Adjoint methods in computational finance

Mike Giles

Mathematical and Computational Finance Group,
Mathematical Institute, University of Oxford

Oxford-Man Institute of Quantitative Finance

12th Winter School on Mathematical Finance

Jan 21-23, 2013

Lecture outline

- Mathematical foundations
  - generic black-box approach
  - algorithmic differentiation
  - automatic differentiation software
  - adjoints for higher-level linear algebra
  - fixed-point iteration
- PDEs and finite difference methods:
  - adjoint PDEs and finite difference methods
  - vanilla pricing calculation
  - sensitivities for linear explicit discretisations
  - nonlinear implicit discretisations
  - what can go wrong?
  - calibration using Fokker-Planck discretisation
  - Greeks using Black-Scholes discretisation
  - local volatility example

Lecture outline

- SDEs and Monte Carlo methods:
  - Monte Carlo simulation
  - LRM and pathwise sensitivity approaches
  - adjoint pathwise approach
  - use of automatic differentiation software
  - multiple payoffs
  - binning and correlation Greeks
  - local volatility example, revisited
  - discontinuous payoffs

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, $10^5\times10^5$, $10^5\times10^5$.

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

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

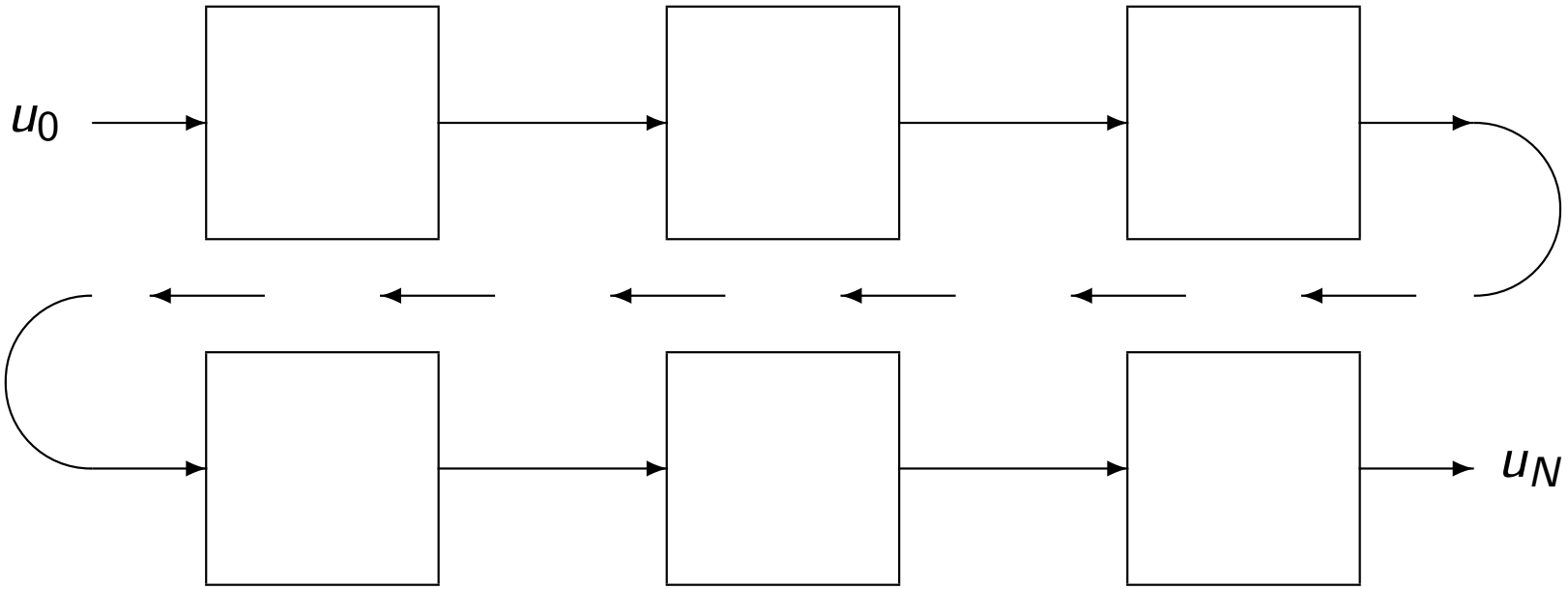
Answer 2: yes, in practice, if A, B, C have dimensions 1×10^5, $10^5\times10^5$, $10^5\times10^5$.

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

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.

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

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 (or at least enough data to compute $D_n^T \overline{u}_{n+1}$) 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:

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

and the final output of interest is the last element of $\mathbf{u}_N$.

Automatic differentiation

In forward mode, differentiation gives

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

and hence

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

This is relatively intuitive – but the next step is not.

Automatic differentiation

In reverse mode, we have

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

and hence

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

where the last element of $\overline{\mathbf{u}}_N$ is 1, and the rest are zero.

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.

$$\overline{a}_n = \overline{a}_{n+1} + \frac{\partial f}{\partial a}\,\overline{c}_{n+1} \qquad \longrightarrow \qquad \overline{a} \mathrel{+}= \frac{\partial f}{\partial a}\,\overline{c}$$

$$\overline{b}_n = \overline{b}_{n+1} + \frac{\partial f}{\partial b}\,\overline{c}_{n+1} \qquad \longrightarrow \qquad \overline{b} \mathrel{+}= \frac{\partial f}{\partial b}\,\overline{c}$$

mathematics computer program

Key observation: 1 multiply in original nonlinear calculation turns into 2 multiply-add operations in reverse mode, so at most the reverse mode costs only a factor 4 more than the original calculation.

(This ignores the cost of memory references.)

Automatic differentiation

A simple example with inputs a, b, c

$$\begin{aligned} d &:= a + b \\ e &:= c\ d \\ f &:= \exp(a) \\ g &:= e + f \end{aligned}$$

Automatic differentiation

A simple example with inputs a, b, c and $\dot{a}, \dot{b}, \dot{c}$

$$
\begin{aligned}
\dot{d} &:= \dot{a} + \dot{b} \\
d &:= a + b \\
\dot{e} &:= \dot{c}\ d + c\ \dot{d} \\
e &:= c\ d \\
\dot{f} &:= \exp(a)\ \dot{a} \\
f &:= \exp(a) \\
\dot{g} &:= \dot{e} + \dot{f} \\
g &:= e + f
\end{aligned}
$$

Automatic differentiation

In reverse mode, we get

$$
\begin{aligned}
d &:= a + b \\
e &:= c\ d \\
f &:= \exp(a) \\
g &:= e + f
\end{aligned}
$$

$$
\begin{aligned}
\overline{e} &+= \overline{g} \\
\overline{f} &+= \overline{g} \\
\overline{a} &+= \exp(a)\ \overline{f} \\
\overline{c} &+= d\ \overline{e} \\
\overline{d} &+= c\ \overline{e} \\
\overline{a} &+= \overline{d} \\
\overline{b} &+= \overline{d}
\end{aligned}
$$

Automatic differentiation tools

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

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

Operator overloading

In forward mode, define a new datatype for value x and sensitivity $\dot{x}$.

Then define operators accordingly:

$$\begin{pmatrix} x \\ \dot{x} \end{pmatrix} + \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x+y \\ \dot{x}+\dot{y} \end{pmatrix}$$

$$\begin{pmatrix} x \\ \dot{x} \end{pmatrix} * \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x\,y \\ y\,\dot{x}+x\,\dot{y} \end{pmatrix}$$

$$\begin{pmatrix} x \\ \dot{x} \end{pmatrix} / \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x/y \\ (1/y)\,\dot{x} - (x/y^2)\,\dot{y} \end{pmatrix}$$

$$\exp\begin{pmatrix} x \\ \dot{x} \end{pmatrix} = \begin{pmatrix} \exp(x) \\ \exp(x)\,\dot{x} \end{pmatrix}$$

Note: this works as well if the sensitivity $\dot{x}$ is a vector so we compute several different sensitivities at the same time.

Operator overloading

Operator overloading doesn't seem to extend naturally to reverse mode.

What is does is to make a record (referred to as a "tape") of all operations performed in the original calculation, and their input operands.

Then, it is possible to work backwards through the tape performing the necessary adjoint calculations.

Typically, the performance is not great (too much data being stored and then brought back) but it can be very useful for code validation.

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

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

Source code transformation

Note that for any choice of $\dot{u}$ and $\overline{v}$,

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

The left hand side comes from the forward mode code which computes $\dot{v}$; the right hand side comes from the reverse mode code which computes $\overline{u}$.

Checking this equality is an important validation step when developing forward and reverse mode sensitivity code, especially when it is done by hand, not through automatic differentation.

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

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

Linear algebra sensitivities

For matrices/vectors A and B of compatible dimensions,

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

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

$$C = A^{-1}$$
$$\dot{C} = -\,C\,\dot{A}\,C$$
$$\overline{A} = -C^T\,\overline{C}\,C^T$$

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

Linear algebra sensitivities

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

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

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

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

this gives a dense matrix $\overline{A}$, at $O(n^2)$ cost – since A is tri-diagonal then only the tri-diagonal elements of $\overline{A}$ need to 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 results are 30-40 years old, but some 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 a Newton iteration, P is the inverse Jacobian, but P could also correspond to a multigrid cycle in an iterative solver – this is relevant to multi-dimensional finite difference methods.

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

$$\overline{u} = -\left(\frac{\partial f}{\partial u}\right)^{T} \left(\left(\frac{\partial f}{\partial v}\right)^{T}\right)^{-1} \overline{v} = \left(\frac{\partial f}{\partial u}\right)^{T} \overline{w}$$

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

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

Final comments

- forward mode sensitivity analysis is just classic linear sensitivity analysis – very intuitive
- reverse mode adjoint sensitivity analysis computes exactly the same value, but does so much more efficiently when you want the sensitivity of one output to many inputs
- the adjoint approach is not intuitive, so don't worry if it seems strange – trust the mathematics, and proceed slowly