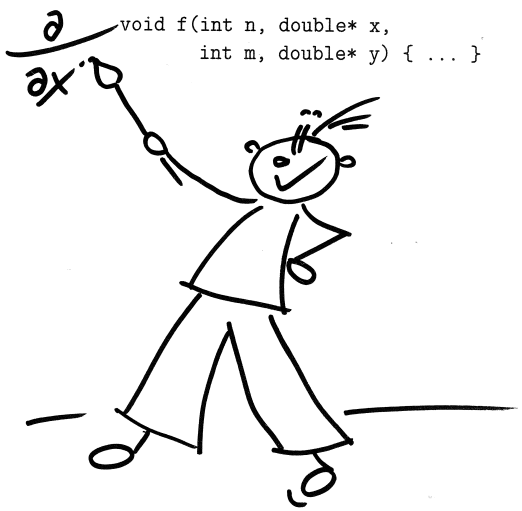
Fast Greeks through Adjoint Algorithmic Differentiation

... and Further Speed-up through Mathematical and Structural Insight

Uwe Naumann

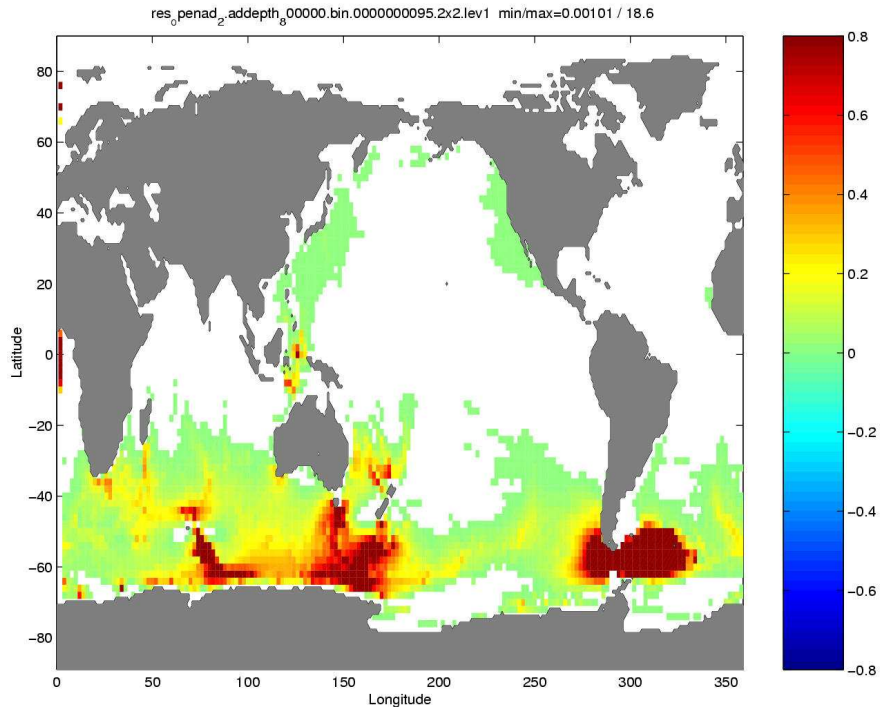


LuFG Informatik 12
Software and **T**ools for **C**omputational **E**ngineering
RWTH Aachen University, Germany
naumann@stce.rwth-aachen.de
www.stce.rwth-aachen.de

and

The **N**umerical **A**lgorithms **G**roup Ltd.
Oxford, United Kingdom
Uwe.Naumann@nag.co.uk

Motivation



MITgcm, (EAPS, MIT)

in collaboration with ANL, MIT, Rice, UColorado

J. Utke, U.N. et al: *OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes* . ACM TOMS 34(4), 2008.

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 :-)

Motivation: Case Study

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

$$\mathcal{L}(u, \lambda) = o(u) - \lambda^T \cdot c(u) \quad .$$

[1] Eugenia Kalnay: Atmospheric Modeling, Data Assimilation and Predictability, Cambridge Uni Press, 2003.

Live

- Lagrangian $\rightarrow$ `f.c`
- tangent-linear Lagrangian by dcc $\rightarrow$ `t1_f.c`
- adjoint Lagrangian by dcc $\rightarrow$ `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`

Live

- 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 ...

- $F(\mathbf{x}) = 0,\ F : \mathbb{R}^n \to \mathbb{R}^n$
  - Newton requires ∇F
  - (matrix-free) Newton-Krylov requires $< \nabla F, \mathbf{x}^{(1)} >$
- $f(\mathbf{x}) \to \min,\ f : \mathbb{R}^n \to \mathbb{R}$
  - quasi-Newton requires ∇f
  - (matrix-free) Newton-Krylov requires $< \nabla^2 f, \mathbf{x}^{(2)} >$
- $f(\mathbf{x}) \to \min$ s.t. $c(\mathbf{x}) = 0,\ f : \mathbb{R}^n \to \mathbb{R},\ c : \mathbb{R}^n \to \mathbb{R}^m$
  - Newton-Lagrange requires ∇f, $< \nabla^2 f, \mathbf{x}^{(2)} >$, $< \nabla c, \mathbf{x}^{(1)} >$, and $< \lambda, \nabla^2 c, \mathbf{x}^{(2)} >$, where λ denotes the vector of Lagrange multipliers
- Uncertainty Quantification by Moments Method
- Global Optimization by McCormick Relaxation

Case Study:

Least Squares Parameter Calibration

Let $y = g(\mathbf{x}, t)$ compute, for example, the price of an asset at some reference time t. Consider

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

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

Solution as ...

- LSQ (e.g. Gauss-Newton; NAG Library routine e04gbc) require $(F, \nabla_{\mathbf{x}} F \in \mathbb{R}^{m\times n})$
- NLP (e.g. quasi-Newton; NAG Library routine e04dgc) require $(G, \nabla_{\mathbf{x}} G \in \mathbb{R}^n)$

...
For differentiation, is there anything else?
Perturbing the inputs – can't imagine this fails.
I pick a small Epsilon, and I wonder ...
...

from: "Optimality" (Lyrics: Naumann; Music: Think of Fool's Garden's "Lemon Tree") in Naumann: The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation. Number 24 in Software , Environments, and Tools, SIAM, 2012. Page xvii

Algorithmic Differentiation (AD)
$y = f(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$

- Tangent-linear model (forward mode AD)

$$\mathbb{R} \ni y^{(1)} = f^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}) \equiv \underset{\in \mathbb{R}^n}{\nabla f(\mathbf{x})} \cdot \underset{\in \mathbb{R}^n}{\mathbf{x}^{(1)}}$$

$$\Rightarrow \quad \nabla f \text{ at } O(n) \cdot Cost(f)$$

- Adjoint model (reverse mode AD)

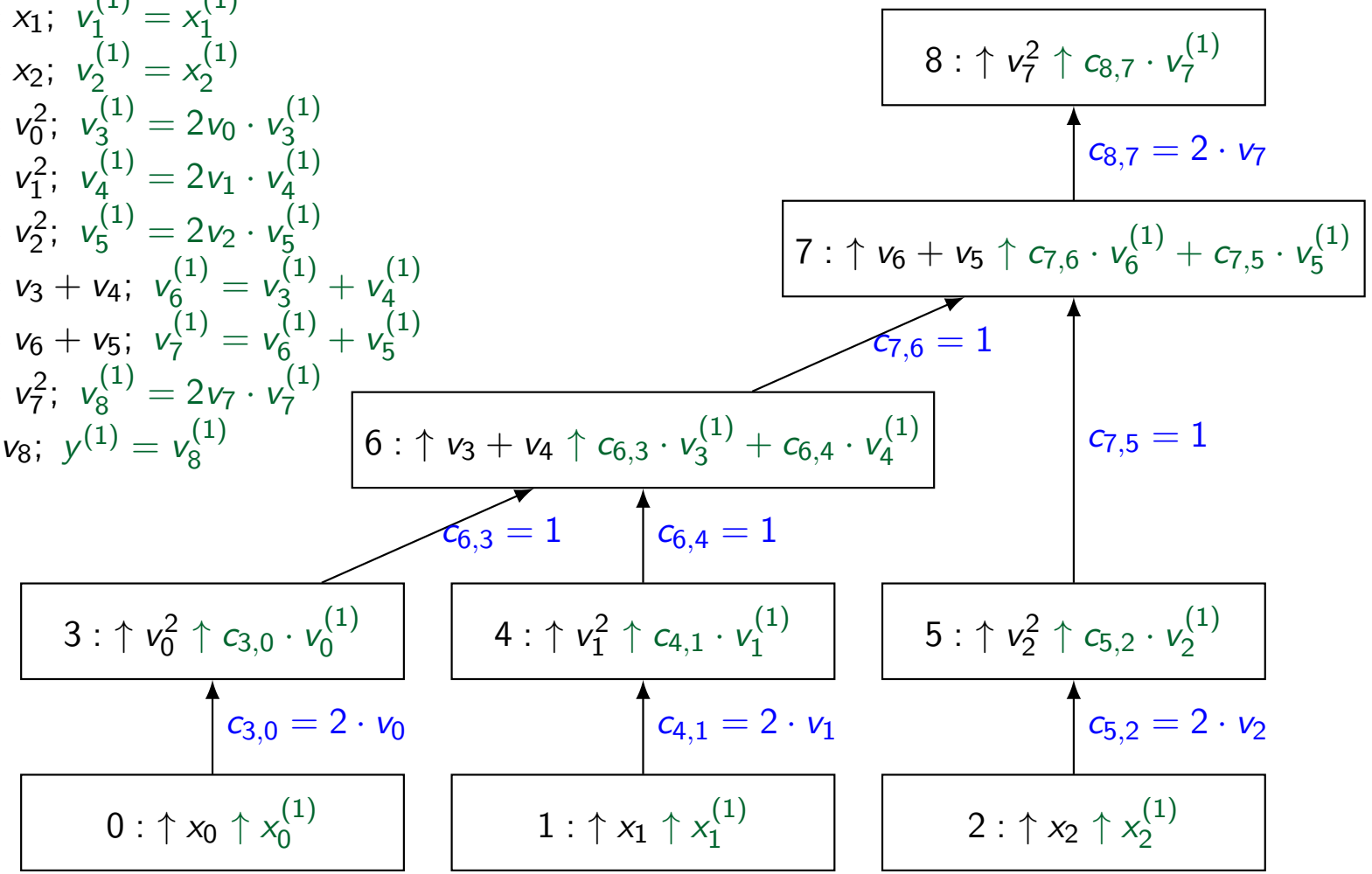
$$\mathbb{R}^n \ni \mathbf{x}_{(1)} = f_{(1)}(\mathbf{x}, y_{(1)}) \equiv \underset{\in \mathbb{R}}{y_{(1)}} \cdot \nabla f(\mathbf{x})$$

$$\Rightarrow \quad \nabla f \text{ at } O(1) \cdot Cost(f)$$

Forward Mode $\Rightarrow$ No Tape/Stack

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

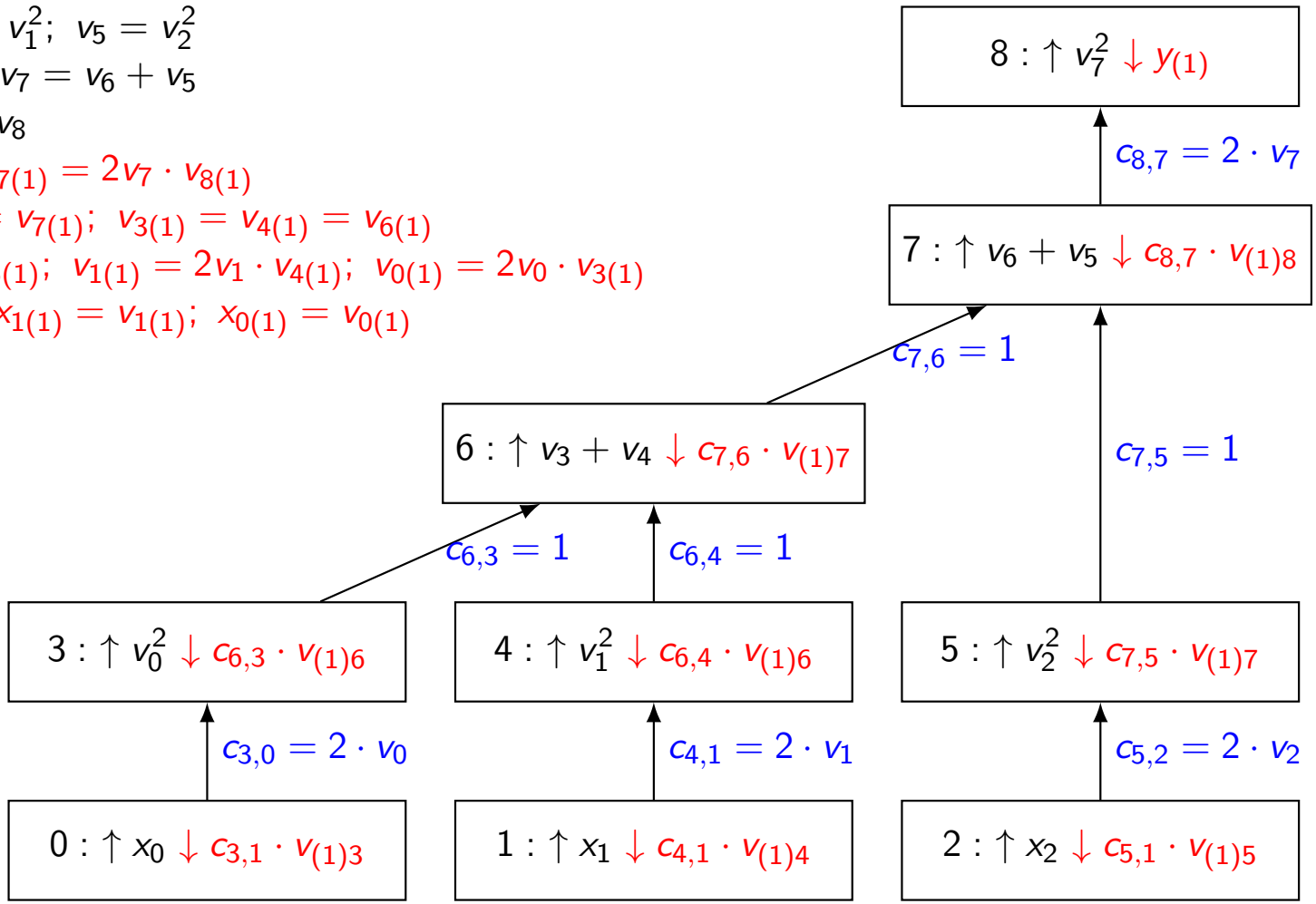
$v_0 = x_0;\ v_0^{(1)} = x_0^{(1)}$
$v_1 = x_1;\ v_1^{(1)} = x_1^{(1)}$
$v_2 = x_2;\ v_2^{(1)} = x_2^{(1)}$
$v_3 = v_0^2;\ v_3^{(1)} = 2v_0 \cdot v_3^{(1)}$
$v_4 = v_1^2;\ v_4^{(1)} = 2v_1 \cdot v_4^{(1)}$
$v_5 = v_2^2;\ v_5^{(1)} = 2v_2 \cdot v_5^{(1)}$
$v_6 = v_3 + v_4;\ v_6^{(1)} = v_3^{(1)} + v_4^{(1)}$
$v_7 = v_6 + v_5;\ v_7^{(1)} = v_6^{(1)} + v_5^{(1)}$
$v_8 = v_7^2;\ v_8^{(1)} = 2v_7 \cdot v_7^{(1)}$
$y = v_8;\ y^{(1)} = v_8^{(1)}$



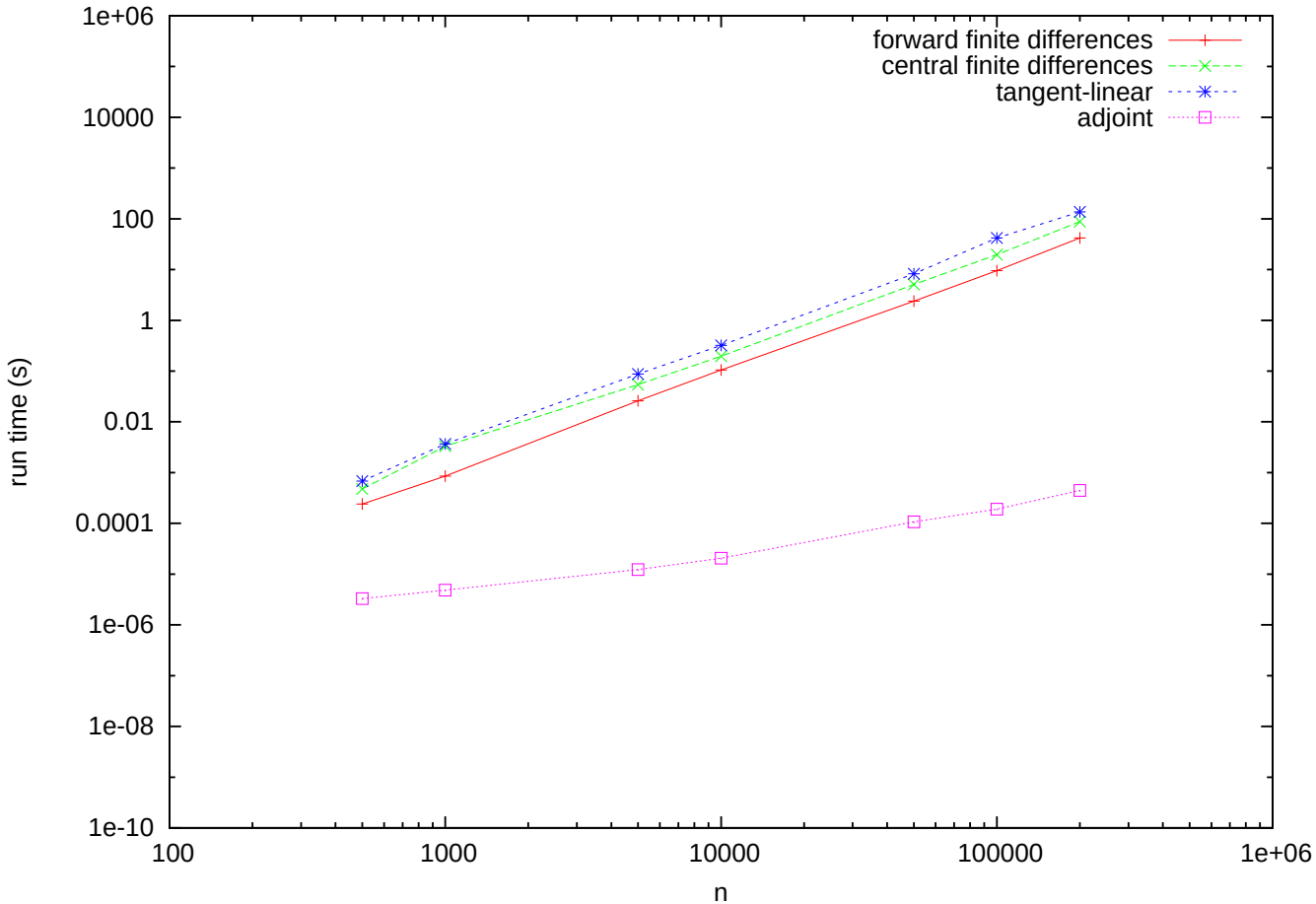
Reverse Mode $\Rightarrow$ Tape/Stack

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

$v_0 = x_0;\ v_1 = x_1;\ v_2 = x_2$
$v_3 = v_0^2;\ v_4 = v_1^2;\ v_5 = v_2^2$
$v_6 = v_3 + v_4;\ v_7 = v_6 + v_5$
$v_8 = v_7^2;\ y = v_8$
$v_{8(1)} = y_{(1)};\ v_{7(1)} = 2v_7 \cdot v_{8(1)}$
$v_{6(1)} = v_{5(1)} = v_{7(1)};\ v_{3(1)} = v_{4(1)} = v_{6(1)}$
$v_{2(1)} = 2v_2 \cdot v_{5(1)};\ v_{1(1)} = 2v_1 \cdot v_{4(1)};\ v_{0(1)} = 2v_0 \cdot v_{3(1)}$
$x_{2(1)} = v_{2(1)};\ x_{1(1)} = v_{1(1)};\ x_{0(1)} = v_{0(1)}$



Performance



Higher-Order AD
$y = f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$

- Second-order tangent-linear model[2]

$$\mathbb{R} \ni y^{(1,2)} = f^{(1,2)}(\mathbf{x}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}) \equiv \underset{\in \mathbb{R}^n}{\mathbf{x}^{(1)}}^T \cdot \underset{\in \mathbb{R}^{n \times n}}{\nabla^2 f(\mathbf{x})} \cdot \underset{\in \mathbb{R}^n}{\mathbf{x}^{(2)}}$$

$$\Rightarrow \quad \nabla^2 f \text{ at } O(n^2)$$

- Second-order adjoint model[3]

$$\mathbb{R}^n \ni \mathbf{x}^{(2)}_{(1)} = f^{(2)}_{(1)}(\mathbf{x}, \mathbf{x}^{(2)}, y_{(1)}) \equiv y_{(1)} \cdot \nabla^2 f(\mathbf{x}) \cdot \mathbf{x}^{(2)}$$

$$\Rightarrow \quad \nabla^2 f \cdot \mathbf{x}^{(2)} \text{ at } O(1) \text{ resp. } \nabla^2 f \text{ at } O(n)$$

- 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))

AD Tool Support @ STCE/NAG

- Source transformation (dcc, NAG Fortran compiler)

```
1 void a1_f(int n, double *x, double *a1_x,
2                  double &y, double &a1_y) {
3   y=0; for (int i=0;i<n;i++) y=y+x[i]*x[i];
4   double rd=y; y=y*y; double rcp=y; y=rd;
5   a1_y=2*y*a1_y;
6   for (int i=n-1;i>=0;i--)
7     a1_x[i]+=2*x[i]*a1_y;
8   a1_y=0; y=rcp;
9 }
```

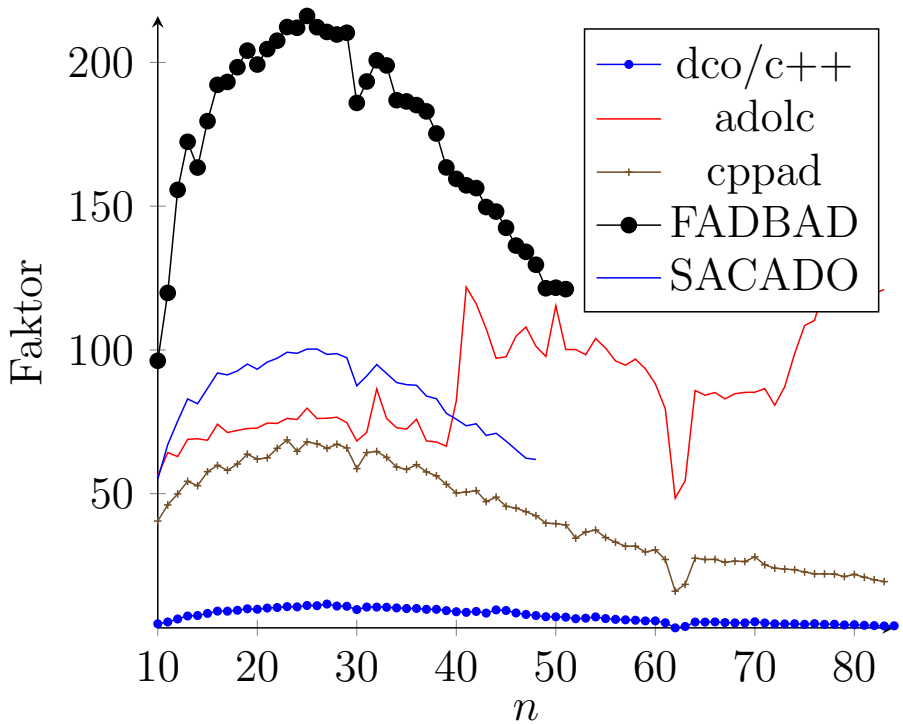
- Overloading (dco/c++, dco/fortran)

```
1 template<class DType>
2 void f(int n, DType *x, DType &y) {
3   y=0;
4   for (int i=0;i<n;i++) y=y+x[i]*x[i];
5   y=y*y;
6 }
```

- Reality → Hybrid Targeted AD

dco/c++ (version 1.*) Competition

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!



Structural Insight
e.g. Concurrency in Least Squares

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

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

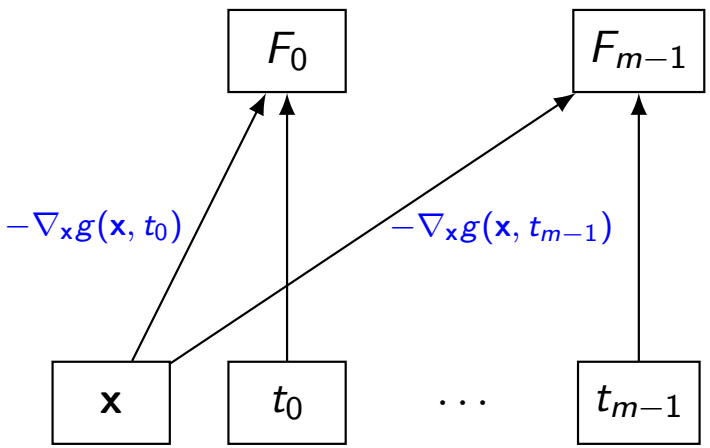
$$\nabla_{\mathbf{x}} F = (\nabla_{\mathbf{x}} F_i)_{i=0}^{m-1} = -(\nabla_{\mathbf{x}} g(\mathbf{x},t_i))_{i=0}^{m-1}.$$

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

Well, not necessarily ...

Structural Analysis

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

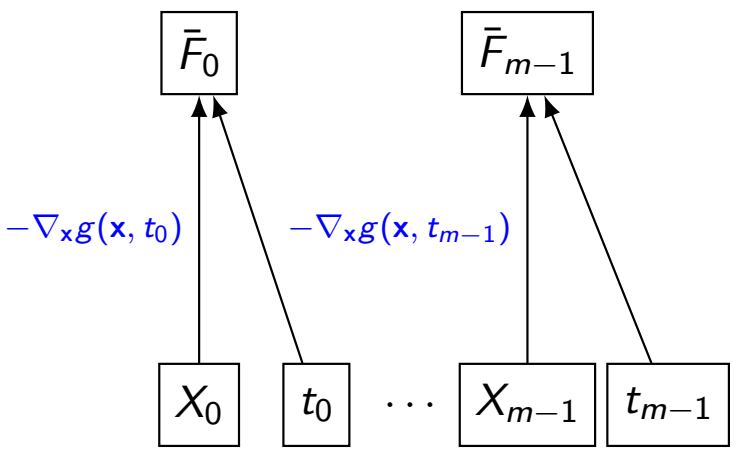


m runs of the adjoint code with $F_{(1)}$ ranging over the Cartesian basis vectors in $\mathbb{R}^m$ are required for the row-wise accumulation of $\nabla_{\mathbf{x}} F$.

Computational Cost

Complete independence of the $g(\mathbf{x}, t_i)$ can be established by expanding $\mathbf{x} = (x_i) \in \mathbb{R}^n$ 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

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



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

$$X_{(1)} = (1, \ldots, 1) \cdot \nabla_{\mathbf{x}} \bar{F} = \nabla_{\mathbf{x}} F$$

at the computational cost of $O(1) \cdot Cost(F)$.

Memory Consumption

- The obvious (*embarassing*) 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 $\nabla_{\mathbf{x}} G =< \frac{\partial G}{\partial F}(F), \nabla_{\mathbf{x}} F(\mathbf{x}) >$ is straight forward:
  - $\frac{\partial G}{\partial F}(F) = 2 \cdot F$ turns out to be trivial.
  - Hence, $\nabla_{\mathbf{x}} G =< 2 \cdot F, \nabla \bar{F} >$.
- A wide range of trade-offs between computational costs and memory requirement follow.

Mathematical Insight e.g. First-Order Optimality Conditions

Consider

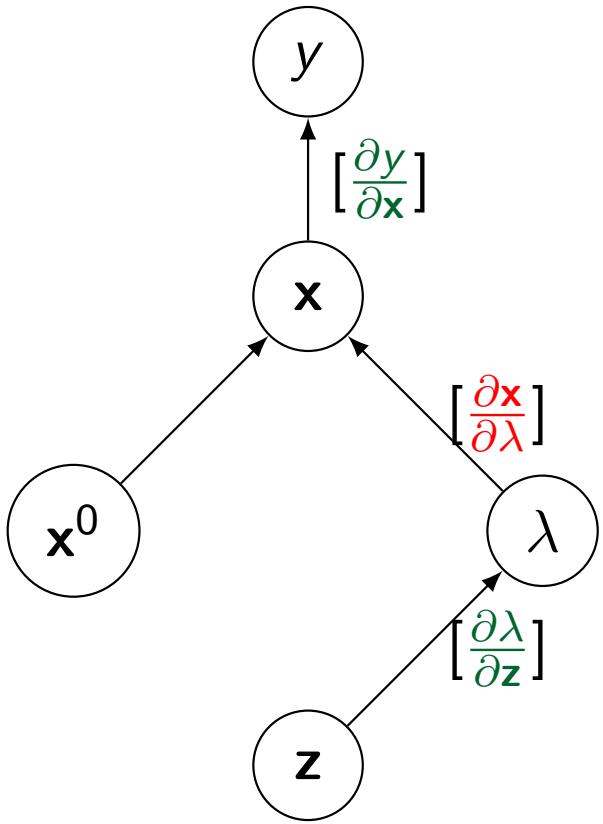
$$\lambda = P(\mathbf{z});\ \mathbf{x} = S(\mathbf{x}^0, \lambda);\ y = p(\mathbf{x})$$

where S denotes a solver for the NLP

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

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

Methods for computing $\mathbf{x}_{(1)} =< y_{(1)}, \frac{\partial y}{\partial \mathbf{x}} >$ and $\mathbf{z}_{(1)} =< \lambda_{(1)}, \frac{\partial \lambda}{\partial \mathbf{z}} >$ are assumed to be available.



Adjoint NLP Solvers ...

Differentiation of the first-order optimality condition

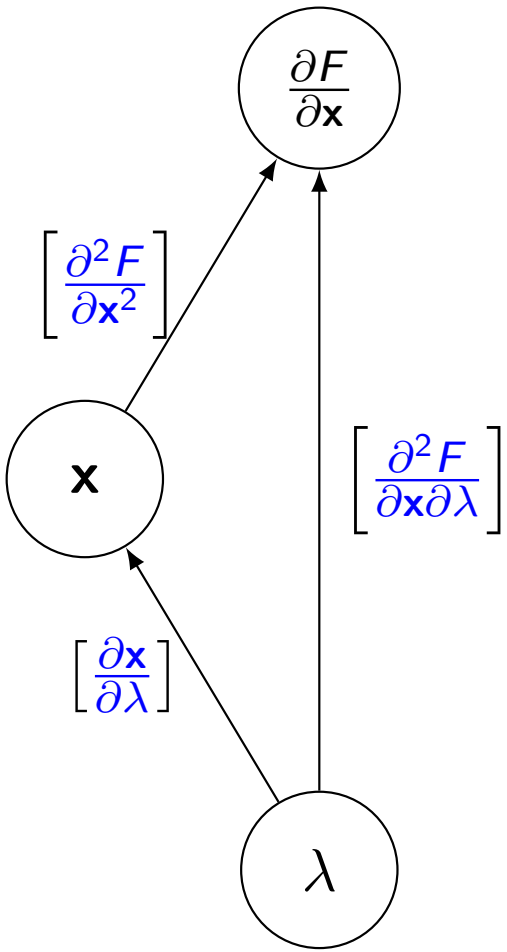
$$\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}(\lambda), \lambda) = 0$$

at the solution $\mathbf{x}(\lambda)$ wrt. λ yields

$$\frac{\partial^2 F}{\partial \mathbf{x} \partial \lambda} + \frac{\partial^2 F}{\partial \mathbf{x}^2} \cdot \frac{\partial \mathbf{x}}{\partial \lambda} = 0$$

$$\Rightarrow \quad \frac{\partial \mathbf{x}}{\partial \lambda} = -\frac{\partial^2 F}{\partial \mathbf{x}^2}^{-1} \cdot \frac{\partial^2 F}{\partial \mathbf{x} \partial \lambda}$$

$$\Rightarrow \quad \frac{\partial \mathbf{x}}{\partial \lambda}^T = -\frac{\partial^2 F}{\partial \mathbf{x} \partial \lambda}^T \cdot \frac{\partial^2 F}{\partial \mathbf{x}^2}^{-T} .$$



... at the Cost of a Single Newton Step

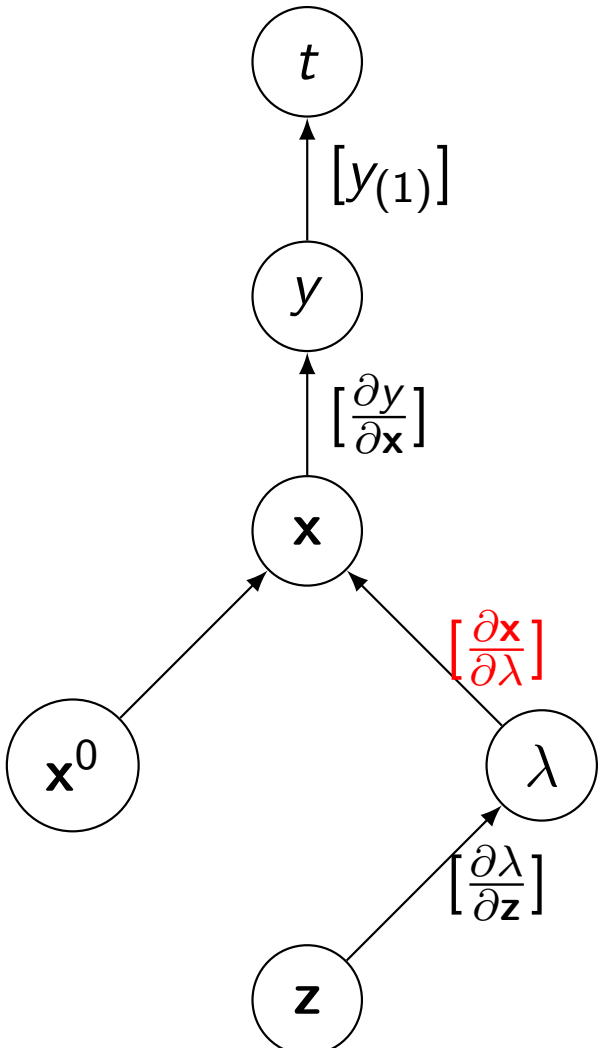
The computation of the adjoint

$$\lambda_{(1)} =< \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial \lambda} >= -\frac{\partial^2 F}{\partial \mathbf{x} \partial \lambda}^T \cdot \underbrace{\frac{\partial^2 F}{\partial \mathbf{x}^2}^{-T} \cdot \mathbf{x}_{(1)}}_{(:=\mathbf{z})}$$

amounts to the solution of the linear system

$$\frac{\partial^2 F}{\partial \mathbf{x}^2} \cdot \mathbf{z} = \mathbf{x}_{(1)} \quad \left(= \frac{\partial y}{\partial \mathbf{x}}^T \cdot y_{(1)} \right)$$

followed by a single call of the second-order adjoint of F to compute $-\frac{\partial^2 F}{\partial \mathbf{x} \partial \lambda}^T \cdot \mathbf{z}$. 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.



Adjoint NAG Library
Example: e04gbc


```
#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? ($\rightarrow y = x^2 + 0.1 \cdot \sin(100 * x)$)
- it delivers (sub-)derivatives of the given implementation; Is this what you want? ($\rightarrow y = |x|$)
- 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.

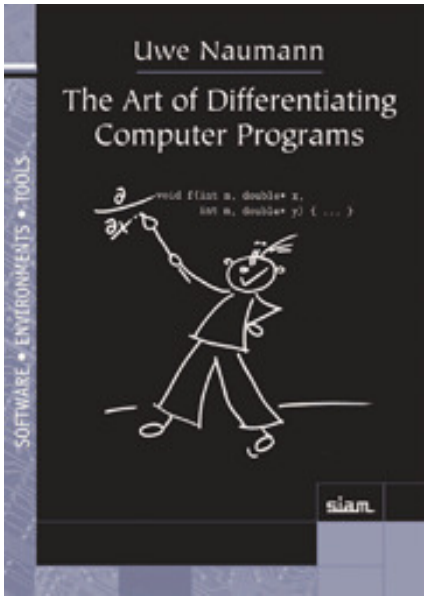
[4] no matter which tool you use!

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:
The Art of Differentiating Computer Programs.
SIAM, 2012.

naumann@stce.rwth-aachen.de

Uwe.Naumann@nag.co.uk