

# Adjoint Parameter Calibration (in Computational Finance)

The Art of Differentiating Computer Programs<sup>1</sup>

Uwe Naumann

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




---

<sup>1</sup>U. Naumann: *The Art of Differentiating Computer Programs*, SIAM 2011.  
To appear.

- ▶ Solve

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

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

- ▶  $\nabla G(\mathbf{x})$  by adjoint AD – why and how?
- ▶ (Simplified) Live case study
- ▶ Toward an adjoint NAG Library
- ▶ Adjoint AD is not plug-and-play!

Let  $f(\mathbf{x}, r_i)$  compute, for example, the price of an asset at some reference point  $r_i$ , e.g. time. Consider

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

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

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

The NAG Library provides

- ▶ least-squares solvers (e.g., e04gbc) asking for  $(F : \mathbb{R}^n \rightarrow \mathbb{R}^m, \nabla F \in \mathbb{R}^{m \times n})$
- ▶ nonlinear programming solvers (e.g., e04dgc) asking for  $(G, \nabla G \in \mathbb{R}^n)$

User needs to provide

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

Let  $G(\mathbf{x}, \mathbf{r}, \mathbf{y})$  be implemented as

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

Gradient  $\nabla_{\mathbf{x}}G$  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

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

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  $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$  and  $\mathbf{x}, \mathbf{x}^{(1)} \in \mathbb{R}^n$

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

and hence the Jacobian at  $O(n) \cdot \text{Cost}(F)$  with **machine accuracy** by letting  $\mathbf{x}^{(1)}$  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,

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

becomes

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

```
void objfun (int n, const double x[],  
            double *objf, double g[], ...)  
    dco::t1s::type *t1s_x, t1s_objf;  
    ...  
    for (int i=0;i<n;i++) t1s_x[i]=x[i];  
    ...  
    for (int i=0;i<n;i++) {  
        set(t1s_x[i],1.0,1) ;  
        G(m,n,t1s_x,r,y,&t1s_objf) ;  
        set(t1s_x[i],0.0,1) ;  
        get(t1s_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  $\mathbf{y}_{(1)} \in \mathbb{R}^m$

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

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

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

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

Conservative estimates for dco:

- ▶  $Cost(\nabla G \cdot \mathbf{x}^{(1)}) = 1.25 \cdot Cost(G)$  in tangent-linear mode
- ▶  $Cost(\nabla G) = 10 \cdot Cost(G)$  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,

```
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**

```
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 ...

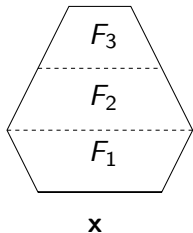
5000 iterations of e04dgc with  $\nabla G \in \mathbb{R}^n$  computed by central finite differences or by dco's first-order scalar tangent-linear mode took  $> 9$  and  $> 7$  minutes, respectively.

Well, let us try adjoint mode ... (6 sec.) :-))))

Tangent-Linear

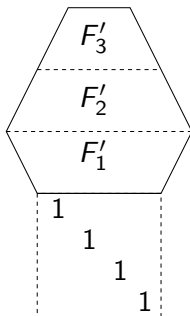
Adjoint

$$\mathbf{y} = F(\mathbf{x}) = \dots$$



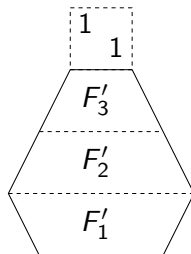
$$(F_3 \circ F_2 \circ F_1)(\mathbf{x})$$

$F$  at  $Cost(F)$



$$F'_3(F'_2(F'_1 \cdot \mathbf{x}^{(1)}))$$

$F'$  at  $O(n) \cdot Cost(F)$



$$(F'_1)^T(((F'_2)^T((F'_3)^T \cdot \mathbf{y}_{(1)})))$$

$F'$  at  $O(m) \cdot Cost(F)$

Algorithmic Differentiation (AD) delivers **exact** (up to machine accuracy) first and higher derivatives **of implementations** of  $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$  **as computer programs**.

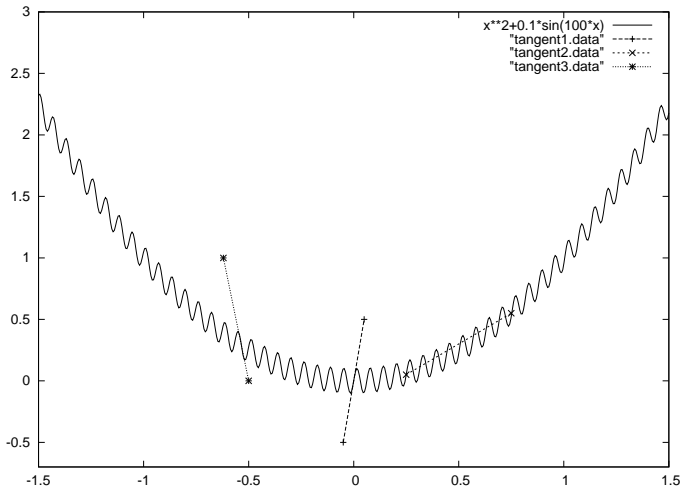
or

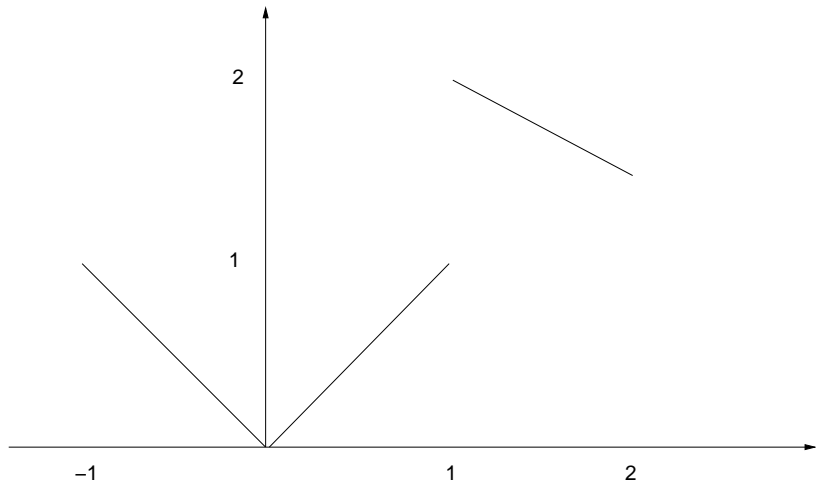
*We differentiate what you implemented – not what you possibly intended to implement.*

**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.

$$y = f(x) = x^2 + 0.1 \cdot \sin(100 \cdot x)$$





Inside of a larger parameter estimation problem, we use the NAG library routine

```
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.

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

⇒ many technical and combinatorial<sup>2</sup> problems

⇒ AD (programming) tools require **educated users**

---

<sup>2</sup>U. Naumann and O. Schenk, eds.: *Combinatorial Scientific Computing*, Chapman & Hall / CRC Press 2011. To appear.

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.