Adjoint Parameter Calibration
(in Computational Finance)
The Art of Differentiating Computer Programs\textsuperscript{1}

Uwe Naumann

Software and Tools for Computational Engineering
RWTH Aachen University, Germany
and
The Numerical Algorithms Group Ltd., Oxford, UK

\textsuperscript{1}U. Naumann: \textit{The Art of Differentiating Computer Programs}, SIAM 2011.
To appear.
Solve

\[
\min_{x \in \mathbb{R}^n} G(x, r, y) = \sum_{i=0}^{m-1} (y_i - f(x, r_i))^2 \quad \text{s.t.} \quad l \leq c(x) \leq u
\]

using NAG Library routines (focus on first-order).

- \(\nabla G(x)\) by adjoint AD – why and how?
- (Simplified) Live case study
- Toward an adjoint NAG Library
- Adjoint AD is not plug-and-play!
Let $f(x, r_i)$ compute, for example, the price of an asset at some reference point $r_i$, e.g. time. Consider

$$F(x, r, y) = (y_i - f(x, r_i))_{i=0,...,m}$$

$$G(x, r, y) = \langle F(x, r, y), F(x, r, y) \rangle = \sum_{i=0}^{m-1} (y_i - f(x, r_i))^2$$

for given $(r_i, y_i)_{i=0,...,m-1}$

The NAG Library provides

- least-squares solvers (e.g., e04gbc) asking for $(F : \mathbb{R}^n \to \mathbb{R}^m, \nabla F \in \mathbb{R}^{m \times n})$
- nonlinear programing solvers (e.g., e04dgc) asking for $(G, \nabla G \in \mathbb{R}^n)$
User needs to provide

```c
void objfun (int n, const double x[],
             double *objf, double g[], ...)
```

Let $G(x, r, y)$ be implemented as

```c
void G (int m, int n, const double x[],
        const double r[],
        const double y[],
        double *objf)
```

Gradient $\nabla_x G$ by ...

- hand-coding?
- symbolic differentiation using computer algebra systems?
- finite difference quotients?
Fitting a vector of $n$ parameters $\mathbf{x} \in \mathbb{R}^n$ of a polynomial

$$f(\mathbf{x}, r_i) = \sum_{j=0}^{n-1} x_j \cdot r_i^j$$

of degree $n - 1$ to given data $\mathbf{y} \in \mathbb{R}^m$ at reference points $\mathbf{r} \in \mathbb{R}^m$ yields the objective

$$G(\mathbf{x}, \mathbf{r}, \mathbf{y}) = \sum_{i=0}^{m-1} (y_i - f(\mathbf{x}, r_i))^2$$

We time 5000 iterations of e04dgc with $\nabla G \in \mathbb{R}^n$ approximated by central finite differences.

Will get back to this later ... :-(((((}
Motivation:

- Hand-coding can be tedious and error-prone; Derivative code needs to be kept manually in line with original code.
- Computer algebra systems are of (very) limited help.
- Finite differences deliver inaccurate sensitivity information; convergence of the optimization methods can suffer. Each input needs to be perturbed individually.

The tangent-linear (also forward) mode of AD computes for $F : \mathbb{R}^n \to \mathbb{R}^m$ and $x, x^{(1)} \in \mathbb{R}^n$

$$\mathbb{R}^m \ni y^{(1)} = \nabla F \cdot x^{(1)}$$

and hence the Jacobian at $O(n) \cdot \text{Cost}(F)$ with machine accuracy by letting $x^{(1)}$ range over the Cartesian basis vectors in $\mathbb{R}^n$. 
active type `dco::t1s::type` contains function values $v$ and directional derivatives $v^{(1)}$

operators and intrinsic functions are overloaded for `dco::t1s::type`

type of active variables needs to be changed by the user to `dco::t1s::type`; for example,

```cpp
void G (int m, int n, const double x[],
        const double r[], const double y[],
        double *objf)
```

becomes

```cpp
void G (int m, int n, const dco::t1s::type x[],
        const double r[], const double y[],
        dco::t1s::type *objf)
```
```cpp
void objfun ( int n, const double x[],
             double *objf, double g[], ... )

dco::tls::type *tls_x, tls_objf;
...
for ( int i=0; i<n; i++)
    tls_x[i]=x[i];
...
for ( int i=0; i<n; i++) {
    set(tls_x[i],1.0,1);
    G(m,n,tls_x,r,y,&tls_objf);
    set(tls_x[i],0.0,1);
    get(tls_objf,g[i],1);
}
...}
```
same parameter calibration problem ...

We time 5000 iterations of e04dgc with $\nabla G \in \mathbb{R}^n$ computed by dco in first-order scalar tangent-linear mode.

Will get back to this later ... :-(((
The adjoint (also: reverse) mode of AD computes for $y(1) \in \mathbb{R}^m$

$$\mathbb{R}^n \ni x(1) = \nabla F^T \cdot y(1)$$

and hence the Jacobian at $O(m) \cdot \text{Cost}(F)$ with machine accuracy by letting $y(1)$ range over the Cartesian basis vectors in $\mathbb{R}^m$.

Computational cost is $\mathcal{R} \cdot m \cdot \text{Cost}(F)$ where, typically, $\mathcal{R} = [50, \ldots, 3]$

Adjoint AD yields cheap ($O(1) \cdot \text{Cost}(G)$) gradients (and cheap projected Hessians.)
Conservative estimates for dco:

- \( \text{Cost}(\nabla G \cdot x^{(1)}) = 1.25 \cdot \text{Cost}(G) \) in tangent-linear mode
- \( \text{Cost}(\nabla G) = 10 \cdot \text{Cost}(G) \) in adjoint mode.

The adjoint NLP solver outperforms the tangent-linear NLP solver for \( n > 8 \).

For \( n = k \cdot 8 \), we observe a speedup by a factor of \( k \).
active type dco::a1s::type contains function values $v$ and virtual address $&v$ of a recording of the current variable

operators and intrinsic functions are overloaded for dco::a1s::type to record a tape

type of active variables needs to be changed by the user to dco::a1s::type; for example,

```cpp
void G (int m, int n, const dco::a1s::type x[],
       const double r[], const double y[],
       dco::a1s::type *objf)
```

adjoints are propagated from outputs to inputs by interpretation of the tape
```cpp
void objfun ( int n, const double x[],
              double *objf, double g[] ), ...
dco::als::type *als_x, als_objf; ...
tape *t=dco::als::tape::create();
for ( int i=0; i<n; i++) {
    als_x[i]=x[i]; t->register_variable(als_x[i]);
}
G(m,n,als_x,r,y,&als_objf) ;
get(als_objf,*objf);
set(als_objf,1.0,-1);
t->interpret_adjoint();
for ( int i=0; i<n; i++) get(als_x[i],g[i],-1);
...
dco::als::tape::remove(t) ;
```
same parameter calibration problem ...

5000 iterations of e04dgc with $\nabla G \in \mathbb{R}^n$ computed by central finite differences or by dco’s first-order scalar tangent-linear mode took $> 9$ and $> 7$ minutes, respectively.

Well, let us try adjoint mode ... (6 sec. :-) )))
Algorithmic Differentiation of \( F = \circ_{i=1}^{k} F_i \) where \( F_i : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{m_i} \)

\[
\begin{align*}
\mathbf{y} &= F(\mathbf{x}) = \ldots \\
(F_3 \circ F_2 \circ F_1)(\mathbf{x}) &= F'_3(F'_2(F'_1 \cdot \mathbf{x}^{(1)})) \\
F' &\text{ at } \text{Cost}(F) \\
F' &\text{ at } O(n) \cdot \text{Cost}(F) \\
F' &\text{ at } O(m) \cdot \text{Cost}(F)
\end{align*}
\]
Algorithmic Differentiation (AD) delivers exact (up to machine accuracy) first and higher derivatives of implementations of $F : \mathbb{R}^n \to \mathbb{R}^m$ as computer programs.

or

We differentiate what you implemented – not what you possibly intended to implement.

Assumption: The given implementation of $F$ is $d$ times continuously differentiable at all points of interest.

Fact: AD (also known as Automatic Differentiation) is not fully automatic and never will be except for simple cases.
Is it derivatives you want?

\[ y = f(x) = x^2 + 0.1 \cdot \sin(100 \times x) \]
Do derivatives exist?
Inside of a larger parameter estimation problem, we use the NAG library routine

```c
void nag_heston_price(..., s[m_s], t[m_t],
                        sigmav, corr, eta, var0, p[m] ...)
```

to compute \( m \equiv m_s \cdot m_t \) prices \( p \in \mathbb{R}^m \) of a European option using Heston’s stochastic volatility model for \( m_s \) given strike prices \( s \in \mathbb{R}^{m_s} \) and \( m_t \) times to expiry \( t \in \mathbb{R}^{m_t} \).

The four Heston parameters sigmav, corr, eta, and var0 depend on \( n \) global parameters to be calibrated.

An adjoint version of `nag_heston_price` is provided. dco supports the use of such external adjoint routines.
AD is not Plug-and-Play!

- data flow reversal in adjoint mode (checkpointing)
- performance through exploitation of structure and sparsity
- handling and exploitation of parallelism (ampi)
- coupling with source transformation tools (dcc)
- black-box library routines

⇒ many technical and combinatorial² problems

⇒ AD (programming) tools require educated users

You need algorithmic differentiation if

- finite differences cannot be trusted
- finite differences or exact forward sensitivities are too expensive
- you are un(able/willing) to build and solve the adjoint system manually

For large simulation codes in C++ you need to invest several (wo)man months to reach a sustained $\mathcal{R} < 20$.

Maintenance of the adjoint model is crucial and is supported tremendously by the use of AD tools.

We can help you to get started.