Projet Calibration A5

BRAUN Arthur, CABAL Paul-Louis

January 22, 2024

Contents

Partie I - Densités risque neutre

Question 1 - Densité risque neutre et comparaison

Dans cette première partie, l'objectif est de Calibrer une densité risque neutre sur les données comprenant les prix d'exercices à des strikes différents avec la formule de Breeden Litzenberg et la Méthode de David Shimko. Ensuite, il nous faudra comparer à une densité Gaussienne.

Voici les libraires que nous utiliserons :

```
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
from matplotlib import cm
from matplotlib.ticker import LinearLocator
import pandas as pd
import random
```

Mise en forme des données

Pour faciliter l'utilisation des données, nous utilisons un dataframe pandas appelé "prices1" :

	Strike1	Prix1	Maturité1
0	95.0	10.93	1.0
1	96.0	9.55	1.0
2	97.0	8.28	1.0
3	98.0	7.40	1.0
4	99.0	6.86	1.0
5	100.0	6.58	1.0
6	101.0	6.52	1.0
7	102.0	6.49	1.0

Implémentation Breeden Letzenberg

La formule de Breeden-Litzenberger que nous allons implémenter est une méthode qui va nous aider à extraire la densité de probabilité implicite des prix d'options.
Elle repose sur le principe que la seconde dérivée du prix d'une option par rapport au prix d'exercice est équivalente à la densité de probabilité de l'actif sous-jacent à l'échéance de l'option.

Voici la formule que nous implémentons :

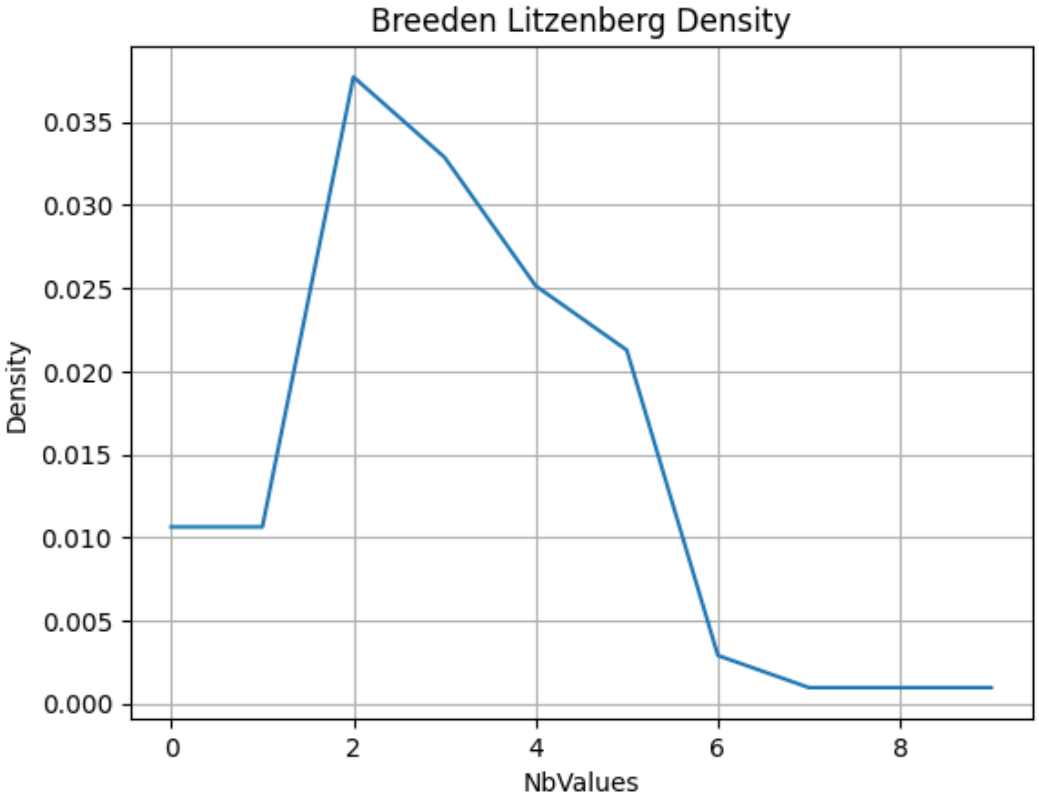
$$\frac{C(K+\Delta K,T)-2C(K,T)+C(K-\Delta K,T)}{(\Delta K)^2}$$

Voici le code Python :

```
def breeden_litzenberger_density(prices, strikes, r, T,t):
    densities = []
    for i in range(1, len(strikes) - 1):
        second_derivative = (prices[i-1] - 2 * prices[i] + prices[i+1])
        strike_diff = strikes[i+1] - strikes[i-1]
        density = np.exp(r * T-t[i]) * second_derivative / (strike_diff ** 2)
        densities.append(density)

    first_density = densities[0]
    last_density = densities[-1]
    densities.insert(0, first_density)
    densities.append(last_density)
    return densities
```

Dans l'implémentation, nous avons rajouté les premières et dernières valeurs de manière à ce que le nombre de valeurs soit cohérent avec les données fournies.

Voici la courbe de densité que nous obtenons en appliquant la formule de Breeden Litzenberg :



Méthode de David Shimko

La méthode de Shimko consiste à estimer la densité de probabilité en recalculant les prix des options à partir de la volatilité implicite interpolée.

Dans cette partie, nous nous sommes appuyés sur l'article de David Shimko appelé Bounds of probability pour comprendre la méthode.

Pour mettre en place la méthode, voici les étapes que nous devons faire :

- Calcul de la volatilité implicite
- Interpolation volatilité implicite
- Calcul des prix avec la volatilité interpolée

I - Première étape : Calcul de la volatilité implicite

Pour cela nous avons besoin de définir les différentes fonctions pour calculer les prix selon le modèle de Black-Scholes et retrouver la volatilité implicite à partir de nos données.

```
def norm_pdf(x):
    return (1.0 / np.sqrt(2 * np.pi)) * np.exp(-0.5 * x * x)
```

La fonction norm_pdf(x) est la fonction de densité de probabilité de la distribution normale standard :

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

```
def black_scholes_vega(S0, K, T, r, sigma):
    d1 = (np.log(S0 / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    return S0 * np.sqrt(T) * norm_pdf(d1)
```

La fonction black_scholes_vega(...) calcule le Véga de l'option Black-Scholes, qui mesure la sensibilité du prix de l'option aux changements de volatilité :

$$\text{Vega}(S_0, K, T, r, \sigma) = S_0\sqrt{T} * N(d1)$$

$$d_1 = \frac{\ln\left(\frac{S_0}{K}\right) + \left(r + \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}}$$

```
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
```

La formule de Black Scholes :

$$C(S, K, T, r, \sigma) = SN(d_1) - Ke^{-rT}N(d_2)$$

Avec :

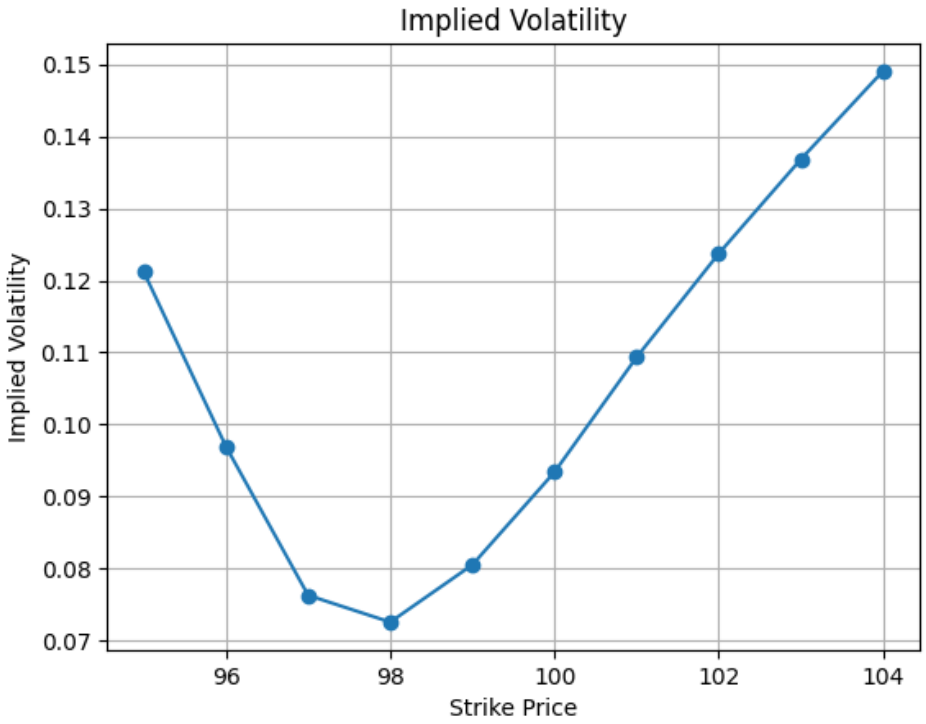
$$d_2 = d_1 - \sigma\sqrt{T}$$

```
def implied_volatility_newton_raphson(S0, K, T, r, market_price, sigma_initial_guess=0.2,
max_iterations=100, epsilon=1e-6):
    sigma = sigma_initial_guess
    for i in range(max_iterations):
        price = black_scholes_call_price(S0, K, T, r, sigma)
        vega = black_scholes_vega(S0, K, T, r, sigma)

        sigma = sigma - (price - market_price) / vega
        if abs(price - market_price) < epsilon:
            return sigma

    return sigma
```

La fonction implied_volatility_newton_raphson(...) utilise la méthode de Newton-Raphson pour trouver la volatilité implicite dans le modèle Black-Scholes :

$$\sigma_{\text{new}} = \sigma_{\text{old}} - \frac{C(S_0, K, T, r, \sigma_{\text{old}}) - \text{Prix du marché}}{\text{Véga}(S_0, K, T, r, \sigma_{\text{old}})}$$

À l'aide de ces fonctions, nous calculons la Volatilité implicite et la mettons dans notre dataframe pour faciliter l'utilisation plus tard, voici ce que nous obtenons lorsque nous traçons la vol Imlicite sur le graphique :



II - Interpolation volatilité implicite

Maintenant, il nous faut interpoler la volatilité implicite. Pour cela, nous avons testé avec deux méthodes :

- Interpolation quadratique
- interpolation par moindres carrés

II-1- Interpolation quadratique :

Nous effectuons une interpolation quadratique de la courbe de volatilité implicite en utilisant trois points de données de cette courbe. Le but est d'utiliser un polynôme de second degré pour estimer les valeurs entre les points connus.

```
data_points = [
    (95.0, 0.121141),
    (96.0, 0.096921),
    (97.0, 0.076285),
    (98.0, 0.072562),
    (99.0, 0.080513),
    (100.0, 0.093383),
    (101.0, 0.109328),
    (102.0, 0.123711),
    (103.0, 0.136804),
    (104.0, 0.149040)
]
```

```
X1, Y1 = data_points[0]
X2, Y2 = data_points[4]
X3, Y3 = data_points[9]

A = [
    [1, X1, X1**2],
    [1, X2, X2**2],
    [1, X3, X3**2]
]
B = [Y1, Y2, Y3]


def determinant(matrix):
    return (matrix[0][0] * (matrix[1][1] * matrix[2][2] - matrix[1][2] * matrix[2][1]) -
            matrix[0][1] * (matrix[1][0] * matrix[2][2] - matrix[1][2] * matrix[2][0]) +
            matrix[0][2] * (matrix[1][0] * matrix[2][1] - matrix[1][1] * matrix[2][0]))

# Cramer
def solve_quadratic_system(A, B):
    det_A = determinant(A)
    if det_A == 0:
        raise ValueError("Pas de solution unique")

    A0_matrix = [[B[0], A[0][1], A[0][2]], [B[1], A[1][1], A[1][2]], [B[2], A[2][1], A[2][2]]]
    A1_matrix = [[A[0][0], B[0], A[0][2]], [A[1][0], B[1], A[1][2]], [A[2][0], B[2], A[2][2]]]
    A2_matrix = [[A[0][0], A[0][1], B[0]], [A[1][0], A[1][1], B[1]], [A[2][0], A[2][1], B[2]]]

    A0 = determinant(A0_matrix) / det_A
    A1 = determinant(A1_matrix) / det_A
    A2 = determinant(A2_matrix) / det_A

    return A0, A1, A2
A0, A1, A2 = solve_quadratic_system(A, B)
print(f"Coeffs: A0={A0}, A1={A1}, A2={A2}")
```

Explication du code :

data_points : Une liste de tuples où chaque tuple contient deux éléments : le prix d'exercice d'une option (X) et sa volatilité implicite (Y). Ces points sont extraits de la courbe de volatilité implicite observée sur le marché.

Les variables X1, Y1, X2, Y2, X3, Y3 : Ces variables stockent les coordonnées des trois points choisis pour l'interpolation. Ces points sont représentatifs de la courbe et permettent de construire le polynôme interpolateur.

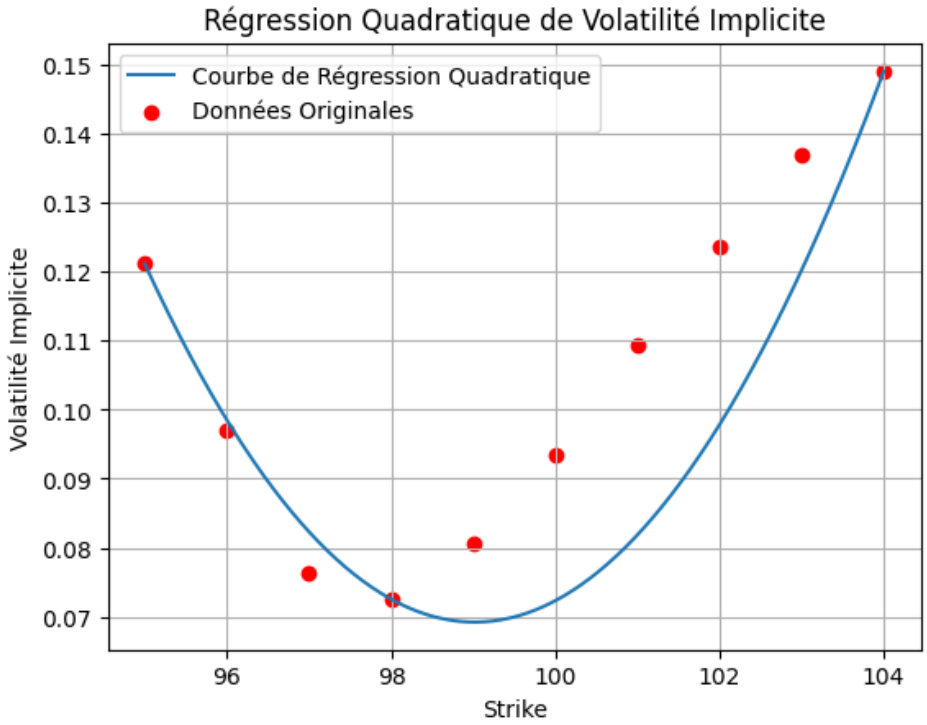
Matrice A : Une matrice 3x3 où chaque ligne correspond à l'expression polynomiale 1,X,X**2 évaluée pour chaque prix d'exercice des points sélectionnés. Cette matrice représente le système linéaire à résoudre pour trouver les coefficients du polynôme d'interpolation

Vecteur B : Un vecteur contenant les valeurs de volatilité implicite correspondant à chaque point sélectionné.

Fonction determinant(matrix) : Calcule le déterminant d'une matrice 3x3, pour appliquer la règle de

Cramer dans la résolution du système linéaire.

Fonction solve_quadratic_system(A, B) : Résout le système linéaire en utilisant la règle de Cramer pour trouver les coefficients, A0, A1, A2, du polynôme quadratique A0 + A1X + A2X**2



Par la suite nous voulions voir lors de la comparaison la différence entre les méthode d'interpolation sachant que cette méthode d'interpolation quadratiques fait passé un polynome par trois points obligatoires de nos données.

Nous allons aussi faire une interpolation méthode des moindres carrés

II-2- Interpolation moindres carrés :

Pour cette méthode, nous avons refait un code différent pour pouvoir s'adapter le plus facilement a la méthode des moindres carrés

```
def solve_linear_system(A, b):
n = len(A)
for i in range(n):
    # Normalisation de la ligne pivot
    pivot = A[i][i]
    for j in range(i, n):
        A[i][j] /= pivot
    b[i] /= pivot
    for k in range(i + 1, n):
        factor = A[k][i]
        for j in range(i, n):
            A[k][j] -= factor * A[i][j]
        b[k] -= factor * b[i]
x = [0 for _ in range(n)]
for i in range(n - 1, -1, -1):
    x[i] = b[i] - sum(A[i][j] * x[j] for j in range(i + 1, n))
```

```
    return x

    def least_squares_fit(x, y, degree=2):
    # Création de la matrice de Vandermonde
    A = [[xi**n for n in range(degree + 1)] for xi in x]
    # Transposition de A
    At = [list(i) for i in zip(*A)]
    AtA = [[sum(At[i][k] * A[k][j] for k in range(len(x))) for j in range(degree + 1)]
    for i in range(degree + 1)]
    Aty = [sum(y[k] * At[i][k] for k in range(len(x))) for i in range(degree + 1)]
    coef = solve_linear_system(AtA, Aty)
    return coef

x = prices1['Strike1']
y = prices1['VolImplicite']
coefs = least_squares_fit(x, y, 2)
print("Coefficients du polynôme :", coefs)
```

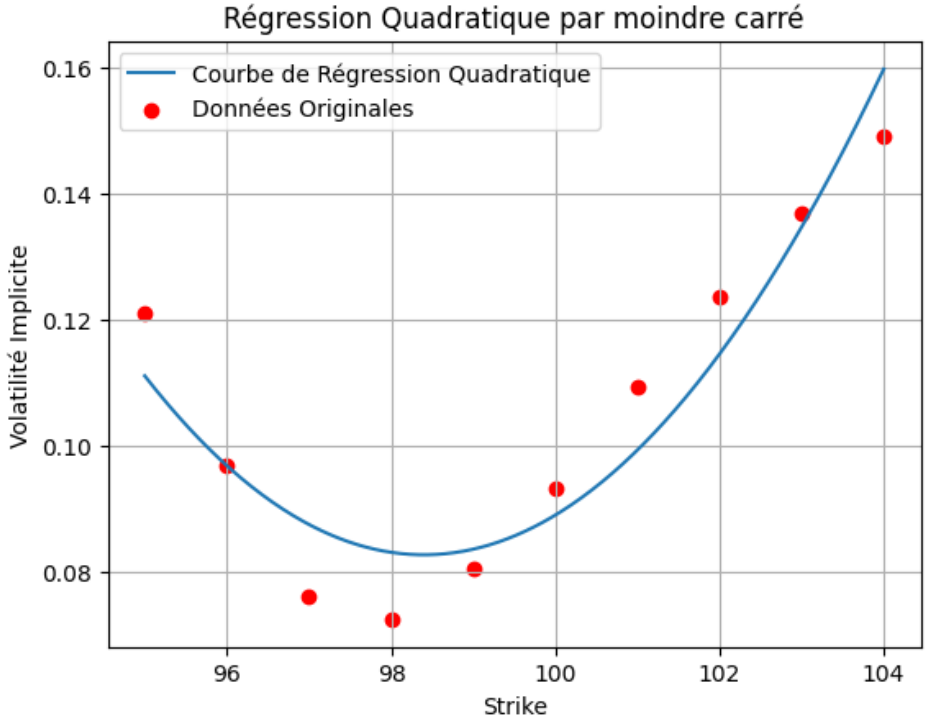
least_squares_fit(x, y, degree), prend en entrée des listes de valeurs x (prix d'exercice) et y (vol implicites). Elle effectue une interpolation polynomiale de degré 2 sur les données.
Matrice A: La matrice de Vandermonde est créée à partir des valeurs de x. Chaque ligne de A est une liste des puissances de xi de 0 à degree.

Transposition At: La matrice A est transposée pour le produit matriciel.

Produit matriciel AtA: Le produit des matrices transposées de A et A est calculé.

Produit matriciel Aty: Le produit de la matrice transposée de A et du vecteur y est calculé. Ce vecteur est le côté droit du système linéaire.

solve_linear_system(A, b): Cette fonction résout le système linéaire formé par la matrice normale AtA et le vecteur Aty (pivot de Gauss).

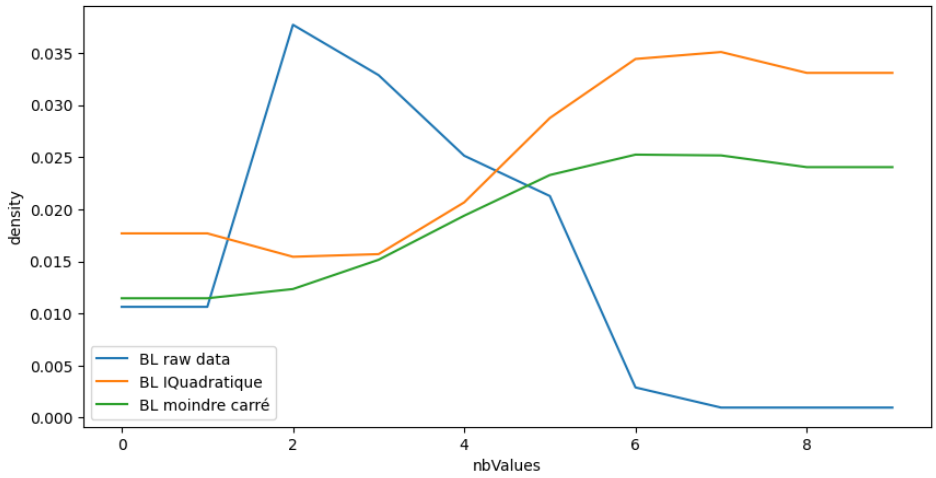


III- Calcul des prix avec la volatilité interpolé

Maintenant que nous avons obtenu les polynômes d'interpolation quadratique, nous calculons les prix avec les valeurs de volatilité interpolées avec la formule de Black-Scholes :

	Strike1	Prix1	Maturité1	VolImplicite	VolInterpolMC	Price_VolInterpolMC	VolInterpol	Price_VolInterpol
0	95.0	10.93	1.0	0.121141	0.111103	10.670362	0.121141	10.930007
1	96.0	9.55	1.0	0.096921	0.096885	9.549116	0.098517	9.589450
2	97.0	8.28	1.0	0.076285	0.087572	8.546387	0.082324	8.418135
3	98.0	7.40	1.0	0.072562	0.083162	7.671338	0.072562	7.400007
4	99.0	6.86	1.0	0.080513	0.083655	6.953010	0.069231	6.544037
5	100.0	6.58	1.0	0.093383	0.089053	6.435187	0.072331	5.899612
6	101.0	6.52	1.0	0.109328	0.099354	6.158222	0.081862	5.539049
7	102.0	6.49	1.0	0.123711	0.114559	6.142343	0.097823	5.511816
8	103.0	6.47	1.0	0.136804	0.134667	6.386768	0.120216	5.824581
9	104.0	6.46	1.0	0.149040	0.159679	6.879871	0.149040	6.459981

et appliquons la formule de Breeden Litzenberg sur les prix calculés à partir de la volatilité interpolée.



On peut voir visuellement qu'il y a une différence entre plusieurs types d'interpolation, nous avons implémenté les formules du papier de David Shimko en suivant les étapes décrites.
Nous allons maintenant lisser et calculer les prix interpolés pour 1000 strikes compris entre 90 et 120 pour capter les changements de densité pour des valeurs de strike plus élevés.

```
def Plus_Strike(start, stop, num):
    if num == 1:
        return [start]
    step = (stop - start) / (num - 1)
    return [start + step*i for i in range(num)]
```

Maintenant on passe au calcul des prix interpolé avec la volatilité interpolé avec les coefficients quadratique et par la méthode des moindres carrés.

- Quadratique

```
strikes = Plus_Strike(90, 120, 1000)
results = pd.DataFrame(strikes, columns=['Strike'])
```

```
results['Volatility_Inter_Quad'] = A0 + A1 * results['Strike'] + A2 * results['Strike']**2
results['Price_Quad'] = results.apply(lambda row: black_scholes_call_price(S0,row['Strike'], T, r
results['maturité']=1

results['Density_BL_Quad'] = BL_density_Shimko_Quad =

breeden_litzenberger_density(results['Price_Quad'],strikes,r,T,results['maturité'])

total_density_B = sum(BL_density_Shimko_Quad)
BL_normalized_density_B = [d / total_density_B for d in BL_density_Shimko_Quad]
BL_normalized_density_B
results['Density_BL_Quad_norm'] = BL_normalized_density_B
results
```

Nous calculons les valeurs normalisées des densités

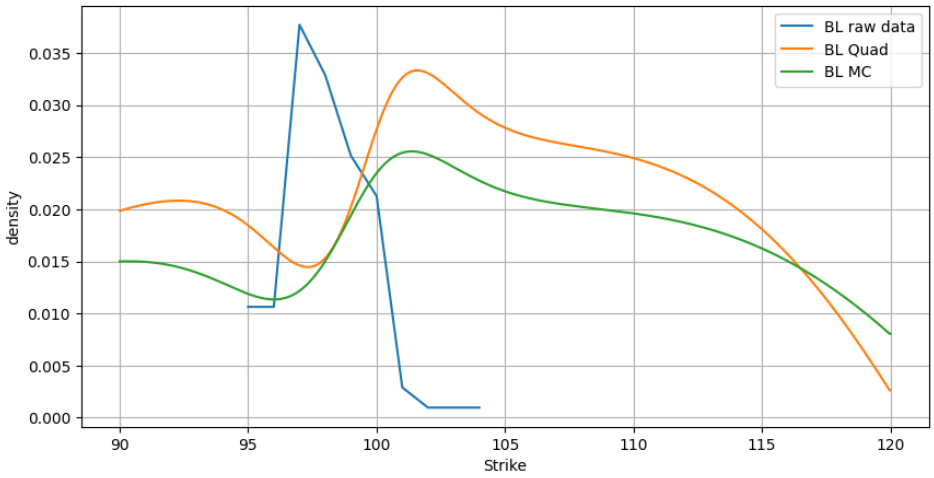
- Moindres carrées

```
A0b=23.822681830484413
A1b=-0.4825218564257689
A2b= 0.0024518556510651004
```

```
results['Volatility_Inter_MC'] = A0b + A1b * results['Strike'] + A2b * results['Strike']**2
results['Price_MC'] = results.apply(lambda row: black_scholes_call_price(S0, row['Strike'], T, r,

results['Density_BL_MC'] = BL_density_Shimko_MC = breeden_litzenberger_density(results['Price_MC'

total_density_MC = sum(BL_density_Shimko_MC)
BL_normalized_density_MC = [d / total_density_MC for d in BL_density_Shimko_MC]
BL_normalized_density_MC
results['Density_BL_MC_norm'] = BL_normalized_density_MC
results
```

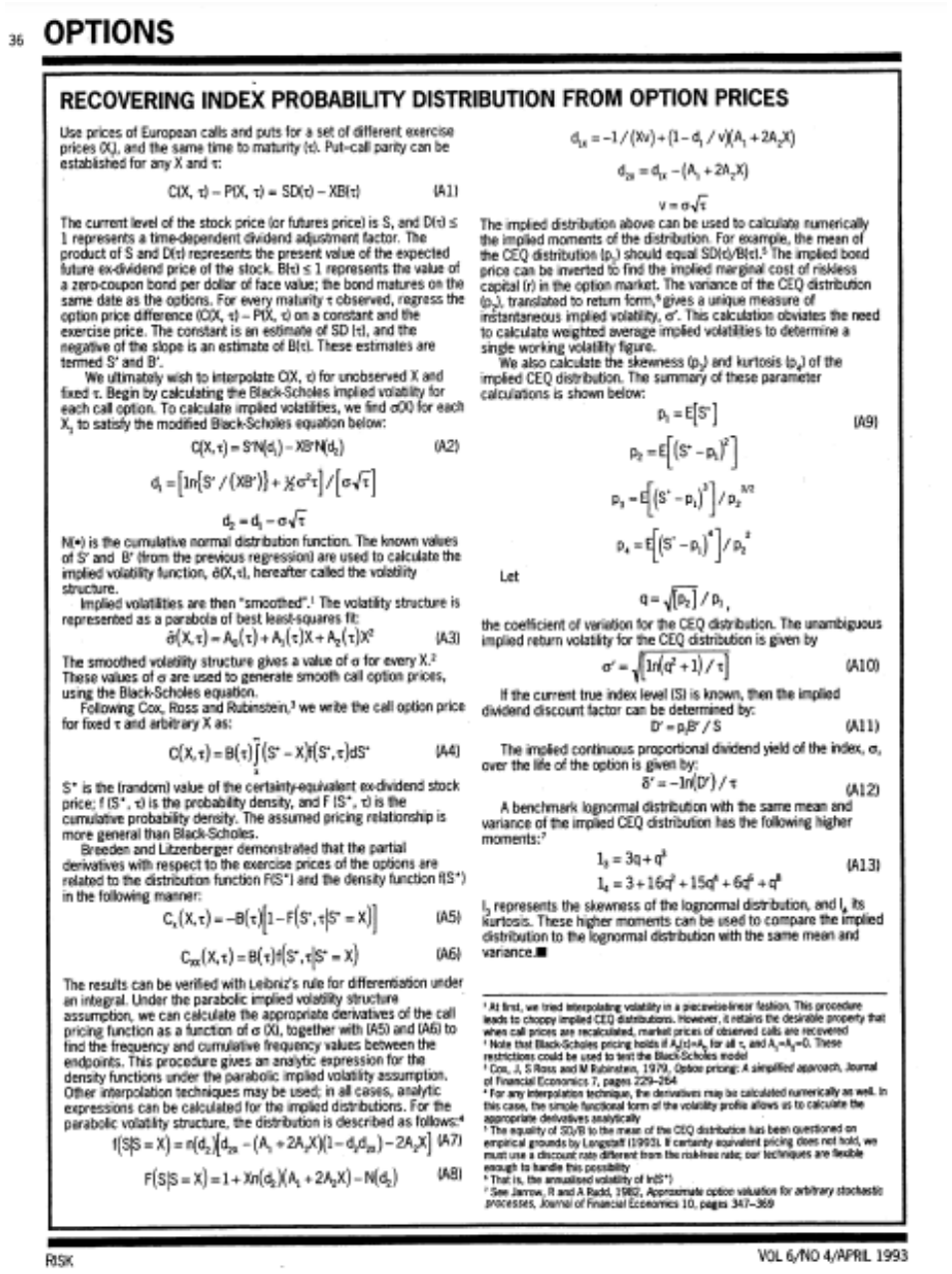
	Strike	Volatility_Inter_Quad	Price_Quad	maturité	Density_BL_Quad	Density_BL_Quad_norm	Volatility_Inter_MC	Price_MC	Density_BL_MC	Density_BL_MC_norm
0	95.000000	0.121141	10.930007	1	0.018437	0.000772	0.111103	10.670362	0.011871	0.000616
1	95.009009	0.120908	10.917110	1	0.018437	0.000772	0.110953	10.659724	0.011871	0.000616
2	95.018018	0.120676	10.904228	1	0.018420	0.000772	0.110803	10.649096	0.011863	0.000616
3	95.027027	0.120445	10.891361	1	0.018403	0.000771	0.110654	10.638478	0.011855	0.000615
4	95.036036	0.120214	10.878510	1	0.018385	0.000770	0.110505	10.627869	0.011847	0.000615
...	...	...	...	...	...	...	...	...	...	...
995	103.963964	0.147890	6.431720	1	0.029277	0.001226	0.158693	6.857937	0.022789	0.001182
996	103.972973	0.148176	6.438748	1	0.029261	0.001226	0.158939	6.863392	0.022779	0.001182
997	103.981982	0.148464	6.445801	1	0.029245	0.001225	0.159185	6.868866	0.022768	0.001181
998	103.990991	0.148752	6.452879	1	0.029230	0.001224	0.159432	6.874359	0.022757	0.001181
999	104.000000	0.149040	6.459981	1	0.029230	0.001224	0.159679	6.879871	0.022757	0.001181

Voici l'organisation des données que nous obtenons pour comparer plus facilement par la suite
Voici les densités obtenues par BL avec deux méthodes d'interpolation différentes en suivant les étapes par la méthode de Shimko



On remarque qu'avec la méthode de Shimko la dynamique liée à la volatilité implicite est retranscrite dans la densité par rapport à la méthode de Breeden Litzenberg sur les datas ne contenant pas l'information de la dynamique smile de volatilité.

Interpretation paramétrique des densités obtenues avec Breeden Litzenberg
Maintenant en reprenant le papier de David Shimko nous avons voulu implémenter les formules pour bien les comprendre.



36 OPTIONS

RECOVERING INDEX PROBABILITY DISTRIBUTION FROM OPTION PRICES

Use prices of European calls and puts for a set of different exercise prices (X), and the same time to maturity (τ). Put-call parity can be established for any X and τ:

$$C(X,\tau) - P(X,\tau) = SD(\tau) - XB(\tau) \quad (A1)$$

The current level of the stock price (or futures price) is S, and $D(\tau) \le 1$ represents a time-dependent dividend adjustment factor. The product of S and $D(\tau)$ represents the present value of the expected future ex-dividend price of the stock. $B(\tau) \le 1$ represents the value of a zero-coupon bond per dollar of face value; the bond matures on the same date as the options. For every maturity τ observed, regress the option price difference ($C(X,\tau) - P(X,\tau)$) on a constant and the exercise price. The constant is an estimate of $SD(\tau)$, and the negative of the slope is an estimate of $B(\tau)$. These estimates are termed S' and B'.

We ultimately wish to interpolate $C(X,\tau)$ for unobserved X and fixed τ. Begin by calculating the Black-Scholes implied volatility for each call option. To calculate implied volatilities, we find $\sigma(X)$ for each X_i to satisfy the modified Black-Scholes equation below:

$$C(X,\tau) = S'N(d_1) - XB'N(d_2) \quad (A2)$$

$$d_1 = \left[\ln\left\{S'/(XB')\right\} + \tfrac{1}{2}\sigma^2\tau\right] / \left[\sigma\sqrt{\tau}\right]$$

$$d_2 = d_1 - \sigma\sqrt{\tau}$$

$N(\cdot)$ is the cumulative normal distribution function. The known values of S' and B' (from the previous regression) are used to calculate the implied volatility function, $\hat{\sigma}(X,\tau)$, hereafter called the volatility structure.

Implied volatilities are then "smoothed".[1] The volatility structure is represented as a parabola of best least-squares fit:

$$\hat{\sigma}(X,\tau) = A_0(\tau) + A_1(\tau)X + A_2(\tau)X^2 \quad (A3)$$

The smoothed volatility structure gives a value of σ for every X.[2] These values of σ are used to generate smooth call option prices, using the Black-Scholes equation.

Following Cox, Ross and Rubinstein,[3] we write the call option price for fixed τ and arbitrary X as:

$$C(X,\tau) = B(\tau)\int_X^\infty (S^* - X) f(S^*,\tau)\,dS^* \quad (A4)$$

S^* is the (random) value of the certainty-equivalent ex-dividend stock price; $f(S^*,\tau)$ is the probability density, and $F(S^*,\tau)$ is the cumulative probability density. The assumed pricing relationship is more general than Black-Scholes.

Breeden and Litzenberger demonstrated that the partial derivatives with respect to the exercise prices of the options are related to the distribution function $F(S^*)$ and the density function $f(S^*)$ in the following manner:

$$C_X(X,\tau) = -B(\tau)\left[1 - F(S^*,\tau \mid S^* = X)\right] \quad (A5)$$

$$C_{XX}(X,\tau) = B(\tau) f(S^*,\tau \mid S^* = X) \quad (A6)$$

The results can be verified with Leibniz's rule for differentiation under an integral. Under the parabolic implied volatility structure assumption, we can calculate the appropriate derivatives of the call pricing function as a function of $\sigma(X)$, together with (A5) and (A6) to find the frequency and cumulative frequency values between the endpoints. This procedure gives an analytic expression for the density functions under the parabolic implied volatility assumption. Other interpolation techniques may be used; in all cases, analytic expressions can be calculated for the implied distributions. For the parabolic volatility structure, the distribution is described as follows:[4]

$$f(S \mid S = X) = n(d_2)\left[d_{2X} - (A_1 + 2A_2X)(1 - d_2 d_{2X}) - 2A_2X\right] \quad (A7)$$

$$F(S \mid S = X) = 1 + Xn(d_2)(A_1 + 2A_2X) - N(d_2) \quad (A8)$$

$$d_{1X} = -1/(Xv) + (1 - d_1/v)(A_1 + 2A_2X)$$

$$d_{2X} = d_{1X} - (A_1 + 2A_2X)$$

$$v = \sigma\sqrt{\tau}$$

The implied distribution above can be used to calculate numerically the implied moments of the distribution. For example, the mean of the CEQ distribution (p_1) should equal $SD(\tau)/B(\tau)$.[5] The implied bond price can be inverted to find the implied marginal cost of riskless capital (r) in the option market. The variance of the CEQ distribution (p_2), translated to return form,[6] gives a unique measure of instantaneous implied volatility, σ'. This calculation obviates the need to calculate weighted average implied volatilities to determine a single working volatility figure.

We also calculate the skewness (p_3) and kurtosis (p_4) of the implied CEQ distribution. The summary of these parameter calculations is shown below:

$$p_1 = E\left[S^*\right] \quad (A9)$$

$$p_2 = E\left[\left(S^* - p_1\right)^2\right]$$

$$p_3 = E\left[\left(S^* - p_1\right)^3\right] / p_2^{3/2}$$

$$p_4 = E\left[\left(S^* - p_1\right)^4\right] / p_2^{2}$$

Let

$$q = \sqrt{\left[p_2\right]} / p_1,$$

the coefficient of variation for the CEQ distribution. The unambiguous implied return volatility for the CEQ distribution is given by

$$\sigma' = \sqrt{\left[\ln\left(q^2 + 1\right)/\tau\right]} \quad (A10)$$

If the current true index level (S) is known, then the implied dividend discount factor can be determined by:

$$D' = p_1 B' / S \quad (A11)$$

The implied continuous proportional dividend yield of the index, σ, over the life of the option is given by:

$$\delta' = -\ln(D')/\tau \quad (A12)$$

A benchmark lognormal distribution with the same mean and variance of the implied CEQ distribution has the following higher moments:[7]

$$l_3 = 3q + q^3 \quad (A13)$$

$$l_4 = 3 + 16q^2 + 15q^4 + 6q^6 + q^8$$

l_3 represents the skewness of the lognormal distribution, and l_4 its kurtosis. These higher moments can be used to compare the implied distribution to the lognormal distribution with the same mean and variance.■

[1] At first, we tried interpolating volatility in a piecewise-linear fashion. This procedure leads to choppy implied CEQ distributions. However, it retains the desirable property that when call prices are recalculated, market prices of observed calls are recovered.

[2] Note that Black-Scholes pricing holds if $A_1(\tau) = A_2$ for all τ, and $A_2 = A_3 = 0$. These restrictions could be used to test the Black-Scholes model.

[3] Cox, J, S Ross and M Rubinstein, 1979, Option pricing: A simplified approach, Journal of Financial Economics 7, pages 229–264.

[4] For any interpolation technique, the derivatives may be calculated numerically as well. In this case, the simple functional form of the volatility profile allows us to calculate the appropriate derivatives analytically.

[5] The equality of SD/B to the mean of the CEQ distribution has been questioned on empirical grounds by Longstaff (1993). If certainty equivalent pricing does not hold, we must use a discount rate different from the risk-free rate; our techniques are flexible enough to handle this possibility.

[6] That is, the annualised volatility of $\ln(S^*)$.

[7] See Jarrow, R and A Rudd, 1982, Approximate option valuation for arbitrary stochastic processes, Journal of Financial Economics 10, pages 347–369.

Source : David Shimko [1]

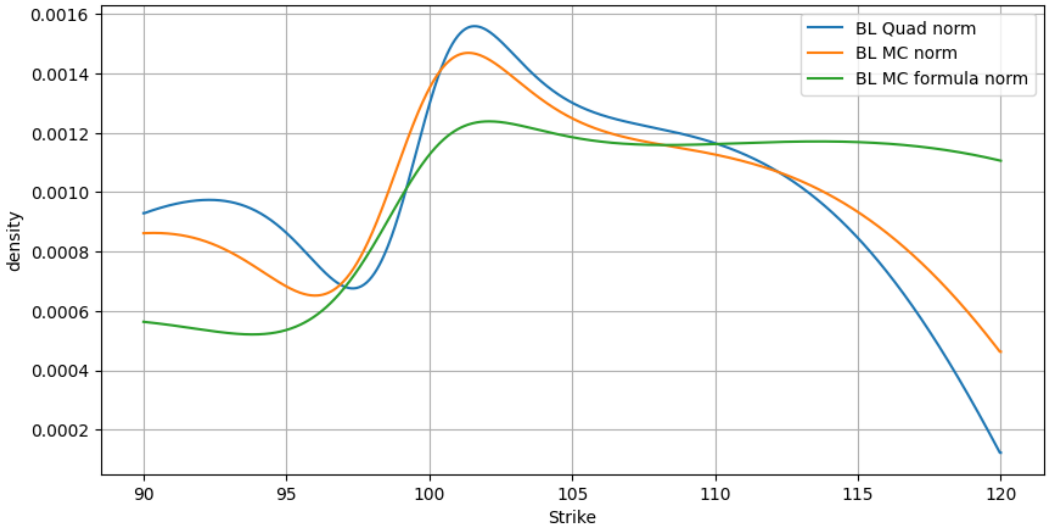
Nous avons implémenté les fonctions pour calculer f(x) la fonction densité selon la volatilité interpolée avec les coefficients A0, A1, A2

```
def calculate_d1(S, K, T, r, sigma):
return (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))

def calculate_d2(d1, sigma, T):
    return d1 - sigma * np.sqrt(T)

def calculate_v(sigma, T):
    return sigma * np.sqrt(T)

def calculate_d1x(v, x,d1,d2,A1,A2):
```

[1] SHIMKO David, 1993, "Bounds of probability", Risk, 6, p.36.

```
    return -1/(v *x ) + (1 - d1/v)*(A1 + 2*A2*x)

def calculate_d2x(d1x, A1, A2, x):
    return d1x - (A1 + 2 * A2 * x)

# Calcul de la densité de probabilité f(S=x), équation (A7)
def density_function(d2,d2x,x,A1,A2):
    term1 = norm_pdf(d2)
    term2 = (d2x - (A1 + 2*A2 * x) * (1-d2 * d2x) - 2 * A2 * x)
    density = term1 * term2
    return density

# Calcul de la fonction de distribution cumulative F(S=x), équation (A8)
def cumulative_distribution_function(d2,x,A1,A2):
    cumulative_distribution = 1 + x * norm_pdf(d2) * (A1 + 2 * A2 * x) - norm_pdf(d2)
    return cumulative_distribution


for index, row in results.iterrows():
    d1 = calculate_d1(S0, row['Strike'], T, r, row['Volatility_Inter_MC'])
    d2 = calculate_d2(d1, row['Volatility_Inter_MC'], T)
    v = calculate_v(row['Volatility_Inter_MC'], T)
    x = row['Strike']
    d1x = calculate_d1x(v, x, d1, d2, A1b, A2b)
    d2x = calculate_d2x(d1x, A1b, A2b, x)
    density = density_function(d2, d2x, x, A1b, A2b)
    results.at[index, 'Density'] = density
    cdf = cumulative_distribution_function(d2, x, A1b, A2b)
    results.at[index, 'Cumulative Distribution'] = cdf
print(results)


Density=results['Density']

total_density_dn = sum(Density)
Density_norm = [d / total_density_dn for d in Density]
results['Density_Norm']=Density_norm
results
```

Et voici le graphique que nous obtenons :



Nous remarquons que la densité obtenue de manière plus théorique avec les formules tirées du papier de D.Shimko est très proche de ce que nous obtenons. C'est une observation qui confirme nos résultats précédent. Mais nous voyons qu'avec les formules théoriques il y a un problème ou niveau des queues de distributions comme annoncé dans le papier de D.Shimko qui propose de modéliser les queues de ditributions avec des fonctions de densités log normales

Gaussienne et comparaison

Pour générer une Gaussienne, nous allons utiliser l'algorithme de Box Muller et allons simuler les tirages avec une moyenne mu la moyenne des strikes et l'écart des strikes pour sigma.
Nous avons essayé de calculer les deux paramètres mu et sigma avec l'hypothèse de risque neutre en les définissant comme ceci :

$$\mu = S_0 e^{rT}$$

$$\sigma = \sigma_{Imp}\sqrt{T}$$

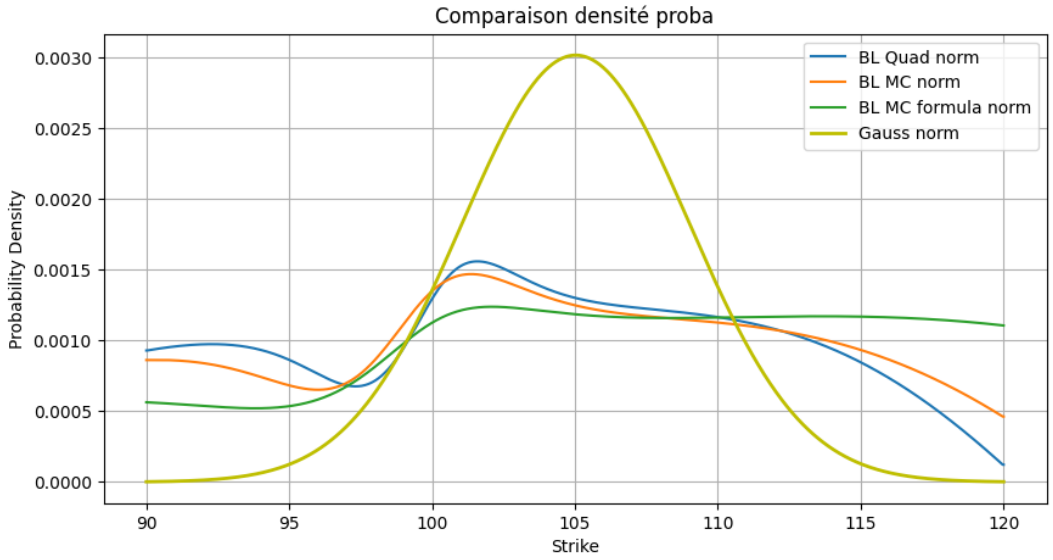
Mais nous avons rencontré un problème au niveau de la loi que nous n'arrivions pas à calibrer pour la comparer correctement.

Voici le code que nous avons utilisé :

```
def box_muller(mu, sigma, num_samples=10000):
samples = []
for _ in range(num_samples // 2):
    u1, u2 = random.random(), random.random()
    z0 = np.sqrt(-2 * np.log(u1)) * np.cos(2 * np.pi * u2)
    z1 = np.sqrt(-2 * np.log(u1)) * np.sin(2 * np.pi * u2)
    samples.append(sigma * z0 + mu)
    samples.append(sigma * z1 + mu)
return samples


sum = 0
count = 0
for strike in prices1['Strike1']:
    sum += strike
    count += 1
```

```
mu = sum / count

sum_squared_diffs = 0
for strike in prices1['Strike1']:
    sum_squared_diffs += (strike - mu) ** 2
sigma = (sum_squared_diffs / count) ** 0.5

samples = box_muller(mu, sigma)
count, bins, ignored = plt.hist(samples, bins=30, density=True, alpha=0.6, color='g')
pdf = (1 / (sigma * np.sqrt(2 * np.pi))) * np.exp(- (bins - mu)**2 / (2 * sigma**2))
```

Nous avons redéfini une la pdf pour y inclure mu et sigma par rapport a celle que nous avons implémentée dès le début. Nous traçons la gaussienne par rapport aux autres courbes :



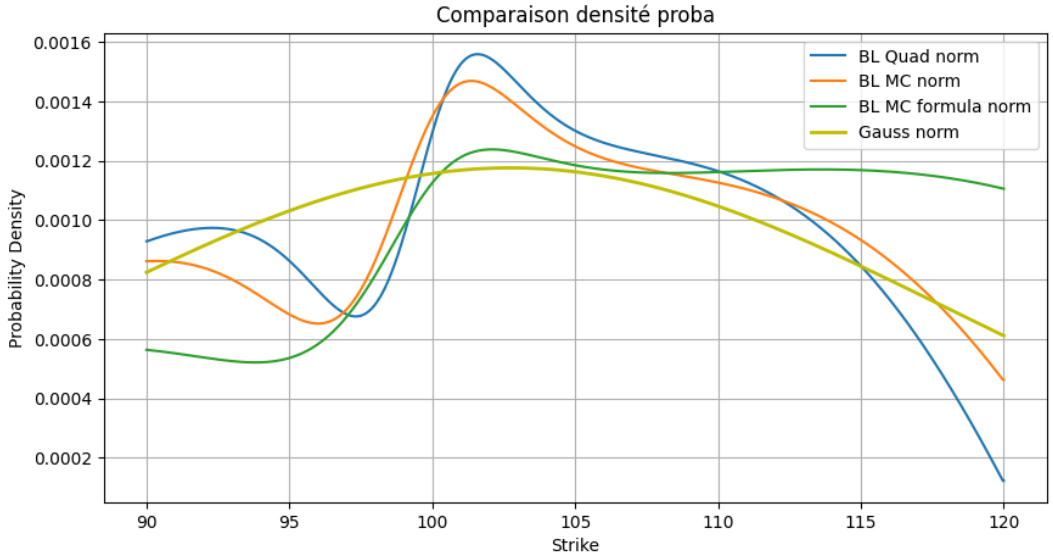
Nous voyons que l'on peut mieux faire alors nous voulons utiliser un algorithme d'optimisation pour trouver les paramètres optimaux mu et sigma. Nous implémentons Nelder Mead :

```
def densite_normale(x, mu, sigma):
    return (1 / (sigma * sqrt(2 * pi))) * exp(-0.5 * ((x - mu) / sigma) ** 2)

def objectif(params, plage, densite_breeden):
    mu, sigma = params
    densite_gauss = [densite_normale(x, mu, sigma) for x in plage]
    mse = sum((dg - db) ** 2 for dg, db in zip(densite_gauss, densite_breeden)) / len(plage)
    return mse


def nelder_mead2(x1, x2, x3, f,max_iterations):

    for i in range(max_iterations):

        if f(x1)>f(x2) and f(x3)<f(x2):
            a=x1
            x1=x3
            x3=a
```

```
        if f(x3)<f(x2) and f(x1)<f(x3):
            a=x2
            x2=x3
            x3=a
        if f(x3)<f(x1) and f(x1)<f(x2):
            a=x1
            x1=x2
            x2=x3
            x3=a
        if f(x2)<f(x1) and f(x1)<f(x3):
            a=x1
            x1=x2
            x2=a
        if f(x2)<f(x3) and f(x3)<f(x1):
            a=x1
            x1=x2
            x2=x3
            x3=a


        x0=[(x1[0]+x2[0])/2,(x1[1]+x2[1])/2]

        xr=[2*x0[0]-x3[0],2*x0[1]-x3[1]]

        if f(xr)<f(x1):
            xe=[x0[0]+2*(x0[0]-x3[0]),x0[1]+2*(x0[1]-x3[1])]
            if f(xe)<=f(xr):
                x3=xe
            else:
                x3=xr
        if f(xr)<f(x2):
            x3=xr
        if f(xr)>=f(x2):
            if f(xr)>=f(x2) and f(xr)<f(x3):
                xc=[0.5*(x0[0]+xr[0]),0.5*(x0[1]+xr[1])]
                if f(xc)<f(xr):
                    x3=xc
                else:
                    x2=[0.5*(x1[0]+x2[0]),0.5*(x1[1]+x2[1])]
                    x3=[0.5*(x1[0]+x3[0]),0.5*(x1[1]+x3[1])]
            if f(xr)>=f(x3):
                xc=[0.5*(x0[0]+x3[0]),0.5*(x0[1]+x3[1])]
                if f(xc)<=f(x3):
                    x3=xc
                else:
                    x2=[0.5*(x1[0]+x2[0]),0.5*(x1[1]+x2[1])]
                    x3=[0.5*(x1[0]+x3[0]),0.5*(x1[1]+x3[1])]

    return[(x1[0]+x2[0]+x3[0])/3,(x1[1]+x2[1]+x3[1])/3]

x1 = [90, 0.00000001]
x2 = [102, 10]
x3 = [110, 50]

plage = results['Strike']
densite_breeden = results['Density_BL_Quad']
```

```
resultats = nelder_mead2(x1, x2, x3, lambda params: objectif(params, plage, densite_breeden),
max_iterations=1000)

mu_optimal, sigma_optimal = resultats
print("mu optimal:", mu_optimal)
print("sigma optimal:", sigma_optimal)

results['Gaussienne_fit'] = results['Strike'].apply(lambda x: densite_normale(x, mu_optimal,
sigma_optimal))
Gauss_fit=results['Gaussienne_fit']

total_density_Gaussn_fit = sum(Gauss_fit)
Gauss_fit_norm = [d / total_density_Gaussn_fit for d in Gauss_fit]
results['Gauss_fit_Norm']=Gauss_fit_norm
results
```

Par la suite, on obtient $\mu = 102.73$ et $\sigma = 15.1$
Et voici la gaussienne que nous obtenons :



Maintenant comparons les densités obtenues en les comparant selon différents critères les quatres premiers moments (implémentés d'après le papier de D.Shimko), Skewness, Khurtosis, moyenne, et écart type et le test de Kolmogorov, Smirnov

Pour les quatre premiers moments, nous nous sommes basés sur ces formules :

$$p_1 = E\left[S^*\right]$$

$$p_2 = E\left[\left(S^* - p_1\right)^2\right]$$

$$p_3 = E\left[\left(S^* - p_1\right)^3\right] / p_2^{3/2}$$

$$p_4 = E\left[\left(S^* - p_1\right)^4\right] / p_2^{2}$$

Voici le code est le résultat pour ces statistiques :

```
distributions = {
    'BL_Quadratic': results['Density_BL_Quad_norm'],
    'BL_MC': results['Density_BL_MC_norm'],
    'BL_Theorical': results['Density_Norm'],
    'Gaussienne_fit' : results['Gauss_fit_Norm'],
}

def calculer_moments(data):
    moyenne = sum(data) / len(data)
    variance = sum((x - moyenne) ** 2 for x in data) / len(data)
    skewness = (sum((x - moyenne) ** 3 for x in data) / len(data)) / (variance ** 1.5)
    kurtosis = ((sum((x - moyenne) ** 4 for x in data) / len(data)) / (variance ** 2)) - 3
    return moyenne, variance, skewness, kurtosis

comparaison = {}
for name, data in distributions.items():
    moments = calculer_moments(data)
    comparaison[name] = moments

df_comparaison = pd.DataFrame.from_dict(comparaison, orient='index', columns=['Moyenne', 'Varianc
df_comparaison
```

Voici le dataframe de résultat :

	Moyenne	Variance	Skewness	Kurtosis
BL_Quadratic	0.001	1.014361e-07	-0.436022	-0.046206
BL_MC	0.001	6.623500e-08	0.066359	-1.011378
BL_Theorical	0.001	7.034335e-08	-0.980991	-0.862499
Gaussienne_fit	0.001	2.470151e-08	-0.739332	-0.511169

Par la suite, nous avons essayé d'implémenter le test de Kolmogorov Smirnov mais sans les libraires, nous n'avons pas eu de résultats concluants... Voici quand même les code que nous avons essayé d'implémenter

```
def empirical_cdf(sorted_data):
    n = len(sorted_data)
    return [i/n for i in range(1, n+1)]

def ks_test(cdf1, cdf2):
    ks_statistic = max(abs(c1 - c2) for c1, c2 in zip(cdf1, cdf2))
    return ks_statistic

cdf_distributions = {name: empirical_cdf(data) for name, data in distributions.items()}


data = []

for name1, cdf1 in cdf_distributions.items():
```

```
        for name2, cdf2 in cdf_distributions.items():
            if name1 != name2:
                ks_statistic = ks_test(cdf1, cdf2)
                data.append({'Distribution1': name1, 'Distribution2': name2, 'KS_Statistic': ks_stati


    df_comparaison_testKS = pd.DataFrame(data)
    df_comparaison_testKS
```

Ce que nous pouvons conclure :

Moyenne (Toutes égales à 0.001) : Toutes les densités calibrées semblent être centrées autour de la même moyenne, qui est très proche de zéro. Cela suggère que, en moyenne, les prix d'exercice considérés sont proches du prix actuel de l'actif sous-jacent, ajusté pour le coût de portage dans un contexte risque-neutre. Même si la normalisation que l'on a faite peut avoir un impact.

Variance :

On remarque que la densité 'BL_Quadratic' montre une variance relativement plus élevée, ce qui indique une plus grande incertitude ou une plus large gamme de résultats possibles pour le prix de l'actif sous-jacent, selon les prix des options.
'Gaussienne_fit'a la plus petite variance, ce qui implique que les prix des options suggèrent une distribution plus étroite autour du prix d'exercice attendu.

Skewness (Asymétrie) :

'BL_MC' a une légère asymétrie positive, ce qui pourrait indiquer une légère anticipation du marché d'une augmentation des prix de l'actif sous-jacent. Pour les trois densités, 'BL_Quadratic', 'BL_Theorical', et 'Gaussienne_fit', elles montrent une asymétrie négative, avec 'BL_Theorical' présentant la plus forte asymétrie négative.
Cela peut refléter une anticipation du marché d'une baisse des prix de l'actif sous-jacent, ou une probabilité plus élevée de résultats inférieurs au prix actuel.

Kurtosis :

Les densités, 'BL_MC' et 'BL_Theorical', avec un kurtosis négatif, montrent des distributions avec des queues plus fines que celles d'une distribution gaussienne, indiquant une probabilité moindre d'occurrences extrêmes. Les densités, 'BL_Quadratic' et 'Gaussienne_fit' ont un kurtosis plus proche de zéro, suggérant une distribution des événements extrêmes semblable à une distribution normale.

Donc, ces observations pourraient suggérer que les participants du marché, en moyenne, n'anticipent pas de grands écats par rapport au prix actuel de l'actif sous-jacent, tout en montrant une certaine préoccupation pour des résultats défavorables plus faibles que le prix actuel.
La distribution 'BL_Quadratic' suggère une plus grande incertitude quant aux prix futurs, tandis que la distribution 'Gaussienne_fit' semble indiquer que les participants du marché s'attendent à moins de volatilité dans les prix futurs.

Finalement, cela montre que la méthode de Shimko pour estimer la densité ajoute une information supplémentaire en prenant en compte l'interpolation du smile de volatilité pour voir les tendances futures sur les prix d'options à différents strikes.

Question 2 - Tirage aléatoire des prix dans notre densité

Maintenant, nous voulons vérifier, en faisant des tirages dans la loi implicite, nous prendrons celle calculée à partir de l'interpolation quadratique. le but est de vérifier si l'on trouve un prix de modèle proche du prix de marché pour toutes les options.

Nous avons essayé de faire un code basé sur l'idée d'utiliser la méthode d'inversion de la fonction de répartition cumulative pour générer des prix simulés à partir de votre densité risque-neutre.

Et ensuite, pour chaque prix simulé, calculez le prix théorique de l'option en utilisant la formule de Black-Scholes au strike associé.

Nous n'avons pas réussi à optimiser notre code pour qu'il s'exécute assez rapidement avec une bonne précision.

Partie II – Interpolation et volatilité locale

Question 3

On importe les données du TD préalablement rentrées dans un Excel :

```
prices=pd.read_excel('datatd5.xlsx', usecols='B,C,D')
```

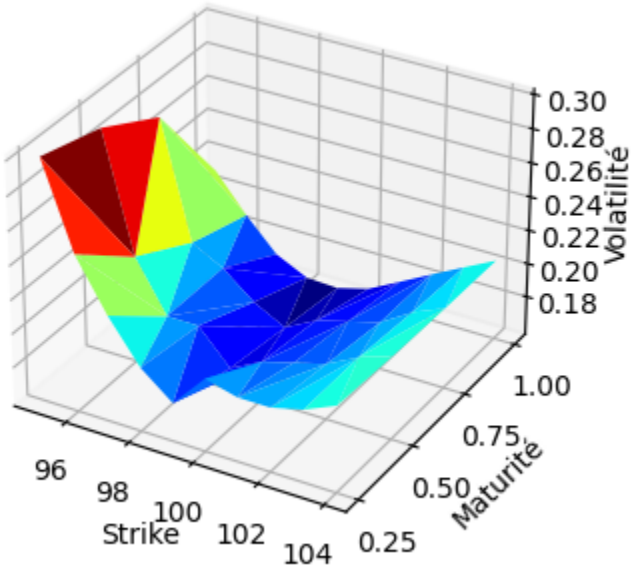
On calcule la volatilité implicite de chaque option :

```
volatility=[]
S0=100
r=0

for i in range(len(prices)):
    volatility.append(implied_volatility_newton_raphson(S0, prices['Strike'][i],
    prices['Maturité'][i], r, prices['Prix'][i]))

prices['Volatility']=volatility
```

NB: La fonction "implied_volatility_newton_raphson" est la même que celle de la partie 1 (cf. page 4).

On plot la nappe de volatilité :

```
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
surf = ax.plot_trisurf((prices['Strike']), prices['Maturité'], prices['Volatility'],
cmap=cm.jet, linewidth=0.2)
plt.show()
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.show()
```



Pour l'interpolation, nous utilisons une interpolation linéaire, en passant par l'intermédiaire de 2 options de strike 99.5 et de maturités 6 mois et 9 mois :

```
def interpolation(vol1, vol2, vol3, vol4):
    inter1=(vol1+vol2)/2 #Volatilité d'une option K=99.5 Maturité 9 mois
    inter2=(vol3+vol4)/2 #Volatilité d'une option K=99.5 Maturité 6 mois
    interf=(2/3)*inter1+(1/3)*inter2 #Volatilité d'une option K=99.5 Maturité 8 mois
    return interf
```

On extrait du dataset les volatilités utiles (strikes 99 et 100 et maturités 6 et 9 mois):

```
vol1 = prices[(prices['Strike'] == 99) & (prices['Maturité'] == 0.75)]['Volatility'].values[0]
vol2 = prices[(prices['Strike'] == 100) & (prices['Maturité'] == 0.75)]['Volatility'].values[0]
vol3 = prices[(prices['Strike'] == 99) & (prices['Maturité'] == 0.5)]['Volatility'].values[0]
vol4 = prices[(prices['Strike'] == 100) & (prices['Maturité'] == 0.5)]['Volatility'].values[0]
```

On calcule la vol puis le prix :

```
vol=interpolation(vol1, vol2, vol3, vol4)
print(vol)

S0=100
K=99.5
T=8/12
r=0
print(black_scholes_call_price(S0, K, T, r, vol))
```

On obtient, pour l'option de strike 99.5 et de maturité 8 mois, une volatilité implicite de 0.1717 et un prix de 5.826.

Question 4

EDP

Dans le cadre de la méthode par EDP, on va calculer le prix de l'option de strike $K = 100$, à différents prix de sous-jacent et différents temps en partant de la condition terminale : $C(T, S) = (S_T - K)_+$
On utilise l'équation suivante :

$$r_t C(t, S_t) = \delta_t C + r_t S_t \delta_S C + \frac{1}{2}\sigma(t, S_t)^2 S_t^2 \delta_S^2 C$$

Or, on travaille avec $r = 0$ donc on obtient :

$$\delta_t C + \frac{1}{2}\sigma(t, S_t)^2 S_t^2 \delta_S^2 C = 0$$

Ce qui revient à :

$$\frac{C(t, S_t) - C(t - \tau, S_t)}{\tau} + \frac{1}{2}\sigma(t, S_t)^2 S_t^2 \frac{C(t, S_t + \xi) - 2C(t, S_t) + C(t, S_t - \xi)}{\xi^2} = 0$$

Donc finalement :

$$C(t - \tau, S_t) = C(t, S_t) + \tau * \left[\frac{1}{2}\sigma(t, S_t)^2 S_t^2 \frac{C(t, S_t + \xi) - 2C(t, S_t) + C(t, S_t - \xi)}{\xi^2}\right]$$

Ici, $\sigma(t, S_t)^2 = a + b\left[\rho(x - m) + \sqrt{(x - m)^2 + \sigma^2}\right]$ avec a,b,ρ,m et σ paramètres du modèle SVI et $x = \ln \frac{K}{F_T}$ la log-moneyness. Ici, $F_T = S$ car on travaille avec des taux sans-risque nuls.

On aboutit à ce code, en partant de S_T allant de 0 à 200 et avec 100 pas de temps :

```
size=(200,101)
data=np.zeros(size)
for j in range(100,0,-1):
    print(j)
    for i in range(199):
        if j==100:
            data[i][j]=np.max([0,(200-i)*ksi-K])
        else:
            S=(200-i)*ksi
            x=np.log(K/S)
            sigmaimp=np.sqrt(a+b*(rho*(x-m)+np.sqrt((x-m)**2)+sigma**2))
            data[i][j]=data[i][j+1]+tau*(0.5*(sigmaimp**2)*(S**2)*
            ((data[i-1][j+1]-2*data[i][j+1]+data[i+1][j+1])/ksi**2))
```

Malheureusement, ce code nous donnait des résultats incohérents que nous n'avons pas réussi à comprendre et à corriger. Nous nous sommes donc concentrés sur la méthode de Monte Carlo.

Monte Carlo

Dans le cadre de la méthode de Monte-Carlo, on simule pas à pas différents prix de sous-jacent à partir de S_0 :

$$S_\tau = S_0 \exp\left(-\frac{\sigma(S_\tau, \tau)^2 \tau}{2} + \sigma(S_0, \tau) W_1\right)$$

Puis :

$$S_{k\tau} = S_{(k-1)\tau} \exp\left(-\frac{\sigma(S_\tau, \tau)^2 \tau}{2} + \sigma(S_\tau, \tau) W_k\right)$$

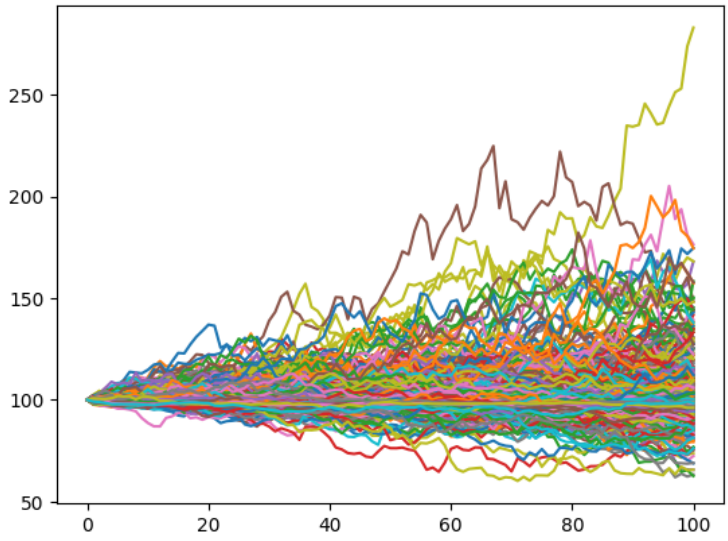
Avec n le nombre de pas et $k \in [1, n]$

On utilise ce code pour créer une simulation avec les paramètres de modèles à définir :

```
def simulations(a, b, sigma, rho, m, K, T, nb_steps, nb_simulations, S0):
simulations=[]
np.random.seed(123)
for j in range(nb_simulations):
```

```
        sim=[S0]
        tau=T/nb_steps
        S=S0
        for i in range(nb_steps):
            x=np.log(K/S)
            sigmaimp=np.sqrt(a+b*(rho*(x-m)+np.sqrt((x-m)**2)+sigma**2))
            W=np.random.normal(0,1)
            S=S*np.exp((-(sigmaimp**2)*tau)/2+sigmaimp*(np.sqrt(tau)*W))
            sim.append(S)
        simulations.append(sim)
    return simulations
```

Voici un exemple de 1000 simulations avec $T = 1$, $K = 100$, $S0 = 100$ et 100 pas de temps, ainsi que des paramètres de modèles arbitraires : $a = 0$, $b = 0.35$, $\rho = 0$, $m = 0.075$ et $\sigma = 0.02$.



Suite à une simulation de Monte-Carlo, pour calculer le prix de l'option et l'erreur standard, on utilise ces deux fonctions :

```
def Pricing(simulations, K):
    sum=0
    for path in simulations:
        payoff=np.max([0, path[-1]-K])
        sum+=payoff
    return sum/nb_simulations

def StandardError(simulations, K):
    sum=0
    mean=Pricing(simulations, K)
    for path in simulations:
        payoff=np.max([0, path[-1]-K])
        deviation=(payoff-mean)**2
        sum+=deviation
        var=sum/nb_simulations
    return np.sqrt(var/nb_simulations)
```

Pour estimer quels étaient les paramètres optimaux pour le modèle, nous avons testé plusieurs combinaisons avec ce code :

```
df = pd.DataFrame(columns = ['rho', 'a', 'b', 'sigma', 'm', 'Prix', 'StandardError'])
for rho in (0, -0.4, -0.8):
    for sigma in (0.01, 0.02, 0.03):
        for b in (0.35, 0.425, 0.5):
            for m in (0, 0.075, 0.15):
                for a in (0, 0.05, 0.1):
                    test=simulations(a,b,sigma,rho,m,K,T,nb_steps,nb_simulations,S0)
                    new_row = {'rho': rho, 'a': a, 'b' : b, 'sigma' : sigma, 'm' : m,
                    'Prix' : Pricing(test,K), 'StandardError' : StandardError(test,K)}
                    if Pricing(test,K)<7 and Pricing(test,K)>6:
                        df.loc[len(df)] = new_row
```

Cela nous permet de comparer, pour une option de strike 100 et de maturité 1 an, le prix de marché et le prix calculé grâce à la méthode de Monte-Carlo et avec 243 combinaisons de paramètres différents du modèle SVI (3 valeurs pour chacun des paramètres). Pour indication, ce code a un temps de compilation d'environ 2 minutes. Ces valeurs ont été choisies proches des intervalles d'étude sur la slide 70 du cours et après plusieurs tatonnements lors du projet. Le prix de marché de l'option étant de 6.58, on décide arbitrairement de retenir les paramètres qui obtiennent des prix entre 6 et 7. Cela aboutit au résultat suivant :

	rho	a	b	sigma	m	Prix	StandardError
0	0	0	0.350	0.01	0.075	6.311906	0.610764
1	0	0	0.425	0.01	0.075	6.914123	0.723314
2	0	0	0.350	0.02	0.075	6.370519	0.612514
3	0	0	0.350	0.03	0.075	6.444039	0.614886

On peut donc voir qu'il convient d'utiliser plutôt $\rho = 0$, $a = 0$ et $m = 0.075$. Nous allons tester plus de valeurs de b et σ, en réutilisant le code précédent et en fixant ρ, a et m. Nous resserrons aussi les valeurs autour du prix de marché pour être plus précis :

```
df = pd.DataFrame(columns = ['rho', 'a', 'b', 'sigma', 'm', 'Prix', 'StandardError'])
rho=0
m=0.075
a=0
for sigma in (0.01, 0.015, 0.02, 0.025, 0.03):
    for b in (0.35, 0.36, 0.37, 0.38, 0.39, 0.4):
        test=simulations(a,b,sigma,rho,m,K,T,nb_steps,nb_simulations,S0)
        new_row = {'rho': rho, 'a': a, 'b' : b, 'sigma' : sigma, 'm' : m,
        'Prix' : Pricing(test,K), 'StandardError' : StandardError(test,K)}
        if Pricing(test,K)<6.68 and Pricing(test,K)>6.48:
            df.loc[len(df)] = new_row
```

On obtient ce résultat :

	rho	a	b	sigma	m	Prix	StandardError
0	0	0	0.38	0.010	0.075	6.550086	0.655935
1	0	0	0.39	0.010	0.075	6.630561	0.670836
2	0	0	0.37	0.015	0.075	6.502479	0.641627
3	0	0	0.38	0.015	0.075	6.585144	0.656758
4	0	0	0.39	0.015	0.075	6.665437	0.671664
5	0	0	0.37	0.020	0.075	6.540357	0.642614
6	0	0	0.38	0.020	0.075	6.623887	0.657706
7	0	0	0.36	0.025	0.075	6.494229	0.628658
8	0	0	0.37	0.025	0.075	6.579230	0.643789
9	0	0	0.38	0.025	0.075	6.661782	0.658824
10	0	0	0.36	0.030	0.075	6.531990	0.629970
11	0	0	0.37	0.030	0.075	6.619127	0.645156

On obtient plusieurs simulations avec des prix très proches du marché, mais devant faire un choix, et les erreurs standards étant relativement comparables, nous avons pris le plus proche:

- $b = 0.37$ et $\sigma = 0.025$

Ainsi, nous obtenons ce vecteur de paramètres : $a = 0$, $b = 0.37$, $rho = 0$, $m = 0.075$, $sigma = 0.025$
Après avoir fixé les paramètres $rho = 0$, $m = 0.075$, $sigma = 0.025$, nous voulons utiliser, pour chaque option, la méthode de Nelder Mead appliquée à cette fonction :

$$(CModel(a, b, K, T) - CMarche(K, T))^2$$

, afin de trouver les paramètres a et b qui la minimisent.
On a le code suivant pour la méthode de Nelder Mead en 2 dimensions :

```
def nelder_mead2(x1, x2, x3, f,max_iterations):

    for i in range(max_iterations):

        if f(x1)>f(x2) and f(x3)<f(x2):
            a=x1
            x1=x3
            x3=a
        if f(x3)<f(x2) and f(x1)<f(x3):
            a=x2
            x2=x3
            x3=a
        if f(x3)<f(x1) and f(x1)<f(x2):
            a=x1
            x1=x2
            x2=x3
            x3=a
        if f(x2)<f(x1) and f(x1)<f(x3):
            a=x1
            x1=x2
            x2=a
        if f(x2)<f(x3) and f(x3)<f(x1):
            a=x1
            x1=x2
            x2=x3
            x3=a
```

```
        x0=[(x1[0]+x2[0])/2,(x1[1]+x2[1])/2]

        xr=[2*x0[0]-x3[0],2*x0[1]-x3[1]]

        if f(xr)<f(x1):
            xe=[x0[0]+2*(x0[0]-x3[0]),x0[1]+2*(x0[1]-x3[1])]
            if f(xe)<=f(xr):
                x3=xe
            else:
                x3=xr
        if f(xr)<f(x2):
            x3=xr
        if f(xr)>=f(x2):
            if f(xr)>=f(x2) and f(xr)<f(x3):
                xc=[0.5*(x0[0]+xr[0]),0.5*(x0[1]+xr[1])]
                if f(xc)<f(xr):
                    x3=xc
                else:
                    x2=[0.5*(x1[0]+x2[0]),0.5*(x1[1]+x2[1])]
                    x3=[0.5*(x1[0]+x3[0]),0.5*(x1[1]+x3[1])]
            if f(xr)>=f(x3):
                xc=[0.5*(x0[0]+x3[0]),0.5*(x0[1]+x3[1])]
                if f(xc)<=f(x3):
                    x3=xc
                else:
                    x2=[0.5*(x1[0]+x2[0]),0.5*(x1[1]+x2[1])]
                    x3=[0.5*(x1[0]+x3[0]),0.5*(x1[1]+x3[1])]

    return[(x1[0]+x2[0]+x3[0])/3,(x1[1]+x2[1]+x3[1])/3]
```

Puis ce code afin de trouver le minimum de la fonction :

```
def CMarché(K,T):
    return prices[(prices['Strike'] == K) & (prices['Maturité'] == T)]['Prix'].values[0]

def CModel(K,T,a,b):
    return Pricing(simulations(a,b,sigma,rho,m,K,T,nb_steps,nb_simulations,S0),K)

def target(param):
    return (CModel(K,T,param[0],param[1])-CMarché(K,T))**2

nelder_mead2([0,0],[0.1,0],[0,0.4],target,100)
```

Nous avons tenté plusieurs valeurs initiales différentes mais nous n'avons trouvé aucun resultat satisfaisant.

Question 6

L'algorithme de simulation du modèle PDV est le suivant :

```
def simulationsPDV(beta0, beta1, beta2, lambda1, lambda2, K, T, nb_steps, nb_simulations, S0):
    simulations=[]
    np.random.seed(123)
    for j in range(nb_simulations):
        sim=[S0]
```

```
        tau=T/nb_steps
        S=S0
        K1=lambda1*np.exp(-lambda1*tau)
        K2=lambda1*np.exp(-lambda2*tau)
        for i in range(nb_steps):
            somme1=0
            somme2=0
            for j in range(0,i-1):
                somme1+=(sim[j+1]/sim[j])*((i-j)/T)*K1
                somme2+=(sim[j+1]/sim[j])**2*((i-j)/T)*K2
            sigmaimp=beta0+beta1*somme1+beta2*np.sqrt(somme2)
            W=np.random.normal(0,1)
            S=S*np.exp((-(sigmaimp**2)*tau)/2+sigmaimp*(np.sqrt(tau)*W))
            sim.append(S)
        simulations.append(sim)
    return simulations
```

Les résultats sont difficilement exploitables. Cela est sûrement dû aux mauvais paramètres du modèle mais même après plusieurs tatonnements, nous n'avons pas trouvé de résultats convenables.

Partie III – Test d'algorithmes d'optimisation

Dans cette section, 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. Le but est de 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.