Group member:

Course Project: An Arbitrage-Free Smile Interpolator

Project report should be submitted as a Jupyter notebook (.ipynb). Each project group should have 1 to 3 members. The report should be written based on this notebook. Please make your formulas and code, test results, observations, intermediate results, improvements made/suggested (optional), and conclusions clear.

Objectives

- Implement an arbitrage free smile interpolator SmileAF.
- Use the arbitrage free smile interpolator to construct local volatility model
- Use PDE with local volatility model to price a given set of European options (strike in delta $\times$ maturity)
- Compare the price errors of arbitrage-free smile interpolator and the cubic spline smile interpolator
- Open questions for bonus points: improve the algorithms in terms of precision and efficiency (smile interpolation, implied vol, local vol, PDE, calibration report). One area to consider for improvement is the construction of the tails (wings).

Smile Arbitrage

1. European call prices are monotonically decreasing with respect to the strike:

$$C(S_0, K_1, T, \sigma(K_1), r, q) \geq C(S_0, K_2, T, \sigma(K_2), r, q) \text{ for } K_1 < K_2 \tag{1}$$

1. The European call price as a function of strike has to be convex every where: for any three points $K_1 < K_2 < K_3$

$$\frac{C(K_2) - C(K_1)}{K_2 - K_1} < \frac{C(K_3) - C(K_2)}{K_3 - K_2}$$

or

$$C(K_2) < C(K_3)\frac{K_2 - K_1}{K_3 - K_1} + C(K_1)\frac{K_3 - K_2}{K_3 - K_1} \tag{2}$$

This is also equivalent to "butterfly price has to be non-negative".

When Could Smile Arbitrage Happen?

The undiscounted call price is the expectation of payoff under risk neutral measure

$$C(K) = E[\max(S - K, 0)]$$

And expectation is an integral over the probability density function $p(s)$

$$C(K) = \int_K^{+\infty} (s - K)p(s)ds$$

The 1st non-arbitrage condition translates to

$$C(K_1) - C(K_2) = \left[\int_{K_1}^{K_2} (s - K_1)p(s)ds + \int_{K_2}^{+\infty} (K_2 - K_1)p(s)ds\right]$$

which is positive by definition if $K_2 > K_1$.

The 2nd non-arbitrage condition translates to

$$\begin{aligned} &C(K_3)\frac{K_2 - K_1}{K_3 - K_1} + C(K_1)\frac{K_3 - K_2}{K_3 - K_1} - C(K_2) \\ =&\frac{K_3 - K_2}{K_3 - K_1}\int_{K_1}^{K_2} (s - K_1)p(s)ds + \frac{K_2 - K_1}{K_3 - K_1}\int_{K_2}^{K_3} (K_3 - s)p(s)ds \end{aligned}$$

which is also positive by definition if $K_3 > K_2 > K_1$.

So, when could smile arbitrage happen? **When the probability density does not exist**. If we can start with valid probability density function $p(s)$, arbitrage-freeness is guaranteed by construction.

Arbitrage Free Smile (Based on [Fengler 2009])

- We consider smile construction for a given expiry T.
- Start with N discrete sample strike points

$$\vec{k} = [k_1, k_2, \ldots, k_N]^\top$$

- Try to solve for undiscounted call prices for these N sample points

$$\vec{c} = [c_1, c_2, \dots, c_N]^\top$$

- For the undiscounted call price $C(K)$ for any K, we can interpolate using cubic spline over the sample points (k_i, c_i). (Note that we are using cubic spline to interpolate the prices, not volatility)

- The second derivative of call price with respect to strike is the probability density function:

$$\frac{dC}{dK} = d\frac{\int_K^\infty Sp(S)dS}{dK} - d\frac{K\int_K^\infty p(S)dS}{dK} = -Kp(K) - \left(\int_K^\infty p(S)dS - Kp(K)\right) = -\int_K^\infty p(S)dS$$
$$\frac{d^2C}{dK^2} = p(K)$$

So c_i'' is probability density function at k_i, we denote it as p_i

- Second derivatives in cubic spline interpolation form line segments. Cubic spline on $C(K)$, means linearly interpolate on probability density. If p_i are all positive, the whole pdf is positive by construction --- **no smile arbitrage**.

- For tails --- call prices are almost linear if strike is very far away from spot, we can use **natural cubic spline**: $p_1 = p_N = 0$.

- Our problem is to solve for $[c_1, c_2, \dots, c_N, p_2, \dots, p_{N-1}]$

Inputs to our problem

Same as our Cubic Spline smile interpolator, we have the input marks to start with to construct the Arb-Free(AF) smile interpolator:

- **Marks**: strike to volatility pairs, denote as $(\hat{k}_j, \sigma_j)$, for $j \in [1, 2, \dots, M]$. In our case, $M = 5$.

We would like to match the marks exactly. And we cannot directly construct a cubic spline using the M points --- too coarse and distribution is not realistic.

Problem Definition

- We use $N = 50$ sample points, ranging from $[k_1 = Se^{(r_d - r_f)T - \frac{1}{2}\sigma_{ATM}^2 T - 5\sigma_{ATM}\sqrt{T}}, k_N = Se^{(r_d - r_f)T - \frac{1}{2}\sigma_{ATM}^2 T + 5\sigma_{ATM}\sqrt{T}}]$, i.e., ± 5 standard deviation based on σ_{ATM}.
- σ_{ATM} is implied volatility of the middle point of the input marks.

- We also assume the strike of the middle point of the input marks is the forward --- ATM forward convention.
- The sample points are equally spaced, denote the length of the segment $u = \frac{k_N - k_1}{N-1}$
- We would like the call prices to be as smooth as possible --- minimize the change of the slopes
- We want to match exactly the M input marks.
- This is a constrained optimization problem.

Constraints

- Cubic spline interpolation imposes the constraints that the left and right first derivative of a point have to match, it can be derived by matching the first derivative of the left and right segments for point i we have the condition

$$c_{i+1} + c_{i-1} - 2c_i = (\frac{2}{3}p_i + \frac{1}{6}p_{i+1} + \frac{1}{6}p_{i-1})u^2$$

The cubic spline constraints translate to the linear system

$$\underbrace{\begin{pmatrix} 1 & -2 & 1 & 0 & \dots & 0 \\ 0 & 1 & -2 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 1 & -2 & 1 \end{pmatrix}}_{\vec{Q}_{(N-2)\times N}} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{pmatrix} = u^2 \underbrace{\begin{pmatrix} \frac{2}{3} & \frac{1}{6} & 0 & \dots & 0 \\ \frac{1}{6} & \frac{2}{3} & \frac{1}{6} & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \\ 0 & \dots & 0 & \frac{1}{6} & \frac{2}{3} \end{pmatrix}}_{\vec{R}_{(N-2)\times(N-2)}} \begin{pmatrix} p_2 \\ p_3 \\ \vdots \\ p_{N-1} \end{pmatrix}$$

If we define

$$\vec{x} = \begin{pmatrix} \vec{c}^{\top} \\ \vec{p}^{\top} \end{pmatrix}, \quad \vec{A} = (\vec{Q}, -\vec{R})$$

we can represent the constraint as:

$$\vec{A}x = \vec{0} \quad \textbf{--- Constraint 1}$$

- The call prices at the input marks $\hat{k}_j, j \in [1, 2, \dots, M]$ can be represented by cubic spline interpolation

$$C(\hat{k}_j) = ac_i + bc_{i+1} + \frac{(a^3-a)u^2}{6}p_i + \frac{(b^3-b)u^2}{6}p_{i+1} \quad \textbf{--- Constraint 2}$$

where

$$a = \frac{k_{i+1} - \hat{k}_j}{u}, \quad b = 1 - a$$

and $[k_i, k_{i+1}]$ here represents the segment that $\hat{k}_j$ falls in.

- p_i are densities, so

$$p_i > 0 \quad \textbf{--- Constraint 3}$$

- Integrating the density function we should get 1.0 (recall that density function are linearly interpolated)

$$u\sum p_i = 1.0 \quad \textbf{--- Constraint 4}$$

- Natural cubic spline, p_1 and p_N are zero, so we could solve directly c_1 and c_N

$$c_1 = Se^{(r_d - r_f)T} - k_1, \quad c_N = 0 \quad \textbf{--- Constraint 5}$$

- Call prices are monotonically decreasing:

$$c_{i+1} - c_i \leq 0 \quad \text{for } i \in \{1, 2, \ldots, N-1\} \quad \textbf{--- Constraint 6}$$

Objective Function

- Fill the rest of the DOF using objective function (soft constraints)
- [Fengler 2009] tried minimizing the below to achieve smoothness on p:

$$\int_{k_1}^{k_N} p(S)^2 dS = \text{constant} \times \vec{p}^{\top} \vec{R} \vec{p}$$

Using $\vec{x}$ as variable and define

$$\vec{H}_{(2N-2)\times(2N-2)} = \begin{pmatrix} \vec{0} & \vec{0} \\ \vec{0} & \vec{R}_{(N-2)\times(N-2)} \end{pmatrix}$$

the problem becomes minimizing

$$\vec{x}^\top \vec{H} \vec{x}$$

Problem Formulation

We can formulate our problem as

$$\min \quad \vec{x}^\top \vec{H} \vec{x}$$

subject to constraints 1 to 5.

- All the constraints are linear function of $\vec{x}$
- Our objective function is quadratic and the matrix $\vec{H}$ is positive semi-definite
- Global solution exists, and (relatively) efficient to solve

Tips

- To solve the quadratic programming problem, we can use the CVXOPT package:
  - http://cvxopt.org/examples/tutorial/qp.html
  - https://buildmedia.readthedocs.org/media/pdf/cvxopt/dev/cvxopt.pdf
- Write down the exact formulas using the same symbols used by CVXOPT QP problem's documentation in the above docs, then translate them into code. This will make debugging easier.
- To check whether solver's result makes sense, examine if the constraints are satisified, and if the call prices are smooth and match the input.
- If test run takes too long, reduce the number of grid points in PDE pricer, or skip the calibration report and inspect the volatility surface first.
- It might be easier to plot implied vol, call prices, PDF, and the marks to check the result.
- use bisect.bisect_left to find the bucket $\hat{k}$ belongs to (https://docs.python.org/3/library/bisect.html)

References

[Fengler 2009] Arbitrage-free smoothing of the implied volatility surface, Quantitative Finance, 2009

Implementation

In [ ]:

Below are some building blocks for the project. You contribution should be in SmileAF class. You can modify any other classes or methods. If you do so, please describe your modification in the project report.

In [4]:
```
import math
from enum import Enum
from scipy import optimize
import bisect
from scipy.interpolate import CubicSpline
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

class PayoffType(Enum):
    Call = 0
    Put = 1

def cnorm(x):
    return (1.0 + math.erf(x / math.sqrt(2.0))) / 2.0

def fwdDelta(fwd, stdev, strike, payoffType):
    d1 = math.log(fwd / strike) / stdev + stdev / 2
    if payoffType == PayoffType.Call:
        return cnorm(d1)
    elif payoffType == PayoffType.Put:
        return - cnorm(-d1)
    else:
        raise Exception("not supported payoff type", payoffType)

# solve for the K such that Delta(S, T, K, vol) = delta
def strikeFromDelta(S, r, q, T, vol, delta, payoffType):
    fwd = S * math.exp((r-q) * T)
    if payoffType == PayoffType.Put:
        delta = -delta
    f = lambda K: (fwdDelta(fwd, vol * math.sqrt(T), K, payoffType) - delta)
    a, b = 0.0001, 10000
    return optimize.brentq(f, a, b)

class ImpliedVol:
    def __init__(self, ts, smiles):
```

```
        self.ts = ts
        self.smiles = smiles
    # linear interpolation in variance, along the strike line
    def Vol(self, t, k):
        # locate the interval t is in
        pos = bisect.bisect_left(self.ts, t)
        # if t is on or in front of first pillar,
        if pos == 0:
            return self.smiles[0].Vol(k)
        if pos >= len(self.ts):
            return self.smiles[-1].Vol(k)
        else:  # in between two brackets
            prevVol, prevT = self.smiles[pos-1].Vol(k), self.ts[pos-1]
            nextVol, nextT = self.smiles[pos].Vol(k), self.ts[pos]
            w = (nextT - t) / (nextT - prevT)
            prevVar = prevVol * prevVol * prevT
            nextVar = nextVol * nextVol * nextT
            return  math.sqrt((w * prevVar + (1-w) * nextVar)/t)
        return

    def dVoldK(self, t, k):
        return (self.Vol(t, k+0.001) - self.Vol(t, k-0.001)) / 0.002
    def dVoldT(self, t, k):
        return (self.Vol(t+0.005, k) - self.Vol(t, k)) / 0.005
    def dVol2dK2(self, t, k):
        return (self.Vol(t, k+0.001) + self.Vol(t, k-0.001) - 2*self.Vol(t, k)) / 0.000001

class LocalVol:
    def __init__(self, iv, S0, rd, rf):
        self.iv = iv
        self.S0 = S0
        self.rd = rd
        self.rf = rf
    def LV(self, t, s):
        if t < 1e-6:
            return self.iv.Vol(t, s)
        imp = self.iv.Vol(t, s)
        dvdk = self.iv.dVoldK(t, s)
        dvdt = self.iv.dVoldT(t, s)
        d2vdk2 = self.iv.dVol2dK2(t, s)
        d1 = (math.log(self.S0/s) + (self.rd-self.rf)*t + imp * imp * t / 2) / imp / math.sqrt(t)
        numerator = imp*imp + 2*t*imp*dvdt + 2*(self.rd-self.rf)*s*t*imp*dvdk
        denominator = (1+s*d1*math.sqrt(t)*dvdk)**2 + s*s*t*imp*(d2vdk2 - d1 * math.sqrt(t) * dvdk * dvdk)
        localvar = min(max(numerator / denominator, 1e-8), 1.0)
```

```
        if numerator < 0: # floor local volatility
            localvar = 1e-8
        if denominator < 0: # cap local volatility
            localvar = 1.0
        return math.sqrt(localvar)

class EuropeanOption():
    def __init__(self, assetName, expiry, strike, payoffType):
        self.assetName = assetName
        self.expiry = expiry
        self.strike = strike
        self.payoffType = payoffType
    def payoff(self, S):
        if self.payoffType == PayoffType.Call:
            return max(S - self.strike, 0)
        elif self.payoffType == PayoffType.Put:
            return max(self.strike - S, 0)
        else:
            raise Exception("payoffType not supported: ", self.payoffType)
    def valueAtNode(self, t, S, continuation):
        if continuation == None:
            return self.payoff(S)
        else:
            return continuation

# Black-Scholes analytic pricer
def bsPrice(S, r, q, vol, T, strike, payoffType):
    fwd = S * math.exp((r-q) * T)
    stdev = vol * math.sqrt(T)
    d1 = math.log(fwd / strike) / stdev + stdev / 2
    d2 = d1 - stdev
    if payoffType == PayoffType.Call:
        return math.exp(-r * T) * (fwd * cnorm(d1) - cnorm(d2) * strike)
    elif payoffType == PayoffType.Put:
        return math.exp(-r * T) * (strike * cnorm(-d2) - cnorm(-d1) * fwd)
    else:
        raise Exception("not supported payoff type", payoffType)

# PDE pricer with local volatility
def pdePricerX(S0, r, q, lv, NX, NT, w, trade):
    # set up pde grid
    mu = r - q
    T = trade.expiry
    X0 = math.log(S0)
```

```
    vol0 = lv.LV(0, S0)
    srange = 5 * vol0 * math.sqrt(T)
    maxX = X0 + (mu - vol0 * vol0 * 0.5)*T + srange
    minX = X0 - (mu - vol0 * vol0 * 0.5)*T - srange
    dt = T / (NT-1)
    dx = (maxX - minX) / (NX-1)
    # set up spot grid
    xGrid = np.array([minX + i*dx for i in range(NX)])
    # initialize the payoff
    ps = np.array([trade.payoff(math.exp(x)) for x in xGrid])
    # backward induction
    for j in range(1, NT):
        # set up the matrix, for LV we need to update it for each iteration
        M = np.zeros((NX, NX))
        D = np.zeros((NX, NX))
        for i in range(1, NX - 1):
            vol = lv.LV(j*dt, math.exp(xGrid[i]))
            M[i, i - 1] = (mu - vol * vol / 2.0) / 2.0 / dx - vol * vol / 2 / dx / dx
            M[i, i] = r + vol * vol / dx / dx
            M[i, i + 1] = -(mu - vol * vol / 2.0) / 2.0 / dx - vol * vol / 2 / dx / dx
            D[i, i] = 1.0
        # the first row and last row depends on the boundary condition
        M[0, 0], M[NX - 1, NX - 1] = 1.0, 1.0
        rhsM = (D - dt * M) * w + (1 - w) * np.identity(NX)
        lhsM = w * np.identity(NX) + (D + dt * M) * (1 - w)
        inv = np.linalg.inv(lhsM)

        ps = rhsM.dot(ps)
        ps[0] = dt*math.exp(-r*j*dt) * trade.payoff(math.exp(xGrid[0])) # discounted payoff
        ps[NX-1] = dt*math.exp(-r*j*dt) * trade.payoff(math.exp(xGrid[NX-1]))
        ps = inv.dot(ps)
    # linear interpolate the price at S0
    return np.interp(X0, xGrid, ps)
```

Below are the smile interpolators and smile constructor. You need to implement SmileAF. Note that smileFromMarks takes a parameter smileInterpMethod. When it is 'AF', SmileAF is used.

In [5]:
```
class SmileAF:
    def __init__(self, strikes, vols, T):
        self.atmvol = vols[int(len(vols)/2)]
        self.fwd = strikes[int(len(strikes)/2)]
        self.T = T
        self.N = 50
```

```
        stdev = self.atmvol * math.sqrt(T)
        kmin = self.fwd * math.exp(-0.5*stdev*stdev-5 * stdev)
        kmax = self.fwd * math.exp(-0.5*stdev*stdev+5 * stdev)
        u = (kmax - kmin) / (self.N - 1)
        self.ks = [kmin + u * i for i in range(0, self.N)]
        self.cs = np.zeros(self.N)  # undiscounted call option prices
        self.ps = np.zeros(self.N)  # densities
        self.u = u
        # now we need to construct our constrained optimization problem to solve for cs and ps
        # ... YOUR CODE HERE ... to solve for self.cs and self.ps
        # ...

        # now we obtained cs and ps, we do not interpolate for price for any k and imply the vol,
        # since at the tails the price to vol gradient is too low and is numerically not stable.
        # Instead, we imply the volatilities for all points between put 10 delta and call 10 delta input points
        # then we make the vol flat at the wings by setting the vols at kmin and kmax,
        # we then construct a cubic spline interpolator on the dense set of volatilities so that it's C2
        # and faster then implying volatilities on the fly.
        # note that this treatment of tail is simplified. It could also introduce arbitrage.
        # In practice, the tails should be calibrated to a certain distribution.
        def implyVol(k, prc, v):
            stdev = v * math.sqrt(self.T)
            d1 = (math.log(self.fwd / k)) / stdev + 0.5 * stdev
            d2 = (math.log(self.fwd / k)) / stdev - 0.5 * stdev
            return self.fwd * cnorm(d1) - k * cnorm(d2) - prc
        khmin = bisect.bisect_left(self.ks, strikes[0])
        khmax = bisect.bisect_right(self.ks, strikes[len(strikes)-1])
        kks = [0] * ((khmax+1) - (khmin-1) + 2)
        vs = [0] * ((khmax+1) - (khmin-1) + 2)
        for i in range(khmin-1, khmax+1):
            prc = self.Price(self.ks[i])
            f = lambda v: implyVol(self.ks[i], prc, v)
            a, b = 1e-8, 10
            vs[i - (khmin-1) + 1] = optimize.brentq(f, a, b)
            kks[i - (khmin-1) + 1] = self.ks[i]
        kks[0] = kmin
        vs[0] = vs[1]
        kks[len(kks)-1] = kmax
        vs[len(vs)-1] = vs[len(vs)-2]

        self.vs = vs
        self.cubicVol = CubicSpline(kks, vs, bc_type=((1, 0.0), (1, 0.0)), extrapolate=True)

    def Vol(self, k):
```

```
        if k < self.ks[0]:  # scipy cubicspline bc_type confusing, extrapolate by ourselfs
            return self.vs[0]
        if k > self.ks[-1]:
            return self.vs[-1]
        else:
            return self.cubicVol(k)

    # undiscounted call price - given cs and ps,
    # we can obtain undiscounted call price for any k via cubic spline interpolation
    def Price(self, k):
        if k <= self.ks[0]:
            return self.fwd - k
        if k >= self.ks[self.N-1]:
            return 0.0
        pos = bisect.bisect_left(self.ks, k)
        a = (self.ks[pos] - k) / self.u
        b = 1 - a
        c = (a * a * a - a) * self.u * self.u / 6.0
        d = (b * b * b - b) * self.u * self.u / 6.0
        return a * self.cs[pos-1] + b * self.cs[pos] + c*self.ps[pos-1] + d*self.ps[pos]

class SmileCubicSpline:
    def __init__(self, strikes, vols):
        # add additional point on the right to avoid arbitrage
        self.strikes = strikes + [1.1 * strikes[-1] - 0.1 * strikes[-2]]
        self.vols = vols + [vols[-1] + (vols[-1] - vols[-2]) / 10]
        self.cs = CubicSpline(strikes, vols, bc_type=((1, 0.0), (1, 0.0)), extrapolate=True)

    def Vol(self, k):
        if k < self.strikes[0]:  # scipy cubicspline bc_type confusing, extrapolate by ourselfs
            return self.vols[0]
        if k > self.strikes[-1]:
            return self.vols[-1]
        else:
            return self.cs(k)

def smileFromMarks(T, S, r, q, atmvol, bf25, rr25, bf10, rr10, smileInterpMethod):
    c25 = bf25 + atmvol + rr25/2
    p25 = bf25 + atmvol - rr25/2
    c10 = bf10 + atmvol + rr10/2
    p10 = bf10 + atmvol - rr10/2

    ks = [ strikeFromDelta(S, r, q, T, p10, 0.1, PayoffType.Put),
           strikeFromDelta(S, r, q, T, p25, 0.25, PayoffType.Put),
```

```
            S * math.exp((r-q)*T),
            strikeFromDelta(S, r, q, T, c25, 0.25, PayoffType.Call),
            strikeFromDelta(S, r, q, T, c10, 0.1, PayoffType.Call) ]
    # print(T, ks)
    if smileInterpMethod == "CUBICSPLINE":
        return SmileCubicSpline(ks, [p10, p25, atmvol, c25, c10])
    elif smileInterpMethod == "AF":
        return SmileAF(ks, [p10, p25, atmvol, c25, c10], T)
```

Below is a calibration report that shows the calibration error of local volatility PDE pricer.

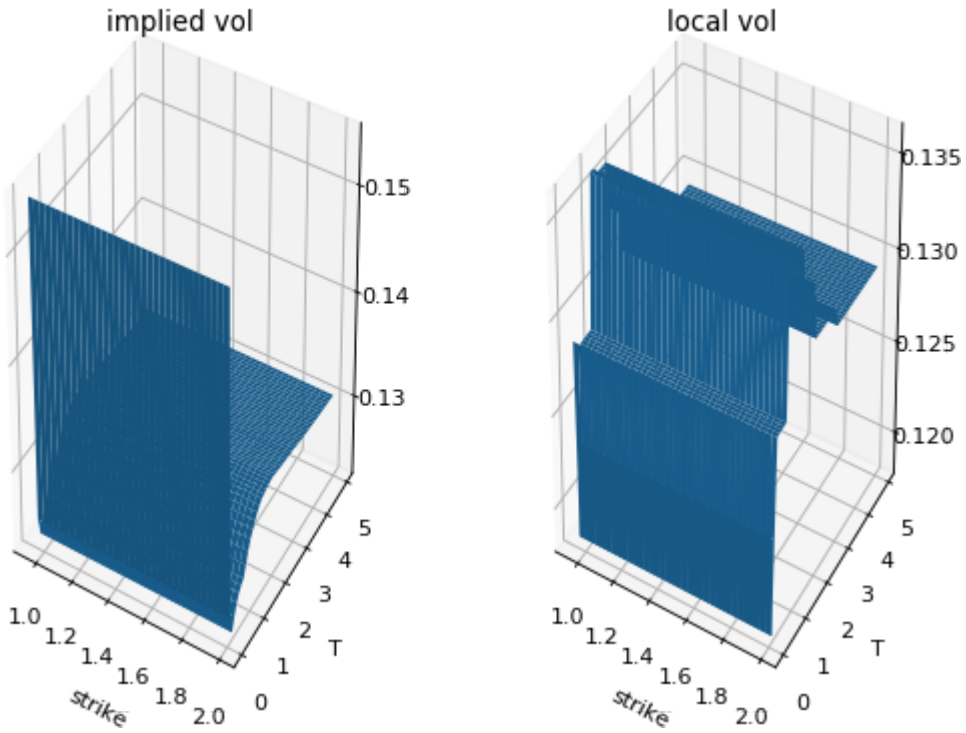
In [6]:
```
import matplotlib.pyplot as plt
import numpy as np

def createTestFlatVol(S, r, q, smileInterpMethod):
    pillars = [0.02, 0.04, 0.06, 0.08, 0.16, 0.25, 0.75, 1.0, 1.5, 2, 3, 5]
    atmvols = [0.155, 0.1395, 0.1304, 0.1280, 0.1230, 0.1230, 0.1265, 0.1290, 0.1313, 0.1318, 0.1313, 0.1305, 0.1295]
    bf25s = np.zeros(len(atmvols))
    rr25s = np.zeros(len(atmvols))
    bf10s = np.zeros(len(atmvols))
    rr10s = np.zeros(len(atmvols))
    smiles = [smileFromMarks(pillars[i], S, r, q, atmvols[i], bf25s[i], rr25s[i], bf10s[i], rr10s[i], smileInterpMethod) for i in r
    return ImpliedVol(pillars, smiles)

def createTestImpliedVol(S, r, q, sc, smileInterpMethod):
    pillars = [0.02, 0.04, 0.06, 0.08, 0.16, 0.25, 0.75, 1.0, 1.5, 2, 3, 5]
    atmvols = [0.155, 0.1395, 0.1304, 0.1280, 0.1230, 0.1230, 0.1265, 0.1290, 0.1313, 0.1318, 0.1313, 0.1305, 0.1295]
    bf25s = [0.0016, 0.0016, 0.0021, 0.0028, 0.0034, 0.0043, 0.0055, 0.0058, 0.0060, 0.0055, 0.0054, 0.0050, 0.0045, 0.0043]
    rr25s = [-0.0065, -0.0110, -0.0143, -0.0180, -0.0238, -0.0288, -0.0331, -0.0344, -0.0349, -0.0340, -0.0335, -0.0330, -0.0330]
    bf10s = [0.0050, 0.0050, 0.0067, 0.0088, 0.0111, 0.0144, 0.0190, 0.0201, 0.0204, 0.0190, 0.0186, 0.0172, 0.0155, 0.0148]
    rr10s = [-0.0111, -0.0187, -0.0248, -0.0315, -0.0439, -0.0518, -0.0627, -0.0652, -0.0662, -0.0646, -0.0636, -0.0627, -0.0627]
    smiles = [smileFromMarks(pillars[i], S, r, q, atmvols[i], bf25s[i]*sc, rr25s[i]*sc, bf10s[i]*sc, rr10s[i]*sc, smileInterpMethod
    return ImpliedVol(pillars, smiles)

def plotTestImpliedVolSurface(S, r, q, iv):
    tStart, tEnd = 0.02, 5
    ts = np.arange(tStart, tEnd, 0.1)
    fwdEnd = S*math.exp((r-q)*tEnd)
    kmin = strikeFromDelta(S, r, q, tEnd, iv.Vol(tEnd, fwdEnd), 0.1, PayoffType.Put)
    kmax = strikeFromDelta(S, r, q, tEnd, iv.Vol(tEnd, fwdEnd), 0.1, PayoffType.Call)
    ks = np.arange(kmin, kmax, 0.01)
    vs = np.ndarray((len(ts), len(ks)))
    lv = LocalVol(iv, S, r, q)
```

```
    lvs = np.ndarray((len(ts), len(ks)))
    for i in range(len(ts)):
        for j in range(len(ks)):
            vs[i, j] = iv.Vol(ts[i], ks[j])
            lvs[i, j] = lv.LV(ts[i], ks[j])
    hf = plt.figure(figsize=(8, 6), dpi=80)
    ha = hf.add_subplot(121, projection='3d')
    hb = hf.add_subplot(122, projection='3d')
    X, Y = np.meshgrid(ks, ts)
    ha.plot_surface(X, Y, vs)
    ha.set_title("implied vol")
    ha.set_xlabel("strike")
    ha.set_ylabel("T")
    hb.plot_surface(X, Y, lvs)
    hb.set_title("local vol")
    hb.set_xlabel("strike")
    hb.set_ylabel("T")
    plt.show()

# the PDE calibration error report takes a implied volatility surface,
# verifies the pricing error of the pde pricer with local volatility surface
def pdeCalibReport(S0, r, q, impliedVol):
    ts = [0.02, 0.04, 0.06, 1/12.0, 1/6.0, 1/4.0, 1/2.0, 1, 2, 5]
    ds = np.arange(0.1, 1.0, 0.1)
    # ds = np.arange(0.5, 1.7, 0.1)
    err = np.zeros((len(ds), len(ts)))
    fig, ax = plt.subplots()

    ax.set_xticks(np.arange(len(ts)))
    ax.set_xlabel("T")
    ax.set_yticks(np.arange(len(ds)))
    ax.set_ylabel("Put Delta")
    ax.set_xticklabels(map(lambda t : round(t, 2), ts))
    ax.set_yticklabels(map(lambda d : round(d, 1), ds))

    # create local vol surface
    lv = LocalVol(impliedVol, S0, r, q)
    # Loop over data dimensions and create text annotations.
    for i in range(len(ds)):
        for j in range(len(ts)):
            T = ts[j]
            K = strikeFromDelta(S0, r, 0, T, impliedVol.Vol(T, S0*math.exp(r*T)), ds[i], PayoffType.Put)
            payoff = PayoffType.Put
            trade = EuropeanOption("ASSET1", T, K, payoff)
```

```
            vol = impliedVol.Vol(ts[j], K)
            bs = bsPrice(S0, r, q, vol, T, K, payoff)
            pde = pdePricerX(S0, r, q, lv, max(50, int(50 * T)), max(50, int(50 * T)), 0.5, trade)
            # normalize error in 1 basis point per 1 unit of stock
            err[i, j] = math.fabs(bs - pde)/S0 * 10000
            ax.text(j, i, round(err[i, j], 1), ha="center", va="center", color="w")
    im = ax.imshow(err)
    ax.set_title("Dupire Calibration PV Error Matrix")
    fig.tight_layout()
    plt.show()
```

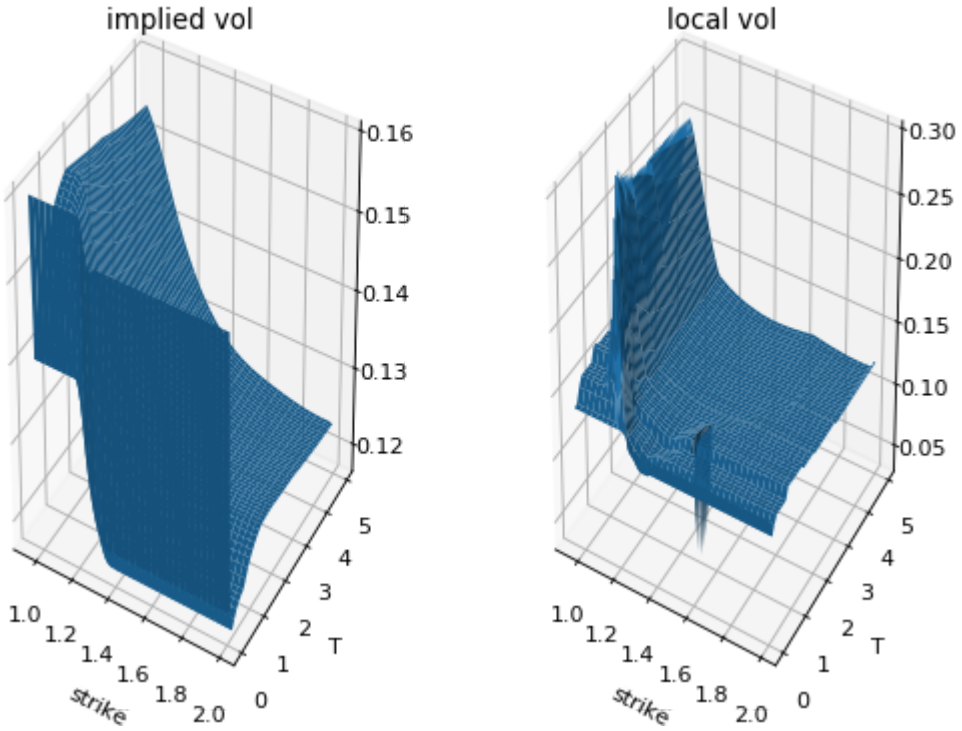
We test with no smile case first. In the calibration error report, we are showing the error in basis points -- 0.01% with respect to 1 notional.

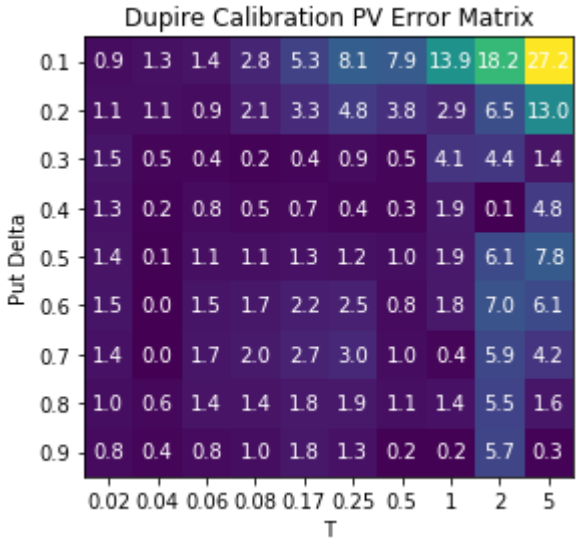
In [7]:
```
S, r, q = 1.25805, 0.01, 0.0
iv = createTestImpliedVol(S, r, q, sc = 0.0, smileInterpMethod='CUBICSPLINE')
plotTestImpliedVolSurface(S, r, q, iv)
pdeCalibReport(S, r, q, iv)
```



Then test smile case with CubicSpline, with a mild smile (tuned by the coeffiicent sc)

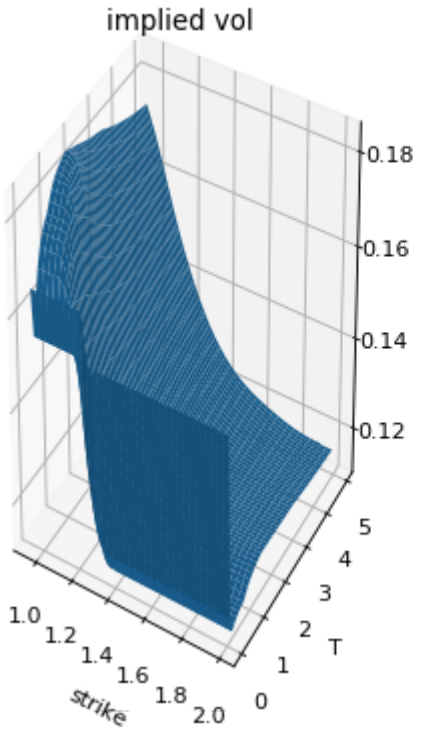
In [ ]:
```
iv = createTestImpliedVol(S, r, q, sc = 0.5, smileInterpMethod='CUBICSPLINE')
plotTestImpliedVolSurface(iv)
pdeCalibReport(S, r, q, iv)
```

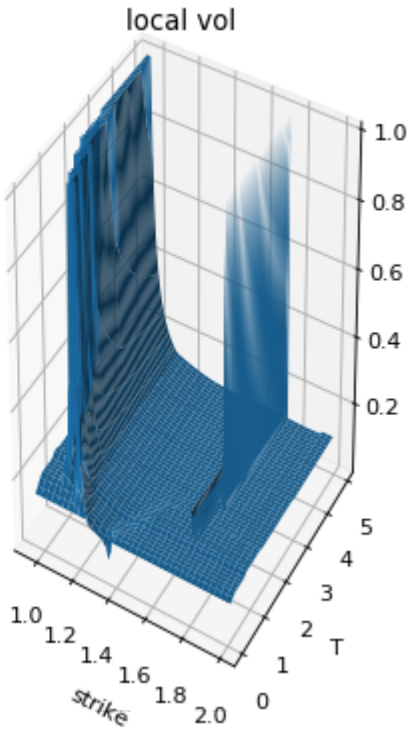


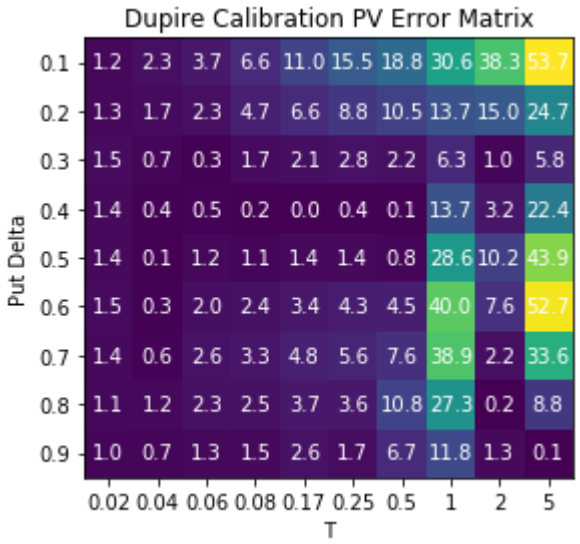


Then test smile case with CubicSpline, with a the input smile (sc = 1.0). It can be seen that the short end low strike region has some smile arbitrage. The calibration errors become larger.

In [ ]:
```
iv = createTestImpliedVol(S, r, q, sc = 1.0, smileInterpMethod='CUBICSPLINE')
plotTestImpliedVolSurface(iv)
pdeCalibReport(S, r, q, iv)
```


implied vol
0.18
0.16
0.14
0.12
1.0
1.2
1.4
1.6
1.8
2.0
strike
0
1
2
3
4
5
T


local vol
1.0
0.8
0.6
0.4
0.2
1.0
1.2
1.4
1.6
1.8
2.0
strike
0
1
2
3
4
5
T


Dupire Calibration PV Error Matrix
Put Delta
0.1 1.2 2.3 3.7 6.6 11.0 15.5 18.8 30.6 38.3 53.7
0.2 1.3 1.7 2.3 4.7 6.6 8.8 10.5 13.7 15.0 24.7
0.3 1.5 0.7 0.3 1.7 2.1 2.8 2.2 6.3 1.0 5.8
0.4 1.4 0.4 0.5 0.2 0.0 0.4 0.1 13.7 3.2 22.4
0.5 1.4 0.1 1.2 1.1 1.4 1.4 0.8 28.6 10.2 43.9
0.6 1.5 0.3 2.0 2.4 3.4 4.3 4.5 40.0 7.6 52.7
0.7 1.4 0.6 2.6 3.3 4.8 5.6 7.6 38.9 2.2 33.6
0.8 1.1 1.2 2.3 2.5 3.7 3.6 10.8 27.3 0.2 8.8
0.9 1.0 0.7 1.3 1.5 2.6 1.7 6.7 11.8 1.3 0.1
0.02 0.04 0.06 0.08 0.17 0.25 0.5 1 2 5
T

Your test cases with SmileAF

In [ ]:
```
iv = createTestImpliedVol(S, r, q, sc = 1.0, smileInterpMethod='AF')
plotTestImpliedVolSurface(iv)
pdeCalibReport(S, r, q, iv)
```