A Practical Introduction to Adjoint Algorithmic Differentiation for Greeks

Martin Burke (with Claude)

Draft

Abstract

A pricing model returns one number. A risk manager needs hundreds: the sensitivity of that number to every spot, every point of every volatility surface, every tenor of every curve, every correlation. The oldest way to get them is to nudge each input in turn and reprice, which costs one revaluation per input. For an exotic options book priced by Monte Carlo, that arithmetic is the binding constraint on how often risk can be computed and how finely it can be decomposed. Adjoint algorithmic differentiation removes the constraint. It treats the pricer not as a formula to be re-derived but as a *program*—a graph of elementary operations—and applies the chain rule to that graph backwards, from the output towards the inputs. One backward sweep produces the *entire* gradient, and a theorem guarantees it costs no more than a small constant multiple of a single pricing, *independent of how many inputs there are*. This note develops that idea from the chain rule, for a reader comfortable with calculus and Monte Carlo but assuming no background in automatic differentiation or derivatives risk. We build a working adjoint engine, check it against closed-form Black–Scholes Greeks to machine precision, and measure its cost on a worst-of autocallable: the full gradient arrives in $4.2\times$ the time of one price, where central bumping would need $2n$. We then treat what makes exotics hard—discontinuous payoffs that defeat the pathwise derivative, the memory cost of the tape and how checkpointing buys it back, second-order risk by forward-over-reverse, and the calibration step, whose adjoint converts model sensitivities into the market sensitivities a desk can actually hedge with.

Contents

1 One price, hundreds of sensitivities

Start with the trade this note keeps returning to. A **worst-of autocallable** is a note sold over the counter to an investor who wants a coupon larger than a deposit pays and is willing to take equity risk to get it. It is written on a small basket of underlyings—three shares, say. Once a year, the issuer looks at the *worst* performer in the basket, measured against its level on the day the trade was struck. If even the worst one is at or above its starting level, the note redeems early: the investor gets their money back plus an accumulated coupon, and the trade is over. If not, the note runs another year. If it survives to maturity and the worst performer has fallen below some deeper level—sixty per cent of its start, say—the investor stops being a lender and becomes an owner: they receive that fallen performance rather than their principal.

That is a paragraph of English, and it prices in a few lines of Monte Carlo. The difficulty is not the price. The difficulty is everything the desk has to know *about* the price once the trade is on the books.

1.1 Counting the inputs

The trade above is exposed to the level of each of the three shares, so there are three spot sensitivities. It is exposed to the volatility of each share—but not to one number per share, because volatility is not one number. It is quoted as a *surface*, a grid of implied volatilities across option strikes and expiries, and the trade's exposure is different in each cell of that grid. A modest surface with five expiries and seven strikes is thirty-five numbers per share. Each share pays dividends, forecast out along a term structure: eight more numbers each. Discounting uses an interest-rate curve with a dozen tenors. And because the payoff depends on the *worst* of the three, it depends on how the three move together: three pairwise correlations.

Figure 1 adds these up. The total is 147 market inputs, for one trade. A desk does not hold one trade; it holds a book of them, sharing underlyings and surfaces, and the number of distinct inputs grows with the number of names in the book rather than the number of trades. Several hundred is routine; several thousand is not unusual.

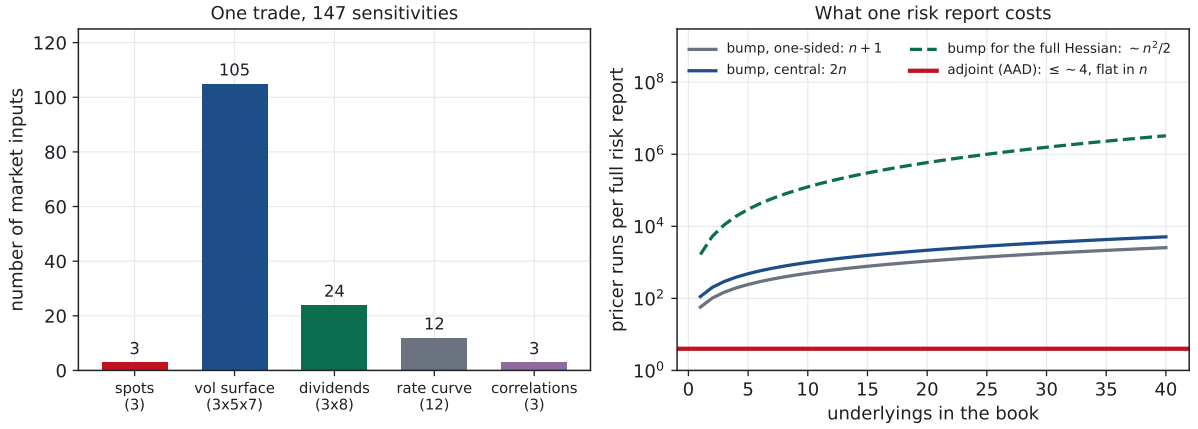

Figure 1: Left: where the market inputs of a single three-underlying autocallable come from. The volatility surfaces dominate. **Right:** how many times the pricer must run to produce one full risk report, as the book grows. Bumping scales with the number of inputs; the full second-order matrix scales with its square; the adjoint method is the flat line.

1.2 What a Greek is, and why there are so many

A **Greek** is a partial derivative of the trade's value with respect to one market input. The name comes from the letters conventionally attached to the first few. **Delta** is the derivative with respect to an underlying's price: if delta is 0.3, the position gains about $0.30 for a $1 rise.

It answers "how many shares do I buy to be insulated from a small move?" **Gamma** is the derivative of delta—how fast the hedge goes stale as the market moves. **Vega** is the derivative with respect to a volatility, and in practice it is not one number but a *bucketed* vector: the sensitivity to each cell of the surface separately, because a desk hedges volatility with actual options at actual strikes and expiries, and needs to know which ones. Sensitivity to interest rates is **rho**, likewise decomposed by tenor.

The reason the list is long is that these are the coordinates of a hedge. A single aggregate "volatility sensitivity" cannot be hedged, because there is no instrument that is exposed to all volatilities at once in the same proportions. To neutralise the risk you must know the exposure in the same coordinates as the instruments available to trade. That is why risk reports are long vectors rather than short summaries, and it is the entire source of the computational problem.

1.3 The bumping arithmetic

The direct way to get these numbers requires no theory. To find the sensitivity to input x_i, move it slightly, reprice, and divide:

$$\frac{\partial V}{\partial x_i} \approx \frac{V(x + he_i) - V(x)}{h}, \tag{1}$$

where e_i is the unit vector in the i-th coordinate. This is called **bumping**, or finite differencing. It works on any pricer, requires no access to its internals, and takes about an hour to implement.

It also costs one revaluation per input. With n inputs, a one-sided risk report is $n + 1$ pricings; the more accurate two-sided version in (3) below is $2n$. For the trade of Figure 1, that is 295 Monte Carlo runs to risk a single position. If each run takes ten seconds—modest for a path-dependent multi-asset payoff at production path counts—one risk report takes fifty minutes, and it is stale before it finishes.

The consequences are not just slowness. They shape what the desk is able to know. Risk gets computed overnight rather than continuously, so intraday hedging runs on yesterday's numbers. Sensitivities get aggregated into coarse buckets to keep the count down, discarding exactly the detail needed to choose hedge instruments. Second-order risk—the full matrix of cross-sensitivities, $O(n^2)$ revaluations—is quietly not computed at all, or is approximated so crudely that it misleads. And scenario analysis, which requires the whole apparatus re-run under stressed markets, becomes a weekly exercise.

1.4 What we are going to do instead

The observation that unlocks everything is that a pricer is a *program*. It is a finite sequence of elementary operations—additions, multiplications, exponentials, comparisons—applied to the inputs to produce the output. Calculus has a rule for differentiating a composition of functions, and a program is nothing but a composition of functions. So the derivative of the program can be computed by *another program*, obtained mechanically from the first.

There are two ways to apply the chain rule to a composition, corresponding to two ways of associating a product of matrices, and they have wildly different costs. Applied left to right, you get one input's sensitivities per pass: n passes for n inputs, no better than bumping. Applied right to left—starting at the output and working backwards—you get *all* the input sensitivities in a single pass. That is the adjoint method, and its cost does not depend on n at all.

The rest of this note develops that claim, tests it, and then deals with the things that make it harder in practice than in principle: Monte Carlo, discontinuous payoffs, memory, second derivatives, and calibration.

2 How Greeks are computed today

Before replacing bumping it is worth being precise about what is wrong with it, because two of its three problems are not obvious, and one of them—the interaction with Monte Carlo noise—is much the most serious in practice.

2.1 The bump size has no good answer

The approximation in (1) carries a truncation error. Taylor's theorem gives

$$\frac{V(x+h)-V(x)}{h} \;=\; V'(x) + \tfrac{h}{2}V''(x) + O(h^2), \tag{2}$$

so the error from one-sided bumping falls only in proportion to h. Using a symmetric bump cancels the second-order term:

$$\frac{V(x+h)-V(x-h)}{2h} \;=\; V'(x) + \tfrac{h^2}{6}V'''(x) + O(h^4), \tag{3}$$

which is why the **central difference** is preferred: its error falls like h^2. It costs two repricings per input rather than one, which is the trade that makes a risk report $2n$ pricings instead of $n+1$.

Both formulas suggest taking h as small as possible. Floating-point arithmetic disagrees. $V(x+h)$ and $V(x-h)$ are computed to a relative accuracy of about $\varepsilon \approx 10^{-16}$, so their difference carries an absolute error around $\varepsilon|V|$, and dividing by $2h$ inflates it to $\varepsilon|V|/2h$. As h shrinks, this **cancellation error** grows without bound: two nearly equal numbers are subtracted and the leading digits, which agree, annihilate, leaving a result dominated by the noise in the trailing digits.

The total error is therefore a sum of one term growing with h and another growing as h shrinks. Figure 2 shows the resulting U-shape, computed on a Black–Scholes call where the true Greeks are known in closed form. The best achievable accuracy for delta is a relative error of about 9×10^{-13}, at a bump of roughly 2×10^{-6} of spot. The adjoint value, on the same trade, is exact to the last bit—its relative error against the closed-form delta is zero, and against vega is 1.8×10^{-16}, one unit in the last place.

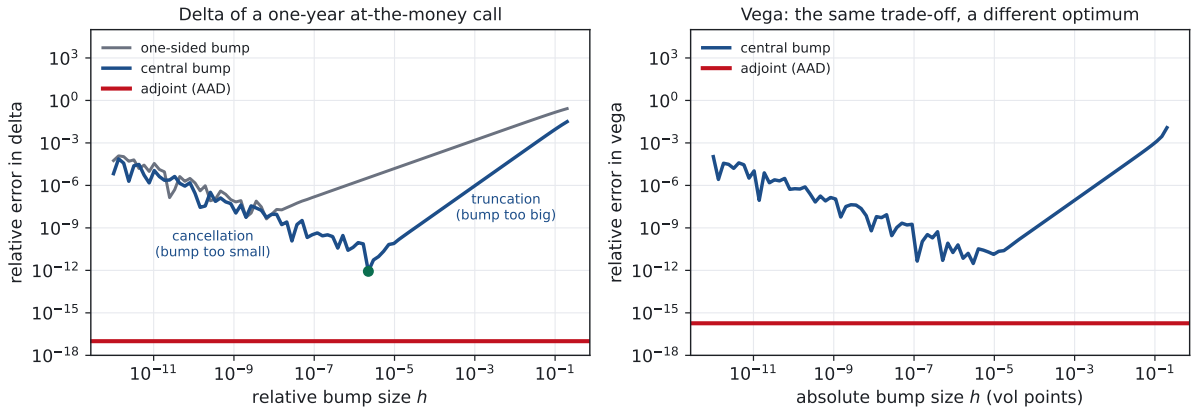

Figure 2: Measured relative error of bumped Greeks against the closed-form values, for a one-year at-the-money call. Truncation error dominates on the right, floating-point cancellation on the left, and the optimum sits in a narrow valley whose location differs between delta and vega—so no single bump size is right for a whole risk report. The adjoint answer (red) is exact to machine precision at every scale.

Notice that the optimal h is not the same for delta and for vega, and it would differ again for rho, and again for a different trade, and again for the same trade after the market moves. In

production this is handled by hard-coded bump conventions—one per cent of spot, one volatility point, one basis point—which are chosen for interpretability rather than accuracy and sit far to the right of the optimum in Figure 2. The resulting Greeks are typically accurate to three or four digits. That is often good enough. It is not always good enough, and one is rarely in a position to know which.

2.2 The serious problem: bumping a Monte Carlo price

An exotic is priced by simulation, so $V(x)$ is not a function but an *estimate*: an average over a finite sample of paths, carrying a standard error. Bumping subtracts two estimates and divides by a small number, which is precisely the operation that magnifies noise.

Write $\hat{V}(x)$ for the simulated price. If the two repricings use *independent* random draws, then

$$\operatorname{Var}\left[\frac{\hat{V}(x+h)-\hat{V}(x-h)}{2h}\right] \;=\; \frac{\operatorname{Var}[\hat{V}(x+h)]+\operatorname{Var}[\hat{V}(x-h)]}{4h^2}, \tag{4}$$

and the $1/h^2$ is fatal. Halving the bump doubles the noise in the Greek. With a bump of one per cent, the standard error of the delta is roughly a hundred times the standard error of the price.

The standard remedy is **common random numbers**: use the identical set of simulated draws for both repricings. The two estimates then become strongly positively correlated, and the variance of their difference collapses, because the covariance term in $\operatorname{Var}[A-B] = \operatorname{Var}[A]+\operatorname{Var}[B]-2\operatorname{Cov}[A,B]$ does most of the work. This is not a minor tuning detail. It is the difference between a usable number and an unusable one, and the left panel of Figure 6 measures the gap: bumping with independent draws is $32\times$ noisier than the adjoint estimate on the same number of paths. Bumping with common random numbers is, statistically, just as good as the adjoint. Its problem is not accuracy; it is that you must pay for it $2n$ times.

2.3 Pathwise and likelihood-ratio estimators

There is a third approach, older than automatic differentiation, which is worth understanding because the adjoint method turns out to be a mechanised version of one of them.

Write the simulated price as an average of a payoff over paths, $\hat{V} = \frac{1}{P}\sum_p g(x,\omega_p)$, where ω_p is the p-th draw of random numbers. The **pathwise** estimator differentiates *inside* the average:

$$\frac{\partial V}{\partial x} \;=\; \mathbb{E}\left[\frac{\partial g}{\partial x}(x,\omega)\right] \;\approx\; \frac{1}{P}\sum_{p=1}^{P}\frac{\partial g}{\partial x}(x,\omega_p). \tag{5}$$

Each path's payoff is differentiated with respect to the input, holding the random draw fixed, and the derivatives are averaged. Interchanging differentiation and expectation like this is legitimate when g is Lipschitz continuous in x—which fails for exactly the discontinuous payoffs exotics are built from, a problem we take up in §7.

The **likelihood-ratio** estimator moves the derivative onto the probability density instead of the payoff:

$$\frac{\partial V}{\partial x} \;=\; \mathbb{E}\left[g(\omega)\,\frac{\partial \log p_x(\omega)}{\partial x}\right], \tag{6}$$

where p_x is the density of the simulated paths. Because the payoff is never differentiated, this works for any payoff, discontinuous or not. The price is variance: the score $\partial \log p_x/\partial x$ has mean zero and typically large spread, so the estimator is noisy. In Figure 6 it is about $8\times$ noisier than the pathwise adjoint. It remains valuable as an unbiased reference against which smoothed pathwise estimates can be checked, and we use it that way in §7.

The pathwise estimator is the one that generalises. Computing $\partial g/\partial x$ by hand for a complicated payoff and a multi-factor model is an error-prone exercise that must be redone

whenever the payoff changes. Adjoint algorithmic differentiation is, in essence, a way of getting $\partial g/\partial x$ for *every* x at once, mechanically, from the code that computes g.

3 A pricer is a graph

The shift in viewpoint that makes everything work is to stop looking at the pricing *formula* and start looking at the pricing *program*.

Consider a small piece of a pricer: the Black–Scholes d_1 term, evaluated on inputs S, K, σ, T. Written as mathematics it is a single expression. Executed by a computer it is a sequence of operations, each with at most two arguments, each storing its result in a temporary:

$$\begin{aligned} v_1 &= S/K, & v_4 &= \sigma^2, \\ v_2 &= \log v_1, & v_5 &= \tfrac{1}{2} v_4 T, \\ v_3 &= \sigma\sqrt{T}, & v_6 &= (v_2 + v_5)/v_3. \end{aligned}$$

This sequence is a **computational graph**: a directed acyclic graph whose nodes are intermediate values and whose edges record which values feed which operations. Every pricer, however complicated—a Monte Carlo loop, a PDE solver, a tree, a calibration routine—is such a graph when unrolled. Loops become long chains of repeated structure. Branches select which edges exist for the particular inputs at hand.

The graph matters because the chain rule is a statement about graphs. If y depends on u, which depends on x, then

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\,\frac{\partial u}{\partial x}, \tag{7}$$

and when several paths through the graph connect x to y, their contributions add. Every operation in the graph is elementary—we know the derivative of multiplication, of log, of $\sqrt{\cdot}$—so all the local pieces are available. What remains is a choice about the *order* in which to multiply them together.

3.1 Two directions, two costs

Suppose the pricer is a chain of m stages, $x \to v_1 \to v_2 \to \cdots \to y$. The chain rule gives the derivative as a product of Jacobian matrices,

$$\frac{\partial y}{\partial x} = \underbrace{\frac{\partial y}{\partial v_m}}_{1\times m_m}\ \underbrace{\frac{\partial v_m}{\partial v_{m-1}}}_{m_m\times m_{m-1}}\ \cdots\ \underbrace{\frac{\partial v_1}{\partial x}}_{m_1\times n}, \tag{8}$$

and matrix multiplication is associative: the answer is the same however we bracket it, but the cost is not.

Bracket from the *right* and you repeatedly multiply a matrix by an $m_1 \times n$ matrix—you are propagating n columns forward, one per input. This is **forward mode**, and its cost is proportional to n.

Bracket from the *left* and you repeatedly multiply a *row vector* by a matrix. There is only ever one row, because there is only one output. This is **reverse mode**, or the **adjoint** method, and its cost is proportional to the number of outputs—which, for a pricer, is one.

That asymmetry is the whole of the subject. A pricer has many inputs and one output, so reverse mode is the right bracketing, and the saving is a factor of n.

4 Forward mode: one input at a time

Forward mode is worth constructing first, because it is the intuitive one and it makes clear what reverse mode is avoiding.

Fix an input direction—say we want sensitivities to S. Attach to every intermediate v a second number, its **tangent**

$$\dot{v} \;=\; \frac{\partial v}{\partial S}, \tag{9}$$

the derivative of that intermediate with respect to the chosen input. Seed the inputs: $\dot{S} = 1$ and $\dot{\sigma} = \dot{r} = \cdots = 0$. Then push tangents forward alongside values, using the differentiation rule for each operation as it executes:

$$\begin{aligned} w = u + v \quad &\Longrightarrow \quad \dot{w} = \dot{u} + \dot{v}, \\ w = uv \quad &\Longrightarrow \quad \dot{w} = u\dot{v} + v\dot{u}, \\ w = \exp(u) \quad &\Longrightarrow \quad \dot{w} = \exp(u)\,\dot{u}, \\ w = \log(u) \quad &\Longrightarrow \quad \dot{w} = \dot{u}/u. \end{aligned} \tag{10}$$

These are the ordinary rules of calculus, applied one operation at a time. When the program finishes, $\dot{y}$ is $\partial V/\partial S$—exactly, with no bump size and no cancellation.

The implementation is elegant: replace every real number in the pricer with a pair $(v, \dot{v})$ and overload the arithmetic. These pairs are **dual numbers**, and they behave like $a + b\epsilon$ with $\epsilon^2 = 0$: the b component tracks exactly the tangent rules of (10). The engine used for this note's measurements implements them as a layer beneath the reverse-mode core of Appendix C, which is what makes the second-order construction of §8 possible.

The catch is in the seeding. One run gives sensitivities to one input, because the seed picks out one direction. To get n Greeks you run the augmented program n times. Forward mode is exact where bumping is approximate—a real improvement—but it does nothing about the cost, which remains proportional to n.

Forward mode is the right tool when the shape is reversed: few inputs, many outputs. Computing the sensitivity of an entire yield curve to one parameter is a forward-mode problem. Pricing is not.

5 Reverse mode: the whole gradient in one sweep

Reverse mode inverts the question. Instead of asking "how does everything respond to a change in this one input?", it asks "how does this one output respond to a change in everything?"

Attach to every intermediate v its **adjoint**

$$\bar{v} \;=\; \frac{\partial y}{\partial v}, \tag{11}$$

the sensitivity of the *final output* to that intermediate. Note the reversal: the tangent $\dot{v}$ has the chosen input in the denominator, the adjoint $\bar{v}$ has the output in the numerator. A tangent is a column of the Jacobian; an adjoint is a row.

Adjoints cannot be computed as the program runs forward, because at the moment v is created we do not yet know what the output will be, let alone how it depends on v. They must be computed *backwards*, starting from the end.

5.1 The adjoint recurrence

At the output itself the answer is immediate: $\bar{y} = \partial y/\partial y = 1$. This is the **seed**.

Now work backwards. Suppose the program contained the operation $w = f(u, v)$. The output y depends on u only through w (and through any other operations u fed). The chain

rule says that u's contribution through this particular edge is $\bar{w}\,\partial w/\partial u$. Summing over every operation u fed gives the rule that defines the method:

$$\bar{u} \;+\!=\; \bar{w}\,\frac{\partial w}{\partial u} \qquad \text{for every edge } u \to w. \tag{12}$$

The accumulation (+= rather than =) is essential: an intermediate used in three places receives three contributions, and the total sensitivity is their sum. This is where the "several paths through the graph add" clause of the chain rule is discharged.

The rule must be applied in *reverse topological order*—each node is processed only once every operation that consumed it has already contributed. Since the operations were recorded in execution order, reverse order is simply the recorded list read backwards. Table 1 collects the local partials for the common operations.

Operation	**Forward (tangent)**	**Reverse (adjoint)**
$w = u + v$	$\dot{w} = \dot{u} + \dot{v}$	$\bar{u} \mathrel{+}= \bar{w}; \quad \bar{v} \mathrel{+}= \bar{w}$
$w = u - v$	$\dot{w} = \dot{u} - \dot{v}$	$\bar{u} \mathrel{+}= \bar{w}; \quad \bar{v} \mathrel{-}= \bar{w}$
$w = uv$	$\dot{w} = v\dot{u} + u\dot{v}$	$\bar{u} \mathrel{+}= v\bar{w}; \quad \bar{v} \mathrel{+}= u\bar{w}$
$w = u/v$	$\dot{w} = (\dot{u} - w\dot{v})/v$	$\bar{u} \mathrel{+}= \bar{w}/v; \quad \bar{v} \mathrel{-}= w\bar{w}/v$
$w = \exp u$	$\dot{w} = w\dot{u}$	$\bar{u} \mathrel{+}= w\,\bar{w}$
$w = \log u$	$\dot{w} = \dot{u}/u$	$\bar{u} \mathrel{+}= \bar{w}/u$
$w = \sqrt{u}$	$\dot{w} = \dot{u}/(2w)$	$\bar{u} \mathrel{+}= \bar{w}/(2w)$
$w = \Phi(u)$	$\dot{w} = \phi(u)\dot{u}$	$\bar{u} \mathrel{+}= \phi(u)\,\bar{w}$

Table 1: Local differentiation rules. Each reverse rule is the transpose of the corresponding forward rule—the same local partials, applied in the opposite direction. Φ and ϕ are the standard normal distribution and density.

5.2 A worked example by hand

Take a deliberately tiny function with a shared intermediate, so the accumulation rule is visible:

$$y \;=\; (\log x_1)\cdot x_2 + \sqrt{\log x_1}, \qquad x_1 = e, \quad x_2 = 3. \tag{13}$$

The forward sweep records four operations:

$$v_1 = \log x_1 = 1, \qquad v_2 = v_1 x_2 = 3, \qquad v_3 = \sqrt{v_1} = 1, \qquad y = v_2 + v_3 = 4.$$

Now sweep backwards. Seed $\bar{y} = 1$. The last operation was $y = v_2 + v_3$, so by the addition rule $\bar{v_2} = 1$ and $\bar{v_3} = 1$. The operation before it was $v_3 = \sqrt{v_1}$, contributing $\bar{v_1} \mathrel{+}= \bar{v_3}/(2v_3) = 1/2$. Before that, $v_2 = v_1 x_2$ contributes $\bar{v_1} \mathrel{+}= x_2\bar{v_2} = 3$ and $\bar{x_2} \mathrel{+}= v_1\bar{v_2} = 1$. The intermediate v_1 has now received both of its contributions and totals $\bar{v_1} = 3.5$. Finally $v_1 = \log x_1$ gives $\bar{x_1} = \bar{v_1}/x_1 = 3.5/e \approx 1.288$.

Both derivatives dropped out of one backward pass. Differentiating (13) by hand confirms them: $\partial y/\partial x_2 = \log x_1 = 1$ and $\partial y/\partial x_1 = (x_2 + 1/(2\sqrt{\log x_1}))/x_1 = 3.5/e$. Figure 3 shows the same calculation on the graph.

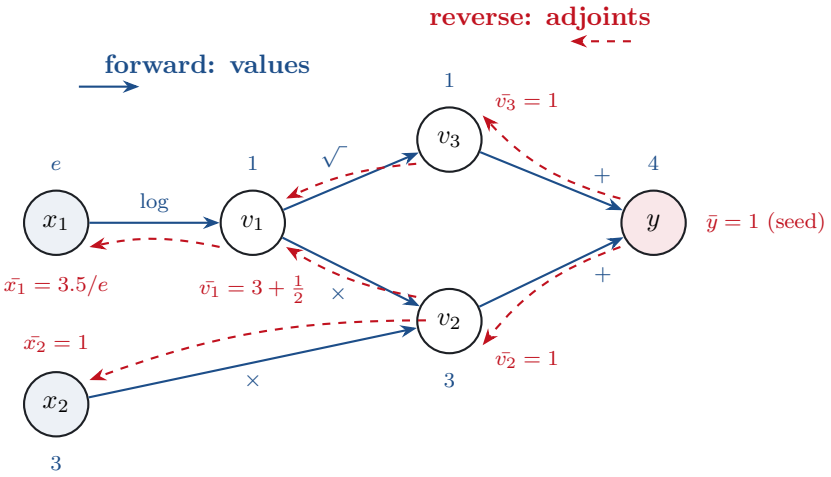

Figure 3: The computational graph of (13). Solid blue edges carry values forward; dashed red edges carry adjoints backward. The node v_1 feeds two operations, so it receives two adjoint contributions, 3 and $\frac{1}{2}$, which *add*. One backward pass produces both input derivatives.

5.3 The algorithm

Written out, the method has two phases.

Adjoint algorithmic differentiation of a scalar-valued program

Forward sweep ▷ *execute and record*

for each operation $v_k = f_k(v_i, v_j)$ in execution order:
evaluate v_k;
record on the *tape*: the operation's parents i, j and the local partials $\partial v_k/\partial v_i$, $\partial v_k/\partial v_j$.

Reverse sweep ▷ *propagate sensitivities*

set $\bar{v} \leftarrow 0$ for all v; set $\bar{y} \leftarrow 1$;
for each recorded operation v_k, in *reverse* order:
$\bar{v}_i \mathrel{+}= \bar{v}_k \cdot \partial v_k/\partial v_i$;
$\bar{v}_j \mathrel{+}= \bar{v}_k \cdot \partial v_k/\partial v_j$.

Result: $\bar{x}_i = \partial y/\partial x_i$ for every input, from one forward and one reverse pass.

Algorithm 1: The two sweeps. Everything else in this note is engineering around these eight lines.

5.4 The cost theorem

The reverse sweep visits each recorded operation exactly once and does a bounded amount of arithmetic there—one multiply and one add per parent. The forward sweep does the original work plus the recording. So the total cost of obtaining the *entire* gradient is a constant multiple of the cost of evaluating the function, and—this is the point—the constant does not involve n:

$$\operatorname{cost}\big(V(x),\, \nabla V(x)\big) \;\le\; \omega \cdot \operatorname{cost}\big(V(x)\big), \qquad \omega \approx 3\text{–}4, \tag{14}$$

independently of the dimension n. This is the **cheap gradient principle**, and the bound $\omega \le 4$ under a standard accounting of operations is due to Baur and Strassen [1]; see Griewank and Walther [2] for the modern treatment and Appendix B for the argument.

It is worth pausing on how strong this is. Computing one Greek by bumping costs two pricings. Computing a hundred and forty-seven Greeks by bumping costs two hundred and ninety-five. Computing all one hundred and forty-seven by the adjoint method costs about four. The saving is not a constant factor; it grows without limit as the book grows.

5.5 Measuring it

Theorems about operation counts do not always survive contact with an implementation, so Figure 4 measures the real thing. The pricer is the worst-of autocallable of §1, run on 8,000 paths, with the number of underlyings increased to sweep the input count from 9 to 164. The baseline is the same pricer written in plain array code with no differentiation machinery at all.

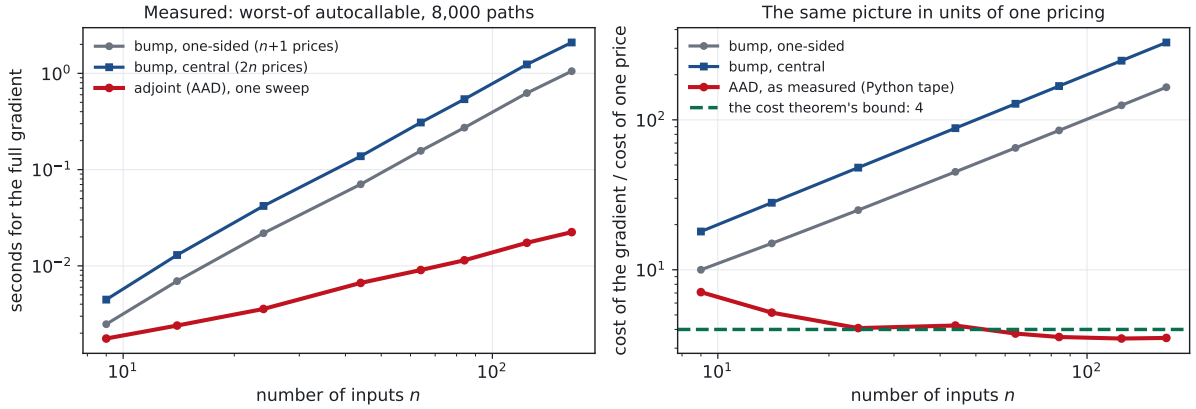

Figure 4: Measured cost of a full gradient. **Left:** wall-clock seconds. **Right:** the same data in units of one undifferentiated pricing. Bumping rises with n; the adjoint sweep is flat, and settles at 4.2× one pricing—inside the bound of (14). Timings come from the machine that built this note; the shape and the ratio are what matter.

The measured ratio is 4.2, essentially the theoretical constant. Since central bumping costs $2n$ pricings, the adjoint method is already cheaper at $n = 3$ inputs, and by $n = 164$ it is faster by a factor of about eighty. The left panel shows the two lines diverging exactly as predicted: one with slope one on a log-log plot, the other flat.

Two honest caveats. First, this engine is written in Python with operator overloading, and it achieves a good constant only because the pricer is vectorised across paths, so the interpreter overhead is amortised over 8,000 payoffs per operation. A scalar-per-path implementation in Python would be far worse. Second, the constant in production C++ adjoint code is typically in the same 2–5 range, so the picture transfers, but the absolute times do not: both bars would shrink by orders of magnitude together.

6 The tape, and what it costs

The reverse sweep needs the local partials, and the local partials depend on the intermediate *values*. Since the sweep runs backwards while the values were produced forwards, they must be stored. That store is the **tape**, and it is the price of the method.

6.1 What is on it

For each elementary operation the tape holds the operation's parents and enough information to reconstruct its local partials. For $w = uv$ that means retaining u and v; for $w = \exp u$ it is enough to retain w itself. The tape is therefore proportional in length to the number of operations *executed*— which, for a Monte Carlo pricer, means proportional to the number of time steps, and independent of the number of inputs.

That independence is worth stating plainly, because it is the mirror image of the cost result. Adding a hundred more Greeks to the risk report costs nothing in tape. Adding a hundred more time steps to the simulation costs a hundred more steps' worth.

The left panel of Figure 5 measures it. Each simulation step of the worst-of autocallable adds 38 tape nodes. At 20,000 paths, with values stored per path, that is about 6 MB per step. Sixty-four steps fill 390 MB. Extrapolating to a five-year trade with daily steps—1,260 steps, entirely ordinary for a barrier-monitored payoff—gives roughly 7.7 GB of tape for a single pricing run. That is the real constraint on naive AAD, and it is a memory constraint, not a speed one.

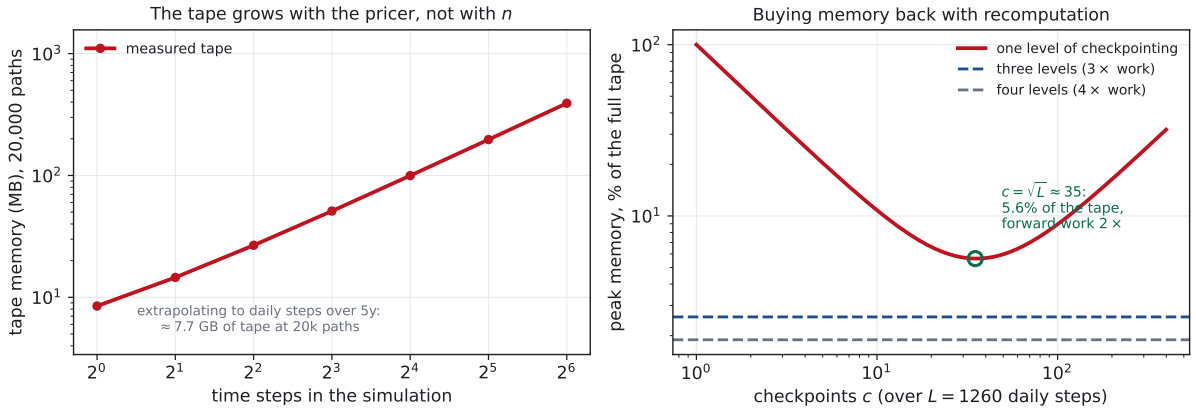

Figure 5: Left: measured tape growth for the autocallable pricer. Memory scales with simulation steps, not with the number of Greeks. **Right:** checkpointing. Storing c intermediate states and recomputing each segment on demand gives peak memory proportional to $L/c + c$, minimised at $c = \sqrt{L}$: for a 1,260-step simulation that is 5.6% of the full tape for twice the forward work. Nesting the scheme reduces it further.

6.2 Checkpointing

The way out is to trade memory for arithmetic. Rather than taping the whole simulation, store the model *state* at c checkpoints along the path and tape only one segment at a time. To sweep backwards, restore the last checkpoint, re-run forward through that segment while taping, sweep back through it, discard the tape, and move to the previous checkpoint.

Peak memory is then the c checkpoint states plus one segment's tape, proportional to $c + L/c$ for L steps. This is minimised at $c = \sqrt{L}$, giving memory $O(\sqrt{L})$ instead of $O(L)$—for $L = 1260$, a reduction to 5.6% of the full tape—at the cost of running the forward simulation twice. Applying the idea recursively, with checkpoints between checkpoints, gives memory $O(d\,L^{1/d})$ for d levels and d forward passes: three levels bring the 1,260-step tape to 2.6% of full. The optimal schedules are a solved problem, and the standard reference implementation is Griewank's `revolve` algorithm [3].

6.3 Two ways to build it

Operator overloading is what this note's engine does and what most adoptions start with. Replace the pricer's floating-point type with a custom class whose arithmetic operators, besides computing the value, append a record to a global tape. The pricing code itself is untouched apart from its type declarations. It is quick to adopt, handles arbitrary control flow naturally, and carries a runtime cost for the indirection and the tape allocation.

Source transformation takes the pricer's source code and emits new source code for the adjoint. A compiler-like tool analyses the program and generates the reverse sweep statically. The generated code is fast—no tape records for values the compiler can prove are recoverable, no virtual dispatch— but the tools are demanding, and code with heavy branching or external library calls can defeat them. Tapenade [12] is the long-standing example.

In practice large derivatives libraries use overloading with heavy optimisation: expression templates to fuse arithmetic before it reaches the tape, custom allocators, and hand-written

adjoints for hot inner kernels. CoDiPack [13] and ADOL-C [14] are representative. The machine-learning frameworks solved the same problem for a different workload; JAX and PyTorch are reverse-mode AD engines with a numerical-computing front end, and their `grad` is Algorithm 1.

7 Adjoints through Monte Carlo

Everything so far applies to any program. Monte Carlo pricing adds one structural feature that turns out to be extremely convenient, and one that is genuinely difficult.

7.1 The convenient part: the average is linear

The simulated price is an average over paths, $\hat{V} = \frac{1}{P}\sum_p g(x, \omega_p)$, and differentiation commutes with a finite sum. So

$$\frac{\partial \hat{V}}{\partial x_i} \;=\; \frac{1}{P}\sum_{p=1}^{P} \frac{\partial g(x,\omega_p)}{\partial x_i}, \tag{15}$$

which says the adjoint of the averaged price is the average of the per-path adjoints. Operationally this means the tape does not need to hold all P paths' graphs. The seed $\bar{\hat{V}} = 1$ enters the average node, which distributes $1/P$ to each path, and each path's adjoint sweep is independent of every other. Paths can be run in blocks, or in parallel, with adjoints accumulated into a shared gradient vector as they finish—the memory-efficient structure that Giles and Glasserman set out in the paper that introduced the technique to finance [4].

When the pricer is vectorised across paths, as the engine here is, the same structure appears differently: each tape node holds an array of P values, and the sweep does array arithmetic. The tape length is then the number of operations in the pricer, not the number of operations times paths—which is why 271 nodes suffice for a three-asset, three-year autocallable at any path count.

Because the adjoint estimator is exactly the pathwise estimator of (5) computed mechanically, it inherits the pathwise estimator's statistical properties. That is visible in Figure 6: the adjoint and the common-random-numbers bump agree to three significant figures in standard error, run after run. They are estimating the same quantity the same way. The adjoint just does not charge $2n$ pricings for it.

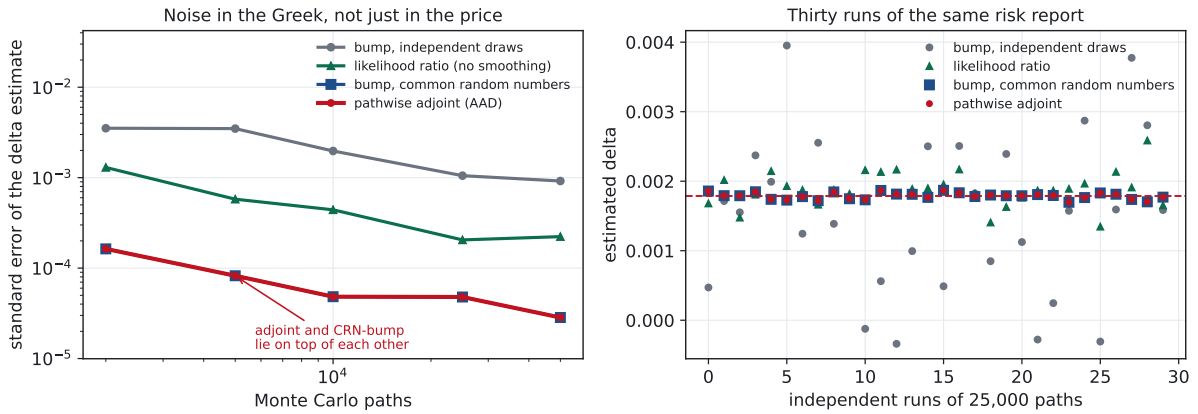

Figure 6: Delta of the autocallable, estimated 24 ways at each path count. Bumping with independent draws is 32× noisier than the pathwise adjoint; the likelihood-ratio estimator, which needs no smoothing, is about 8× noisier. Bumping with common random numbers matches the adjoint's precision exactly—and costs $2n$ pricings to do it.

7.2 The difficult part: payoffs with jumps

The interchange of derivative and expectation in (5) needs the payoff to be continuous in the input along each path. Exotics are built from features that violate this deliberately. A digital pays a fixed amount if a level is crossed and nothing if it is not. A barrier extinguishes a claim outright. The autocallable of this note does both: the early-redemption test is a step, and the knock-in at maturity is a step.

For such a payoff, moving S infinitesimally leaves almost every path's payoff *exactly* unchanged—the path either triggers or it does not, and a sufficiently small nudge does not change which. The pathwise derivative is therefore zero almost everywhere, and averaging zeros gives zero. But the price plainly does depend on S: nudging S changes the *probability* of triggering, and probability is what the average is measuring. The derivative of the expectation exists and is non-zero; the expectation of the pathwise derivative is zero. They are not equal, and the pathwise estimator is simply wrong here.

This is not a defect of AAD. Any pathwise method inherits it, and a differentiation engine will faithfully return the wrong answer, because the program it was given genuinely has zero derivative. The fix must come from the modelling, not the tooling. Three approaches are used.

Smoothing. Replace the step by a steep but continuous approximation—a logistic of half-width ϵ, or a narrow call spread—so the payoff becomes differentiable and the pathwise estimator becomes valid. This introduces a bias that vanishes with ϵ, while the variance grows as ϵ shrinks, because fewer and fewer paths land in the narrow band where the derivative is non-negligible. The result is another U-shaped trade-off, shown in Figure 7, and measured here against a likelihood-ratio reference which needs no smoothing and is therefore unbiased. The optimum for this trade at 60,000 paths sits at $\epsilon \approx 0.026$ in units of performance—about two and a half percentage points either side of the barrier. It is worth noting that call-spread smoothing is not purely a numerical convenience: a trading desk cannot hedge a true digital either, and will risk-manage it as a spread, so the smoothed payoff is arguably the more honest representation of what is actually being run.

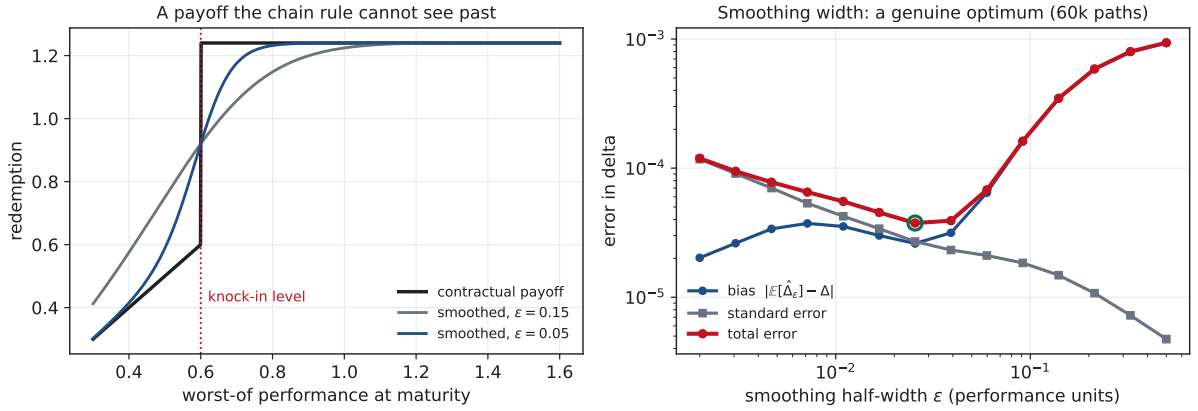

Figure 7: Left: the autocallable's maturity payoff as a function of the worst performer, with two smoothing widths. **Right:** measured bias and standard error of the pathwise adjoint delta against the smoothing width, with the unbiased likelihood-ratio value as reference. Both tails are bad: too much smoothing prices the wrong payoff, too little leaves too few paths carrying the derivative.

Hybrid estimators. Use the pathwise adjoint for the smooth part of the payoff and the likelihood-ratio estimator for the discontinuous part, splitting the payoff into the two pieces. This is unbiased and needs no smoothing parameter, at the cost of more implementation and the likelihood-ratio term's variance. The systematic version of this idea is the vibrato technique [6], which conditions on the final step and differentiates the resulting smooth conditional expectation.

Conditional expectation. Where a closed form exists for the payoff conditional on the path

up to the last step—often it does for barriers and digitals under a Gaussian step—substitute it. The conditioned payoff is smooth, and the pathwise adjoint then applies without approximation.

7.3 Early exercise

Callable and Bermudan-style products are priced by regression methods such as Longstaff–Schwartz [11], which estimate a continuation value from simulated paths and compare it to the immediate exercise value. The exercise decision is a step function of that comparison, so the same discontinuity appears.

There is a helpful piece of structure here. At the optimal exercise boundary the two values are equal by construction, so the payoff is continuous across the decision even though the *policy* jumps. The standard result is that the derivative of the value with respect to the exercise boundary vanishes at the optimum—an envelope theorem—so one may hold the exercise decision fixed while differentiating and still obtain the correct sensitivity to first order. In practice this means the exercise indicator is treated as a constant on the tape, and the regression coefficients are usually frozen too. The approximation is good where the boundary is well estimated and degrades where it is not.

8 Second-order risk

Gamma—the derivative of delta—is not optional. It governs how quickly a delta hedge decays, and it is what the desk's rebalancing cost depends on. For a multi-asset trade the object of interest is the whole matrix of second derivatives,

$$H_{ij} \;=\; \frac{\partial^2 V}{\partial x_i \partial x_j}, \tag{16}$$

whose diagonal in the spot block holds the gammas, whose off-diagonal spot entries hold the cross-gammas—how asset i's delta responds to asset j's price—and whose spot-versus-volatility block holds the vannas.

By bumping, this costs $O(n^2)$ pricings: about $n(n+3)/2$ for the symmetric matrix by central differences. At $n = 147$ that is eleven thousand pricings, which is why in a bumping shop the full second-order matrix is usually not computed.

8.1 Forward over reverse

The trick is to compose the two modes. Run the reverse sweep, but with dual numbers instead of ordinary floating point as the underlying arithmetic. Every value and every adjoint then carries a tangent alongside it.

Concretely: seed the forward direction at input j by setting $\dot{x}_j = 1$ and all other input tangents to zero, then run Algorithm 1 in dual arithmetic. The value part of the result is the gradient $\partial V/\partial x_i$ as before. The tangent part is

$$\frac{\partial}{\partial x_j}\left(\frac{\partial V}{\partial x_i}\right) \;=\; H_{ij}, \tag{17}$$

the j-th column of the Hessian. This is a **Hessian-vector product**, and it costs a small constant times one pricing—roughly $4\times$, the same constant as before, because dual arithmetic is a constant-factor overhead on each operation.

Repeating for $j = 1, \dots, n$ gives the full matrix in $O(n)$ pricing-equivalents rather than $O(n^2)$. Figure 8 shows the result for the three-asset autocallable: a 19×19 matrix in under a third of a second. The strongest available check on the implementation is that the matrix comes

out symmetric to 1.8×10^{-16}—machine precision—even though the two triangles are computed by entirely separate passes with different seeds.

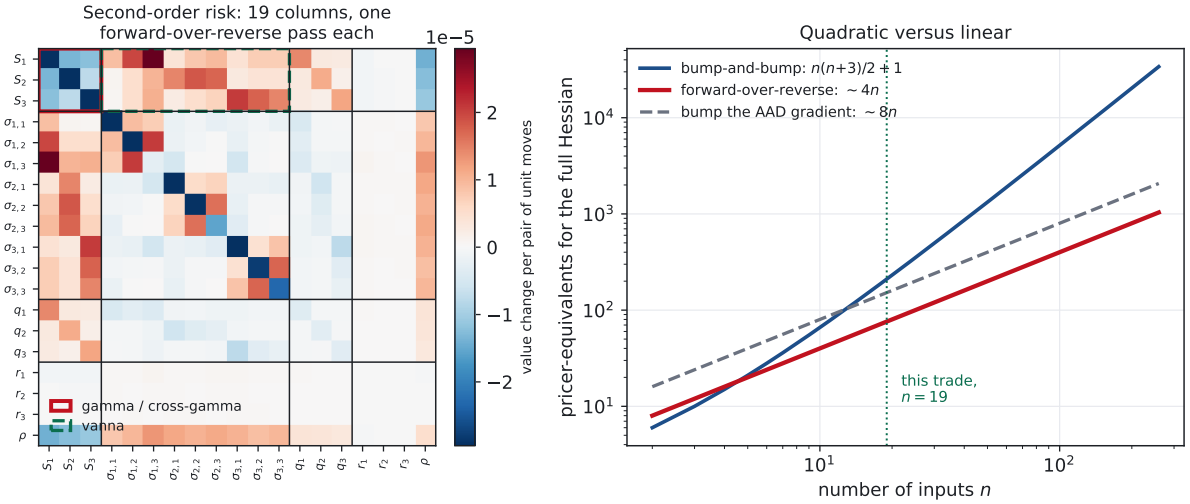

Figure 8: Left: the measured second-order risk matrix of the three-asset autocallable, rescaled so each axis is one unit of the move a desk quotes against (1% of spot, one volatility point, one basis point). Block boundaries separate spots, volatilities, dividends, rates and correlation. **Right:** cost. Bumping is quadratic in n; forward-over-reverse is linear.

Two practical notes. If only a few columns are wanted—the spot block, say, which is what gamma hedging needs—compute only those columns, at 4× one pricing each; the full matrix is rarely the requirement. And if an AAD gradient already exists, bumping *it* gives the Hessian at about $8n$ pricing-equivalents with almost no new code: worse than forward-over-reverse by a factor of two, and it reintroduces a bump size, but it is a legitimate first step.

9 Differentiating through the calibration

There is a gap between what the adjoint sweep produces and what a desk can hedge with, and closing it is the last essential piece.

9.1 Model Greeks are not market Greeks

The pricer takes model parameters as inputs—in our case, a piecewise-constant volatility $\sigma_1, \sigma_2, \sigma_3$ over the three annual periods. Running the adjoint gives $\partial V/\partial \sigma_k$: the sensitivity to a model parameter. But σ_k is not tradable. Nobody quotes it, and no hedge can be put on in it.

What is tradable is the set of vanilla options the model was calibrated to. The number the desk needs is $\partial V/\partial m_k$, the sensitivity of the exotic to the *market price* of the k-th hedging vanilla—because that says how many of those vanillas to buy.

These are different numbers, and not by a small factor. The model parameters were chosen to fit the market quotes, so moving one quote forces a re-solve that moves several parameters at once. A change in the two-year quote changes σ_2 directly and σ_3 in compensation, since the three-year quote must continue to be matched. The exotic feels all of it.

The naive route is brute force: bump a market quote, re-run the whole calibration, re-price the exotic. That costs one full calibration plus one full pricing per quote, and calibrations are themselves expensive iterative solves. For a surface with a hundred quotes it is prohibitive.

9.2 The implicit function theorem does it in one solve

Calibration solves a system. Write the calibrated parameters as θ (the vector of σ_k), the market quotes as m, and the calibration conditions as

$$g(\theta, m) \;=\; 0, \tag{18}$$

where g_k is the difference between the model's price of the k-th instrument and its market quote. The solution defines θ as an implicit function $\theta(m)$. Differentiating (18) totally with respect to m,

$$\frac{\partial g}{\partial \theta}\,\frac{\mathrm{d}\theta}{\mathrm{d}m} + \frac{\partial g}{\partial m} \;=\; 0 \qquad \Longrightarrow \qquad \frac{\mathrm{d}\theta}{\mathrm{d}m} \;=\; -\left(\frac{\partial g}{\partial \theta}\right)^{-1}\frac{\partial g}{\partial m}. \tag{19}$$

This is the implicit function theorem, and it says something important: the sensitivity of the calibrated parameters to the quotes depends only on the *Jacobian at the solution*, not on the path the solver took to get there. The iterations are irrelevant. One does not differentiate through the Newton loop—one differentiates the equation the loop was solving.

Chaining with the exotic's model Greeks gives the market Greeks:

$$\frac{\mathrm{d}V}{\mathrm{d}m} \;=\; \left(\frac{\mathrm{d}\theta}{\mathrm{d}m}\right)^{\top}\nabla_\theta V. \tag{20}$$

And there is a better way to evaluate it than forming $\mathrm{d}\theta/\mathrm{d}m$ at all. Substituting (19) and grouping from the left gives the **adjoint form**: solve the single transposed linear system

$$\left(\frac{\partial g}{\partial \theta}\right)^{\top}\lambda \;=\; \nabla_\theta V, \qquad \text{then} \qquad \frac{\mathrm{d}V}{\mathrm{d}m} \;=\; -\left(\frac{\partial g}{\partial m}\right)^{\top}\lambda. \tag{21}$$

One linear solve, of the size of the parameter vector, converts model Greeks into market Greeks—for *any* exotic, reusing the same factorised Jacobian across the whole book. The same right-to-left grouping that made reverse mode cheap makes this cheap, and for the same reason.

9.3 Measuring it

Figure 9 carries this out. The model is calibrated to three market vanillas by Newton's method, with the Jacobian $\partial g/\partial\theta$ supplied by the adjoint engine itself. The exotic's model vegas come from one reverse sweep. The market vegas come from one triangular solve.

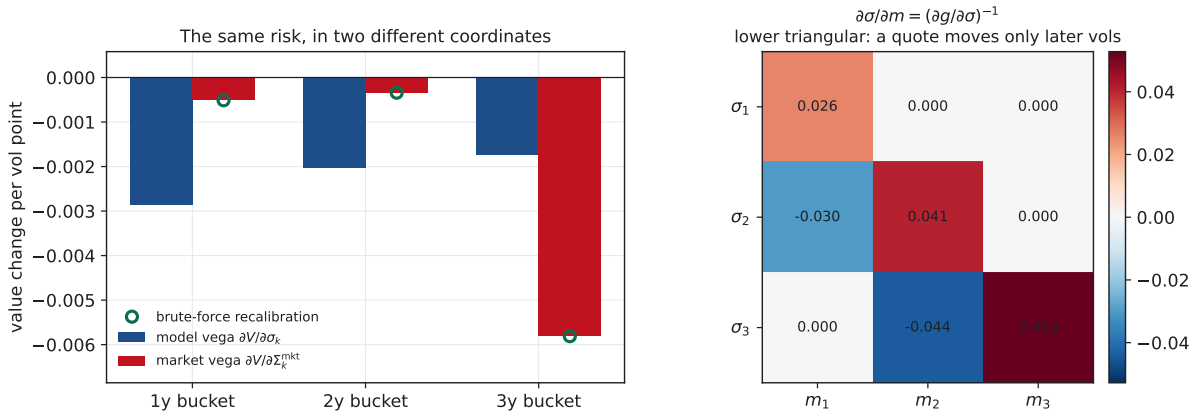

Figure 9: Left: the same risk in two coordinate systems. Model vega is the sensitivity to an untradable parameter; market vega is the sensitivity to the quoted hedge instrument. Circles are brute-force recalibration, agreeing with the adjoint result to 4×10^{-13}. **Right:** the transformation matrix $\mathrm{d}\theta/\mathrm{d}m$. It is lower triangular because a quote at one maturity constrains the volatility over that period and every later one.

The redistribution is substantial. The three-year model vega is the smallest of the three, but the three-year *market* vega is by far the largest—moving the three-year quote forces a large compensating move in σ_3, because σ_1 and σ_2 are pinned by the shorter quotes. A desk hedging on model vegas would put on materially the wrong trade.

The check that matters is the open circles: brute-force recalibration, bumping each market quote and re-solving, agrees with the one-solve adjoint result to thirteen decimal places. Same answer, at a small fraction of the cost.

The example here is deliberately small so it can be verified exactly. Nothing in the argument depends on that. When the calibration is an over-determined least-squares fit of a stochastic-volatility model to a hundred quotes, $\partial g/\partial\theta$ becomes the Gauss–Newton normal matrix and the solve becomes its factorisation—but (21) is unchanged, and the factorisation is already computed by the calibrator.

10 Practical pitfalls

The theory is short. The engineering is where AAD projects succeed or fail.

Non-differentiable kernels. max, min, absolute value, floors and caps are differentiable except on a set of measure zero, and an engine handles them by picking a branch. That is correct almost everywhere and is fine for a smooth expectation. But it means the sensitivity is silently a one-sided derivative at the kink, and it means a payoff built entirely from steps returns zero rather than an error. Worth knowing where the kinks are.

There is a diagnostic in this. Comparing the adjoint gradient of the worst-of autocallable against central bumping shows a gap that shrinks *linearly* with the bump size, not quadratically as (3) promises. That is the signature of the min over the three assets: at each bump, a handful of paths switch which asset is the worst, and the kink degrades the convergence. The adjoint value is the exact limit; the bumped value is converging to it from outside.

Discrete inputs and lookups. Interpolation on a volatility surface, day count conventions, calendar logic—anything that indexes into a table—has zero derivative with respect to the index. If the interpolation weights depend on the inputs, those weights must be on the tape, or the resulting Greeks silently omit a contribution.

Testing. The correctness test for an AAD implementation is bumping. This sounds circular but is not: bumping is a completely independent implementation path, and agreement to five or six digits on a smooth payoff is strong evidence. Test at the level of individual operations first (each rule in Table 1 against a closed form), then whole pricers against analytic Greeks where they exist. The Black–Scholes case in this note agrees to 10^{-16}; that check is worth running in CI. For the Hessian, symmetry is a free and searching test—the two triangles are computed independently.

Memory before speed. The first serious AAD deployment usually runs out of memory before it runs out of time. Instrument the tape early, and plan for checkpointing at the level of simulation steps.

Random numbers must be frozen. The adjoint sweep assumes the recorded graph is the one that produced the value. If the pricer draws random numbers lazily, or reseeds, or uses a thread-dependent stream, the reverse sweep will compute derivatives of a different function than the one evaluated. Draws must be fixed inputs to the taped computation.

Parallelism. Adjoint accumulation is a $+=$ into shared state, so parallel path blocks race on the gradient vector. The standard fix is per-thread gradient buffers reduced at the end.

11 Where it pays, and where it does not

AAD is not free. It costs a substantial engineering commitment: the pricing library must be written in, or converted to, a differentiable style; the tape must be managed; the discontinuity

problems of §7 must be confronted trade type by trade type. It is worth being clear about when that price is worth paying.

AAD pays	**Bumping is fine**
Many inputs per valuation—bucketed vegas, curve deltas, correlation matrices. The saving scales with n.	Few inputs. Below $n \approx 4$ the adjoint's constant overhead is not recovered.
Expensive valuations: Monte Carlo, PDE solvers, nested simulation.	Closed-form or near-instant pricers, where $2n$ evaluations of something cheap is still cheap.
Risk needed repeatedly—intraday, or under many scenarios—so the build cost amortises.	One-off or exploratory analysis.
XVA and capital, where an exposure profile is itself an expensive nested calculation and the sensitivity count is enormous.	Products whose payoffs are dominated by discontinuities that resist smoothing.
Second-order risk actually required.	Legacy pricers that cannot be modified, or that call opaque external libraries.

Table 2: When the engineering investment is justified.

The strongest case, historically, has been XVA. Computing a credit valuation adjustment means simulating the future exposure of an entire netting set across thousands of scenarios and dates—a calculation whose single evaluation already takes minutes, and whose sensitivities run to thousands of inputs spanning rates, credit spreads and volatilities across every currency in the book. Bumping that is not slow; it is impossible. This is where AAD moved from a research topic to infrastructure, and it is why most large dealers now have an adjoint-capable pricing library.

A reasonable adoption path exists for a desk that does not want to rewrite everything. Start with the highest-input, highest-cost product. Implement operator-overloading AAD on that pricer alone, with bumping retained as the regression test. Add checkpointing when the tape becomes the constraint. Extend to second order only when a specific need appears. And treat the calibration adjoint of §9 as a separate, self-contained piece of work—it is independent of the pricer, applies to the whole book at once, and often delivers the largest single improvement in the usefulness of the risk numbers.

A Jacobian-vector and vector-Jacobian products

The two modes are cleanly described in terms of what they can compute in a single pass, and it is worth having the language.

Let $F : \mathbb{R}^n \to \mathbb{R}^m$ with Jacobian $J \in \mathbb{R}^{m \times n}$. Forming J explicitly costs mn entries and is usually neither necessary nor affordable. What the two modes give is:

Forward mode computes a **Jacobian-vector product** (JVP), $J\dot{x}$, for a chosen input direction $\dot{x} \in \mathbb{R}^n$, at a constant times the cost of one evaluation of F. Setting $\dot{x} = e_j$ extracts the j-th *column* of J: the sensitivity of every output to input j. Recovering all of J takes n passes.

Reverse mode computes a **vector-Jacobian product** (VJP), $\bar{y}^\top J$, for a chosen output direction $\bar{y} \in \mathbb{R}^m$, at a constant times the cost of one evaluation. Setting $\bar{y} = e_i$ extracts the i-th *row*: the sensitivity of output i to every input. Recovering all of J takes m passes.

So the choice is dictated by shape. A pricer has $m = 1$ and n large, so one reverse pass suffices and forward mode would need n. A calibration Jacobian has m residuals and n parameters of comparable size, and either mode costs $\min(m, n)$ passes.

The Hessian construction of §8 is now easy to name. The gradient map $x \mapsto \nabla V(x)$ is a function $\mathbb{R}^n \to \mathbb{R}^n$ whose Jacobian is the Hessian. Applying forward mode to it—a JVP of a VJP—gives $H\dot{x}$ in one pass, which is forward-over-reverse.

B The cost theorem

The claim of (14) is that the gradient costs a bounded multiple of the function, independent of n. The argument is a counting exercise.

Let the program consist of elementary operations $v_k = f_k(v_i, v_j)$, each with at most two arguments. Write OPS for the total count. The forward evaluation costs OPS operations by definition.

The reverse sweep visits each recorded operation exactly once. At operation k it computes at most two local partials and performs at most two multiply-accumulates, $\bar{v}_i \mathrel{+}= \bar{v}_k \, \partial v_k / \partial v_i$. The local partials of the elementary operations are themselves elementary: for $w = uv$ they are v and u, already available; for $w = \exp u$ the partial is w; for $w = \log u$ it is $1/u$. So the work at each node is bounded by a small constant c, giving a reverse sweep of at most $c \cdot \mathrm{OPS}$.

Total: $(1 + c)\,\mathrm{OPS}$. Under the standard accounting in which multiplication, addition and division are each counted as one operation, the bound works out to $\omega \le 4$; Baur and Strassen [1] give the original result for rational functions, and Griewank and Walther [2] give the general treatment.

Nowhere does n enter. The reverse sweep does the same work whether one input derivative is wanted or all of them, because it computes adjoints for every intermediate regardless—the inputs are simply the intermediates that happen to have no parents.

Two things the theorem does not promise. It is a bound on *operations*, not on wall-clock time; tape allocation and the memory traffic of the reverse sweep are real costs not captured by the count, which is why measured constants land nearer 4 than 2. And it says nothing about memory, which is the subject of §6.

C A reverse-mode engine in eighty lines

The following is the core of the engine used for every measurement in this note. It is operator-overloading reverse mode with a global tape. Values are numpy arrays so that the whole thing vectorises across Monte Carlo paths.

```
import numpy as np

TAPE = []                       # every operation, in execution order

class Var:
    """A node: a value, its parents, and the local partials."""
    __array_ufunc__ = None      # so ndarray * Var defers to Var.__rmul__

    def __init__(self, v, parents=()):
        self.v = np.asarray(v, dtype=float)
        self.parents = parents        # ((parent, dself_dparent), ...)
        self.adj = None
        TAPE.append(self)

    def __add__(self, o):
        o = asvar(o)
        return Var(self.v + o.v, ((self, 1.0), (o, 1.0)))

    def __mul__(self, o):
        o = asvar(o)
        return Var(self.v * o.v, ((self, o.v), (o, self.v)))

    def __truediv__(self, o):
        o = asvar(o); out = self.v / o.v
```

```
        return Var(out, ((self, 1.0 / o.v), (o, -out / o.v)))

    __radd__ = __add__
    __rmul__ = __mul__
```

The elementary functions follow the same pattern: compute the value, record the local partial.

```
from scipy.stats import norm

def asvar(x):
    return x if isinstance(x, Var) else Var(x)

def vcdf(x):                            # standard normal CDF
    x = asvar(x)
    return Var(norm.cdf(x.v), ((x, norm.pdf(x.v)),))

def vexp(x):
    x = asvar(x); e = np.exp(x.v)
    return Var(e, ((x, e),))

def vlog(x):
    x = asvar(x)
    return Var(np.log(x.v), ((x, 1.0 / x.v),))

def vsqrt(x):
    x = asvar(x); s = np.sqrt(x.v)
    return Var(s, ((x, 0.5 / s),))
```

The reverse sweep itself is eight lines. Everything above exists only to feed it.

```
def backward(y):
    """Seed the output and sweep the tape in reverse.  Afterwards every
    Var x on the tape carries x.adj = dy/dx."""
    for n in TAPE:
        n.adj = None
    y.adj = np.ones_like(y.v)
    for n in reversed(TAPE):
        if n.adj is None:
            continue
        for parent, local in n.parents:
            c = sum_to(n.adj * local, parent.v.shape)
            parent.adj = c if parent.adj is None else parent.adj + c
```

One bookkeeping helper is needed because the pricer is vectorised: an adjoint arrives with whatever shape numpy broadcasting produced, and must be reduced back to its parent's shape before it is accumulated.

```
def sum_to(g, shape):
    """Undo numpy broadcasting so an adjoint matches its parent's shape."""
    g = np.asarray(g, dtype=float)
    if g.shape == shape:
        return g
    if np.broadcast_shapes(g.shape, shape) == shape:
```

```
        return np.broadcast_to(g, shape)
    while g.ndim > len(shape):
        g = g.sum(axis=0)
    for i, s in enumerate(shape):
        if s == 1 and g.shape[i] != 1:
            g = g.sum(axis=i, keepdims=True)
    return g.reshape(shape)
```

Used on a Black–Scholes call, one reverse sweep produces four Greeks:

```
def bs_call(S, K, r, q, sigma, T):
    sT = vsqrt(sigma * sigma * T)
    d1 = (vlog(S / K) + (r - q + 0.5 * sigma * sigma) * T) / sT
    return S * vexp(-q * T) * vcdf(d1) - K * vexp(-r * T) * vcdf(d1 - sT)

S, r, q, v = Var(100.0), Var(0.03), Var(0.015), Var(0.22)
price = bs_call(S, K=105.0, r=r, q=q, sigma=v, T=1.5)
backward(price)
#   S.adj -> delta,  v.adj -> vega,  r.adj -> rho,  q.adj -> dividend rho
#   all four agree with the closed forms to 1e-15 relative
```

The tape for this is 37 nodes. Sanity checks worth automating: every rule in Table 1 against a closed form; the Black–Scholes Greeks above; the gradient of any smooth pricer against central bumping at $h \sim 10^{-6}$; and, for second order, symmetry of the Hessian, which is a searching test because the two triangles are computed by separate passes.

D Adjoint of a Newton solver

§9 asserted that one does not differentiate through the solver's iterations. This appendix makes the point concrete on the simplest possible case, implied volatility.

Implied volatility Σ solves $BS(\Sigma) - m = 0$ for a market price m, by Newton iteration

$$\Sigma_{k+1} \;=\; \Sigma_k - \frac{BS(\Sigma_k) - m}{BS'(\Sigma_k)}. \tag{22}$$

A differentiation engine handed this loop will happily tape all of it: twenty or so iterations, each containing a full option pricing, each contributing to the tape. The reverse sweep then produces the derivative of the *approximation after twenty iterations*—correct to the solver's tolerance, but at twenty times the tape and twenty times the work.

The implicit function theorem does it in one line. Since $BS(\Sigma(m)) = m$ identically,

$$BS'(\Sigma)\,\frac{\mathrm{d}\Sigma}{\mathrm{d}m} = 1 \qquad\Longrightarrow\qquad \frac{\mathrm{d}\Sigma}{\mathrm{d}m} \;=\; \frac{1}{BS'(\Sigma)} \;=\; \frac{1}{\text{vega}}. \tag{23}$$

The derivative of implied volatility with respect to price is the reciprocal of vega—exact, one division, and it needs only the converged Σ and a quantity the last Newton step already computed.

The general recipe follows the same shape. For any solver returning θ with $g(\theta, m) = 0$:

1. Run the solver with taping *disabled*. Iterations are scratch work.
2. Treat the converged θ as an input to the taped computation.
3. Supply the adjoint of the solve by hand: given $\bar\theta$, solve the transposed system $(\partial g/\partial\theta)^\top \lambda = \bar\theta$ and propagate $\bar m \mathrel{+}= -(\partial g/\partial m)^\top \lambda$.

Most AAD frameworks expose a hook for exactly this—a user-defined adjoint attached to a black-box function. The same pattern covers fixed-point iterations, linear solves (whose adjoint is a solve with the transposed matrix), eigenvalue problems, and optimisers. The rule is always to differentiate the *equation that was solved*, never the algorithm that solved it.

E Glossary

Adjoint ($\bar{v}$) $\partial y/\partial v$: the sensitivity of the program's final output to an intermediate. Computed backwards.

AAD Adjoint algorithmic differentiation: reverse-mode AD applied to pricing code.

Autocallable A note that redeems early with a coupon if an underlying is above a level on an observation date; the running example of this note.

Bumping Estimating a derivative by perturbing an input and repricing; finite differencing (§2).

Cheap gradient principle The result that the full gradient costs a bounded multiple—about four—of one function evaluation, independent of the number of inputs (14).

Checkpointing Storing intermediate states and recomputing segments on demand, trading arithmetic for tape memory (§6).

Common random numbers Reusing the identical simulated draws across repricings so their difference is not swamped by Monte Carlo noise.

Cross-gamma $\partial^2 V/\partial S_i \partial S_j$: how one underlying's delta responds to another's price.

Delta, gamma, vega, rho First and second derivatives of value with respect to spot, spot twice, volatility, and interest rate.

Dual number A pair $(v, \dot{v})$ with $\epsilon^2 = 0$ arithmetic; the representation used by forward mode.

Forward (tangent) mode Propagating $\dot{v} = \partial v/\partial x_j$ forward alongside values; one input per pass (§4).

Hessian-vector product $H\dot{x}$ in one forward-over-reverse pass, without forming H (§8).

Implicit function theorem The result behind calibration adjoints: differentiate the equation the solver satisfied, not the solver's iterations (19).

Likelihood-ratio estimator Moving the derivative onto the density rather than the payoff; unbiased for discontinuous payoffs, higher variance (6).

Market Greek Sensitivity to a quoted, tradable instrument—what a hedge is expressed in. Contrast *model Greek* (§9).

Operator overloading Building the tape by replacing the numeric type and overriding its arithmetic; contrast *source transformation* (§6).

Pathwise estimator Differentiating the payoff inside the Monte Carlo average, holding the random draw fixed (5). What AAD mechanises.

Reverse (adjoint) mode Sweeping the recorded graph backwards from $\bar{y} = 1$, producing the whole gradient in one pass (§5).

Seed The initial adjoint $\bar{y} = 1$ that starts the reverse sweep.

Smoothing Replacing a discontinuous payoff with a continuous approximation so the pathwise derivative exists; trades bias against variance (§7).

Tape The record of executed operations and their local partials, needed because the reverse sweep runs opposite to the order the values were produced (§6).

Vanna $\partial^2 V / \partial S \, \partial \sigma$: how delta responds to volatility.

XVA Valuation adjustments (credit, funding, capital) computed by nested simulation over a whole netting set; the workload that made AAD infrastructure.

References

[1] W. Baur, V. Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22:317–330, 1983. (The original cheap-gradient result.)

[2] A. Griewank, A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. 2nd edition, SIAM, 2008. (The standard reference.)

[3] A. Griewank, A. Walther. Algorithm 799: `revolve` — an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software*, 26:19–45, 2000.

[4] M. Giles, P. Glasserman. Smoking adjoints: fast Monte Carlo Greeks. *Risk*, 19(1):88–92, 2006. (The paper that brought adjoints to derivatives pricing.)

[5] P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, 2004. (Chapter 7 covers pathwise and likelihood-ratio estimators.)

[6] M. Giles. Vibrato Monte Carlo sensitivities. In *Monte Carlo and Quasi-Monte Carlo Methods 2008*, Springer, 369–382, 2009.

[7] L. Capriotti. Fast Greeks by algorithmic differentiation. *Journal of Computational Finance*, 14(3):3–35, 2011.

[8] L. Capriotti, M. Giles. Algorithmic differentiation: adjoint Greeks made easy. *Risk*, 25(9):92–98, 2012.

[9] A. Savine. *Modern Computational Finance: AAD and Parallel Simulations*. Wiley, 2018. (A production-oriented C++ treatment.)

[10] M. Henrard. *Algorithmic Differentiation in Finance Explained*. Palgrave Macmillan, 2017.

[11] F. A. Longstaff, E. S. Schwartz. Valuing American options by simulation: a simple least-squares approach. *Review of Financial Studies*, 14:113–147, 2001.

[12] L. Hascoet, V. Pascual. The Tapenade automatic differentiation tool. *ACM Transactions on Mathematical Software*, 39(3):1–43, 2013.

[13] M. Sagebaum, T. Albring, N. R. Gauger. High-performance derivative computations using CoDiPack. *ACM Transactions on Mathematical Software*, 45(4):1–26, 2019.

[14] A. Walther, A. Griewank. Getting started with ADOL-C. In *Combinatorial Scientific Computing*, Chapman-Hall, 181–202, 2012.