Stochastic Simulation: Lecture 4

Prof. Mike Giles

Oxford University Mathematical Institute

IPA

The third approach is called pathwise sensitivities in finance, or IPA (infinitesimal perturbation analysis) in other settings.

We start by expressing the expectation as an integral w.r.t. the random inputs. If these are uncorrelated Normals, then

$$V(\theta) \equiv \mathbb{E}\left[f(\theta, Z)\right] = \int f(\theta; Z)\ p_Z(Z)\ \mathrm{d}Z$$

where $p_Z(Z)$ is the joint Normal probability distribution, and differentiate this to get

$$\frac{\partial V}{\partial \theta} = \int \frac{\partial f}{\partial \theta}\ p_Z\ \mathrm{d}Z = \mathbb{E}\left[\frac{\partial f}{\partial \theta}\right]$$

with $\partial f/\partial\theta$ being evaluated at fixed Z.

Note: this needs $f(\theta, Z)$ to be differentiable w.r.t. θ but can prove it's OK provided $f(\theta, Z)$ is continuous and piecewise differentiable

IPA

This leads to the estimator

$$\frac{1}{N}\sum_{i=1}^{N}\frac{\partial f^{(i)}}{\partial \theta}$$

which is the derivative of the usual price estimator

$$\frac{1}{N}\sum_{i=1}^{N} f^{(i)}$$

Can give incorrect estimates when $f(\theta, Z)$ is discontinuous.

e.g. for indicator function $f = \mathbf{1}_{Z>\theta}$, $\partial f/\partial\theta = 0$ so estimated derivative is zero – clearly wrong.

IPA

Extension to second derivatives is straightforward

$$\frac{\partial^2 V}{\partial \theta^2} = \int \frac{\partial^2 f}{\partial \theta^2} \, p_Z \, \mathrm{d}Z$$

with $\partial^2 f / \partial \theta^2$ also being evaluated at fixed Z.

However, this requires $f(\theta, Z)$ to have a continuous first derivative – a problem in practice in many finance applications

IPA

In a simple finance case with a single underlying asset we have

$$S(T) = S(0)\ \exp\left((r - \tfrac{1}{2}\sigma)T + \sigma Z\right)$$

so

$$\log S(T) = \log S(0) + (r - \tfrac{1}{2}\sigma^2)T + \sigma Z$$

and hence

$$\frac{1}{S(T)}\ \frac{\partial S(T)}{\partial \theta} = \frac{1}{S(0)}\ \frac{\partial S(0)}{\partial \theta} + \left(\frac{\partial r}{\partial \theta} - \sigma \frac{\partial \sigma}{\partial \theta}\right) T + \frac{\partial \sigma}{\partial \theta} Z$$

and then

$$\frac{\partial f}{\partial \theta} = \frac{\partial f}{\partial S(T)}\ \frac{\partial S(T)}{\partial \theta}$$

IPA

Extension to multivariate case is straightforward

$$S_k(T) = S_k(0)\ \exp\left((r-\tfrac{1}{2}\sigma_k^2)T + \sum_l L_{kl}Z_l\right)$$

so

$$\log S_k(T) = \log S_k(0) + (r-\tfrac{1}{2}\sigma_k^2)T + \sum_l L_{kl}Z_l$$

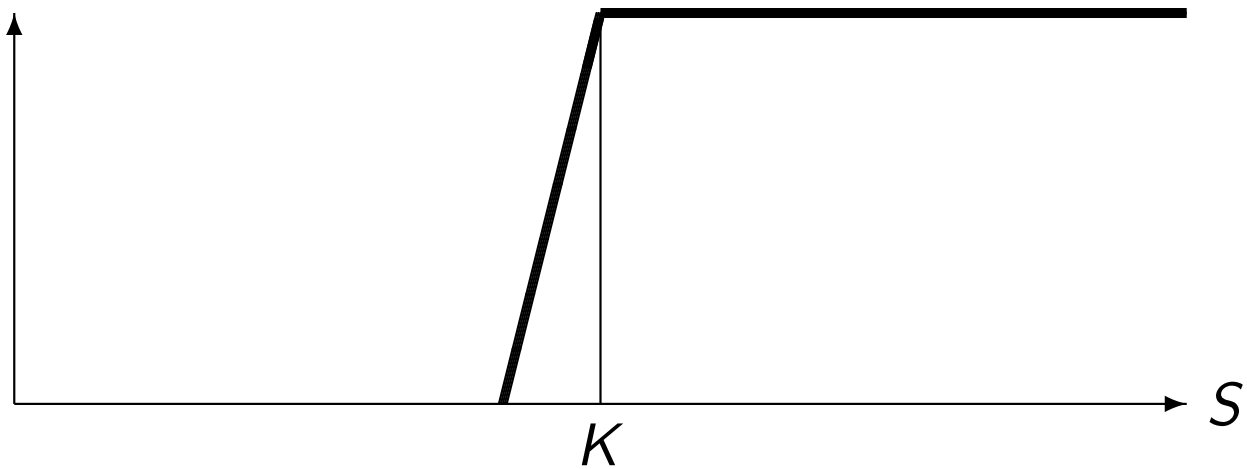
and hence

$$\frac{1}{S_k(T)}\frac{\partial S_k(T)}{\partial\theta} = \frac{1}{S_k(0)}\frac{\partial S_k(0)}{\partial\theta} + \left(\frac{\partial r}{\partial\theta} - \sigma_k\frac{\partial\sigma_k}{\partial\theta}\right)T + \sum_l \frac{\partial L_{kl}}{\partial\theta}Z_l$$

IPA

To handle output functions which do not have the necessary continuity/smoothness one can modify the payoff

In finance it is common to use a piecewise linear approximation which is fine for first order sensitivities.



IPA

The standard call option definition can be smoothed by integrating the smoothed Heaviside function

$$H_\varepsilon(S-K) = \Phi\left(\frac{S-K}{\varepsilon}\right)$$

with $\varepsilon \ll K$, to get

$$f(S) = (S-K)\ \Phi\left(\frac{S-K}{\varepsilon}\right) + \frac{\varepsilon}{\sqrt{2\pi}} \exp\left(-\frac{(S-K)^2}{2\,\varepsilon^2}\right)$$

This will allow the calculation of first and second order derivatives

IPA

- IPA is usually the best approach (simplest, lowest variance and least cost) when it is applicable – needs continuous output function for first derivatives
- function smoothing can be used to make IPA applicable to discontinuous functions and for second derivatives
- alternatively, combine IPA with finite differences for second derivatives
- another benefit of LRM and IPA is that you can compute almost everything in single precision – at least twice as fast on CPUs and GPUs
  (Warning: use double precision when averaging!)
- extra benefit of IPA is very efficient adjoint implementation when many different first order sensitivities are needed

A question!

Given matrices A, B, C is $(A\,B)\,C$ equivalent to $A\,(B\,C)$?

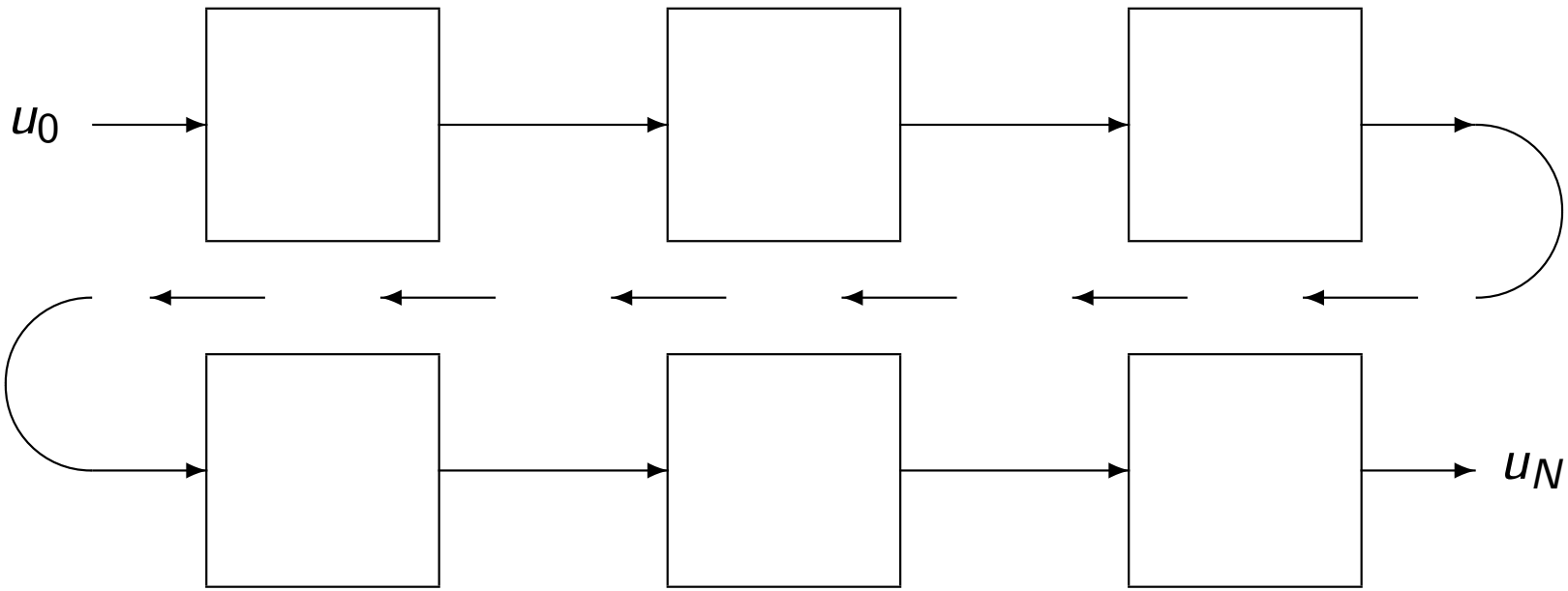
Answer 1: yes, in theory, and also in practice if A, B, C are square

Answer 2: no, in practice, if A, B, C have dimensions 1×10^4, $10^4 \times 10^4$, $10^4 \times 10^4$.

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

Generic black-box problem

An input vector u_0 leads to a scalar output u_N:



Each black-box could be a mathematical step,
or a computer code, or even a single computer instruction

Assumption: each step is 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
- each element of u_0 requires its own sensitivity calculation; cost proportional to number of inputs

Adjoint "reverse mode" approach effectively multiplies from left to right by evaluating the transpose from right to left

$$\left(D_{N-1} \; D_{N-2} \; \ldots \; D_1 \; D_0 \right)^T = D_0^T \; D_1^T \; \ldots \; D_{N-2}^T \; D_{N-1}^T$$

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

This is all the same mathematics as back-propagation in machine learning!

It looks easy (?) – what's the catch?

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

A warning

Suppose our analytic problem with input x has solution

$$u = x$$

and our discrete approximation with step size h is

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

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

I have actually seen problems like this in real applications

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.

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!

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, TAF/TAC++, ADIFOR)

Operator overloading

- define new datatype and associated arithmetic operators
- very natural for forward mode, but also works for reverse mode

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

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

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

Source code transformation

- programmer supplies 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

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

Numerical differentiation

Suppose we have MATLAB code to compute $f(x)$ (with x and $f(x)$ both scalar) and we want to compute the derivative $f'(x)$.

What can we do? Performing a Taylor series expansion,

$$f(x+\Delta x) \approx f(x) + \Delta x\, f'(x) + \tfrac{1}{2}\Delta x^2\, f''(x) + \tfrac{1}{6}\Delta x^3\, f'''(x)$$

$$\begin{aligned}
\Longrightarrow \qquad \frac{f(x+\Delta x) - f(x)}{\Delta x} &\approx f'(x) + \tfrac{1}{2}\Delta x\, f''(x), \\
\frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} &\approx f'(x) + \tfrac{1}{6}\Delta x^2\, f'''(x), \\
\frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{\Delta x^2} &\approx f''(x) + \tfrac{1}{24}\Delta x^2\, f''''(x).
\end{aligned}$$

Numerical differentiation

These are finite difference approximations, and they are the basis for the finite difference method for approximating PDEs.

In Monte Carlo methods, we use similar ideas (often referred to as "bumping") for computing sensitivities (the "Greeks" in finance)

The problem with taking $\Delta x \ll 1$ is inaccuracy due to finite precision arithmetic, in which there is a relative rounding error of size 2^{-S} where S is the size of the mantissa.

Numerical differentiation

Error in computing $f(x+\Delta x) - f(x)$ is roughly of size $2^{-S} f(x)$, so error in computing one-sided difference estimate for $f'(x)$ is of order

$$\frac{2^{-S} f(x)}{2\Delta x}$$

while the finite difference error is $O(\Delta x)$.

To balance errors, want

$$\frac{2^{-S}}{\Delta x} \sim \Delta x \quad \Longrightarrow \quad \Delta x \sim 2^{-S/2}.$$

In single precision, this means taking $\Delta x \sim 10^{-3}$, and getting an error which is roughly of size 10^{-3}. This is not great, and making Δx smaller or bigger will make things worse.

This is why many people use double precision when doing "bumping" for sensitivity analysis.

Complex Variable Trick

This is a very useful "trick", which I learned about from this very short article:

> *"Using Complex Variables to Estimate Derivatives of Real Functions", William Squire and George Trapp, SIAM Review, 40(1):110-112, 1998.*

which now has 465 citations.

Complex Variable Trick

Suppose $f(z)$ is a complex analytic function, and $f(x)$ is real when x is real.

Then

$$f(x+i\,\Delta x) \approx f(x) + i\,\Delta x\, f'(x) - \tfrac{1}{2}\Delta x^2\, f''(x) - i\,\tfrac{1}{6}\Delta x^3\, f'''(x)$$

and hence

$$\frac{\mathrm{Im} f(x+i\,\Delta x)}{\Delta x} \approx f'(x) - \tfrac{1}{6}\Delta x^2\, f'''(x)$$

Now, we can take $\Delta x \ll 1$, and there is no problem due to finite precision arithmetic.

I typically use $\Delta x = 10^{-10}$!

Complex Variable Trick

There are just a few catches, because $f(z)$ must be analytic:

- need analytic extensions for $\min(x, y)$, $\max(x, y)$ and $|x|$
- need analytic extensions to certain functions, e.g. MATLAB's `normcdf`
- in MATLAB, must be aware that A' is the Hermitian of A (complex conjugate transpose), so use $A.'$ for the simple transpose.

Using this, can very simply "differentiate" almost any MATLAB or C/C++ code for a real function $f(x)$.