Adjoint methods for option pricing, Greeks and calibration using PDEs and SDEs

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford-Man Institute of Quantitative Finance

Outline

- introductory ideas
- black-box adjoints
- high-level linear algebra adjoints
- automatic differentiation
- backward and forward PDEs for pricing
- backward and forward discrete equations
- what can go wrong with PDEs?
- modular approach to calibration
- Monte Carlo pathwise sensitivities
- path-dependent payoffs
- binning
- handling discontinuities

A question!

Given compatible matrices A, B, C does it matter how one computes the product $A\,B\,C$? (i.e. $(A\,B)\,C$ or $A\,(B\,C)$?)

A question!

Given compatible matrices A, B, C does it matter how one computes the product $A\,B\,C$? (i.e. $(A\,B)\,C$ or $A\,(B\,C)$?)

Answer 1: no, in theory, and also in practice if A, B, C are square

A question!

Given compatible matrices A, B, C does it matter how one computes the product $A\,B\,C$? (i.e. $(A\,B)\,C$ or $A\,(B\,C)$?)

Answer 1: no, in theory, and also in practice if A, B, C are square

Answer 2: yes, in practice, if A, B, C have dimensions 1×10^5, $10^5\times 10^5$, $10^5\times 10^5$.

$$\begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

A question!

Given compatible matrices A, B, C does it matter how one computes the product $A\,B\,C$? (i.e. $(A\,B)\,C$ or $A\,(B\,C)$?)

Answer 1: no, in theory, and also in practice if A, B, C are square

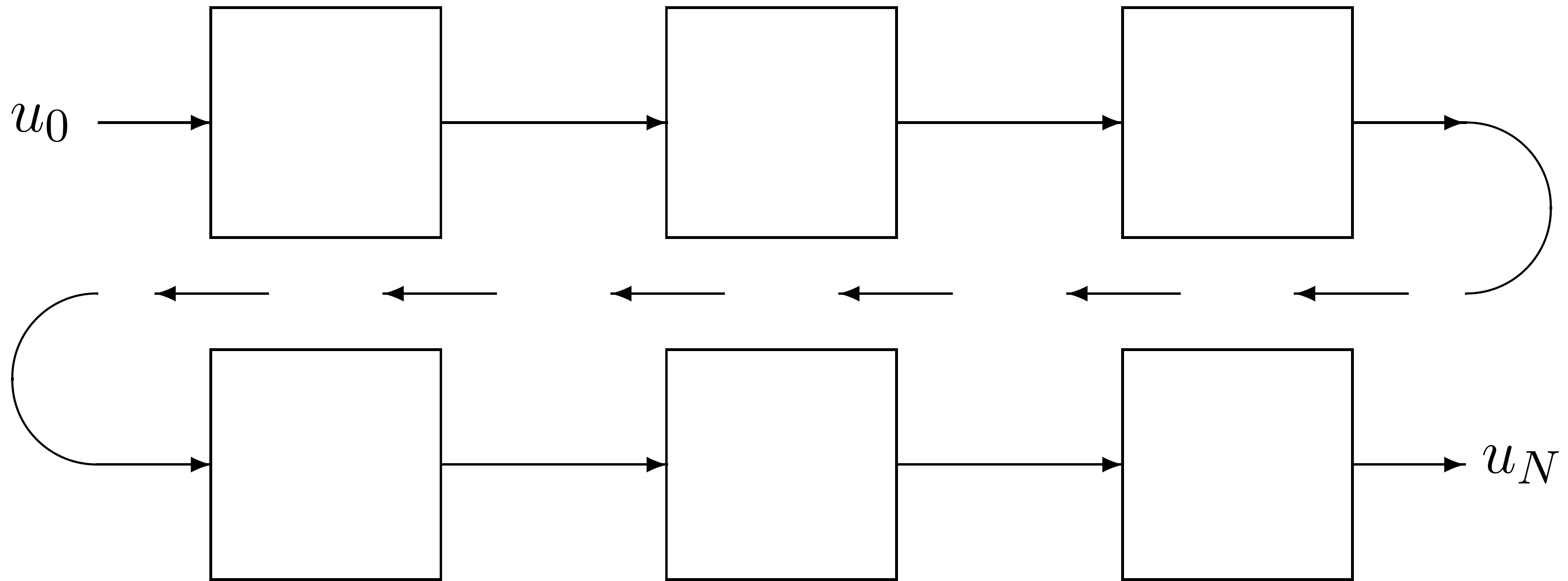
Answer 2: yes, in practice, if A, B, C have dimensions 1×10^5, $10^5\times 10^5$, $10^5\times 10^5$.

$$\left(\begin{array}{ccccc} \cdot & \cdot & \cdot & \cdot & \cdot \end{array} \right) \left(\begin{array}{ccccc} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{array} \right) \left(\begin{array}{ccccc} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{array} \right)$$

Point: this is all about computational efficiency

Generic black-box problem

An input vector u_0 leads to a scalar output u_N:



Each box could be a mathematical step (calibration, spline, pricing) or a computer code, or one computer instruction

Key assumption: each step is (locally) differentiable

Generic black-box problem

Let $\dot{u}_n$ represent the derivative of u_n with respect to one particular element of input u_0. Differentiating black-box processes gives

$$\dot{u}_{n+1} = D_n \, \dot{u}_n, \qquad D_n \equiv \frac{\partial u_{n+1}}{\partial u_n}$$

and hence

$$\dot{u}_N = D_{N-1} \, D_{N-2} \, \ldots \, D_1 \, D_0 \, \dot{u}_0$$

- standard “forward mode” approach multiplies matrices from right to left – very natural
- each element of u_0 requires its own sensitivity calculation – cost proportional to number of inputs

Generic black-box problem

Let $\overline{u}_n$ be the derivative of output u_N with respect to u_n.

$$\overline{u}_n \equiv \left(\frac{\partial u_N}{\partial u_n} \right)^T = \left(\frac{\partial u_N}{\partial u_{n+1}} \; \frac{\partial u_{n+1}}{\partial u_n} \right)^T = D_n^T \, \overline{u}_{n+1}$$

and hence

$$\overline{u}_0 = D_0^T \; D_1^T \; \ldots \; D_{N-2}^T \, D_{N-1}^T \, \overline{u}_N$$

and $\overline{u}_N = 1$.

- $\overline{u}_0$ gives sensitivity of u_N to all elements of u_n at a fixed cost, not proportional to the size of u_0.
- a different output would require a separate adjoint calculation; cost proportional to number of outputs

Generic black-box problem

It looks easy (?) – what's the catch?

- need to do original nonlinear calculation to compute and store D_n before doing adjoint reverse pass
  – storage requirements can be significant for PDEs
- practical implementation can be tedious if hand-coded
  – use automatic differentiation tools
- need care in treating black-boxes which involve a fixed point iteration
- derivative may not be as accurate as original approximation

Automatic differentiation

We now consider a single black-box component, which is actually the outcome of a computer program.

A computer instruction creates an additional new value:

$$\mathbf{u}_{n+1} \;=\; \mathbf{f}_n(\mathbf{u}_n) \;\equiv\; \begin{pmatrix} \mathbf{u}_n \\ f_n(\mathbf{u}_n) \end{pmatrix},$$

A computer program is the composition of N such steps:

$$\mathbf{u}_N = \mathbf{f}_{N-1} \circ \mathbf{f}_{N-2} \circ \ldots \circ \mathbf{f}_1 \circ \mathbf{f}_0(\mathbf{u}_0).$$

Automatic differentiation

In forward mode, differentiation gives

$$\dot{\mathbf{u}}_{n+1} = D_n \, \dot{\mathbf{u}}_n, \quad D_n \equiv \begin{pmatrix} I_n \\ \partial f_n / \partial \mathbf{u}_n \end{pmatrix},$$

and hence

$$\dot{\mathbf{u}}_N = D_{N-1} \; D_{N-2} \ldots \; D_1 \; D_0 \; \dot{\mathbf{u}}_0.$$

Automatic differentiation

In reverse mode, we have

$$\overline{\mathbf{u}}_n = \left(D_n\right)^T \overline{\mathbf{u}}_{n+1}.$$

and hence

$$\overline{\mathbf{u}}_0 = (D_0)^T \, (D_1)^T \ldots \, (D_{N-2})^T \, (D_{N-1})^T \, \overline{\mathbf{u}}_N.$$

Note: need to go forward through original calculation to compute/store the D_n, then go in reverse to compute $\overline{\mathbf{u}}_n$

Automatic differentiation

At the level of a single instruction

$$c = f(a, b)$$

the forward mode is

$$\begin{pmatrix} \dot{a} \\ \dot{b} \\ \dot{c} \end{pmatrix}_{n+1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ \frac{\partial f}{\partial a} & \frac{\partial f}{\partial b} \end{pmatrix} \begin{pmatrix} \dot{a} \\ \dot{b} \end{pmatrix}_{n}$$

and so the reverse mode is

$$\begin{pmatrix} \overline{a} \\ \overline{b} \end{pmatrix}_{n} = \begin{pmatrix} 1 & 0 & \frac{\partial f}{\partial a} \\ 0 & 1 & \frac{\partial f}{\partial b} \end{pmatrix} \begin{pmatrix} \overline{a} \\ \overline{b} \\ \overline{c} \end{pmatrix}_{n+1}$$

Automatic differentiation

This gives a prescriptive algorithm for reverse mode differentiation.

$$
\begin{aligned}
\overline{a} &= \overline{a} + \frac{\partial f}{\partial a}\, \overline{c} \\
\overline{b} &= \overline{b} + \frac{\partial f}{\partial b}\, \overline{c}
\end{aligned}
$$

Manual implementation is possible but can be tedious, so automated tools have been developed, following two approaches:

- operator overloading (ADOL-C, FADBAD++)
- source code transformation (Tapenade, TAF/TAC++)

Source code transformation

- programmer supplies black-box code which takes u as input and produces $v = f(u)$ as output
- in forward mode, AD tool generates new code which takes u and $\dot{u}$ as input, and produces v and $\dot{v}$ as output

$$\dot{v} = \left(\frac{\partial f}{\partial u}\right) \dot{u}$$

- in reverse mode, AD tool generates new code which takes u and $\overline{v}$ as input, and produces v and $\overline{u}$ as output

$$\overline{u} = \left(\frac{\partial f}{\partial u}\right)^T \overline{v}$$

Linear algebra sensitivities

Low-level automatic differentiation is very helpful, but a high-level approach is sometimes better (e.g. when using libraries)

Won't go through derivation – just present results.

Notation: $\dot{C}_{ij} \equiv \dfrac{\partial C_{ij}}{\partial \, \text{input}}, \quad \overline{C}_{ij} \equiv \dfrac{\partial \, \text{output}}{\partial C_{ij}}$

(Note: some literature defines $\overline{C}$ as the transpose)

Linear algebra sensitivities

$$C = A + B$$

$$\dot{C} = \dot{A} + \dot{B}$$

$$\overline{A} = \overline{C},$$
$$\overline{B} = \overline{C}$$

$$C = A\,B$$

$$\dot{C} = \dot{A}\,B + A\,\dot{B}$$

$$\overline{A} = \overline{C}\,B^T$$
$$\overline{B} = A^T\,\overline{C}$$

$$C = A^{-1}$$

$$\dot{C} = -\,C\,\dot{A}\,C$$

$$\overline{A} = -C^T\,\overline{C}\,C^T$$

$$C = A^{-1}B$$

$$\dot{C} = A^{-1}(\dot{B} - \dot{A}\,C)$$

$$\overline{B} = (A^T)^{-1}\,\overline{C}, \;\; \overline{A} = -\overline{B}\,C^T$$

Linear algebra sensitivities

One important little catch: when A is a tri-diagonal matrix, and B and C are both vectors,

$$C = A^{-1} B$$

$$\dot{C} = A^{-1} (\dot{B} - \dot{A}\, C)$$

$$\overline{B} = (A^T)^{-1}\, \overline{C}, \qquad \overline{A} = -\overline{B}\, C^T$$

this gives a dense matrix $\overline{A}$, at $O(n^2)$ cost – since A is tri-diagonal then only the tri-diagonal elements of $\overline{A}$ should be computed, at $O(n)$ cost

Linear algebra sensitivities

Others:

- matrix determinant
- matrix polynomial $p_n(A)$ and exponential $\exp(A)$
- eigenvalues and eigenvectors of A, assuming no repeated eigenvalues
- SVD (singular value decomposition) of A, assuming no repeated singular values
- Cholesky factorisation of symmetric A

Most of the adjoint results are 30-40 years old, but not widely known.

Fixed point iteration

Suppose a black-box computes output v from input u by solving the nonlinear equations

$$f(u,v) = 0$$

using the fixed-point iteration

$$v_{n+1} = v_n - P(u,v_n)\; f(u,v_n).$$

For Newton iteration P is the inverse Jacobian, but P could also correspond to a multigrid cycle in an iterative solver.

Fixed point iteration

A naive forward mode differentiation uses the fixed-point iteration

$$\dot{v}_{n+1} = \dot{v}_n - \left(\frac{\partial P}{\partial u}\,\dot{u} + \frac{\partial P}{\partial v}\,\dot{v}_n \right)\ f(u, v_n) - P(u, v_n) \left(\frac{\partial f}{\partial u}\,\dot{u} + \frac{\partial f}{\partial v}\,\dot{v}_n \right)$$

but it is more efficient to use

$$\dot{v}_{n+1} = \dot{v}_n - P(u, v)\ \left(\frac{\partial f}{\partial u}\,\dot{u} + \frac{\partial f}{\partial v}\,\dot{v}_n \right)$$

to iteratively solve

$$\frac{\partial f}{\partial u}\,\dot{u} + \frac{\partial f}{\partial v}\,\dot{v} = 0$$

Fixed point iteration

Since

$$\dot{v} = -\left(\frac{\partial f}{\partial v}\right)^{-1} \frac{\partial f}{\partial u}\,\dot{u},$$

the adjoint is

$$\overline{u} = -\left(\frac{\partial f}{\partial u}\right)^{T} \left(\frac{\partial f}{\partial v}\right)^{-T} \overline{v} = \left(\frac{\partial f}{\partial u}\right)^{T} \overline{w}$$

where

$$\left(\frac{\partial f}{\partial v}\right)^{T} \overline{w} + \overline{v} = 0.$$

Fixed point iteration

This can be solved iteratively using

$$\overline{w}_{n+1} = \overline{w}_n - P^T(u,v) \left(\left(\frac{\partial f}{\partial v} \right)^T \overline{w}_n + \overline{v} \right)$$

and this is guaranteed to converge (well!) since

$$P^T(u,v) \left(\frac{\partial f}{\partial v} \right)^T$$

has the same eigenvalues as

$$P(u,v) \, \frac{\partial f}{\partial v}.$$

Forward and reverse PDEs

Suppose we are interested in the forward PDE

$$\frac{\partial p}{\partial t} = L_t\, p,$$

where L_t is a spatial operator, subject to Dirac initial data $p(x,0) = \delta(x{-}x_0)$, and we want the value of the output functional

$$(p(\cdot,T),\ f) \equiv \int p(x,T)\, f(x)\, \mathrm{d}x.$$

The adjoint spatial operator L_t^* is defined by the identity

$$(L_t v, w) = (v, L_t^* w), \quad \forall v, w$$

assuming certain homogeneous b.c.'s.

Forward and reverse PDEs

If $u(x,t)$ is the solution of the adjoint PDE

$$\frac{\partial u}{\partial t} = -L_t^* \, u,$$

subject to "initial" data $u(x,T) = f(x)$ then

$$\begin{aligned}
(p(\cdot,T),\, u(\cdot,T)) - (p(\cdot,0),\, u(\cdot,0)) &= \int_0^T \left(\frac{\partial p}{\partial t},\, u \right) + \left(p,\, \frac{\partial u}{\partial t} \right) \, \mathrm{d}t \\
&= \int_0^T (L_t p,\, u) - (p,\, L_t^* u) \, \mathrm{d}t \\
&= 0,
\end{aligned}$$

and hence $u(x_0, 0) = (p(\cdot, T), f)$.

Forward and reverse FDEs

Suppose the forward problem has the discrete equivalent

$$p_{n+1} = A_n\, p_n$$

where A_n is a square matrix.

If there are N timesteps, then the output has the form

$$f^T M\, p_N = f^T\, M\, A_{N-1}\, A_{N-2}\, \ldots\, A_0\, p_0.$$

where M is a symmetric “mass” matrix, which may be diagonal.

Forward and reverse FDEs

Taking the transpose, this can be re-expressed as

$$p_0^T v_0$$

where

$$v_0 = A_0^T \, \ldots \, A_{N-2}^T \, A_{N-1}^T \, M \, f^T$$

the adjoint solution v_n is defined by

$$v_n = A_n^T v_{n+1}$$

subject to “initial” data $v_N = M^T f$.

Forward and reverse FDEs

It is sometimes more appropriate to work with

$$u_n = (M^{-1})^T v_n,$$

in which case we have

$$u_n = (M A_n M^{-1})^T u_{n+1}$$

subject to “initial” data

$$u_N = f,$$

and the output functional is $p_0^T M u_0$.

Financial relevance

Fokker-Planck (or forward Kolmogorov) equation:

$$\frac{\partial p}{\partial t} + \frac{\partial}{\partial x}(a\,p) = \frac{1}{2}\,\frac{\partial^2}{\partial x^2}\left(b^2\,p\right)$$

for probability density $p(x,t)$ for path S_t satisfying the SDE

$$\mathrm{d}S_t = a(S_t,t)\,\mathrm{d}t + b(S_t,t)\,\mathrm{d}W_t.$$

Backward Kolmogorov (or discounted Feynman-Kac) equation:

$$\frac{\partial u}{\partial t} + a\,\frac{\partial u}{\partial x} + \frac{1}{2}\,b^2\,\frac{\partial^2 u}{\partial x^2} = 0$$

where $u(x,t) = \mathbb{E}[f(S_T)|S_t = x]$

Financial relevance

The spatial operators are

$$L\,p \equiv -\frac{\partial}{\partial x}\left(a\,p\right) + \frac{1}{2}\,\frac{\partial^2}{\partial x^2}\left(b^2\,p\right)$$

and

$$L^*\,u \equiv a\,\frac{\partial u}{\partial x} + \frac{1}{2}\,b^2\,\frac{\partial^2 u}{\partial x^2}$$

The identity

$$(Lv, w) = (v, L^* w), \quad \forall v, w$$

can be verified by integration by parts, assuming

$a\,v\,w, b^2 v\,\dfrac{\partial w}{\partial x}, b^2 \dfrac{\partial v}{\partial x}\,w$ are zero on boundary.

Financial relevance

Discrete equations are usually formulated for backward equation:

$$u_n = B_n u_{n+1}$$

subject to payoff data $u_N = f$, and the output is $e^T u_n$ where e is a unit vector with a single non-zero entry.

The equivalent discrete adjoint problem is

$$P_{n+1} = B_n^T P_n$$

subject to initial data $P_0 = e$, and the output is $P_N^T f$.

P_n is a vector of discrete probabilities – need to divide by grid spacing to get approximation to probability density.

Financial relevance

With implicit time-marching, we have an equation like

$$A_n \, u_n = C_n \, u_{n+1}$$

so

$$B_n \equiv A_n^{-1} C_n$$

In this case,

$$B_n^T \equiv C_n^T (A_n^T)^{-1}$$

so

$$P_{n+1} = C_n^T (A_n^T)^{-1} P_n$$

Note time reversal: multiply by C_n and then by A_n^{-1} turns into multiply by $(A_n^T)^{-1}$ and then by C_n^T

Financial relevance

Which is better – forward or reverse?

- forward is best for pricing multiple European options
  - a single forward calculation and then a separate vector dot product for each option
  - particularly useful when calibrating a model to vanilla options?
- backward is only possibility for American options, and also gives Delta and Gamma approximations for free

FDE sensitivities

Suppose we want to compute $P = e^T u_0$ where $u_N = f$ and

$$u_n = B_n \, u_{n+1}.$$

Now suppose that f and B_n depend on some parameter θ, and we want to compute the sensitivity to θ.

Standard "forward mode" sensitivity analysis gives $\dot{P} = e^T \dot{u}_0$ where $\dot{u}_N = \dot{f}$ and

$$\dot{u}_n = B_n \, \dot{u}_{n+1} + \dot{b}_n$$

where

$$\dot{b}_n \equiv \dot{B}_n \, u_{n+1}$$

FDE sensitivities

What is “reverse mode” adjoint?

Work “backwards” applying the linear algebra rules.

$$\overline{u}_0 = e$$

$$\overline{u}_{n+1} = B_n^T\, \overline{u}_n, \quad \overline{b}_n = \overline{u}_n$$

$$\overline{f} = \overline{u}_N$$

FDE sensitivities

This gives $\overline{f}$ and $\overline{b}_n$ and then payoff sensitivity is given by

$$\overline{\theta} = \overline{f}^T \dot{f} + \sum_n \overline{b}_n^T \dot{b}_n$$

This can be evaluated using AD software, or hand-coded following the AD algorithm.

$\theta, u_{n+1} \longrightarrow B_n\, u_{n+1}$	original code
$\theta, u_{n+1} \longrightarrow \dot{B}_n\, u_{n+1}$	forward mode, keeping u_{n+1} fixed
$\theta, u_{n+1}, \overline{b}_n \longrightarrow \overline{\theta}$ incr	reverse mode, keeping u_{n+1} fixed

FDE sensitivities

We now add 2 extra ingredients:

- nonlinearity (e.g. American options using penalty method)
- implicit time-marching

Including these, "forward mode" sensitivity analysis gives $\dot{P} = e^T \dot{u}_0$ where $\dot{u}_N = \dot{f}$ and

$$A_n\, \dot{u}_n = C_n\, \dot{u}_{n+1} + \dot{b}_n,$$

for some $A_n, C_n, \dot{b}_n$, and "reverse mode" gives

$$\overline{u}_{n+1} = C_n^T\, (A_n^T)^{-1} \overline{u}_n, \quad \overline{b}_n = (A_n^T)^{-1}\, \overline{u}_n$$

FDE sensitivities

This again gives $\overline{f}$ and $\overline{b}_n$ and AD ideas can then be used to compute $\overline{\theta}$.

So far, I have talked of θ being a single input parameter, but it can be a vector of input parameters.

The key is that they all use the same $\overline{f}$ and $\overline{b}_n$, and it is just this final AD step which depends on θ, and the cost is independent of the number of parameters.

What can go wrong?

Differentiation like this gives the sensitivity of the numerical approximation to changes in the input parameters.

This is not necessarily a good approximation to the true sensitivity

Simplest example: a digital put option with strike K when wanting to compute $\dfrac{\partial V}{\partial K}$, the sensitivity of the option price to the strike

What can go wrong?

Using the simplest numerical approximation,

$$f_i = H(K - S_i)$$

and so $\dot{f} = 0$ which leads to a zero sensitivity!

Using a better approximation

$$f_i = \frac{1}{\Delta S} \int_{S_i - \frac{1}{2}\Delta S}^{S_i + \frac{1}{2}\Delta S} H(K - S) \, \mathrm{d}S$$

gives an $O(\Delta S^2)$ approximation to the price, and an $O(\Delta S)$ approximation to the sensitivity to K.

What can go wrong?

More generally, discontinuities are not the only problem.

Suppose our analytic problem with input x has solution

$$u = x^2$$

and our discrete approximation with step size $h \ll 1$ is

$$u_h = x^2 + h^2 \sin(x/h)$$

then $u_h - u = O(h^2)$ but $u_h' - u' = O(h)$

This seems to be typical, that in bad cases you lose one order of convergence each time you differentiate.

What can go wrong?

Careful construction of the approximation can usually avoid these problems.

In the digital put case, the problem was the strike moving across the grid.

Solution: move the grid with the strike at maturity $t = T$, keeping the end at the current time $t = 0$ fixed.

$$\log S_i(t) = \log S_i^{(0)} + (\log K - \log K^{(0)}) \frac{t}{T}$$

This uses a baseline grid $S_i^{(0)}$ corresponding to the true strike $K^{(0)}$ then considers perturbations to this which move with the strike.

Use of adjoint sensitivities

Fokker-Planck discretisation:

- standard calculation goes forward in time, then performs a separate vector dot product for each vanilla European option
- adjoint sensitivity calculation goes backward in time, gives sensitivity of vanilla prices to initial prices, model constants
- if the Greeks are needed for each option, then a separate adjoint calculation is needed for each – might be better to use "forward mode" AD instead, depending on number of parameters and options
- one adjoint calculation can give a weighted average of Greeks – useful for calibrating a model to market data

Use of adjoint sensitivities

A calibration procedure might find the optimum vector of parameters θ which minimises the mean square difference between vanilla option model prices and market prices:

$$\frac{1}{2} \sum_k \left(C^{(k)}_{model}(\theta) - C^{(k)}_{market} \right)^2$$

Gradient-based optimisation would need to compute

$$\sum_k \left(C^{(k)}_{model} - C^{(k)}_{market} \right) \frac{\partial C^{(k)}_{model}}{\partial \theta}$$

which is just a weighted average (with both positive and negative weights) of the Greeks.

Use of adjoint sensitivities

Since the vanilla option price is of the form

$$C^{(k)}_{model} = f_k^T P_N$$

then, provided f_k does not depend on θ, the adjoint calculation works backwards in time from the “initial” condition:

$$\overline{P}_N = \sum_k \left(C^{(k)}_{model} - C^{(k)}_{market)} \right) f_k$$

Use of adjoint sensitivities

Black-Scholes / backward Kolmogorov discretisation:

- standard calculation goes backward in time for pricing an exotic option, with possible path-dependency and optional exercise
- adjoint sensitivity calculation goes forward in time, giving sensitivity of price to initial prices, model constants, etc.

Use of adjoint sensitivities

Many applications may involve a process which goes through several stages:

- market implied vol $\sigma_I \implies$ local vol σ_l at a few points using Dupire's formula
- local vol σ_l at a few points $\implies \sigma_l, \sigma_l'$ through cubic spline procedure
- $\sigma_l, \sigma_l' \implies \sigma$ at FDE grid points using cubic spline interpolation
- σ at FDE grid points $\implies$ exotic option value V using FDE calculation

Use of adjoint sensitivities

To obtain the sensitivity of the option value to changes in the market implied vol, go through all of the stages in the reverse order:

- $\overline{V} \implies \overline{\sigma}$
- $\overline{\sigma} \implies \overline{\sigma_l}, \overline{\sigma'_l}$
- $\overline{\sigma_l}, \overline{\sigma'_l} \implies \overline{\sigma_l}$
- $\overline{\sigma_l} \implies \overline{\sigma_I}$

Each stage needs to be developed and validated separately, then they all fit together in a modular way.

Use of adjoint sensitivities

It is not necessary to use adjoint techniques at each stage.

For example, the final stage in the last example computes

$$\overline{\sigma_I} = \left(\frac{\partial \sigma_l}{\partial \sigma_I} \right)^T \overline{\sigma_l}$$

The matrix

$$\frac{\partial \sigma_l}{\partial \sigma_I}$$

can be obtained by forward mode sensitivity analysis (more expensive), or approximated by bumping (more expensive and less accurate)

Monte Carlo sensitivities

Pathwise sensitivity analysis is very simple, in concept

Monte Carlo estimate for option value

$$M^{-1} \sum_{m=1}^{M} P(S^{(m)})$$

Standard pathwise estimate for sensitivity

$$M^{-1} \sum_{m=1}^{M} \frac{\partial P}{\partial S} \, \dot{S}^{(m)}$$

where $\dot{S}$ is path sensitivity, keeping fixed all of the random numbers

Monte Carlo sensitivities

The corresponding adjoint (reverse mode) sensitivity is

$$M^{-1} \sum_{m=1}^{M} \overline{\theta}^{(m)}$$

where $\overline{\theta}^{(m)}$ corresponds to $\left(\dfrac{\partial P}{\partial \theta}\right)^T$ for m^{th} path

Note: the adjoint sensitivity is the same as the standard pathwise sensitivity, so it is valid under the same conditions (e.g. $P(\theta)$ Lipschitz and piecewise differentiable)

Monte Carlo sensitivities

Largely a straightforward application of reverse mode AD, but a few new things to discuss

- path-dependent payoffs (Asian and lookback options)
- efficiency improvement for handling multiple European payoffs (Christoph Kaebe & Ekkehard Sachs)
- binning for expensive pre-processing steps (Luca Capriotti)
- handling discontinuous payoffs

Path dependent payoffs

A single path calculation (for a given set of random numbers) can be described by

$$S_{n+1} = f_n(\theta; S_n), \quad n = 0, \ldots, N-1$$

with payoff $P(S)$ depending on the whole path.

Forward mode sensitivity analysis gives

$$\dot{S}_{n+1} = B_n \, \dot{S}_n + \dot{b}_n, \quad n = 0, \ldots, N-1$$

with payoff sensitivity

$$\dot{P} = \sum_{n=0}^{N} \frac{\partial P}{\partial S_n} \, \dot{S}_n$$

Path dependent payoffs

When computing Delta, we have $\dot{b}_n = 0$ and so

$$\dot{P} = \sum_{n=0}^{N} \frac{\partial P}{\partial S_n} \, B_{n-1} \, B_{n-2} \, \ldots \, B_0 \, \dot{S}_0$$

This is equal to $\overline{S}_0^T \, \dot{S}_0$ if the adjoint solution is defined by

$$\overline{S}_N = \left(\frac{\partial P}{\partial S_N} \right)^T$$

and

$$\overline{S}_n = B_n^T \, \overline{S}_{n+1} + \left(\frac{\partial P}{\partial S_n} \right)^T, \quad n = N-1, \ldots, 0$$

Path dependent payoffs

When $\dot{S}_0 = 0$, and there is just one $\dot{b}_n$ which is non-zero, then the payoff sensitivity is

$$\dot{P} \;=\; \frac{\partial P}{\partial S_N}\, B_{N-1} \,\ldots\, B_{n+1}\, \dot{b}_n \;=\; \overline{S}_{n+1}^{\,T}\, \dot{b}_n$$

In the most general case therefore, we have

$$\dot{P} = \overline{S}_0^{\,T}\, \dot{S}_0 + \sum_{n=0}^{N-1} \overline{S}_{n+1}^{\,T}\, \dot{b}_n$$

so $\overline{b}_n \equiv \overline{S}_{n+1}$

Path dependent payoffs

Having discussed the maths, the good news is that all of the details should be handled automatically by the AD tools.

If `step(n,theta,S)` performs the nth timestep, taking θ, S_n as input and returning S_{n+1}, then the adjoint routine `step_b(n,theta,theta_b,S,S_b)` takes inputs $\theta, \overline{\theta}, S_n, \overline{S}_{n+1}$ and returns an updated $\overline{\theta}$ and $\overline{S}_n$.

The only thing you have to add to $\overline{S}_n$ is $\left(\frac{\partial P}{\partial S_n}\right)^T$.

This could also be handled by AD, but maybe simpler to do it by hand – e.g. for lookback options you just need to store which timestep has the minimum or maximum, whereas AD would need to store lots of other info.

Path dependent payoffs

An alternative point of view / approach is to make the payoff depend only on the final state S_N by augmenting the state:

- $\displaystyle\sum_n S_n$ for Asian options
- $\displaystyle\min_n S_n, \max_n S_n$ for lookback options

Doing it this way, the whole adjoint code can be generated by AD.

Path dependent payoffs

Some more implementation detail:

- first, go forward through the path storing the state S_n at each timestep (corresponds to “checkpointing” in AD terminology)
- then, go backwards through the path, using reverse mode AD for each step – this will re-do the internal calculations for the timestep and then do its adjoint
- when hand-coding for maximum performance, I also store the result of any very expensive operations (typically `exp`) to avoid having to re-do these

Note that this is different from applying AD to the entire path, which would require a lot of storage – it’s cheaper to re-calculate the internal variables rather than fetch them from main memory

Multiple European payoffs

Suppose that you have

- n_θ input parameters
- n_P different payoffs
- dimension d path simulation

If n_θ is smallest, use forward mode sensitivity analysis

If n_P is smallest, use reverse mode sensitivity analysis

What if d is smallest?

Multiple European payoffs

Going back to original matrix question, what is the best way of computing this?

$$
\begin{pmatrix}
\cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix}
\begin{pmatrix}
\cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot
\end{pmatrix}
\begin{pmatrix}
\cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix}
\begin{pmatrix}
\cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix}
$$

Multiple European payoffs

The most efficient approach is

- perform d adjoint calculations to determine

$$\frac{\partial S_N}{\partial \theta}$$

- perform d forward sensitivity calculations to determine

$$\frac{\partial P_k}{\partial S_N}$$

- combine these to obtain

$$\frac{\partial P_k}{\partial \theta} = \frac{\partial P_k}{\partial S_N} \, \frac{\partial S_N}{\partial \theta}$$

Binning

The need for binning is best demonstrated by the case of correlation Greeks.

The standard pricing calculation has three stages

- perform Cholesky factorisation
- do M path calculations
- compute average and confidence interval

How do we compute the adjoint sensitivity to the correlation coefficients?

Binning

If we apply the reverse mode AD approach to the entire calculation, then we get an estimate of

- the sensitivity of the price
- the sensitivity of the confidence interval,
  not the confidence interval for the sensitivity!

To get the confidence interval for the sensitivity, for each path we can do the adjoint of the Cholesky factorisation, so we compute $\overline{\theta}$ for each path and then compute an average and confidence interval in the usual way.

However, this greatly increases the computational cost.

Binning

The binning approach splits the M paths into K groups.

For each group, it uses the full AD approach to efficiently compute an estimate of the price sensitivity.

It then uses the variability between the group averages to estimate the confidence interval.

Needs

- $K \gg 1$ to get a good estimate of the confidence interval
- $K \ll M$ for cost of K adjoint Cholesky calculations to be smaller than M path calculations

Binning

The same approach can be used for a Monte Carlo version of the earlier example with local volatility:

- market implied vol $\sigma_I \implies$ local vol σ_l at a few points using Dupire's formula
- local vol σ_l at a few points $\implies \sigma_l, \sigma_l'$ through cubic spline procedure
- M Monte Carlo path calculation, using spline evaluation to obtain local volatility
- compute average and confidence interval

The adjoint of the path calculation will contribute increments to $\overline{\sigma_l}$ and $\overline{\sigma_l'}$. Then, for each group of paths, can use adjoint of first two stages to get an estimate for the sensitivity to market implied vol data.

Non-smooth payoffs

The biggest limitation of the pathwise sensitivity method (both forward mode and reverse mode) is that it cannot handle discontinuous payoffs.

There are 3 main ways to deal with this:

- explicitly smoothed payoffs
- using conditional expectation to smooth the payoff
- “vibrato” Monte Carlo

Of course, one can also switch to Likelihood Ratio Method or Malliavin calculus, but then I don’t see how one gets the efficiency benefits of adjoint methods.

Non-smooth payoffs

Explicitly-smoothed payoffs replace the discontinuous payoff by a smooth (or at least continuous) alternative.

Digital options $P(S) \equiv H(S\!-\!K)$ can be replaced by a piecewise linear version, or something much smoother:

$$\Phi\left(\frac{S-K}{\delta}\right)$$

This introduces an $O(\delta^2)$ error due to the smoothing, but with Richardson extrapolation this can be improved to $O(\delta^4)$ by using

$$\frac{4}{3}\,\Phi\left(\frac{S-K}{\delta}\right) - \frac{1}{3}\,\Phi\left(\frac{S-K}{2\delta}\right)$$

Non-smooth payoffs

Implicitly-smoothed payoffs use conditional expectations.

My favourite is for barrier options, where a Brownian Bridge conditional expectation computes the probability that the path has crossed the barrier within a timestep.
(see Glasserman's book, pp. 366-370)

This improves the weak convergence to first order, and makes the payoff differentiable.

Non-smooth payoffs

With digital options, can stop the path simulation one timestep before maturity.

Conditional on the value S_{N-1}, an Euler discretisation for the final timestep gives a Gaussian p.d.f. for S_N:

$$S_N = S_{N-1} + \mu_{N-1} \Delta t + \sigma_{N-1} \Delta W_{N-1}$$

In simple cases one can then analytically evaluate

$$\mathbb{E}\left[\, P(S_N) \,|\, S_{N-1} \,\right]$$

and this will be a smooth function of S_{N-1} so we can use the pathwise sensitivity method.

Non-smooth payoffs

Continuing this digital example, in more complicated multi-dimensional cases it is not possible to analytically evaluate the conditional expectation.

Instead, one can apply the Likelihood Ratio Method for the final step – I called this the "vibrato" method because of the uncertainty in the final value S_N

Need to read my paper for full details. Its main weakness is that the variance is $O(\Delta t^{-1/2})$, but it is much better than the $O(\Delta t^{-1})$ variance of the standard Likelihood Ratio Method, and you get the benefit of adjoints.

Malliavin calculus will give an $O(1)$ variance, but no adjoint efficiency gains, I think.

Conclusions

- adjoints can be very efficient for option pricing, calibration and sensitivity analysis
- same result as “standard” approach but a much lower computational cost
- basic elements of discrete adjoint analysis are very simple, although real applications can get quite complex
- automatic differentiation ideas are very important, even if you don’t use AD software

Further reading

M.B. Giles and P. Glasserman. ‘Smoking adjoints: fast Monte Carlo Greeks’, RISK, 19(1):88-92, 2006

M.B. Giles and P. Glasserman. ’Smoking Adjoints: fast evaluation of Greeks in Monte Carlo calculations’. Numerical Analysis report NA-05/15, 2005.

— *original RISK paper, and longer version with appendix on AD*

M. Leclerc, Q. Liang, I. Schneider. ’Fast Monte Carlo Bermudan Greeks’, RISK, 22(7):84-88, 2009.

L. Capriotti and M.B. Giles. ‘Fast correlation Greeks by adjoint algorithmic differentiation’, RISK, 23(5):77-83, 2010

— *correlation Greeks and binning*

L. Capriotti and M.B. Giles. ‘Algorithmic differentiation: adjoint Greeks made easy’, RISK, to appear, 2012

— *use of AD*

Further reading

M.B. Giles. ’Monte Carlo evaluation of sensitivities in computational finance’. Numerical Analysis report NA-07/12, 2007.
— *use of AD, and introduction of Vibrato idea*

M.B. Giles. ’Vibrato Monte Carlo sensitivities’. In Monte Carlo and Quasi-Monte Carlo Methods 2008, Springer, 2009.
— *Vibrato Monte Carlo for discontinuous payoffs*

C. Kaebe, J.H. Maruhn and E.W. Sachs. ’Adjoint-based Monte Carlo calibration of financial market models’. Finance and Stochastics, 13(3):351-379, 2009.
— *adjoint Monte Carlo sensitivities and calibration*

Further reading

M.B. Giles ‘On the iterative solution of adjoint equations’, pp.145-152 in Automatic Differentiation: From Simulation to Optimization, G. Corliss, C. Faure, A. Griewank, L. Hascoet, U. Naumann, editors, Springer-Verlag, 2001.

— *adjoint treatment of time-marching and fixed point iteration*

M.B. Giles. ’Collected matrix derivative results for forward and reverse mode algorithmic differentiation’. In Advances in Automatic Differentiation, Springer, 2008.

M.B. Giles. ’An extended collection of matrix derivative results for forward and reverse mode algorithmic differentiation’. Numerical Analysis report NA-08/01, 2008.

— *two papers on adjoint linear algebra, second has MATLAB code and tips on code development and validation*