Fast Greeks through Adjoint Algorithmic Differentiation
... and Further Speed-up through Mathematical and Structural Insight

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
MITgcm, (EAPS, MIT)

in collaboration with ANL, MIT, Rice, UColorado


Plot: A 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 :-)
Consider the PDE-constrained optimization problem \( \min_{u(x,0)} J(u, u^{\text{obs}}) \) where

\[
J(u, u^{\text{obs}}) \equiv \int_{\Omega} \left( u(x, T) - u^{\text{obs}}(x) \right)^2 \, dx
\]

subject to viscous Burger’s equation

\[
\frac{\partial u}{\partial t} + u \cdot \frac{\partial u}{\partial x} - \frac{1}{R} \cdot \frac{\partial^2 u}{\partial x^2} = 0
\]

and \( R = 1000 \), initial condition \( u(x, 0) \), and boundary condition \( u(x, t) = 0 \) for \( x \in \Gamma \).

**Discretize PDE**, e.g., using Leap Frog / Du Fort-Frankel scheme\(^1\) (\( \Rightarrow c(u) \)) and objective (\( \Rightarrow o(u) \)), Solution requires the **gradient** and the (tangent projected) Hessian of the Lagrangian

\[
\mathcal{L}(u, \lambda) = o(u) - \lambda^T \cdot c(u)
\]

- Lagrangian → f.c
- tangent-linear Lagrangian by dcc → t1_f.c
- adjoint Lagrangian by dcc → a1_f.c
- drivers: \( \Omega = [0, 1] \), \( T = 1 \), 600 grid points, 2000 time steps
  - t1_main.cpp: 600 calls of t1_f.cpp
  - a1_main.cpp: 1 call of a1_f.cpp
- g++ t1_main.cpp -o t1_main
- time ./t1_main
- Gradient by tangent-linear Lagrangian took about 20 sec.
- Gradient by adjoint Lagrangian takes less than one seconds
  - g++ a1_main.cpp -o a1_main
  - time ./a1_main
- diff t1.out a1.out
- Adjoint runs out of memory for larger problem sizes (e.g. 3000 time steps)... research :-(
(Sub-)Derivatives for ...

- $F(x) = 0, \ F : \mathbb{R}^n \rightarrow \mathbb{R}^n$
  - Newton requires $\nabla F$
  - (matrix-free) Newton-Krylov requires $< \nabla F, x^{(1)} >$

- $f(x) \rightarrow \text{min}, \ f : \mathbb{R}^n \rightarrow \mathbb{R}$
  - quasi-Newton requires $\nabla f$
  - (matrix-free) Newton-Krylov requires $< \nabla^2 f, x^{(2)} >$

- $f(x) \rightarrow \text{min s.t. } c(x) = 0, \ f : \mathbb{R}^n \rightarrow \mathbb{R}, \ c : \mathbb{R}^n \rightarrow \mathbb{R}^m$
  - Newton-Lagrange requires $\nabla f, < \nabla^2 f, x^{(2)} >, < \nabla c, x^{(1)} >$, and
  - $< \lambda, \nabla^2 c, x^{(2)} >$, where $\lambda$ denotes the vector of Lagrange multipliers

- Uncertainty Quantification by Moments Method

- Global Optimization by McCormick Relaxation
Let \( y = g(x, t) \) compute, for example, the price of an asset at some reference time \( t \). Consider

\[
\min_{x \in \mathbb{R}^n} G(x, t, o) = F^T \cdot F \equiv \sum_{i=0}^{m-1} (o_i - g(x, t_i))^2
\]

for given observations \((t_i, o_i)_{i=0}^{m-1}\) and residual \( F = (o_i - g(x, t_i))_{i=0}^{m-1}\) with arbitrary pricer function \( g : \mathbb{R}^{n+1} \rightarrow \mathbb{R} \).

Solution as ...

- **LSQ** (e.g. Gauss-Newton; NAG Library routine e04gbc) require \((F, \nabla_x F \in \mathbb{R}^{m \times n})\)
- **NLP** (e.g. quasi-Newton; NAG Library routine e04dgc) require \((G, \nabla_x G \in \mathbb{R}^n)\)
For differentiation, is there anything else?
Perturbing the inputs – can’t imagine this fails.
I pick a small Epsilon, and I wonder ...

Algorithmic Differentiation (AD)

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

- **Tangent-linear model (forward mode AD)**

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

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

- **Adjoint model (reverse mode AD)**

  \[ \mathbb{R}^n \ni x^{(1)} = f_{(1)}(x, y^{(1)}) \equiv y^{(1)} \cdot \nabla f(x) \in \mathbb{R} \]

  \[ \Rightarrow \nabla f \text{ at } O(1) \cdot \text{Cost}(f) \]
Forward Mode $\Rightarrow$ No Tape/Stack

e.g., $y = \left(\sum_{i=0}^{n-1} x_i^2\right)^2$, $n = 3$

$v_0 = x_0; \quad v_0^{(1)} = x_0^{(1)}$
$v_1 = x_1; \quad v_1^{(1)} = x_1^{(1)}$
$v_2 = x_2; \quad v_2^{(1)} = x_2^{(1)}$
$v_3 = v_0^2; \quad v_3^{(1)} = 2v_0 \cdot v_3^{(1)}$
$v_4 = v_1^2; \quad v_4^{(1)} = 2v_1 \cdot v_4^{(1)}$
$v_5 = v_2^2; \quad v_5^{(1)} = 2v_2 \cdot v_5^{(1)}$
$v_6 = v_3 + v_4; \quad v_6^{(1)} = v_3^{(1)} + v_4^{(1)}$
$v_7 = v_6 + v_5; \quad v_7^{(1)} = v_6^{(1)} + v_5^{(1)}$
$v_8 = v_7^2; \quad v_8^{(1)} = 2v_7 \cdot v_7^{(1)}$
$y = v_8; \quad y^{(1)} = v_8^{(1)}$
Reverse Mode $\Rightarrow$ Tape/Stack

e.g., $y = \left(\sum_{i=0}^{n-1} x_i^2\right)^2$, $n = 3$

\[
\begin{align*}
\nu_0 &= x_0, \quad \nu_1 = x_1, \quad \nu_2 = x_2 \\
\nu_3 &= \nu_0^2, \quad \nu_4 = \nu_1^2, \quad \nu_5 = \nu_2^2 \\
\nu_6 &= \nu_3 + \nu_4, \quad \nu_7 = \nu_6 + \nu_5 \\
\nu_8 &= \nu_7^2; \quad y = \nu_8 \\
\nu_8(1) &= y(1); \quad \nu_7(1) = 2\nu_7 \cdot \nu_8(1) \\
\nu_6(1) &= \nu_5(1) = \nu_7(1); \quad \nu_3(1) = \nu_4(1) = \nu_6(1) \\
\nu_2(1) &= 2\nu_2 \cdot \nu_5(1); \quad \nu_1(1) = 2\nu_1 \cdot \nu_4(1); \quad \nu_0(1) = 2\nu_0 \cdot \nu_3(1) \\
x_2(1) &= \nu_2(1); \quad x_1(1) = \nu_1(1); \quad x_0(1) = \nu_0(1)
\end{align*}
\]
Performance

![Graph showing performance with varying run times](image-url)
Higher-Order AD
\[ y = f(x) : \mathbb{R}^n \to \mathbb{R} \]

- Second-order tangent-linear model\(^2\)
  \[ \mathbb{R} \ni y^{(1,2)} = f^{(1,2)}(x, x^{(1)}, x^{(2)}) \equiv x^{(1)\top} \cdot \nabla^2 f(x) \cdot x^{(2)} \in \mathbb{R}^n \times \mathbb{R}^n \in \mathbb{R}^n \]
  \[ \Rightarrow \nabla^2 f \text{ at } O(n^2) \]

- Second-order adjoint model\(^3\)
  \[ \mathbb{R}^n \ni x^{(2)}_{(1)} = f^{(2)}_{(1)}(x, x^{(2)}, y^{(1)}) \equiv y^{(1)} \cdot \nabla^2 f(x) \cdot x^{(2)} \in \mathbb{R}^n \]
  \[ \Rightarrow \nabla^2 f \cdot x^{(2)} \text{ at } O(1) \text{ resp. } \nabla^2 f \text{ at } O(n) \]

- Higher-order tangent-linear (fofo...fof) and adjoint (fofo...for) models are derived recursively

\(^2\)fof: forward-over-forward
\(^3\)for: forward-over-reverse (≡rof≡ror → symmetry, associativity)
Source transformation (dcc, NAG Fortran compiler)

```c
void a1_f(int n, double *x, double *a1_x,
           double &y, double &a1_y) {
    y = 0; for (int i = 0; i < n; i++) y = y + x[i] * x[i];
    double rd = y; y = y * y;
    double rcp = y; y = rd;
    a1_y = 2 * y * a1_y;
    for (int i = n - 1; i >= 0; i --)
        a1_x[i] += 2 * x[i] * a1_y;
    a1_y = 0; y = rcp;
}
```

Overloading (dco/c++, dco/fortran)

```c
template<class DType>
void f(int n, DType *x, DType &y) {
    y = 0;
    for (int i = 0; i < n; i++) y = y + x[i] * x[i];
    y = y * y;
}
```

Reality → Hybrid Targeted AD
Reference problem: M. Matyka: Hydro Dynamica 3d, University of Wroclaw (3-D Navier-Stokes solver using SIMPLE scheme, single execution of black-box adjoint, 2GB memory); no tricks!
\[
\min_{x \in \mathbb{R}^n} G(x, t, o) = F^T \cdot F \equiv \sum_{i=0}^{m-1} (o_i - g(x, t_i))^2.
\]

Let \( m \geq n \). Solvers require the Jacobian of \( F = (o_i - g(x, t_i))_{i=0}^{m-1} \) that is defined as

\[
\nabla_x F = (\nabla_x F_i)_{i=0}^{m-1} = - (\nabla_x g(x, t_i))_{i=0}^{m-1}.
\]

Since \( m \geq n \), this (dense) Jacobian should be accumulated in tangent-linear mode at the computational cost of \( O(n) \cdot \text{Cost}(F) \), should it not?

Well, not necessarily ...
Note that the $g(x, t_i)$, that yield the individual rows of the Jacobian $\nabla_x F = - (\nabla_x g(x, t_i))^{m-1}_{i=0}$, are nearly independent in the sense that they do not share any intermediate values, except for the input vector $x$.

$m$ runs of the adjoint code with $F^{(1)}$ ranging over the Cartesian basis vectors in $\mathbb{R}^m$ are required for the row-wise accumulation of $\nabla_x F$. 
Complete independence of the \( g(x, t_i) \) can be established by expanding \( x = (x_i) \in \mathbb{R}^n \) to a matrix \( X = (x_{i,j}) \in \mathbb{R}^{n \times m} \) s.t. \( x_{i,j} = x_i \) for \( j = 0, \ldots, m - 1 \) yielding the expanded function

\[
\bar{F}(X, t, o) = (\bar{F}_i)_{i=0}^{m-1} \equiv (o_i - g(x_i,*, t_i))_{i=0}^{m-1}.
\]

A single run of the adjoint code with \( \bar{F}^T(1) = (1, \ldots, 1) \) yields

\[
X(1) = (1, \ldots, 1) \cdot \nabla_x \bar{F} = \nabla_x F
\]

at the computational cost of \( O(1) \cdot \text{Cost}(F) \).
The obvious (embarrassing) parallelism can be exploited by taping single $\bar{F}_i$ or sequences thereof followed by immediate interpretation and deallocation of the associated tape memory.

The above is easily parallelized using MPI and/or OpenMP.

Extension of the above observations to the computation of $\nabla_x G = \langle \frac{\partial G}{\partial F}(F), \nabla_x F(x) \rangle$ is straight forward:

- $\frac{\partial G}{\partial F}(F) = 2 \cdot F$ turns out to be trivial.
- Hence, $\nabla_x G = \langle 2 \cdot F, \nabla \bar{F} \rangle$.

A wide range of trade-offs between computational costs and memory requirement follow.
Consider

$$\lambda = P(z); \ x = S(x^0, \lambda); \ y = p(x)$$

where $S$ denotes a solver for the NLP

$$\min_{x \in \mathbb{R}^n} F(x, \lambda)$$

in the context of a bi-level NLP $\min_{z \in \mathbb{R}^m} f(z)$ requiring $\nabla f(z)$ and, hence, the adjoint $\lambda(1) \equiv < x(1), \nabla S > = \frac{\partial x}{\partial \lambda}^T \cdot \frac{\partial y}{\partial x}$.

Methods for computing $x(1) = < y(1), \frac{\partial y}{\partial x} >$ and $z(1) = < \lambda(1), \frac{\partial \lambda}{\partial z} >$ are assumed to be available.
Differentiation of the first-order optimality condition

\[ \frac{\partial F}{\partial x}(x(\lambda), \lambda) = 0 \]

at the solution \( x(\lambda) \) wrt. \( \lambda \) yields

\[ \frac{\partial^2 F}{\partial x \partial \lambda} + \frac{\partial^2 F}{\partial x^2} \cdot \frac{\partial x}{\partial \lambda} = 0 \]

\[ \Rightarrow \frac{\partial x}{\partial \lambda} = - \frac{\partial^2 F}{\partial x^2} \cdot \frac{\partial F}{\partial x \partial \lambda} \]

\[ \Rightarrow \frac{\partial x^T}{\partial \lambda} = - \frac{\partial^2 F}{\partial x \partial \lambda} \cdot \frac{\partial^2 F^T}{\partial x^2} \]
The computation of the adjoint

\[ \lambda(1) = \langle \mathbf{x}(1), \frac{\partial \mathbf{x}}{\partial \lambda} \rangle = - \left( \frac{\partial^2 F}{\partial \mathbf{x} \partial \lambda} \right)^T \cdot \frac{\partial^2 F}{\partial \mathbf{x}^2} \cdot \mathbf{x}(1) \]

(amounts to the solution of the linear system)

\[ \frac{\partial^2 F}{\partial \mathbf{x}^2} \cdot \mathbf{z} = \mathbf{x}(1) \quad \left(= \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \cdot \mathbf{y}(1) \right) \]

followed by a single call of the second-order adjoint of \( F \) to compute \(- \frac{\partial^2 F}{\partial \mathbf{x} \partial \lambda} \cdot \mathbf{z}. \) The Hessian \( \frac{\partial^2 F}{\partial \mathbf{x}^2}(\mathbf{x}, \lambda) \in \mathbb{R}^{n \times n} \) (-vector products) can be accumulated in second-order adjoint mode AD.
#include <nag.h>
#include <nage04_a1s.h>

void nag_opt_lsq_deriv_a1s ( 
 Integer m, Integer n,
  void (*lsqfun)(Integer m, Integer n,
                double x[], double fvec[], double fjac[],
                Integer tdfjac, Nag_Comm *comm),
  double x[],                     // solution
  double x_a1s[],                 // adjoint solution
  double *fsumsq, double fvec[], double fjac[],
  Integer tdfjac, Nag_E04_Opt *options,
  Nag_Comm *comm,                 // parameters
  Nag_Comm *comm_a1s,             // adjoint parameters
  NagError *fail
)
Conclusion I: The Bad News

Black-box AD will probably fail on your code\(^4\) because

- it assumes differentiability of the function and data-flow continuity of its implementation; It will fail on, e.g.,

\[
y = \begin{cases} 
3 \cdot x & x = 0 \\
2 \cdot x & x \neq 0 
\end{cases}
\]

- it delivers first and higher derivatives with machine accuracy; Is this what you want? (→ \(y = x^2 + 0.1 \cdot \sin(100 \times x)\))

- it delivers (sub-)derivatives of the given implementation; Is this what you want? (→ \(y = |x|\))

- it assumes availability of a sufficient amount of memory to store the variables that are required for the data flow reversal (e.g., the tape) in adjoint mode.

\(^4\)no matter which tool you use!
Conclusion II: The Good News

White-box AD has the potential to produce robust, efficient, and sustainable first- and higher-order tangent-linear and/or adjoint versions of your flow solver if

- you are willing to learn AD; (Well done!)
- you are willing to invest the required development time;
- your AD tool is flexible enough to comply with the requirements of your tailored AD solution;
- your AD tool produces efficient first-order adjoint code (\(\rightarrow\) relative run time);
- your AD tool helps you to detect and exploit special structure and/or sparsity within your problem;
- the code generated by your AD tool is able to handle/exploit parallelism (OpenMP, MPI, accelerators).
To get started ...

U. Naumann:  

naumann@stce.rwth-aachen.de

Uwe.Naumann@nag.co.uk