Adjoint Methods in Computational Finance
Software Tool Support for Algorithmic Differentiation

Uwe Naumann

LuFG Informatik 12
Software and Tools for Computational Engineering
RWTH Aachen University, Germany
naumann@stce.rwth-aachen.de
www.stce.rwth-aachen.de

and

The Numerical Algorithms Group Ltd.
Oxford, United Kingdom
Uwe.Naumann@nag.co.uk
Motivation: Cheap Greeks
Mathematical/Numerical Model

\[ T = T(t, x, c(x)) : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \] is given as the solution of the 1D diffusion equation

\[
\frac{\partial T}{\partial t} = c(x) \cdot \frac{\partial^2 T}{\partial x^2}
\]

over the domain \( \Omega = [0, 1] \) and with initial condition \( T(0, x) = i(x) \) for \( x \in \Omega \) and Dirichlet boundary \( T(t, 0) = b_0(t) \) and \( T(t, 1) = b_1(t) \) for \( t \in [0, 1] \).

The numerical solution is based on a central finite difference / implicit (backward) Euler integration scheme that exploits linearity of the residual \( r \) in \( T \) (single evaluation of constant Jacobian \( \frac{\partial r}{\partial T} \) and factorization).
Motivation: Cheap Greeks

Job

We aim to analyze sensitivities of the predicted $T(x, c(x))$ with respect to $c(x)$ or even calibrate the uncertain $c(x)$ to given observations $O(x)$ for $T(x, c(x))$ at time $t = 1$ by solving the (possibly constrained) least squares problem

$$
\min_{c(x)} f(c(x), x) \quad \text{where} \quad f \equiv \int_{\Omega} (T(1, x, c(x)) - O(x))^2 \, dx.
$$

Assuming a spatial discretization of $\Omega$ with $n$ grid points we require

$$
\frac{\partial f}{\partial c} = \frac{\partial f}{\partial T(1)} \cdot \frac{\partial T(1)}{\partial c} \in \mathbb{R}^n \quad (\rightarrow \text{scalar adjoint integration at } O(1))
$$

and possibly

$$
\frac{\partial^2 f}{\partial c^2} \in \mathbb{R}^{n \times n} \quad (\rightarrow 2\text{nd-order vector adjoint integrations at } O(n))
$$
Motivation: Cheap Greeks

Performance (Live ...)

Application of Algorithmic Differentiation (AD) tool \texttt{dco/c++} in black-box mode (→ “Getting Started” section in user guide):

\[
\frac{\partial f}{\partial c} \quad (n=300, \, m=50)
\]

<table>
<thead>
<tr>
<th></th>
<th>central FD</th>
<th>adjoint AD</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time (s)</td>
<td>15.2</td>
<td>0.7</td>
</tr>
</tbody>
</table>

\[
\frac{\partial^2 f}{\partial c^2} \quad (n=100, \, m=50)
\]

<table>
<thead>
<tr>
<th></th>
<th>central FD</th>
<th>adjoint AD</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time (s)</td>
<td>63.6</td>
<td>3.9</td>
</tr>
</tbody>
</table>

... while ignoring accuracy for the time being ...
Nice to have?

MITgcm, (EAPS, MIT)

in collaboration with ANL, MIT, Rice, UColorado


Plot: A tangent-linear computation / finite difference approximation for 64,800 grid points at 1 min each would keep us waiting for a month and a half ... :-((( We can do it in less than 10 minutes thanks to adjoints computed by a differentiated version of the MITgcm :-)

Flow (... through this presentation)

- AD Models
- AD Code
- Case Study: LIBOR
- AD of Numerical Methods
- Case Study: Local Volatilities
- Conclusion

(©M.Towara with OpenFOAM)
U. Naumann: 
The Art of Differentiating Computer Programs. 
An Introduction to Algorithmic Differentiation. 
Number 24 in Software, Environments, and Tools, 

U. Naumann and O. Schenk, eds.: 
Combinatorial Scientific Computing. 
Chapman & Hall/CRC Computational Science Series 
Taylor and Francis Group, 2012.
Tangents and Adjoints by AD

\[ y = f(x) : \mathbb{R}^n \rightarrow \mathbb{R} \]

- **Tangent-Linear Code** \( f^{(1)} \) (forward AD)

\[
y^{(1)} = \nabla f(x) \cdot x^{(1)} \\
\quad \in \mathbb{R}^n \quad \in \mathbb{R}^n
\]

\[ \Rightarrow \nabla f \text{ at } O(n) \cdot \text{Cost}(f) \]

**Note:** Approximation by bumping.

- **Adjoint Code** \( f_{(1)} \) (reverse AD)

\[
x^{(1)} = y^{(1)} \cdot \nabla f(x) \\
\quad \in \mathbb{R}
\]

\[ \Rightarrow \nabla f \text{ at } O(1) \cdot \text{Cost}(f) \]
“Hello World” Example

\[ y = f(x) = \sin(e^x) \]

```c
void f(double &z) {
    z = exp(z);
    z = sin(z);
}
```

![Graph generated with Gnuplot](generated with Gnuplot)
Tangent Code: 
\[
\begin{pmatrix}
  y^{(1)} \\
y
\end{pmatrix} = 
\begin{pmatrix}
  f'(x) \cdot x^{(1)} \\
f(x)
\end{pmatrix}
\]

```c
void t1_f(double &z, double &t1_z) {
    z = exp(z);
    z = sin(z);
}
```
Tangent Code: 
\[
\begin{pmatrix}
  y^{(1)} \\
  y \\
\end{pmatrix}
= 
\begin{pmatrix}
  f'(x) \\
  f(x) \\
\end{pmatrix}
\]

```c
void t1_f(double &z, double &t1_z) {
    z = exp(z);
    z = sin(z);
    t1_z = exp(z) * t1_z; // WRONG z!
    t1_z = cos(z) * t1_z;
}
```
Tangent Code: \[
\begin{pmatrix}
y^{(1)} \\
y
\end{pmatrix}
= \begin{pmatrix}
f'(x) \cdot x^{(1)} \\
f(x)
\end{pmatrix}
\]

```c
void t1_f(double &z, double &t1_z) {
    t1_z = exp(z) * t1_z;
    z = sin(z);
    z = exp(z);
    t1_z = cos(z) * t1_z; // WRONG z!
}
```
Tangent Code: \[
\begin{pmatrix}
  y^{(1)} \\
  y
\end{pmatrix} = 
\begin{pmatrix}
  f'(x) \cdot x^{(1)} \\
  f(x)
\end{pmatrix}
\]

```c
void t1_f(double &z, double &t1_z) {
    t1_z=exp(z)*t1_z;
    z=exp(z);
    t1_z=cos(z)*t1_z;
    z=sin(z); // CORRECT z!
}
```
Tangent Code: \[
\begin{pmatrix}
  y^{(1)} \\
  y
\end{pmatrix} = \begin{pmatrix}
  f'(x) \cdot x^{(1)} \\
  f(x)
\end{pmatrix}
\]

```c
void t1_f(double &z, double &t1_z) {
    t1_z=exp(z)*t1_z;
    z=exp(z);
    t1_z=cos(z)*t1_z;
    z=sin(z);
}
```

Driver:

```c
... double z=...; t1_z=1;
t1_f(z,t1_z);
cout « t1_z « endl;
...```

![Graph generated with Gnuplot](generated with Gnuplot)
void a1_f(double &z, double &a1_z) {
    z = exp(z);
    z = sin(z);
}

Adjoint Code: \[
\begin{pmatrix}
    y \\
    x(1)
\end{pmatrix}
= 
\begin{pmatrix}
    f(x) \\
    y(1) \cdot f'(x)
\end{pmatrix}
\]
Adjoint Code: \[
\begin{pmatrix}
    y \\
    x(1)
\end{pmatrix}
= \begin{pmatrix}
    f(x) \\
    y(1) \cdot f'(x)
\end{pmatrix}
\]

```c
void a1_f(double &z, double &a1_z) {
    z=exp(z);
    z=sin(z);
    a1_z=cos(z)*a1_z;  // WRONG z!
    a1_z=exp(z)*a1_z;
}
```
Adjoint Code: \[
\begin{pmatrix}
  y \\
  x(1)
\end{pmatrix} = \begin{pmatrix}
  f(x) \\
  y(1) \cdot f'(x)
\end{pmatrix}
\]

```c
void a1_f(double &z, double &a1_z) {
    z=exp(z);
push(z);
z=sin(z);
pop(z);
a1_z=cos(z)*a1_z;
a1_z=exp(z)*a1_z;  // WRONG z!
}
```
Adjoint Code: \[
\begin{pmatrix}
y \\
x(1)
\end{pmatrix}
= \begin{pmatrix}
f(x) \\
y(1) \cdot f'(x)
\end{pmatrix}
\]

```c
void a1_f(double &z, double &a1_z) {
    push(z);
    z=exp(z);
    push(z);
    z=sin(z);
    pop(z);
    a1_z=cos(z)*a1_z;
    pop(z);
    a1_z=exp(z)*a1_z;
} // WRONG FUNCTION VALUE!
```
Adjoint Code: \[
\begin{pmatrix}
    y \\
    x(1)
\end{pmatrix}
= \begin{pmatrix}
    f(x) \\
    y(1) \cdot f'(x)
\end{pmatrix}
\]

```c
void a1_f(double &z, double &a1_z) {
    push(z);
    z=exp(z);
    push(z);
    z=sin(z);
    store(z);
    pop(z);
    a1_z=cos(z)*a1_z;
    pop(z);
    a1_z=exp(z)*a1_z;
    restore(z);
}
```

“You’re entering a world of pain ...”\(^1\) if MEM < 24 Bytes ...

\(^1\)John Goodman ("Walter") in *The Big Lebowski*
Adjoint Code: \[
\begin{pmatrix}
    y \\
    x(1)
\end{pmatrix} = \begin{pmatrix}
    f(x) \\
    y(1) \cdot f'(x)
\end{pmatrix}
\]

```c
void al_f(double &z, double &a1_z) {
    ...
}
```

Driver:

```c
...
double z=..., a1_z=1;
al_f(z,a1_z);
cout << a1_z << endl;
...
```

(generated with Gnuplot)
Higher-Order AD

\[ y = f(x) : \mathbb{R}^n \rightarrow \mathbb{R} \]

- **Second-Order Tangent-Linear Code** (forward-over-forward AD)

  \[ y^{(1,2)} = x^{(1)T} \cdot \nabla^2 f(x) \cdot x^{(2)} \]

  \[ \in \mathbb{R}^n \]

  \[ \in \mathbb{R}^{n \times n} \]

  \[ \in \mathbb{R}^n \]

  \[ \Rightarrow \nabla^2 f \text{ at } O(n^2) \cdot \text{Cost}(f) \]

- **Second-Order Adjoint Code** (forward-over-reverse AD)

  \[ x^{(2)}_{(1)} = y_{(1)} \cdot \nabla^2 f(x) \cdot x^{(2)} \]

  \[ \Rightarrow \nabla^2 f \cdot x^{(2)} \text{ at } O(1) \cdot \text{Cost}(f) \text{ resp. } \nabla^2 f \text{ at } O(n) \cdot \text{Cost}(f) \]

- Higher-order tangent-linear (FoFo...FoF) and adjoint (FoFo...FoR) models are derived recursively
Consider the **LIBOR market model** of Brace, Gatarek & Musiela (1997) as implemented by Glasserman & Zhao (1999) and used by Giles and Glasserman (2006) in their “*Smoking Adjoint: Fast Monte Carlo Greeks*” article in *Risk* for the computation of

\[ \Delta \equiv \frac{\partial g}{\partial X(0)} \]

and

\[ \Gamma \equiv \frac{\partial^2 g}{\partial X(0)^2} \]

with discounted payout \( g \in \mathbb{R} \) and initial value of the underlying \( X(0) \in \mathbb{R}^n \) (forward Libor rates) in the context of the Euler integration scheme.
... 15 swaptions expiring in $N = 80$ periods; dco exploiting structure

$\Delta$ ($10^5$ Monte Carlo path simulations)

<table>
<thead>
<tr>
<th>function</th>
<th>bumping</th>
<th>adjoint AD</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time (s)</td>
<td>1.2</td>
<td>70.6</td>
</tr>
<tr>
<td>relative run time</td>
<td>1</td>
<td>58.8</td>
</tr>
</tbody>
</table>

Relative run time / 80 (Projection of $\Gamma$)

<table>
<thead>
<tr>
<th>function</th>
<th>bumping</th>
<th>adjoint AD</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time (s)</td>
<td>0.15</td>
<td>162</td>
</tr>
<tr>
<td>relative run time / 80 (Projection of $\Gamma$)</td>
<td>1/80</td>
<td>13.45</td>
</tr>
</tbody>
</table>

$\Gamma$ ($10^4$ Monte Carlo path simulations)

$^2$e.g., cheap sum of columns
Suppose that some quant decides to replace certain numerical kernels (linear algebra, interpolation, quadrature, nonlinear programming, ...) in the given simulation code by library routines (e.g. NAG Library).

Furthermore, suppose that this person is the lucky owner of an adjoint version of this simulation generated by some AD tool (e.g. dco).

Switching to the library will break the adjoint code unless the library routines are available as source or are supported by the AD tool (NAG Library support by dco).
Example: Embedded Newton-Raphson

\[ f(z(\lambda), \lambda) = \sin(\lambda \cdot e^z) = 0 \text{ (tangent-linear AD)} \]

\[ \lambda = P(x) \quad // \text{source} \]
\[ z = S(f, f', z^0, \lambda) \quad // \text{library} \]
\[ y = p(z, x) \quad // \text{source} \]

\[ \begin{pmatrix} \lambda \\ \lambda^{(1)} \end{pmatrix} = P^{(1)}(x, x^{(1)}) \quad // \text{AD applicable to source} \]
\[ \begin{pmatrix} z \\ z^{(1)} \end{pmatrix} = ??? \quad // \text{missing library routine :-{(} \]
\[ \begin{pmatrix} y \\ y^{(1)} \end{pmatrix} = p^{(1)}(z, z^{(1)}, x, x^{(1)}) \quad // \text{AD applicable to source} \]
Developer’s Perspective on Library

\[ z_{i+1} = z_i - \frac{f(z_i)}{f'(z_i)} \quad \text{for } i = 0, \ldots \]

\[
\begin{pmatrix}
  (z_i^{(1)})_{i+1} \\
  (z_i_{i+1})
\end{pmatrix} = \begin{pmatrix}
  2 + \frac{f(z_i) \cdot f''(z_i)}{(f'(z_i))^2} \\
  z_i - \frac{f(z_i)}{f'(z_i)}
\end{pmatrix} \cdot z_i^{(1)} \quad \text{for } i = 0, \ldots
\]

Requires user to provide \( f''(z_i) \cdot z_i^{(1)} \).

→ dco
And they all lived happily ever after ...

\[
\begin{pmatrix}
\lambda \\
\lambda^{(1)}
\end{pmatrix} = P^{(1)}(x, x^{(1)}) \quad \text{// dco applicable to source}
\]

\[
\begin{pmatrix}
z \\
z^{(1)}
\end{pmatrix} = S^{(1)}(f, f', f'', z^0, \lambda, \lambda^{(1)}) \quad \text{// AD-intrinsic library routine :-)}
\]

\[
\begin{pmatrix}
y \\
y^{(1)}
\end{pmatrix} = p^{(1)}(z, z^{(1)}, x, x^{(1)}) \quad \text{// dco applicable to source}
\]

AD tool dco:
- delivers \( f' \) and \( f'' \) as well as \( P^{(1)} \) and \( p^{(1)} \)
- knows “intrinsic” \( S^{(1)} \)
Example: NAG Least-Squares Solver

e04gb

\[ \min_{x \in \mathbb{R}^n} f(x, p) = \sum_{i=1}^{m} (F_i(x, p))^2, \quad y = F(x, p) : \mathbb{R}^{n+np} \rightarrow \mathbb{R}^m \]

1 E04GB_A1S
2 ( ... , F, F_A1S, ! F
3 ... , x, x_A1S, ! initial guess/solution
4 ... )

1 F_A1S
2 ( ... , x, x_A1S, ! uses global p, p_A1S
3 ... , y, y_A1S, ! current point
4 ... , y_A1S, ! current residual
5 JAC, ! current Jacobian
6 JAC_A1S, ! projected Hessian
7 ... )
Consider pricing a European call option by solving

\[
\frac{\partial V}{\partial t} + \frac{1}{2} \cdot \sigma^2 \cdot S^2 \cdot \frac{\partial^2 V}{\partial S^2} + (r - \gamma) \cdot S \cdot \frac{\partial V}{\partial S} - r \cdot V = 0, \quad t \in [0, T],
\]

for \( P = V(S', t') \in \mathbb{R} \) at some future time \( t' \) and for some current price \( S' \) of the underlying asset.

The volatility \( \sigma = \sigma(S, t) \) is given as function of strike \( K \), interest \( r = r(t) \), dividend \( \gamma = \gamma(t) \), maturity \( T \), and implied volatility \( \Theta = \Theta(S, t, K, T) \in \mathbb{R}^{m \Theta \times n \Theta} \).

Case Study: Local Volatility Model
Implementation

\[ P^{(1)} = F^{(1)}(r, r^{(1)}, \gamma, \gamma^{(1)}, K, K^{(1)}, \Theta, \Theta^{(1)}) \]

\[ (r^{(1)}, \gamma^{(1)}, K^{(1)}, \Theta^{(1)}) = F^{(1)}(r, \gamma, K, \Theta, P^{(1)}) \]

Interpolate1D\(^{(1)}\)
Interpolate2D\(^{(1)}\)

\( r^{(1)} \in \mathbb{R}^{m_r} \)
\( \gamma^{(1)} \in \mathbb{R}^{m_\gamma} \)
\( K^{(1)} \in \mathbb{R} \)
\( \Theta^{(1)} \in \mathbb{R}^{m_\theta \times n_\theta} \)
# Case Study: Local Volatility Model

## Performance (250 × 500 PDE Mesh)

<table>
<thead>
<tr>
<th>Expression</th>
<th>FD³ (s)</th>
<th>AD (s)</th>
<th>AD Mode</th>
<th>Memory (gb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{\partial P}{\partial r} \in \mathbb{R}^{11} )</td>
<td>4</td>
<td>12</td>
<td>R(F?)</td>
<td>2.75</td>
</tr>
<tr>
<td>( \frac{\partial P}{\partial \gamma} \in \mathbb{R}^{11} )</td>
<td>+</td>
<td>↑⁴</td>
<td>R(F?)</td>
<td>↑</td>
</tr>
<tr>
<td>( \frac{\partial P}{\partial \Theta} \in \mathbb{R}^{10 \times 40} )</td>
<td>+</td>
<td>↑</td>
<td>R(F?)</td>
<td>↑</td>
</tr>
<tr>
<td>Diag( \left( \frac{\partial^2 P}{\partial \Theta^2} \right) \in \mathbb{R}^{10 \times 40} )</td>
<td>100</td>
<td>100</td>
<td>FoF</td>
<td>0</td>
</tr>
<tr>
<td>( \frac{\partial^2 P}{\partial S' \partial \Theta} \in \mathbb{R}^{10 \times 40} )</td>
<td>200</td>
<td>24</td>
<td>FoR</td>
<td>5.5</td>
</tr>
<tr>
<td>( \frac{\partial^2 P}{\partial \Theta^2} \in \mathbb{R}^{(10 \times 40)^2} )</td>
<td>200000</td>
<td>1338</td>
<td>FoR</td>
<td>5.5</td>
</tr>
</tbody>
</table>

³ (poor) approximation
⁴ \( t_1 \Rightarrow t_2 \): no additional computation time / persistent memory requirement
Acknowledgments

- Jacques du Toit (The Numerical Algorithms Group)
- Claudia Yastremiz (Barclays Capital)
- Klaus Leppkes (STCE @ RWTH Aachen)
- Viktor Moseniks (The Numerical Algorithms Group and STCE @ RWTH Aachen)
Summary

- First-order Greeks at constant relative computational cost
- Second-order Greeks at linear relative computational cost
- AD tool \textit{dco}
  - supports
    - first- and higher-order Greeks
    - growing set of NAG library routines
    - “user-defined intrinsics”
    - loop- and subroutine call-level checkpointing
    - adjoint MPI
    - dynamic data flow analysis
  - written in high-end C++
  - relative adjoint cost amounts to 2-15 primal runs

... as illustrated by large number of successful applications
Conclusion: Work Flow

f (small scale) :-(_ (blame user ...))
|(semi-automatic)

_SOURCE TRANSFORMATION_ ___:(___ OVERLOADING
(dcc) (dco/c++)
|                                   |
|                                   |
|                                   |
| Authentic: (semi-automatic)       |

|                                 |
|                                 |
|                                 |

| Testing / Debugging             |
|                                 |
| Testing / Debugging             |
|                                 |
|                                 |
|

___:-(___ continuous adjoint)
Conclusion: Work Flow

... |
    F (large scale) __ :-) |
    :-) |
    :-) |

--------- PROFILING, CHECKPOINTING |
| (call tree reversal |
| bisection, revolve) |
| TESTING / DEBUGGING |

...
... and then there is Exa scale ...
I’m sittin’ in front of the computer screen.
Newton’s second iteration is what I’ve just seen.
It’s not quite the progress that I would expect
from a code such as mine
— no doubt it must be perfect!
Just the facts are not supportive, and I wonder ...

\[
\min_{x \in \mathbb{R}^n} f(x)
\]
\[
\nabla f(x^*) = 0
\]
\[
\nabla^2 f(x^i) \cdot \delta^i = -\nabla f(x^i)
\]
\[
x^{i-1} = x^i + \alpha \cdot \delta^i
\]

---

5 **Lyrics:** The Art of Differentiating Computer Programs. ... Page xvii

**Music:** Think of Fool’s Garden’s “Lemon Tree”
My linear solver is state-of-the-art. It does not get better wherever I start. For differentiation is there anything else? Perturbing the inputs – can’t imagine this fails. I pick a small Epsilon, and I wonder ...

\[ A \cdot x = b \]

\[ A \equiv \nabla^2 f(x^i) \]

\[ x \equiv \delta^i \]

\[ b = -\nabla f(x^i) \]

\[ b_j \approx \frac{f(x^i - \epsilon \cdot e_j) - f(x^i + \epsilon \cdot e_j)}{2 \cdot \epsilon} \]
I wonder how, but I still give it a try. The next change in step size is bound to fly. ’cause all I’d like to see is simply optimality.

Epsilon, in fact, appears to be rather small. A factor of ten should improve it all. ’cause all I’d like to see is nearly optimality.

A DAD ADADA DAD ADADA DADAD.
A few hours later my talk’s getting rude. The sole thing descending seems to be my mood. How can guessing the Hessian only take this much time? N squared function runs appear to be the crime. The facts support this thesis, and I wonder ...

Isolation due to KKT
Isolation – why not simply drop feasibility?
The guy next door’s been sayin’ again and again:
An adjoint Lagrangian might relieve my pain.
Though I don’t quite believe him, I surrender.

I wonder how but I still give it a try:
Gradients and Hessians in the blink of an eye.
Still all I’d like to see is simply optimality.
Epsilon itself has finally disappeared.
Reverse mode AD works, no matter how weird,
and I’m about to see local optimality.

\[ \mathcal{L}_{(1)}(\mathbf{x}, \mathbf{x}^{(2)}, \lambda, l_{(1)}, l_{(1)}^{(2)}) = l_{(1)} \cdot \nabla \mathcal{L} \]
Yes, I wonder, I wonder ... 

I wonder how but I still give it a try: Gradient and Hessians in the blink of an eye. Still all I’d like to see ... I really need to see ... now I can finally see my cherished optimality :-)