Implementation and testing of the implied volatility surface and numerical pricing of European options
FE5116 Project

ZHIYUAN WANG, ZHANG YIQING, HONG QIUXIA, SHAO ZHIMING, CHEN XINRUI

Abstract

This documentation provides a comprehensive review of implementation details, code structures, and testing framework for FE5116 final project which focus on volatility surface interpolation and generic European option pricing in the FX Market.

Contents

1 Project Overview

In this section, we provide an overview of both the project and implementation codes. This project focus on pricing any European options in the FX Market given sets of observations of vanillas European options.

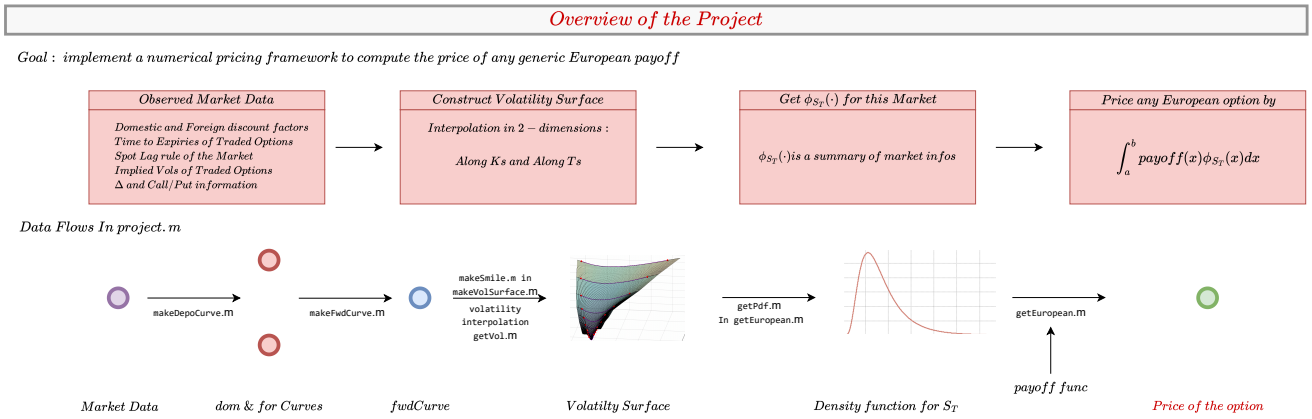

Figure 1.1: Overview of the project

The above Figure illustrates the outlines of this project. Given market observations, we first translating the market data into pairs of (K_i, σ_i). Then by interpolating in two dimensions, we summarize the market information into a surface from which we can get the volatility estimates for any T[1] and any K. Through finite difference, we get the pdf of S_T, which can be seen as the 'kernel' for pricing any European style options given terminal payoff function. We also present a summary on all the functions and in/out puts for a quick overview.

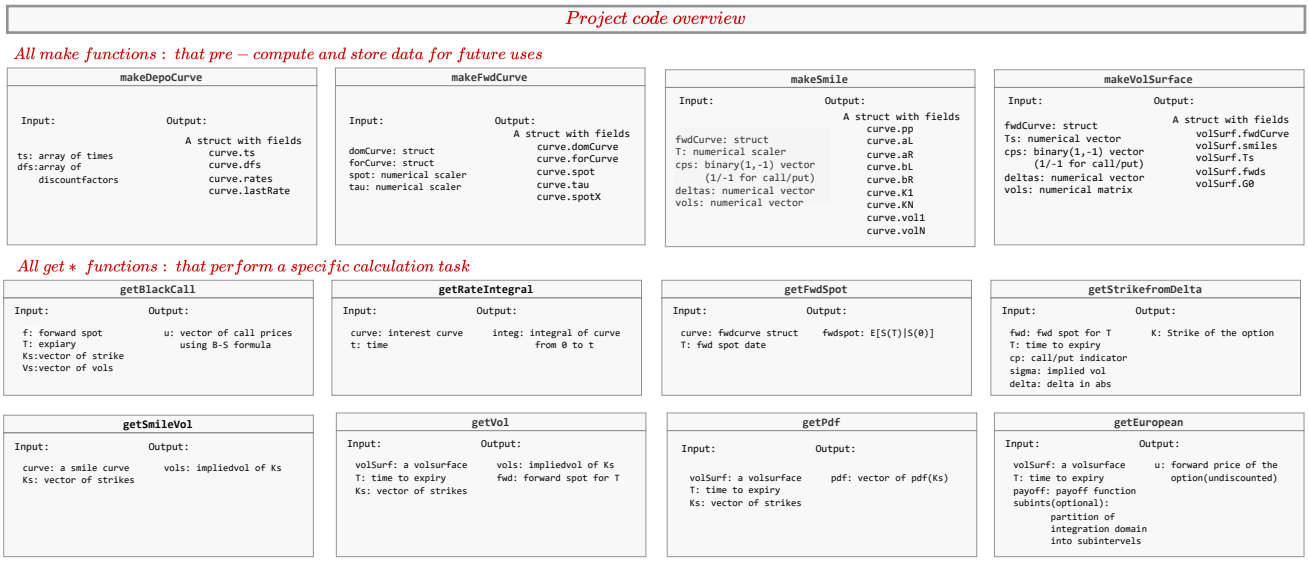

Figure 1.2: Summary of project codes

[1] As long as T is less than current largest expiry in the market since extrapolation is not allowed

2 Implementation

2.1 Input Argument Validation

To reduce errors caused by unexpected user input, we use input argument validation for each defined function to ensure that the input variable must satisfy certain structure (data types or within certain range).

Here we handle the argument validations in different manners for `make` and `get` functions. This is due to two considerations,

1. We notice that in the `getMarket.m` different vectors have different types(row or columns vectors). However, in internal handling we use dot operators for vectorization. To avoid a row and column vector generates a matrix instead of a desired element-wise calculated vector, we use argument validation to restrict all input vectors are columns (if not, converted to column vector). Thus, all vectors are handled as column vectors in `make` functions internally.

```
arguments
    fwdCurve (1,1) struct
    Ts (:,1) double {mustBeGreaterThanOrEqual(Ts,0)}
    cps (:,1) double {mustBeMember(cps, [-1, 1])}
    deltas (:,1) double {mustBeGreaterThanOrEqual(deltas,0),
        ↪ mustBeLessThanOrEqual(deltas,1)}
    vols (:,:) double {mustBeGreaterThanOrEqual(vols,0)}
end
```

Listing 1: Argument validation for makeVolSurface

2. For `get` functions, we usually hope that the output has the same type (column or row) as the input vector. Thus, in `get` functions we only asset they are the same type of vector and let the user to input the aligned inputs.

```
arguments
    f (1,1) double {mustBeGreaterThanOrEqual(f,0)}
    T (1,1) double {mustBeGreaterThanOrEqual(T,0)}
    Ks (:,:) double {mustBeGreaterThanOrEqual(Ks,0)}
    Vs (:,:) double {mustBeGreaterThanOrEqual(Vs,0)}
end
assert(all(size(Ks) == size(Vs)), 'Input value size much match.');
```

Listing 2: Argument validation for getBlackCall

We also performed argument validations including:

1. Non-Negative tenor, volatility, strikes, dfs and prices.
2. Delta value must belongs to $[0, 1]$, `cps` can only take values from $\{-1, 1\}$

2.2 Black formula

2.2.1 getBlackCall.m

The function *getBlackCall* computes the Black formula for the forward price of call options in a vectorized manner based on the values of $d_1$and d_2, as well as f and K_s.

- **Inputs** f: Forward spot for time T, i.e. $E[S(T)]$. T: Time to expiry. K_s: Strikes vector. V_s: Implied BS volatilities vector.
- **Output** u : Call options undiscounted prices vector.

The cumulative normal distribution function proceeds as follows:

$$
\begin{aligned}
u &= f \cdot \mathrm{N}(d_1) - K_s \cdot \mathrm{N}(d_2) \\
d_1 &= \frac{\ln\left(\frac{f}{K_t}\right) - 0.5 \cdot Vs^2 \cdot T}{Vs \cdot \sqrt{T}} \\
d_2 &= d_1 - V, \sqrt{T}
\end{aligned}
$$

We use vectorization to speed it up.

```
d1 = (log(f ./ Ks) + 0.5 .* Vs.^2 .* T) ./ (Vs.* sqrt(T));
d2 = d1- Vs .* sqrt(T);
u = f .* normcdf(d1) - Ks .* normcdf(d2);
```

Extreme case handling: When $K_s = 0$, we set $u = f$. When $K_s \to 0$, to enhance computational stability, we set $u = 0$ where $K_s < 1e-8$. We also set $u = 0$ when $|u| < 1e-8$. Furthermore, in cases where $V_s = 0$, we adjust u accordingly to avoid numerical issues. Specifically, we set $u(V_s == 0) = \max(f - K_s, 0)$.

The output Option Price as a Function of Ks and Vs can be seen in figure2.1, which is smooth.

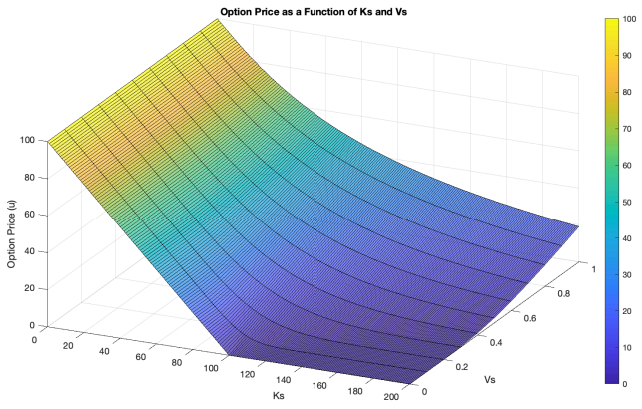

Figure 2.1: Option Price as a Function of Ks and Vs

2.3 Interest rate interpolation

2.3.1 makeDepoCurve.m

- **Inputs** ts: Array, time to settlement in years. dfs: Array, discount factors.
- **Ouput** $curve$: Struct, containing data rates.

The first time interval $rates_1$ and the subsequent intervals $rates_i$: $\text{rates}_1 = -\frac{\ln(\text{dfs}_1)}{t_1}$, $\text{rates}_i = -\frac{\ln(\text{dfs}_i/\text{dfs}_{i-1})}{t_i - t_{i-1}}$

The domCurve constructed with market data is shown in figure2.2, which is smooth when the data is dense.

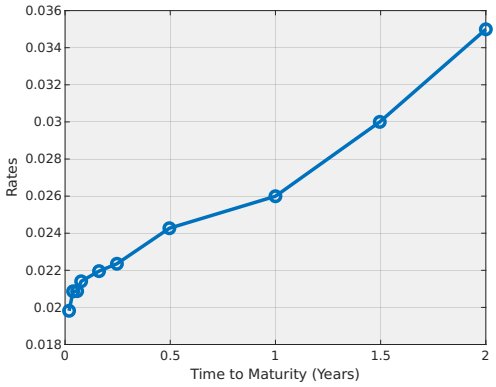

Figure 2.2: Rates vs Time to maturity

Structure of Curve Object

The `curve` object contains necessary attributes to calculate interest rate. Below is the detailed description of each property and their data types respectively.

- `ts` (vector): time to maturity.
- `dfs` (vector): discount factors.
- `rates` (vector): interest rate.

2.3.2 getRateIntegral.m

- **Inputs** $curve$: Precomputed data about an interest rate curve. t: Time.
- **Ouput** $integ$: Integral of the rate function from 0 to t.

This pre-computed approach enhances efficiency by avoiding redundant computations during runtime.

Then, the `getRateIntegral` function computes the integral of the local rate function from 0 to t as follows:

$$\text{integ} = \begin{cases} \text{rates}_1 \cdot t & \text{if } t \le t_1 \\ \text{rates}_n \cdot (t - t_n) + \sum_{i=1}^{n-1} (\text{rates}_i \cdot (t_i - t_{i-1})) & \text{if } t \ge t_n \\ \sum_{i=1}^{index-1} (\text{rates}_i \cdot (t_i - t_{i-1})) + \text{rates}_{\text{index}} \cdot (t - t_{\text{index}-1}) & \text{otherwise} \end{cases}$$

In this formula, rates_i is the i_{th} local rate, t_i is the i_{th} settlement time, $index$ is the smallest index satisfying $t \geq t_{index}$.

We use vectorization to speed it up:

```
time_interval = t - curve.ts(end);
integral_before_last_point = sum(diff([curve.ts]) .* curve.rates(2:end));
integ = curve.rates(end) * time_interval + curve.ts(1) * curve.rates(1) +
    ↪ integral_before_last_point;
```

2.4 Forward spot $G_0(T)(= S_T)$

2.4.1 makeFwdCurve.m

The makeFwdCurve function creates fwdCurve market object with domestic IR curve, foreign IR curve, spot exchange rate S_0 and time lag between spot and settlement τ. It is important to take note that only S_0, the τ-days forward price of cash rate X_0 is directly observable in the market, hence X_0 is calculated using following equation:

$$X_0 = \frac{S_0}{e^{\int_0^\tau r(u) - y(u)\, du}}$$

The integral calculation $\int_0^\tau r(u) - y(u)\, du$ is achieved by calling getRateIntegral function. The calculated cash exchange rate X_0 is stored in the structure fwdCurve as spotX.

Structure of fwdCurve Object

The `curve` object contains necessary attributes to calculate forward spot rate. Below is the detailed description:

- `domCurve` (struct): Domestic IR curve.
- `forCurve` (struct): Foreign IR curve.
- `spot` (double): spot exchange rate S_0.
- `tau` (double): Time lag between spot and settlement.
- `spotX` (double): Cash exchange rate X_0

2.4.2 getFwdSpot.m

The getFwdSpot function takes in two inputs: The fwdCurve object constructed in section 2.4.1 and forward spot date T. It calculates forward spot rate $G_0(T)$ using below equation:

$$G_0(T) = X_0 e^{\int_0^{T+\tau} [r(u) - y(u)]\, du}$$

Similar as 2.4.1, the integral part $\int_0^{T+\tau} [r(u) - y(u)]\, du$ is calculated using getRateIntegral. The getFwdSpot function returns $G_0(T)$ as fwdSpot.

2.5 Conversion of deltas to strikes

2.5.1 getStrikeFromDelta.m

The getStrikeFromDelta function takes in 5 inputs: forward spot rate fwd, option expiry time T, call/put indicator cp (1 for call, -1 for put), implied Black vol $sigma$ and delta in absolute value $delta$. We first calculate d_1 from option delta following below formula using matlab's built-in $norminv$ function:

$$|\Delta_{call}| = N(d_1)$$

$$|\Delta_{put}| = N(-d_1)$$

The function outputs strike K using the explicit form solution of rearranging Black-Scholes formula:

$$d_1 = \frac{\ln F_T - \ln K}{\sigma\sqrt{T}} + \frac{1}{2}\sigma\sqrt{T}$$

$$K = F_T e^{-d_1\sigma\sqrt{T}+\frac{1}{2}\sigma^2 T}$$

2.6 Interpolation of implied volatility in strike direction

2.6.1 smile object and makeSmile.m

makeSmile

The function `makeSmile` constructs a volatility smile curve based on provided market data and parameters.

The volatility smile curve is constructed using the following function:

$$\sigma(K) = \begin{cases} \text{spline}(K_1) + a_L \tanh(b_L(K_1 - K)), & K < K_1 \\ \text{spline}(K), & K_1 \le K \le K_N \\ \text{spline}(K_N) + a_R \tanh(b_R(K - K_N)), & K > K_N \end{cases} \tag{2.1}$$

The function is implemented as below:

1. Verify that all input vectors (`cps`, `deltas`, `vols`) are of the same length.
2. Calculate strikes from deltas using `getStrikeFromDelta`.
3. Check for arbitrage using convexity conditions on the calculated call prices and strikes.
4. Compute spline coefficients using **natural cubic spline**(not default spline).
5. Compute parameters a_L, b_L, a_R, and b_R by some first derivatives conditions.

$$\begin{cases} a_R b_R = \sigma'(K_N) \\ -a_L b_L = \sigma'(K_1) \\ 0.5 = \tanh^2(b_R(K_R - K_N)) \\ 0.5 = \tanh^2(b_L(K_1 - K_L)) \end{cases} \Rightarrow \begin{cases} b_L = \dfrac{\text{arctanh}(\sqrt{0.5})}{K_1 - K_L} \\ b_R = \dfrac{\text{arctanh}(\sqrt{0.5})}{K_R - K_N} \\ a_R = \dfrac{\sigma'(K_N)}{b_R} \\ a_L = -\dfrac{\sigma'(K_1)}{b_L} \end{cases}$$

Note: Taking only the positive value for the square root is feasible. When calculating the volatility extrapolation, only the term $curve.a_L \times tanh(curve.b_L \times (curve.K_1 - K))$ is used. This indicates that the sign of the final result solely depends on the sign of $a_L \times b_L$, which corresponds to $\sigma'(K_1)$. The same reasoning applies to `a_R, b_R`.

The output is a struct. Details are provided below:

Structure of Smile Object

The `curve` object contains several properties that are essential for constructing a volatility smile.

- `pp` (struct): A cubic spline structure. This struct includes coefficients and breakpoints of the spline.
- `aL`, `aR` (double): See equation2.1.
- `bL`, `bR` (double): See equation2.1.
- `K1`, `KN` (double): The smallest and largest strike that bound the region where `spline(K)` directly applies.
- `vol1`, `volN` (double): The volatilities at K_1 and K_N.

2.6.2 getSmileVol.m

The function `getSmileVol` computes implied volatilities for a given set of strike prices based on a pre-computed volatility smile curve.

- **Inputs** `curve`: A struct containing the pre-computed smile data. `Ks`: A vector of strike prices.
- **Output** `vols`: A vector of implied volatilities corresponding to `Ks`.

The function proceeds as follows:

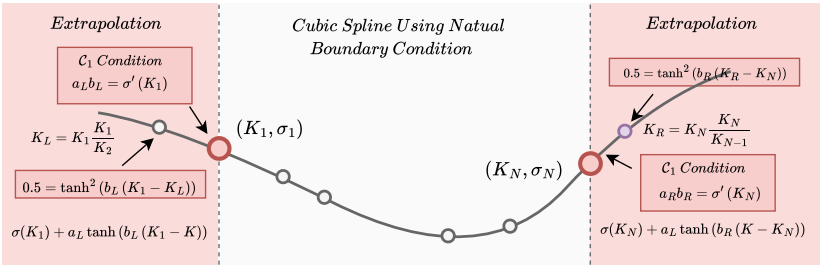

Figure 2.3: Illustration of smile interpolation

We use vectorization to speed it up:

```
vols = zeros(size(Ks));
leftIndices = Ks <= curve.K1;
rightIndices = Ks >= curve.KN;
middleIndices = ~leftIndices & ~rightIndices;
vols(leftIndices) = curve.vol1 + curve.aL * tanh(curve.bL * (curve.K1 - Ks(leftIndices))
    ↪ );
```

```
vols(rightIndices) = curve.volN + curve.aR * tanh(curve.bR * (Ks(rightIndices) - curve.
    ↪ KN));
vols(middleIndices) = ppval(curve.pp, Ks(middleIndices));
```

The output curve can be seen in the test `splinePlot`(See figure3.1), which is smooth.

2.7 Construction of implied volatility surface

2.7.1 volSurface object and makeVolSurface.m

makeVolSurface

The makeVolSurface function constructs an implied volatility surface from market data inputs. The function ensures that the constructed surface adheres to no-arbitrage conditions by performing checks on the ATM calls. To be more concrete, the function mainly achieve two things,

1. **Smile Construction**: For each tenor, the function computes an implied volatility `'smile'` using the `makeSmile` function. The results are stored in a cell array.
2. **No-Arbitrage Check**: The function calculates ATM call prices using the Black-Scholes formula(`getBlackCall`) for each tenor, using forward prices as strikes. It checks whether the price of a call option is increasing along moneyness lines: for any pair of maturities $T_1 < T_2$ and strike K, it must be

$$C\left(K, T_1\right) < C\left(K\frac{G_0\left(T_2\right)}{G_0\left(T_1\right)}, T_2\right)$$

which in ATM case is effectually simplified into,

$$C\left(G_0(T_1), T_1\right) < C\left(G_0\left(T_2\right), T_2\right)$$

If this condition is violated, an error is thrown.

Structure of VolSurface Object

The volSurface object in the function is designed to store all the necessary data related to the implied volatility surface that we've constructed. The structure of this object is listed as below,

- `fwdCurve`: A struct, containing the forward curve data used to compute forward prices at different expiry times.
- `smiles`: A cell array of structs, each cell contains a struct representing the volatility smile for a specific expiry time.
- `Ts`: A vector of expiry times
- `fwds`: A vector of the forward prices at each tenor specified in `'Ts'`.
- `G0`: A numeric scalar, the initial or current spot price derived from the `'fwdCurve'`.

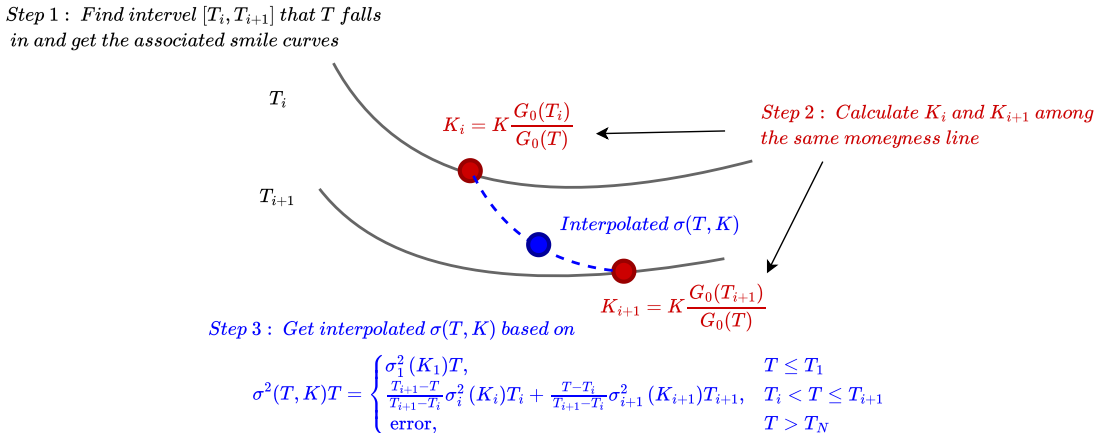

Figure 2.4: Illustration of volatility interpolation

2.7.2 getVol.m

The `getVol` function interpolates implied volatilities from a given volatility surface for specific strikes and maturities.

The interpolation process consists of the following steps, as depicted in the accompanying diagram:

1. Identify the interval $[T_i, T_{i+1}]$ that the target expiry T falls into and retrieve the associated smile curves.
2. Calculate K_i and K_{i+1} among the same moneyness line using the formula:

$$K_i = K\frac{G_0(T_i)}{G_0(T)}, \quad K_{i+1} = K\frac{G_0(T_{i+1})}{G_0(T)}$$

3. Obtain the interpolated $\sigma(T, K)$ based on linear interpolation of variance:

$$\sigma^2(T,K)T = \begin{cases} \sigma_i^2(K_i)T_i, & \text{if } T \le T_1 \\ \frac{T_{i+1}-T}{T_{i+1}-T_i}\sigma_i^2(K_i)T_i + \frac{T-T_i}{T_{i+1}-T_i}\sigma_{i+1}^2(K_{i+1})T_{i+1}, & \text{if } T_i < T \le T_{i+1} \\ \text{error}, & \text{if } T > T_N \end{cases}$$

In addition, to make the function robust for vector inputs of Strikes, we vectorize the code in calculating the interpolated vols.

By constructing grids for T and K, we construct the volatility surface given the market data. The alignment of market data points with the interpolated smiles ensures that the model accurately reflects market prices.(Indicated by purple lines and red dots). Additionally, the consistency of the surface's gradient demonstrates the smoothness of the surface, indicating the robustness of the interpolation methods used to construct the volatility surface.

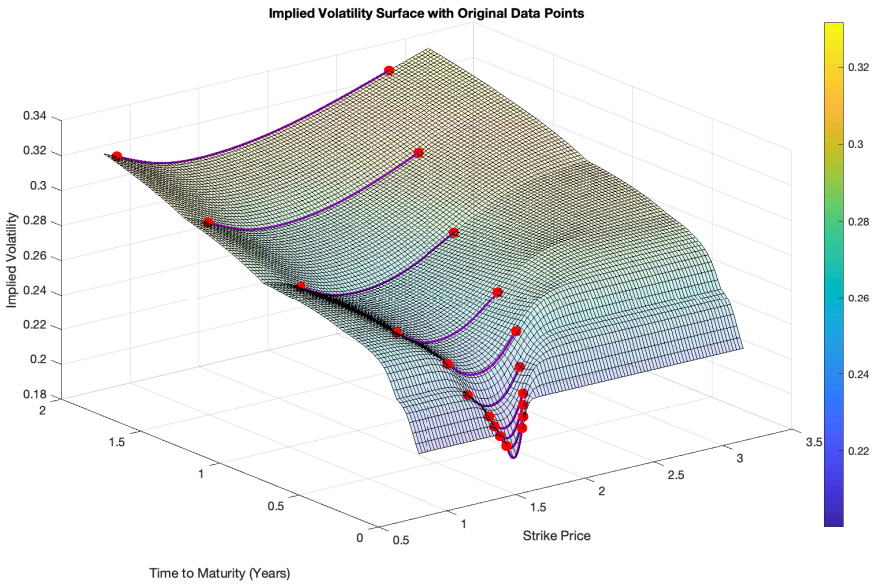

Figure 2.5: Implied Volatility Surface

2.8 Compute the probability density function of S_T

2.8.1 getPdf.m

The getPdf function computes the risk-neutral probability density function $\phi_{S_T}(x)$ using finite differences numerical method. As illustrated by below equation, the *pdf* is equivalent to the second derivative of T-forward price of call option $C(K,T)$ with respect to strike K.

$$\frac{d^2C(K)}{dK^2} = \phi_{S_T}(K)$$

This function takes in 3 inputs: volatility surface object $volSurf$ constructed in section 2.7.1, time to maturity T and vector of strikes Ks. The finite difference method for computing second order derivative is as follows. We choose h proportionally to K by multiplying K with $1e-3$.

$$\frac{d^2f(x)}{dx^2} = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

Therefore,

$$\phi_{S_T}(K) = \frac{d^2C(K)}{dK^2} = \frac{C_{\sigma(K+h)}(K+h) - 2C(K) + C_{\sigma(K-h)}(K-h)}{h^2}$$

where volatility σ is computed by passing same inputs into getVol function and call option price is calculated by calling getBlackCall with same inputs plus an extra volatility parameter. When input Ks is a vector, function getPdf will return an equal-sized vector. When input Ks is a scalar, this function will correspondingly return a scalar only.

Figure 2.6 is a sample output of *getPdf* function. It is smooth bell-shaped which fits our expectation.

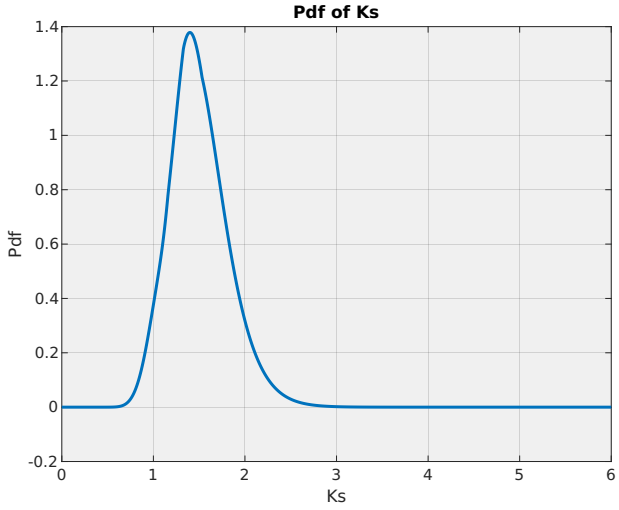

Figure 2.6: Probability Density Function

2.9 Compute forward prices of European options

2.9.1 getEuropean.m

The getEuropean function calculates the forward price of European options based on below integral:

$$V = \int_0^{\infty} f(x)\phi_{S_T}(x)\,dx$$

The function takes in three mandatory inputs (a $volSurface$ object constructed in section 2.7.1, option maturity time T, payoff function $payoff$) and one optional input $subints$: an extra feature to partition the integration domain. If $subints$ is not specified, it would default to $[0, +\infty]$. The integration function is multiplication of $payoff$ and pdf function obtained in section 2.8.

The implementation uses matlab's built-in *integral* function to integrate every sub-interval specified in $subints$ in for loop.

3 Tests Suite

In this section, we introduce the test suite we build for this project. We designed a test class for each of the `get` functions and wrap them into a test suite `test_suite.m`. All tests will be conducted at once and generate a test report in pdf format by running this script.

3.1 getBlackCall → testBlackCall

The tests for the `getBlackCall` function are contained within the testBlackCall class. The testing methods of this class are defined as follows:

3.1.1 testZeroStrike

- **Purpose** :Test u under zero strike conditions.
- **Test Method:** get u returned by the function and $expected_u$ by manual calculation under given f, T, K_s, and V_S
- **Pass/Fail Criteria:** check if u is equal to $expected_u$, with a tolerance of $1e-4$.
- **Test Result:** Pass.

3.1.2 testExtremeCases

- **Purpose** :Test u under a range of extreme scenarios.
- **Test Method:** a loop is used to test a large number of different combinations of K_s, and V_S.
- **Pass/Fail Criteria:**check if u is non-negative and does not exceed f.
- **Test Results:** At the beginning, test did not pass. After adding some constraints in `getBlackCall` function, the test passed.
- **Comments:**
  - When some $V_S = 0$, u becomes NaN. To handle this, we sets u to $\max(f - Ks(Vs == 0), 0)$.
  - Due to calculation errors, some u become a very close negative number. So,we set $u = 0$ when $|u| < 1e-8$.
  - Ultimately, all test passed, showing the `getBlackCall` function performs well.

  For specific values, negative option prices may occur due to computational errors. To address this, the `getBlackCall` function includes a threshold value `u_threshold = 1e-8;` and sets option prices below this threshold to zero.

3.2 getRateIntegral → testRateIntegral

The tests for the `getRateIntegral` function are contained within the testRateIntegral class. The testing methods of this class are defined as follows:

- **Purpose:** The test is to validate the integral of the local rate function from 0 to a given time t in different cases.
- **Test Method:**
  - `testRateIntegralZeroTime`: Test $integ$ when $t = 0$.
  - `testRateIntegralLastTime`: Test $integ$ when $t_s(end) < t <= t + 30days$.
  - `testRateIntegralbeyond`: Test $integ$ when $t > t + 30days$.
  - `testRateIntegralTenorTime`: Test $integ$ when $t = ts(i)$.
- **Execution**:
  - we set $ts = [0, 1, 2, 3]$ in all cases.
  - `testRateIntegralZeroTime`: get $integ$ when $t = 0$.

- `testRateIntegralLastTime`: get *integ* when $t = 3.05$ and get *expectedInteg* by manual calculation .
- `testRateIntegralbeyond`: get *integ* when $t = 4$.
- `testRateIntegralTenorTime`: get *integ* when $t = 1$.

- **Pass/Fail Criteria:**
  - `testRateIntegralZeroTime`: check if *integ* is equal to 0.
  - `testRateIntegralLastTime`: check if *integ* is equal to expectedInteg.
  - `testRateIntegralbeyond`:check if the function throws an error with the expected error identifier 'ErrorId:Integralbeyond' .
  - `testRateIntegralTenorTime`: check if *integ* is equal to $dfs(2)$.
- **Test Results:** All tests passed.

3.3 getFwdSpot → testFwdSpot

There are two tests for **`getFwdSpot`** function: **`testFwdAccuracy`** and **`testFwdNonNegative`**.

- **Purpose:** **`testFwdAccuracy`** compares the returned value of **`getFwdSpot`** to a set of correct calculation results. **`testFwdNonNegative`** is the property test for forward price as it cannot be negative.
- **Testing method:** Unit-test for **`testFwdAccuracy`**, Property-test for **`testFwdNonNegative`**.
- **Execution:** After constructing a pre-set $fwdCurve$, the test function is invoked on different option expiry dates. Then on each T, it compares the function returned value to known results/zero.
- **Pass/Fail Criteria:** Compare actual results and expected results with 1e-4 absolute tolerance.
- **Test Results:** Pass

3.4 getStrikeFromDelta → testStrike

There are two tests for **`getStrikeFromDelta`** function: **`testStrikeAccuracy`** and **`testStrikeNonNegative`**. The test logic is parallel to section 3.3.

3.5 getSmileVol → testSmileVol

The tests for the **`getSmileVol`** function are contained within the **`testSmileVol`** class. The testing methods of this class are defined as follows:

3.5.1 samelengthInput

- **Purpose:** This test verifies errors invoked by inputs with different lengths.
- **Testing Method:** The test is executed by passing vectors of different lengths to `makeSmile` and expecting an error, thus validating the function's ability to handle input length mismatches.
- **Pass/Fail Criteria:** The test passes if an error with identifier 'ErrorId:vectorLengthMismatch' is produced. The absence of an error or an incorrect error identifier results in a test failure.
- **Test Results:** Pass.

3.5.2 noconvexarbitrageInput

- **Purpose:** This test checks the proper detection of arbitrage based on the convexity of Calls and Strikes.
- **Testing Method:** The method involves calling `makeSmile` with input vectors designed to simulate a non-convex option pricing scenario that should theoretically lead to arbitrage opportunities.
- **Pass/Fail Criteria:** The test passes if `makeSmile` throws an error with the ID 'ErrorId:arbitrageDetected'. Otherwise, it fails.
- **Test Results:** Pass

3.5.3 InterpolationPoints

- **Purpose:** This test verifies that if called with one of the points used to interpolate, it must return the original point.
- **Testing Method:** The test employs a set of predetermined Ks to generate vols, then checks if the calculated vols for these predetermined Ks are equal to the original vols.
- **Execution:** Inputs include:`cps`, `deltas`, `vols` and `T`. A smile curve is generated using `makeSmile`. Vols are calculated at original strikes and compared with the original vols.
- **Pass/Fail Criteria:** The test passes if: The interpolated volatilities at original strikes match the original volatilities within an absolute tolerance of $1e-4$. It fails if the condition is not met.
- **Test Results:** Pass.

3.5.4 checkFixedPoints

- **Purpose:** This test verifies that the `getSmileVol` function accurately computes the implied vols.
- **Testing Method:** The test employs a set of predetermined Ks to generate vols, then checks if the calculated vols for some specific strike prices are equal to the actual vols.
- **Execution:** Construct a smile curve like 3.5.3. Vols are calculated at fixed strikes and compared with the expected vols.
- **Pass/Fail Criteria:** The test passes if: The volatilities are accurately calculated within an absolute tolerance of $1e-4$. It fails if the condition is not met.
- **Test Results:** Pass.

3.5.5 atmvolMin

- **Purpose:** This test confirms the assumption that at-the-money (ATM) volatility is the lowest compared to volatilities at other strike prices.
- **Testing Method:** The test involves generating a volatility smile, determining the ATM volatility using ATM strike, and comparing it against volatilities at nearby strikes to ensure it is the lowest.
- **Execution:** The ATM strike is calculated using the forward spot rate from `getFwdSpot`. ATM vol and additional vols around the ATM strike are calculated using `getSmileVol`. These volatilities are then compared to verify that the ATM volatility is the lowest.

- **Pass/Fail Criteria:** The test passes if the ATM volatility is no greater than all other computed volatilities in the tested range. Otherwise, it fails.
- **Test Results:** Pass.

3.5.6 splinePlot

- **Purpose:** This test is to visually confirm that the interpolated volatility smile curve is qualitatively smooth.
- **Testing Method:** The test generates a spline-based volatility smile from predefined input data and visualizes the curve along with notable points(K1, KN and the at-the-money (ATM) strike) to assess the smoothness and continuity of the curve.
- **Pass/Fail Criteria:** The test is considered successful if the curve appears smooth and continuous.

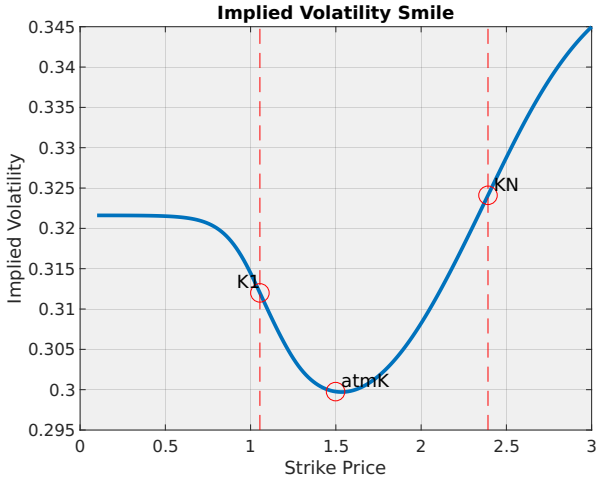

Figure 3.1: Implied Volatility Smile

- **Test Results:** The plot of a smile curve(figure3.1) looks qualitatively smooth. The `atmvolMin` test could also be confirmed as the atmvol is the lowest point.

3.6 getVol → testVol

The tests for the `getVol` function are contained within the `testVol` class. In this testclass we defined a specific case with known outputs,

```
1   methods(TestClassSetup)
2       function createVolSurface(testCase)
3           Ts = [0.1; 0.5; 1];
4           domCurve = makeDepoCurve (Ts , [0.99962;0.99922;0.99882] );
5           forCurve = makeDepoCurve (Ts , [0.99923;0.99845;0.99766] );
6           testCase.fwdCurve = makeFwdCurve ( domCurve , forCurve , 1.5 , 2 / 365 );
7           vols = [20.80 20.20 20.00; 21.32 20.71 20.50 ;21.84 21.21 21.00]/100;
8           testCase.volSurf = makeVolSurface (testCase.fwdCurve , Ts , [-1,-1,1], [0.1 ,
                ↪ 0.25 , 0.5], vols);
9       end
10  end
```

3.6.1 testInterpolation

- **Purpose:** Check whether the interpolation function generated expected output for new data point.
- **Testing Method:** Unit-test.
- **Pass/Fail Criteria:** Define a specific simple test case with volatility pre-computed as expected result. Test Passes when expected and actual results matches with in 1e-4 absolute tolerance.
- **Test Results:** Pass

  For input `T` = 0.75, `Ks` = [1,2], we get actual output:1.4986 and [0.2226 0.2075] which aligns with the expected output.

3.6.2 testInterpolated

- **Purpose:** Check whether the interpolation function generated expected output for **interpolated** point (that we use to build the interpolation).
- **Testing Method:** Unit-test.
- **Pass/Fail Criteria:** Pick one original point from the vol surface as the expected result. Test Passes when expected and actual results matches with in 1e-4 absolute tolerance.
- **Test Results:** Pass

3.6.3 testErrorOnExtrapolation

- **Purpose:** Verify error invoked by extrapolation.
- **Testing Method:** Unit-test.
- **Pass/Fail Criteria:** Check whether the captured error inditifier aligns with the one defined in the function.
- **Test Results:** Pass

  By input $T = 2$, the testCase error `getVol:ExtrapolationNotAllowed` which aligns with the one we defined.

3.7 getPdf → testPdf

There are two tests for **`getPdf`** function: **`testIntegralPdfEqualsOne`** and **`testMeanEqualsFwd`**.

3.7.1 testIntegralPdfEqualsOne

- **Purpose:** To check whether integral of probability density function equals to one.
- **Testing method:** Unit-test
- **Execution:** After constructing a pre-set $volSurf$, the test function calculates the integral of **`getPdf`** using matlab's built-in **`integral`** function.
- **Pass/Fail Criteria:** Compare actual integral with theoretical value 1 with 1e-3 absolute tolerance.
- **Test Results:** Pass

3.7.2 testIntegralPdfEqualsOne

- **Purpose:** To check whether integral of probability density function equals one.
- **Testing method:** Unit-test
- **Execution:** After constructing a pre-set $volSurf$, the test function calculates the mean of `getPdf` by below equation:

$$\int_0^{+\infty} xf(x)\,dx$$

- **Pass/Fail Criteria:** Compare actual mean with theoretical value forward returned by `getFwdSpot` with 1e-3 absolute tolerance.
- **Test Results:** Pass

3.8 getEuropean → testEuropean

3.8.1 testPutCallParity

- **Purpose:** To check whether option price returned by `getEuropean` fulfills put-call parity property.
- **Testing method:** Unit-test
- **Execution:** After constructing a pre-set $fwdCurve$ and $volSurf$, the test function invokes `getEuropean` to calculate the call option price and put option price with the same strike K. They are expected to follow put-call parity as below formula:

$$C(K) - P(K) = (Fwd - K) * df$$

- **Pass/Fail Criteria:** Compare LHS to RHS of above equation with 1e-4 absolute tolerance.
- **Test Results:** Pass

3.8.2 testEqualsBS

- **Purpose:** To check in extreme market conditions (i.e., no smile), the call option price obtained by `getEuropean` is equivalent to the theoretical Black-Scholes price obtained by `getBlackCall`.
- **Testing method:** Unit-test
- **Execution:** We first construct a volSurface where volatility is constant (0.25 in our example). Then the test function invokes `getEuropean` to calculate the call option price. Then we calculate the expected BS price using `getBlackCall`.
- **Pass/Fail Criteria:** Compare `getEuropean` result to `getBlackCall` with 1e-4 absolute tolerance.
- **Test Results:** Pass

3.8.3 testReplicateDigital

- **Purpose:** To check a digital option can be replicated by vanilla calls numerically.
- **Testing method:** Unit-test

  Mathematically terminal payoff satisfies

- **Execution:** By the following manipulation of terminal payoffs,

$$1_{\{S_T>K\}} = \frac{\max(S_T - (K-\epsilon), 0) - \max(S_T - (K+\epsilon), 0)}{2\epsilon}$$

  We use the `getEuropean` function to generate prices given the above left and right hand sides payoff functions. Here we set $\epsilon = K \times 10^{-4}$.

- **Pass/Fail Criteria:** Compare the prices of the above two derivatives with 1e-7 absolute tolerance.

- **Test Results:** Pass