Monte Carlo estimation of Greeks
(Smoking Adoints: parts 2 and 3)

Mike Giles

giles@comlab.ox.ac.uk

Oxford University Computing Laboratory
Oxford-Man Institute of Quantitative Finance

Acknowledgment: research funding from Microsoft and EPSRC
“Smoking Adjoint”

Paper with Paul Glasserman in *Risk* in 2006 on the use of adjoints in computing pathwise sensitivities attracted a lot of interest, and questions:

- what is involved in practice in creating an adjoint code, and can it be simplified?
- do we really have to differentiate the payoff?
- what about non-differentiable payoffs?
- what about American options? (NOT addressed yet!)
Outline

- different approaches to computing Greeks
- adjoint pathwise sensitivities
- use of automatic differentiation
- “vibrato” Monte Carlo for non-differentiable payoffs
Generic Problem

Stochastic differential equation with general drift and volatility terms:

\[ dS(t) = a(S, t) \, dt + b(S, t) \, dW(t) \]

For a simple European option we want to compute the expected discounted payoff value dependent on the terminal state:

\[ V = \mathbb{E}[f(S(T))] \]

Note: the drift and volatility functions are almost always differentiable, but the payoff \( f(S) \) is often not.
Euler discretisation with timestep $h$:

$$
\hat{S}_{n+1} = \hat{S}_n + a(\hat{S}_n, t_n) h + b(\hat{S}_n, t_n) \Delta W_n
$$

Simplest Monte Carlo estimator for expected payoff is an average of $M$ independent path simulations:

$$
M^{-1} \sum_{i=1}^{M} f(\hat{S}^{(i)}_{T/h})
$$

Greeks: for hedging and risk management we also want to estimate derivatives of expected payoff $V$
Simple Problem

For Geometric Brownian motion

\[ dS(t) = r \, S \, dt + \sigma \, S \, dW(t) \]

the SDE can be solved analytically to give

\[ S(T) = S_0 \exp \left( (r - \frac{1}{2} \sigma^2) T + \sigma W(T) \right) \]

In this case, we can directly sample \( W(T) \) to get

\[ V \equiv \mathbb{E} [f(S(T))] \approx M^{-1} \sum_{i=1}^{M} f(S^{(i)}) \]

– will use this to explain approaches to calculating sensitivities
Evaluating sensitivities

Simplest approach is to use a finite difference approximation,

\[
\frac{\partial V}{\partial \theta} \approx \frac{V(\theta + \Delta \theta) - V(\theta - \Delta \theta)}{2 \Delta \theta}
\]

\[
\frac{\partial^2 V}{\partial \theta^2} \approx \frac{V(\theta + \Delta \theta) - 2V(\theta) + V(\theta - \Delta \theta)}{(\Delta \theta)^2}
\]

– very simple, but expensive and inaccurate if \(\Delta \theta\) is too big or too small
Evaluating sensitivities

For simple cases where we know the terminal probability distribution and so

\[ V \equiv \mathbb{E} [f(S(T))] = \int f(S) \, p_S(S) \, dS \]

we can differentiate this to get

\[ \frac{\partial V}{\partial \theta} = \int f \, \frac{\partial p_S}{\partial \theta} \, dS = \int f \, \frac{\partial (\log p_S)}{\partial \theta} \, p_S \, dS = \mathbb{E} \left[ f \, \frac{\partial (\log p_S)}{\partial \theta} \right] \]

This is the Likelihood Ratio Method – can handle non-differentiable payoffs, but doesn’t generalise well to cases where we have to simulate the whole path.
Evaluating sensitivities

Alternatively, for simple Geometric Brownian Motion

\[ V \equiv \mathbb{E} [f(S(T))] = \int f(S(T)) \ p_W(W) \ dW \]

and differentiating this gives

\[ \frac{\partial V}{\partial \theta} = \int \frac{\partial f}{\partial S} \ \frac{\partial S(T)}{\partial \theta} \ p_W \ dW = \mathbb{E} \left[ \frac{\partial f}{\partial S} \ \frac{\partial S(T)}{\partial \theta} \right] \]

with \( \frac{\partial S(T)}{\partial \theta} \) being evaluated at fixed \( W \).

This is the pathwise sensitivity approach – it can’t handle non-differentiable payoffs, but does generalise well to cases where we have to simulate the whole path.
Evaluating sensitivities

The generalisation involves differentiating the Euler path discretisation,

\[ \hat{S}_{n+1} = \hat{S}_n + a(\hat{S}_n, t_n) h + b(\hat{S}_n, t_n) \Delta W_n \]

holding fixed the Brownian increments, to get

\[ \frac{\partial \hat{S}_{n+1}}{\partial \theta} = \left( 1 + \frac{\partial a}{\partial S} h + \frac{\partial b}{\partial S} \Delta W_n \right) \frac{\partial \hat{S}_n}{\partial \theta} + \frac{\partial a}{\partial \theta} h + \frac{\partial b}{\partial \theta} \Delta W_n \]

leading to

\[ \frac{\partial \hat{V}}{\partial \theta} = M^{-1} \sum_{i=1}^{M} \frac{\partial f}{\partial S}(\hat{S}_N^{(i)}) \frac{\partial \hat{S}_N^{(i)}}{\partial \theta}. \]
Adjoint sensitivities

The adjoint approach is an efficient implementation of pathwise sensitivities.

Consider a process in which a vector input $\alpha$ leads to a final state vector $S$ which is used to compute a scalar payoff $P$

$$
\alpha \rightarrow S \rightarrow P
$$

Taking $\dot{\alpha}, \dot{S}, \dot{P}$ to be the derivatives w.r.t. $j^{th}$ component of $\alpha$, then

$$
\dot{S} = \frac{\partial S}{\partial \alpha} \dot{\alpha}, \quad \dot{P} = \frac{\partial P}{\partial S} \dot{S},
$$

and hence

$$
\dot{P} = \frac{\partial P}{\partial S} \frac{\partial S}{\partial \alpha} \dot{\alpha}.
$$
Adjoint sensitivities

Alternatively, defining $\overline{\alpha}, \overline{S}, \overline{P}$ to be the derivatives of $P$ with respect to $\alpha, S, P$, then

$$\overline{\alpha} \overset{\text{def}}{=} \left( \frac{\partial P}{\partial \alpha} \right)^T = \left( \frac{\partial P}{\partial S} \frac{\partial S}{\partial \alpha} \right)^T = \left( \frac{\partial S}{\partial \alpha} \right)^T \overline{S},$$

and similarly

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

giving

$$\overline{\alpha} = \left( \frac{\partial S}{\partial \alpha} \right)^T \left( \frac{\partial P}{\partial S} \right)^T \overline{P}.$$
Adjoint sensitivities

The two are mathematically equivalent, since

\[ \dot{P} = \frac{\partial P}{\partial \alpha} \dot{\alpha} = \bar{\alpha}^T \dot{\alpha} = \bar{\alpha}_j \]

but the adjoint approach approach is much cheaper because a single calculation gives \( \bar{\alpha} \), the sensitivity of \( P \) to each one of the elements of \( \alpha \).

- standard approach: cost proportional to number of Greeks
- adjoint approach: cost independent
- crossover point for cost: 4 – 6 Greeks?
Adjoint sensitivities

Note that the standard approach goes forward

\[ \dot{\alpha} \rightarrow \dot{S} \rightarrow \dot{P} \]

while the adjoint approach does the reverse

\[ \overline{\alpha} \leftarrow \overline{S} \leftarrow \overline{P}. \]

These correspond to the forward and reverse modes of AD (Automatic Differentiation). “Smoking Adjoint” paper extended this to multiple timesteps in the path calculation | instead, we’ll now extend it to the steps in a whole computer program.
Automatic Differentiation

A computer instruction creates an additional new value:

\[
\begin{align*}
    u^n &= f^n(u^{n-1}) \equiv \begin{pmatrix} u^{n-1} \\ f_n(u^{n-1}) \end{pmatrix},
\end{align*}
\]

A computer program is the composition of \( N \) such steps:

\[
    u^N = f^N \circ f^{N-1} \circ \ldots \circ f^2 \circ f^1(u^0).
\]
Automatic Differentiation

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

\[ \hat{u}^n = D^n \hat{u}^{n-1}, \quad D^n \equiv \begin{pmatrix} I^{n-1} \\ \partial f_n / \partial u^{n-1} \end{pmatrix}, \]

and hence

\[ \hat{u}^N = D^N D^{N-1} \ldots D^2 D^1 \hat{u}^0 \]
In reverse mode, we consider the sensitivity of one element of the output vector, to get

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

\[\implies \overline{u}^{n-1} = (D^n)^T \overline{u}^n.\]

and hence

\[
\overline{u}^0 = (D^1)^T (D^2)^T \ldots (D^{N-1})^T (D^N)^T \overline{u}^N.
\]

Note: need to go forward through original calculation to compute/store the \(D^n\), then go in reverse to compute \(\overline{u}^n\).
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

\[
\begin{pmatrix}
\bar{a} \\
\bar{b} \\
\bar{c}
\end{pmatrix}^{n-1} = \begin{pmatrix}
1 & 0 & \frac{\partial f}{\partial a} \\
0 & 1 & \frac{\partial f}{\partial b}
\end{pmatrix} \begin{pmatrix}
\bar{a} \\
\bar{b} \\
\bar{c}
\end{pmatrix}^n
\]
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.
Operator overloading

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

\[
x + \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x + y \\ \dot{y} \end{pmatrix} \quad \quad \begin{pmatrix} x \\ \dot{x} \end{pmatrix} + \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x + y \\ \dot{x} + \dot{y} \end{pmatrix}
\]

\[
x \times \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x \times y \\ x \times \dot{y} \end{pmatrix} \quad \quad \begin{pmatrix} x \\ \dot{x} \end{pmatrix} \times \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x \times y \\ \dot{x} \times y + x \times \dot{y} \end{pmatrix}
\]

\[
x / \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x / y \\ -(x / y^2) \times \dot{y} \end{pmatrix} \quad \quad \begin{pmatrix} x \\ \dot{x} \end{pmatrix} / \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x / y \\ \dot{x} / y - (x / y^2) \times \dot{y} \end{pmatrix}
\]
template<typename ADdouble>
void path_calc(const int N, const int Nmat, const double delta,
              ADdouble L[], const double lambda[], const double z[])
{
  int i, n;
  double sqez, lam, con1;
  ADdouble v, vrat;

  for(n=0; n<Nmat; n++) {
    sqez = sqrt(delta)*z[n];

    v = 0.0;
    for (i=n+1; i<N; i++) {
      lam = lambda[i-n-1];
      con1 = delta*lam;
      v += (con1*L[i])/(1.0+delta*L[i]);
      vrat = exp(con1*v + lam*(sqez-0.5*con1));
      L[i] = L[i]*vrat;
    }
  }
}
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

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

- testcase from “Smoking Adjoint” paper
- 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
- originally hand-coded (using the ideas from AD), now used to test the effectiveness of AD tools
Finite differences versus forward pathwise sensitivities:

![Graph showing relative cost vs. Maturity N for finite difference and pathwise methods](image-url)
LIBOR Application

Hand-coded forward versus adjoint pathwise sensitivities:

![Graph showing relative cost versus Maturity N for different sensitivities]

- Forward delta
- Forward delta/vega
- Adjoint delta
- Adjoint delta/vega
LIBOR Application

Timings per path for $N = 40$; the hybrid version uses hand-coded for the path and FADBAD++ for the payoff

<table>
<thead>
<tr>
<th>milliseconds/path</th>
<th>Gnu g++</th>
<th>Intel icc</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>0.37</td>
<td>0.10</td>
</tr>
<tr>
<td>hand-coded forward</td>
<td>0.97</td>
<td>0.52</td>
</tr>
<tr>
<td>hand-coded reverse</td>
<td>0.47</td>
<td>0.19</td>
</tr>
<tr>
<td>FADBAD++ forward</td>
<td>4.30</td>
<td>5.00</td>
</tr>
<tr>
<td>FADBAD++ reverse</td>
<td>6.20</td>
<td>4.86</td>
</tr>
<tr>
<td>hybrid forward</td>
<td>1.02</td>
<td>0.63</td>
</tr>
<tr>
<td>hybrid reverse</td>
<td>0.65</td>
<td>0.35</td>
</tr>
<tr>
<td>TAC++ forward</td>
<td>1.36</td>
<td>0.85</td>
</tr>
<tr>
<td>TAC++ reverse</td>
<td>1.28</td>
<td>0.45</td>
</tr>
</tbody>
</table>
Conclusions?

- Hand-coded is clearly most efficient. I optimised the implementation (tradeoff between storage versus recomputation) in a way the AD tools cannot yet.

- However, AD code is very useful for debugging/validating hand-coded version, and can be used for bits which are not computationally intensive.

- AD is likely to be useful for bigger applications (vital in computational science and engineering).
Vibrato Monte Carlo

One remaining problem – what if payoff is not differentiable?

- LRM
  - estimator variance proportional to $h^{-1}$
- Malliavin calculus
  - recent paper by Glasserman and Chen shows it can be viewed as a pathwise/LRM hybrid
  - might be good choice when few Greeks needed
- new “vibrato” Monte Carlo idea
  - also a pathwise/LRM hybrid
  - variance proportional to $h^{-1/2}$
  - efficient adjoint implementation
Vibrato Monte Carlo

- new idea, based on Glasserman example of conditional expectation for a simple digital option
- output of each SDE path calculation becomes a narrow (multivariate) Normal distribution
- combine pathwise sensitivity for the differentiable SDE, with LRM for the non-differentiable payoff
- avoiding the differentiation of the payoff also simplifies the implementation in real-world setting
Vibrato Monte Carlo

Final timestep of Euler path discretisation is

\[ \hat{S}_N = \hat{S}_{N-1} + a(\hat{S}_{N-1}, t_{N-1}) h + b(\hat{S}_{N-1}, t_{N-1}) \Delta W_{N-1} \]

Instead of using random number generator to get a value for \( \Delta W_{N-1} \), consider the whole distribution of possible values, so \( \hat{S}_N \) has a Normal distribution with mean

\[ \mu(W) = \hat{S}_{N-1} + a(\hat{S}_{N-1}, t_{N-1}) h \]

and standard deviation

\[ \sigma(W) = b(\hat{S}_{N-1}, t_{N-1}) \sqrt{h} \]

where \( W \equiv (\Delta W_0, \Delta W_1, \ldots, \Delta W_{N-2}) \).
Vibrato Monte Carlo

For a particular path given by a particular vector $W$, the expected payoff is

$$\mathbb{E}_Z[f(\mu + \sigma Z)]$$

where $Z$ is a Normal random variable with zero mean and unit variance.

Averaging over all $W$ then gives the same overall expectation as before.

Note also that, for given $W$, $\widehat{S}_N$ has a Normal distribution with

$$p_S(\widehat{S}_N) = \frac{1}{\sqrt{2\pi \sigma}} \exp\left(-\frac{(\widehat{S}_N - \mu)^2}{2\sigma^2}\right)$$
Vibrato Monte Carlo

In the case of a simple digital call with strike $K$, Glasserman uses the analytic solution

$$E_Z[f(\mu + \sigma Z)] = \exp(-rT) \Phi \left( \frac{\mu - K}{\sigma} \right).$$

- for each $W$, the payoff is now smooth, differentiable
- derivative is $O(h^{-1/2})$ near strike, near zero elsewhere
  $\implies$ variance is $O(h^{-1/2})$
- analytic evaluation of conditional expectation not possible in general for multivariate cases
  $\implies$ use Monte Carlo estimation!
Vibrato Monte Carlo

Main novelty comes in calculating the sensitivity.

For a particular $W$, we have a Normal probability distribution for $\hat{S}_N$ and can apply the Likelihood Ratio method to get

$$\frac{\partial}{\partial \theta} \mathbb{E}_Z \left[ f(\hat{S}_N) \right] = \mathbb{E}_Z \left[ f(\hat{S}_N) \frac{\partial (\log p_S)}{\partial \theta} \right],$$

where

$$\frac{\partial (\log p_S)}{\partial \theta} = \frac{\partial (\log p_S)}{\partial \mu} \frac{\partial \mu}{\partial \theta} + \frac{\partial (\log p_S)}{\partial \sigma} \frac{\partial \sigma}{\partial \theta}$$

$$= \frac{Z}{\sigma} \frac{\partial \mu}{\partial \theta} + \frac{Z^2 - 1}{\sigma} \frac{\partial \sigma}{\partial \theta}.$$

Averaging over all $W$ then gives the expected sensitivity.
To improve the variance, we note that

\[ \mathbb{E}[1] = 1 \implies \mathbb{E}_Z \left[ \frac{\partial (\log p_S)}{\partial \theta} \right] = 0 \]

and hence

\[ \frac{\partial}{\partial \theta} \mathbb{E}_Z \left[ f(\hat{S}_N) \right] = \mathbb{E}_Z \left[ \left( f(\mu + \sigma Z) - f(\mu) \right) \frac{\partial (\log p_S)}{\partial \theta} \right]. \]

The quantity

\[ \hat{P} = \left( f(\mu + \sigma Z) - f(\mu) \right) \frac{\partial (\log p_S)}{\partial \theta} \]

has \( O(1) \) variance when \( f(S) \) is Lipschitz.
In the multivariate extension with mean vector $\mu$ and covariance matrix $\Sigma$,

$$\hat{S}(W, Z) = \mu + C Z$$

where $\Sigma = C C^T$ and $Z$ is a vector of uncorrelated Normals. The joint p.d.f. is

$$\log p_S = -\frac{1}{2} \log |\Sigma| - \frac{1}{2} (\hat{S} - \mu)^T \Sigma^{-1} (\hat{S} - \mu) - \frac{1}{2} d \log(2\pi).$$

and so

$$\frac{\partial \log p_S}{\partial \mu} = C^{-T} Z,$$

$$\frac{\partial \log p_S}{\partial \Sigma} = \frac{1}{2} C^{-T} \left( ZZ^T - I \right) C^{-1}$$
Vibrato Monte Carlo

For each $W$, in forward mode we have

$$\alpha, \dot{\alpha} \rightarrow \mu, \dot{\mu}, \Sigma, \dot{\Sigma} \rightarrow \text{payoff + sensitivity}$$

- first bit – pathwise sensitivity calculation
- second bit – Likelihood Ratio Method

For maximum efficiency can use adjoint/reverse mode

$$\alpha \rightarrow \mu, \Sigma \rightarrow \text{payoff}$$

$$\bar{\alpha} \leftarrow \bar{\mu}, \bar{\Sigma} \leftarrow \text{sensitivity}$$

$\bar{\mu}, \bar{\Sigma}$ are coefficients multiplying $\dot{\mu}, \dot{\Sigma}$ in forward mode
Conclusions

Monte Carlo estimation of Greeks is an important problem in computational finance.

Improved methods need ideas from both mathematics:
- Adjoint technique
- Vibrato Monte Carlo

... and computer science:
- Automatic differentiation
Future Work

- application of AD to real-world problems
- implementation/demonstration of vibrato MC
- combining these with multilevel MC (or QMC) for greater savings
- more consideration of pros and cons of Malliavin approach
- parallel execution on NVIDIA graphics cards
Acknowledgements

- Paul Glasserman for collaboration on adjoint technique and discussions on vibrato Monte Carlo
- Ole Stauning for help with FADBAD++
- Michael Vossbeck for TAC++ results

Further information

- www.comlab.ox.ac.uk/mike.giles/
- Email: giles@comlab.ox.ac.uk