The Art of Differentiating Computer Programs\textsuperscript{1}

Algorithmic Differentiation – Why and How?

Uwe Naumann

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

and

The Numerical Algorithms Group Ltd.
Oxford, UK
Uwe.Naumann@nag.co.uk

\textsuperscript{1}See also upcoming SIAM book
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 :-))
Plan

- Motivation
- Algorithmic Differentiation (AD)
- Race
- First Derivative Codes in Numerical Algorithms
- AD in Action (Live)
- Second and Higher Derivative Codes in Numerical Algorithms
- AD in Action (Live)
- Conclusion and Challenges
First Derivative Codes
Tangent-Linear Code

\[ y^{(1)} = \nabla F(x) \cdot x^{(1)}, \quad x^{(1)} \in \mathbb{R}^n, \; y^{(1)} \in \mathbb{R}^m \]

Approximate Tangent-Linear Code (Finite Differences)

\[ y^{(1)} \approx \frac{F(x + h \cdot x^{(1)}) - F(x)}{h} \]

Adjoint Code

\[ x_{(1)} = \nabla F(x)^T \cdot y_{(1)}, \quad x_{(1)} \in \mathbb{R}^n, \; y_{(1)} \in \mathbb{R}^m \]
Accumulation of Jacobian
\( \nabla F \in \mathbb{R}^{m \times n} \) ...

... with machine accuracy at \( O(n) \cdot Cost(F) \) by

\[ y^{(1)} = \nabla F(x) \cdot x^{(1)} \Rightarrow \text{cheap directional derivatives} \]

... (poor?) approximation at \( O(n) \cdot Cost(F) \) by

\[ \nabla F(x) \cdot x^{(1)} \approx \frac{F(x + h \cdot x^{(1)}) - F(x)}{h} \]

... with machine accuracy at \( O(m) \cdot Cost(F) \) by

\[ x^{(1)} = (\nabla F(x))^T \cdot y^{(1)} \Rightarrow \text{cheap gradients} \]
Consider an implementation $^2$ of 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 the viscous Burger’s equation

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

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

Solution requires the gradient of the Lagrangian

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

with discretized constraints $c(u)$ and objective $o(u)$.

- Lagrangian in f.c $\rightarrow$ t1_f.c (tangent-linear) and a1_f.c (adjoint) by derivative code compiler (dcc)
- drivers: $\Omega = [0, 1], \ T = 1, 600$ grid points, $7000$ 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

will get back to this later ...
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 know 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?
Given: Implementation $y = F(x)$ of the residual $y \in \mathbb{R}^n$ of a system of nonlinear equations and a starting point $x^0 \in \mathbb{R}^n$  

Wanted: $x^* \in \mathbb{R}^n$ such that $F(x^*) = 0$  

Solution: Newton algorithm  

$$x^{k+1} = x^k + \alpha_k \cdot \Delta^k.$$  

The Newton step $\Delta^k \equiv - (\nabla F(x^k))^{-1} \cdot \nabla F(x^k)$ is obtained as the solution of the system of linear equations  

$$\nabla F(x^k) \cdot \Delta^k = -F(x^k)$$  

at each iteration $k = 0, 1, \ldots$  

Matrix-free implementations are possible if Krylov subspace methods (e.g. CG, GMRES depending on the properties of $\nabla F(x^k)$) are used (matrix-free preconditioners?).
Given: Implementation \( y = F(x) \) of the objective \( y \in \mathbb{R} \) of a unconstrained nonlinear programming problem

\[
\min_{x \in \mathbb{R}^n} F(x)
\]

Wanted: A minimizer \( x^* \in \mathbb{R}^n \).

Solution: As the simplest line search method steepest descent computes iterates

\[
x^{k+1} = x^k - \alpha_k \cdot B_k^{-1} \cdot \nabla F(x^k)
\]

from some suitable start value \( x^0 \) and with step length \( \alpha_k > 0 \) for \( B_k = I \in \mathbb{R}^{n \times n} \). Convergence can be defined in various ways. The computational effort is dominated by the evaluation of \( \nabla F(x^k) \). Improved quasi-Newton methods, such as BFGS, are also based on \( \nabla F(x^k) \).
First Derivatives
in the NAG Library

- systems of nonlinear equations (c05ubc); user provides
  ```c
  void j_f(Integer n, const double x[],
            double f[], double j[], ...);
  ```

- unconstrained nonlinear optimization (e04dgc); user provides
  ```c
  void g_f(Integer n, const double x[],
            double *f, double g[], ...);
  ```

- unconstrained nonlinear least squares (e04gbc); user provides
  ```c
  void j_f(Integer m, Integer n, const double x[],
            double f[], double j[], ...);
  ```
Gradient by tangent-linear Lagrangian took several minutes.
Gradient by adjoint Lagrangian takes a few seconds
  g++ a1_main.cpp -o a1_main
  time ./a1_main
diff t1.out a1.out
Adjoint for more complex problems / codes ... nontrivial :-}
AD in Action
Algorithmic Differentiation of $F = \circ_{i=1}^{k} F_i$ where $F_i : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{m_i}$

**Forward Mode**

$y = F(x) = \ldots$

$(F_3 \circ F_2 \circ F_1)(x)$

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

$F'_i$ at $\text{Cost}(F)$

**Reverse Mode**

$F' = \prod \ldots$

$(F'_1) \cdot y^{(1)}$

$F'_3$ at $n \cdot O(\text{Cost}(F))$

$F'_i$ at $m \cdot O(\text{Cost}(F))$
E.g., minimization of \( y = f(x) = \left( \sum_{i=0}^{n-1} x_i^2 \right)^2 \) by Steepest Descent:

\[ |\nabla f| < 10^{-4} \text{ after 9 iterations; } |\nabla f| < 10^{-10} \text{ after 1461997 iterations} \]
... implemented as

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

Steepest Descent / BFGS require gradient to be computed by

- finite differences $\rightarrow O(n) \cdot \text{Cost}(F)$
- tangent-linear code $\rightarrow O(n) \cdot \text{Cost}(F)$
- adjoint code $\rightarrow O(1) \cdot \text{Cost}(F)$
1. $n = 4$
   1.1 computation of gradient by finite differences
   1.2 t1-f from f and computation of gradient
   1.3 a1-f from f and computation of gradient

2. $n = 5 \cdot 10^4$
   2.1 run times
   2.2 (in)accuracy of finite differences
... 
for (int i=0; i<n; i++) x[i] = \cos((\text{double}) i);
f(n, x, y);
for (int i=0; i<n; i++) {
    xph[i] += h;
    f(n, xph, yph);
    xph[i] -= h;
    cout << (yph-y)/h << endl;
}
...
We transform the given implementation

```c
void f(int n, double* x, double& y)
```

of the function \( y = F(x) \) into tangent-linear code computing

\[
y = F(x) \\
y^{(1)} = \nabla F(x) \cdot x^{(1)}.
\]

The signature of the resulting tangent-linear subroutine becomes

```c
void t1_f(int n, double* x, double* t1_x, 
            double& y, double& t1_y)
```
We transform the given implementation

```c
void f(int n, double* x, double& y)
```

of the function \( y = F(x) \) into adjoint code computing

\[
y = F(x) \\
x_{(1)} = \left(\nabla F(x)\right)^T \cdot y_{(1)}.
\]

The signature of the resulting adjoint subroutine becomes

```c
void a1_f(int n, double* x, double* a1_x, 
           double& y, double a1_y)
```
void t1_f(int n, double* x, double* t1_x, 
double& y, double& t1_y) {
    t1_y=0;
y=0;
    for (int i=0;i<n;i++) {
        t1_y=t1_y+2*x[i]*t1_x[i];
y=y+x[i]*x[i];
    }
t1_y=2*y*t1_y;
y=y*y;
}
... 

```cpp
for (int i=0; i<n; i++) {
    t1_x[i]=1;
    t1_f(n, x, t1_x, y, t1_y);
    t1_x[i]=0;
    cout << t1_y << endl;
}
...
```
```cpp
stack<double> required_double, result_double;

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];
    required_double.push(y);
    y = y * y;
    result_double.push(y);

    y = required_double.top();
    required_double.pop();
    a1_y = 2 * y * a1_y;
    for (int i = n - 1; i >= 0; i--) a1_x[i] = 2 * x[i] * a1_y;
    y = result_double.top();
    result_double.pop();
}
```
Driver for 1st-Order Adjoint Code

... 

a1_y=1;
a1_f(n,x,a1_x,y,a1_y);
for (int i=0;i<n;i++) cout << a1_x[i] << endl;
...

Observations

- $n = 4; \ g++ \ -O3; \ h = 10^{-8}$
  - runtime negligible
  - `gvimdiff t1_4.out a1_4.out :-)``
  - `gvimdiff fd_4.out t1_4.out :-)``

- $n = 5 \cdot 10^4$
  - `fd: 4.5s; t1: 6.0s; a1: 0.15s``
  - `gvimdiff t1_50000.out a1_50000.out :-)``
  - `gvimdiff fd_50000.out t1_50000.out :-(`
## Quality of Finite Differences

\[ n = 5 \cdot 10^4, \ h = 10^{-8} \]

| \( g[0] \) | 99992.8 |
| \( g[1] \) | 54025.7 |
| \( g[2] \) | -41616 |
| \( g[3] \) | -99003.3 |
| \( g[4] \) | -65374.4 |
| \( g[5] \) | 28359.9 |
| \( g[6] \) | 96011.2 |
| \( g[7] \) | 75388 |
| \( g[8] \) | -14543.5 |
| \( g[9] \) | -91111.7 |
| \( \ldots \) | \( \ldots \) |

| \( g[0] \) | 100002 |
| \( g[1] \) | 54031.3 |
| \( g[2] \) | -41615.5 |
| \( g[3] \) | -99001.3 |
| \( g[4] \) | -65365.7 |
| \( g[5] \) | 28366.8 |
| \( g[6] \) | 96019 |
| \( g[7] \) | 75391.8 |
| \( g[8] \) | -14550.3 |
| \( g[9] \) | -91114.9 |
| \( \ldots \) | \( \ldots \) |
Higher Derivative Codes
Second-Order Tangent-Linear Code

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

Approximate Second-Order Tangent-Linear Code (Finite Differences)

\[ y^{(1,2)} \approx \frac{f(x + h \cdot (x^{(2)} + x^{(1)})) - f(x + h \cdot x^{(2)}) - f(x + h \cdot x^{(1)}) + f(x)}{h^2} \]

Second-Order Adjoint Code

\[ x^{(2)}_1 = y_1 \cdot \nabla^2 f(x) \cdot x^{(2)} \]
Accumulation of Hessian
\( \nabla^2 f \in \mathbb{R}^{n \times n} \)

... with machine accuracy at \( O(n^2) \cdot \text{Cost}(f) \) by

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

... (even worse?) approximation at \( O(n^2) \cdot \text{Cost}(f) \) by

\[
y^{(1,2)} \approx \frac{f(x + h \cdot (x^{(2)} + x^{(1)})) - f(x + h \cdot x^{(2)}) - f(x + h \cdot x^{(1)}) + f(x)}{h^2}
\]

... with machine accuracy at \( O(n) \cdot \text{Cost}(f) \) by

\[
x^{(2)}_{(1)} = y_{(1)} \cdot \nabla^2 f(x) \cdot x^{(2)}
\]
Second Derivatives
in Newton’s Algorithm

Given: Implementation $y = F(x)$ of the objective $y \in \mathbb{R}$ of an unconstrained nonlinear programming problem

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

Wanted: A minimizer $x^* \in \mathbb{R}^n$.

Solution: Newton algorithm is applied to find a stationary point of the gradient $\nabla F(x)$ yielding the computation of iterates

$$x^{k+1} = x^k - \alpha_k \cdot B_k^{-1} \cdot \nabla F(x^k)$$

from some suitable start value $x^0$ and with step length $\alpha_k > 0$ for $B_k = \nabla^2 F(x^k) \in \mathbb{R}^{n \times n}$. The iterative approximation of the Newton step using Krylov-subspace methods yields matrix-free implementations based on a second-order adjoint model.
Given: Equality-constrained nonlinear programming problem

$$\min \ F(x) \ \text{subject to} \ c(x) = 0$$

where both the objective $F : \mathbb{R}^n \to \mathbb{R}$ and the constraints $c : \mathbb{R}^n \to \mathbb{R}^m$ are assumed to be twice continuously differentiable.

Wanted: A feasible minimizer $x^* \in \mathbb{R}^n$.

Solution: Many algorithms are based on the solution of the KKT system

$$\begin{bmatrix} \nabla F(x) - (\nabla c(x))^T \cdot \lambda \\ c(x) \end{bmatrix} = 0$$

using Newton algorithm.
The iteration proceeds as

$$(x_{k+1}, \lambda_{k+1}) = (x_k, \lambda_k) + \alpha_k \cdot (\Delta^x_k, \Delta^\lambda_k)$$

where the $k$-th Newton step is computed as the solution of the linear system

$$
\begin{bmatrix}
\nabla_{xx} \mathcal{L}(x_k, \lambda_k) & -(\nabla c(x_k))^T \\
\n\nabla c(x_k) & 0
\end{bmatrix}
\begin{bmatrix}
\Delta^x_k \\
\Delta^\lambda_k
\end{bmatrix}
=
\begin{bmatrix}
(\nabla c(x_k))^T \cdot \lambda_k - \nabla F(x_k) \\
-c(x_k)
\end{bmatrix}
$$

Matrix-free implementations of Krylov-subspace methods compute the residual of the constraints ($c(x_k)$), tangent projections of the Hessian of the Lagrangian ($\langle \nabla_{xx} \mathcal{L}(x_k, \lambda_k), v \rangle$), the gradient of the objective ($\nabla F(x_k)$), and tangent and adjoint projections of the Jacobian of the constraints ($\langle \nabla c(x_k), v \rangle$ and $\langle w, \nabla c(x_k) \rangle$).
unconstrained or bound-constrained minima of twice continuously differentiable nonlinear functions (e04lbc); user provides

```c
void g_f(Integer n, const double x[],
          double *y, double g[], ...);
```
and

```c
void h_(Integer n, const double x[],
          double h[], ...);
```
Derivative models of \( k \)-th order are defined as tangent-linear or adjoint models of derivative models of \((k - 1)\)-th order.

**Examples:**

- Third-order tangent-linear model

\[
F^{(1,2,3)}(x, x^{(1)}, x^{(2)}, x^{(3)}) = \langle \nabla^3 F(x), x^{(1)}, x^{(2)}, x^{(3)} \rangle, \quad x^{(i)} \in \mathbb{R}^n
\]

- Fourth-order adjoint model

\[
F^{(2,3,4)}_{(1)}(x, y_{(1)}, x^{(2)}, x^{(3)}, x^{(4)}) = \langle y_{(1)}, \nabla^4 F(x), x^{(2)}, x^{(3)}, x^{(4)} \rangle
\]

\[
\quad x^{(i)} \in \mathbb{R}^n, \quad y_{(1)} \in \mathbb{R}^m
\]
Given: \( y = F(x) \) with \( F : \mathbb{R} \to \mathbb{R} \) (for notational simplicity) and expected value \( \mu_x \) and variance \( \sigma_x \) of \( x \).

Wanted: Estimates for expected value \( \mu_y \) and variance \( \sigma_y \) of \( y \).

Solution: Method of Moments gives

\[
\mu_y = F(\mu_x) + \frac{F''(\mu_x)}{2} \cdot \sigma_x^2
\]

(approximate mean)

\[
\sigma_y^2 = F'(\mu_x)^2 \sigma_x^2 + F'(\mu_x) F''(\mu_x) S_x \sigma_x^3 + \frac{1}{4} (F''(\mu_x))^2 (K_x - 1) \sigma_x^4
\]

(approximate variance)

for given initial mean \( \mu_x \), variance \( \sigma_x^2 \), skewness \( S_x \), and kurtosis \( K_x \) of \( x \in \mathbb{R} \). Approximation of higher-order moments is based on higher derivatives. E.g., robust optimization.
Given: Boundary-controlled PDE-constrained nonlinear programming problem \( \min_{x(s,t), \ s \in \Gamma} F(x) \) subject to \( c(x) = 0 \) with objective

\[
F(x) = \int_{\Omega} \left( x(s, T) - x^{\text{obs}}(s) \right)^2 \, ds,
\]

(measured) initial condition \( x(s, 0) \) and boundary condition \( x(s, t) \) for \( s \in \Gamma \).

Wanted: Quantification of uncertainties in solution (e.g.) wrt. uncertainties in initial condition.

Solution: Second-order moments of the Newton-Lagrange algorithm require derivatives of up to fourth order.
E.g., minimization of \( y = f(x) = \left( \sum_{i=0}^{n-1} x_i^2 \right)^2 \) by Newton’s method

\[ |\nabla f| < 10^{-4} \text{ after 7 iterations; } |\nabla f| < 10^{-10} \text{ after 22 iterations} \]
... implemented as

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

Newton’s method requires gradient and Hessian to be computed by

- 2nd-order finite differences $\rightarrow O(n^2) \cdot Cost(F)$
- 2nd-order tangent-linear code $\rightarrow O(n^2) \cdot Cost(F)$
- 2nd-order adjoint code $\rightarrow O(n) \cdot Cost(F)$
1. $n = 4$
   1.1 computation of Hessian by 2nd-order finite differences
   1.2 $t_2 - t_1 \_ f$ from $t_1 \_ f$ and computation of Hessian
   1.3 $t_2 \_ a_1 \_ f$ from $a_1 \_ f$ and computation of Hessian

2. $n = 2000$
   2.1 run times
   2.2 (in)accuracy of 2n-order finite differences
const double h=1e−6;
f(n,x,y);
for (int j=0;j<n;j++) {
    for (int i=0;i<=j;i++) {
        xph1[j]+=h; f(n,xph1,yph1); xph1[j]−=h;
        xph2[i]+=h; f(n,xph2,yph2); xph2[i]−=h;
        xph3[j]+=h; xph3[i]+=h; f(n,xph3,yph3);
        xph3[j]−=h; xph3[i]−=h;
        cout "" h[" " j " " ]"" i " " ]==""
            << (yph3−yph2−yph1+y)/(h*h) << endl;
    }
    cout "" g[" " j " " ]=="" (yph1−y)/h << endl;
}
We transform \( t1_f \) into second-order tangent-linear code computing

\[
\begin{align*}
y &= F(x) \\
y^{(2)} &= \langle \nabla F(x), x^{(2)} \rangle \\
y^{(1)} &= \langle \nabla F(x), x^{(1)} \rangle \\
y^{(1,2)} &= \langle \nabla F(x), x^{(1,2)} \rangle + \langle \nabla^2 F(x), x^{(1)}, x^{(2)} \rangle.
\end{align*}
\]

The signature of the second-order tangent-linear subroutine becomes

```c
void t2_t1_f(int n, double *x, double *t2_x, double *t1_x, double *t2_t1_x, double &y, double &t2_y, double &t1_y, double &t2_t1_y);
```
void t2_t1_f(int n, double* x, double* t2_x,
        double* t1_x, double* t2_t1_x,
        double& y, double& t2_y,
        double& t1_y, double& t2_t1_y) {
    t2_t1_y = 0; t1_y = 0; t2_y = 0; y = 0;
    for (int i = 0; i < n; i++) {
        t2_t1_y += 2*(t2_x[i] * t1_x[i] + x[i] * t2_t1_x[i]);
        t1_y += 2*x[i] * t1_x[i];
        t2_y += 2*x[i] * t2_x[i];
        y += x[i] * x[i];
    }
    t2_t1_y = 2*(t2_y * t1_y + y * t2_t1_y);
    t1_y = 2*y * t1_y;
    t2_y = 2*y * t2_y;
    y = y * y;
}
Driver for 2nd-Order Tangent-Linear Code

... for (int j=0; j<n; j++) {
    t2_x[j]=1;
    for (int i=0; i<=j; i++) {
        t1_x[i]=1;
        t2_t1_f(n, x, t2_x, t1_x, t2_t1_x, y, t2_y, t1_y, t2_t1_y);
        t1_x[i]=0;
        cout << "h[" << j << "][" << i << "]=" << t2_t1_y << endl;
    }
    t2_x[j]=0;
    cout << "g[" << j << "]=" << t2_y << endl;
}
...
We transform \( a1_f \) into second-order adjoint code computing

\[
y = F(x) \\
y^{(2)} = < \nabla F(x), x^{(2)} > \\
x_{(1)} = x_{(1)} + < y_{(1)}, \nabla F(x) > \\
x_{(1)}^{(2)} = x_{(1)}^{(2)} + < y_{(1)}, \nabla F(x) > + < y_{(1)}, \nabla^2 F(x), x^{(2)} > .
\]

The signature of the second-order adjoint subroutine becomes

```c
void t2_a1_f(int n, double* x, double* t2_x, double* a1_x, double* t2_a1_x, double& y, double& t2_y, double a1_y, double t2_a1_y);
```
```c
void t2_a1_f(int n, double* x, double* t2_x,
             double* a1_x, double* t2_a1_x,
             double& y, double& t2_y,
             double a1_y, double t2_a1_y) {
    t2_y=0;
y=0;
    for (int i=0;i<n;i++) {
        t2_y+=2*x[i]*t2_x[i];
y+=x[i]*x[i];
    }
t2_required_double.push(t2_y);
required_double.push(y);
t2_y=2*y*t2_y;
y=y*y;
}
```
t2_y = t2_required_double.top();
t2_required_double.pop();
y = required_double.top();
required_double.pop();
t2_a1_y = 2*(t2_y*a1_y + y*t2_a1_y);
a1_y = 2*y*a1_y;
for (int i = n-1; i >= 0; i --) {
    t2_a1_x[i] += 2*(t2_x[i]*a1_y + x[i]*t2_a1_y);
a1_x[i] += 2*x[i]*a1_y;
}
}
Driver for 2nd-Order Adjoint Code

... for (int j=0; j<n; j++) {
    for (int i=0; i<n; i++) {
        x[i] = cos((double) i);
        t2_a1_x[i] = t2_x[i] = a1_x[i] = 0;
    }
    t2_a1_y = 0; a1_y = 1; t2_x[j] = 1;
    t2_a1_f(n, x, t2_x, a1_x, t2_a1_x,
            y, t2_y, a1_y, t2_a1_y);
    for (int i=0; i<=j; i++)
        cout << "h[" << j << "][" << i << "]=" << t2_a1_x[i] << endl;
}
for (int i=0; i<n; i++)
    cout << "g[" << i << "]=" << a1_x[i] << endl;
...
Observations

- $n = 4$; g++ -O3; $h = 10^{-6}$
  - runtime negligible
  - gvimdiff t1_4.out a1_4.out :-)
  - gvimdiff fd_4.out t1_4.out :-(

- $n = 10^3$
  - sof: 4.1s; t2_t1: 3.5s; t2_a1: 1.4s
  - gvimdiff t2_t1_1000.out t2_a1_1000.out :-)
  - gvimdiff sof_1000.out t2_t1_1000.out :-((((

- $n = 2 \cdot 10^3$
  - sof: 26.9s; t2_t1: 22.1s; t2_a1: 5.7s

- $n = 3 \cdot 10^3$
  - sof: 85.2s; t2_t1: 69.7s; t2_a1: 12.9s
Quality of 2nd-order FD
\( n = 2000, \ h = 10^{-6} \)

<table>
<thead>
<tr>
<th>sofd</th>
<th>t2t1/t2a1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h[0][0] = 3958.12 )</td>
<td>( h[0][0] = 4009.29 )</td>
</tr>
<tr>
<td>( h[1][0] = 0 )</td>
<td>( h[1][0] = 4.32242 )</td>
</tr>
<tr>
<td>( h[1][1] = 3958.12 )</td>
<td>( h[1][1] = 4003.63 )</td>
</tr>
<tr>
<td>( h[2][0] = 116.415 )</td>
<td>( h[2][0] = -3.32917 )</td>
</tr>
<tr>
<td>( h[2][1] = 0 )</td>
<td>( h[2][1] = -1.79876 )</td>
</tr>
<tr>
<td>( h[2][2] = 4190.95 )</td>
<td>( h[2][2] = 4002.68 )</td>
</tr>
<tr>
<td>( h[3][0] = 0 )</td>
<td>( h[3][0] = -7.91994 )</td>
</tr>
<tr>
<td>( h[3][1] = 116.415 )</td>
<td>( h[3][1] = -4.27916 )</td>
</tr>
<tr>
<td>( h[3][2] = 232.831 )</td>
<td>( h[3][2] = 3.29586 )</td>
</tr>
<tr>
<td>( h[3][3] = 4074.54 )</td>
<td>( h[3][3] = 4009.13 )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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 (legacy) simulation codes you may have to invest

3, 6, 18, 36

(wo)man months for sustained runtime of adjoint runtime of original simulation of

50, 20, < 10, < 4
Challenges and Conclusion

- data flow reversal (checkpointing)
- activation (templated code)
- AD-specific program analysis
- code complexity
- mixed-language codes

- Develop with adjoints in mind!
- Know your AD developer!
- Know your (AD tool/) compiler!