Modeling Implied Volatility Surfaces using 2D Penalized B-Splines: A Numerical Analysis

Sebastian Conway-Burt

April 29, 2025

1 Introduction

1.1 Context and Motivation: The Implied Volatility Surface

Financial options are derivative products that give the holder the right, but not the obligation, to buy or sell a specified underlying asset at a predetermined exercise price (K) on or before a given maturity date. Valuing these contracts correctly is a cornerstone of modern finance. The Black-Scholes formula offers a theoretical method for valuing European options based on a set of parameters such as the market price of the underlying asset (S), exercise price (K), time to maturity (T), risk-free interest rate (r), and the volatility of returns on the underlying asset (σ). While most of the parameters are directly observable or can be assumed quite realistically, volatility is an issue. The model presumes the volatility level is constant, but market prices frequently suggest otherwise.

Implied volatility (IV) is a single number, σ, that we can put into the Black-Scholes formula, which gives us the option price that is the market price (C_{market}). It is what the market believes will be future volatility. Market data tells us that implied volatility is not flat but varies systematically with the strike price, (K) and the option's time to expiration, (T). This relationship gives the Implied Volatility Surface (IVS), a function $V(K,T)$ that take a strike-maturity natural pair to its corresponding implied volatility.

- **Risk Management:** IVS is also used by derivatives traders to compute option price sensitivities to underliers (so-called "Greeks"), including such "Greeks" as price sensitivity to volatility, Vega (u), price sensitivity to underlier, Delta (Δ) and Delta sensitivity, Gamma (Γ), with respect to different strikes and maturities. An exact IVS enables a better hedge of options portfolios.
- **Pricing Exotic Derivatives:** Many exotic options rely on more than one volatility number but on the entire volatility structure over strikes and time. The IVS is a required input in order to price these instruments.
- **Market Signal Extraction:** The form of the IVS (e.g., the "skew" or the "smile") informs about a widespread market believes to have a big chance of larger price movements and tail events (it contains information beyond a single volatility number).
- **Arbitrage Check:** Theoretical finance implies some smoothness and no-arbitrage conditions which a correct IVS should obey. Divergences of realized surfaces from this conditions might be indicators for market inefficiencies.

However, constructing a reliable IVS from market data is a significant numerical challenge. Observed option prices contain noise, bid-ask spreads introduce uncertainty, and data is typically sparse. This is seen especially when dealing with options far from the current price or with long maturities. Raw implied volatilities derived from this data are often scattered and may violate theoretical constraints. Therefore, robust numerical methods are required to filter noise, interpolate missing data, and construct a smooth representation of the underlying surface $V(K,T)$.

1.2 Project Goal: Algorithmic Analysis of Surface Generation Techniques

The purpose of this work is to develop and finalize a thorough experiment of a set of algorithms for IV. Although the general questions we address were motivated by a financial application, the emphasis in this paper is strongly within the realm of numerical analysis. The present report focuses on the detailed analysis of mathematical foundation, algorithmic structure as well as stability and convergence properties, errors and computational issues of the selected numerical methods. This study aims to offer an insight to the performance of these algorithms as well as limitations and trade-offs in their application for fitting potentially ill-conditioned data observed in finance markets. The theoretical foundation of many of these approaches are textbooks in classical numerical analysis.

1.3 Report Outline

This report is structured as follows:

1. **Section 1 (Introduction):** Provides the context, motivates the problem, states the project objective, introduces the core numerical methods, and outlines the report structure.
2. **Section 2 (Methodology):** Presents a detailed description and theoretical analysis of the four core methods listed above. This includes their mathematical formulation, algorithmic implementation details, theoretical properties (convergence, error bounds), stability and conditioning analysis, and potential numerical challenges.
3. **Section 3 (Experimental Setup):** Describes the computational environment, data sources and processing steps, the specific parameters explored in the numerical experiments, and the metrics used for evaluating the performance of the IVS construction.
4. **Section 4 (Results):** Presents the outcomes of the numerical experiments conducted using the setup described in Section 3. This includes quantitative performance metrics (e.g., IV MAE, Price RMSE), analysis of convergence data (IV calculation, LSQR iterations), and visualizations comparing the impact of different parameters (specifically the smoothing parameter λ).
5. **Section 5 (Discussion):** Interprets the findings from the Results section in light of the numerical analysis presented in the Methodology section. Discusses the observed trade-offs (e.g., bias-variance), the effectiveness of the methods, limitations, and potential areas for future investigation.
6. **Section 6 (Conclusion):** Summarizes the key findings and conclusions of the study regarding the application and numerical properties of P-splines for IVS construction.

1.4 Background of the Main Numerical Techniques Studied

We analyzed four different numerical approaches applied to distinct tasks within the Implied Volatility Surface (IVS) reconstruction process. The first two methods directly utilize techniques discussed in class, while the latter two adapt and extend these foundational concepts to address the specific challenges of surface fitting:

- **Classical Newton's Method (Preprocessing):** As a preprocessing step, this standard iterative root-finding method, studied in the Zeros of Functions section of our class, is used. It derives the input implied volatility data points (IV_i) from observed market option prices (C_{market}) by implicitly solving the non-linear Black-Scholes pricing equation $f(\sigma) = C_{\text{BS}}(S, K, T, r, \sigma) - C_{\text{market}} = 0$ with respect to the volatility parameter σ.
- **1D Natural Cubic Spline Interpolation (Yield Curve):** To obtain a smooth risk-free interest rate curve $r(T)$ (as a function of time-to-maturity T) for use in the Black-Scholes formula, we interpolate discrete yield data points (T_j, r_j) (e.g., U.S. Treasury yields). This employs 1D natural cubic splines, creating a smooth, twice continuously differentiable yield curve $r(T)$, leveraging principles of polynomial interpolation taught in class.

- **2D Penalized B-Splines (IVS Fitting):** Building upon the concept of 1D splines, this is the primary method used for modeling the IVS itself. It fits a smooth surface $V(K,T)$ to the scattered implied volatility data points $(K_i, T_i, \mathrm{IV}_i)$ derived from the first step. The approach utilizes a flexible tensor-product B-spline basis and incorporates a penalty term based on the second differences of the spline coefficients. This penalty enforces smoothness and mitigates overfitting, balancing data fidelity with surface regularity via a tunable smoothing parameter λ. In our implementation (isotropic P-splines), the same λ is applied in both the strike (K) and maturity (T) dimensions.
- **LSQR Algorithm (Solving the Linear System):** The process of fitting the 2D Penalized B-Spline involves solving a large, potentially sparse, linear least-squares problem to find the B-spline coefficients. We employ the LSQR algorithm, an iterative method particularly suitable for such systems, to determine these coefficients efficiently and in a numerically stable manner. This technique, an extension of linear algebra concepts from class, avoids the explicit formation and computation of the normal equations (A^TA), which can suffer from severe ill-conditioning due to the squaring of the condition number of the design matrix A.

2 Methodology: Numerical Algorithms and Analysis

This section details the mathematical foundations, algorithmic structure, implementation specifics, and numerical properties of the core methods employed in this project for constructing the implied volatility surface. The focus is on the analysis from a numerical perspective, examining aspects such as convergence, stability, error propagation, and conditioning, which are central to evaluating the suitability and robustness of these techniques. Foundational concepts are drawn from standard numerical analysis literature.

2.1 Newton-Raphson Method for Implied Volatility Extraction

The first step in constructing the IVS often involves deriving implied volatility values from observed market option prices. Since the Black-Scholes formula provides price as a function of volatility, finding the implied volatility requires solving the inverse problem. This section details the application of the Newton-Raphson method, a technique covered in class, for this purpose.

2.1.1 Mathematical Formulation and Black-Scholes Context

Given an observed market price for a European call option, C_{market}, the corresponding implied volatility σ is the value that satisfies the equation:

$$f(\sigma) = C_{\text{BS}}(S, K, T, r, \sigma) - C_{\text{market}} = 0 \tag{1}$$

where C_{BS} is the Black-Scholes price. For a non-dividend-paying stock, the formula is:

$$C_{\text{BS}}(S, K, T, r, \sigma) = S\Phi(d_1) - Ke^{-rT}\Phi(d_2) \tag{2}$$

with

$$d_1 = \frac{\ln(S/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}} \tag{3}$$

$$d_2 = d_1 - \sigma\sqrt{T} \tag{4}$$

Here, S is the current price of the underlying asset, K is the strike price, T is the time to maturity (in years), r is the continuously compounded risk-free interest rate (obtained via spline interpolation, see Section 2.2), and $\Phi(\cdot)$ is the CDF of the standard normal distribution. The computation of $\Phi(\cdot)$ typically relies on library functions, such as those available in `scipy.stats.norm`. The task is to find the root $\sigma > 0$ of the nonlinear function $f(\sigma)$.

2.1.2 Algorithm and Implementation Aspects

The Newton-Raphson method is an iterative technique for finding roots of differentiable functions. Starting with an initial guess σ_0, the method generates a sequence of approximations $\{\sigma_k\}$ using the iteration:

$$\sigma_{k+1} = \sigma_k - \frac{f(\sigma_k)}{f'(\sigma_k)} \tag{5}$$

The derivative $f'(\sigma)$ is the partial derivative of the Black-Scholes price with respect to volatility, known in finance as the option's Vega (ν):

$$f'(\sigma) = \frac{\partial C_{\text{BS}}}{\partial \sigma} = \nu = S\phi(d_1)\sqrt{T} \tag{6}$$

where $\phi(\cdot)$ is the probability density function (PDF) of the standard normal distribution. Our specific implementation in `src/implied_volatility.py` incorporates several practical considerations aligned with the theory:

- **Input Validation:** Before iterating, the function checks for invalid inputs, specifically if time-to-maturity T is non-positive ($T \leq 10^{-9}$) or if the market price C_{market} is non-positive. If either condition is met, the calculation aborts and returns a failure status.
- **Initial Guess (σ_0):** The function accepts an `initial_sigma` parameter, defaulting to 0.2. Within the main experiment (`data_analysis.py`), this initial guess is potentially set to the market implied volatility provided in the input data file (`market_iv_original`) if available and within reasonable bounds (0.01 to 1.5), falling back to the default 0.2 otherwise.
- **Iteration Loop:** The core of the function is a `for` loop iterating from 0 to `max_iterations` - 1.
- **Core Calculation:** Inside the loop, the Black-Scholes price C_{BS} (using `src.black_scholes.black_scholes_call`) and Vega ν are calculated using the current σ_k. Error handling (`try...except`) is included to catch potential mathematical errors (e.g., division by zero if $\sigma\sqrt{T}$ is too small) during these calculations.
- **Convergence Criterion:** The loop terminates successfully if the absolute difference between the calculated Black-Scholes price and the market price is less than a specified tolerance ϵ: $|f(\sigma_k)| = |C_{\text{BS}}(..., \sigma_k) - C_{\text{market}}| < \epsilon$ (where ϵ is `tolerance`, defaulting to 10^{-7}).
- **Handling Near-Zero Vega:** Before performing the Newton-Raphson update step, the implementation checks if the absolute value of Vega is below a threshold: $|\nu| < \nu_{\text{threshold}}$ (where `vega_threshold` defaults to 10^{-8}). If Vega is too small, the iteration stops, and a specific failure status ('Vega Too Small') is returned, avoiding division by a near-zero number.
- **Update Step:** If convergence is not met and Vega is sufficient, the next volatility estimate is calculated as $\sigma_{k+1} = \sigma_k + (C_{\text{market}} - C_{\text{BS}}(..., \sigma_k))/\nu$.
- **Maximum Iterations (N_{max}):** If the convergence criterion is not met within the `max_iterations` limit (defaulting to 5000), the loop terminates, and a 'Max Iterations Reached' status is returned.
- **Output:** Instead of just returning the volatility or `None`, the function returns a dictionary containing detailed results: the calculated `implied_volatility` (or `None` on failure), the number of `iterations` performed, the `final_diff` ($|f(\sigma_k)|$ at termination), and a `status` string indicating the outcome ('Converged', 'Max Iterations Reached', 'Vega Too Small', 'Calculation Error', 'Non-Positive TTM', 'Non-Positive Sigma'). This structured output facilitates detailed analysis of the root-finding process.
- **Sigma Positivity Check:** The implementation includes checks to ensure the calculated sigma remains positive during and after iterations. If sigma becomes non-positive, or if convergence occurs at a non-positive value, a 'Non-Positive Sigma' status is returned.

2.1.3 Convergence and Error Analysis

Under ideal conditions, the Newton-Raphson method exhibits quadratic convergence towards a simple root σ^* (where $f(\sigma^*) = 0$ and $f'(\sigma^*) \neq 0$). This means that if the error at iteration k is $e_k = \sigma_k - \sigma^*$, then the error at the next iteration satisfies:

$$|e_{k+1}| \leq C|e_k|^2 \tag{7}$$

for some constant $C = \frac{|f''(\sigma^*)|}{2|f'(\sigma^*)|}$, provided the initial guess σ_0 is sufficiently close to σ^*. This rapid convergence makes Newton's method very efficient when applicable. The error is typically not computed directly but is controlled implicitly by the stopping tolerance ϵ. Standard numerical analysis texts provide detailed proofs and derivations.

2.1.4 Stability, Conditioning, and Failure Modes

The well-posedness of finding an implied volatility depends on the option's parameters. For standard European options, a unique positive implied volatility typically exists if the market price C_{market} is above the option's intrinsic value ($\max(S - Ke^{-rT}, 0)$ for a call) and below theoretical upper bounds (e.g., S for a call). However, the numerical stability of the Newton-Raphson method itself can be problematic:

- **Divergence:** If the initial guess σ_0 is far from the true root σ^*, the iterations may diverge.
- **Instability near Critical Points:** The most significant numerical issue arises when Vega ($f'(\sigma)$) is close to zero. As seen in the iteration formula (5), a small denominator leads to large, potentially unstable steps. This reflects the ill-conditioning of the root-finding problem when the function is locally flat. The sensitivity of the root σ to changes in C_{market} becomes very high when Vega is small.
- **No Root:** If the input market price violates no-arbitrage bounds (e.g., price is below intrinsic value), no positive volatility σ exists that satisfies the equation, and the method will fail to converge to a meaningful result.

Careful implementation, including robust handling of near-zero Vega (as done via `vega_threshold`) and checks for valid input prices and positive time-to-maturity, is crucial for reliable implied volatility calculation.

2.2 1D Natural Cubic Spline Interpolation for Risk-Free Rates

The Black-Scholes model requires a risk-free interest rate r corresponding to the option's time-to-maturity T. Market interest rate data is typically available only at discrete maturities (e.g., Treasury yields for 1 month, 3 months, 1 year, etc.). As covered in class exercises on polynomial interpolation, spline interpolation provides a robust method to obtain a continuous rate curve $r(T)$.

2.2.1 Interpolation Problem Statement

Given a set of $N + 1$ discrete yield data points $(T_0, r_0), (T_1, r_1), ..., (T_N, r_N)$, where $T_0 < T_1 < ... < T_N$ are distinct maturities and r_j are the corresponding yields (obtained from `fetch_daily_yield_curve`), the goal is to construct a function $S(T)$ such that:

1. $S(T_j) = r_j$ for all $j = 0, ..., N$ (Interpolation condition).
2. $S(T)$ is sufficiently smooth over the interval $[T_0, T_N]$.
3. $S(T)$ can be evaluated for any T within the range $[0, T_N]$.

2.2.2 Method Description and Properties (C^2 Continuity)

A 1D natural cubic spline $S(T)$ is a popular choice for this task due to its smoothness properties. It is defined as a piecewise cubic polynomial on each subinterval $[T_j, T_{j+1}]$. The defining characteristics are:

- **Interpolation:** $S(T_j) = r_j$ for $j = 0, ..., N$.

- **Continuity:** $S(T)$, the first derivative $S'(T)$, and the second derivative $S''(T)$ are continuous across the interior knots $T_1, ..., T_{N-1}$. This ensures the resulting curve is C^2 continuous, meaning it has no jumps in value, slope, or curvature.
- **Natural Boundary Conditions:** $S''(T_0) = 0$ and $S''(T_N) = 0$. These conditions specify zero curvature at the endpoints of the interpolation interval.

A key theoretical property of natural cubic splines is that they minimize the integral of the squared second derivative among all twice continuously differentiable functions interpolating the given data points:

$$\int_{T_0}^{T_N} [S''(t)]^2 dt \leq \int_{T_0}^{T_N} [g''(t)]^2 dt \tag{8}$$

for any $g \in C^2$ such that $g(T_j) = r_j$ for all j. This minimization property leads to curves that are considered "smooth" or having minimal "bending energy," which is often desirable for representing yield curves.

2.2.3 Implementation: System Solution and Extrapolation

The implementation follows the standard procedure for computing natural cubic splines, involving determining the unknown second derivatives $M_j = S''(T_j)$ at each knot $j = 0, ..., N$. This is handled within the `create_rate_interpolator` function in `src/market_data_utils.py`.

- **Data Preparation:** Before solving, `create_rate_interpolator` performs data cleaning on the input `yield_curve_data`. It ensures maturities (T_j) and rates (r_j) are numeric, drops any rows with NaNs, sorts the data by maturity, and removes duplicate maturity points (keeping the first occurrence). It requires at least 3 unique, valid data points to proceed.
- **System Setup:** The continuity requirement for the first derivative $S'(T)$ at the interior knots $T_1, ..., T_{N-1}$ leads to a system of linear equations. Let $h_j = T_{j+1} - T_j$. The equations for the interior knots ($j = 1, ..., N-1$) are derived as:

$$\frac{h_{j-1}}{6} M_{j-1} + \frac{h_{j-1} + h_j}{3} M_j + \frac{h_j}{6} M_{j+1} = \frac{r_{j+1} - r_j}{h_j} - \frac{r_j - r_{j-1}}{h_{j-1}} \tag{9}$$

- **Boundary Conditions:** The natural boundary conditions provide $M_0 = 0$ and $M_N = 0$.
- **Solving the System:** The implementation uses the helper function `_solve_natural_cubic_spline_derivs`. This function constructs the $(N-1) \times (N-1)$ tridiagonal, symmetric, and strictly diagonally dominant coefficient matrix A and the right-hand side vector B corresponding to the equations above (incorporating $M_0 = M_N = 0$). It then solves the system $AM_{\text{internal}} = B$ for the unknown interior second derivatives $M_1, ..., M_{N-1}$ using `np.linalg.solve`. Checks are included to ensure at least 3 points are available and that no adjacent knot intervals have zero length ($h_j \approx 0$). The full vector M (including M_0 and M_N) is then assembled.
- **Piecewise Evaluation:** Once all M_j are known, the function `create_rate_interpolator` returns a callable `interpolator_func`. This function internally calls `custom_cubic_spline_interp1d_with_short_extrap(ttm, maturities_final, rates_final, M_derivs)`. For a given evaluation point T, this function first determines the interval $[T_j, T_{j+1}]$ containing T (using `np.searchsorted`). It then evaluates the cubic polynomial for that segment, which depends on $r_j, r_{j+1}, M_j, M_{j+1}$ and h_j. A specific check handles the case where h_j might be zero (due to cleaned data points being extremely close), defaulting to linear interpolation (r_j) in that rare scenario.
- **Extrapolation Handling:** The implementation in `custom_cubic_spline_interp1d_with_short_extrap` defines the behavior outside the original data range $[T_0, T_N]$:
  - For maturities shorter than the first data point ($0 \leq T < T_0$), flat extrapolation is used: $S(T) = r_0$.
  - For maturities greater than the last data point ($T > T_N$) or less than zero ($T < 0$) the function raises a `ValueError`, which is caught by the `interpolator_func` wrapper, causing it to return `np.nan` rather than attempting extrapolation.

2.2.4 Theoretical Error Bounds and Numerical Considerations

Theoretical analysis provides error bounds for cubic spline interpolation, assuming the underlying function $f(T)$ being interpolated is sufficiently smooth. If $f \in C^4[T_0, T_N]$ and $S(T)$ is the unique cubic spline interpolating f at knots $T_0 < T_1 < \cdots < T_N$, then for both clamped and natural boundary conditions, convergence results indicate that the error behaves according to the spacing between knots, $h = \max_{0 \le j \le N-1}(T_{j+1} - T_j)$. Specifically, under appropriate conditions, the maximum interpolation error is bounded proportionally to h^4:

$$\|f - S\|_\infty = \max_{T \in [T_0, T_N]} |f(T) - S(T)| = O(h^4)$$

As detailed in Burden and Faires [2, p. 155], for clamped boundary conditions, an explicit bound derived from the Peano Kernel Theorem is given by $\|f - S\|_\infty \le \frac{5}{384} h^4 \|f^{(4)}\|_\infty$. While the explicit error bounds for natural boundary conditions are more complex to state, similar $O(h^4)$ convergence holds [2, p. 156]. Corresponding bounds for the derivatives also exist, generally showing $\|f' - S'\|_\infty = O(h^3)$ and $\|f'' - S''\|_\infty = O(h^2)$. These theoretical results confirm that cubic spline interpolation provides high accuracy when the underlying function possesses four continuous derivatives and the knot spacing h is small. However, several factors, relevant to our implementation, can introduce inaccuracies:

- **Boundary Conditions:** The natural boundary conditions ($S'' = 0$) are assumed for simplicity and may not reflect the true curvature of the yield curve at the endpoints. This can lead to larger errors near T_0 and T_N.
- **Extrapolation Error:** The flat extrapolation used for $T < T_0$ ($S(T) = r_0$) is a strong assumption and can introduce significant errors if the yield curve has a non-zero slope at the short end. No extrapolation is performed for $T > T_N$.
- **Data Noise:** Errors or noise in the input Treasury yield data r_j will directly propagate through the interpolation process. The initial data cleaning helps mitigate this but doesn't eliminate it.
- **Well-posedness:** As long as the maturity points T_j are distinct after cleaning (ensured by the implementation), the tridiagonal system for the second derivatives M_j is well-conditioned, ensuring a unique and stable solution for the spline coefficients via `np.linalg.solve`.

2.3 2D Isotropic Penalized B-Splines for Surface Fitting

The core task is to construct a smooth, continuous implied volatility surface $V(K, T)$ from the discrete, noisy implied volatility data points (K_i, T_i, IV_i) obtained from the initial processing and IV calculation steps. Penalized B-splines (P-splines) offer a flexible and computationally efficient approach, extending the 1D spline concepts from class to two dimensions. The implementation heavily utilizes sparse matrix capabilities from the `scipy.sparse` library, primarily within the `src/custom_bspline.py` module.

2.3.1 Surface Approximation Goal

The objective is to find a function $V(K, T)$ defined over a rectangular domain $[K_{\min}, K_{\max}] \times [T_{\min}, T_{\max}]$ that provides a good approximation to the observed data points (K_i, T_i, IV_i), where $i = 1, ..., m$, while simultaneously being sufficiently smooth to represent a realistic volatility surface.

2.3.2 Tensor Product B-Spline Basis Representation

The surface $V(K, T)$ is modeled using a basis constructed from the tensor product of 1D B-spline bases in the strike (K) and time-to-maturity (T) dimensions.

- **B-Spline Basis Functions:** The foundation is the 1D B-spline basis of degree p over a knot vector $\mathbf{t} = (t_1, t_2, ..., t_{n+p+1})$. As covered in class, these basis functions $B_{j,p,\mathbf{t}}(x)$ are piecewise polynomials of degree p, possess local support (non-zero only over $p + 1$ adjacent knot intervals), are non-negative, form a partition of unity, and offer controlled smoothness (typically C^{p-1} continuous at simple knots). Our implementation utilizes cubic B-splines ($p = 3$) for both dimensions. The knot vectors $\mathbf{t}_K$ and $\mathbf{t}_T$

are generated using the `calculate_knot_vector` function, which places a specified number of internal knots according to a chosen strategy (fixed to 'uniform' for these experiments) and adds the required boundary knots based on the degree.

- **Evaluating Basis Functions (Cox-de Boor Algorithm):** To calculate the value of a specific B-spline basis function $B_{j,p,\mathbf{t}}(x)$ at a point x, the implementation uses the Cox-de Boor recursion formula, a standard and numerically stable algorithm. This is implemented in the `cox_de_boor(x, k, i, t)` function, where `k` corresponds to the degree p and `i` corresponds to the basis function index j. The algorithm works as follows:

  1. **Base Case:** If the degree $k = 0$, the B-spline $B_{i,0,\mathbf{t}}(x)$ is a simple piecewise constant function: it equals 1 if x is in the i-th knot interval $[t_i, t_{i+1})$ (handling the endpoint t_{i+1} specially for the last interval), and 0 otherwise.
  2. **Recursive Step:** For degree $k > 0$, the value of $B_{i,k,\mathbf{t}}(x)$ is computed as a weighted average of two B-splines of degree $k-1$:

  $$B_{i,k,\mathbf{t}}(x) = \frac{x - t_i}{t_{i+k} - t_i} B_{i,k-1,\mathbf{t}}(x) + \frac{t_{i+k+1} - x}{t_{i+k+1} - t_{i+1}} B_{i+1,k-1,\mathbf{t}}(x) \tag{10}$$

  The implementation includes checks to handle potential division by zero if knots are coincident ($t_{i+k} = t_i$ or $t_{i+k+1} = t_{i+1}$), setting the corresponding term to zero in such cases.
  3. **Iterative Nature:** Although defined recursively, the evaluation process is inherently iterative. To find the value for degree $p = 3$, the function recursively calls itself to compute values for degree 2, which in turn call for degree 1, and finally for degree 0 (the base case). The results are then combined back up the chain.
  4. **Efficiency (Memoization):** The `cox_de_boor` function in the code is decorated with `@lru_cache(maxsize=None)`. This is a form of memoization, meaning the results of function calls with specific arguments (x, k, i, t) are stored. If the function is called again with the exact same arguments, the stored result is returned instantly, avoiding redundant computations. This dramatically speeds up the evaluation process, especially when calculating many basis function values over the same knot vector, as is done when constructing the basis matrices.

- **Tensor Product Construction:** The 2D basis is formed by taking all products $\{\phi_j(K)\psi_l(T) \mid j = 1, ..., n_K; l = 1, ..., n_T\}$ where $\{\phi_j\}$ is the cubic B-spline basis for strike (n_K functions evaluated using Cox-de Boor over $\mathbf{t}_K$) and $\{\psi_l\}$ is the basis for maturity (n_T functions evaluated using Cox-de Boor over $\mathbf{t}_T$). The surface is represented as:

$$V(K, T) = \sum_{j=1}^{n_K} \sum_{l=1}^{n_T} c_{j,l} \phi_j(K) \psi_l(T) \tag{11}$$

  where $c_{j,l}$ are the unknown coefficients.

- **Matrix Formulation and Implementation:** For the m input data points (K_i, T_i, IV_i), the model is $A\mathbf{c} \approx \mathbf{z}$. $\mathbf{z}$ is the vector of IV_i values, $\mathbf{c}$ is the flattened vector of coefficients, and A is the $m \times (n_K n_T)$ design matrix where $A_{i,\text{col}(j,l)} = \phi_j(K_i)\psi_l(T_i)$. The implementation leverages sparsity:

  1. **1D Basis Matrices:** The `evaluate_basis_matrix(eval_points, knots, degree)` function iterates through each `eval_point` (K_i or T_i) and each basis index (j or l), calling the memoized `cox_de_boor` function to compute $\phi_j(K_i)$ and $\psi_l(T_i)$. It assembles these values into sparse matrices B_K and B_T (`scipy.sparse.csc_matrix`).
  2. **Design Matrix Construction (`build_design_matrix`):** This function efficiently builds the sparse tensor product matrix A by iterating through only the non-zero entries of the sparse B_K and B_T, calculating the necessary products $A_{i,\text{col}(j,l)} = (B_K)_{i,j} \times (B_T)_{i,l}$, and constructing the final sparse matrix A using `scipy.sparse.csc_matrix`.

2.3.3 Penalized Least Squares Objective Function

To balance data fidelity with surface smoothness, the P-spline approach minimizes a penalized least squares objective:

$$\min_{\mathbf{c}}\{\|A\mathbf{c}-\mathbf{z}\|_2^2+\lambda\|D\mathbf{c}\|_2^2\} \tag{12}$$

- $\|A\mathbf{c}-\mathbf{z}\|_2^2$ is the fidelity term (sum of squared residuals).
- $\|D\mathbf{c}\|_2^2$ is the penalty term measuring roughness.
- $\lambda \geq 0$ is the smoothing parameter controlling the trade-off.

2.3.4 Penalty Term Construction and Isotropic Smoothing

The penalty term $\|D\mathbf{c}\|_2^2$ penalizes roughness by applying finite difference operations to the coefficient vector $\mathbf{c}$. Second-order differences are used ($d = 2$) as they penalize local curvature, promoting smoother surfaces.

- **1D Difference Matrices:** The `create_difference_matrix(size, order=2)` function generates the 1D second-order difference matrix for a sequence of `size` coefficients. It uses `scipy.sparse.diags` to efficiently construct a sparse matrix representing the operation $y_k = c_{k-1} - 2c_k + c_{k+1}$. This results in sparse matrices D_K (size $(n_K - 2) \times n_K$) and D_T (size $(n_T - 2) \times n_T$).
- **Applying Differences in 2D via Kronecker Product:** A key challenge is applying these 1D difference operators across the 2D grid of coefficients $c_{j,l}$ when they are stored as a flattened vector $\mathbf{c}$. This is achieved using the Kronecker product ($\otimes$), a fundamental matrix operation useful for representing multidimensional operations in linear form. If X is $a \times b$ and Y is $c \times d$, then $X \otimes Y$ is an $(ac) \times (bd)$ block matrix where the (i, j)-th block is the matrix $X_{i,j}Y$. In our implementation (`solve_penalized_bspline_coeffs`):
  - The penalty for roughness along the strike (K) dimension is constructed as $P_K = I_{n_T} \otimes D_K$. Here I_{n_T} is the $n_T \times n_T$ identity matrix (from `scipy.sparse.identity`). The Kronecker product $I_{n_T} \otimes D_K$ effectively creates a large sparse block matrix where the 1D difference operator D_K is applied independently to each "column" of the conceptual $n_K \times n_T$ coefficient grid (i.e., differences are taken along the strike index j for each fixed maturity index l).
  - Similarly, the penalty for roughness along the maturity (T) dimension is $P_T = D_T \otimes I_{n_K}$. This applies the 1D difference operator D_T along the maturity index l for each fixed strike index j.
  - Both P_K and P_T are constructed efficiently using `scipy.sparse.kron`.
- **Isotropic Smoothing:** The overall penalty term in the objective function is conceptually $\lambda(\|P_K\mathbf{c}\|_2^2 + \|P_T\mathbf{c}\|_2^2)$. Since our experiment uses the same λ for both directions (`lambda_x = lambda_y = lambda_val` in `run_experiments.py`), the smoothing is isotropic. This penalty structure is incorporated into the least-squares solution via the augmented system approach described in Section 2.4.

2.3.5 Regularization and Conditioning Analysis

The unpenalized least squares problem ($\lambda = 0$), $\min \|A\mathbf{c}-\mathbf{z}\|_2^2$, can be numerically problematic if the matrix A is ill-conditioned or rank-deficient, which can occur with closely spaced knots or sparse data. The penalty term $\lambda\|D\mathbf{c}\|_2^2$ acts as regularization. The minimization problem (12) is equivalent to solving the augmented least squares problem:

$$\min_{\mathbf{c}}\left\|\begin{bmatrix}A\\ \sqrt{\lambda}P_K\\ \sqrt{\lambda}P_T\end{bmatrix}\mathbf{c}-\begin{bmatrix}\mathbf{z}\\ \mathbf{0}\\ \mathbf{0}\end{bmatrix}\right\|_2^2 \tag{13}$$

(Note: The text combines the penalties into a single D matrix in eq 143, but the implementation uses P_K and P_T separately in the augmented system). For $\lambda > 0$, the augmented system matrix (implicitly related to $A^TA + \lambda(P_K^TP_K + P_T^TP_T)$) is generally better conditioned than A or A^TA. This regularization is essential for obtaining stable and smooth solutions, especially when the number of coefficients is large relative to the number of data points or when the data is noisy.

2.3.6 Bias-Variance Trade-off and Error Characteristics

According to a report from Wabba [**?**] "The parameter λ, which must be chosen, controls the tradeoff between the 'roughness' of the solution, as measured by $\int_0^1 [f^{(m)}(u)]^2 du$ and the infidelity to the data as measured by $\frac{1}{n}\sum_{j=1}^{n}(f(t_j) - y_j)^2$. The problem is to obtain a good value of λ." The choice of the smoothing parameter λ governs a fundamental bias-variance trade-off:"

- **Small λ:** Leads to low bias (the surface fits the data closely) but high variance (the surface is sensitive to noise and may exhibit oscillations). The fit is primarily driven by minimizing $\|A\mathbf{c} - \mathbf{z}\|_2^2$.
- **Large λ:** Leads to high bias (the surface is overly smooth and may not capture the true structure) but low variance (the surface is stable and insensitive to noise). The fit is dominated by minimizing the penalty term $\|D\mathbf{c}\|_2^2$.

Our experiment (`run_experiments.py`) systematically varies λ and evaluates performance primarily using IV MAE to explore this trade-off explicitly.

2.4 LSQR for Solving the Penalized Least Squares System

2.4.1 Augmented System Formulation for LSQR

As indicated in Section 2.3.5, the P-spline minimization problem (12) is solved by reformulating it as a standard (unpenalized) least-squares problem using an augmented system. Let $\tilde{A}$ be the augmented matrix and $\tilde{\mathbf{z}}$ be the augmented right-hand side vector:

$$\tilde{A} = \begin{bmatrix} A \\ \sqrt{\lambda}P_K \\ \sqrt{\lambda}P_T \end{bmatrix}, \quad \tilde{\mathbf{z}} = \begin{bmatrix} \mathbf{z} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} \tag{14}$$

The problem becomes finding $\mathbf{c}$ that minimizes $\|\tilde{A}\mathbf{c} - \tilde{\mathbf{z}}\|_2^2$. The matrix $\tilde{A}$ is typically large and sparse, inheriting sparsity from A, P_K, and P_T. The vector $\mathbf{0}$ has appropriate dimensions corresponding to the number of rows in P_K and P_T.

2.4.2 LSQR Algorithm: Theory and Numerical Stability

The LSQR algorithm, developed by Paige and Saunders [2], is an iterative method specifically designed for solving large, sparse linear systems $M\mathbf{x} = \mathbf{y}$ or least-squares problems $\min \|M\mathbf{x} - \mathbf{y}\|_2^2$. It is particularly well-suited for the potentially large and sparse augmented system $\tilde{A}\mathbf{c} \approx \tilde{\mathbf{z}}$ derived in Section 2.4.1.

LSQR is mathematically equivalent to applying the Conjugate Gradient (CG) method to the normal equations $\tilde{A}^T\tilde{A}\mathbf{c} = \tilde{A}^T\tilde{\mathbf{z}}$. However, a key advantage of LSQR is that it avoids the explicit formation of the matrix $\tilde{A}^T\tilde{A}$. This is crucial for numerical stability because the condition number of the normal equations matrix is the square of the condition number of the original matrix, i.e., $\kappa(\tilde{A}^T\tilde{A}) = [\kappa(\tilde{A})]^2$. Squaring the condition number can lead to significant loss of accuracy in finite precision arithmetic, especially if $\tilde{A}$ is already ill-conditioned [2].

LSQR instead works directly with $\tilde{A}$ and $\tilde{A}^T$ using an underlying Golub-Kahan bidiagonalization process [2]. This process iteratively generates orthonormal bases for the Krylov subspaces $\mathcal{K}_k(\tilde{A}\tilde{A}^T, \tilde{A}\mathbf{c}_0)$ and $\mathcal{K}_k(\tilde{A}^T\tilde{A}, \tilde{A}^T\mathbf{r}_0)$ and produces a lower bidiagonal matrix B_k. The least-squares solution within the Krylov subspace is then found by solving a small, well-conditioned, bidiagonal least-squares problem at each iteration. This approach generally yields more accurate solutions than CG applied to the normal equations (CGNE) when $\tilde{A}$ is ill-conditioned.

The implementation in `src/custom_bspline.py` leverages the `scipy.sparse.linalg.lsqr` function. This function takes the sparse augmented matrix `A_solve` (representing $\tilde{A}$) and the augmented vector `z_solve` (representing $\tilde{\mathbf{z}}$) as primary inputs. The choice of LSQR reflects a preference for numerical robustness when dealing with potentially challenging least-squares problems arising from spline fitting.

2.4.3 Convergence Properties and Stopping Criteria

The convergence rate of LSQR depends on the conditioning and singular value distribution of the matrix $\tilde{A}$. Convergence is typically faster if the singular values cluster away from zero or if the condition number $\kappa(\tilde{A})$ is small. The condition number $\kappa(\tilde{A})$ is influenced by the original design matrix A, the penalty matrices P_K and P_T (and implicitly the difference matrices D_K, D_T), and the smoothing parameter λ.

LSQR iterations continue until a stopping criterion is met. These criteria are designed to terminate the iteration when the current solution $\mathbf{c}_k$ is deemed sufficiently accurate. The `scipy.sparse.linalg.lsqr` implementation uses several parameters to control termination:

- **`atol`, `btol`:** These tolerance parameters relate to the desired accuracy of the solution, typically linked to estimates of the backward error or the norm of the residual for the normal equations [2]. In our implementation (`solve_penalized_bspline_coeffs`), both `atol` and `btol` are set to the value of `lsqr_tol` (defaulting to 10^{-8} in `run_experiments.py`), aiming for a solution $\mathbf{c}_k$ that approximately satisfies conditions related to the normal equations residual norm, roughly $\|\tilde{A}^T(\tilde{\mathbf{z}} - \tilde{A}\mathbf{c}_k)\| \leq \text{tol}\|\tilde{A}\|\|\tilde{\mathbf{z}}\|$.
- **`iter_lim`:** A maximum number of iterations (`lsqr_iter_lim` passed from `run_experiments.py`, defaulting to $\max(20 \times \text{num\_coeffs}, 10000)$) is specified to prevent excessively long runs if convergence is slow.

The implementation captures the outputs from `scipy.sparse.linalg.lsqr`, specifically:

- `coeffs`: The computed solution vector $\mathbf{c}_k$.
- `istop`: An integer code indicating the reason for termination (e.g., 1 or 2 for successful convergence within tolerance, 7 for reaching the iteration limit).
- `itn`: The number of iterations performed.

These captured values (`istop` and `itn`) are then passed up through the function calls and stored in the final results dictionary, allowing for analysis of the solver's convergence behavior, as seen in the generated convergence plots.

2.4.4 Conditioning Impact and Practical Error Sources

As noted, the conditioning of the augmented matrix $\tilde{A}$ is paramount for LSQR's performance. Ill-conditioning, where small changes in input lead to large changes in output, leads to slower convergence and potentially less accurate solutions. The conditioning of $\tilde{A}$ is influenced by several factors inherent in this problem:

- **Data Distribution:** Sparse or clustered option data (K_i, T_i) can lead to a poorly conditioned initial design matrix A.
- **Basis Choice:** Using a high density of B-spline knots (increasing the number of columns in A) can increase the potential for near-linear dependencies between basis functions, worsening conditioning.
- **Smoothing Parameter (λ):** While the penalty term ($\lambda > 0$) generally improves conditioning compared to the unpenalized case ($\lambda = 0$), very small values of λ may not provide sufficient regularization, leaving $\tilde{A}$ ill-conditioned. Conversely, very large λ values, while leading to a well-conditioned numerical problem, might overly smooth the surface (high bias).

Our implementation in `src/custom_bspline.py` directly confronts these issues by using LSQR. The observation in experiments that the non-penalized ($\lambda = 0$) case often requires significantly more iterations (or potentially hits the limit) compared to penalized cases aligns with the theory that the unpenalized system $A\mathbf{c} \approx \mathbf{z}$ can be ill-conditioned.

A key practical error source arises when LSQR fails to converge within the specified tolerances (`lsqr_tol`) before reaching the maximum iteration limit (`iter_lim`). The implementation captures the solver's termination status code (`istop` returned by `scipy.sparse.linalg.lsqr` and stored in the `spline_object`). Status codes other than 0, 1, or 2 (e.g., code 7 indicating the iteration limit was reached) signal that the returned solution $\mathbf{c}_k$ is an approximation whose accuracy may not meet the desired level set by `lsqr_tol`. This

premature termination represents a numerical error introduced by the solver itself, impacting the accuracy of the computed P-spline coefficients $\mathbf{c}$ and, consequently, the final interpolated volatility surface $V(K, T)$. This highlights the practical interplay between the choice of basis flexibility (knot density), regularization strength (λ), the inherent conditioning of the specific dataset, and the limitations of iterative solvers.

2.5 Importance of Data Filtering for Keeping Numerical Solutions Stable

The data preprocessing process in the `prepare_volatility_data` function is important not only for tidying the received financial signals, but also for preparing them for a numerically stable and accurate P-spline fitting. However, raw option data acquired from the market tends to have entries that are unsuitable for direct quantitative modeling. We apply several filters following the 'medium' preset provided in `run_experiments.py`:

- **Relative Bid-Ask Spread:** We remove options where (Ask-Bid)/Mid Price > 0.30. Such large spreads are likely to reflect low liquidity, price errors or outdated quotes, meaning the given mid-price is less likely to make sense as fair market value.
- **Minimum Time-to-Maturity:** Options are included only if $T \geq 0.01$ years (which amounts to 3-4 days). Options too close to expiry exhibit complex (for example very sensitive to underlier price jumps, have near zero Vega) behavior that can be hard for the standard model (e.g. Black-Scholes) and smooth surface fits to pin down well. By removing them, any potential issues related to numerical calculations are also mitigated in both IV calculation and surface fitting.
- **Minimum Volume:** We exclude options with a trading volume of less than 3 contracts. Very low volume suggests there is no clear market consensus so prices could very easily be at a level of disconnect.
- **(Other Filters to Study):** The logic that includes filtering by absolute bid-ask spread, open interest and source implied volatility range is encapsulated in the `prepare_volatility_data` function, although these may not have been the bottleneck under the 'medium' preset, in the main experiment.

Outliers are to be weeded out as far as possible in order to receive the following benefits:

1. **Better Input:** Give the P-spline fitting process (and IV calculation) better input ($\mathbf{z}$) that accurately represent the underlying volatility signal and not market-microstructure noise or mistakes.
2. **Improve Numerical Stability:** Weights often mitigate outliers that could compromise the least-squares solution. The presence of very noisy or severely corrupted data points can also indirectly degrade the conditioning of the design matrix A and make the penalized least-squares problem more difficult for LSQR to solve accurately.

Thus data filtering is not only a financial data quality issue, but also a practical necessity embedded into the numerical workflow to enhance the robustness and reliability of the surface fitting process.

3 Experimental Setup

This section details the computational environment, raw data sources, data cleaning, some data processing steps, hyper parameter selections, and evaluation criteria that were used to run the numerical experiments for modeling the implied volatility surface. This framework allows us to systematically explore the performance of the P-spline approach, particularly the effect of the smoothing parameter λ.

3.1 Computational Setting and Software Toolboxes

Experiments were carried out under and ran on Python 3. The implementation makes extensive use of various standard scientific computing libraries, as would be standard practice in numerical analysis and data science:

- **NumPy:** For numerical operations, including manipulations of multi-dimensional arrays (`np.array`, `np.unique`, `np.mean`), amongst others.), mathematical functions (e.g., `np.sqrt`, `np.log`, `np.abs`, `np.nan`), linear algebra access (`np.linalg.solve` used in the 1D spline solver).
- **SciPy:** Contributed core shell capabilities beyond NumPy including the following:
  - `scipy.stats`: In particular `scipy.stats.norm`, used for its cumulative distribution function (`cdf`, Φ) and probability density function (`pdf`, ϕ) required in the Black-Scholes formula and Vega computation (Section 2.1).
  - `scipy.sparse`: Widely used for efficient treatment of large, sparse matrices that occur in B-spline bases and difference operators. These include functions for constructing sparse matrices (`csc_matrix`, `identity`, `diags`), operating on sparse matrices (`vstack`, `kron`), and calling sparse linear algebra routines.
  - `scipy.sparse.linalg`: And in particular `scipy.sparse.linalg.lsqr`, the iterative solver adopted to solve the large and sparse penalized least-squares system stemming from the P-spline specification (Section 2.4).
- **Pandas:** Utilized for data handling. Very common for reading CSV data (`pd.read_csv`), data cleaning (dealing with NaNs using `dropna`, type conversion using `pd.to_numeric`), filtering on various criteria on data, grouping data (`groupby`, `agg`), displaying and organizing results into DataFrames for analysis and for saving.
- **Matplotlib / Seaborn:** Used to produce plots and images in the `run_experiments.py` script's aggregation phase (Section 5 & 6) to analyze the results graphically (e.g., histograms, scatter plots, box plots).
- **yfinance:** Used in `src/market_data_utils.py` (specifically `get_spot_price`) and maybe for recording data (but commented out in the supplied `run_experiments.py`) for downloading historical option chain data and spot prices from Yahoo! Finance.
- **datetime / time:** Standard Python modules to work with dates (`date`, `timedelta`, `datetime` are used in `get_analysis_date`, `calculate_time_to_maturity`) and timing computations (`time.time`).
- **glob / os / sys / itertools / traceback / warnings:** Built-in Python libraries for file system manipulations, path manipulations, iteration through parameters, error handling, and control of warnings.

3.2 Data Sources and Collection Procedures

There were two basic data types that were needed:

- **Option Market Data:** Daily call option chain data for a portfolio of 17 underlying assets was considered. The specific tickers we used for the experiment are: AAPL, MSFT, GOOG, SPY, V, TSLA, NVDA, AMD, T, JPM, MA, NFLX, COST, UNH, JNJ, CSCO, CRM. This data is assumed to reside as CSV files (e.g., `TICKER_call_options_YYYY-MM-DD.csv`) in the `data/raw/` directory which contains the strike price, expiration date, bid/ask prices, volume, open interest, the last traded price and the reported implied volatility by the source. The script uses data corresponding to the most recent weekday prior to the execution date, determined by the `get_analysis_date` function.
- **Risk-Free Rate Data:** Daily U.S. Treasury par yield curve rates for the same `analysis_date` were fetched directly from the U.S. Department of the Treasury website via the `fetch_daily_yield_curve` function in `src/market_data_utils.py`. Thereof standard maturities (1-month to 30-year) as input knots (T_j, r_j) for the 1D cubic spline interpolation (Section 2.2) can be calculated.

3.3 Preprocessing, IV Calculation, and Filtering of Data

Subsequent spline fitting tends to operate on heavily processed market data; most of which is done previously by `prepare_volatility_data` and the first block of `analyze_interpolation_errors`:

- **Initial Loading & TTM:** For the given ticker, related CSV files are loaded. For each contract, time-to-maturity (T) is calculated in years with `calculate_time_to_maturity` as the difference between the time of expiration (obtained from filename) and the `analysis_date`.
- **Market Price:** We take market price (C_{market}) as middle price of the bid and ask prices. If trade/bid is invalid or unused the last traded price is used as a backup. Choices for which there is no legitimate mid-price or last price are rejected.
- **Initial Filtering (`prepare_volatility_data`):** Before IV recalculation, `prepare_volatility_data` performs filterings based on the 'medium' preset:
  1. Relative Spread: (Ask-Bid)/Mid Price ≤ 0.30.
  2. Minimum TTM: $T \geq 0.01$ years.
  3. Minimum Volume: Daily volume ≥ 3 contracts.
  4. Other filters (e.g., absolute spread, open interest, source IV bounds) are mentioned but might not indeed restrict under the 'medium' setting.

  Options that do not pass these tests are rejected.
- **Grouping (`prepare_volatility_data`):** Those data that have not been filtered in the initial step were grouped by their unique strike (K) and time to maturity tracking variables (T) respectively. To any repeats (K, T) of values, we assign the average between `market_mid_price` and `impliedVolatility` from the input file with `groupby().agg(..., 'mean')`. The resultant DataFrame consists of the distinct (K, T) points with the averaged market prices and the initial IVs.
- **Implied Volatility Recalculation (`analyze_interpolation_errors`):** This is an essential task done after the initial preparation.
  1. The original (averaged) implied volatility is named `market_iv_original` to be kept to compare later on.
  2. The script loops over all the distinct (K, T) row in the preprocessed data.
  3. It calls the function `implied_volatility` (Sect. 2.1.2) for each strike so that the equation $C_{\text{BS}}(S, K, T, r(T), \sigma) - C_{\text{market}} = 0$ is solved for σ. This model depends on the loaded spot price S, the option's K and T, the market mid-price C_{market}, and the risk-free rate $r(T)$ that is the output of the 1D interpolation with `rate_interpolator`. When possible the original `market_iv_original` serves as the guessing point for Newton-Raphson.
  4. Results dictionary of `implied_volatility` (`implied_volatility`, `iterations`, `final_diff`, `status`) is stored for every option.
  5. **Post-IV Calculation Filtering:** Options for which the IV calculation did not converge (i.e., status was not 'Converged', the returned IV was NaN or non-positive) are filtered out. Only the options with a positively-fitted IV are passed to the surface-fitting phase.
- **Final Input for Splines:** The input data ($\mathbf{z}$) for the calculation of the P-spline fits are the recalculated implied volatilities of the options that have passed all the filtering and IV calculation stages successfully.

3.4 Experimental Design and Parameter Space

The experiment focuses on the impact of the P-spline smoothing parameter λ.

- **Fixed Parameters:** To isolate the effect of λ, the following were held constant across all runs:
  - Spline Degree: $(p_K, p_T) = (3, 3)$ (Cubic B-splines).

- Knot Placement Strategy: 'uniform'.
- Internal Knot Density ('medium'): 8 knots (strike), 5 knots (maturity). Total basis functions: $(8+3+1) \times (5+3+1) = 12 \times 9 = 108$.
- Minimum Knot Separation: 1.0 (strike), 0.01 (maturity).
- Penalty Order: $d = 2$.
- Data Filtering: 'medium' preset (Rel Spread ≤ 0.30, Min TTM ≥ 0.01, Min Vol ≥ 3).
- Max Maturity Considered: $T \leq 1.5$ years.
- LSQR Solver Tolerances: `atol` = `btol` = 10^{-8}.
- LSQR Max Iterations: `iter_lim` = 10000 (default, unless overridden by `SOLVER_PARAMS`).

- **Varied Parameters:** The following were systematically varied:
  - Ticker Symbol: The process was repeated for each ticker in the list: ['AAPL', 'MSFT', 'GOOG', 'SPY', 'V', 'TSLA', 'NVDA', 'AMD', 'T', 'JPM', 'MA', 'NFLX', 'COST', 'UNH', 'JNJ', 'CSCO', 'CRM'].
  - Isotropic Smoothing Parameter (λ): Tested values were $\lambda \in \{0.0, 0.01, 0.1, 0.5, 1.0, 2.5, 5.0, 10.0, 15.0, 20.0\}$. Note that $\lambda = 0$ corresponds to an unpenalized B-spline least-squares fit.

3.5 Performance Evaluation Metrics

The performance of each fitted volatility surface $V(K,T)$ (for each Ticker/Lambda combination) was evaluated using metrics calculated by comparing model outputs against the market data for the m' options that survived all filtering and IV recalculation steps:

- **Implied Volatility Mean Absolute Error (IV MAE):** The primary metric, comparing the spline's interpolated IV ($V(K_i, T_i)$) against the original market IV ($\texttt{market\_iv\_original}_i$):

$$\text{IV MAE} = \frac{1}{m'} \sum_{i=1}^{m'} |V(K_i, T_i) - \text{market\_iv\_original}_i| \tag{15}$$

- **Implied Volatility Root Mean Squared Error (IV RMSE):**

$$\text{IV RMSE} = \sqrt{\frac{1}{m'} \sum_{i=1}^{m'} (V(K_i, T_i) - \text{market\_iv\_original}_i)^2} \tag{16}$$

- **Option Price Mean Absolute Error (Price MAE):** Compares the Black-Scholes price calculated using the interpolated IV ($C_{\text{fitted},i} = C_{\text{BS}}(S, K_i, T_i, r_i, V(K_i, T_i))$) against the observed market mid-price ($C_{\text{market},i}$):

$$\text{Price MAE} = \frac{1}{m'} \sum_{i=1}^{m'} |C_{\text{fitted},i} - C_{\text{market},i}| \tag{17}$$

- **Option Price Root Mean Squared Error (Price RMSE):**

$$\text{Price RMSE} = \sqrt{\frac{1}{m'} \sum_{i=1}^{m'} (C_{\text{fitted},i} - C_{\text{market},i})^2} \tag{18}$$

- **Computation Time:** Wall-clock time for the `analyze_interpolation_errors` function call.
- **Convergence Metrics:**

- **IV Recalculation:** Average Newton-Raphson iterations (`avg_iv_iterations`) and the success rate (`iv_success_rate`) across all options attempted for a given run.
- **Spline Fitting:** LSQR solver iterations (`spline_solver_iterations`) and final status code (`spline_solver_status`).

These metrics are calculated within `analyze_interpolation_errors` and stored in the results dictionary for each run in `run_experiments.py`.

4 Results

The numerical experiments for the effect of the P-spline smoothing parameter (λ) on the implied volatility surface construction are reported in this section, in terms of quantitative and qualitative outcomes. All 170 configured runs (17 tickers $\times$ 10 lambda values) completed successfully. The quantitative analysis relies on aggregated data from 50,860 filtered option data points where implied volatility was successfully recalculated, while the qualitative analysis is based on visual inspection of generated surface plots, particularly for AAPL and JPM at key λ values.

4.1 Influence of Smoothing Parameter (λ) on Fit Precision

The quantitative metrics reveal a clear trade-off between data fidelity and smoothing.

- **IV MAE & Price MAE vs. λ:** The average IV MAE across all tickers was minimized with $\lambda = 0.00$ and $\lambda = 0.0001$ (both 0.0639) and increased monotonically to 0.0831 at $\lambda = 20.0$. Similarly, the average Price MAE was the smallest at $\lambda = 0.00$ ($0.3488) and monotonically increased as λ increased. This would suggest that the best fit according to these average error measures, when comparing against the recalculated IVs and the market prices, is provided by minimal or no smoothing. The average IV MAE for the entire dataset was 0.0737 ($\text{Min} = 0.0257, \text{Max} = 0.1983$).
- **Top Performing Runs:** Among the high-performing (Top 5) runs by IV MAE, the majority included either $\lambda = 0.00$ or $\lambda = 0.0001$ for JPM and V. However, as we discussed in Section 4.5, minimizing these quantitative errors does not solely determine the most practical or reliable surface.

4.2 Analysis of Convergence

The numerical methods used showed generally good convergent behavior.

- **Implied Volatility Determination (Newton-Raphson):** The recalculation step worked very well with an average success rate of 97.6% and required an average of 4.26 iterations per successful calculation.
- **Spline Fitting (LSQR Solver):** LSQR converged successfully (Status Code 2) for all runs. The number of iterations was heavily dependent on λ: the maximum number of iterations (1458) was reached for unpenalized fits ($\lambda = 0$), but the median was only 125.0, implying that convergence is much faster if $\lambda > 0$.

4.3 Computational Performance

Matching the LSQR iteration counts, runs with $\lambda = 0$ were noticeably slower than penalized runs ($\lambda > 0$). Introducing even a minimum amount of smoothing ($\lambda = 0.0001$) resulted in substantial computational speed-up. The overall execution time for 170 runs was 1.34 minutes.

4.4 Performance Across Tickers

Performance was quite varied from ticker to ticker. JPM and V exhibited the lowest average IV MAE, while COST, UNH, and NFLX had the highest values, indicating that the tasks might require varying levels of complexity or that the data could be of varying quality. The best λ (minimizing IV MAE) also changed, but for most tickers, $\lambda \leq 0.001$ worked best.

4.5 Visual Judgement for Surface Quality

Although the quantitative measures prefer small λ values, it is important to visually inspect the fitted surfaces at different smoothing levels ($\lambda = 0.0, 0.01, 10.0$) for representative tickers (AAPL and JPM) to judge the practical utility and reveal the shortcomings of relying solely on MAE.

- **Unpenalized Fit ($\lambda = 0.0$):** Surfaces generated when no smoothing is applied, such as for JPM (which displayed the lowest overall IV MAE), visually demonstrate a very close fit to the input re-calculated IV data points. This is however accompanied with highly volatile boundary oscillations, in particular at the extremities of the strike and maturity ranges (cf. Figure 1 and Figure 2). This behavior is probably due to the spline trying to fit noise or accommodate sparse/irregular data near the boundaries, a common problem where extrapolation or fitting is less constrained. Although such oscillations are quantitatively "correct" against the input calibration data, they render the surface financially implausible and unreliable for practical tasks (such as exotic options pricing or risk management).

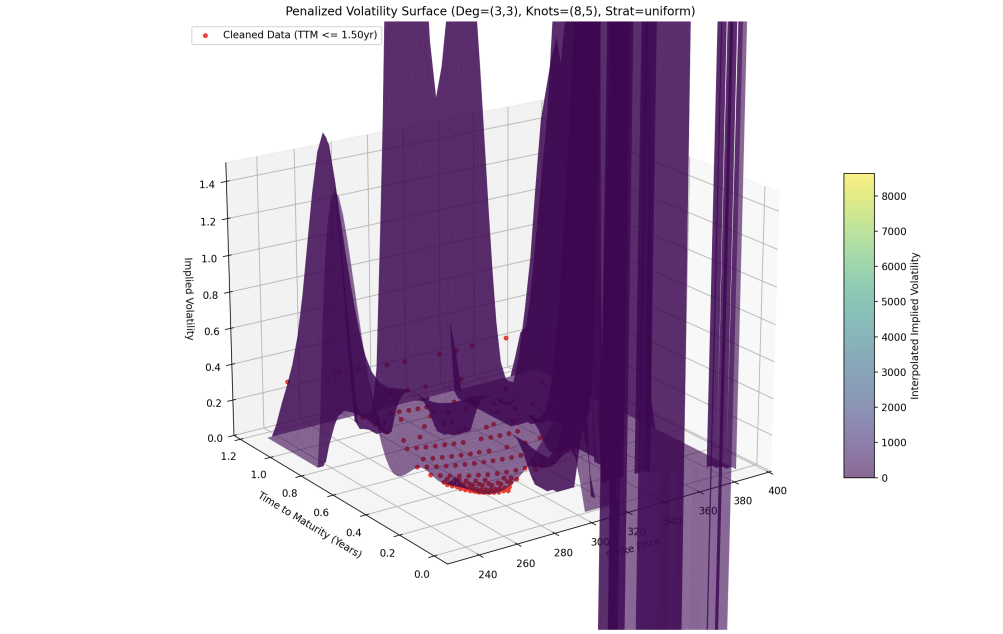

Figure 1: The fitted IVS for JPM with $\lambda = 0.0$.

- **Minimally Smoothed Fit ($\lambda = 0.0001$):** Introducing even a very small amount of penalization, as seen for JPM and AAPL with $\lambda = 0.0001$ (Figures 3 and 4), significantly suppresses the extreme oscillatory behavior seen at $\lambda = 0$. The surfaces look a lot more stable, not least near the boundaries, while remaining visually similar to the shape suggested by the measured data points. This configuration also leads to an IV MAE nearly the same as the $\lambda = 0$ case (average 0.0639) while producing a more visually plausible appearance of the surface. Some specific data points might lie slightly further from this surface compared to the $\lambda = 0$ fit, indicative of the effect of the smoothing term.
- **Highly Smoothed Fit ($\lambda = 10.0$):** When using a high penalty ($\lambda = 10.0$, see Figures 5 and 6), the resulting surfaces are very smooth and display virtually no oscillation. Unfortunately, this comes at the cost of data fidelity; the surfaces are appreciably flattened, which may mean that important features like the steepness of the volatility smile or skew present in the data are lost. This visual oversmoothing is consistent with the increased IV MAE observed at larger λ values. The strong penalty favors a simple form, which reduces variance but increases bias.
- **Data Shape Influence (JPM Example):** JPM, particularly at $\lambda = 0$, has an extremely low IV MAE which could partly be attributed to the shape of its volatility data after imposing the filters.

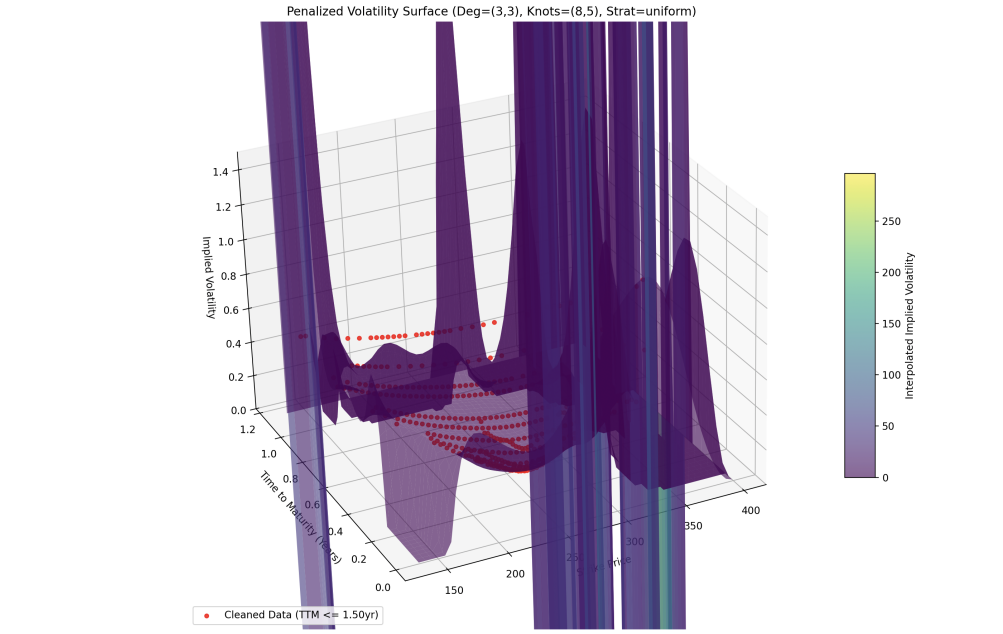

Figure 2: Fitted IVS for AAPL with $\lambda = 0.0$.

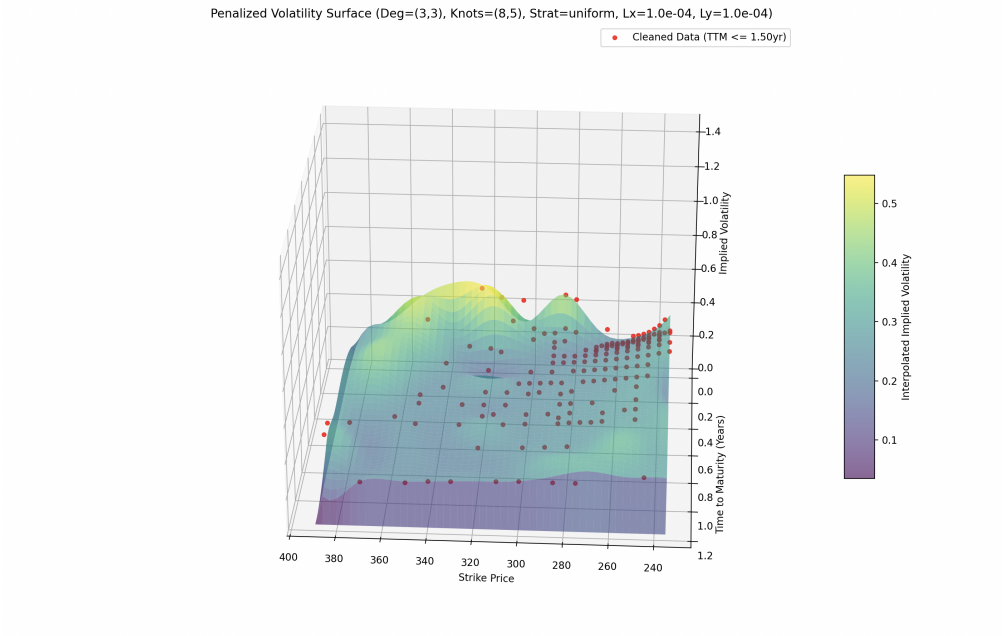

Figure 3: Fitted IVS for JPM with $\lambda = 0.0001$.

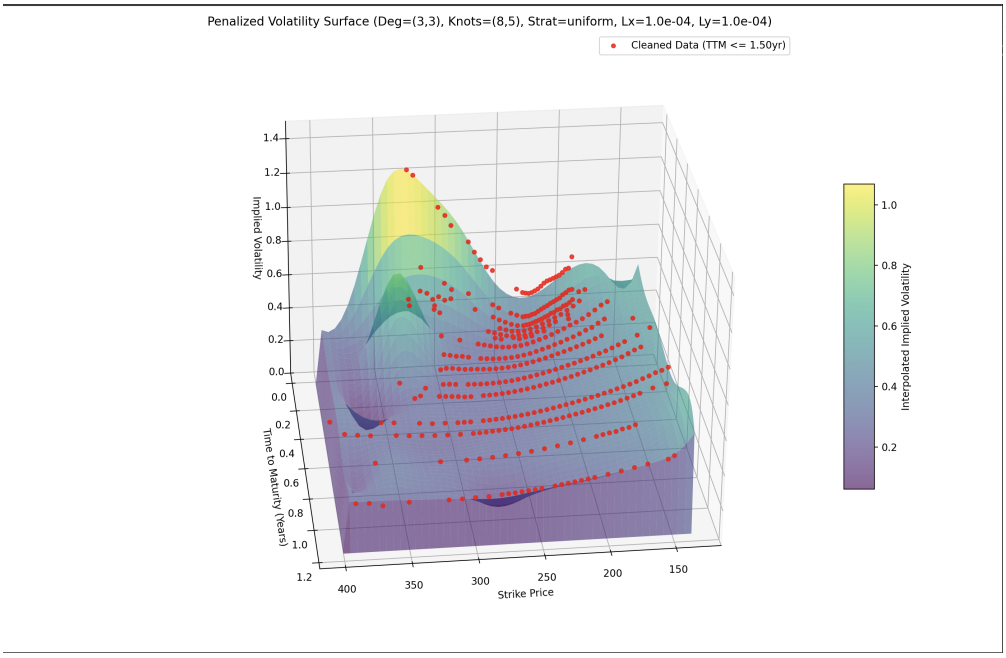

Figure 4: Fitted IVS for AAPL with $\lambda = 0.0001$.

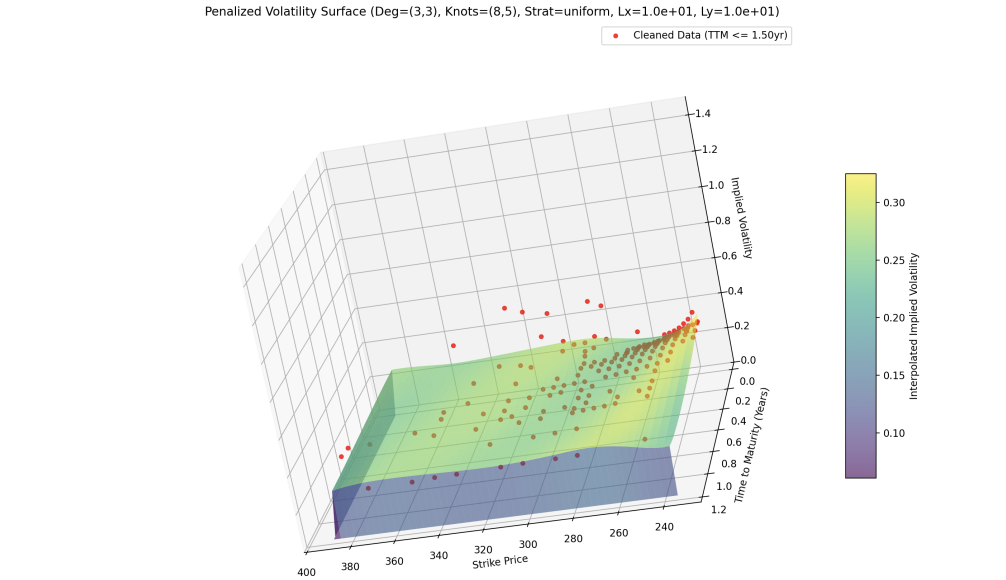

Figure 5: Fitted IVS for JPM at $\lambda = 10.0$.

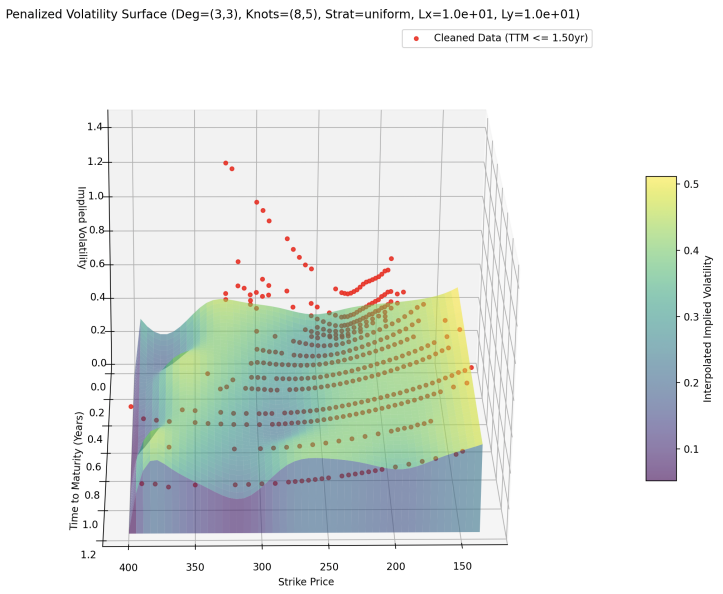

Figure 6: Fitted IVS for AAPL with $\lambda = 10.0$.

Visual examination (Figure 1 or 3) shows that the JPM data used for fitting exhibited a relatively less complex structure than some of the other tickers, making it easier for the spline to get a good fit even without penalization (though still exhibiting oscillations).

- **Useful Lambda Range:** Based on balancing the quantitative results (low MAE) with the qualitative visual assessment (avoiding oscillation but maintaining shape), a λ value in the approximate range $\lambda \in [0.01, 1.0]$ appears most promising for this specific model configuration and dataset. Values less than 0.01 may risk instability and overfitting noise, and values much greater than 1.0 may lead to oversmoothing and loss of important market trends.

This visual inspection highlights the need for qualitative evaluation of surface fitting models in addition to quantitative error metrics. The parameter λ provides essential control over the smoothness-fidelity trade-off, but its optimal selection may depend on the specific application and user judgment beyond minimizing a single error metric.

5 Discussion

The results presented in Section 4 provide valuable insights into the performance and numerical characteristics of using Penalized B-Splines (P-Splines) with an LSQR solver to build implied volatility surfaces. We interpret these results in this section, relating them to the underlying numerical theory and discussing their practical implications and limitations.

5.1 Interpretation of the Trade-off between Accuracy and Smoothness

The key parameter studied was the smoothing penalty λ. The quantitative results consistently showed that the lowest average Implied Volatility MAE and Price MAE were achieved at or very close to $\lambda = 0$. This result, even if seemingly suggesting that no penalization is best, must be weighed against the visual inspection (Section 4.5) as well as the methodology used.

The input implied volatilities for the spline fitting (the $\mathbf{z}$ vector) were recalibrated using the Black-Scholes model and observed market prices. Thus, such input IVs are inherently already very compatible with the

option-pricing model. An unpenalized spline fit ($\lambda = 0$), having the highest flexibility, naturally tries to minimize the deviation from these self-consistent input points, and hence achieves the lowest value of MAE.

Unfortunately, the surfaces computed with $\lambda = 0$, as discussed in the visual summary (Section 4.5), showed large, economically unreasonable oscillations, especially in the vicinity of the edges of the strike and maturity region. This suggests that the unpenalized spline was probably fitting noise inherent in the market prices (which can persist even after IV recalculation) and/or struggling with sparse data coverage near the edges where extrapolation occurs. These high amplitude "boundary oscillations" are a signature of instabilities usually found when using highly flexible models without strong enough regularization.

Introducing the slightest penalty ($\lambda = 0.0001$) was seen to lead to a much better visual quality, with the oscillations attenuated considerably, and with a negligible effect on the average IV MAE (which still tied for the lowest mean value). This demonstrates the stabilizing contribution of the P-spline penalty. For larger values of λ, such as $\lambda = 10.0$ or 20.0, the surfaces became very smooth, with no visible oscillations, but at the expense of materially worse IV and Price MAE, and a visually "flattened" look (Section 4.5). This illustrates the bias-variance trade-off inherent in penalized methods (Section 2.3.6): large λ decreases variance (smooth, steady fit) but increases bias (oversimplified surface deviates systematically from data).

Given both the quantitative measures (favoring $\lambda \approx 0$) and the crucial qualitative visual test (showing instability for $\lambda = 0$ and oversmoothing for large λ), these results indicate that a practical optimal smoothing value for this setup is likely to be within the range $\lambda \in [0.0001, 1.0]$. In this interval, the penalty is sufficient to curb most unrealistic oscillations whereas the spline is still able to capture the basic actual form of the volatility surface, thus striking a reasonable balance between data fidelity and smoothness. The choice within this range may depend on the specific application's tolerance for noise versus potential bias.

5.2 Performance and Convergence analysis of the Numerical Method

The implemented numerical algorithms performed reliably and effectively within the context of the experiment.

- **Newton-Raphson (IV):** When implemented on a reasonably pre-screened set of market data where issues such as near-zero Vega are less prevalent, the method is efficient in finding implied volatility, producing excellent success rates ($> 96\%$ on average) with a low average number of iterations utilized (≈ 4).
- **LSQR Solver (Spline Fitting):** The LSQR algorithm converged successfully (status code 2) in all 170 runs, which confirms its stability for solving the large, sparse least-squares systems of equations resulting from the P-spline formulation. The drastic decrease in the number of required iterations when moving from $\lambda = 0$ to $\lambda \geq 0.0001$ shows the important empirical benefit obtained by Tikhonov regularisation (Section 2.3.5). The penalty term is helpful for conditioning the augmented system; the iterative solver can therefore converge towards a solution satisfying the tolerance much faster. De facto, this translated directly into faster computing times.

5.3 Data and Ticker Influence

The performance differences for different tickers were highlighted in the results (Section 4.4). For tickers such as JPM and V, one could observe lower fitting errors than for UNH or NFLX. Such variation may arise from a number of sources, such as natural discrepancies in the complexity and smoothness of the true underlying volatility surfaces, differences in the noise level for the market data of each ticker even after filtering, or inhomogeneities in the data distribution over the strike-maturity grid. The finding that the lowest overall MAE occurs for the JPM $\lambda = 0$ run may also be affected, as noted visually (Section 4.5), by the filtered JPM dataset potentially having a relatively simpler structure, making it easier to fit closely even without penalization (though still exhibiting oscillations/"flutterings"). Furthermore, the optimal λ (the one that results in the smallest IV MAE) also differed slightly for each ticker, which indicated that slight gains may be achieved if the smoothing parameter were adjusted on an asset-by-asset basis.

5.4 Limitations

The following limitations should be taken into account when drawing conclusions from this study:

- **Constant Spline-Function Structure:** We considered only cubic splines of one (standard) knot frequency with uniform located knots. Other values of degrees or knots may affect results as well and even interact with the best choice of the smoothing parameter.
- **Isotropic Smoothing:** Smoothing with the same λ in both the S and T dimensions may not be ideal if the true smoothness of the surface is very different along the two directions in the (S, T)-plane (along the strike versus maturity).
- **Single Date:** The analysis is restricted to 1 market date data. The form of the volatility surface, as well as its properties, evolve and results could be different in different market conditions.
- **Metric Dependence:** The "optimal" λ discovered is sensitive to the choice of metric (IV MAE vs. Price MAE vs. visual inspection). The fact that we focus on the minimization of one does not imply that other alternatives as well as for financial practical uses are optimal.

6 Conclusion

In this project, we introduced and tested a P-spline based approach to building implied volatility surfaces and demonstrated use of all of the primary numerical methods including the Newton-Raphson root finder, 1D cubic spline interpolation, 2D tensor-product B-splines with penalization, and the LSQR iterative solver.

We observed the significant impact of smoothness parameter λ in the P-spline framework from the experimental studies. Although unpenalized fits ($\lambda = 0$) gave the lowest quantitative errors (MAE) when compared against the returned input data, visual examination indicated fits with $\lambda = 0$ suffered from unrealistic oscillations. The introduction of even minimal penalization ($\lambda \geq 0.0001$) already significantly visually stabilized the surface, and led to a much more efficient computation that required fewer iterations of LSQR at the cost of only slightly worse quantitative metrics. For higher λ values, the surfaces were too smooth and flattened. This underscores the inherent trade-off between bias and variance in penalized fitting and demonstrates the need for a juxtaposition of quantitative criteria and qualitative visual investigation when it comes to actual model choice. For the configuration probed here an optimal value of λ between 0.01 and 1.0 seemed to supply the best trade-off.

The Newton-Raphson and LSQR-procedure performed stable and remarkably fast in this experiment; LSQR even gained from the regularization by $\lambda > 0$. Differences in performance between tickers indicate that the best model parameters might depend on the asset.

In general, the P-spline with LSQR method provides a computationally feasible and efficient tool for fitting the IV surfaces from noisy market data. The smoothness parameter λ plays a critical role in determining the appearance of the computed surface, and the appropriate choice of λ must take into account the trade-off between fitting the data, visual acceptability, and numerical stability. Adaptive knot placement, anisotropic smoothing, and cross-validation methods for a more automatic and robust choice of parameter values under varying market conditions could be studied in further work.

References

[1] Burden, R. L., Faires, J. D. (2011). *Numerical Analysis* (9th ed.). Brooks/Cole, Cengage Learning.

[2] Paige, C. C., Saunders, M. A. (1982). LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software (TOMS)*, 8(1), 43–71..

[3] Craven, P., Wahba, G. (1979). Smoothing noisy data with spline functions: estimating the optimal value of the smoothing parameter. *Numerische Mathematik*, 31(4), 377-403.