“Smoking Adjoints” fast Monte Carlo Greeks

Mike Giles

mike.giles@maths.ox.ac.uk

“Smoking Adjoints” for fast Greeks

- joint work with Paul Glasserman – appeared in Risk magazine in 2006 and now being used by some leading banks
- a particularly efficient way of implementing the pathwise sensitivity approach
- builds on lots of well-established ideas in design optimisation and optimal control theory
- ideal when wanting the sensitivity of one output to changes in many different inputs
- guaranteed to give all first order derivatives for a total cost which is less than factor 4 greater than original calculation

Generic Problem

Suppose we have a multi-dimensional SDE with numerical approximation

$$\widehat{S}_{n+1} = g_n(\widehat{S}_n)$$

and we want to compute sensitivity of a European option

$$\mathbb{E}\left[f(\widehat{S}_M)\right]$$

to changes in S_0 and other input parameters.

Standard pathwise sensitivity

For the Deltas we can define

$$\widehat{s}_n = \frac{\partial \widehat{S}_n}{\partial S_0}$$

with $\widehat{s}_0 = I$, and differentiate each timestep evolution to get

$$\widehat{s}_{n+1} = G_n \, \widehat{s}_n$$

We then have (under certain conditions)

$$\frac{\partial}{\partial S_0} \, \mathbb{E}\left[f(\widehat{S}_M) \right] = \mathbb{E}\left[\frac{\partial f}{\partial \widehat{S}_M} \, \widehat{s}_M \right]$$

with

$$\frac{\partial f}{\partial \widehat{S}_M} \, \widehat{s}_M = \frac{\partial f}{\partial \widehat{S}_M} \, G_{M-1} \, G_{M-2} \, \ldots \, G_1 \, G_0$$

Crucial observation

Evaluating

$$\frac{\partial f}{\partial \widehat{S}_M}\, G_{M-1}\, G_{M-2}\, \ldots\, G_1\, G_0$$

from right to left involves a sequence of matrix-matrix products, each with $O(D^3)$ cost where D is the dimension of the SDE.

Alternatively, evaluating the <u>same</u> expression from left to right involves a sequence of vector-matrix products, each with $O(D^2)$ cost – big savings if D is large.

Important: get the <u>same</u> result either way, so still have usual differentiability requirements of pathwise sensitivity calc

Adjoint formulation

Starting with

$$v_M = \left(\frac{\partial f}{\partial \widehat{S}_M} \right)^T$$

the adjoint iteration is given by

$$v_n = G_n^T \, v_{n+1}$$

and we finish with

$$\frac{\partial}{\partial S_0} \, \mathbb{E} \left[f(\widehat{S}_M) \right] = \mathbb{E} \left[v_0^T \right]$$

Adjoint formulation

Note: we have to first do the path calculation, store everything needed for the G_n, then do the adjoint calculation of the sensitivity.

The storage requirements for a single path are minimal – the storage is then reused for the next path.

However, in PDE applications these storage issues can become significant.

Standard pathwise sensitivity

For the Vegas we can define

$$\widehat{s}_n = \frac{\partial \widehat{S}_n}{\partial \sigma}$$

with $\widehat{s}_0 = 0$, and differentiate each timestep evolution to get

$$\widehat{s}_{n+1} = G_n\, \widehat{s}_n + b_n, \qquad b_n \equiv \frac{\partial g_n}{\partial \sigma}$$

We then have

$$\frac{\partial f}{\partial \widehat{S}_M}\, \widehat{s}_M = \sum_{n=0}^{M-1} \frac{\partial f}{\partial \widehat{S}_M}\, G_{M-1}\, G_{M-2}\, \ldots\, G_{n+2}\, G_{n+1}\, b_n$$

Adjoint formulation

This can be re-expressed as

$$\frac{\partial f}{\partial \widehat{S}_M}\, \widehat{s}_M = \sum_{n=0}^{M-1} v_{n+1}^T b_n$$

where the adjoint variables v_n are as defined before.

Hence we finish with

$$\frac{\partial}{\partial \sigma}\, \mathbb{E}\left[f(\widehat{S}_M) \right] = \mathbb{E}\left[\sum_{n=0}^{M-1} v_{n+1}^T b_n \right]$$

Automatic Differentiation

The explanation above gives the essential ideas, but doesn’t explain the guarantee that all first-order derivatives of a single output can be computed at a cost no more than 4 times greater than the original computation

In addition, in real implementations you would not really store and use the matrices G_n

This brings us to an area of computer science research called **automatic differentiation** (or sometimes **algorithmic differentiation**)

Automatic Differentiation

A computer instruction creates an additional new value:

$$\mathbf{u}^n \; = \; \mathbf{f}^n(\mathbf{u}^{n-1}) \; \equiv \; \begin{pmatrix} \mathbf{u}^{n-1} \\ f_n(\mathbf{u}^{n-1}) \end{pmatrix},$$

A computer program is the composition of N such steps:

$$\mathbf{u}^N = \mathbf{f}^N \circ \mathbf{f}^{N-1} \circ \ldots \circ \mathbf{f}^2 \circ \mathbf{f}^1(\mathbf{u}^0).$$

Automatic Differentiation

In forward mode, differentiation w.r.t. one element of the input vector gives

$$\dot{\mathbf{u}}^n = D^n \, \dot{\mathbf{u}}^{n-1}, \quad D^n \equiv \begin{pmatrix} I^{n-1} \\ \partial f_n / \partial \mathbf{u}^{n-1} \end{pmatrix},$$

and hence

$$\dot{\mathbf{u}}^N = D^N \, D^{N-1} \ldots \, D^2 \, D^1 \, \dot{\mathbf{u}}^0$$

Automatic Differentiation

In reverse mode, we consider the sensitivity of the last element of the output vector (the final value we care about) to get

$$\left(\overline{\mathbf{u}}^{n-1}\right)^T \equiv \frac{\partial u_i^N}{\partial \mathbf{u}^{n-1}} = \frac{\partial u_i^N}{\partial \mathbf{u}^{n}} \, \frac{\partial \mathbf{u}^n}{\partial \mathbf{u}^{n-1}} = \left(\overline{\mathbf{u}}^{n}\right)^T D^n,$$

$$\Longrightarrow \quad \overline{\mathbf{u}}^{n-1} = \left(D^n\right)^T \overline{\mathbf{u}}^{n}.$$

and hence

$$\overline{\mathbf{u}}^{0} = \left(D^1\right)^T \left(D^2\right)^T \ldots \left(D^{N-1}\right)^T \left(D^N\right)^T \overline{\mathbf{u}}^{N}.$$

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

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

and so the reverse mode is

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

Automatic Differentiation

This gives a prescriptive algorithm for reverse mode differentiation.

Again the reverse mode is much more efficient if we want the sensitivity of a single output to multiple inputs.

Key result is that the cost of the reverse mode is at worst a factor 4 greater than the cost of the original calculation, regardless of how many sensitivities are being computed!

The storage of the D^n is minor for SDEs – much more of a concern for PDEs. There are also extra complexities when solving implicit equations through a fixed point iteration.

Automatic Differentiation

Manual implementation of the forward/reverse mode algorithms is possible but tedious.

Fortunately, automated tools have been developed, following one of two approaches:

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

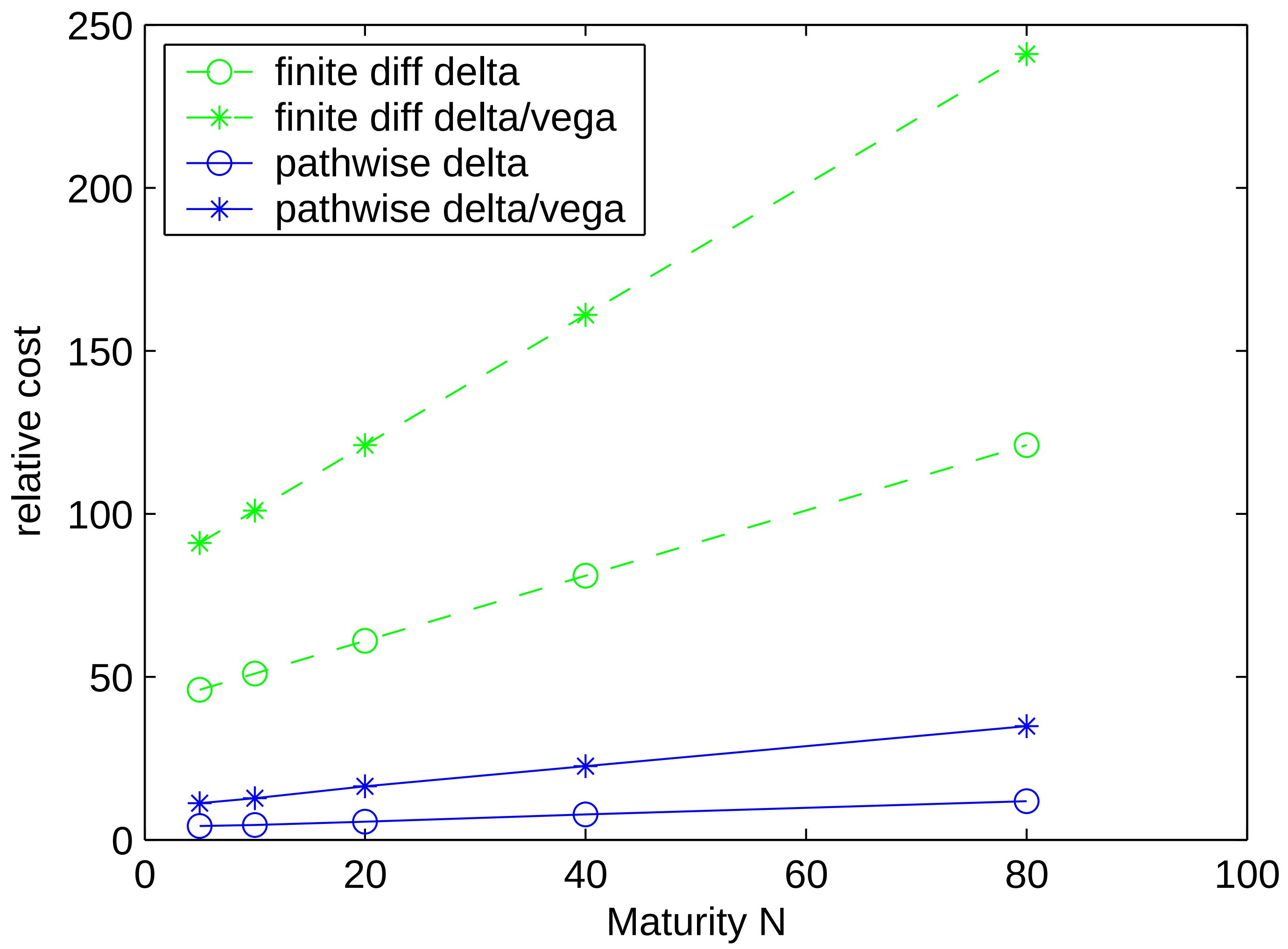
My personal experience is with Tapenade for Fortran, and FADBAD++ for C++. Both are easy to use, Tapenade is as efficient as hand-coded, FADBAD++ less so.

LIBOR Application

- testcase from “Smoking Adjoints” paper
- good real-world example involving stochastic evolution of future interest rates
- test problem performs N timesteps with a vector of $N{+}40$ forward rates, and computes the $N{+}40$ deltas and vegas for a portfolio of swaptions
- hand-coded adjoint for maximum efficiency – only about 50 lines of code so not too painful to do by hand

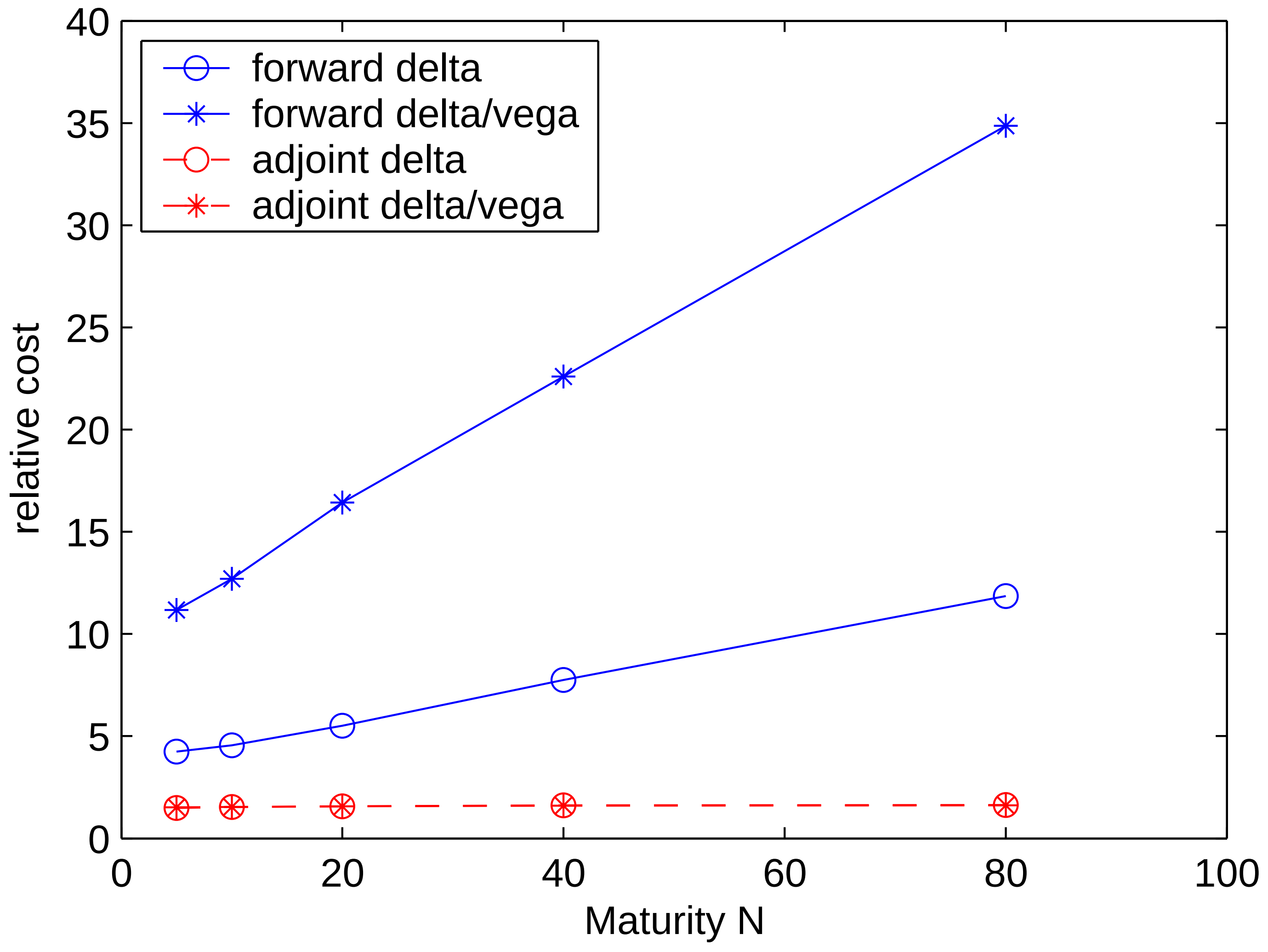
LIBOR Application

Finite differences versus forward pathwise sensitivities:



LIBOR Application

Hand-coded forward versus adjoint pathwise sensitivities:



Closing words

- the need for efficient Greeks means this research has been picked up quickly by the banks
- adjoint approach gives one level of differentiation for very little cost
- extending it to second order derivatives, can get Hessian matrix for usual cost of first order derivatives using pathwise approach
- however, this again raises the limitations due to differentiability – this is something I am working on with a “vibrato” MC idea which combines the best of both pathwise and LRM sensitivity calculations