Second Order Greeks via Adjoint Algorithmic Differentiation

UIUC MSFE

University of Illinois Urbana-Champaign

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

Dec 3, 2025

J.P.Morgan

Introduction

Corporate Advisor: Gan Wang, Vicky Luo
Sponsor Company: J.P. Morgan
Instructor: Prof. Feng

Project Motivation:

- **Problem:** Traditional finite-difference bumping scales poorly—computing n Greeks requires $O(n)$ re-pricings.
- **Solution:** Reverse-mode AAD computes all first-order Greeks in one backward sweep; Hessian AAD (Edge-Pushing & Taylor Expansion) extends this to second-order Greeks.
- **Result:** Achieved great speedup over bumping on both PDE and Monte Carlo pricing engines.

Insight of "Automatic"

- Programs = compositions of **elementary operations**
- Derivatives of each operation are known
- AD applies the **chain rule** systematically

Modes of Automatic Differentiation

- **Forward Mode**
  - Propagates derivatives alongside values
  - Efficient when number of inputs n is small
  - Computes one directional derivative per pass
- **Reverse Mode**
  - Propagates adjoints backward from output
  - Superior when

$$\text{inputs} \gg \text{outputs}$$

  - Ideal for financial models: option pricing, risk, Greeks

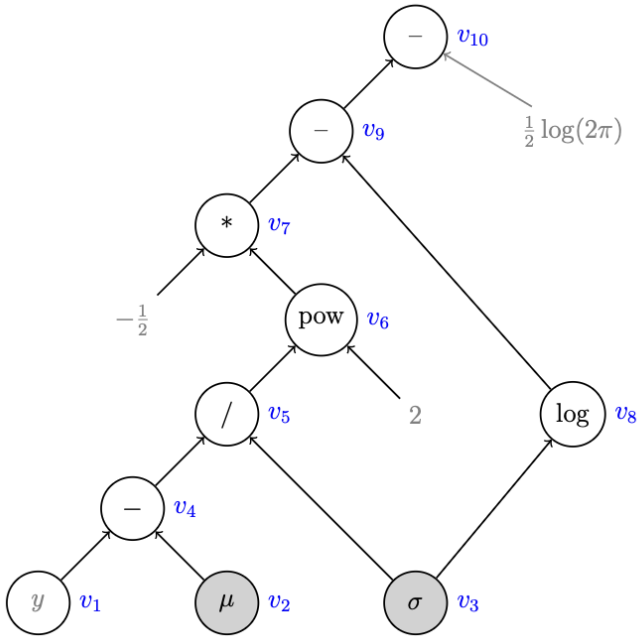


Figure 1: Expression graph for the log-normal density

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

Computation Graph

Edge

- An edge represents a function argument (and also a data dependency).
- Edges are simply pointers to nodes.

Node

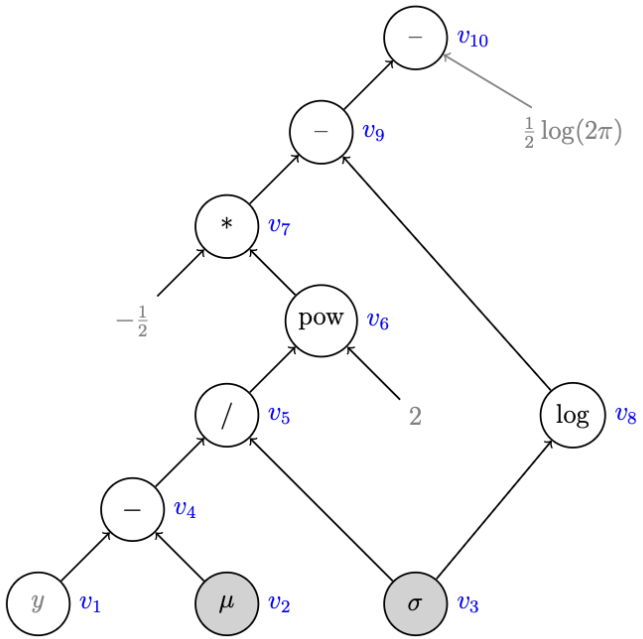
- The head node of an edge is a function of that edge's tail node.
- A node "knows" how to compute its value and the value of its derivative w.r.t. each argument (edge) multiplied by a derivative arriving from an arbitrary input, i.e.,

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

Tape

1. Topological indices of nodes (predecessor list)
2. Forward values
3. Numerical values of Jacobians / gradients

Computation graphs are usually a directed acyclic graph (DAG).



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

Reverse-Mode AAD: Efficiency and Extensions to Hessians

First-order Greeks

- Finite-difference bumping computes each Greek separately:

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

- Reverse-mode AAD yields *all* first-order Greeks in a single backward sweep:

Time complexity

- Finite-difference bumping: $\mathcal{O}(n)$ evaluations of the pricing function.
- Reverse-mode AAD: $\mathcal{O}(1)$ backward sweep (independent of n).
- Thus, when the number of inputs is large, AAD decisively outperforms bumping.

Beyond first order: towards Hessians

- To extend AAD-style efficiency to second-order Greeks (Hessians), we draw on the framework of Gower & Mello (2012), *A new framework for the computation of Hessians*.
- This introduces the **edge-pushing algorithm**, which implements the chain rule for second-order derivatives and propagates Hessian through the graph.
- Our benchmark method for comparison remains finite-difference bumping.

Function f and its Hessian f''

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

where

- $x \in \mathbb{R}^n$: input vector
- $P = [I_n \ 0]$: embedding matrix, $P^\top x = (x, 0, \ldots, 0) \in \mathbb{R}^{n+\ell}$
- Φ_i: i-th state transformation, updates only one register (one component) of the state vector
- $e_\ell^\top$: selects the final register (output)

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

where

- Φ_j': Jacobian of Φ_j, $\quad \Phi_j''$: Hessian
- $\bar{v}^i$: adjoint vector (backpropagated weight)
- $P, P^\top$: map between input space and extended state space

Hessian's Chain Rule

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

Block form:

$$f'' = PWP^\top, \quad \text{with } W = \sum_{i=1}^{\ell} W_i$$

Previous summand W_{i-1}:

$$W_{i-1} = \underbrace{\left((\Phi_1')^\top \cdots (\Phi_{i-2}')^\top \right)}_{\text{left factors}} \left((\bar{v}^{\,i-1})^\top \right) \underbrace{\left(\Phi_{i-1}'' \right)}_{\text{local Hessian at } i-1} \underbrace{\left(\Phi_{i-2}' \cdots \Phi_1' \right)}_{\text{right factors}}$$

Current summand W_i:

$$W_i = \underbrace{\left((\Phi_1')^\top \cdots (\Phi_{i-2}')^\top \right)}_{\text{same as before}} \underbrace{(\Phi_{i-1}')^\top}_{\text{extra}} \left((\bar{v}^{\,i})^\top \right) \underbrace{\left(\Phi_i'' \right)}_{\text{local Hessian at } i} \underbrace{\Phi_{i-1}'}_{\text{extra}} \underbrace{\left(\Phi_{i-2}' \cdots \Phi_1' \right)}_{\text{same as before}}$$

Block Form Three-Step Logic

Core idea: Each node performs the sequence **Push** $\rightarrow$ **Create** $\rightarrow$ **Update** to incrementally construct the Hessian, with each sweep adding one more W_i summand to W.

1. Push Step — This allows W to gain one additional summand.

$$W \quad \leftarrow \quad (\phi_i')^T \, W \, \phi_i'$$

Propagates existing second-order information to the parent nodes through the local Jacobian.

2. Create Step

$$W \quad \leftarrow \quad W + \bar{v}^T \phi_i''$$

Introduces new second-order contributions at the current node, arising from its local Hessian.

3. Update Step

$$\bar{v}^T \quad \leftarrow \quad \bar{v}^T \phi_i'$$

Performs the standard first-order adjoint update (reverse-mode propagation).

Componentwise Form Applied to a Computation Graph

- **Sweeping node 3:** A new nonlinear arc $\{1, 2\}$ is created.
- **Sweeping node 2:** The arc $\{1, 2\}$ is pushed and split into
  - $\{0, 1\}$ with weight $1 \cdot 2v_0$,
  - $\{-1, 1\}$ with weight $1 \cdot 3$.
- **Sweeping node 1:** The arc $\{-1, 1\}$ is pushed and split into
  - $\{-2, -1\}$,
  - $\{-1, -1\}$ with weight $2 \cdot 3 \cdot e^{v-1}$.

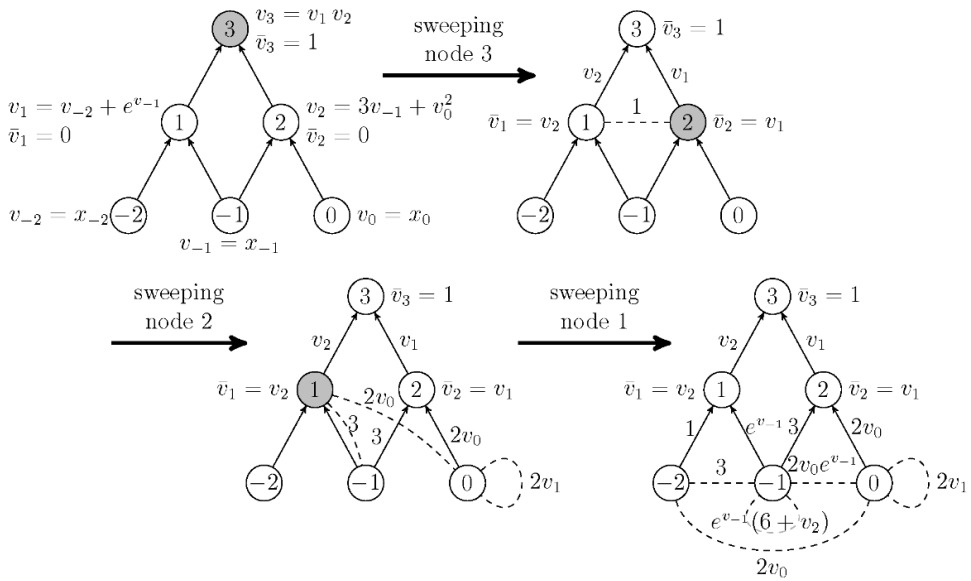


Figure. Edge-pushing applied to a computational graph of $f(x) = (x_{-2} + e^{x-1})(3x_{-1} + x_0^2)$.

Taylor Expansion as an Alternative Hessian Engine

1. Taylor Forward (local expansion)

- Each operator is expanded to second order:

$$f(z+\varepsilon) \approx f(z) + f'(z)\varepsilon + \tfrac{1}{2} f''(z)\varepsilon^2.$$

- Forward sweep propagates (v_0, v_1, v_2) at every node.
- All second-order monomials $(\varepsilon_i^2,\ \varepsilon_i \varepsilon_j)$ generated locally.

2. Taylor Backpropagation (one reverse sweep)

- Forward sweep already provides derivatives of intermediates.
- One reverse sweep folds all monomials back to inputs:

$$H_{ij} = \sum_{k,m} \frac{\partial^2 V}{\partial u_k \partial u_m} \frac{\partial u_k}{\partial x_i} \frac{\partial u_m}{\partial x_j} + \cdots$$

3. Comparing Hessian Engines: Edge–Pushing vs Taylor

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 sweep**
Scaling with steps	graph grows with time	**linear in depth**
Hessian symmetry	needs explicit enforcement	**automatic**

Takeaway: Taylor avoids graph blow-up, preserves symmetry, and provides scalable second–order derivatives.

Implementation of the Taylor Expansion Engine

1. Taylor AD Engine

- Each `ADVar` node stores (v_0, v_1, v_2).
- Overloaded ops (`+`, `-`, `*`, `exp`, `log`, `sqrt`, `norm_cdf`, ...) propagate local second-order Taylor terms through the graph.
- `taylor_backpropagate(target)`: a single reverse sweep returns the full Hessian w.r.t. $(S_0, \sigma, K, r, T, \dots)$.

2. BSM 5D Benchmark (accuracy & speed)

Black–Scholes 5D setting with inputs (S_0, σ, K, r, T); Edge–Pushing (EP) used as the reference Hessian.

Hessian accuracy vs EP

Method	Frobenius rel. error	Comment
Taylor	1.82e−02	matches EP structure
Finite Difference (FD)	1.83e−02	slightly noisier

Runtime (baseline = EP)

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

Taylor achieves EP-level accuracy, with a predictable single-tape structure. FD is extremely fast but less robust for higher-dimensional Greeks.

Implementation of Taylor Expansion within a PDE Framework

1. Embedding Taylor Expansion into the PDE Solver

- Treat PDE coefficients a, b, c as differentiable functions of AD variables.
- Each grid state u^n becomes an `ADVar` that carries (v_0, v_1, v_2).
- Crank–Nicolson update remains inside the AD graph:

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

- One call to `taylor_backpropagate()` returns the entire gradient and Hessian w.r.t. any model parameter (no additional adjoint rules required).

2. Extension to Local Volatility (Dupire SVI)

- Replace constant σ with Dupire local vol $\sigma_{\text{loc}}(K, T)$ from an SVI surface.
- PDE coefficients become nonlinear in (S, t) and depend on SVI parameters $\{a_i, b_i, \rho_i, m_i, \eta_i\}$.
- Taylor automatically differentiates all PDE coefficients, preserving Hessian symmetry under highly nonlinear local vol dynamics.

Takeaway: Taylor provides a fully embedded second–order engine inside the PDE solver, enabling gradients and Hessians w.r.t. all SVI parameters in a single pass.

PDE + Taylor + SVI Results (BSM Setting)

Dupire–SVI local volatility. Grid $M = 121$, $N_{\text{base}} = 100$. 25 SVI parameters $\{a_i, b_i, \rho_i, m_i, \eta_i\}_{i=0}^{4}$.

1. Gradient benchmark: Taylor vs FD (25 parameters)

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

2. Full 25×25 SVI Hessian: Taylor vs FD

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

Conclusion: Taylor more costly for gradients, but for **full Hessian** it is $\sim 7\times$ faster than bumping FD with strong accuracy.

Optimization Stages: From Python to Cython

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. Enables type declarations and direct C API calls.
- **Python dict:** Native Python dictionary with $O(1)$ average-case hash lookups. Leverages highly optimized CPython implementation for edge storage in sparse Hessian graphs.
- **C++ unordered_map:** STL hash table with comparable performance but higher implementation overhead. Pure C++ alternative when full compilation is needed.

Key Finding: Cython + Python dict achieves **12× speedup** with minimal complexity, outperforming pure C++ alternatives.

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
- **Maturities:** Multiple expiration dates

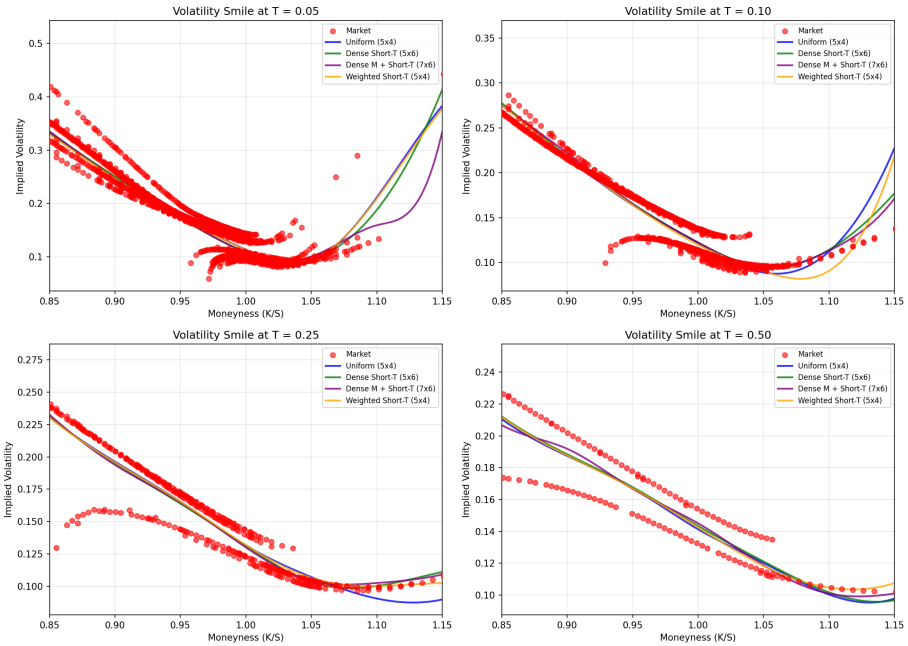
B-spline Parametrization

The calibrated B-spline weights $\mathbf{w} = (w_1, \ldots, w_m)$ define a smooth local volatility surface:

$$\sigma_{\text{loc}}(K, T; \mathbf{w}) = \sum_{i=1}^{m} w_i B_i(K, T),$$

where $B_i(K, T)$ are tensor-product cubic B-spline basis functions.

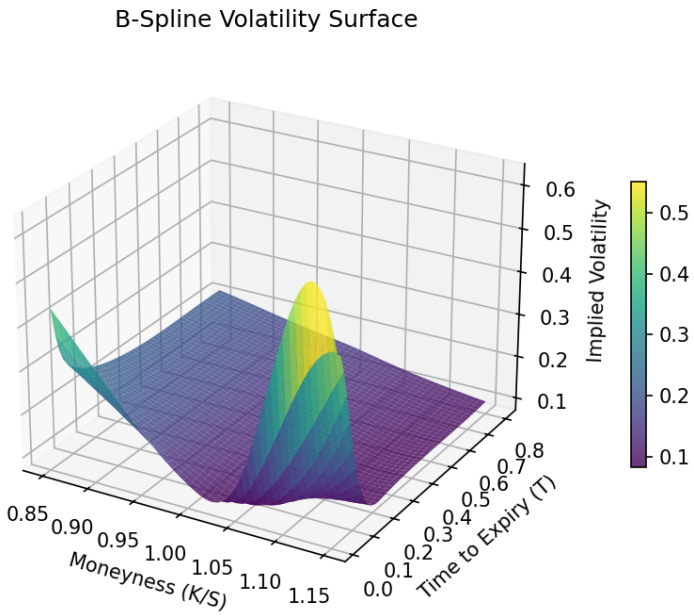
Volatility Smiles: Market vs B-spline Fit

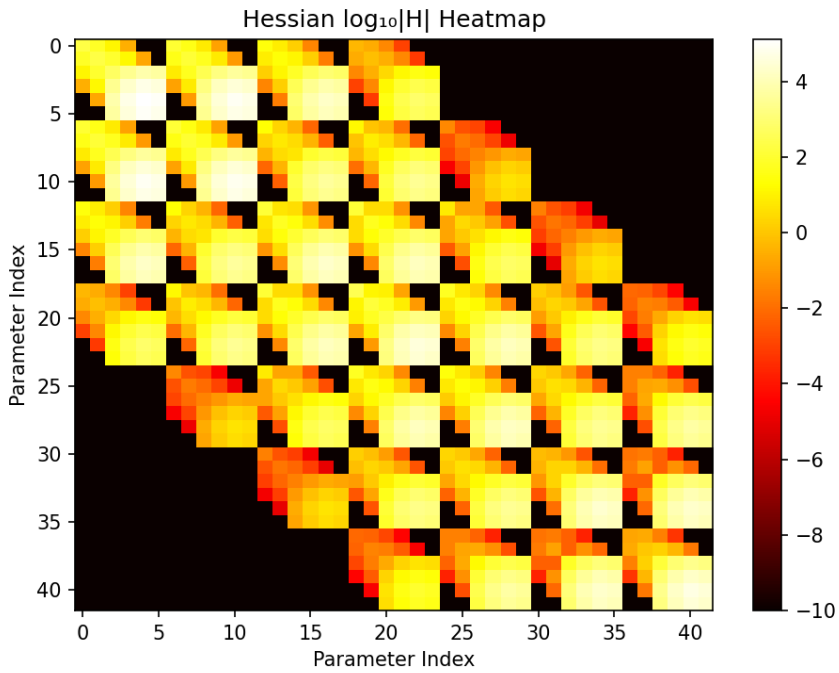


Calibration Method:

- **Optimizer:** L-BFGS-B with AAD gradients
- **Objective:** Minimize pricing error between market and model prices
- **EP Hessian:** Used for validation and second-order optimization

Calibrated B-spline Volatility Surface





Surface Properties:

- **Smoothness:** C^2 continuity across the entire (K, T) domain
- **Locality:** Each basis function has compact support $\Rightarrow$ sparse Hessian
- **Linearity:** σ_{loc} depends linearly on weights $\mathbf{w}$ $\Rightarrow$ PDE coefficients have simple $\mathbf{w}$-dependence
- **Efficiency:** Gradient and Hessian w.r.t. $\mathbf{w}$ computed via AAD Edge-Pushing

B-splines + PDE for Fast Hessian Computation

1. B-spline parametrization of the model

- $$\sigma_{\text{loc}}(t, x; \mathbf{w}) \;=\; \sum_{i=1}^{m} w_i \, B_i(x),$$

  where $\mathbf{w} = (w_1, \ldots, w_m)$ are the spline weights (calibration parameters).
- This gives a smooth, low-dimensional and *linear* dependence of the PDE coefficients on $\mathbf{w}$.

2. PDE structure and dependence on B-splines

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma_{\text{loc}}^2(t, x; \mathbf{w})\, x^2 \, \frac{\partial^2 V}{\partial x^2} + rx\frac{\partial V}{\partial x} - rV = 0,$$

$$A(\mathbf{w})\, V^{n+1} = B(\mathbf{w})\, V^n,$$

where all matrix entries depend on $\mathbf{w}$ *only through the spline expansion.*

- The dependence is smooth and local: at each grid point x_j,

$$\sigma_{\text{loc}}(t_n, x_j; \mathbf{w}) = \sum_{i:\, B_i(x_j) \neq 0} w_i \, B_i(x_j),$$

$$\mathbf{w} \;\longmapsto\; V(0, x_0; \mathbf{w}),$$

Numerical Verification: EP vs Taylor vs Bumping2

Results: 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	412.3 sec	1252 sec	EP: 13.4× faster

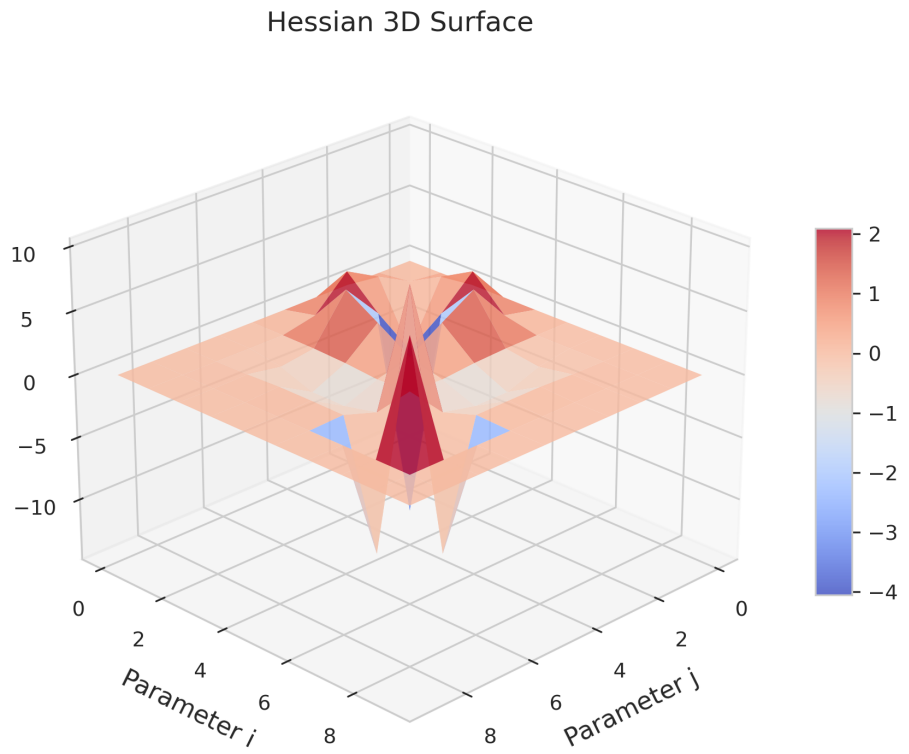
Results: 14-Parameter System ($n_{\text{knots}} = 10$)

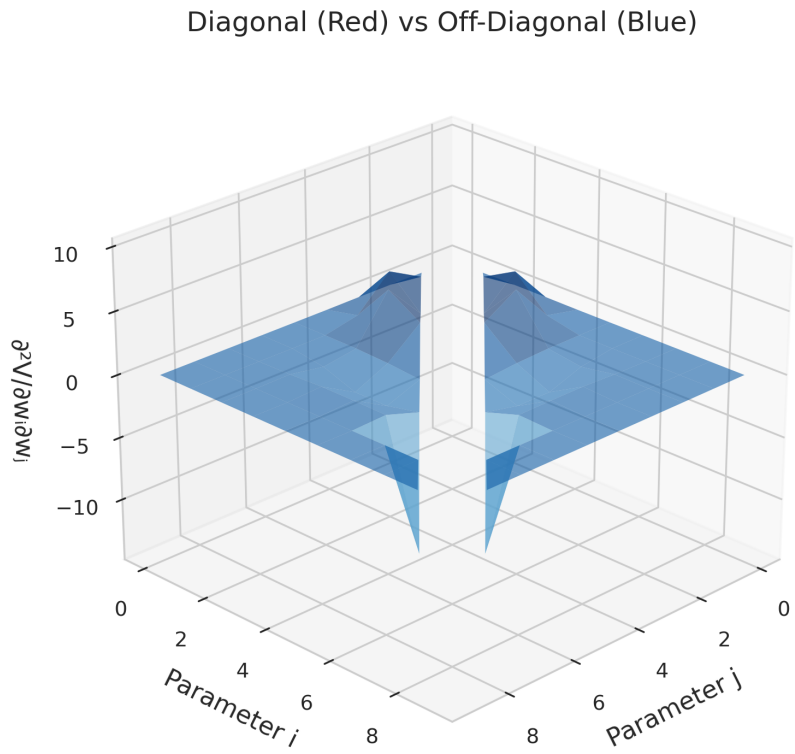
Metric	Edge-Pushing	Taylor	Bumping2	Best Method
Price	12.0485	12.0485	12.0485	All match
Max Element Error	—	—	—	$<0.35\%$
PDE Solved	**1**	**1**	**420**	EP/Taylor: 420× fewer
Computation Time	148 sec	885 sec	4,701 sec	EP: 32× faster

Results: 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	1221 sec	6451 sec	EP: 42× faster

Edge-Pushing Hessian Visualization





10×10 Hessian for B-spline PDE Calibration

- **Sparse structure:** Only 36% non-zero elements due to B-spline compact support
- **Edge-Pushing:** Exploits sparsity to compute full Hessian in one PDE solve
- **Validation:** EP vs numerical Hessian error $< 10^{-8}$ (relative Frobenius norm)
- **Speedup:** 42× faster than Bumping2 for 19-parameter system

Pathwise Differentiation for Basket Option

1. Model Setup

$$B_T = \sum_{i=1}^{n} w_i S_{T,i}, \quad P = e^{-rT}(B_T - K)^+, \quad V = \mathbb{E}[P(\mathbf{S}_T(\theta))].$$

2. Pathwise Estimators (The Chain Rule) Interchanging differentiation and expectation:

$$\text{Gradient:} \quad \frac{\partial V}{\partial \theta_k} = \mathbb{E}\left[\frac{\partial P}{\partial B_T}\frac{\partial B_T}{\partial \theta_k}\right]$$

$$\text{Hessian:} \quad \frac{\partial^2 V}{\partial \theta_k \partial \theta_m} = \mathbb{E}\left[\frac{\partial^2 P}{\partial B_T^2}\frac{\partial B_T}{\partial \theta_k}\frac{\partial B_T}{\partial \theta_m} + \frac{\partial P}{\partial B_T}\frac{\partial^2 B_T}{\partial \theta_k \partial \theta_m}\right]$$

3. Payoff Derivatives For the Call Option Payoff:

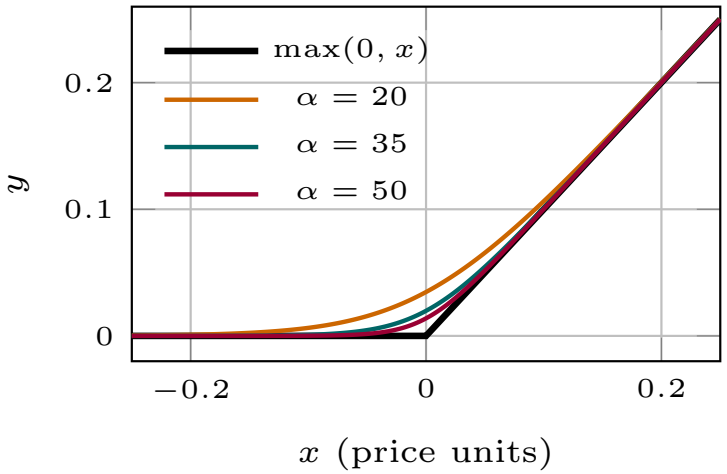
- **1st Order:** $\frac{\partial P}{\partial B_T} = e^{-rT}\mathbf{1}_{\{B_T > K\}}$ (Digital Payoff).
- **2nd Order:** $\frac{\partial^2 P}{\partial B_T^2} = e^{-rT}\delta(B_T - K)$.

  (Note: Handling the Dirac mass δ requires smoothing.)

Softplus Smoothing

Smooth surrogate for the hard payoff:

$$\text{softplus}_\alpha(x) = \frac{1}{\alpha}\ln\left(1+e^{\alpha x}\right) \approx \max(0,x) \quad (\alpha\uparrow)$$



Note: domain clipped to $[-0.25, 0.25]$.

- $\alpha\uparrow$: **sharper** kink, closer to hard payoff; price bias $\downarrow$ but curvature near $x\approx 0$ increases $\Rightarrow$ MC variance / numerical fragility $\uparrow$.
- $\alpha\downarrow$: **smoother** kink, further from hard payoff; price bias $\uparrow$ but typically lower variance and more stable second derivatives.

Softplus bias (vs. ReLU): The maximum pointwise error occurs at the money ($x = 0$):

$$b_\alpha^{\max} = \text{softplus}_\alpha(0) - \max(0,0) = \frac{\ln 2}{\alpha} \approx \frac{0.693}{\alpha}.$$

Softplus α Selection

Representative α results:

α	Γ rel	Γ mean$\lvert\Delta\rvert$	Vn rel	Vn mean$\lvert\Delta\rvert$	Vl rel	Vl mean$\lvert\Delta\rvert$
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

Verdict: Choose $\alpha = \mathbf{30}$.

- Best trade-off: accuracy,variance and speed.
- EP vs. FD speedup: $\mathbf{1.13}\times$.

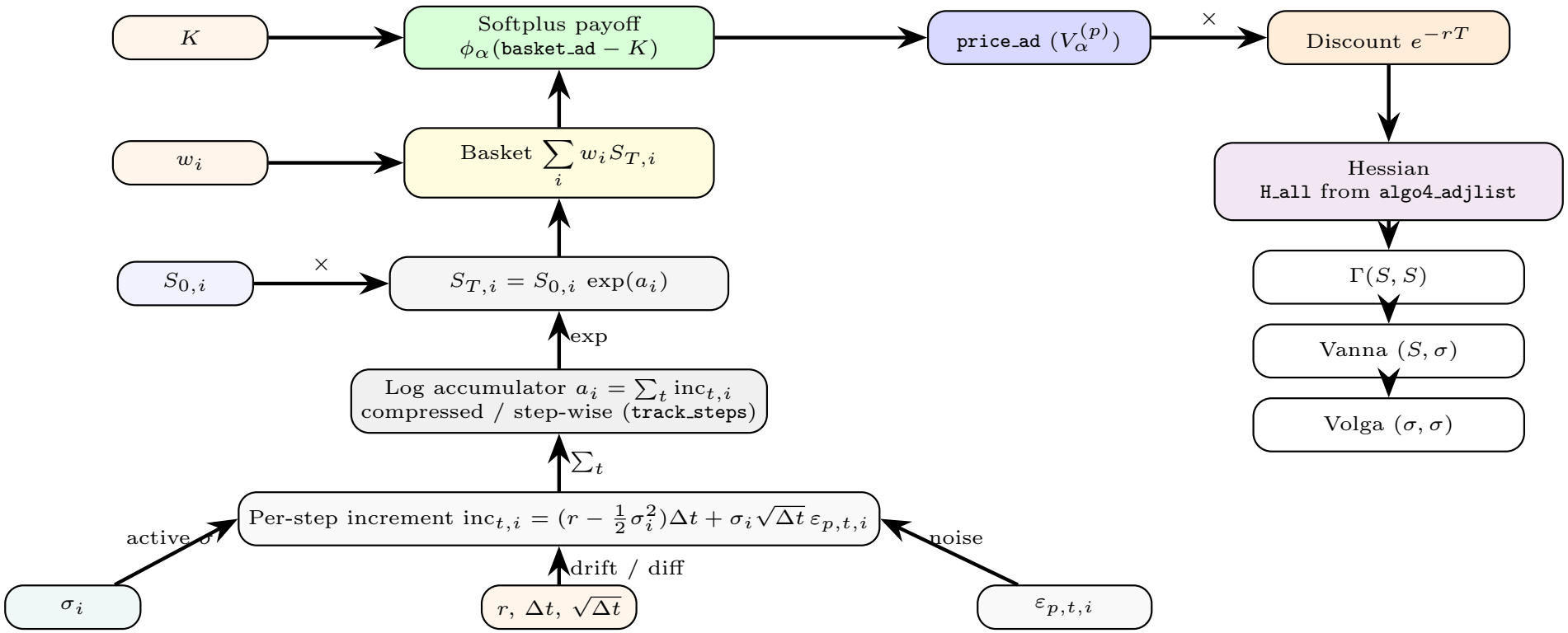
Key stats at $\alpha = 30$:

- Γ rel err: 4.86×10^{-3}
- Vanna/Volga rel err: $\sim 10^{-4}$

Context & Trade-offs:

- **Small α (e.g., 10):**
  Higher Vanna/Volga stability, but higher bias and slower execution ($0.94\times$).
- **Large α (e.g., 100):**
  Fastest speed ($1.18\times$), lower bias but errors increase significantly on account of variance.

EP computation graph for Basket option



- `H_all` is the $2n \times 2n$ Hessian with respect to `[S_vars; Sig_vars]` on a single tape.
- Gamma, Vanna and Volga come from the three blocks:

$$\begin{aligned}\text{Gamma}(S,S) &= H_{\text{all}}[0{:}n,\ 0{:}n],\\ \text{Vanna}(S,\sigma) &= H_{\text{all}}[0{:}n,\ n{:}2n],\\ \text{Volga}(\sigma,\sigma) &= H_{\text{all}}[n{:}2n,\ n{:}2n].\end{aligned}$$

EP vs FD: Accuracy and Speed on Basket option

Accuracy at $n = 4$ (alpha=30, 10000 paths, 50 steps)

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

EP Greeks track the FD baseline very closely; discrepancies are at the level of Monte Carlo noise.

Timing vs dimension (alpha=30, 10000 paths, 50 steps).

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×

As dimension n grows, FD cost scales combinatorially in the number of bumps, while EP keeps a single-tape Hessian and benefits from relatively better scaling.

Conditional Expectation & Vibrato

CE idea: Move the non-smoothness away in the direction of a chosen asset. Use the last-step conditional expectation conditional on $S_{N-1}^{(k)} = x$:

$$\begin{aligned}\phi_k(x) &= \mathbb{E}\big[f(S_T) \mid S_{N-1}^{(k)} = x\big] \\ &= \int_{\mathbb{R}^d} f(z)\, p_T(z \mid x)\, dz.\end{aligned}$$

- $p_T(z \mid x)$: conditional joint density of $S_T = z$ given the (kth) asset satisfies $S_{N-1}^{(k)} = x$.
- f may be non-smooth in the basket direction, but $p_T(\cdot \mid x)$ is C^∞ in x and differentiable.
- $\phi_k(x)$ can be viewed as the basket payoff f smoothed by the transition density p_T in the k-th asset direction, so ϕ_k becomes smooth in x.

Vibrato Goal: Compute $\nabla_x \phi(x)$ where $\phi_k(x)$ comes from the previous CE construction.

Step 1. (Score identity)

$$\begin{aligned}\nabla_x \phi(x) &= \int f(z) \nabla_x p_T(z \mid x)\, dz \\ &= \mathbb{E}\big[f(X_T) \nabla_x \log p_T(X_T \mid x) \mid X_{N-1} \\ &= x.\end{aligned}$$

Step 2. (Vibrato variance reduction)

$$\nabla_x \phi(x) := \mathbb{E}\Big[(f(X_T) - \phi(x)) \nabla_x \log p_T(X_T \mid x) \,\Big|\, \mathrm{X}_{N-1} = x\Big]. \Leftarrow \mathbb{E}[\nabla_x \log p_T \mid x] = 0$$

- CE smooths f by integrating over $p_T(\cdot \mid x)$, removing the strike kink.
- Vibrato differentiates this smooth $\phi(x)$ via the score term. The baseline $\phi(x)$ cancels constant parts $\Rightarrow$ low variance, stable Δ.

Likelihood Ratio Method & Malliavin

LRM idea: Move derivative inside integral:

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

- The derivative acts on the smooth transition density p_T, not directly on the payoff f.
- LRM replaces pathwise differentiation of a nonsmooth f by differentiation of the smooth $\log p_T$, so the estimator remains valid across the strike kink.
- Unbiased LRM estimator for $\Delta = \frac{dF}{dx_0}$:

$$\Delta = \mathbb{E}\Big[f(X_T)\, w(X_T; x_0) \,\Big|\, X_0 = x_0 \Big],$$
$$w(X_T; x_0) = \partial_{x_0} \log p_T(X_T; x_0).$$

Malliavin Integration by Parts

If $F = f(X_T) \in \mathbb{D}^{1,2}$ and $U \in L^2([0,T])$,

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

with $D_s F = f'(X_T) D_s X_T$. Choose U so that

$$\langle DX_T, U \rangle_{L^2} = \partial_{x_0} X_T =: G,$$
$$\Delta = \partial_{x_0} \mathbb{E}[f(X_T)] = [f'(X_T) G],$$
$$H_{x_0} = \delta(U).$$

Why it handles the strike:

- IBP moves the derivative from the *nonsmooth* f' to the *smooth* noise via $\delta(U)$.
- Final estimator uses $f(X_T)$ only (no f') $\Rightarrow$ unbiased and stable across the strike kink.

Kernel & Softplus + FD

Payoff smoothing (Kernel):

$$\begin{aligned} f_\varepsilon(x) &= (f * \rho_\varepsilon)(x) \\ &= \mathbb{E}[f(x+\varepsilon Z)] \\ &= \int_{\mathbb{R}} f(x+\varepsilon z)\,\rho(z)\,dz, \\ \rho_\varepsilon(u) &= \varepsilon^{-1}\rho(u/\varepsilon). \end{aligned}$$

Smooth gradient:

$$\nabla f_\varepsilon(x) = -\frac{1}{\varepsilon}\,\mathbb{E}_Z\left[f(x+\varepsilon Z)\,\nabla \log \rho(Z)\right].$$

- Replace f by f_ε in Monte Carlo.
- Convolution removes the discontinuity of f' at $x = K$.
- ∇f_ε exists $\forall x$, bounded near K.
- Equivalent to adding small noise εZ before differentiation.

Payoff smoothing (Softplus + FD):

$$f_\alpha(x) = \frac{1}{\alpha}\log\left(1 + e^{\alpha(x-K)}\right),$$
$$f_\alpha(x) \downarrow f(x) \text{ as } \alpha \to \infty.$$

Smooth derivatives:

With $\sigma_\alpha(u) = \dfrac{1}{1+e^{-\alpha u}}$, f'$_\alpha(x) = \sigma_\alpha(x-K)$, a smooth transition from 0 to 1 near $x = K$ (instead of the jump of $f'(x) = \mathbf{1}_{\{x>K\}}$).

- The kink in $(x-K)^+$ is replaced by a smooth region of width $\sim 1/\alpha$.
- f_α has a well-behaved curvature near the strike, so FD Greeks are stable and low-variance.
- α trades bias for variance; with finite α, small h, and enough paths, the FD Hessian converges to the Hessian of the smoothed payoff f_α.

Accuracy & Timing vs Assets

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

CE / Vibrato	$n = 3$	$n = 6$	$n = 10$	$n = 15$
CE price error $\lvert P_{\text{CE}} - P_{\text{FD}} \rvert$	1.09×10^{-2}	1.43×10^{-4}	1.18×10^{-4}	6.0×10^{-6}
Vibrato price error $\lvert P_{\text{Vib}} - P_{\text{FD}} \rvert$	1.10×10^{-2}	1.61×10^{-4}	1.16×10^{-4}	7.0×10^{-6}
CE Hessian error $\lVert H_{\text{CE}} - H_{\text{FD}} \rVert_F$	4.38×10^{-4}	8.27×10^{-5}	1.46×10^{-4}	9.55×10^{-5}
Vibrato Hessian error $\lVert H_{\text{Vib}} - H_{\text{FD}} \rVert_F$	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

LRM / Malliavin / Kernel	$n = 3$	$n = 6$	$n = 10$	$n = 15$
LRM price error $\lvert P_{\text{LRM}} - P_{\text{FD}} \rvert$	3.2×10^{-5}	5.0×10^{-5}	7.6×10^{-5}	8.5×10^{-5}
Malliavin price error $\lvert P_{\text{Mal}} - P_{\text{FD}} \rvert$	3.2×10^{-5}	5.0×10^{-5}	7.6×10^{-5}	8.5×10^{-5}
Kernel price error $\lvert P_{\text{Ker}} - P_{\text{FD}} \rvert$	1.6×10^{-5}	2.6×10^{-5}	4.1×10^{-5}	4.4×10^{-5}
LRM Hessian error $\lVert H_{\text{LRM}} - H_{\text{FD}} \rVert_F$	2.70×10^{-3}	7.79×10^{-1}	4.53×10^{-2}	1.293×10^{0}
Malliavin Hessian error $\lVert H_{\text{Mal}} - H_{\text{FD}} \rVert_F$	2.70×10^{-3}	7.79×10^{-1}	4.53×10^{-2}	1.293×10^{0}
Kernel Hessian error $\lVert H_{\text{Ker}} - H_{\text{FD}} \rVert_F$	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

American Option Pricing Framework (MC + LSM + AAD)

1. American Option as Optimal Stopping

The value of an American option can be written as an optimal stopping problem:

$$V(t, S_t) = \sup_{\tau \in \Theta_{t,T}} \mathbb{E}\left[e^{-r(\tau - t)} \psi(S_\tau)\right],$$

where:

- $\Theta_{t,T}$ is the set of admissible stopping times,
- $\psi(\cdot)$ is the discounted payoff at exercise,
- The optimal exercise time is

$$\tau^* = \inf\{u \geq t : V(u, S_u) = \psi(S_u)\}.$$

This formulation already shows that the value depends on an *optimal boundary* separating the exercise and continuation regions.

2. Free Boundary and PDE View

In the continuous-time PDE setting, V solves an obstacle problem:

$$\max\left(\psi - u,\ u_t + \mathcal{L}u\right) = 0,$$

where the free boundary is the moving interface between $\{u = \psi\}$ (exercise region) and $\{u > \psi\}$ (continuation).

- At this interface the $\max(\cdot)$ creates a *kink*.
- The gradient and Hessian of u can change abruptly across the free boundary.
- This is exactly what makes American options more delicate for sensitivity analysis than European options.

Our goal is to design an AAD framework that respects this free-boundary structure, while still allowing stable second-order Greeks.

LSM Implementation and the Need for Freezing τ

3. Longstaff–Schwartz Monte Carlo

In practice we approximate the optimal stopping rule using LSM:

$$V_t = \max\Big(\text{payoff}_t,\ \mathbb{E}[V_{t+\Delta t} \mid S_t]\Big),$$

where the conditional expectation is estimated via regression, e.g.

$$\mathbb{E}[V_{t+\Delta t} \mid S_t] \approx \beta_0 + \beta_1 S_t + \beta_2 S_t^2.$$

- Each Monte Carlo path may exercise at a different date.
- Once exercised, the path stops contributing future cashflows.
- The collection of exercise decisions approximates the free boundary in the PDE picture.

4. Why This is Hard for AD

In a naive implementation:

- The exercise rule is re-evaluated every time we bump (S_0, σ, r).
- Tiny perturbations can change which side of the $\max(\cdot)$ a path falls on.
- The computational graph branches differently under different bumps.

Consequences for differentiation:

- The price map $V(S_0, \sigma, r)$ becomes piecewise smooth with non-differentiable kinks.
- Direct reverse-mode AD through such branching produces noisy, unstable, or even meaningless second-order Greeks.

This motivates a modification of the algorithm rather than the AD tool: the **Frozen-τ** method.

Frozen-τ Method and Smoothness

5. Frozen-τ Method (How)

We separate the problem into two stages:

- **Stage 1:** run a standard LSM algorithm once and, for each path i, record the first exercise time

$$\tau_i = \text{first } t \text{ where payoff}_t \geq \text{continuation}_t.$$

- **Stage 2:** during AAD, we reuse the same simulated paths and keep all τ_i fixed.

With τ_i frozen, the pricing formula becomes

$$V(S_0, \sigma, r) \approx \frac{1}{N} \sum_{i=1}^{N} e^{-r\tau_i} \, \psi(S_{\tau_i}^{(i)}),$$

which is a smooth function of (S_0, σ, r) given the fixed exercise times.

6. Smoothness Comparison

Aspect	No Freeze	Frozen-τ
Exercise logic	Decision flips	Fixed from 1st pass
Computational graph	Branching	Deterministic
Differentiability	Non-differentiable	Smooth
AD stability	Unstable / noisy	Stable 1st&2nd

Interpretation:

- Freezing τ removes the free-boundary effect from the AD tape.
- Any pathwise decision change is pre-captured in fixed τ_i.
- Pricing becomes smooth in (S_0, σ, r).
- Enables stable first- and second-order Greeks via reverse-mode AAD.

AAD Engine and Scalability Settings

7. AAD Setup for Second-Order Greeks

Treat pricing parameters as AD variables:

$$x = (S_0, \sigma, r), \qquad V = V(x),$$

and extract the partial Hessian:

$$H_{ij} = \frac{\partial^2 V}{\partial x_i \partial x_j}, \quad i, j \in \{S_0, \sigma, r\}.$$

Main second-order sensitivities:

$$\Gamma = V_{S_0 S_0}, \quad \text{Vanna} = V_{S_0 \sigma}, \quad \text{Volga} = V_{\sigma\sigma}.$$

Two AAD Hessian engines implemented:

- **Edge-Pushing (EP)** – recovers Hessian via adjacency list.
- **Taylor-backprop** – propagates local Taylor coefficients.

$\rightarrow$ **Frozen-τ** ensures smooth tape, allowing both engines to compute all gradients and Hessian elements from *a single reverse sweep*.

8. Scalability Settings (for Large Monte Carlo)

Even with one reverse pass, Hessian computation over thousands of paths is memory-intensive. We control cost reduction using three mechanisms:

- **mini-batch:** Split paths into smaller batches so Hessian accumulation uses memory $\mathcal{O}(\text{batch})$ instead of $\mathcal{O}(N)$.
- `max_batch`**:** Hard upper bound on reverse-sweep size to prevent RAM spikes and avoid EP tape blow-up.
- `cap_active`**:** Remove inactive paths after Frozen-τ, reducing Hessian cost by 20–40% and improving EP stability.

These settings allow AAD-based Hessian computation to scale to tens of thousands of paths without sacrificing accuracy.

American Option Validation: Effect of `cap_active`

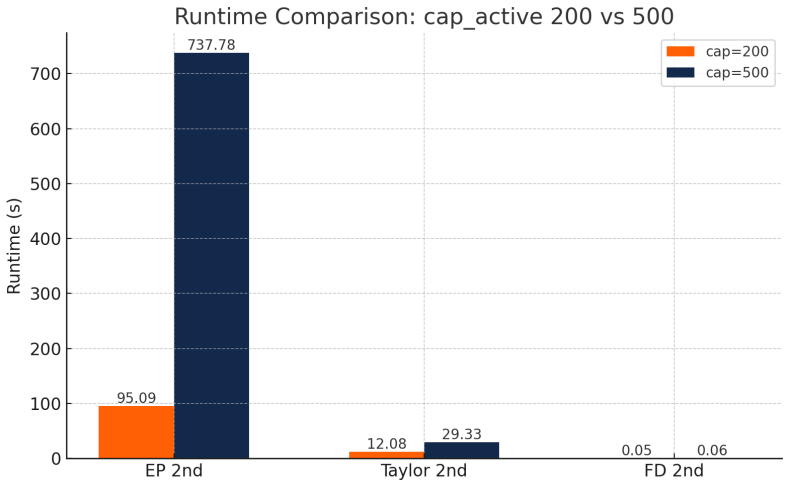


Figure 2: *

Runtime comparison of 2nd-order Greeks.

Hessian Error vs FD (2nd Order)

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

`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

Interpretation

- Increasing `cap_active` from 200 to 500 causes the EP 2nd-order runtime to grow from **95s → 738s**.
- Hessian accuracy stays unchanged (identical max error and relative error).
- **Conclusion:** `cap_active=200` already achieves FD-level accuracy while being **8× faster**.

American Option Validation: Mini-Batch Granularity

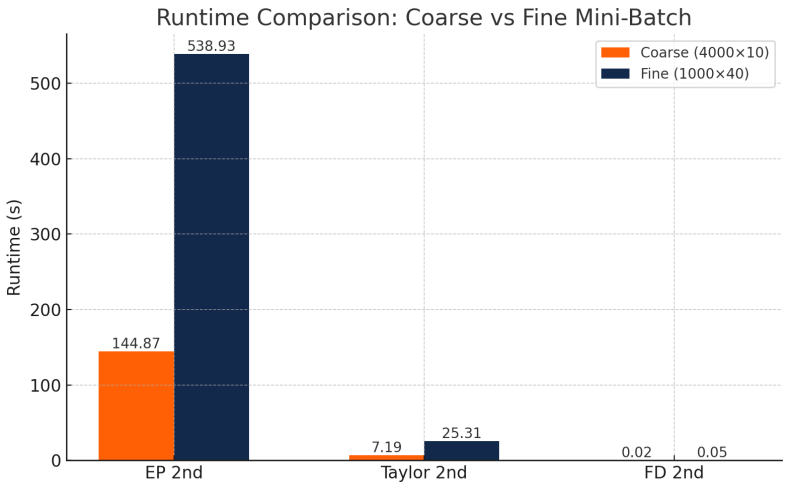


Figure 3: *

Runtime comparison under different mini-batch designs.

Hessian Error vs FD (2nd Order)

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

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

Interpretation

- Total paths fixed at 40,000; only batching granularity changed.
- **Coarse** batching runs much faster but shows larger EP–FD Hessian discrepancy.
- **Fine** batching significantly improves Hessian accuracy.
- **Conclusion:** fine batching for benchmark accuracy; coarse batching for speed.

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
Implementation	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 • MC simulations

Project Achievements: AAD for Second-Order Greeks