Numerical Methods II

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

"Smoking Adjoints" for fast Greeks

- joint work with Paul Glasserman – appeared in Risk magazine in 2006 and now being used by quite a few 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 $\widehat{s}_n = \dfrac{\partial \widehat{S}_n}{\partial S_0}$ with $\widehat{s}_0 = I$, and differentiate each timestep evolution to get

$$\widehat{s}_{n+1} = D_n \, \widehat{s}_n, \qquad D_n \equiv \frac{\partial g(\widehat{S}_n)}{\partial \widehat{S}_n}$$

We then have (under the usual 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

$$\frac{\partial f}{\partial \widehat{S}_M} \, \widehat{s}_M = \frac{\partial f}{\partial \widehat{S}_M} \, D_{M-1} \, D_{M-2} \, \ldots \, D_1 \, D_0$$

Crucial observation

Evaluating

$$\frac{\partial f}{\partial \widehat{S}_M} \, D_{M-1} \, D_{M-2} \, \ldots \, D_1 \, D_0$$

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 same 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 same 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

$$v_n = D_n^T \; v_{n+1}$$

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 D_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

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

We then have

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

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 D_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:

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

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 w.r.t. one element of the input vector 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 consider the sensitivity of an output scalar v to get

$$(\overline{\mathbf{u}}_n)^T \equiv \frac{\partial v}{\partial \mathbf{u}_n} = \frac{\partial v}{\partial \mathbf{u}_{n+1}} \frac{\partial \mathbf{u}_{n+1}}{\partial \mathbf{u}_n} = \left(\overline{\mathbf{u}}_{n+1}\right)^T D_n,$$

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

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.

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:

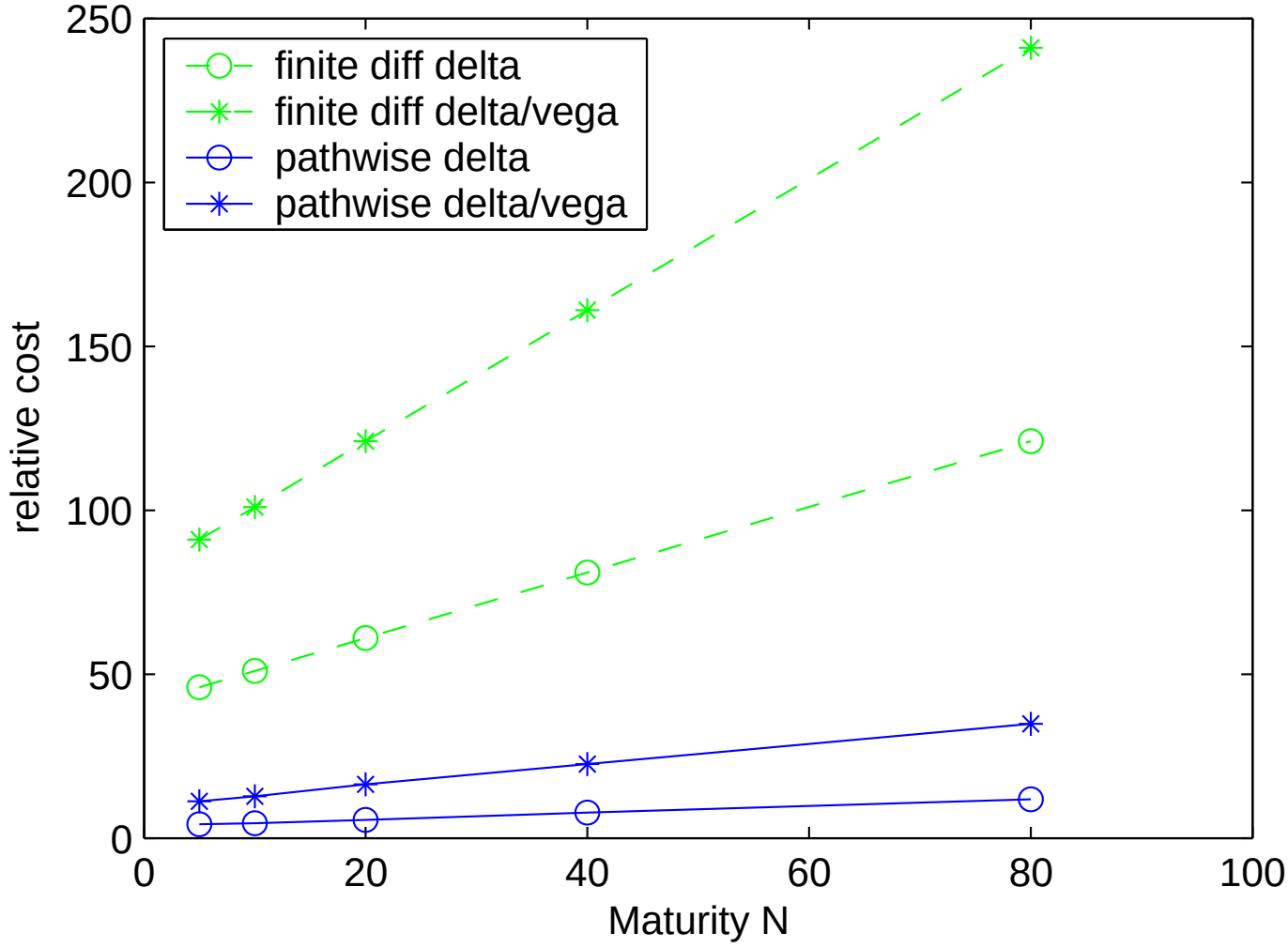
- operator overloading (ADOL-C, FADBAD++, ado)
- source code transformation (Tapenade, TAC++)

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

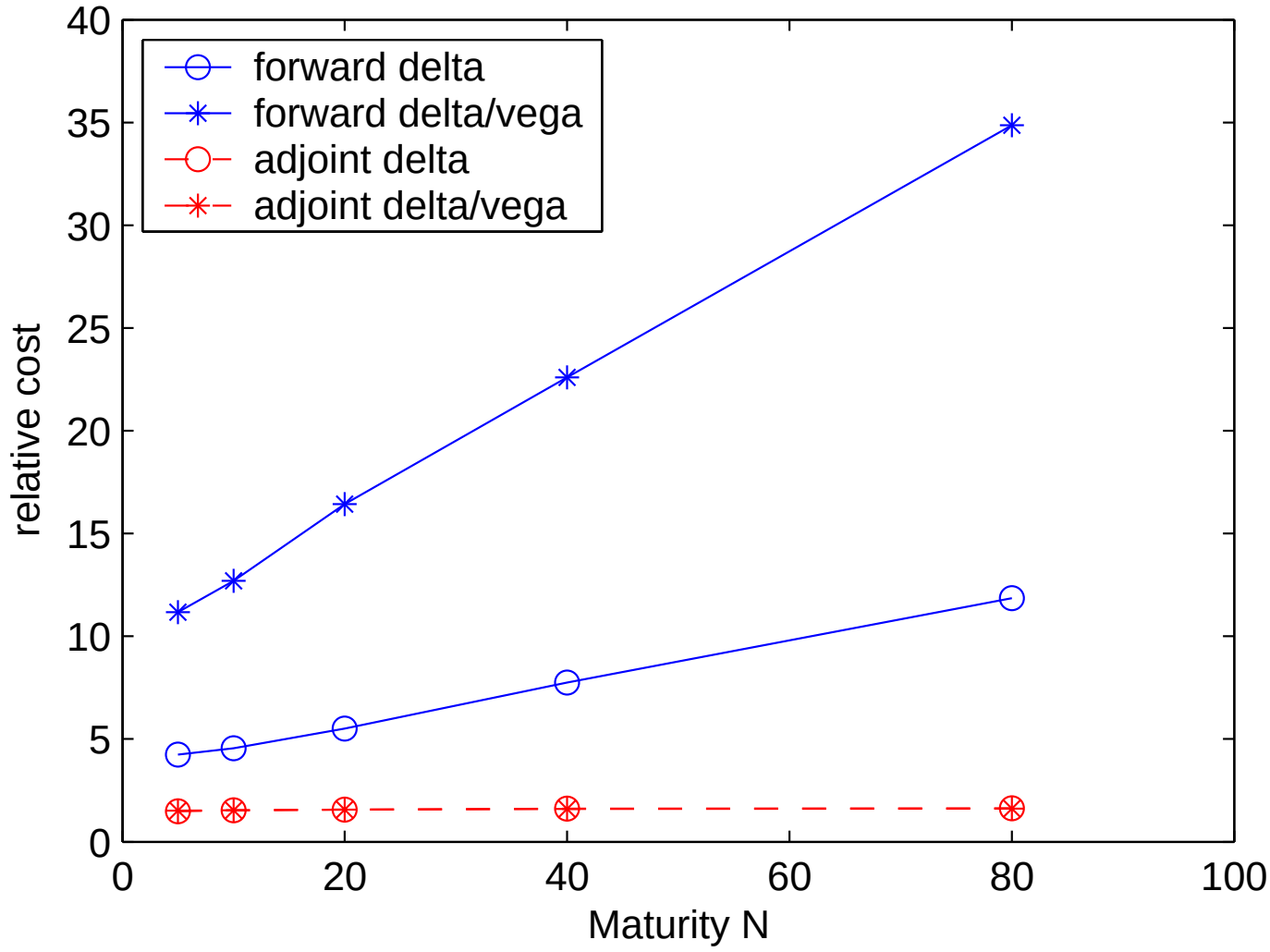
LIBOR Application

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 was picked up quite quickly by some banks
- adjoint approach gives one level of differentiation for very little cost
- for second order Greeks I would combine with "bumping"
- NAG's `ado` operator-overloading tool developed by Prof. Uwe Naumann at RTWH Aachen University is the leading approach used in the industry
- for more info on adjoints see my webpage: `people.maths.ox.ac.uk/gilesm/codes/libor_AD/`