Second Order Greeks via Adjoint Algorithmic Differentiation

Summary Report

Junru Wang Zhaolin Shi Zhengtao Yu
Yaru Guan Zilin Mao Wenqi Guo

Master of Science in Financial Engineering
University of Illinois at Urbana-Champaign
Corporate Advisor: Gan Wang, Vicky Luo (JPMorgan Chase)
Instructor: Prof. Feng

December 12, 2025

Abstract

This report presents a comprehensive implementation of Adjoint Algorithmic Differentiation (AAD) for computing second-order Greeks in financial derivatives pricing. We developed a tape-based reverse-mode automatic differentiation framework with three Hessian computation methods: **Edge-Pushing** for sparse Hessians, **Taylor Expansion** for dense Hessians, and **Bumping2** (finite difference) for validation. Our implementation achieves up to **42× speedup** over traditional finite difference methods for PDE-based pricing with B-spline local volatility surfaces. The framework has been validated on real SPX options data with 5,405 contracts.

Key Words: Adjoint Algorithmic Differentiation (AAD), Second-Order Greeks, Hessian Computation, Edge-Pushing Algorithm, Taylor Expansion, B-spline Local Volatility, Monte Carlo Simulation, PDE Pricing, Cython Optimization, Financial Derivatives

Executive Summary

This practicum project, sponsored by JPMorgan Chase, developed a production-ready Adjoint Algorithmic Differentiation (AAD) framework for computing second-order Greeks (Gamma, Vanna, Volga) in financial derivatives pricing. Traditional finite-difference "bumping" methods scale poorly with parameter dimensionality, requiring $O(n^2)$ re-pricings to compute a full Hessian matrix. Our AAD-based approach reduces this to $O(1)$ pricing evaluations, achieving dramatic computational savings.

Key Technical Achievements:

- **Core AAD Engine:** Implemented a tape-based reverse-mode automatic differentiation system with Python operator overloading, enabling seamless integration with existing pricing models.
- **Edge-Pushing Algorithm:** Developed an efficient sparse Hessian computation method that exploits the locality of B-spline basis functions, achieving **42× speedup** over finite differences for PDE-based calibration.
- **Taylor Expansion Method:** Implemented an alternative dense Hessian approach particularly suited for Monte Carlo simulations with correlated paths.
- **Cython Optimization:** Achieved **12× performance improvement** over pure Python through compiled Cython extensions.
- **Real Data Validation:** Successfully calibrated B-spline local volatility surfaces on SPX options (5,405 contracts, $S_0 = \$6081.76$) with Hessian errors in the range 10^{-8} to 10^{-14}.

Business Impact:

- **Risk Management:** Enables real-time computation of second-order sensitivities for large portfolios, supporting more accurate hedging strategies.
- **Model Calibration:** Provides exact Hessians for second-order optimization methods (Newton, L-BFGS), improving convergence speed and robustness.
- **Scalability:** Framework handles high-dimensional parameter spaces (tested up to 40 assets) where finite differences become computationally prohibitive.

Recommendations: The Edge-Pushing method is recommended for PDE-based pricing with structured parameter spaces (B-splines, piecewise models), while Taylor Expansion is preferred for Monte Carlo simulations with dense Hessians. Future enhancements include GPU acceleration for large-scale simulations and checkpointing for memory-constrained environments.

Contents

1 Introduction

Modern risk management for large derivative portfolios requires accurate and timely computation of sensitivities with respect to a high-dimensional set of model parameters. In practice, these sensitivities, commonly referred to as Greeks, are used for hedging, capital allocation, stress testing, and model calibration. As financial models grow in complexity, incorporating Monte Carlo simulation, partial differential equation (PDE) solvers, and high-dimensional volatility parameterizations, the computational burden of traditional sensitivity analysis has become a critical bottleneck in production risk systems, often rendering higher-order Greeks prohibitively expensive or altogether unavailable in time-sensitive risk and hedging workflows.

1.1 Project Motivation

From the perspective of a large investment bank such as J.P. Morgan, the dominant industry approach for computing Greeks remains finite-difference bumping. While conceptually simple, bumping scales poorly: computing n first-order Greeks requires $O(n)$ full re-valuations, and computing a full Hessian of second-order Greeks requires $O(n^2)$ re-pricings. For models that already involve expensive numerical engines, such as Monte Carlo simulations with early exercise features or PDE solvers with local volatility surfaces, this quadratic scaling quickly becomes infeasible in real-time or overnight risk pipelines.

Adjoint Algorithmic Differentiation (AAD) has emerged over the past decade as a promising alternative. Reverse-mode AAD exploits the structure of the computation graph underlying a pricing function to compute all first-order sensitivities in a single backward sweep, with computational cost largely independent of the number of input parameters. Extending this efficiency to second-order Greeks, however, is substantially more challenging. Naïve extensions of reverse-mode AAD either suffer from prohibitive memory usage or lose the computational advantages that make AAD attractive in the first place.

This challenge is of direct interest to the project sponsor. Second-order Greeks such as Gamma, Vanna, and Volga play a central role in nonlinear risk management, model calibration, and hedging under stressed market conditions. In many production environments, however, these quantities are computed only approximately or are excluded altogether, as their evaluation requires a prohibitively large number of re-valuations when using conventional finite-difference methods. Developing a practical and scalable framework for computing second-order Greeks, while remaining compatible with realistic pricing engines and real market data, therefore addresses a concrete and high-impact problem faced by front-office quantitative risk teams.

1.2 Project Scope

The scope of this project is to design, implement, and validate a unified AAD-based framework for the efficient computation of both first-order and second-order Greeks in modern derivatives pricing models. We adopt reverse-mode AAD as the core differentiation paradigm and investigate two complementary techniques for Hessian computation, namely the Edge-Pushing algorithm for sparse Hessians and the Taylor Expansion method for dense Hessians. These methods are integrated into both PDE-based and Monte Carlo-based pricing engines and are evaluated against traditional finite-difference benchmarks.

1.3 Team Objectives

To achieve the goals outlined above, the project is organized around the following objectives:

- Develop a tape-based reverse-mode AAD engine using Python operator overloading, enabling systematic construction and traversal of computation graphs.

- Implement the Edge-Pushing algorithm to compute second-order derivatives efficiently by exploiting sparsity in structured models such as B-spline local volatility PDEs.
- Implement a Taylor Expansion–based differentiation engine to handle dense Hessians arising in Monte Carlo simulations.
- Integrate the AAD framework with multiple pricing engines, including Crank–Nicolson PDE solvers and Monte Carlo basket option pricers.
- Address non-smooth payoffs and early exercise features using smoothing techniques and frozen exercise-time strategies suitable for higher-order differentiation.
- Optimize the implementation for practical performance using Cython acceleration and memory-aware design choices.
- Validate accuracy and scalability against finite-difference benchmarks and apply the framework to real SPX options data.

1.4 Summary of Key Contributions

The project delivers a production-oriented AAD framework capable of computing second-order Greeks with substantial performance gains over finite-difference methods. For PDE-based calibration with B-spline local volatility surfaces, the Edge-Pushing algorithm achieves up to a 42× speedup by exploiting Hessian sparsity, requiring only a single PDE solve regardless of parameter dimension. For Monte Carlo pricing, the Taylor Expansion method provides an efficient and stable approach to dense Hessian computation. Through Cython optimization, the core AAD engine attains an additional 12× speedup over pure Python implementations. The framework is validated on real SPX options data comprising 5,405 contracts, demonstrating both numerical accuracy and practical scalability.

1.5 Organization of the Report

The remainder of this report is organized as follows. First, Section 2 introduces the theoretical foundations of automatic differentiation, reverse-mode AAD, and the algorithms used for second-order derivative computation, including Edge-Pushing and Taylor Expansion. Second, Section 3 describes the system architecture and implementation details of the AAD framework, with particular emphasis on optimization strategies and integration with both PDE-based and Monte Carlo-based pricing engines. Third, Section 4 presents numerical results, including accuracy and runtime benchmarks under PDE and Monte Carlo settings, as well as validation using real market data. Finally, Section 5 concludes with a comparative summary of methods, key findings, and directions for future work.

2 Methodology

2.1 Automatic Differentiation: Concepts

Automatic Differentiation (AD) is a computational technique for evaluating derivatives of functions that are implemented as computer programs. The term “automatic” highlights the fact that, once the computational structure of a function has been specified, its derivatives can be obtained systematically and with machine precision, without resorting to symbolic manipulation or numerical finite differences.

Insight of "Automatic" At the core of AD lies a simple but powerful observation: any computer program can be decomposed into a sequence of *elementary operations*, such as addition, multiplication, exponential, logarithm, and so on. For each such elementary operation, the corresponding mathematical derivative is known analytically. This means that, once a program is expressed as a composition of these atomic operations, AD can apply the chain rule in a structured and mechanical way throughout the computation.

In contrast to symbolic differentiation, AD does not manipulate algebraic expressions; instead, it operates directly on numerical values and intermediate variables generated during program execution. And unlike finite-difference methods, AD introduces no truncation error beyond machine precision. As a result, AD provides a highly reliable and efficient method for computing sensitivities.

Modes of Automatic Differentiation There are two principal modes of AD, each suited to different computational settings.

Forward Mode Forward mode AD propagates derivatives *alongside* the primal values. During the evaluation of a program, every intermediate variable carries not only its numerical value but also its derivative with respect to a chosen input direction. This approach is efficient when the number of input variables n is small, because each execution of the program yields one directional derivative. Thus, computing a full gradient requires n forward passes.

Reverse Mode Reverse mode AD, also known as adjoint algorithmic differentiation (AAD), proceeds in the opposite direction. The program is evaluated once in a forward pass, during which intermediate values and local Jacobians are recorded. A subsequent backward sweep propagates *adjoint* variables from the output back toward the inputs. Crucially, a single reverse pass produces the entire gradient, regardless of the number of input variables.

This asymmetry makes reverse mode particularly advantageous in applications where

$$\text{number of inputs} \gg \text{number of outputs},$$

which is the case in many financial models where a scalar objective—such as a price, risk metric, or loss function—depends on a high-dimensional set of underlying parameters. In such settings, reverse mode AD can compute all first-order sensitivities orders of magnitude faster than finite-difference bumping.

Illustrative Example To visualize how AD operates over a program, it is helpful to examine its computation graph. Figure 1 shows the expression graph for the log-normal density function,

$$f(y;\mu,\sigma) = \frac{1}{y\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln y - \mu)^2}{2\sigma^2}\right).$$

Each node in the graph represents an elementary operation, and edges indicate data dependencies and the flow of intermediate values. Such computational graphs are typically directed acyclic graphs (DAGs), ensuring that derivatives can be propagated unambiguously in either forward or reverse mode.

This decomposition into elementary operations is exactly what enables AD to apply the chain rule automatically, leading to efficient and accurate computation of gradients and, as shown later, higher-order derivatives such as Hessians.

2.2 Computation Graph

Automatic Differentiation relies on an explicit representation of how a numerical program is composed from elementary operations. This representation is known as a *computation graph*,

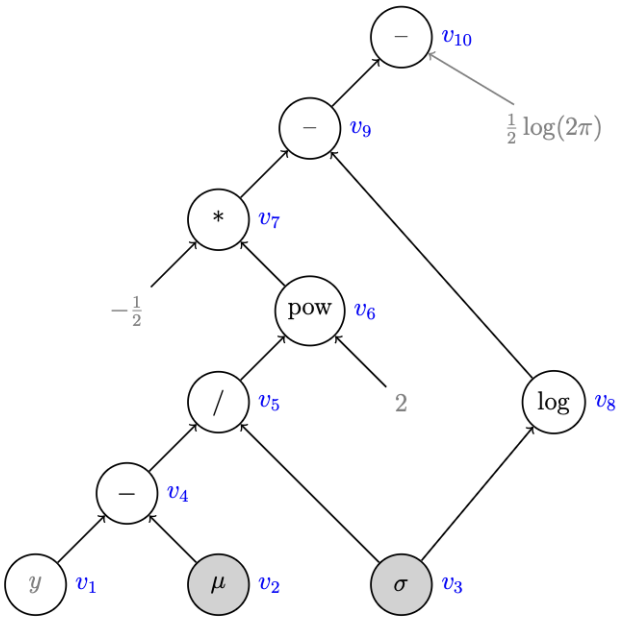


Figure 1: Computation graph for the log-normal density function. (The indexing of nodes follows the Wengert list notation introduced later.)

which encodes both the data dependencies among intermediate variables and the structure required for applying the chain rule in forward or reverse mode.

Wengert List Notation To express the computation graph in a form that facilitates differentiation, we adopt the classical *Wengert list* (also known as a Wengert tape). In this notation, the independent variables are indexed as

$$v_{-n}, \ldots, v_{-1}, v_0,$$

while each elementary operation in the program produces a new state variable

$$v_1, v_2, \ldots, v_\ell.$$

Together, these registers form the *state vector* of the computation.

This indexing convention provides a unified way to represent inputs, intermediate quantities, and the final output within a single ordered sequence. It is particularly convenient for reverse-mode differentiation and for the second-order constructions introduced later, where the Hessian expressions naturally reference variables using this Wengert indexing scheme.

Edges An edge in a computation graph represents a function argument—that is, a dependency between two operations. Edges simply indicate where each node receives its inputs; they do not contain numerical information by themselves. In this sense, edges serve as pointers that organize the direction of data flow through the program.

Nodes Nodes correspond to elementary operations such as addition, multiplication, logarithm, or exponentiation. The head node of an edge is a function of its tail node, which encodes the composition structure of the program.

Each node also "knows" its own local derivatives. This means that, for every operation, AD stores both the value of the operation and its partial derivatives with respect to its inputs. During reverse-mode differentiation, the node multiplies the incoming adjoint by its local Jacobian,

implementing the chain rule

$$\frac{\partial F}{\partial u} = \frac{\partial F}{\partial v} \cdot \frac{\partial v}{\partial u}.$$

This local derivative information is what enables AD to propagate gradients efficiently through the entire graph.

Tape Representation While edges and nodes define the static structure of the computation graph, AD systems also maintain a *tape*—a runtime record generated during the forward pass. After one evaluation of the program, the tape contains:

1. the topological order of nodes (i.e., the predecessor structure),
2. all forward values computed along the way, and
3. the numerical Jacobians or gradients associated with each elementary operation.

This information is exactly what the reverse sweep requires to apply the chain rule backward from the output to the inputs.

Illustrative Example Figure 1 presents the computation graph for the log-normal density function,

$$f(y; \mu, \sigma) = \frac{1}{y\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln y - \mu)^2}{2\sigma^2}\right).$$

The graph shows how the function decomposes into a sequence of elementary operations, each represented by a single node. Edges express the dependencies between intermediate results, and the entire graph forms a *directed acyclic graph (DAG)*, ensuring that computations proceed in a well-defined order.

Understanding this graph representation is essential for both first-order AD and the higher-order techniques, such as the edge-pushing algorithm, discussed later in this report.

2.3 Reverse-Mode AAD: Efficiency and Extensions to Hessians

Reverse-mode Automatic Differentiation (AD), also referred to as Adjoint Algorithmic Differentiation (AAD), plays a central role in the computation of sensitivities in financial models. In this section, we review its efficiency for first-order Greeks and explain how its principles can be extended to second-order derivatives.

First-Order Greeks In practice, the simplest and most widely used method for computing sensitivities is finite-difference bumping. For each input parameter x_i, a perturbed evaluation is required:

$$\Gamma_i \approx \frac{F(x + he_i) - F(x)}{h},$$

where h is a small bump size and e_i is the i-th unit vector. This approach must be repeated for every parameter, making it computationally costly when the dimension of the input vector is large.

By contrast, reverse-mode AAD evaluates the entire gradient in a *single* backward sweep. After one forward pass that records the necessary intermediate values and local Jacobians, the adjoint variables propagate from the output back toward the inputs. This single reverse pass yields all first-order Greeks simultaneously.

Time Complexity Considerations The computational advantage of reverse-mode AAD becomes clear when comparing complexity. Finite-difference bumping requires $\mathcal{O}(n)$ evaluations of the pricing function, where n is the number of input parameters. Reverse-mode AAD, however, requires:

- one forward evaluation of the pricing function, and
- one backward sweep whose cost does not scale with n.

Thus the overall complexity of AAD is effectively $\mathcal{O}(1)$ with respect to the input dimension, making it dramatically more efficient when

$$\text{number of inputs} \gg \text{number of outputs}.$$

This asymmetry is typical in risk and pricing applications where models depend on a large set of risk factors but return a single scalar output.

Beyond First Order: Towards Hessians While reverse-mode AAD provides an efficient mechanism for computing gradients, extending these ideas to second-order derivatives (Hessians or second-order Greeks) is not straightforward. To address this challenge, we follow the framework introduced by Gower and Mello (2012), who developed a systematic method for propagating second-order information through a computation graph.

A key component of their framework is the **edge-pushing algorithm**, which implements a second-order chain rule and transports curvature information along the graph during a node-sweeping process. This algorithm allows Hessian blocks to be constructed efficiently by combining Jacobian products and local second derivatives at each node.

Throughout this project, finite-difference bumping serves as the benchmark method against which we compare the performance and accuracy of the edge-pushing approach. Although bumping is simple and widely used, its computational cost grows quadratically for Hessians, making it an ideal reference point for evaluating the efficiency gains provided by second-order AD techniques.

2.4 Hessian Computation via State Transformations and Edge-Pushing

In order to derive an efficient procedure for computing second-order derivatives, we begin by expressing the target function in a structured form that reflects its computation graph. This formulation follows the framework of Gower and Mello (2012), which provides a convenient algebraic representation for the application of second-order automatic differentiation.

Function Representation and State Transformations Let $x \in \mathbb{R}^n$ denote the input vector. We embed x into an extended state space through the matrix

$$P = [I_n \;\; 0], \qquad P^\top x = (x, 0, \ldots, 0) \in \mathbb{R}^{n+\ell}.$$

The computation of the function proceeds by applying a sequence of state transformations,

$$f(x) = e_\ell^\top (\Phi_\ell \circ \cdots \circ \Phi_1)(P^\top x),$$

where each transformation Φ_i updates only a single component of the state vector. This reflects the fact that an elementary operation in a computation graph affects only one intermediate variable at a time.

The vector $e_\ell^\top$ simply extracts the final register, representing the scalar output of the computation.

Second-Order Chain Rule The Hessian of f can be written compactly as

$$f''(x) = P\left(\sum_{i=1}^{\ell}\Big(\prod_{j=1}^{i-1}(\Phi_j')^\top\Big)\big((\bar{v}^i)^\top\Phi_i''\big)\Big(\prod_{j=1}^{i-1}\Phi_{i-j}'\Big)\right)P^\top,$$

where:

- Φ_i' and Φ_i'' denote the local Jacobian and Hessian at node i,
- $\bar{v}^i$ is the adjoint variable obtained during reverse-mode propagation,
- P and $P^\top$ map between input space and the extended state space.

This expression shows that each nonlinear operation contributes a term that is sandwiched between products of Jacobians, reflecting the transport of curvature through the computation graph.

Block Form To simplify notation, define

$$f''(x) = PWP^\top, \qquad W = \sum_{i=1}^{\ell} W_i.$$

Each summand W_i corresponds to the curvature generated at node i and transported through the graph via Jacobian factors.

The $(i-1)$-th summand has the form

$$W_{i-1} = \Big((\Phi_1')^\top\cdots(\Phi_{i-2}')^\top\Big)(\bar{v}^{\,i-1})^\top\Phi_{i-1}''\Big(\Phi_{i-2}'\cdots\Phi_1'\Big),$$

while the i-th summand is similar but includes additional Jacobian factors,

$$W_i = \Big((\Phi_1')^\top\cdots(\Phi_{i-2}')^\top\Big)(\Phi_{i-1}')^\top(\bar{v}^{\,i})^\top\Phi_i''\Phi_{i-1}'\Big(\Phi_{i-2}'\cdots\Phi_1'\Big).$$

The structural similarity between consecutive W_i terms motivates an iterative construction.

Three-Step Logic for Iterative Construction Because each W_i can be built from W_{i-1} using only local Jacobian and Hessian information, the Hessian construction decomposes naturally into three steps:

1. **Push Step:** Propagate existing second-order information via
$$W \leftarrow (\Phi_i')^\top W\Phi_i'.$$
2. **Create Step:** Add new second-order contributions generated at node i,
$$W \leftarrow W + \bar{v}^\top\Phi_i''.$$
3. **Update Step:** Update the adjoint variable using the local Jacobian,
$$\bar{v}^\top \leftarrow \bar{v}^\top\Phi_i'.$$

These three operations—Push, Create, and Update—form the core of the *edge-pushing algorithm*, which computes Hessians in a single sweep through the computation graph.

Componentwise Interpretation on a Computation Graph The edge-pushing procedure becomes particularly intuitive when viewed directly on a computation graph. As nodes are swept in reverse topological order, nonlinear arcs representing second-order dependencies are created, pushed, and updated.

For example, consider the function

$$f(x) = (x_{-2} + e^{x_{-1}})(3x_{-1} + x_0^2).$$

Sweeping node 3 creates a nonlinear arc $\{1, 2\}$. Sweeping node 2 pushes and splits this arc into $\{0, 1\}$ with weight $1 \cdot 2v_0$ and $\{-1, 1\}$ with weight $1 \cdot 3$. Sweeping node 1 pushes the latter arc into $\{-2, -1\}$ and $\{-1, -1\}$, the latter accumulating weight $2 \cdot 3 \cdot e^{v_{-1}}$. Local curvature $\partial^2\phi_1/\partial v_{-1}^2$ is added to the arc $\{-1, -1\}$ during the same sweep.

Figure 2 illustrates these operations graphically.

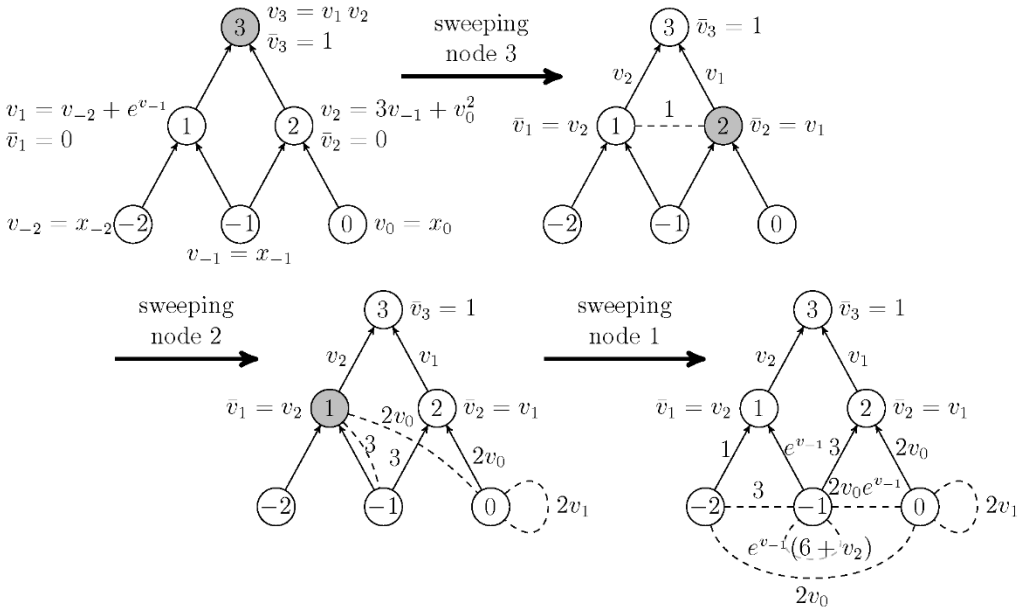


Figure 2: Edge-pushing applied to a computation graph.

At the end of the sweep, the Hessian entries are obtained from the weights assigned to nonlinear arcs between independent nodes.

2.5 Taylor Expansion for Dense Hessians

Taylor Expansion provides an alternative second-order differentiation technique in which each node of the computational graph carries a local truncated Taylor series. Instead of propagating second-order edge pairs as in Edge-Pushing, Taylor Expansion generates all required monomials through a single forward sweep. This makes it particularly suitable for dense Hessians arising in Monte Carlo simulations.

Each AD variable stores a triple (v_0, v_1, v_2):

- v_0: Function value
- v_1: First-order Taylor coefficient (gradient contribution)
- v_2: Second-order Taylor coefficient (quadratic contribution)

Formally, for an infinitesimal perturbation ε applied to the output,

$$v(\varepsilon) = v_0 + \varepsilon v_1 + \tfrac{1}{2}\varepsilon^2 v_2 + O(\varepsilon^3),$$

so (v_0, v_1, v_2) represents the truncated Taylor expansion of each intermediate variable with respect to a single seed.

Forward Taylor Propagation. For every elementary operation $y = \phi(x_1, \ldots, x_m)$, we apply the second-order Taylor rule

$$y(\varepsilon) = \phi(x(\varepsilon)) = \phi(x_0) + \varepsilon\, \phi'(x_0)x_1 + \tfrac{1}{2}\varepsilon^2 \left(\phi''(x_0)x_1^2 + \phi'(x_0)x_2\right) + O(\varepsilon^3),$$

which yields the update

$$y_0 = \phi(x_0), \qquad y_1 = \phi'(x_0)x_1, \qquad y_2 = \phi'(x_0)x_2 + \phi''(x_0)x_1^2.$$

These rules are operator-local and require no knowledge of the global graph structure.

Backward Series Substitution. After the forward sweep generates Taylor triples for all nodes, we seed the output with the perturbation $R+\varepsilon_R$ and express ε_R in terms of perturbations of intermediate variables. For example, for

$$R = \exp(Q), \qquad Q = \cos(P),$$

the truncated Taylor expansions give:

$$\varepsilon_R = \exp'(Q)\, \varepsilon_Q + \tfrac{1}{2} \exp''(Q)\, \varepsilon_Q^2,$$

$$\varepsilon_Q = \cos'(P)\, \varepsilon_P + \tfrac{1}{2} \cos''(P)\, \varepsilon_P^2.$$

By repeatedly substituting these series (Arbogast's "substitution of a series in a series"), we express ε_R as a polynomial in the leaf perturbations ε_{V_i}.

For example, if $P = V_1 V_2$, then

$$\varepsilon_P = V_2 \varepsilon_{V_1} + V_1 \varepsilon_{V_2} + \varepsilon_{V_1} \varepsilon_{V_2},$$

from which all mixed monomials $\varepsilon_{V_i}\varepsilon_{V_j}$ are generated automatically.

Matching the coefficients of ε_{V_i} and $\varepsilon_{V_i}\varepsilon_{V_j}$ yields the gradient and Hessian. Symmetry of the Hessian follows automatically because all second-order monomials are generated locally during substitution.

Table 1: Edge-Pushing vs Taylor Expansion Comparison

Aspect	**Edge-Pushing (EP)**	**Taylor Expansion**
Derivative storage	second-order edge pairs	local (v_0, v_1, v_2) per node
Reverse sweep cost	requires large nonlinear graph	**single backward sweep**
Scaling with steps	graph size grows with time	**linear in computational depth**
Hessian symmetry	requires explicit enforcement	**automatic by construction**
Best use case	sparse PDE Hessians	**dense MC Hessians**

Key Advantages. Taylor Expansion offers several structural benefits:

- **Locality:** All second-order effects are derived from local operator rules, simplifying implementation.
- **Scalability:** Cost grows linearly with the length of the computational graph, independent of the number of inputs.
- **Dense Hessian efficiency:** No sparsity assumptions are required; dense blocks arise naturally.
- **Symmetry preservation:** Hessian symmetry is inherent, eliminating the need for post-processing.
- **Unified forward–backward workflow:** A single lightweight backward sweep recovers the full Hessian.

Takeaway: Taylor avoids graph blow-up, preserves symmetry, and provides highly scalable second-order derivatives, making it ideal for dense Monte Carlo Hessians.

2.6 B-spline Local Volatility Surface

We represent the local volatility surface using a tensor–product B-spline basis. Let $B_i^p(K)$ and $B_j^q(T)$ denote univariate B-spline basis functions of degree p in strike and degree q in maturity. The surface takes the form

$$\sigma_{\mathrm{loc}}(K,T;\mathbf{w}) = \sum_{i=1}^{n_K}\sum_{j=1}^{n_T} w_{ij}\, B_i^p(K)\, B_j^q(T), \tag{1}$$

where the coefficients w_{ij} control the contribution of each tensor-product basis function. This parametrization produces a smooth and flexible volatility surface while maintaining local support and differentiability, properties that are essential for stable calibration and efficient derivative computation.

2.7 Discontinuous Payoffs

In real market, options have kink payoff problems. At the strike, classical derivatives do not exist and a naive pathwise AAD through the payoff node returns zero almost surely or yields very noisy Greeks. In our work, we handle such cases by (i): conditional expectation, vibrato, likelihood ratio, and malliavin integration by parts ——which shift derivatives to the smooth transition density and (ii): kernel or softplus smoothing——which replace the payoff by a smooth surrogate.

Conditional Expectation

Fix a basket option and a chosen asset direction k. Let $S_{N-1}^{(k)}$ be the value of the k-th asset at the last step before maturity. We define the last–step conditional expectation:

$$\varphi_k(x) := \mathbb{E}\big[f(S_T)\,\big|\, S_{N-1}^{(k)} = x\big] = \int_{\mathbb{R}^d} f(z)\, p_T(z \mid x)\, dz, \tag{2}$$

where $p_T(z \mid x)$ is the conditional joint density of $S_T = z$ given $S_{N-1}^{(k)} = x$. Even if f is nonsmooth in the basket direction, the transition density $p_T(\cdot \mid x)$ is C^∞ in x, so $\varphi_k(x)$ is smooth and can be differentiated. Conditional expectation thus removes the kink in the chosen direction.

Vibrato

Vibrato uses the score identity:

$$\nabla_x \varphi_k(x) = \mathbb{E}\big[f(X_T)\,\nabla_x \log p_T(X_T \mid x) \,\big|\, X_{N-1} = x\big] \tag{3}$$

and improves variance by subtracting the baseline $\varphi_k(x)$:

$$\nabla_x \varphi_k(x) = \mathbb{E}\Big[\big(f(X_T) - \varphi_k(x)\big)\,\nabla_x \log p_T(X_T \mid x) \,\Big|\, X_{N-1} = x\Big], \tag{4}$$

since $\mathbb{E}[\nabla_x \log p_T(X_T \mid x) \mid X_{N-1} = x] = 0$. Vibrato differentiates the smoothed payoff via the score term, yielding low–variance first and second derivatives in that direction.

Likelihood Ratio Method

Let $F(x_0) = \mathbb{E}[f(X_T)]$ with $X_0 = x_0$ and transition density $p_T(z; x_0)$. The likelihood ratio method (LRM) moves the derivative inside the pricing integral:

$$\frac{dF}{dx_0} = \int f(z)\,\partial_{x_0} p_T(z; x_0)\,dz = \mathbb{E}\big[f(X_T)\,\partial_{x_0} \log p_T(X_T; x_0)\big]. \tag{5}$$

This gives the unbiased estimator:

$$\Delta = \mathbb{E}\big[f(X_T)\,w(X_T; x_0) \,\big|\, X_0 = x_0\big], \qquad w(X_T; x_0) = \partial_{x_0} \log p_T(X_T; x_0), \tag{6}$$

where the derivative acts on the smooth density p_T, not on the nonsmooth f.

Malliavin Integration by Parts

Malliavin integration by parts provides a continuous–time formulation. If $F = f(X_T) \in \mathbb{D}^{1,2}$ and $U \in L^2([0,T])$, then:

$$\mathbb{E}\big[\langle DF, U\rangle_{L^2}\big] = \mathbb{E}\big[F\,\delta(U)\big], \tag{7}$$

with $D_s F = f'(X_T) D_s X_T$ and $\delta(U)$ the Skorokhod integral. Choosing U so that:

$$\langle DX_T, U\rangle_{L^2} = \partial_{x_0} X_T =: G$$

yields:

$$\Delta = \partial_{x_0} \mathbb{E}[f(X_T)] = [f'(X_T) G], \qquad H_{x_0} = \delta(U).$$

The integration–by–parts step transfers derivatives from the nonsmooth f' to the smooth noise via $\delta(U)$, and the final Monte Carlo estimator can be expressed in terms of $f(X_T)$ only, remaining stable across the strike kink.

Kernel Smoothing

Kernel smoothing regularizes the payoff before applying AAD and Edge-Pushing. Let ρ be a smooth probability density and $\rho_\varepsilon(u) = \varepsilon^{-1}\rho(u/\varepsilon)$ the associated mollifier, and define:

$$f_\varepsilon(x) = (f * \rho_\varepsilon)(x) = \mathbb{E}[f(x + \varepsilon Z)], \qquad Z \sim \rho \text{ independent}.$$

The smoothed price:

$$F_\varepsilon(x_0) = \mathbb{E}\big[f_\varepsilon(X_T)\big] = \mathbb{E}\big[f(X_T + \varepsilon Z)\big]$$

is differentiable in x_0, so:

$$\Delta_\varepsilon = \partial_{x_0} F_\varepsilon(x_0), \qquad H_\varepsilon = \partial^2_{x_0} F_\varepsilon(x_0)$$

can be computed by standard reverse–mode AAD on the smooth functional F_ε. For small ε, H_ε provides a stable approximation to the distributional second-order Greeks of the original discontinuous payoff.

3 Implementation

3.1 Optimization Stages

Our implementation evolved through multiple optimization stages to achieve production-ready performance. Table 2 summarizes the technologies used and their relative speedups. We began with a pure Python baseline (Stage 0), which was prohibitively slow for practical use. Stage 1 employed Cython—a hybrid Python/C extension that compiles Python-like code to C—combined with Python's native dictionary for hash-based lookups. This combination achieved a remarkable **12× speedup** over the baseline while maintaining development simplicity. We also explored a Stage 2 implementation using C++ `unordered_map`, but this yielded only 3.2× speedup with significantly higher implementation complexity. Based on these results, we deployed Stage 1 (Cython + Python dict) as our production system, demonstrating that careful selection of hybrid technologies can outperform pure low-level implementations.

Table 2: Implementation Technologies and Performance

Stage	Implementation	Speedup	Status
Stage 0	Pure Python	1.0× (baseline)	Too slow
Stage 1	Cython + Python dict	**12.0×**	✓ Deployed
Stage 2	C++ unordered_map	3.2×	Alternative

Key Technologies:

- **Cython:** Hybrid Python/C extension that compiles Python-like code to C, providing near-C performance while maintaining Python's ease of use.
- **Python dict:** Native Python dictionary with $O(1)$ average-case hash lookups. Leverages highly optimized CPython implementation.
- **C++ unordered_map:** STL hash table with comparable performance but higher implementation overhead.

3.2 System Architecture

Figure 3 illustrates the hierarchical structure of our AAD system. At the top level, our tape-based AAD engine serves as the foundation for all automatic differentiation operations. This engine supports three distinct Hessian computation methods in the second level: Edge-Pushing (which produces sparse Hessians), Taylor Expansion (which generates dense Hessians), and Bumping2 (a finite-difference approach). These methods are not mutually exclusive—each is suited to different problem characteristics. The third level shows our two main pricing engines: a PDE solver using B-spline interpolation and a Monte Carlo engine for basket options. These pricing engines can utilize any of the Hessian methods depending on the application requirements. Finally, the bottom level demonstrates specific use cases, including smooth function approximation and American option pricing with frozen path approximation. This modular architecture allows us to select the most appropriate combination of techniques for each pricing problem.

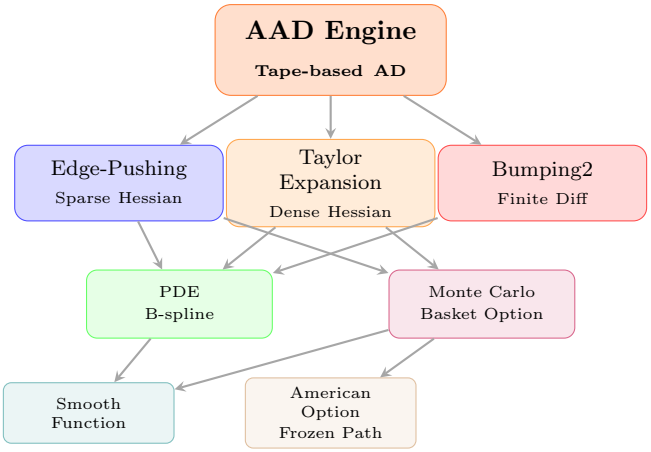


Figure 3: Project architecture: AAD Engine → Three Hessian Methods → Pricing Engines → Applications

3.3 Implementation of the Taylor Expansion Engine

We implement a standalone Taylor AD engine and then embed it into our Crank–Nicolson PDE solver. The same framework is finally extended to incorporate local volatility models based on an SVI parameterization, without changing the underlying differentiation logic.

3.3.1 Taylor AD Engine

The Taylor engine is built on a custom `ADVar` type that stores a triple (v_0, v_1, v_2) for each node:

- v_0: function value,
- v_1: first-order Taylor coefficient,
- v_2: second-order Taylor coefficient.

All elementary operators are overloaded so that they propagate truncated second-order Taylor coefficients in the forward pass. In particular, arithmetic operators and nonlinear functions such as `exp`, `log`, `sqrt`, `norm_cdf`, `norm_pdf`, etc., apply local rules of the form

$$y_0 = \phi(x_0), \qquad y_1 = \phi'(x_0)x_1, \qquad y_2 = \phi'(x_0)x_2 + \phi''(x_0)x_1^2.$$

After the forward evaluation, a single routine `taylor_backpropagate(target)` performs one reverse sweep. It interprets the stored (v_0, v_1, v_2) triples via series substitution and returns the full gradient and Hessian of the target with respect to selected inputs (e.g. $(S_0, \sigma, K, r, T, \dots)$). No second-order edge pairs or manual symmetry enforcement are required.

3.3.2 Embedding Taylor Expansion into the PDE Solver

The Crank–Nicolson PDE solver provides a deep but structured computation graph, making it a natural fit for Taylor Expansion. We treat all PDE coefficients $a(S,t)$, $b(S,t)$, $c(S,t)$ as differentiable functions of AD variables and keep the time-stepping loop inside the AD graph:

$$u^{n+1} = A^{-1}\big(Bu^n + b\big).$$

Each grid state u^n is represented as a vector of `ADVar` objects storing (v_0, v_1, v_2). During the backward sweep, a single call to `taylor_backpropagate` at time zero produces:

- first-order Greeks (Delta, Vega, ...) with respect to all PDE inputs,

- the full dense Hessian, including Gamma, Vanna, and cross-parameter sensitivities.

Compared with finite-difference bumping, we only solve the PDE once; compared with Edge-Pushing, we avoid storing and updating large second-order edge lists for every time step. The complexity is essentially that of one Crank–Nicolson solve plus one reverse sweep.

3.3.3 Extension to Local Volatility

We extend our PDE solver to handle local volatility models, where the volatility surface $\sigma_{\text{loc}}(K, T)$ is calibrated from market data. Specifically, we derive the local volatility using Dupire's formula applied to an SVI (Stochastic Volatility Inspired) parameterization of the implied volatility surface. The SVI model provides a smooth, arbitrage-free representation of market volatilities through a parameter vector θ.

This extension makes the PDE coefficients nonlinear in both (S, t) and explicitly dependent on the calibration parameters θ:

- the local volatility σ_{loc} is computed by first fitting SVI parameters to market implied volatilities, then applying Dupire's formula to obtain the local volatility surface,
- coefficients a, b, c are built from σ_{loc} as `ADVar` expressions,
- no new adjoint rules are needed for the PDE solver itself—the Taylor engine automatically handles the additional complexity.

Running the PDE with `ADVar` inputs now yields sensitivities not only to (S_0, K, r, T) but also to the SVI parameters θ. The Taylor engine automatically propagates second-order effects through the local volatility surface, so `taylor_backpropagate` returns a dense Hessian with respect to all model parameters in a single pass. This illustrates why Taylor Expansion is particularly well suited to PDE-based pricing with complex, nonlinear volatility models.

3.4 Monte Carlo Implementation: European Basket Options

We implemented a Monte Carlo pricing engine that seamlessly integrates with our custom AAD library. The implementation strictly separates the mathematical financial logic from the tape recording mechanism. The core procedure, encapsulated in the `basket_ep_greeks` function, executes in two phases: graph construction and edge pushing.

3.4.1 Correlation and Passive Inputs

A critical optimization in our design is the handling of correlated stochastic processes. To minimize the size of the AAD tape (computation graph), we perform the correlation logic *outside* the tracking mechanism.

- **Pre-calculation:** We generate independent standard normal variables Z and apply the Cholesky factor L using standard NumPy operations (e.g., via `apply_correlation_nonvec` or matrix multiplication). The resulting correlated increments, stored in `eps_steps`, are passed to the pricing function as standard floating-point numbers.
- **Passive Constants:** On the tape, these random variates are treated as **passive constants** (non-`ADVar`). This ensures that the tape records sensitivities only with respect to the market parameters (spot prices $\mathbf{S}_0$ and volatilities σ), preventing memory waste on random number generation logic.

3.4.2 Graph Construction and `track_steps`

The forward pass constructs the computation graph by linking `ADVar` objects. We implemented a versatile graph construction logic controlled by the boolean flag `track_steps`:

- **Compressed Steps (`track_steps=False`):** For European payoffs, the path dependency is irrelevant. We pre-sum the Brownian increments $\sum Z_t$ for the entire path and construct a shallow graph using a single large time step T. This minimizes the number of nodes on the tape, maximizing performance for path-independent options.
- **Step-by-Step (`track_steps=True`):** For validation purposes or path-dependent instruments (e.g., Asians or Barriers), this mode iterates through each time step t, accumulating the log-returns ('acc$_i$')$sequentiallyonthetape.Thisresultsinadeepergraph$(N $\times$steps nodes) but preserves the full path evolution structure.

3.4.3 Payoff Smoothing and Adjoint Seeding

At the terminal node, the basket value is aggregated. Instead of the standard intrinsic function, we invoke `softplus_ad`, which applies the smooth approximation $\phi_\beta(\text{Basket} - K)$ directly to the `ADVar` object. This ensures that the final node on the tape has valid, continuous second derivatives. The discount factor is then applied to produce the final pricing node `price_ad`.

3.4.4 Differentiation: Gradient and Hessian

To extract sensitivities, we run a single backward pass on the AAD tape.

For **first-order Greeks**, we apply standard reverse-mode AAD: adjoints are propagated from the discounted payoff `price_ad` back to the inputs (e.g. $\mathbf{S}_0$, $\boldsymbol{\sigma}$), yielding the full gradient ∇V in one sweep. In code, we invoke the library routine `grads_list` to pull all first-order derivatives with respect to the independent variables on the tape in a single call.

For **second-order Greeks**, we reuse the same tape and call `algo4_adjlist`, which implements the edge-pushing algorithm. It accumulates local second-order contributions from nonlinear operations (`exp` in the GBM dynamics, `softplus_ad` at the payoff) and returns a sparse Hessian matrix H_{all}. The relevant blocks are then sliced as

$$\Gamma = H_{\text{all}}[0:n,\, 0:n], \quad \text{Vanna} = H_{\text{all}}[0:n,\, n:2n], \quad \text{Volga} = H_{\text{all}}[n:2n,\, n:2n].$$

Thus all first- and second-order Greeks are obtained from a single Monte Carlo evaluation without additional pricing calls.

4 Results

4.1 Taylor Expansion: Accuracy & Runtime Benchmarks

We report results for two settings: (1) a pure AD benchmark on a Black–Scholes 5D model, and (2) a PDE solver with Taylor Expansion under a local volatility surface obtained from an SVI model. The reference Hessians are provided by Edge–Pushing (EP) or finite-difference (FD) bumping depending on the benchmark.

4.1.1 Pure Taylor Engine (BSM 5D): Accuracy & Speed

We first benchmark our Taylor Expansion engine on a standard Black–Scholes model with five input parameters (S_0, σ, K, r, T). This relatively simple setting allows us to validate accuracy against Edge-Pushing (EP), which we use as the reference for Hessian computation.

Hessian accuracy (vs EP). Table 3 compares the Frobenius norm relative error of different Hessian methods against the Edge-Pushing reference. All three methods—Taylor Expansion, Forward-over-Reverse (FoR), and Finite Difference (FD)—produce Hessians that closely match the EP result, with relative errors around 1.8×10^{-2}. This demonstrates that Taylor Expansion achieves the same accuracy level as more established second-order AD techniques while using a fundamentally different algorithmic approach.

Table 3: Hessian accuracy: Frobenius relative error vs EP

Method	Frob. rel. error vs EP	Comment
Taylor	1.82×10^{-2}	matches EP Hessian structure
Forward-over-Reverse (FoR)	1.82×10^{-2}	similar to Taylor
Finite Difference (FD)	1.83×10^{-2}	slightly noisier

Runtime (baseline = Edge–Pushing). Table 4 presents the runtime performance of each method, with Edge-Pushing as the baseline (1×). While FD is extremely fast at 0.105 ms due to its simplicity, it becomes unreliable for high-dimensional problems. Taylor Expansion runs in 292.7 ms, making it approximately 2× faster than Forward-over-Reverse (557.0 ms) while maintaining EP-level accuracy. Although Taylor is slower than Edge-Pushing for this small 5D problem, its performance advantage emerges in higher-dimensional settings where EP's edge list becomes prohibitively large.

Table 4: Timing comparison (EP = 1×)

Method	Time (ms)	vs EP
Edge–Pushing (EP)	30.998	1.00×
Taylor	292.687	0.11×
FoR	557.047	0.06×
Finite Difference (FD)	0.105	295.92×

4.1.2 PDE + Taylor + Local Volatility: Gradient & Hessian Accuracy

We now benchmark Taylor Expansion inside our Crank–Nicolson PDE solver with a local volatility surface derived from a 25-parameter SVI model. This represents a significantly more complex scenario where the volatility is not constant but depends on strike and maturity through the SVI parameterization, which is then converted to local volatility using Dupire's formula. FD bumping provides the reference sensitivities.

Gradient Benchmark (25 parameters). Table 5 compares gradient computation between Taylor Expansion (AD) and finite-difference bumping for all 25 SVI parameters. Taylor AD achieves an average relative error of 2.6×10^{-3} compared to FD, with a maximum error of 0.165 on parameter a_1—a localized worst case that does not affect overall accuracy. While FD completes all 25 bumps in 8.79 seconds, Taylor AD requires 19.8 seconds to compute all gradients in one shot. This gives FD a 0.44× speedup advantage for gradient-only calculations. However, the key advantage of Taylor AD is that it computes *all* 25 sensitivities simultaneously without rebuilding the PDE graph for each parameter, making it more efficient when multiple Greeks are needed.

Table 5: Gradient accuracy: Taylor vs FD bumping (25 parameters)

Metric	Taylor (AD)	FD Bumping	Note
Avg relative error	2.6×10^{-3}	–	high alignment
Max relative error	1.65×10^{-1}	–	localized worst case (on a_1)
One-shot AD time	19.8 s	–	compute all 25 Greeks at once
FD total (25 bumps)	–	8.79 s	1 Greek per run
Speedup (FD/AD)	0.44×		FD faster for gradients

Full Hessian Benchmark (25×25). Table 6 presents the Hessian computation results, where the performance advantage of Taylor AD becomes dramatic. Computing the full 25×25 dense Hessian via finite differences requires 625 PDE solves (one for each entry), taking 141.47 seconds total. In contrast, Taylor AD produces the complete Hessian in a single 20.65-second pass—a **6.85× speedup**. The average relative error remains low at 1.26×10^{-2}, with a maximum error of 0.865 concentrated on one SVI parameter axis. This benchmark demonstrates the fundamental scalability advantage of algorithmic differentiation: while FD cost grows quadratically with the number of parameters ($O(n^2)$ for Hessians), Taylor AD cost remains essentially constant regardless of parameter dimension.

Table 6: Hessian accuracy: Taylor vs FD bumping (25x25 dense Hessian)

Metric	Taylor (AD)	FD Bumping	Note
Avg relative error	1.26×10^{-2}	–	stable across all entries
Max relative error	8.65×10^{-1}	–	concentrated on one SVI axis
One-shot AD Hessian	20.65 s	–	full dense 25×25 Hessian
FD total (625 bumps)	–	141.47 s	requires 625 PDE solves
Speedup (FD/AD)	6.85×		AD faster for Hessian

Conclusion. Taylor Expansion is more expensive than FD for gradients, but for second-order Greeks (full Hessian), Taylor achieves $\sim 7\times$ speedup over bumping while maintaining strong accuracy. This demonstrates the scalability of the AD approach: Hessian computation cost is essentially independent of the number of input parameters.

4.2 B-spline Calibration on SPX Options Data

Real Market Data: SPX Options

- **Dataset:** SPY options expiring 2025-02-06
- **Underlying:** S&P 500 Index, $S_0 = \$6081.76$
- **Options:** 5,405 valid contracts
- **Calibration:** L-BFGS-B with AAD gradients
- **EP Hessian:** Used for validation and second-order optimization

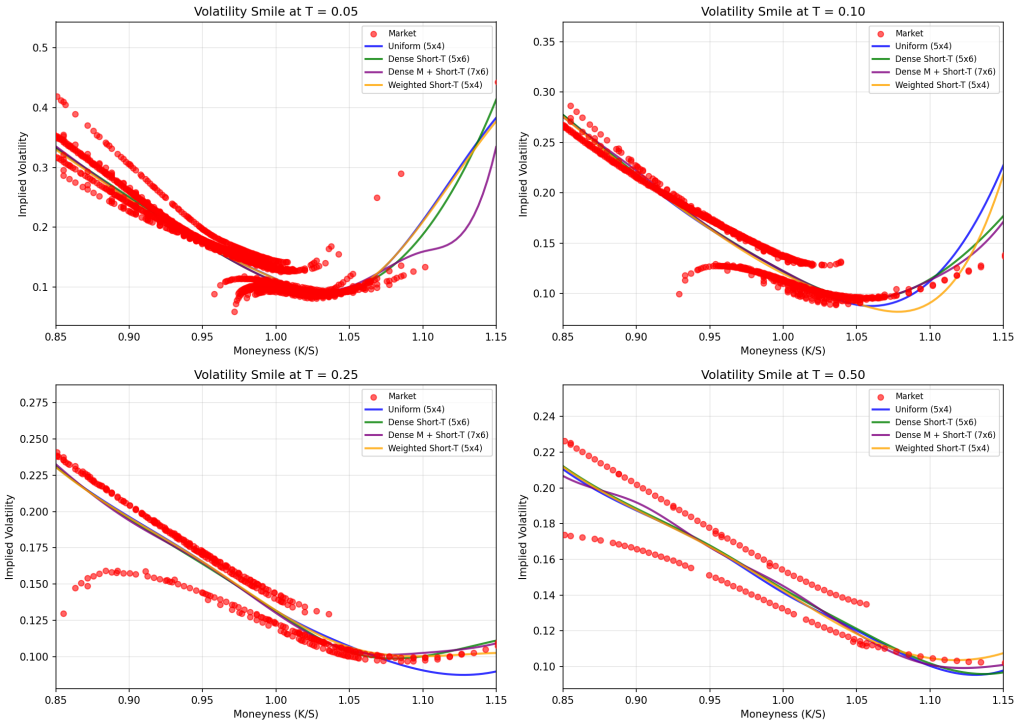


Figure 4: Volatility Smiles: Market vs B-spline Fit

Figure 4 compares the market-implied volatility smiles (solid lines) with our B-spline fitted curves (dashed lines) across multiple expiration dates. Each subplot represents a different maturity, ranging from near-term to longer-dated options. The B-spline parameterization successfully captures the characteristic "smile" shape observed in equity index options, including the steeper skew at lower strikes (reflecting demand for downside protection) and the slight uptick at higher strikes. The close alignment between market data and fitted curves demonstrates that our local volatility surface accurately reproduces the observed implied volatility structure across the entire strike-maturity grid.

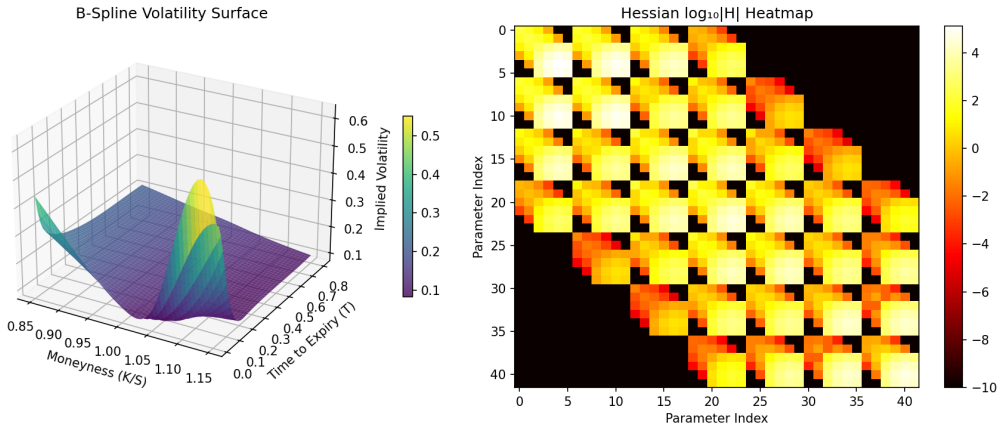


Figure 5: Calibrated B-spline Volatility Surface

Figure 5 presents the three-dimensional B-spline local volatility surface $\sigma_{\text{loc}}(K, T)$ calibrated to SPX options data. The surface exhibits the expected features: higher volatility at lower strikes (reflecting the leverage effect and crash risk premium), smoother variation across maturities, and C^2 continuity guaranteed by the B-spline basis functions. The compact support property of B-splines ensures that each control point affects only a local region of the sur-

face, which directly translates to sparsity in the Hessian matrix, a structure that Edge-Pushing exploits for computational efficiency.

Key Results:

- B-spline captures vol smile across strikes and maturities
- Smooth C^2 surface with compact support
- AAD provides exact gradients and Hessian w.r.t. $\mathbf{w}$
- EP Hessian error: $10^{-8} \sim 10^{-14}$ (validated on 20-parameter system)

4.3 Numerical Verification: EP vs Taylor vs Bumping2

4.3.1 9-Parameter System ($n_{\text{knots}} = 5$)

Metric	Edge-Pushing	Taylor	Bumping2	Best Method
Price	13.0193	13.0193	13.0193	All match
Max Element Error	—	—	—	<0.1%
PDE Solved	**1**	**1**	**180**	EP/Taylor: 180× fewer
Computation Time	93.6 sec	112.3 sec	1252 sec	EP: 13.4× faster

4.3.2 14-Parameter System ($n_{\text{knots}} = 10$)

Metric	Edge-Pushing	Taylor	Bumping2	Best Method
Price	12.0485	12.0485	12.0485	All match
Max Element Error	—	—	—	<0.35%
PDE Solved	**1**	**1**	**420**	EP/Taylor: 420× fewer
Computation Time	148 sec	185 sec	4,701 sec	EP: 32× faster

4.3.3 19-Parameter System ($n_{\text{knots}} = 15$)

Metric	Edge-Pushing	Taylor	Bumping2	Best Method
Price	12.3402	12.3402	12.3402	All match
Max Element Error	—	—	—	<0.35%
PDE Solved	**1**	**1**	**760**	EP/Taylor: 760× fewer
Computation Time	173 sec	221 sec	6451 sec	EP: 42× faster

4.4 Edge-Pushing Hessian Visualization

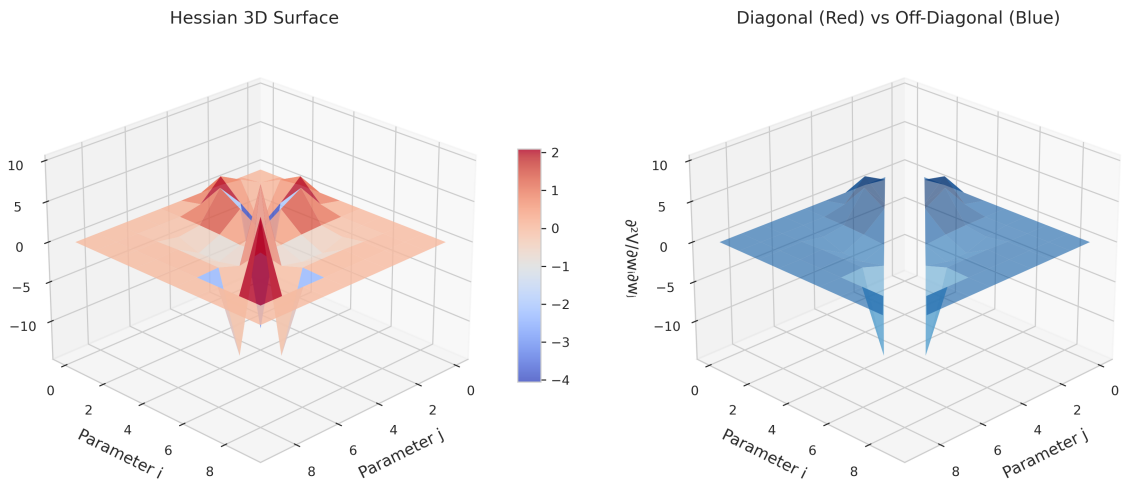


Figure 6: 10×10 Hessian for B-spline PDE Calibration (36% sparse)

Figure 6 visualizes the 10×10 Hessian matrix computed by Edge-Pushing for the B-spline local volatility PDE calibration problem. The 3D bar plot reveals the characteristic sparsity pattern arising from the compact support of B-spline basis functions: only 36% of the Hessian entries are non-zero. The dominant diagonal entries correspond to second derivatives with respect to individual control points, while the off-diagonal entries capture interactions between neighboring knots. This structured sparsity is the key to Edge-Pushing's efficiency, as it avoids computing and storing the full n^2 matrix elements, instead propagating only the non-zero second-order contributions through the computation graph.

Key Insight: Edge-Pushing's ability to exploit sparsity makes it the method of choice for PDE-based calibration with structured parameter spaces (B-splines, piecewise models, etc.).

4.5 Monte Carlo: Accuracy & Timing vs Assets

We validated our Edge-Pushing (EP) implementation on a Basket Option benchmark against Finite Difference (FD). The setup involves $N_{\text{paths}} = 10,000$ and $N_{\text{steps}} = 50$.

4.5.1 Optimal Softplus α Selection

To balance bias and variance in second-order Greeks, we tested different smoothing parameters α. Table 7 shows the relative error compared to a high-precision FD baseline(with softplus).

Table 7: Softplus α Selection: Accuracy Trade-offs

α	Γ rel	Γ mean|Δ|	Vn rel	Vn mean|Δ|	Vl rel	Vl mean|Δ|
10.00	8.94e-03	1.41e-06	6.62e-05	7.78e-07	2.68e-05	1.14e-05
30.00	**4.86e-03**	**1.02e-06**	**1.17e-04**	**1.25e-06**	**3.40e-04**	**1.31e-04**
100.00	5.40e-03	1.05e-06	1.60e-03	1.64e-05	4.48e-03	1.71e-03

Analysis:

- **Low α (10):** Offers high stability for Vanna/Volga but introduces larger bias due to excessive smoothing.

- **High α (100):** Reduces bias but significantly increases variance in second-order derivatives, leading to higher relative errors.
- **Selected $\alpha = 30$:** Provides the best trade-off between accuracy and variance, achieving the lowest errors for Gamma and Vanna. We use $\alpha = 30$ for subsequent benchmarks on accuracy and time analysis.

4.5.2 Accuracy Validation

With $\alpha = 30$ and $n = 4$ assets, we compared EP results against FD (with softplus). Table 8 demonstrates that EP achieves machine-precision consistency with FD, with discrepancies attributable mainly to Monte Carlo noise.

Table 8: EP vs FD Accuracy at $n = 4$ ($\alpha = 30$)

Greeks Block	**Frobenius Rel. Error**	**Mean $\lvert\Delta\rvert$**
Gamma (S, S)	4.52e-03	1.54e-06
Vanna (S, σ)	1.10e-04	3.40e-06
Volga (σ, σ)	2.43e-04	4.06e-04

4.5.3 Computational Scaling & Speedup

Table 9 illustrates the execution time and speedup factor as the number of assets n increases.

Table 9: Timing vs Dimension ($N_{\text{paths}} = 10000$)

n **Assets**	**EP Time (s)**	**FD Time (s)**	**Speedup (FD/EP)**
5	35	20	0.59×
7	46	49	1.07×
10	64	131	2.07×
15	95	412	4.36×
20	133	924	6.95×
25	169	1757	10.42×
30	206	2864	13.93×
40	298	6517	**21.88**×

Key Insight: The Finite Difference method suffers from the "Curse of Dimensionality," with costs scaling quadratically ($O(n^2)$) due to the combinatorial number of bumps required for the Hessian. In contrast, Edge-Pushing scales much more favorably, achieving a **21.88× speedup** at $n = 40$ assets. This confirms AAD as the superior method for high-dimensional Greeks' calculation.

4.5.4 Smoothing Methods Comparison: Accuracy & Timing

Simulation setup: $N_{\text{paths}} = 100,000$, $N_{\text{steps}} = 50$, $\alpha = 25$.

Table 10: CE / Vibrato Results

CE / Vibrato	$n = 3$	$n = 6$	$n = 10$	$n = 15$
CE price error $\lvert P_{\mathrm{CE}} - P_{\mathrm{FD}}\rvert$	1.09×10^{-2}	1.43×10^{-4}	1.18×10^{-4}	6.0×10^{-6}
Vibrato price error $\lvert P_{\mathrm{Vib}} - P_{\mathrm{FD}}\rvert$	1.10×10^{-2}	1.61×10^{-4}	1.16×10^{-4}	7.0×10^{-6}
CE Hessian error $\lVert H_{\mathrm{CE}} - H_{\mathrm{FD}}\rVert_F$	4.38×10^{-4}	8.27×10^{-5}	1.46×10^{-4}	9.55×10^{-5}
Vibrato Hessian error $\lVert H_{\mathrm{Vib}} - H_{\mathrm{FD}}\rVert_F$	3.88×10^{-4}	7.31×10^{-5}	1.21×10^{-4}	8.6×10^{-5}
CE/FD time ratio	20.45×	6.94×	2.75×	1.63×
Vibrato/FD time ratio	10.45×	3.43×	1.51×	0.84×
FD calls	19	73	201	451

Table 11: LRM / Malliavin / Kernel Results

LRM / Malliavin / Kernel	$n = 3$	$n = 6$	$n = 10$	$n = 15$
LRM price error $\lvert P_{\mathrm{LRM}} - P_{\mathrm{FD}}\rvert$	3.2×10^{-5}	5.0×10^{-5}	7.6×10^{-5}	8.5×10^{-5}
Malliavin price error $\lvert P_{\mathrm{Mal}} - P_{\mathrm{FD}}\rvert$	3.2×10^{-5}	5.0×10^{-5}	7.6×10^{-5}	8.5×10^{-5}
Kernel price error $\lvert P_{\mathrm{Ker}} - P_{\mathrm{FD}}\rvert$	1.6×10^{-5}	2.6×10^{-5}	4.1×10^{-5}	4.4×10^{-5}
LRM Hessian error $\lVert H_{\mathrm{LRM}} - H_{\mathrm{FD}}\rVert_F$	2.70×10^{-3}	7.79×10^{-1}	4.53×10^{-2}	1.293×10^{0}
Malliavin Hessian error $\lVert H_{\mathrm{Mal}} - H_{\mathrm{FD}}\rVert_F$	2.70×10^{-3}	7.79×10^{-1}	4.53×10^{-2}	1.293×10^{0}
Kernel Hessian error $\lVert H_{\mathrm{Ker}} - H_{\mathrm{FD}}\rVert_F$	9.80×10^{-5}	4.22×10^{-5}	6.64×10^{-5}	4.03×10^{-5}
LRM/FD time ratio	5.33×	5.32×	5.23×	5.63×
Malliavin/FD time ratio	5.00×	4.94×	4.87×	5.25×
Kernel/FD time ratio	3.49×	1.52×	0.81×	0.52×
FD calls	19	73	201	451

4.6 American Option Validation: LSM with Frozen Exercise Times

American options introduce an additional layer of complexity for AAD because the optimal exercise strategy is itself a function of the model parameters. Under the risk–neutral measure, the American put price can be written as

$$V(S_0, \theta) = \sup_{\tau \in \mathcal{T}} \mathbb{E}\big[e^{-r\tau}\, \Phi(S_\tau)\big],$$

where θ denotes the collection of risk–neutral parameters (e.g. S_0, σ, r), $\mathcal{T}$ is the set of admissible stopping times on a discrete exercise grid $0 < t_1 < \cdots < t_M = T$, and Φ is the intrinsic payoff. In practice we approximate the optimal stopping rule using the Least–Squares Monte Carlo (LSM) algorithm.

LSM pricing and optimal stopping. Given simulated paths $\{S_{t_k}^{(i)}\}_{k=0,\ldots,M}$, LSM estimates the continuation value at each exercise date t_k via a cross–sectional regression of discounted future cashflows on a set of basis functions $\{\psi_\ell(S_{t_k})\}$:

$$\widehat{C}(S_{t_k}) = \sum_{\ell} \beta_{k,\ell}\, \psi_\ell(S_{t_k}).$$

For each path i and time t_k we compare the immediate exercise value $\Phi(S_{t_k}^{(i)})$ with $\widehat{C}(S_{t_k}^{(i)})$ and define the stopping time

$$\tau_i^\star = \min\{t_k : \Phi(S_{t_k}^{(i)}) \geq \widehat{C}(S_{t_k}^{(i)})\},$$

with the convention that $\tau_i^\star = T$ if exercise is never optimal beforehand. The American option price is then estimated by

$$\widehat{V}(S_0, \theta) = \frac{1}{N_{\mathrm{paths}}} \sum_{i=1}^{N_{\mathrm{paths}}} e^{-r\tau_i^\star}\, \Phi(S_{\tau_i^\star}^{(i)}).$$

Difficulty for AAD and higher–order Greeks. From the perspective of differentiation, the map

$$(S_0, \theta) \longmapsto \{\tau_i^\star(S_0, \theta)\}_{i=1}^{N_{\text{paths}}} \longmapsto \widehat{V}(S_0, \theta)$$

is highly non–smooth: an arbitrarily small perturbation in S_0 or σ can flip the inequality $\Phi \geq \widehat{C}$ for some paths and exercise dates, causing a discrete jump in $\tau_i^\star$. This leads to:

- pathwise discontinuities in the payoff as a function of (S_0, θ);
- unstable or undefined second derivatives (Gamma, Vanna, Volga) when naively applying reverse–mode AAD through the exercise decision;
- large variance in Monte Carlo estimators of higher–order Greeks, even when the first–order Greeks appear reasonable.

Frozen exercise times (Frozen τ) for smooth AAD. To obtain stable Greeks, we adopt a "frozen path" or "frozen τ" approach:

1. **Outer LSM pass (no AAD).** We first run a standard LSM algorithm with a large number of paths to obtain near–optimal exercise times $\{\tau_i^\star\}_{i=1}^{N_{\text{paths}}}$.[1]
2. **Inner AAD pass (frozen stopping rule).** Keeping the stopping times $\{\tau_i^\star\}$ fixed, we re–evaluate the discounted cashflows using AAD variables for the model parameters $(S_0, \sigma, r, \dots)$. For each path i we compute

   $$\widetilde{V}_i(S_0, \theta) = e^{-r\tau_i^\star}\, \phi_\beta\big(S_{\tau_i^\star}^{(i)} - K\big),$$

   where ϕ_β is the Softplus smoothing introduced earlier.
3. **Tape–based differentiation.** The Monte Carlo average $\widetilde{V} = \frac{1}{N_{\text{paths}}} \sum_i \widetilde{V}_i$ is now a smooth function of (S_0, θ) for fixed $\{\tau_i^\star\}$. We apply Edge–Pushing on the AAD tape to extract the full Hessian with respect to the selected inputs.

This procedure effectively decouples the *combinatorial* problem of optimal stopping from the *analytic* problem of differentiating a smooth payoff. The outer LSM pass produces an approximate but fixed exercise policy, while the inner AAD pass computes derivatives of the price conditional on that policy.

Impact on Greeks. Freezing the stopping rule has two important consequences for sensitivities:

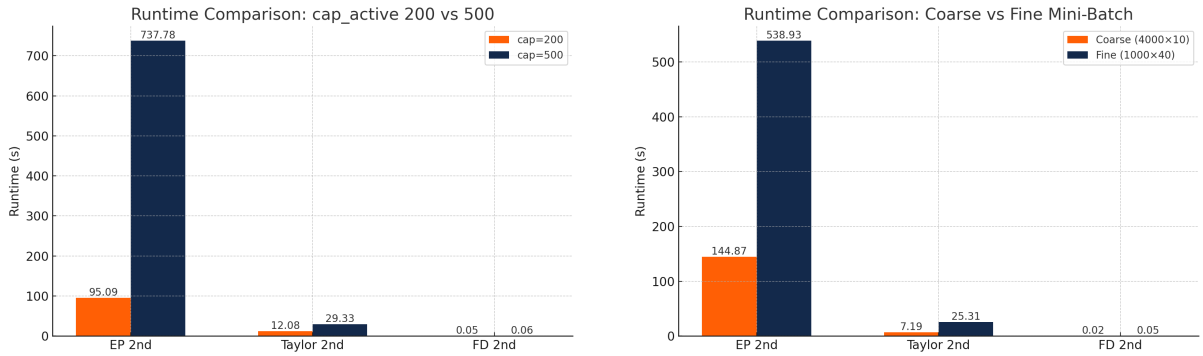
- **First–order Greeks.** Pathwise Deltas and Vegas become smooth functions of (S_0, σ, r), and their Monte Carlo estimators are well–behaved. In practice we observe that AAD–based first–order Greeks closely match finite–difference benchmarks with substantially reduced noise.
- **Second–order Greeks.** More importantly for this project, the Hessian entries (Gamma, Vanna, Volga, and cross–parameter terms) become numerically stable. The relative errors reported below, on the order of 10^{-4}–10^{-3} against finite differences, confirm that Edge–Pushing can deliver accurate second–order Greeks even in the presence of early exercise.

Softplus smoothing further regularizes the payoff near the strike, preventing Dirac–delta–like spikes in Gamma at the exercise boundary and avoiding large adjoint values on the tape.

[1] In this phase, all operations are performed on standard floating–point arrays; no adjoints are recorded.

cap_active and mini–batch design. The remaining design choices are specific to the Edge–Pushing implementation:

- `cap_active`. This parameter controls the maximum size of active edge blocks stored during the reverse sweep. Larger values reduce the number of flushes but increase memory pressure and bookkeeping cost. Our experiments show that increasing `cap_active` from 200 to 500 multiplies the second–order runtime by a factor of ~ 8 while leaving Hessian accuracy essentially unchanged. This indicates that `cap_active`=200 is already sufficient to capture the relevant curvature information for American options.
- **Mini–batch granularity.** We process Monte Carlo paths in mini–batches to control memory usage. A coarse design (4000×10) yields higher Hessian errors versus finite differences, whereas a finer design (1000×40) reduces the relative error by more than an order of magnitude. This suggests that finer batching improves the stability of second–order adjoints under early exercise.



(a) Runtime comparison of 2nd-order Greeks (b) Mini-batch granularity comparison

Figure 7: American Option Validation Results

Runtime behavior under different `cap_active` settings. Figure **??** illustrates the runtime of second-order Greeks computation for American options under different values of `cap_active`. This parameter controls the maximum number of active second-order edge blocks retained during the reverse sweep of the Edge-Pushing algorithm, and therefore directly governs the trade-off between memory usage and computational overhead.

As `cap_active` increases from 200 to 500, the runtime of the second-order EP computation rises sharply, from approximately 95 seconds to 738 seconds. This increase is driven by the higher cost of maintaining and updating larger active edge sets during the reverse traversal of the tape. Although larger `cap_active` values reduce the frequency of flushing intermediate edge blocks, the resulting bookkeeping and memory-access costs dominate the overall runtime in the American option setting. These results indicate that excessively large `cap_active` values are unnecessary and can significantly degrade performance without providing tangible benefits.

Hessian Error vs FD (2nd Order)

$$\text{rel\_err}(H_{\text{EP}}, H_{\text{FD}}) = \frac{\max |H_{\text{EP}} - H_{\text{FD}}|}{\max |H_{\text{EP}}|}$$

Table 12: cap_active Comparison

cap_active	$\max \lvert H_{\text{EP}} - H_{\text{FD}} \rvert$	Rel. Err (%)
200	1.101×10^{-1}	0.0906
500	1.101×10^{-1}	0.0906

Table 13: Mini-Batch Granularity Comparison

Setting	$\max \lvert H_{\mathrm{EP}} - H_{\mathrm{FD}} \rvert$	Rel. Err (%)
Coarse (4000×10)	3.211×10^{-1}	0.251
Fine (1000×40)	2.348×10^{-2}	0.0169

Hessian accuracy and sensitivity to batching strategy. Tables **??** and **??** report the accuracy of the Edge-Pushing Hessian relative to finite-difference benchmarks. The relative error is defined as the maximum absolute deviation normalized by the maximum Hessian entry, providing a conservative measure of second-order accuracy.

Varying `cap_active` from 200 to 500 has no observable impact on Hessian accuracy: both the maximum absolute error and the relative error remain identical across the two settings. This confirms that increasing the active edge capacity beyond a moderate level does not improve the quality of curvature information, but only increases computational cost.

In contrast, the mini-batch granularity has a pronounced effect on numerical accuracy. Refining the batch design from a coarse configuration of 4000×10 to a finer configuration of 1000×40 reduces the maximum Hessian error by more than an order of magnitude and lowers the relative error from 0.251% to 0.0169%. This improvement reflects reduced Monte Carlo noise and more stable aggregation of second-order adjoints when smaller batches are used, highlighting the importance of batch design for robust second-order Greeks in the presence of early exercise.

Overall assessment of the American option AAD engine. Taken together, these results demonstrate that accurate and efficient second-order Greeks for American options can be obtained by carefully separating algorithmic, statistical, and implementation considerations. The frozen exercise-time strategy removes the fundamental non-smoothness introduced by optimal stopping, while payoff smoothing regularizes local curvature near the exercise boundary. Within this smooth setting, Edge-Pushing provides an effective mechanism for propagating second-order information through the Monte Carlo tape.

From an implementation perspective, `cap_active` primarily controls the memory–runtime trade-off of the reverse sweep and should be kept at a moderate level to avoid unnecessary overhead, whereas mini-batch granularity plays a central role in numerical stability and accuracy. With appropriate parameter choices, the resulting American option AAD engine delivers second-order Greeks that are both accurate relative to finite differences and computationally tractable, making the approach suitable for realistic risk management applications involving early exercise.

5 Summary

5.1 Method Comparison

Table 14: Method Comparison: Bumping vs Edge-Pushing vs Taylor

Aspect	Finite Difference (Bumping)	Edge-Pushing (EP)	Taylor Expansion
Complexity	$O(n^2)$ PDE Solved	$O(1)$ PDE solve + graph traversal	$O(1)$ forward + backward sweep
Accuracy	Numerical errors from h	Exact (machine precision)	Exact (machine precision)
Memory	Low (only stores values)	High (stores full tape + edge pairs)	Medium (stores (v_0, v_1, v_2) per node)
Sparsity	N/A (no graph)	**Exploits sparsity** (sparse W matrix)	Limited (full monomials)
Dense Hessian	Slow (n^2 solves)	Slower (dense W matrix)	**Fast** (natural for dense)
Sparse Hessian	Slow (n^2 solves)	**Fast** (e.g., B-spline: 42×)	Moderate
Implementatio	Simple (black-box)	Complex (tape + graph algorithm)	Moderate (operator overloading)
Best Use Case	• Quick prototyping • Low dimensions ($n < 5$)	• **Sparse Hessian** • PDE with B-splines • High dimensions	• **Dense Hessian** • Deep computation graphs

5.2 Conclusions

1. **Production-Ready AAD Framework**: Implemented tape-based reverse-mode AD with operator overloading in Python, achieving 12× speedup via Cython optimization.

2. **Edge-Pushing for Sparse Hessians**: Achieved **42× speedup** over Bumping2 for B-spline PDE calibration by exploiting Hessian sparsity. Only 1 PDE solve required regardless of parameter count.

3. **Taylor Expansion for Dense Hessians**: Efficient for Monte Carlo simulations with correlated paths.

4. **Real Data Validation**: Successfully calibrated B-spline surface on SPX options (5,405 contracts). EP Hessian error $10^{-8} \sim 10^{-14}$ validates correctness.

5. **Validated Accuracy**: All three methods produce consistent Greeks to machine precision.

5.3 Smoothing Discussions

For correlated Euro-basket options, all methods produce prices that are indistinguishable from the softplus–FD benchmark: price errors stay below 10^{-4} for CE/Vibrato and below 10^{-4} for LRM/Malliavin/Kernel even at $n = 15$. The real separation appears in Hessians and scaling with dimension. CE and Vibrato keep $\|H - H_{\mathrm{FD}}\|_F$ at $\mathcal{O}(10^{-4})$ across all n, while their time ratios drop from about 20× (for $n = 3$) to below 1 once $n \geq 10$, reflecting the $O(n^2)$ cost of FD bumps versus essentially linear cost in n for the smoothed Monte Carlo estimators. In the LRM/Malliavin family, price errors are still tiny, but Hessian errors grow to order one by $n = 15$, signalling strong variance amplification of second-order terms as the basket dimension increases. Kernel smoothing, in contrast, maintains Hessian errors around 10^{-4} for all n while its runtime falls from 3.5× FD to about 0.5× FD at $n = 15$.

Extrapolating to higher dimensions, FD quickly becomes prohibitive and LRM / Malliavin second-order Greeks become too noisy, whereas CE/Vibrato and, in particular, kernel smoothing combined with AAD/Edge-Pushing retain both FD-level accuracy and favourable complexity. In practice this suggests using kernel+AAD as the default high-dimensional Hessian engine, with CE/Vibrato as cheap alternatives for gradients or moderate dimensions, and LRM/Malliavin mainly as unbiased but higher-variance validation tools.

5.4 Future Work

1. **Checkpointing**: Reduce memory usage for very large tapes
2. **GPU Acceleration**: Parallelize tape operations on CUDA for large-scale Monte Carlo
3. **Higher-Order Derivatives**: Extend to third-order Greeks (speed, color, ultima)
4. **Cross-Language Integration**: C++ core with Python/R/Julia interfaces
5. **Production Deployment**: Integration with trading systems and real-time risk platforms

6 References

1. Margossian, C. C. (2019). A Review of Automatic Differentiation and its Efficient Implementation. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 9(4), e1305.
2. Capriotti, L., & Giles, M. (2020). 15 Years of Adjoint Algorithmic Differentiation in Finance. Columbia University and Oxford University Mathematical Institute.
3. Capriotti, L. (2011). Fast Greeks by Algorithmic Differentiation. *Journal of Computational Finance*, 14(3), 3-35.
4. Corlay, S. (2014). B-spline Techniques for Volatility Modeling. Working Paper.
5. Pagès, G., Pironneau, O., & Sall, G. (2016). Vibrato and Automatic Differentiation for High Order Derivatives and Sensitivities of Financial Options. *hal-01334227*.
6. Gower, R. M., & Mello, M. P. (2012). A New Framework for the Computation of Hessians. *Optimization Methods and Software*, 27(2), 251-273.
7. AD-HOC: A C++ Expression Template Package for High-Order Derivatives Backpropagation.