Projet Model Calibration

Gaspard MARTY, Edouard MINIAU

24 janvier 2025

1 Local Volatility

1.1 Black-Scholes volatility

Listing 1 – Fonctions Black-Scholes et implied volatility

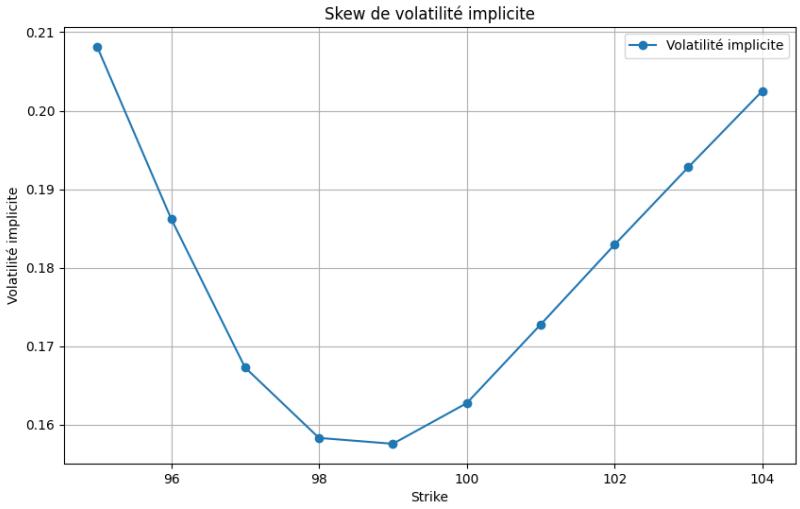
```
def black_scholes_call_price_for_IM(S, K, T, r, sigma):
    if T <= 0:
        return max(S - K, 0)
    d1 = (np.log(S / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    return S * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)

def implied_volatility(price, S, K, T, r, tol=1e-5, max_iter=1000):
    lower_bound = 0.0001
    upper_bound = 5.0
    iteration = 0
    while iteration < max_iter:
        mid = (lower_bound + upper_bound) / 2
        mid_price = black_scholes_call_price_for_IM(S, K, T, r, mid)
        if abs(mid_price - price) < tol:
            return mid
        elif mid_price < price:
            lower_bound = mid
        else:
            upper_bound = mid
        iteration += 1
    return mid
```

Listing 2 – Implémentation

```
implied_vols = []
for K, price in zip(strike, prices):
    vol = implied_volatility(price, S, K, T, rf)
    implied_vols.append(vol)
```

Strike	Option Price	Implied Volatility
95	10.93	0.208123
96	9.55	0.186236
97	8.28	0.167306
98	7.40	0.158340
99	6.86	0.157592
100	6.58	0.162762
101	6.52	0.172779
102	6.49	0.182937
103	6.47	0.192824
104	6.46	0.202512

TABLE 1 – Tableau avec taille réduite



La figure ci-dessus représente le smile de volatilité calibré par dichotomie.

1.2 SVI et SVI généralisée

1.2.1 SVI

Le modèle SVI sert ici à calculer les vols de n'importe quel strike à partir des 10 observations de l'énoncé. Pour ceci, nous avons besoin de calibrer les paramètres de ce modèle grâce à l'algorithme de Nelder-Mead en minimisant l'erreur quadratique entre les données observées et modélisées

Listing 3 – SVI

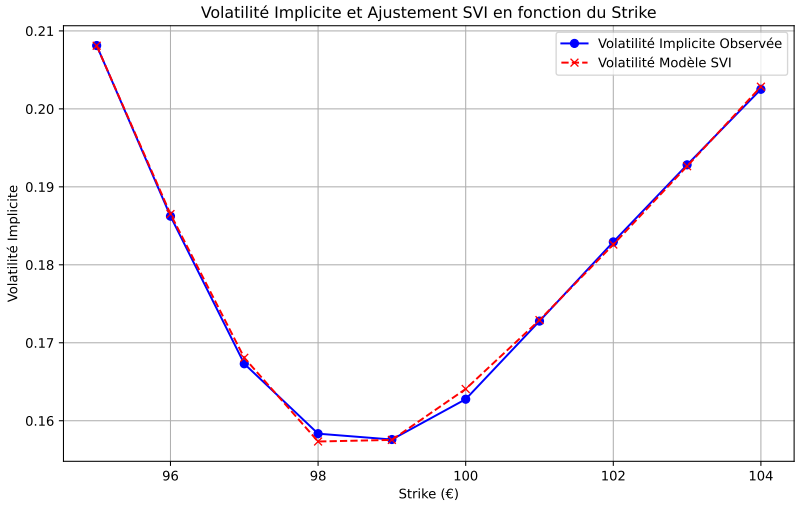
```
def svi_model(x, a, b, rho, m, sigma):
    return a + b * (rho * (x - m) + np.sqrt((x - m)**2 + sigma**2))

def cost_function_1(params, x_data, y_data):
    a, b, rho, m, sigma = params
    y_pred = svi_model(x_data, a, b, rho, m, sigma)
    return np.sum((y_data - y_pred)**2)

def nelder_mead(initial_params, cost_function, x_data, y_data, tol=1e-6,
max_iter=1000):
    n = len(initial_params)
    new_params = [initial_params]
    for i in range(n):
        noise_params = initial_params.copy()
        noise_params[i] += 0.1
        new_params.append(noise_params)

    for step in range(max_iter):
        new_params.sort(key=lambda params: cost_function(params, x_data,
        y_data))
        best = new_params[0]
        worst = new_params[-1]
        second_worst = new_params[-2]

        x0 = np.mean(new_params[:-1], axis=0)

        xr = x0 + (x0 - worst)
        if cost_function(xr, x_data, y_data) < cost_function(best, x_data,
        y_data):
            xe = x0 + 2 * (x0 - worst)
            if cost_function(xe, x_data, y_data) < cost_function(xr, x_data,
            y_data):
```

```
                new_params[-1] = xe
            else:
                new_params[-1] = xr
        elif cost_function(xr, x_data, y_data) < cost_function(second_worst,
        x_data, y_data):
            new_params[-1] = xr
        else:
            xc = x0 + 0.5 * (worst - x0)
            if cost_function(xc, x_data, y_data) < cost_function(worst,
            x_data, y_data):
                new_params[-1] = xc
            else:
                for i in range(1, len(new_params)):
                    new_params[i] = best + 0.5 * (new_params[i] - best)

        if np.max([np.linalg.norm(new_params[i] - best) for i in range(1,
        len(new_params))]) < tol:
            break
    return best
```

Listing 4 – Implémentation

```
log_moneyness = np.log(strike / (S*(1+rf*T)))
a, b, rho, m, sigma = 0.1, 0.1, -0.1, 0.0, 0.1
params = np.array([a, b, rho, m, sigma])
params_optimized = nelder_mead(params, cost_function_1, log_moneyness,
implied_vols)
```

Paramètres calibrés avec Nelder-Mead : a=0.1327503264197134, b=1.7080380876092522, rho=-0.3487622531551353, m=-0.023087910128464836, sigma=-0.01467695915604258

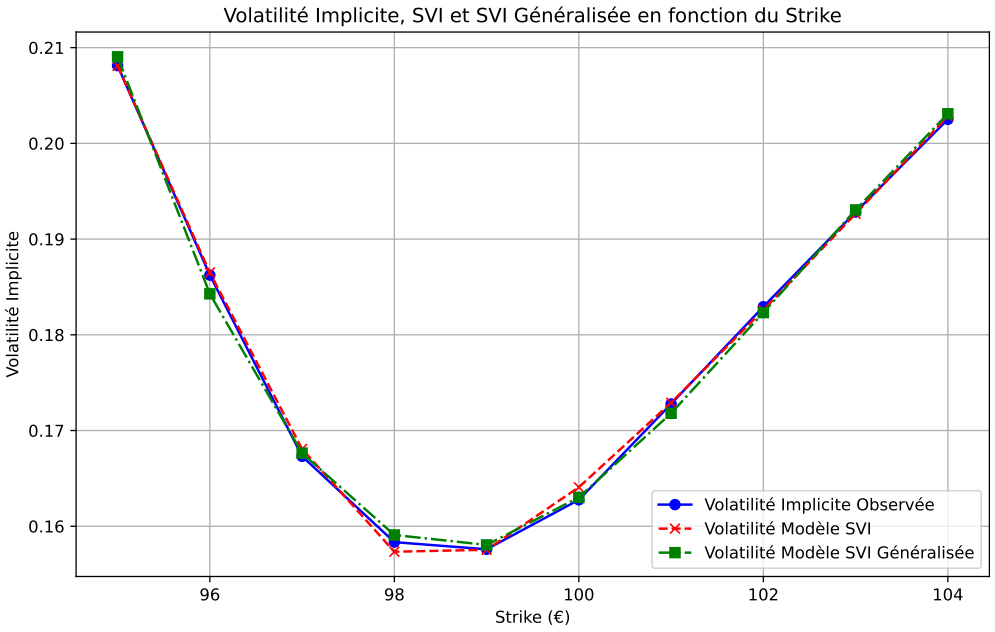


1.2.2 SVI généralisée

Listing 5 – SVI G

```
def svi_generalized_model(x, a, b, rho, m, sigma, beta):
    z = x  / (beta ** np.abs(x - m))
    return a + b * (rho * z + np.sqrt(z**2 + sigma**2))

def cost_function_generalized(params, x_data, y_data):
    a, b, rho, m, sigma, beta = params
    y_pred = svi_generalized_model(x_data, a, b, rho, m, sigma, beta)
    return np.sum((y_data - y_pred)**2)

a_g, b_g, rho_g, m_g, sigma_g, beta_g = 0.1, 0.1, -0.1, 0.0, 0.1, 1.25
params_generalized = np.array([a_g, b_g, rho_g, m_g, sigma_g, beta_g])

params_optimized_generalized = nelder_mead(params_generalized,
cost_function_generalized, log_moneyness, implied_vols)
a_g, b_g, rho_g, m_g, sigma_g, beta_g = params_optimized_generalized

generalized_svi_vols = svi_generalized_model(log_moneyness, a_g, b_g, rho_g,
m_g, sigma_g, beta_g)
```

Paramètres calibrés avec Nelder-Mead (SVI Généralisée) : a=0.04738673756240744, b=0.14058448542298385, rho=0.33825723824102044, m=0.3423877832287773, sigma=-0.8333148764113061, beta=0.00026381738240097684



Les résultats montrent que les deux modèles capturent bien le skew de volatilité.

1.3 Local volatility

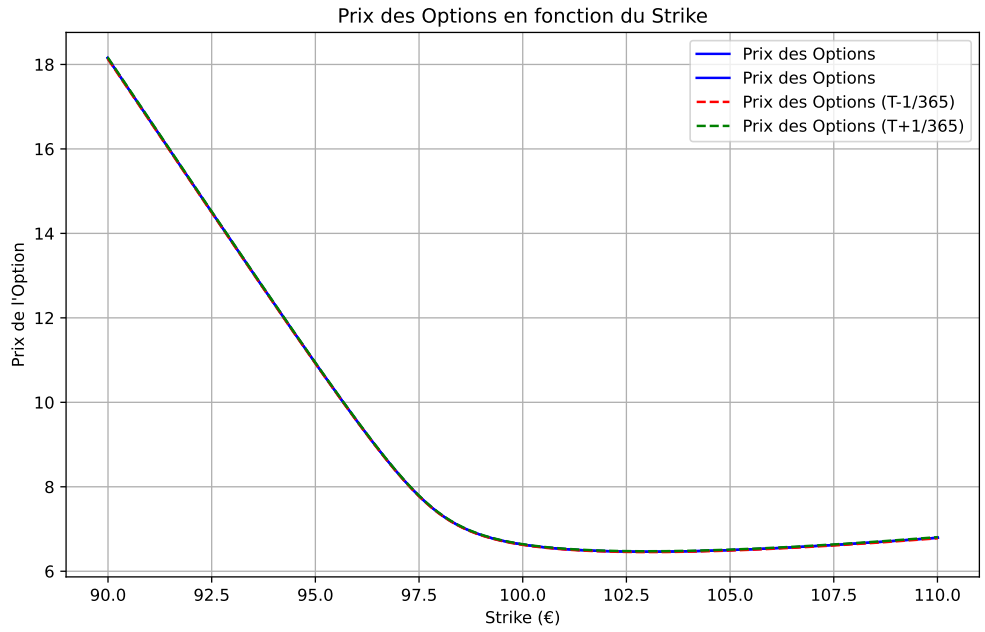
Listing 6 – Dupire local volatility

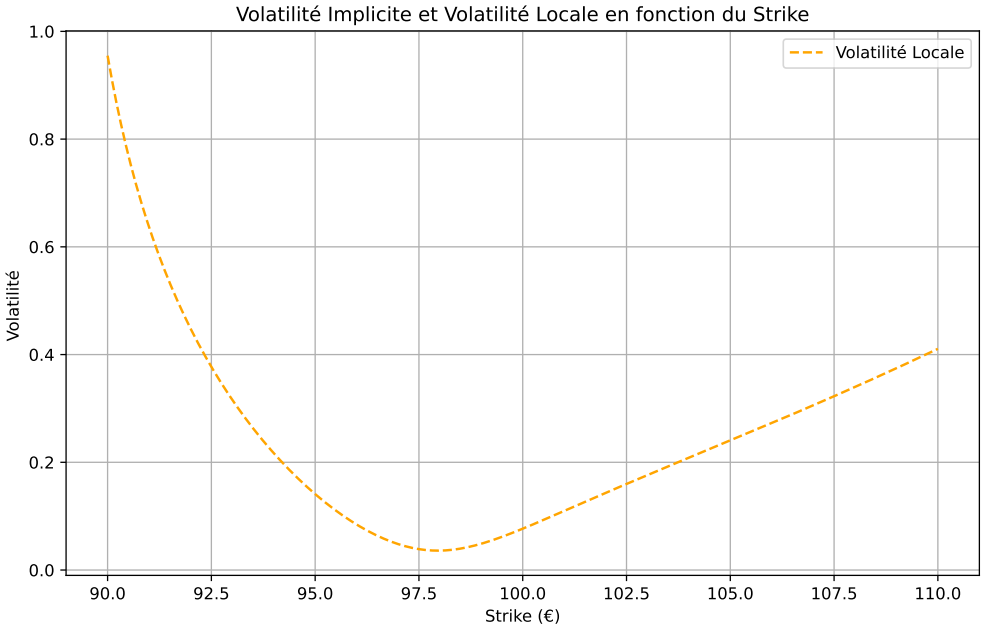
```
def dupire_local_volatility(a, b, rho, m, sigma,S=100, T=1, rf=0.002,
model='svi_model', beta=1.25):
    price_strikes = np.arange(89.99, 110.02, 0.01)
    log_moneyness_fine = np.log(price_strikes / (S*(1+rf*T)))
    log_moneyness_fine_inf = np.log(price_strikes / (S*(1+rf*(T-1/365))))
    log_moneyness_fine_sup = np.log(price_strikes / (S*(1+rf*(T+1/365))))

    if model == 'generalized_svi_model' or model == 'g' or model ==
    'generalized':
        vols_svi = svi_generalized_model(log_moneyness_fine, a, b, rho, m,
        sigma, beta)
        vols_svi_inf = svi_generalized_model(log_moneyness_fine_inf, a, b,
        rho, m, sigma, beta)
        vols_svi_sup = svi_generalized_model(log_moneyness_fine_sup, a, b,
        rho, m, sigma, beta)
    else:
        vols_svi = svi_model(log_moneyness_fine, a, b, rho, m, sigma)
        vols_svi_inf = svi_model(log_moneyness_fine_inf, a, b, rho, m, sigma)
        vols_svi_sup = svi_model(log_moneyness_fine_sup, a, b, rho, m, sigma)

    option_price_strikes = [black_scholes_call_price_for_IM(S, K, T, rf, vol)
    for K, vol in zip(price_strikes, vols_svi)]
    option_price_strikes_inf = [black_scholes_call_price_for_IM(S, K, T
    1/365, rf, vol) for K, vol in zip(price_strikes, vols_svi_inf)]
    option_price_strikes_sup = [black_scholes_call_price_for_IM(S, K,
    T+1/365, rf, vol) for K, vol in zip(price_strikes, vols_svi_sup)]

    dK = 0.01
    dC_dK = [(option_price_strikes[i+1] - option_price_strikes[i-1]) / (2 *
    dK) for i in range(1, len(price_strikes) - 1)]
    d2C_dK2 = [(option_price_strikes[i+1] - 2 * option_price_strikes[i] +
    option_price_strikes[i-1]) / dK**2 for i in range(1, len(price_strikes) -
    1)]

    dC_dT = [(option_price_strikes_sup[i] - option_price_strikes_inf[i]) / (2
    * 1/365) for i in range(1, len(price_strikes) - 1)]

    local_volatility = [max(0,np.sqrt(2*( dC_dT + rf * price_strikes * dC_dK)
    / (price_strikes**2 * d2C_dK2))) for price_strikes,dC_dT,dC_dK, d2C_dK2
    in zip(price_strikes,dC_dT,dC_dK, d2C_dK2)]

     return price_strikes, local_volatility

price_strikes,local_volatility =dupire_local_volatility(a, b, rho, m, sigma)
```

Pour calculer la volatilité locale, on utilise la formule de Dupire

$$\sigma(T,K) = \sqrt{\frac{2\frac{\partial C}{\partial T} + rK\frac{\partial C}{\partial K}}{K^2\frac{\partial^2 C}{\partial K^2}}}$$

Afin de calculer les dérivées première et seconde par rapport à K, on utilise des accroissements finis sur des petits intervalles de strike. On peut calculer la volatilité et donc le prix de l'option à ces strikes grâce au model SVI estimé précédemment ainsi que la formule de Black and Scholes. Afin de gérer la dérivée de C par rapport à t, nous recalculons tous les prix d'options avec un décalage de temps de +/- 1/365, permettant ainsi de calculer les accroissements finis.





La volatilité de Dupire locale, montre un smile de volatilité, reflétant l'impact des anticipations de marché sur la dynamique des prix des options.

1.4 SVI on the three option closest to the money

Le but de cette partie est d'arriver à calibrer parfaitement le modèle SVI pour les 3 options autour de la monnaie. Ceci est possible car le modèle SVI a plus de 3 paramètres.

Listing 7 – Implémentation

```
strike_Q4 = np.array([99, 100, 101])
prices_Q4 = np.array([6.86, 6.58, 6.52])
implied_vols_Q4 = []
for K, price in zip(strike_Q4, prices_Q4):
    vol = implied_volatility(price, S, K, T, rf)
    implied_vols_Q4.append(vol)
```

Strike	Option Price	Implied Volatility
99	6.86	0.157592
100	6.58	0.162762
101	6.52	0.172779

TABLE 2 – Tableau des prix d'options et de volatilité implicite

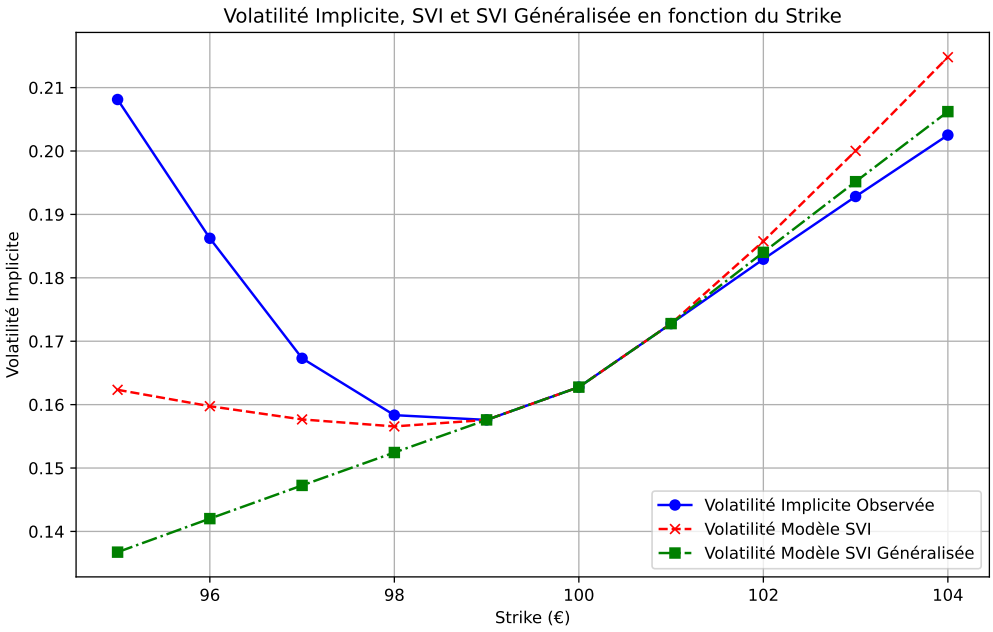
Listing 8 – Implémentation

```
log_moneyness_Q4 = np.log(strike_Q4 / (S*(1+rf*T)))
print(log_moneyness_Q4)

a_Q4, b_Q4, rho_Q4, m_Q4, sigma_Q4 = 0.1, 0.1, -0.1, 0.0, 0.1
params_Q4 = np.array([a_Q4, b_Q4, rho_Q4, m_Q4, sigma_Q4])

params_optimized_Q4 = nelder_mead(params_Q4, cost_function_1,
log_moneyness_Q4, implied_vols_Q4)
```

```
Paramètres calibrés avec Nelder-Mead:
a = 0.14330634409038884, b = 0.9776210357649092, rho = 0.6697239921204954,
m = -0.004282156063552058, sigma = -0.01822843325209278

Strike          Implied Vols      Log Moneyness
99              0.1576            -0.0120
100             0.1628            -0.0020
101             0.1728            0.0079
```

Listing 9 – SVI and generalized SVI

```
svi_vols_Q4 = svi_model(log_moneyness, *params_optimized_Q4)
a_g_Q4, b_g_Q4, rho_g_Q4, m_g_Q4, sigma_g_Q4, beta_g_Q4 = 0.1, 0.1, -0.1,
0.0, 0.1, 1.25
params_generalized_Q4 = np.array([a_g_Q4, b_g_Q4, rho_g_Q4, m_g_Q4,
sigma_g_Q4, beta_g_Q4])

params_optimized_generalized_Q4 = nelder_mead(params_generalized_Q4,
cost_function_generalized, log_moneyness_Q4, implied_vols_Q4)
a_g_Q4, b_g_Q4, rho_g_Q4, m_g_Q4, sigma_g_Q4, beta_g_Q4 =
params_optimized_generalized_Q4
generalized_svi_vols_Q4 = svi_generalized_model(log_moneyness, a_g_Q4,
b_g_Q4, rho_g_Q4, m_g_Q4, sigma_g_Q4, beta_g_Q4)
```

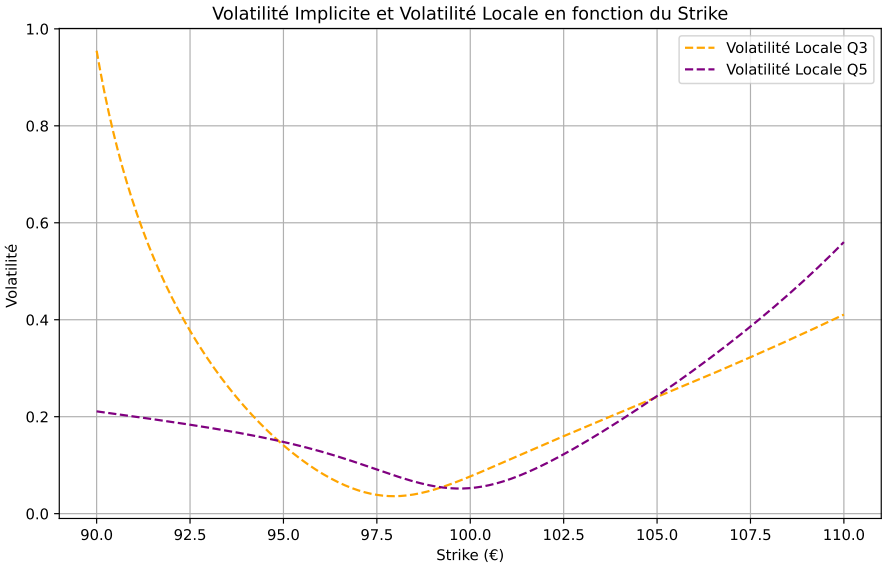


On remarque sur ce graphe que les modèles SVI ont bien été calibrés (les estimations des 3 options sont égales aux observations). Une contrepartie de cette calibration autour de la monnaie implique une moins bonne approximation des volatilités loin de la monnaie

1.5 Dupire with new SVI

Listing 10 – Dupire

```
price_strikes_Q4,local_volatility_Q4 =dupire_local_volatility(a_Q4, b_Q4,
rho_Q4, m_Q4, sigma_Q4)
```



Du fait de la calibratrion à la monnaie, on peut affirmer que la vol locale proche du strike =100 est meilleure sur la nouvelle courbe (violette), tandis que l'ancienne courbe est meilleure loin de la monnaie.

1.6 Stress Up and Down

Afin de stresser le prix de l'option de strike 100 sans créer d'arbitrage, 2 conditions sont nécéssaires. Premièrement, le prix d'une option ne peut pas être supérieur (resp. inf) qu'une option ayant un strike inf (resp. sup). Deuxièmement, la convexité de la courbe de C(k) doit être respéctée suivant ainsi l'inéquation suivante :

$$C(k_i) < 0.5 \times (C(k_{i-1}) + C(k_{i+1}))$$

Listing 11 – Stress up and down

```
prices_stressed_down, prices_stressed_up = prices.copy()

index_102 = np.where(strike == 102)[0][0]
index_101 = np.where(strike == 101)[0][0]
index_100 = np.where(strike == 100)[0][0]
index_99 = np.where(strike == 99)[0][0]

while prices_stressed_up[index_100] + 0.02 < prices_stressed_up[index_99] and
(prices_stressed_up[index_100] < 0.5* (prices_stressed_up[index_99] +
prices_stressed_up[index_101])) :

    prices_stressed_up[index_100] += 0.01

while prices_stressed_down[index_100] - 0.02 >
prices_stressed_down[index_101] and \
        (prices_stressed_down[index_100] < 0.5
        (prices_stressed_down[index_99] + prices_stressed_down[index_101]))\
        and (prices_stressed_down[index_101] < 0.5
        (prices_stressed_down[index_100] + prices_stressed_down[index_102])) :

    prices_stressed_down[index_100] -= 0.01
```

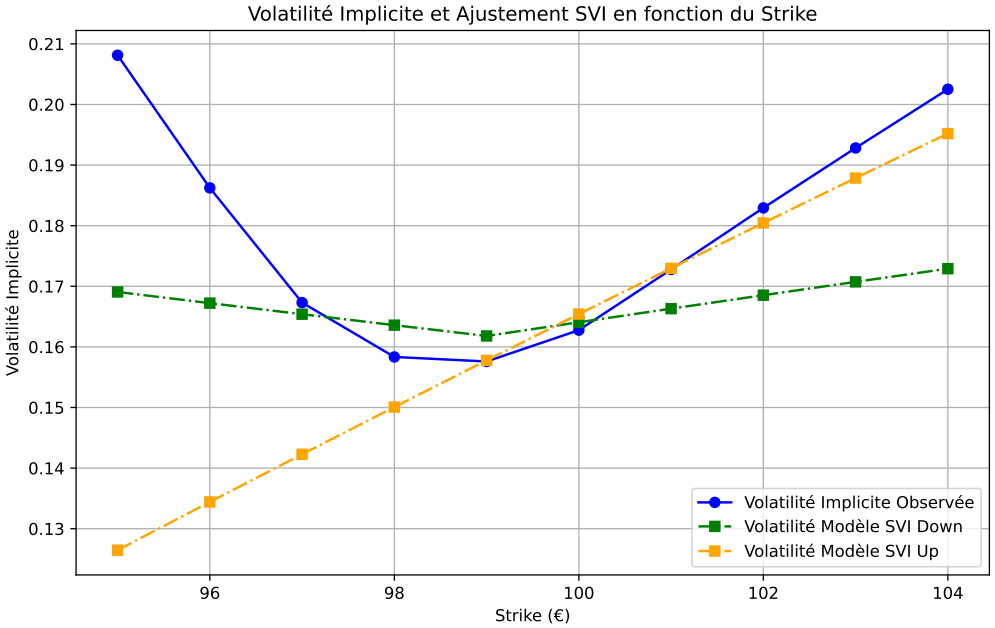
prices : [10.93 9.55 8.28 7.4 6.86 **6.58** 6.52 6.49 6.47 6.46]
prices stressed up : [10.93 9.55 8.28 7.4 6.86 **6.7** 6.52 6.49 6.47 6.46]
prices stressed down : [10.93 9.55 8.28 7.4 6.86 **6.54** 6.52 6.49 6.47 6.46]

Listing 12 – Stress up and down

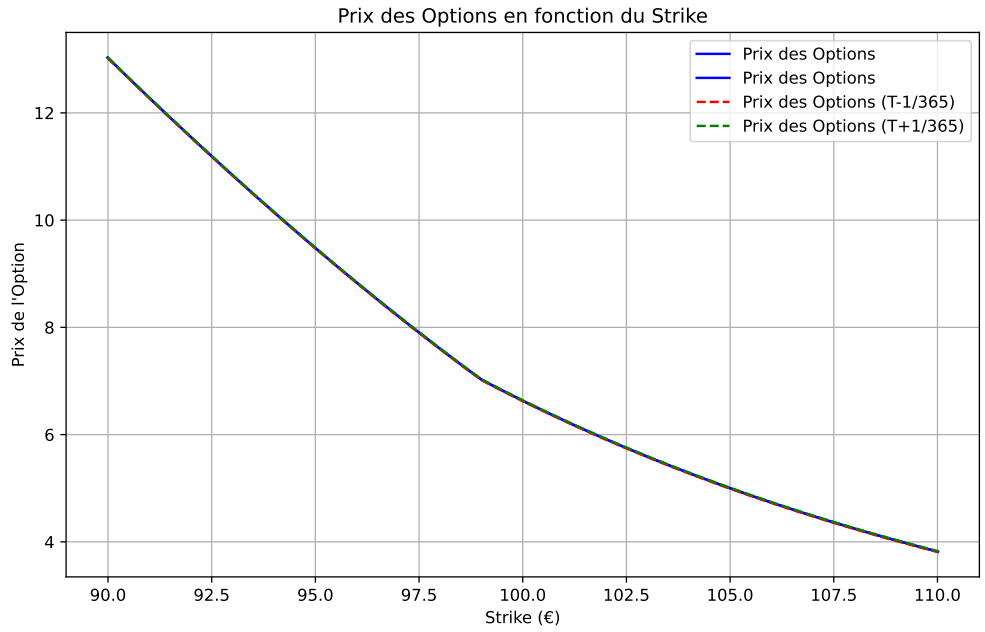
```
strike_Q6 = np.array([99, 100, 101])
prices_down_Q6 = np.array([prices_stressed_down[index_99],
prices_stressed_down[index_100], prices_stressed_down[index_101]])
prices_up_Q6 = np.array([prices_stressed_up[index_99],
prices_stressed_up[index_100], prices_stressed_up[index_101]])

implied_vols_down_Q6 = []
implied_vols_up_Q6 = []

for K, price_down in zip(strike_Q6, prices_down_Q6):
    vol1 = implied_volatility(price_down, S, K, T, rf)
    implied_vols_down_Q6.append(vol1)

for K, price_up in zip(strike_Q6, prices_up_Q6):
    vol2 = implied_volatility(price_up, S, K, T, rf)
    implied_vols_up_Q6.append(vol2)

log_moneyness_Q6 = np.log(strike_Q6 / (S*(1+rf*T)))

a_down_Q6, b_down_Q6, rho_down_Q6, m_down_Q6, sigma_down_Q6 = 0.1, 0.1, -0.1,
0.0, 0.1
params_down_Q6 = np.array([a_down_Q6, b_down_Q6, rho_down_Q6, m_down_Q6,
sigma_down_Q6])
a_up_Q6, b_up_Q6, rho_up_Q6, m_up_Q6, sigma_up_Q6 = 0.1, 0.1, -0.1, 0.0, 0.1
params_up_Q6 = np.array([a_up_Q6, b_up_Q6, rho_up_Q6, m_up_Q6, sigma_up_Q6])

params_optimized_down_Q6 = nelder_mead(params_down_Q6, cost_function_1,
```

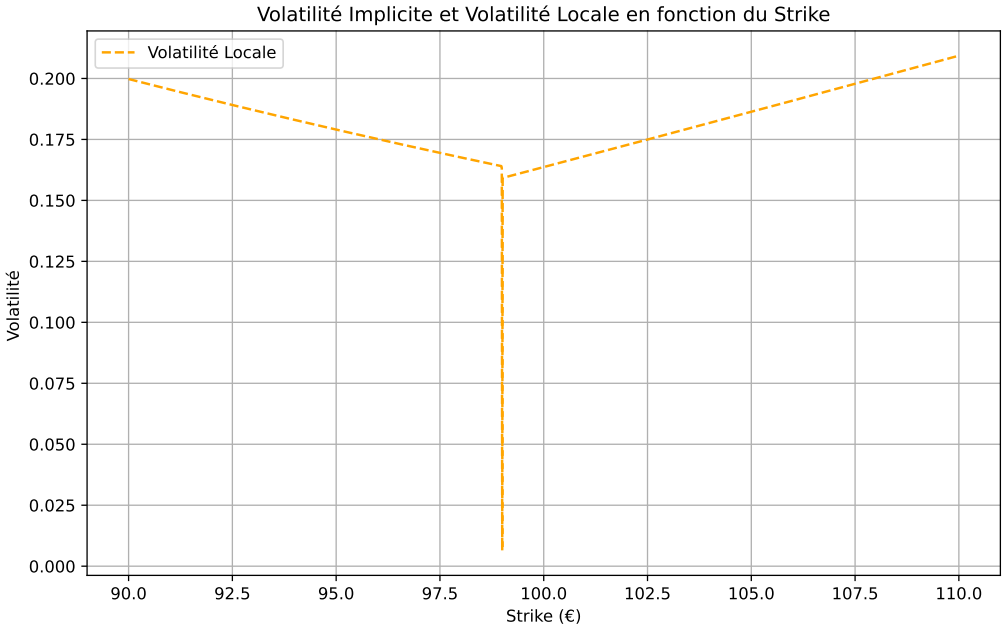
```
log_moneyness_Q6, implied_vols_down_Q6)
params_optimized_up_Q6 = nelder_mead(params_up_Q6, cost_function_1,
log_moneyness_Q6, implied_vols_up_Q6)

svi_vols_up_Q6 = svi_model(log_moneyness, a_up_Q6, b_up_Q6, rho_up_Q6,
m_up_Q6, sigma_up_Q6)
svi_vols_down_Q6 = svi_model(log_moneyness, a_down_Q6, b_down_Q6,
rho_down_Q6, m_down_Q6, sigma_down_Q6)

price_strikes_down_Q6,local_volatility_down_Q6
=dupire_local_volatility(a_down_Q6, b_down_Q6, rho_down_Q6, m_down_Q6,
sigma_down_Q6)
price_strikes_up_Q6,local_volatility_up_Q6 =dupire_local_volatility(a_up_Q6,
b_up_Q6, rho_up_Q6, m_up_Q6, sigma_up_Q6)
```

```
Paramètres calibrés avec Nelder-Mead pour prices down : a=0.161801815206521,
b=0.20070590934081872, rho=0.12286581970697258, m=-0.012048339971123881,
sigma=-1.4039450794395113e-10
Paramètres calibrés avec Nelder-Mead pour prices up : a=0.017991308779075278,
b=0.26186284013588157, rho=1.900052510000962, m=-0.19611921440673458,
sigma=-3.108824537630415e-09
```

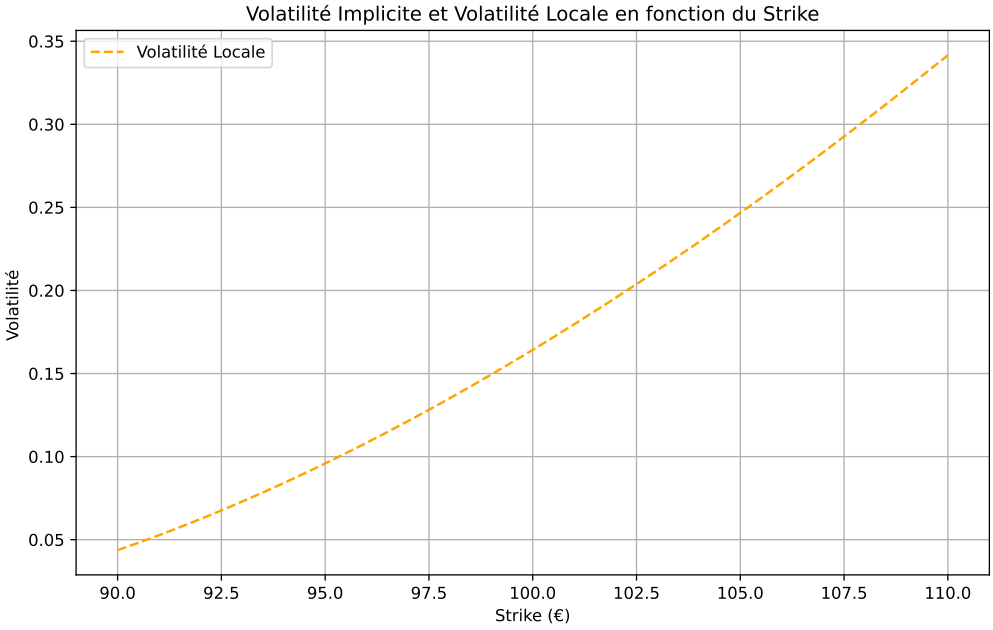


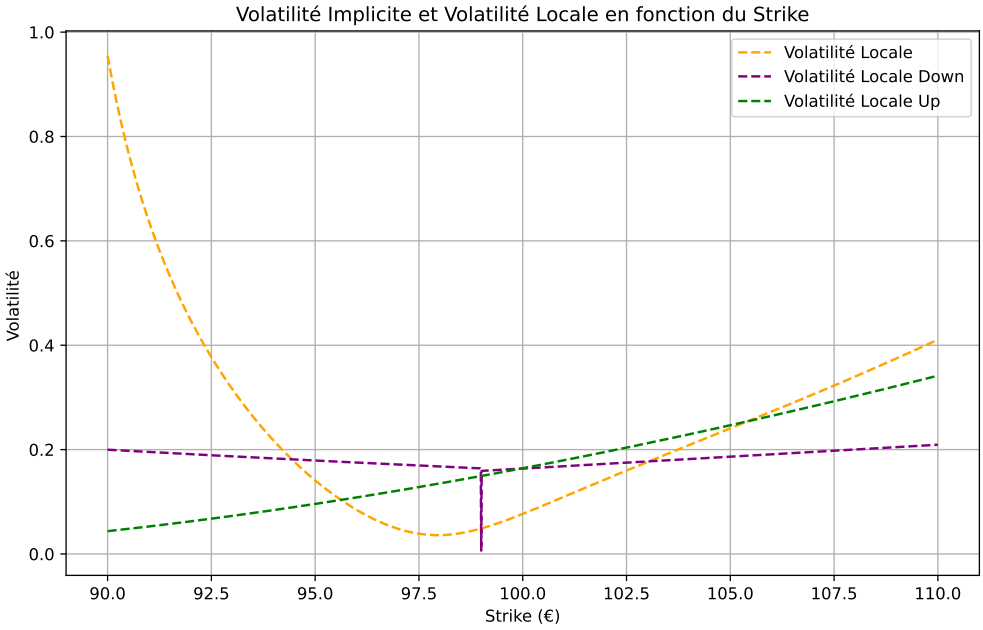
La courbe orange (stress up) est satisfaisante car pour les strike autour de la monnaie, les estimations sont proches des observations (le prix de l'option strike 100 ayant augmenté). En revanche la courbe verte (stress down est moins satisfaisante car ne matche pas les observations. Il est donc possible qu'il nous manque une condition d'arbitrage limitant le stress down.





Le graph ci-dessus représente la vol locale avec le strike 100 stress down. On remarque notamment un pic qui est dû à une discontinuité dans le prix de l'option (graph au dessus) causant un dirac dans la dérivée première et renforçant ainsi notre hypothèse précédente sur un manquement de condition d'arbitrage.


Volatilité Implicite et Volatilité Locale en fonction du Strike
Volatilité Locale
Volatilité
Strike (€)
0.35
0.30
0.25
0.20
0.15
0.10
0.05
90.0
92.5
95.0
97.5
100.0
102.5
105.0
107.5
110.0


Volatilité Implicite et Volatilité Locale en fonction du Strike
Volatilité Locale
Volatilité Locale Down
Volatilité Locale Up
Volatilité
Strike (€)
1.0
0.8
0.6
0.4
0.2
0.0
90.0
92.5
95.0
97.5
100.0
102.5
105.0
107.5
110.0

2 Stochastic volatility

2.1 Heston model

Listing 13 – Heston

```
def heston_cf(u, spot, var0, kappa, theta, sigma, rho, tau, rate):
  i  = 1j
  a  = kappa * theta
  rspi = rho * sigma * u * i

  dval = np.sqrt((rspi - kappa)**2 + (u*i + u**2)*sigma**2)
  gval = (kappa - rspi + dval)/(kappa - rspi - dval)

  factor1 = np.exp(rate * u * i * tau)
  factor2 = spot**(u * i) * ((1 - gval * np.exp(dval * tau))
                             / (1 - gval))**(-2*a/sigma**2)

  factor3 = np.exp(a * tau * (kappa - rspi + dval)/sigma**2
                     + var0 * (kappa - rspi + dval)*(1 - np.exp(dval*tau))
                       /(sigma**2 * (1 - gval*np.exp(dval*tau))))

  return factor1 * factor2 * factor3

def heston_integrand(u, spot, var0, kappa, theta, sigma, rho, tau, rate,
strike):
  i = 1j
  params = (spot, var0, kappa, theta, sigma, rho, tau, rate)

  num   = math.exp(-rate * tau) * heston_cf(u - i, *params) \
                  - strike * heston_cf(u, *params)
  denom       = i * u * (strike**(i*u))

  return (num / denom).real

def heston_call_price(spot, var0, kappa, theta, sigma, rho, tau, rate,
strike):
    pars_for_integrand = (spot, var0, kappa, theta, sigma, rho, tau, rate,
    strike)
    val, _ = quad(heston_integrand, 0, 100, args=pars_for_integrand,
    limit=200)
    return (spot - strike * math.exp(-rate * tau))/2.0 + val / math.pi

df = pd.DataFrame({'Strikes': strike, 'Prices': prices})

# Parametres Heston
kappa = 2.0
theta = 0.05
sigma = 0.3
rho   = -0.5
var0  = 0.05

# Calcul des prix
list_heston = []
for K in strike:
    c_heston = heston_call_price(
        spot=S, var0=var0,
        kappa=kappa, theta=theta, sigma=sigma, rho=rho,
        tau=T, rate=rf,
        strike=K
    )
    list_heston.append(c_heston)
```

Strike	Price	Heston Price
95	10.93	11.411657
96	9.55	10.834906
97	8.28	10.276592
98	7.40	9.736846
99	6.86	9.215754
100	6.58	8.713365
101	6.52	8.229681
102	6.49	7.764664
103	6.47	7.318232
104	6.46	6.890261

Listing 14 – Heston

```
def simulate_heston_paths(S0, v0, kappa, theta, sigma, rho, r,
                          T=1.0, n_paths=10000, n_steps=200):

    dt = T / n_steps

    S = np.full(n_paths, S0, dtype=float)
    V = np.full(n_paths, v0, dtype=float)

    cov_2x2 = np.array([
        [1.0, rho],
        [rho, 1.0]
    ])
    L = np.linalg.cholesky(cov_2x2)

    for step in range(n_steps):
        Z = np.random.randn(n_paths, 2)
        Zcorr = Z @ L.T  # shape (n_paths,2)

        dW1 = Zcorr[:,0]*math.sqrt(dt)
        dW2 = Zcorr[:,1]*math.sqrt(dt)

        Vnew = V + kappa*(theta - V)*dt + sigma*np.sqrt(np.maximum(V,0.0))*dW2
        Vnew = np.maximum(Vnew, 0.0)

        Snew = S + r*S*dt + np.sqrt(np.maximum(V,0.0))*S*dW1
        Snew = np.maximum(Snew, 1e-12)

        S = Snew
        V = Vnew

    return S


def heston_mc_call_price(S0, v0, kappa, theta, sigma, rho, r,
                         T, strike, n_paths=10000, n_steps=200):

    S_T = simulate_heston_paths(S0, v0, kappa, theta, sigma, rho, r,
                                T=T, n_paths=n_paths, n_steps=n_steps)
    payoff = np.maximum(S_T - strike, 0.0)
    return math.exp(-r*T)*np.mean(payoff)

def cost_heston_mc(params, strikes, market_prices):

    global S, r, T

    kappa, theta, sigma, rho, v0 = params

    if kappa<=1e-8 or theta<=1e-8 or sigma<=1e-8 or v0<=1e-8 or abs(rho)>=1.0:
```

```
        return 1e10

    n_paths = 3000
    n_steps = 100

    sse = 0.0
    for K, obs in zip(strikes, market_prices):
        c_mc = heston_mc_call_price(S, v0, kappa, theta, sigma, rho, r,
                                    T, K, n_paths, n_steps)
        diff = c_mc - obs
        sse += diff*diff
    return sse

globals()['S'] = S
globals()['r'] = rf
globals()['T'] = T

kappa, theta, sigma, rho, v0 = 2.0, 0.05, 0.3, -0.5, 0.05
initial_guess = [kappa, theta, sigma, rho, v0]
best_params = nelder_mead(initial_guess, cost_function=cost_heston_mc,
x_data=strike, y_data=prices, tol=1e-4, max_iter=100)

cost = cost_heston_mc(best_params, strike, prices)
```

```
Best params : kappa=2.0381789656939375, theta=0.024181204960624814,
sigma=0.3507685522039328, rho=-0.5335167688297956, v0=0.05108089859675349
Cost final : 5.405966192609224
```

Les résultats montrent que les prix calculés avec le modèle de Heston sont globalement proches des prix observés sur le marché. Cela peut indiquer une légère différence dans la dynamique de la volatilité par rapport aux attentes du marché. Le coût final obtenu après calibration est de 5.41, ce qui traduit un bon ajustement du modèle aux données du marché, bien que des améliorations puissent être envisagées pour affiner l'estimation des paramètres.

2.2 Heston model calibration

Listing 15 – Heston

```
idx_atm = [i for i,k in enumerate(strike) if k in [99,100,101]]
st_atm  = [strike[i] for i in idx_atm]
pr_atm  = [prices[i]  for i in idx_atm]

best_params_atm = nelder_mead(initial_guess, cost_function=cost_heston_mc,
x_data=st_atm, y_data=pr_atm, tol=1e-4, max_iter=100)
cost_atm = cost_heston_mc(best_params_atm, st_atm, pr_atm)
```

```
Best params (ATM) : kappa=2.0227142534135725, theta=0.019381505796743494,
sigma=0.36814372382109994, rho=-0.540652662888111, v0=0.04855914097123516
Cost final (ATM) = 0.31151790738403723
```

Les résultats de la calibration du modèle de Heston sur les options at-the-money (ATM) montrent un ajustement beaucoup plus précis que sur l'ensemble des strikes, avec un coût final de seulement 0.31 contre 5.41 précédemment. Cela indique que le modèle s'adapte particulièrement bien aux prix des options situées autour du prix d'exercice de 100

2.3 Heston with stress

Listing 16 – Heston

```
df_mod = df.copy()
idx_100 = df_mod.index[df_mod['Strikes']==100][0]
df_mod.loc[idx_100,'Prices'] = 6.7

best_params_mod = nelder_mead(initial_guess, cost_function=cost_heston_mc,
x_data=df_mod['Strikes'].values, y_data=df_mod['Prices'].values, tol=1e-4,
max_iter=100)

cost_mod = cost_heston_mc(best_params_mod, df_mod['Strikes'].values,
df_mod['Prices'].values)

df_mod = df.copy()
idx_100 = df_mod.index[df_mod['Strikes']==100][0]
df_mod.loc[idx_100,'Prices'] = 6.54

best_params_mod = nelder_mead(initial_guess, cost_function=cost_heston_mc,
x_data=df_mod['Strikes'].values, y_data=df_mod['Prices'].values, tol=1e-4,
max_iter=100)

cost_mod = cost_heston_mc(best_params_mod, df_mod['Strikes'].values,
df_mod['Prices'].values)
```

```
Best params (price stressed up) : kappa=2.018100190689598,
theta=0.03288767794609475, sigma=0.35018952154602423,
rho=-0.4490007519880994, v0=0.03768776024380841
Cost final (price streesed up) = 3.826796069553655
Best params (price stressed down) : kappa=2.039991266810284,
theta=0.031129247570301496, sigma=0.33983165889863665,
rho=-0.4872256898640667, v0=0.039374960310151454
Cost final (price stressed down) = 6.053537211820535
```

Les résultats de la calibration du modèle de Heston sous stress sur le prix du strike 100 montrent des variations intéressantes des paramètres et des coûts d'ajustement. Lorsque le prix est augmenté à 6.7, le coût final est 3.83, indiquant un meilleur ajustement que dans le cas de base (5.41). En revanche, lorsque le prix est diminué à 6.54, le coût final grimpe à 6.05, traduisant un ajustement plus difficile du modèle.