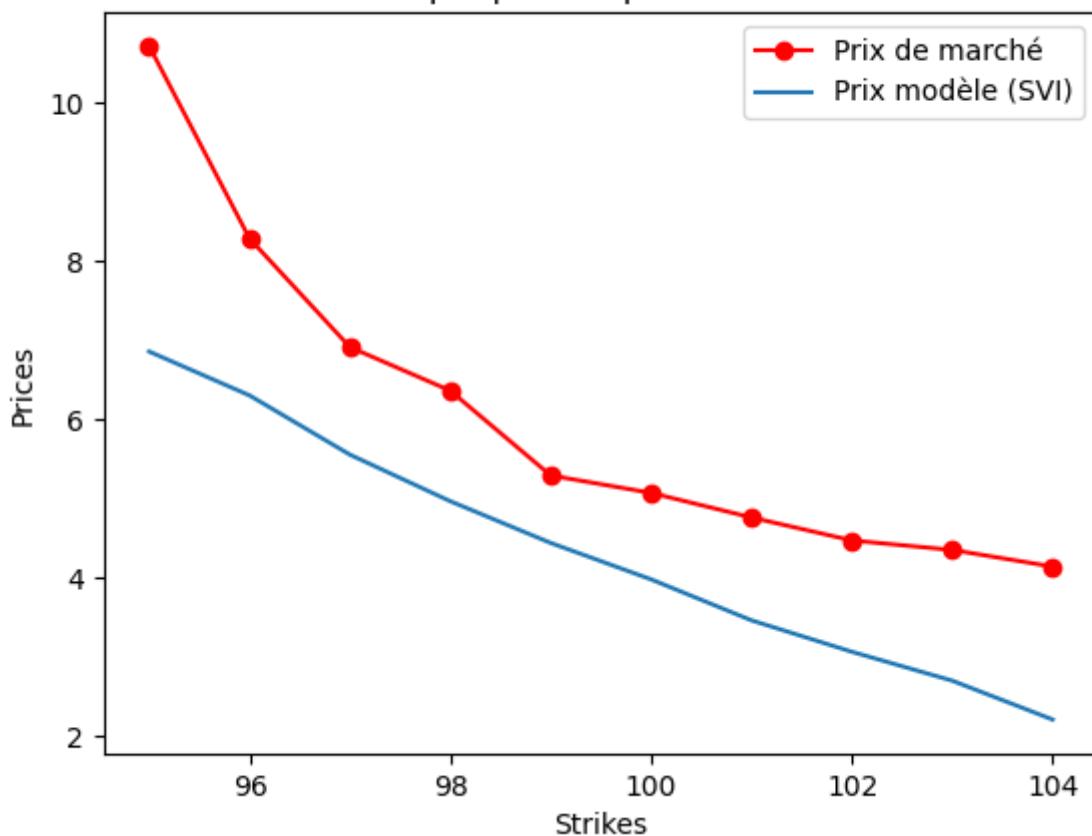
Graphique des prix obtenus



```
In [ ]: check_results_svi(result_svi_6m_recuit['x'],df_6_months)
```

```
Prix par simulation de Monte Carlo pour K = 95: 6.85
Market Price = 10.71, Difference = 3.86
Prix par simulation de Monte Carlo pour K = 96: 6.30
Market Price = 8.28, Difference = 1.98
Prix par simulation de Monte Carlo pour K = 97: 5.55
Market Price = 6.91, Difference = 1.36
Prix par simulation de Monte Carlo pour K = 98: 4.96
Market Price = 6.36, Difference = 1.40
Prix par simulation de Monte Carlo pour K = 99: 4.44
Market Price = 5.29, Difference = 0.85
Prix par simulation de Monte Carlo pour K = 100: 3.98
Market Price = 5.07, Difference = 1.09
Prix par simulation de Monte Carlo pour K = 101: 3.46
Market Price = 4.76, Difference = 1.30
Prix par simulation de Monte Carlo pour K = 102: 3.06
Market Price = 4.47, Difference = 1.41
Prix par simulation de Monte Carlo pour K = 103: 2.70
Market Price = 4.35, Difference = 1.65
Prix par simulation de Monte Carlo pour K = 104: 2.21
Market Price = 4.14, Difference = 1.93
16.814258344759573
```

Graphique des prix obtenus



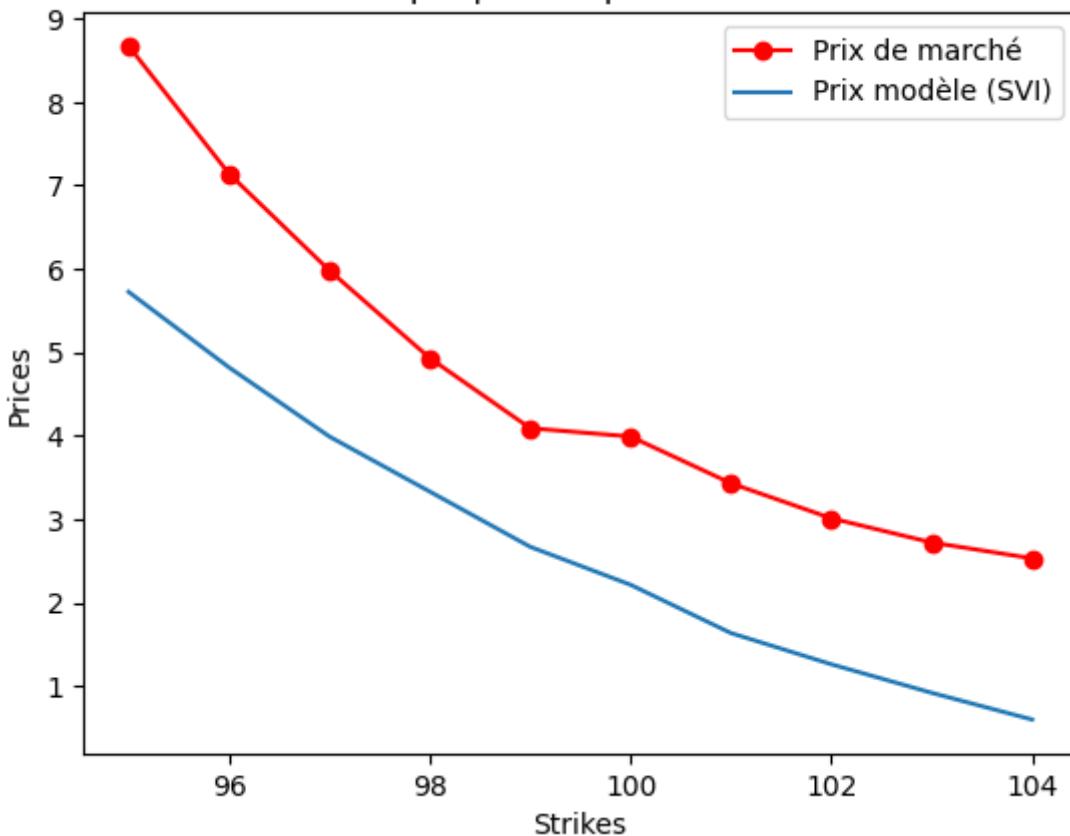
```
In [ ]: check_results_svi(result_svi_3m_recuit['x'], df_3_months)
```

```

Prix par simulation de Monte Carlo pour K = 95: 5.72
Market Price = 8.67, Difference = 2.95
Prix par simulation de Monte Carlo pour K = 96: 4.82
Market Price = 7.14, Difference = 2.32
Prix par simulation de Monte Carlo pour K = 97: 3.99
Market Price = 5.98, Difference = 1.99
Prix par simulation de Monte Carlo pour K = 98: 3.33
Market Price = 4.93, Difference = 1.60
Prix par simulation de Monte Carlo pour K = 99: 2.67
Market Price = 4.09, Difference = 1.42
Prix par simulation de Monte Carlo pour K = 100: 2.21
Market Price = 3.99, Difference = 1.78
Prix par simulation de Monte Carlo pour K = 101: 1.63
Market Price = 3.43, Difference = 1.80
Prix par simulation de Monte Carlo pour K = 102: 1.26
Market Price = 3.01, Difference = 1.75
Prix par simulation de Monte Carlo pour K = 103: 0.92
Market Price = 2.72, Difference = 1.80
Prix par simulation de Monte Carlo pour K = 104: 0.60
Market Price = 2.53, Difference = 1.93
19.334558625417735

```

Graphique des prix obtenus



```
In [ ]: result_svi_6m = nelder_mead(cost_function_SVI,np.array([0.1,0.1,0.1,0.1,0.1]),df_6_
result_svi_3m = nelder_mead(cost_function_SVI,np.array([0.1,0.1,0.1,0.1,0.1]),df_3_
```

```
In [ ]: result_svi_9m_recuit = recuit_simule(cost_function_SVI,np.array([0.1,0.1,0.1,0.1,0.
```

```
In [ ]: def PSO(cost_function, problem_bounds, num_particles, w, phi_p, phi_g, max_iter,*args):
    bounds = np.array(problem_bounds)
    global_best_position = None
    global_best_value = float('inf')

    swarm = []
    for i in range(num_particles):
        position = np.random.uniform(low=bounds[:, 0], high=bounds[:, 1], size=bour
```

```

        velocity = np.random.uniform(low=-1, high=1, size=bounds.shape[0])
        best_position = np.copy(position)
        best_value = float('inf')
        swarm.append([position, velocity, best_position, best_value])

    for iteration in range(max_iter):
        for particle in swarm:
            value = cost_function(particle[0], args)
            if value < particle[3]:
                particle[3] = value
                particle[2] = np.copy(particle[0])
            if value < global_best_value:
                global_best_value = value
                global_best_position = np.copy(particle[0])

        for particle in swarm:
            r_p = np.random.uniform(0, phi_p, size=particle[1].shape)
            r_g = np.random.uniform(0, phi_g, size=particle[1].shape)
            cognitive_component = r_p * (particle[2] - particle[0])
            social_component = r_g * (global_best_position - particle[0])
            particle[1] = w * particle[1] + cognitive_component + social_component

            particle[0] += particle[1]
            particle[0] = np.clip(particle[0], bounds[:,0], bounds[:,1])

    result = {'x': global_best_position, 'fun': global_best_value}
    return result

```

Test PSO pour le modèle SVI

```

In [ ]: # Define the bounds for the problem, for example: [[-10, 10], [-10, 10]] for a 2-di
problem_bounds = np.array([[-1, 1], [-1, 1], [-1, 1], [-1, 1], [-1, 1]])

num_particles=30
w=0.7
phi_p=2.05
phi_g=2.05
max_iter=1000

# Initialize PSO with the objective function and bounds
result_12m_PSO_SVI = PSO(cost_function_SVI, problem_bounds, num_particles, w, phi_p,
result_9m_PSO_SVI = PSO(cost_function_SVI, problem_bounds, num_particles, w, phi_p,
result_6m_PSO_SVI = PSO(cost_function_SVI, problem_bounds, num_particles, w, phi_p,
result_3m_PSO_SVI = PSO(cost_function_SVI, problem_bounds, num_particles, w, phi_p,

print(result_12m_PSO_SVI)
print(result_9m_PSO_SVI)
print(result_6m_PSO_SVI)
print(result_3m_PSO_SVI)

{'x': array([-3.29686635e-02, -1.00000000e+00, -1.00000000e+00, -1.00000000e+00,
-1.04455974e-07]), 'fun': 0.002969276709495695}
{'x': array([-5.09580495e-16, 1.00000000e+00, -1.00000000e+00, -1.00000000e+00,
-2.38946704e-01]), 'fun': 0.005438513313005945}
{'x': array([-5.77835797e-04, -7.76701960e-01, -6.42801631e-01, -2.80845685e-02,
-2.25623055e-02]), 'fun': 0.0004477113126709815}
{'x': array([ 6.26676024e-10, -1.00000000e+00, 8.38733238e-01, 4.48165686e-02,
-1.54850644e-02]), 'fun': 0.000980963000544994}

```

```
In [ ]: param_PSO_SVI_concat = [result_12m_PSO_SVI['x'], result_9m_PSO_SVI['x'], result_6m_PSO_SVI['x'], result_3m_PSO_SVI['x']]
```

```
PSO_SVI_param=pd.DataFrame(param_PSO_SVI_concat,index=[ "12 months","9 months","6 months"])
PSO_SVI_param
```

	<b>a</b>	<b>b</b>	<b>rho</b>	<b>m</b>	<b>sigma</b>
<b>12 months</b>	-3.296866e-02	-1.000000	-1.000000	-1.000000	-1.044560e-07
<b>9 months</b>	-5.095805e-16	1.000000	-1.000000	-1.000000	-2.389467e-01
<b>6 months</b>	-5.778358e-04	-0.776702	-0.642802	-0.028085	-2.256231e-02
<b>3 months</b>	6.266760e-10	-1.000000	0.838733	0.044817	-1.548506e-02

```
In [ ]: check_results_svi(result_12m_PSO_SVI[ 'x' ],df_12_months)
```

```
In [ ]: check_results_svi(result_9m_PSO_SVI[ 'x' ],df_9_months)
```

```
In [ ]: check_results_svi(result_6m_PSO_SVI[ 'x' ],df_6_months)
```

```
In [ ]: check_results_svi(result_3m_PSO_SVI[ 'x' ],df_3_months)
```

```
In [ ]: problem_bounds = np.array([[0, 2], [-2, 2],[-2, 2], [-2, 2],[-2, 2]])

num_particles=30
w=0.7
phi_p=0.5
phi_g=0.5
max_iter=1000
```

Test PSO pour PDV

```
In [ ]: result_12m_PDV = PSO(cost_function_PDV, problem_bounds,num_particles, w, phi_p, phi_g)
result_9m_PDV = PSO(cost_function_PDV, problem_bounds,num_particles, w, phi_p, phi_g)
result_6m_PDV = PSO(cost_function_PDV, problem_bounds,num_particles, w, phi_p, phi_g)
result_3m_PDV = PSO(cost_function_PDV, problem_bounds,num_particles, w, phi_p, phi_g)

print(result_12m_PDV)
print(result_9m_PDV)
print(result_6m_PDV)
print(result_3m_PDV)
```

```
In [ ]: check_result_PDV(result_12m_PDV,df_12_months,1)
```